

# Behavioral Program Queries

using

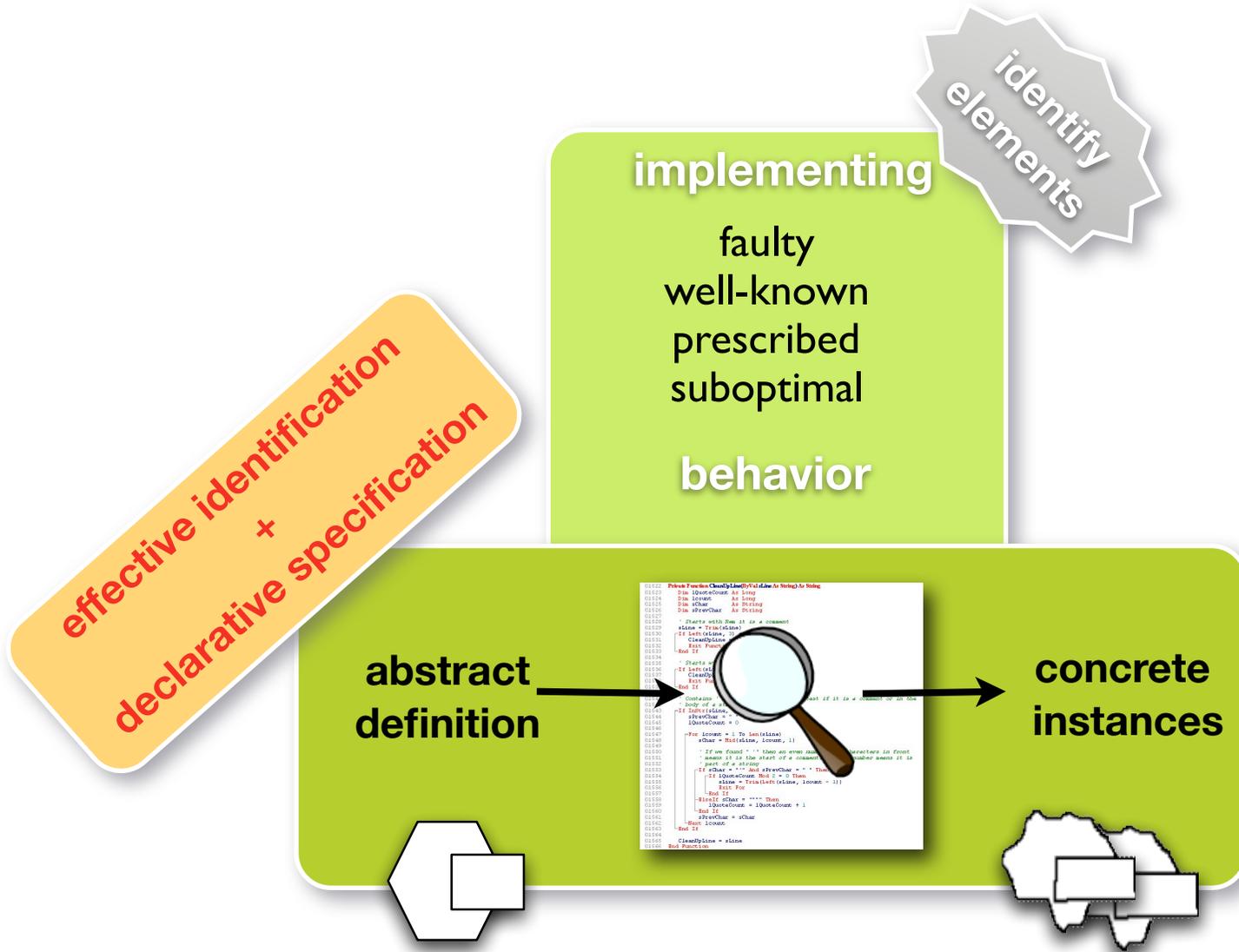
# Logic Source Code Templates

Coen De Roover<sup>▲</sup> Johan Brichau<sup>\*</sup>

▲ Programming Technology Lab - *Vrije Universiteit Brussel* - Belgium

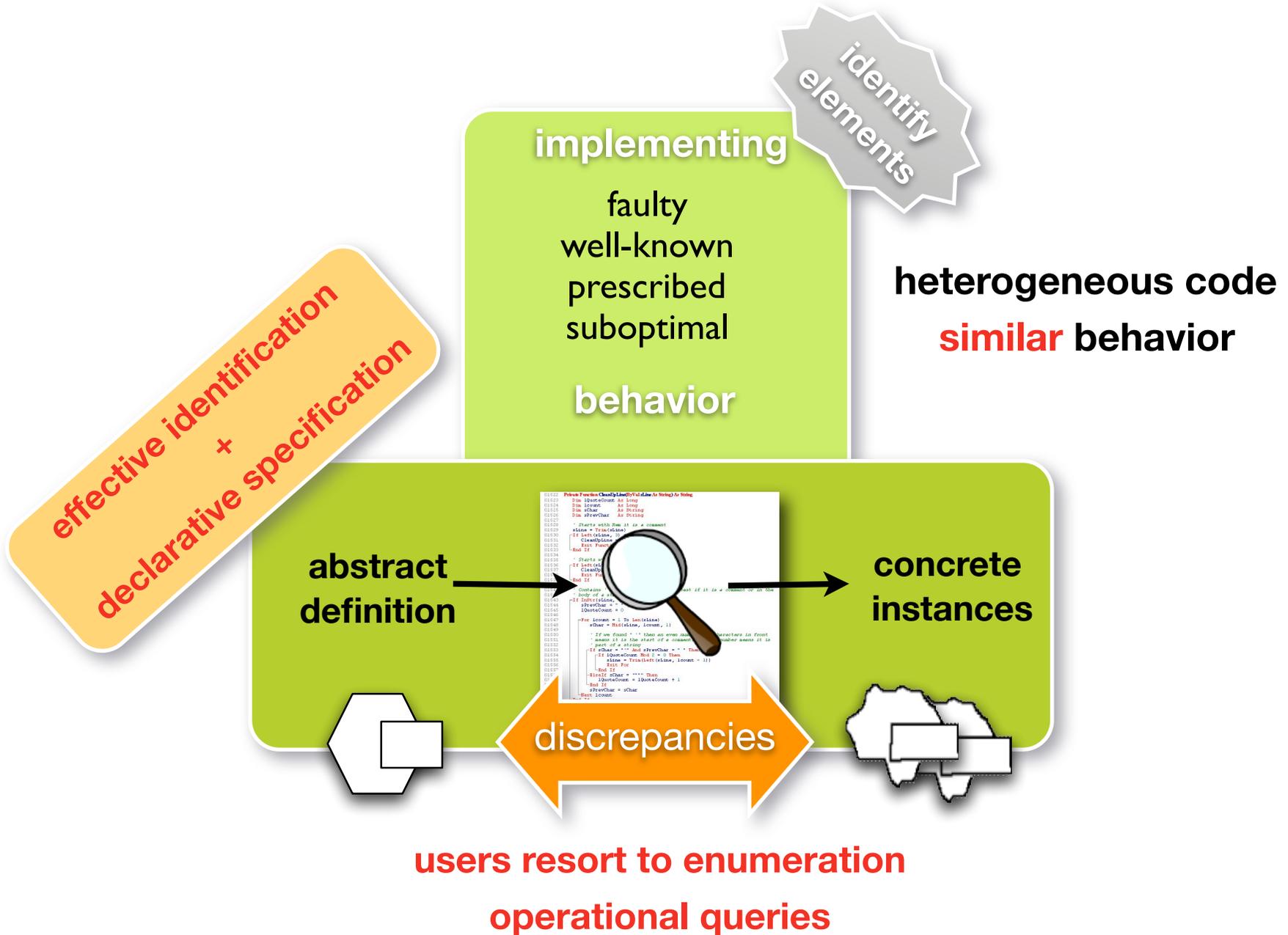
\* Département d'Ingénierie Informatique - *Université catholique de Louvain* - Belgium

# Context: Program Queries





# Context: Program Queries



# Motivating Example: Detecting Getter Methods

```
1 ?m getsFragment: ?g ofFieldDeclaration: ?f in: ?c if
2   ?f isFieldDeclarationInClassDeclaration: ?c,
3   ?f fieldDeclarationHasFragment: ?g,
4   ?g variableDeclarationFragmentHasName: ?name,
5   ?m isMethodDeclarationInClassDeclaration: ?c
6   ?m methodDeclarationHasBody: block(?s),
7   ?s contains: returnStatement(?name)
```

Should detect getter methods in an imaginary\* LMP language.

\* any similarities with Soul minus the extensions I'm about to present are purely incidental and unintentional ;)

# Motivating Example: Detecting Getter Methods

```

1 ?m getsFragment: ?g ofFieldDeclaration: ?f in: ?c if
2 ?f isFieldDeclarationInClassDeclaration: ?c,
3 ?f fieldDeclarationHasFragment: ?g,
4 ?g variableDeclarationFragmentHasName: ?name,
5 ?m isMethodDeclarationInClassDeclaration: ?c
6 ?m methodDeclarationHasBody: block(?s),
7 ?s contains: returnStatement(?name)

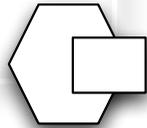
```

Should detect getter methods in an imaginary\* LMP language.

```

class Y {
  private X var;
  public X getVar() {
    return var;
  }
}

```



\* any similarities with Soul minus the extensions I'm about to present are purely incidental and unintentional ;)

# Motivating Example: Detecting Getter Methods

```

1 ?m getsFragment: ?g ofFieldDeclaration: ?f in: ?c if
2 ?f isFieldDeclarationInClassDeclaration: ?c,
3 ?f fieldDeclarationHasFragment: ?g,
4 ?g variableDeclarationFragmentHasName: ?name,
5 ?m isMethodDeclarationInClassDeclaration: ?c
6 ?m methodDeclarationHasBody: block(?s),
7 ?s contains: returnStatement(?name)

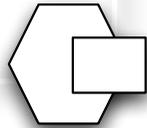
```

Should detect getter methods in an imaginary\* LMP language.

```

class Y {
  private X var;
  public X getVar() {
    return var;
  }
}

```



```

class Y {
  private X var;
  private Y self() {
    return this;
  }
  public X getVar() {
    return this.self().var;
  }
}

```



\* any similarities with Soul minus the extensions I'm about to present are purely incidental and unintentional ;)

# Motivating Example: Detecting Getter Methods

```

1 ?m getsFragment: ?g ofFieldDeclaration: ?f in: ?c if
2 ?f isFieldDeclarationInClassDeclaration: ?c,
3 ?f fieldDeclarationHasFragment: ?g,
4 ?g variableDeclarationFragmentHasName: ?name,
5 ?m isMethodDeclarationInClassDeclaration: ?c
6 ?m methodDeclarationHasBody: block(?s),
7 ?s contains: returnStatement(?name)

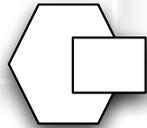
```

Should detect getter methods in an imaginary\* LMP language.

```

class Y {
  private X var;
  public X getVar() {
    return var;
  }
}

```



```

class Y {
  private X var;
  public X notGetVar(X var) {
    return var;
  }
}

```



```

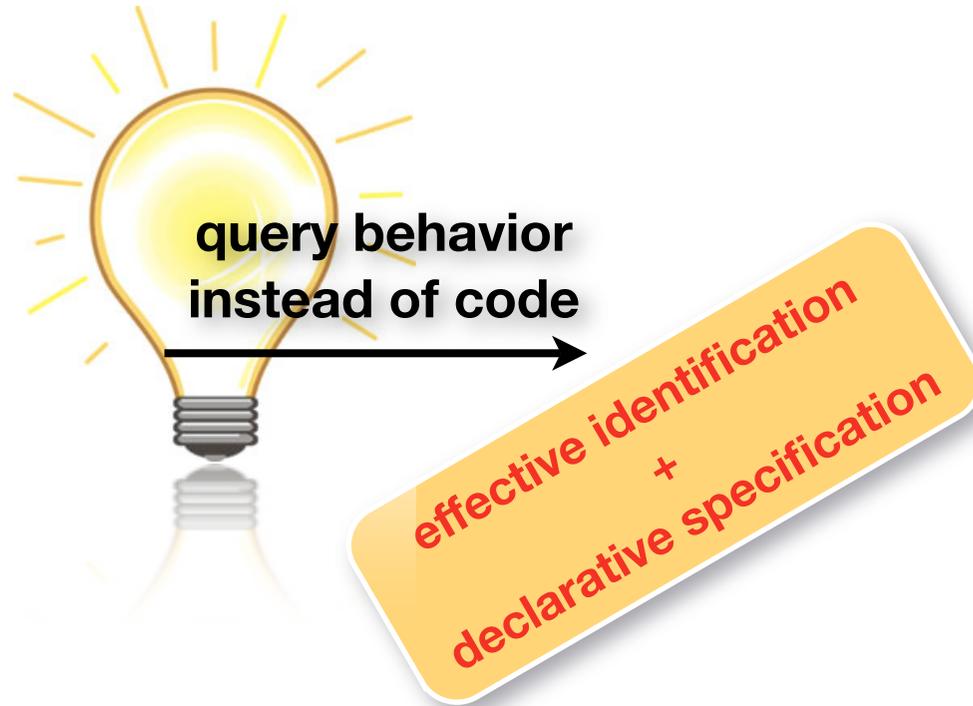
class Y {
  private X var;
  private Y self() {
    return this;
  }
  public X getVar() {
    return this.self().var;
  }
}

```

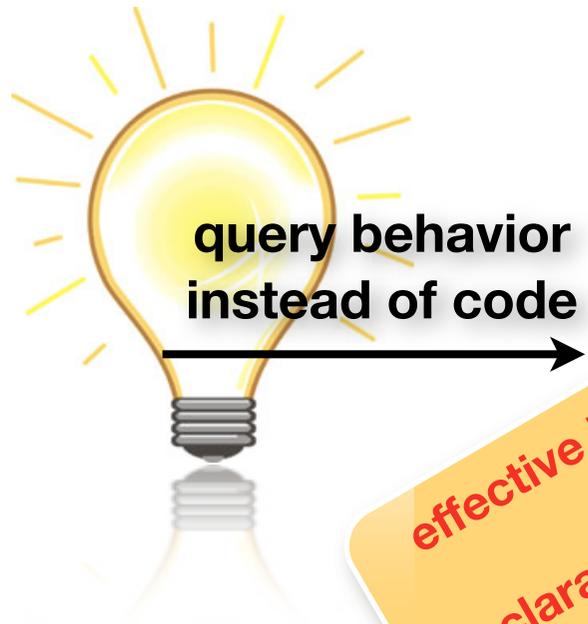


\* any similarities with Soul minus the extensions I'm about to present are purely incidental and unintentional ;)

# Isn't there a Straightforward Solution?

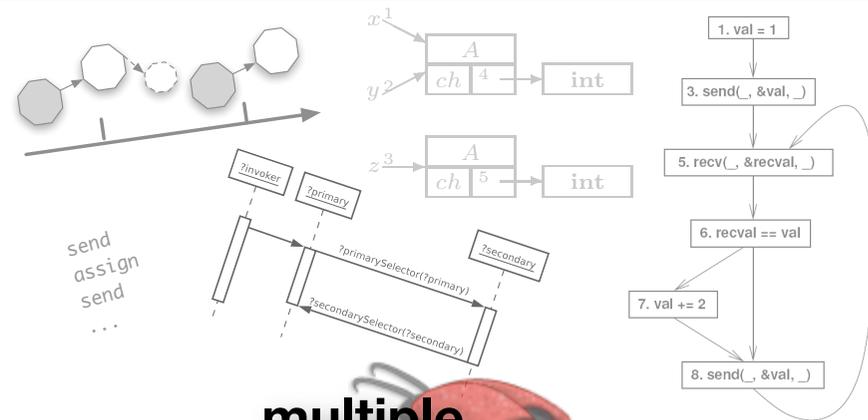


# Isn't there a Straightforward Solution?

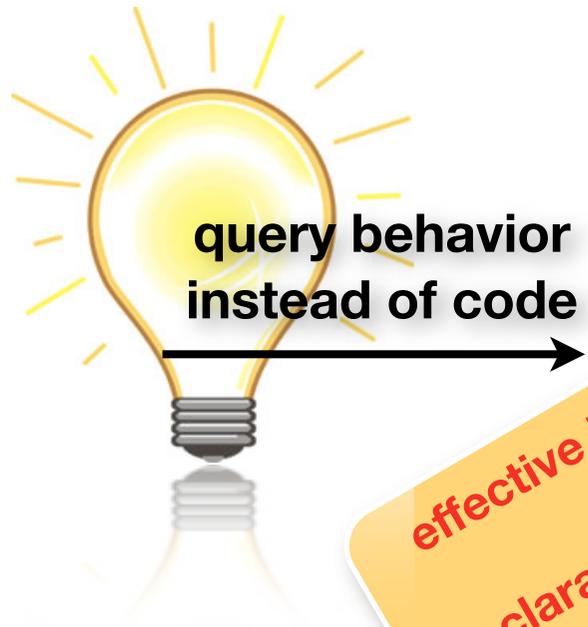


effective identification  
+  
declarative specification

multiple  
possibilities



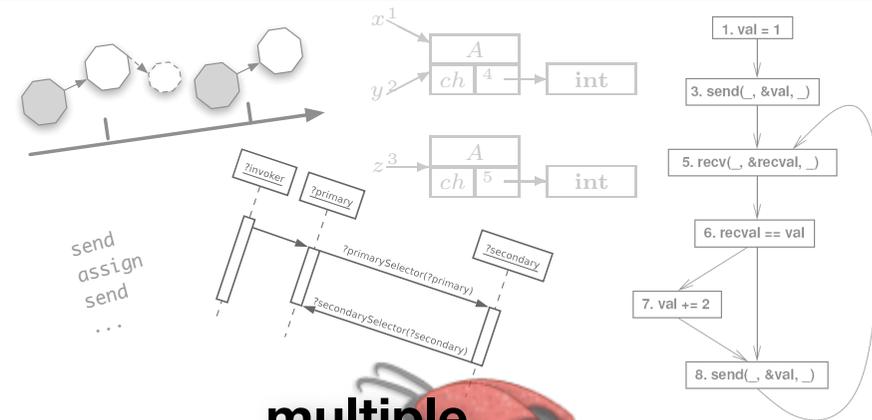
# Isn't there a Straightforward Solution?



effective identification  
+  
declarative specification

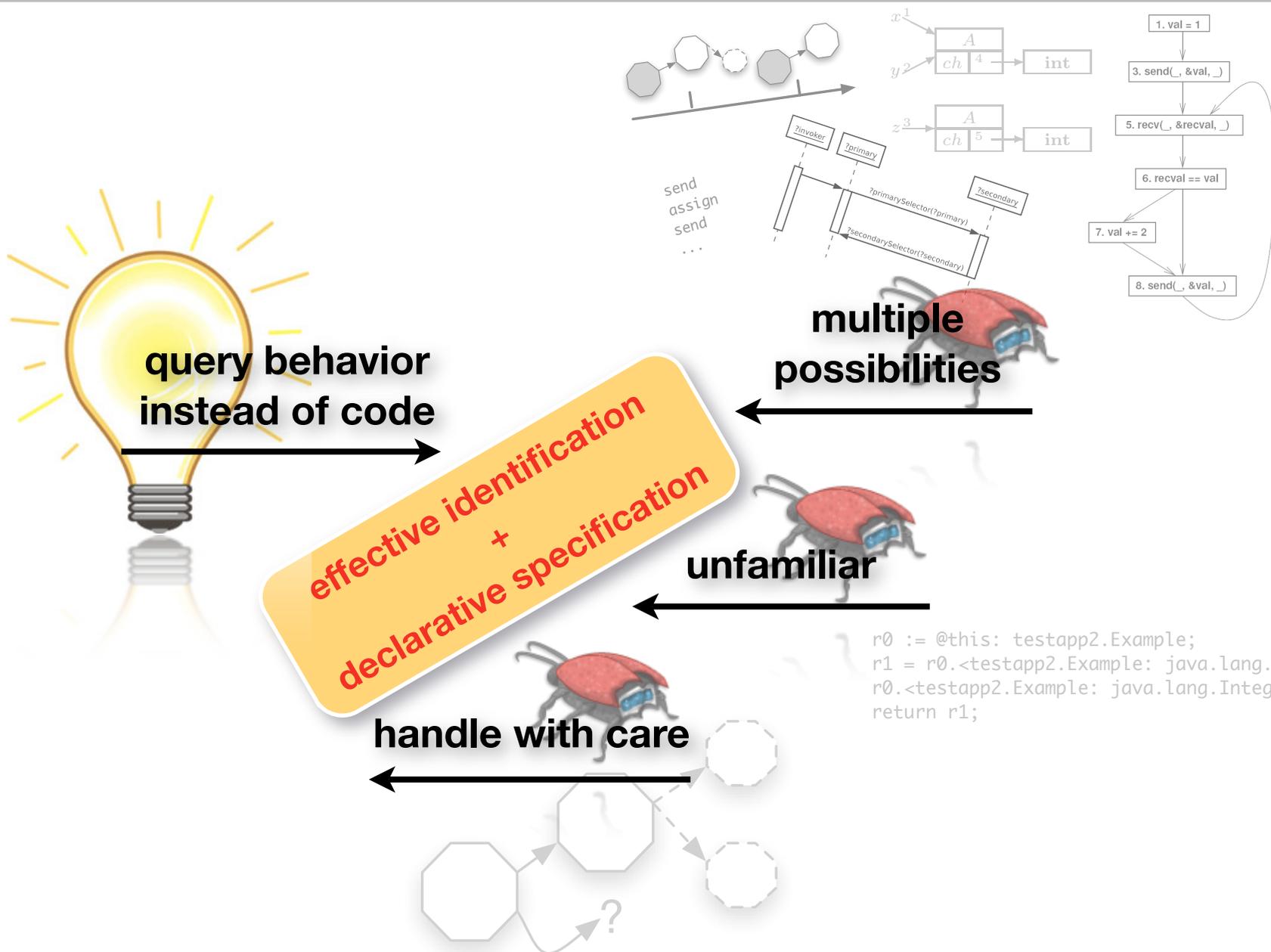
multiple  
possibilities

unfamiliar



```
r0 := @this: testapp2.Example;
r1 = r0.<testapp2.Example: java.lang.Integer buffer>;
r0.<testapp2.Example: java.lang.Integer buffer> = null;
return r1;
```

# Isn't there a Straightforward Solution?



# So, instead ..

```
if jtClassDeclaration(?c){  
    class ?c {  
        private ?type ?field;  
        public ?type ?name() { return ?field; }  
    }  
}
```



```
public Integer gethour() {  
    return this.hour;  
}  
public Integer gethourlazy() {  
    if(hour==null)  
        hour = this.currentHour();  
    return hour;  
}  
public Integer getBuffer() {  
    Integer temp;  
    temp = this.self().buffer;  
    return temp;  
}
```



# Solution Cornerstones

## declarative pattern specification

concrete source code templates

integrated in logic language

## effective pattern identification

resolved by **fuzzy** logic program

**behavioral similarity** determined statically

by open **unification** framework

# Solution Cornerstones

user familiarity

**declarative pattern specification**



**concrete source code templates**

**integrated in logic language**

**effective pattern identification**

resolved by **fuzzy** logic program

**behavioral similarity** determined statically

by open **unification** framework

# Solution Cornerstones

user familiarity

**declarative pattern specification**

template composition



**concrete source code templates**



**integrated in logic language**

**effective pattern identification**

resolved by **fuzzy** logic program

**behavioral similarity** determined statically

by open **unification** framework

# Solution Cornerstones

user familiarity

**declarative pattern specification**

template composition



**concrete source code templates**



**integrated in logic language**



uniform **unification**  
across templates and  
ordinary conditions

**effective pattern identification**

resolved by **fuzzy** logic program

**behavioral similarity** determined statically

by open **unification** framework

# Solution Cornerstones

user familiarity

**declarative pattern specification**

template composition



**concrete source code templates**



**integrated in logic language**



uniform **unification**  
across templates and  
ordinary conditions

**effective pattern identification**

similarity template/match



**resolved by fuzzy logic program**

**behavioral similarity determined statically**  
**by open unification framework**

# Solution Cornerstones

user familiarity

**declarative pattern specification**

template composition



**concrete source code templates**



**integrated in logic language**



uniform **unification**  
across templates and  
ordinary conditions

**effective pattern identification**

similarity template/match



**resolved by fuzzy logic program** → customizable

**behavioral similarity determined statically**  
**by open unification framework**

# Solution Cornerstones

user familiarity

**declarative pattern specification**

template composition



**concrete source code templates**



**integrated in logic language**



uniform **unification**  
across templates and  
ordinary conditions

**effective pattern identification**

similarity template/match



**resolved by fuzzy logic program** → customizable

**behavioral similarity determined statically**  
**by open unification framework**



transparent to user

# Solution Cornerstones

user familiarity

**declarative pattern specification**

template composition



**concrete source code templates**



**integrated in logic language**



uniform **unification**  
across templates and  
ordinary conditions

**effective pattern identification**

similarity template/match



**resolved by fuzzy logic program**



customizable

**behavioral similarity determined statically**

**by open unification framework**



family of analyses



transparent to user

# Solution Cornerstones

user familiarity

**declarative pattern specification**

template composition



**concrete source code templates**



**integrated in logic language**



uniform **unification**  
across templates and  
ordinary conditions

**effective pattern identification**

similarity template/match



**resolved by fuzzy logic program** → customizable

**behavioral similarity determined statically**

**by open unification framework** → approximated results



family of analyses



transparent to user

# Fuzzy Logic Programming

## logic of quantified truth

rules annotated with partial **truth degrees**

✓ **confidence** in instances detected by rule

fuzzy logic

## weighted logic rules

$q : c \text{ if } q_1, \dots, q_n \text{ where } c \in ]0, 1]$

similar to  
f-Prolog  
[1990:liu]

# Fuzzy Logic Programming

## logic of quantified truth

rules annotated with partial **truth degrees**

✓ **confidence** in instances detected by rule

fuzzy logic

## weighted logic rules

$q : c \text{ if } q_1, \dots, q_n \text{ where } c \in ]0, 1]$

confidence  
in conclusion  $q$  given absolute  
truth of  $q_1, \dots, q_n$

similar to  
f-Prolog  
[1990:liu]

# Fuzzy Logic Programming

## logic of quantified truth

rules annotated with partial **truth degrees**

✓ **confidence** in instances detected by rule

fuzzy logic

## fuzzy resolution procedure

**conclusions** from **partially satisfied premises**

✓ detect partially adhering instances

## weighted logic rules

$q : c \text{ if } q_1, \dots, q_n \text{ where } c \in ]0, 1]$

## fuzzy resolution procedure

$\tau(q) = c * \min(\tau(q_1), \dots, \tau(q_n))$

confidence  
in conclusion  $q$  given absolute  
truth of  $q_1, \dots, q_n$

similar to  
f-Prolog  
[1990:liu]

# Fuzzy Logic Programming

## logic of quantified truth

rules annotated with partial **truth degrees**

✓ **confidence** in instances detected by rule

fuzzy logic

## fuzzy resolution procedure

**conclusions** from **partially satisfied premises**

✓ detect partially adhering instances

## weighted logic rules

$q : c \text{ if } q_1, \dots, q_n \text{ where } c \in ]0, 1]$

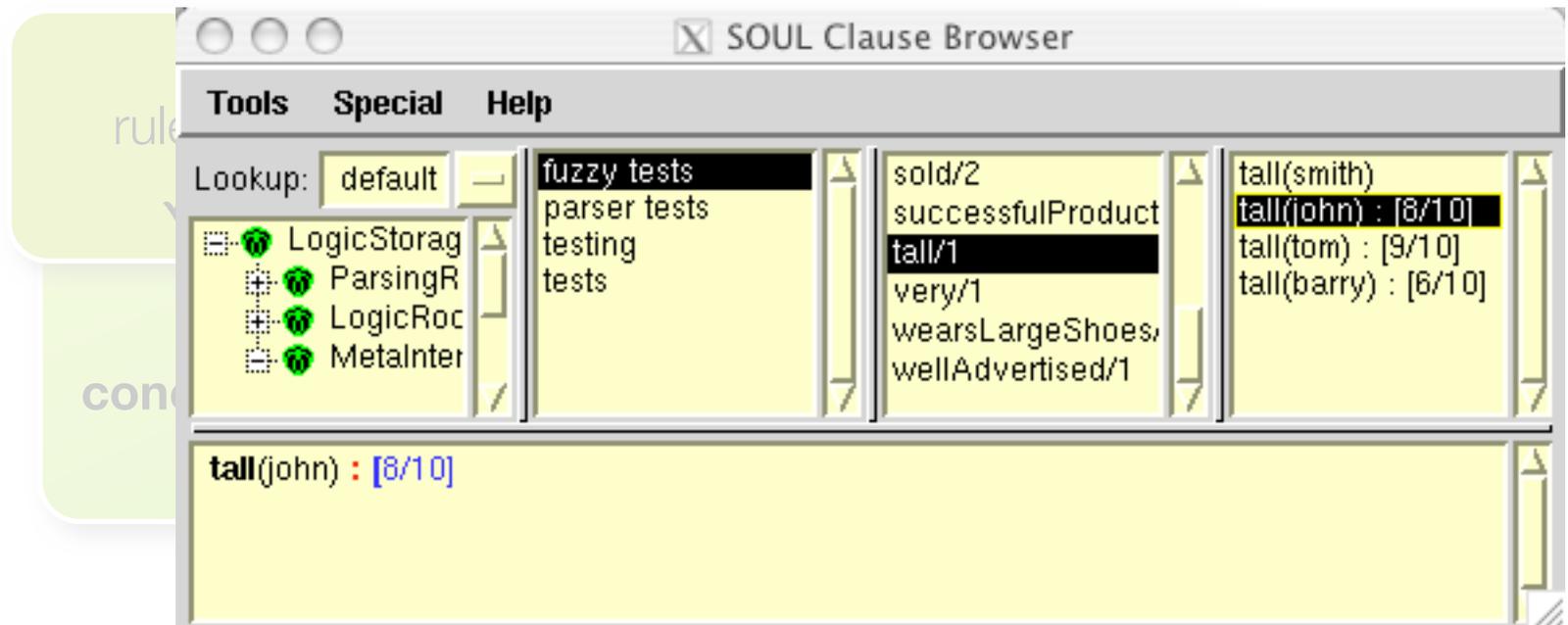
## fuzzy resolution procedure

$\tau(q) = c * \min(\tau(q_1), \dots, \tau(q_n))$

confidence  
in conclusion  $q$  given absolute  
truth of  $q_1, \dots, q_n$

similar to  
f-Prolog  
[1990:liu]

# Fuzzy Logic Programming



## weighted logic rules

$q : c \text{ if } q_1, \dots, q_n \text{ where } c \in ]0, 1]$

## fuzzy resolution procedure

$\tau(q) = c * \min(\tau(q_1), \dots, \tau(q_n))$

confidence  
in conclusion  $q$  given absolute  
truth of  $q_1, \dots, q_n$

similar to  
f-Prolog  
[1990:liu]

# Fuzzy Logic Programming

SOUL Clause Browser

Tools Special Help

Lookup: default

- LogicStorage
- ParsingR
- LogicRoc
- MetalInter

fuzzy tests  
parser tests  
testing  
tests

sold/2  
successfulProduct  
tall/1  
very/1  
wearsLargeShoes,  
wellAdvertised/1

tall(smith)  
tall(john) : [8/10]  
tall(tom) : [9/10]  
tall(barry) : [6/10]

tall(john) : [8/10]

SOUL Clause Browser

Tools Special Help

Lookup: default

- LogicStorage
- ParsingR
- LogicRoc
- MetalInter

fuzzy tests  
parser tests  
testing  
tests

sold/2  
successfulProduct  
tall/1  
very/1  
wearsLargeShoes,  
wellAdvertised/1

wearsLargeShoes

wearsLargeShoes(?x) : [9/10] if  
tall(?x)

confidence  
inclusion  $q$  given absolute  
truth of  $q_1, \dots, q_n$

1 ]

similar to  
f-Prolog  
[1990:liu]

$$l(q) = c * \min(l(q_1), \dots, l(q_n))$$

# Fuzzy Logic Programming

The image shows two overlapping windows from the SOUL system. The top window, titled "SOUL Clause Browser", contains the query: `if wearsLargeShoes(?p) : ?t`. Below the query, it shows "Lookup in: default" and "4 solutions in 3 ms". The "Evaluator" is set to "Evaluator" with a "Configure" button. The bottom window, titled "SOUL Querybrowser", displays the results in a tree view. The root node is "?t ?p". It has four children, each representing a solution with a score in parentheses: `(81/100) #tom`, `(27/50) #barry`, `(18/25) #john`, and `(9/10) #smith`. The "#smith" node is highlighted. To the right of the query editor, there are buttons for "All Results", "All Results Int.", "Next Result", and "Basic Inspect". Below these is a "Variable View Ordering" section with a list containing "?t" and "?p", and buttons for "Apply" and "Clear".

# Open Template Compilation

```
jtExpression(?cast) { (java.lang.Object) ?expression }
```

# Open Template Compilation

constraints on possible bindings



```
jtExpression(?cast) { (java.lang.Object) ?expression }
```

1/ **cast** expression in **base** program

2/ **unifying** types

3/ **unifying** subexpressions

# Open Template Compilation

constraints on possible bindings



```
jtExpression(?cast) { (java.lang.Object) ?expression }
```

1/ **cast** expression in **base** program

2/ **unifying** types

3/ **unifying** subexpressions

**compiled by fuzzy logic program called by interpreter**

results in multiple logic queries for each parse tree of a template

# Open Template Compilation

constraints on possible bindings



```
jtExpression(?cast) { (java.lang.Object) ?expression }
```

1/ **cast** expression in **base** program

2/ **unifying** types

3/ **unifying** subexpressions

**compiled by fuzzy logic program called by interpreter**

results in multiple logic queries for each parse tree of a template

**weighted** by their **tolerance** for **mismatches**

```
if jtStatement(?block) { ?s1; ?s2; ?s3;
```

layers of indirection, intertwining statement, loops, ...

# Multiple Program Representations

# Multiple Program Representations

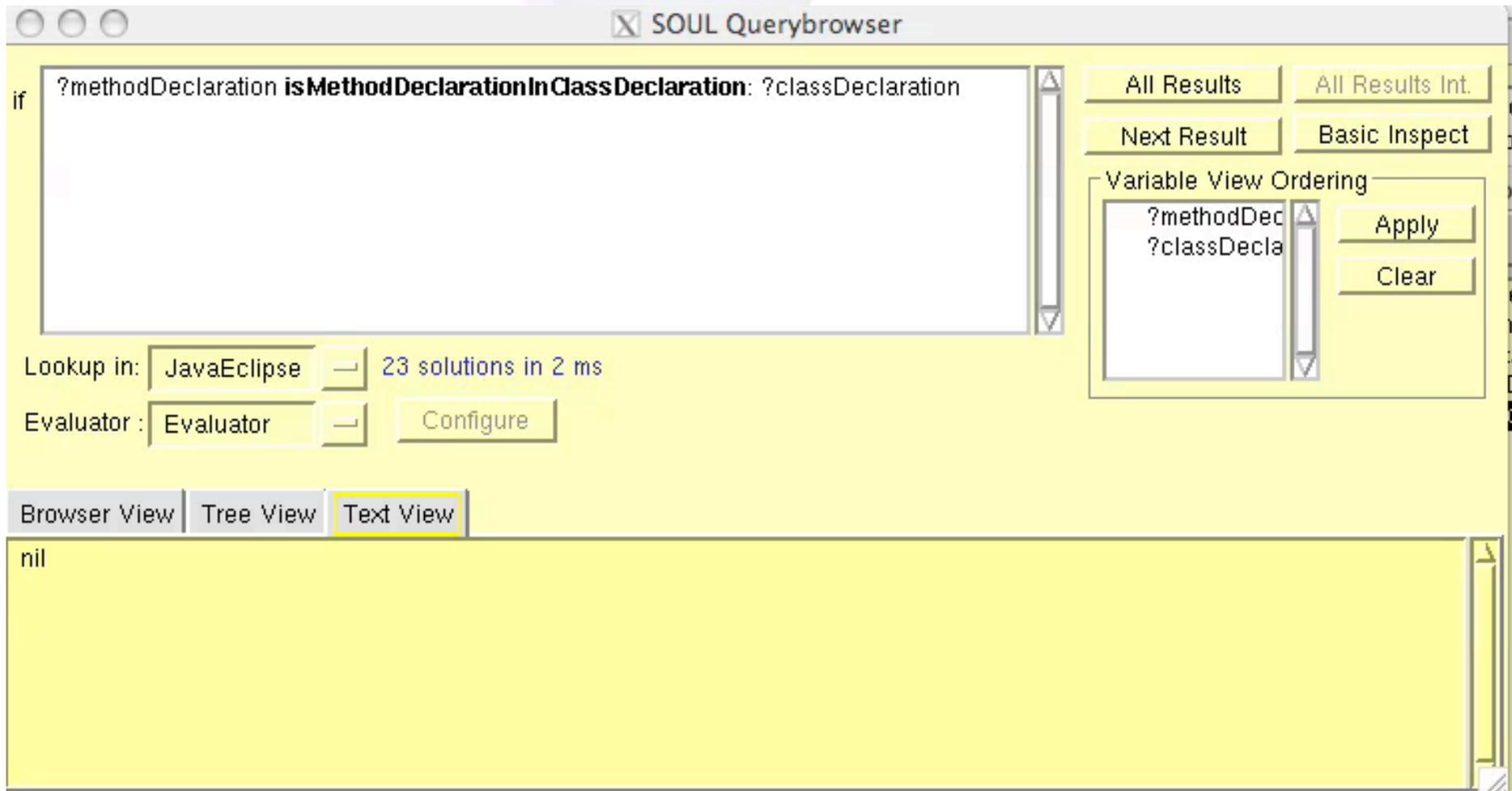
## Abstract Syntax Tree

live  org.eclipse.jdt.core.dom.ASTNode instances  
**NOT** reified as logic facts

# Multiple Program Representations

## Abstract Syntax Tree

live  org.eclipse.jdt.core.dom.ASTNode instances



The screenshot shows the SOUL Querybrowser window with the following details:

- Title Bar:** SOUL Querybrowser
- Query:** `if ?methodDeclaration isMethodDeclarationInClassDeclaration: ?classDeclaration`
- Buttons:** All Results, All Results Int., Next Result, Basic Inspect.
- Variable View Ordering:** A list containing `?methodDec` and `?classDecla`, with `Apply` and `Clear` buttons.
- Lookup in:** JavaEclipse (dropdown), 23 solutions in 2 ms
- Evaluator:** Evaluator (dropdown), `Configure` button.
- View Modes:** Browser View, Tree View, **Text View** (selected).
- Result:** nil

# Multiple Program Representations

## Abstract Syntax Tree

live  org.eclipse.jdt.core.dom.ASTNode instances  
**NOT** reified as logic facts

# Multiple Program Representations

## Abstract Syntax Tree

live  org.eclipse.jdt.core.dom.ASTNode instances  
**NOT** reified as logic facts

## Dataflow Analysis: points-to

safe **approximation** of run-time **objects** a reference points to

x.y



```
{ AllocNode 3 new java.lang.Integer in method  
<testapp2.Example: void main(java.lang.String[])>,  
...  
AllocNode 7 new java.lang.Integer in method  
<testapp2.Example: void main(java.lang.String[])> }
```

# Multiple Program Representations

## Abstract Syntax Tree

live  org.eclipse.jdt.core.dom.ASTNode instances  
**NOT** reified as logic facts

## Dataflow Analysis: points-to

safe **approximation** of run-time **objects** a reference points to

$x.y \rightarrow$  { AllocNode 3 new java.lang.Integer in method  
 <testapp2.Example: void main(java.lang.String[])> ,  
 ...  
 AllocNode 7 new java.lang.Integer in method  
 <testapp2.Example: void main(java.lang.String[])> }

## Control flow

between

within methods

# Multiple Program Representations

The screenshot shows the SOUL Querybrowser application window. The title bar reads "SOUL Querybrowser". The main text area contains the following query:

```
if jtClassDeclaration(?class) {  
  class ?c {  
    ?modList ?type ?name() { return ?e; }  
  }  
},  
equals(?p, [?e retrievePointsToSet])
```

Below the query, the "Lookup in:" field is set to "JavaEclipse" and shows "9 solutions in 177 ms". The "Evaluator:" field is set to "Evaluator" with a "Configure" button next to it.

On the right side, there are several buttons: "All Results", "All Results Int.", "Next Result", and "Basic Inspect". Below these is a "Variable View Ordering" section with a list of variables: "?e", "?c", "?type", "?name", and "?class". There are "Apply" and "Clear" buttons next to this list.

At the bottom, there are three tabs: "Browser View", "Tree View", and "Text View". The "Text View" tab is selected, and the content area below it displays "nil".

# Multiple Program Representations

## Abstract Syntax Tree

live  org.eclipse.jdt.core.dom.ASTNode instances  
**NOT** reified as logic facts

## Dataflow Analysis: points-to

safe **approximation** of run-time **objects** a reference points to

$x.y \rightarrow$  { AllocNode 3 new java.lang.Integer in method  
 <testapp2.Example: void main(java.lang.String[])>,  
 ...  
 AllocNode 7 new java.lang.Integer in method  
 <testapp2.Example: void main(java.lang.String[])> }

## Control flow

between

within methods

# Smalltalk Open Unification Language

Unification with logic terms

Unification with other  instances

# Smalltalk Open Unification Language

## Unification with logic terms

easily identify elements of interest

```
1 if ?c isCompilationUnit,  
2   [?c types size > 1]  
  
3 if compilationUnit(packageDeclaration(simpleName(['testapp'])), ?, ?) isCompilationUnit  
  
4 if ?c isCompilationUnit,  
5   ?c hasPackage: ?p,  
6   ?p hasName: ?n,  
7   ?n isSimpleName,  
8   ?n hasIdentifier: ['testapp']
```

Unification with other  instances

# Smalltalk Open Unification Language

**Unification with logic terms**      powered by structural reflection

easily identify elements of interest

```

1 if ?c isCompilationUnit,
2   [?c types size > 1]

3 if compilationUnit(packageDeclaration(simpleName(['testapp'])), ?, ?) isCompilationUnit

4 if ?c isCompilationUnit,
5   ?c hasPackage: ?p,
6   ?p hasName: ?n,
7   ?n isSimpleName,
8   ?n hasIdentifier: ['testapp']

```

**Unification with other**  **instances**

# Smalltalk Open Unification Language

**Unification with logic terms** powered by structural reflection

easily identify elements of interest

```

1 if ?c isCompilationUnit,
2   [?c types size > 1]

3 if compilationUnit(packageDeclaration(simpleName(['testapp'])), ?, ?) isCompilationUnit

4 if ?c isCompilationUnit,
5   ?c hasPackage: ?p,
6   ?p hasName: ?n,
7   ?n isSimpleName,
8   ?n hasIdentifier: ['testapp']

```

**Unification with other  instances** overrides default identity comparison

implement recurring comparisons

Expression  $\stackrel{?}{=}$  ParenthesizedExpression

Type  $\stackrel{?}{=}$  TypeDeclaration

SimpleName  $\stackrel{?}{=}$  QualifiedName

# Smalltalk Open Unification Language

**Unification with logic terms** powered by structural reflection

easily identify elements of interest

```
1 if ?c isCompilationUnit,  
2   [?c types size > 1]  
  
3 if compilationUnit (packageDeclaration (simpleName (['testapp'])), ?, ?) isCompilationUnit  
  
4 if ?c isCompilationUnit,  
5   ?c hasPackage: ?p,  
6   ?p hasName: ?n,  
7   ?n isSimpleName,  
8   ?n hasIdentifier: ['testapp']
```

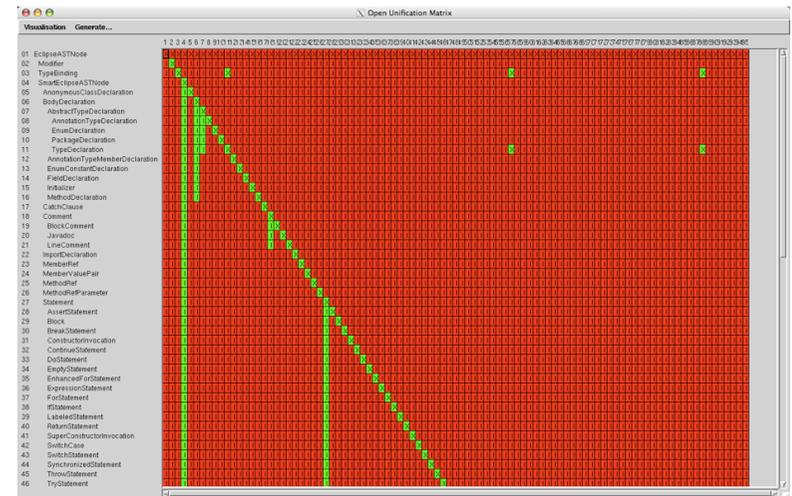
**Unification with other instances**  overrides default identity comparison

implement recurring comparisons

Expression  $\stackrel{?}{=}$  ParenthesizedExpression

Type...  $\stackrel{?}{=}$  TypeDeclaration

SimpleName  $\stackrel{?}{=}$  QualifiedName



# Smalltalk Open Unification Language

**Unification with logic terms** powered by structural reflection

easily identify elements of interest

```
1 if ?c isCompilationUnit,  
2   [?c types size > 1]  
  
3 if compilationUnit (packageDeclaration (simpleName (['testapp'])), ?, ?) isCompilationUnit  
  
4 if ?c isCompilationUnit,  
5   ?c hasPackage: ?p,  
6   ?p hasName: ?n,  
7   ?n isSimpleName,  
8   ?n hasIdentifier: ['testapp']
```

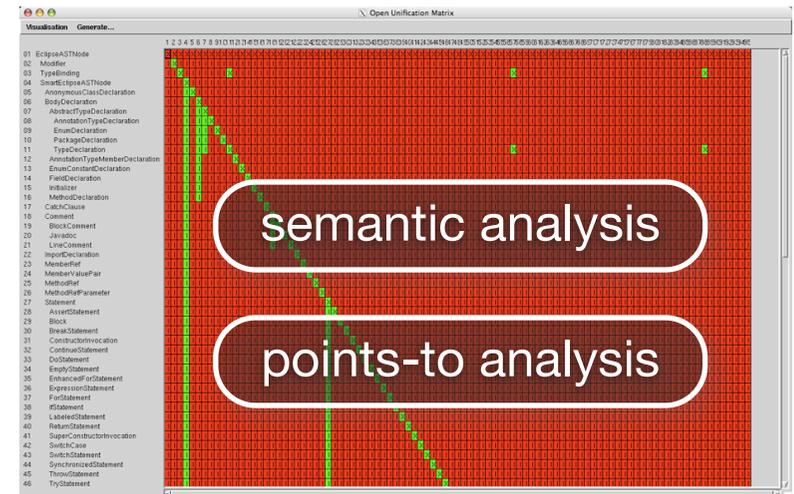
**Unification with other instances**  overrides default identity comparison

implement recurring comparisons

Expression  $\stackrel{?}{=}$  ParenthesizedExpression

Type...  $\stackrel{?}{=}$  TypeDeclaration

SimpleName  $\stackrel{?}{=}$  QualifiedName



# Unification based on Semantic Analysis

**across logic and template conditions**

# Unification based on Semantic Analysis

*Conciseness*

**across logic and template conditions**

# Unification based on Semantic Analysis

*Conciseness*

*Correctness*

**across logic and template conditions**

# Unification based on Semantic Analysis

## name resolution

```
1 if jtClassDeclaration(?class) {  
2     class FooClass extends testapp2.Example implements Collection {}  
3 }  
4 if jtClassDeclaration(?class) {  
5     class FooClass extends Example implements java.util.Collection {}  
6 }  
  
package testapp2;  
public class FooClass extends testapp2.Example implements Collection {}
```

Conciseness

Correctness

across logic and template conditions

# Unification based on Semantic Analysis

## name resolution

```
1 if jtClassDeclaration(?class) {
2     class FooClass extends testapp2.Example implements Collection {}
3 }
4 if jtClassDeclaration(?class) {
5     class FooClass extends Example implements java.util.Collection {}
6 }

package testapp2;
public class FooClass extends testapp2.Example implements Collection {}
```

Conciseness

## scoping rules

```
1 ?m getsFragment: ?g ofFieldDeclaration: ?f in: ?c if
2 ?f isFieldDeclarationInClassDeclaration: ?c,
3 ?f fieldDeclarationHasFragment: ?g,
4 ?g variableDeclarationFragmentHasName: ?name,
5 ?m isMethodDeclarationInClassDeclaration: ?c
6 ?m methodDeclarationHasBody: block(?s),
7 ?s contains: returnStatement(?name)

private Integer f;
public Integer notGettingF(Integer f) { return f; }
```

Correctness

across logic and template conditions

# Unification based on Points-to Analysis

```
if jtStatement(?s1){ return ?expression; },  
    jtStatement(?s2){ return ?expression; },  
differs(?s1, ?s2)
```

# Unification based on Points-to Analysis

*if* jtStatement(*?s1*) { return *?expression*; },  
 jtStatement(*?s2*) { return *?expression*; },  
 differs(*?s1*, *?s2*)

<i>?s1</i> = return foo;	<i>?s2</i>
	return foo;
	return this.foo;
	return this.self().foo;
	x = foo; return x;
	return o.returnArgument(foo);

**syntactically differing return statements  
 possibly returning overlapping sets of objects**

# Unification based on Points-to Analysis

```

if jtStatement(?s1){ return ?expression; },
    jtStatement(?s2){ return ?expression; },
    differs(?s1, ?s2)
  
```

constraint over values returned at run-time

Behavioral Similarity

?s1 = return foo;	?s2
	return foo;
	return this.foo;
	return this.self().foo;
	x = foo; return x;
	return o.returnArgument(foo);

syntactically differing return statements  
possibly returning overlapping sets of objects

# Template Resolution Semantics

## design guideline

close to concrete source code of prototypical implementation  
match many implementation variants

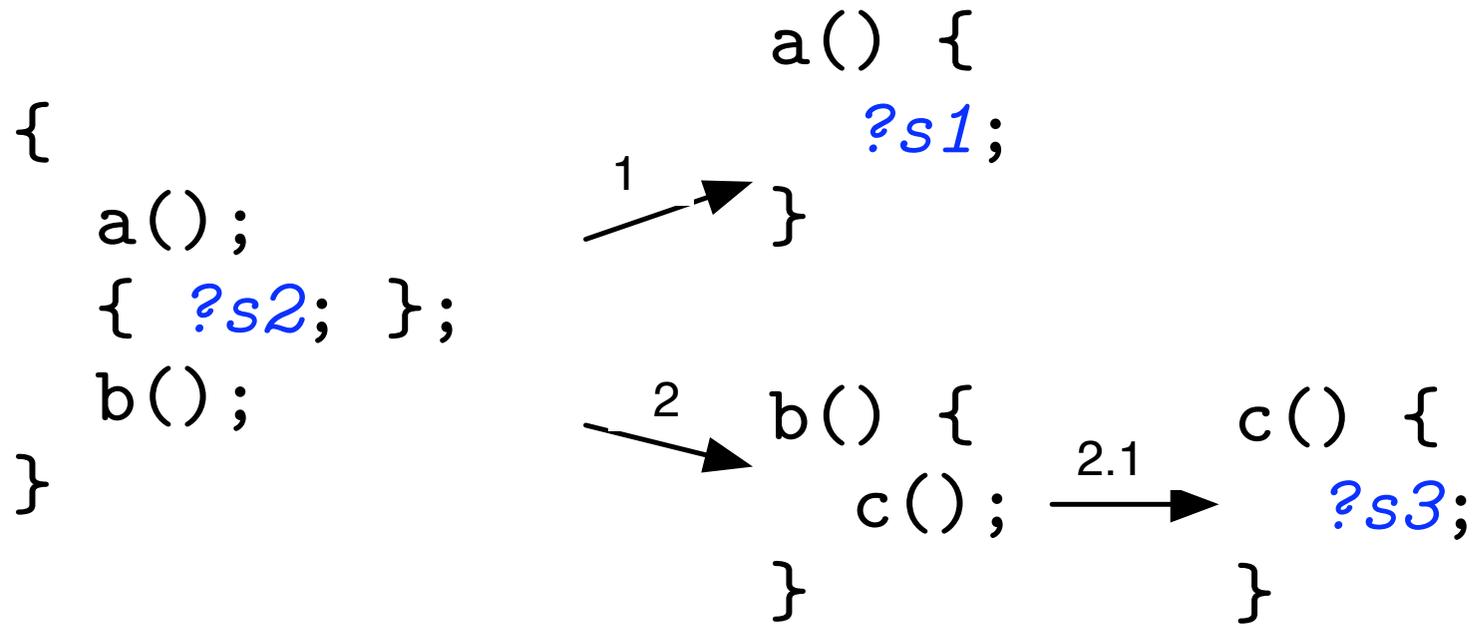
interpreted as freely as possible

**BUT** every single element in template adds constraints

`while(?cond) ?s;`    **vs**    `while(?cond) { ?s; }`

HOWEVER: an open implementation, so amenable by application programmer

# Template Semantics: Sequences



*if* jtStatement( *?block* ) { *?s1*; *?s2*; *?s3*; }



**1/ control flow** constraint

*?s*'s in interprocedural control-flow of *?block*

*?s2* follows: *?s1*, *?s3* follows: *?s2*

**2/ kind** constraint (statement or expression statement)

# Template Semantics: Statements and Expressions

```
jtMethodDeclaration(?m) {
  ?modList ?type ?name(?argList) {
    return ?x.m();
  }
}
```

```
public A method() {
  A local = ?x.m();
  return local;
}
```

- has a **return statement** `return(?val)` among its **lexical** statements
- has an expression `?e` which matches `?x.m()`
  - in its inter-procedural **control flow** (before the statement)
- the statement's returned expression `?val` **unifies with** `?e`

# An Example: Concurrent Modification Exceptions

```
if jtStatement(?s) {  
    while(?iterator.hasNext()) {  
        ?collection.add(?element);  
    }  
},  
jtExpression(?iterator){?collection.iterator()}
```



```
public List list;  
  
public void insertElement(Object x) {  
    Iterator i = list.iterator();  
    while(i.hasNext()) {  
        Object o = i.next();  
        operation(x, (Collection) this.self().list);  
    }  
}  
  
public void operation(Object o, Collection c) {  
    c.add(o);  
}
```



# An Example: Concurrent Modification Exceptions

```

if jtStatement(?s) {
  while(?iterator.hasNext()) {
    ?collection.add(?element);
  }
},
jtExpression(?iterator){?collection.iterator()}

```



```

public List list;

public void insertElement(Object x) {
  Iterator i = list.iterator();
  while(i.hasNext()) {
    Object o = i.next();
    operation(x, (Collection) this.self().list);
  }
}

public void operation(Object o, Collection c) {
  c.add(o);
}

```

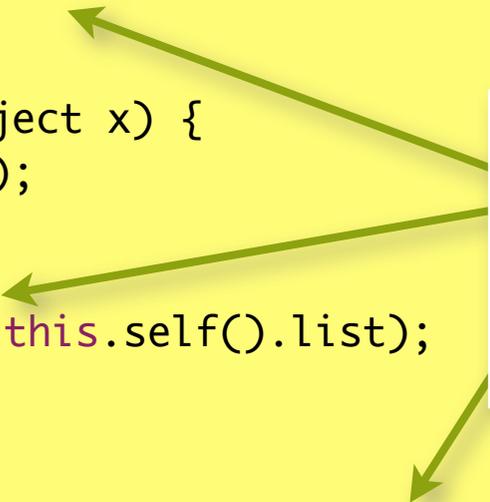
**?s**



**in cflow  
of while**



**unify  
according  
to  
dataflow**




# Summary

## **concrete code templates integrated in logic program query language**

unification uniform across regular and template conditions

composition by logic connectives

open template compilation rules

## **quantified similarity template-match**

fuzzy logic

## **open unification framework**

incorporates static analyses in user-transparent way

query conciseness & correctness

statically determined behavioral similarity

**The End**