Roel Wuyts
Programming Technology Lab
Vrije Universiteit Brussel

Declarative Reasoning in Smalltalk: the implementation and use of SOUL

ESUG Summer School '98, Brescia

Overview

1. Introduction
2. Logic Programming
3. Implementation of SOUL
4. Declarative Framework
5. Future Work
6. Conclusion
7. Demonstration (System - Tools)

Map

1. Introduction

2. Logic Programming

3. Implementation of SOUL

4. Declarative Framework

5. Future Work

6. Conclusion

7. Demonstration (System - Tools)

1. Introduction: Context

- Evolution in OO Software Engineering: extend reusability, adaptibility, maintainability, customizability, …

  from implementation to design
- Drawbacks:
  - current systems form tangled web of communicating objects
  - No explicit link between design structures and code

1. Introduction : Context

- Link between implementation and design is lost

  ⇨ No support for design techniques like for example design patterns

- Making the link:
  - ***Query*** an existing system
  - ***Enforce*** in new system

1. Introduction: Context

- In the development process there is a need to reason on a high-level about the structure of object-oriented systems

⇨ explicit, general, declarative system to express and extract structural relationships in class-based object-oriented systems

⇨ querying and enforcement of structure becomes possible

1. Introduction: Example

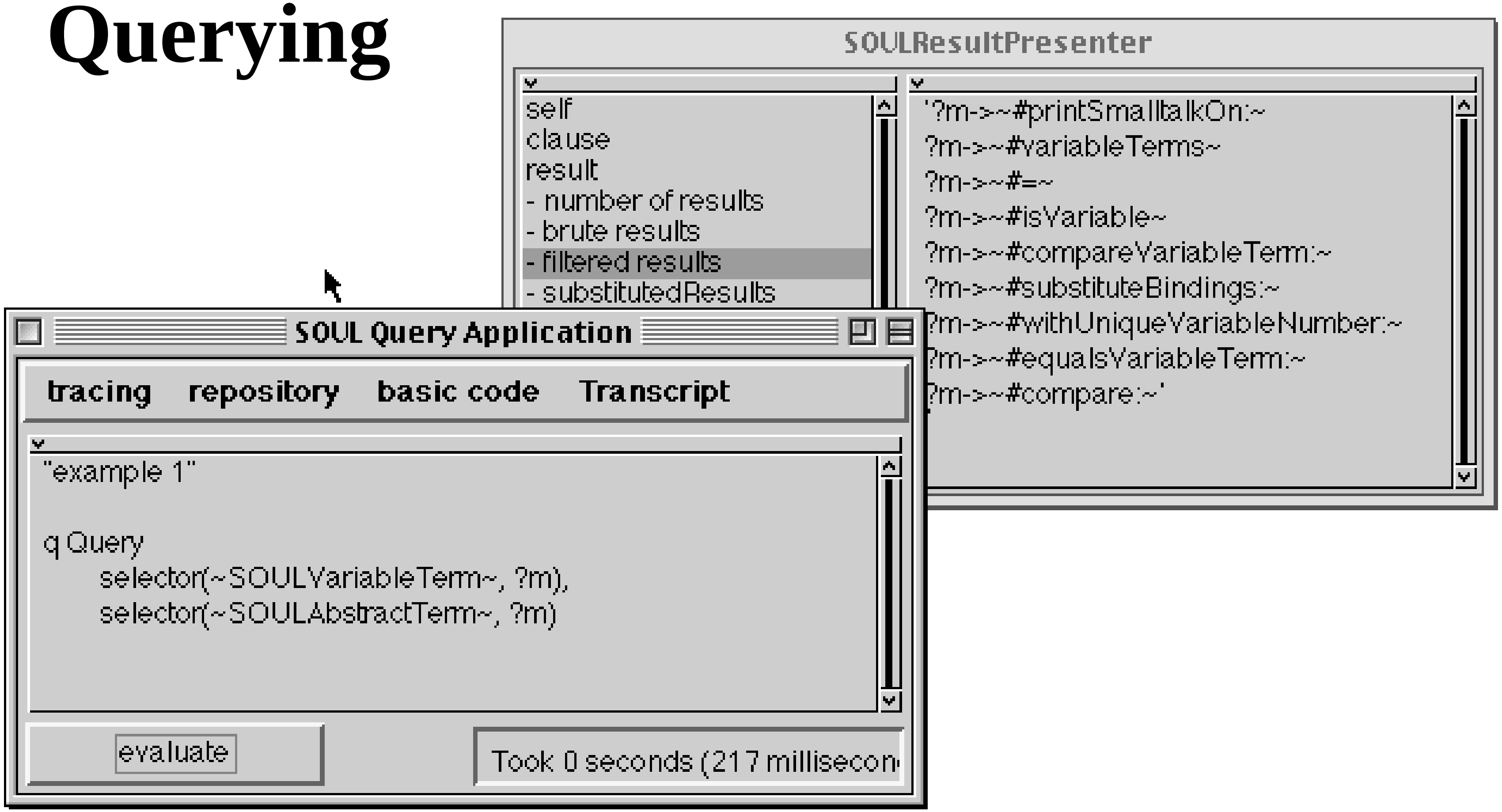
- Express structural information
  - For querying an existing system
  - For enforcement

- Common Methods:

```
Query
    selector(?class1,?selector),
    selector(?class2,?selector)
```

1. Introduction: Example

Querying



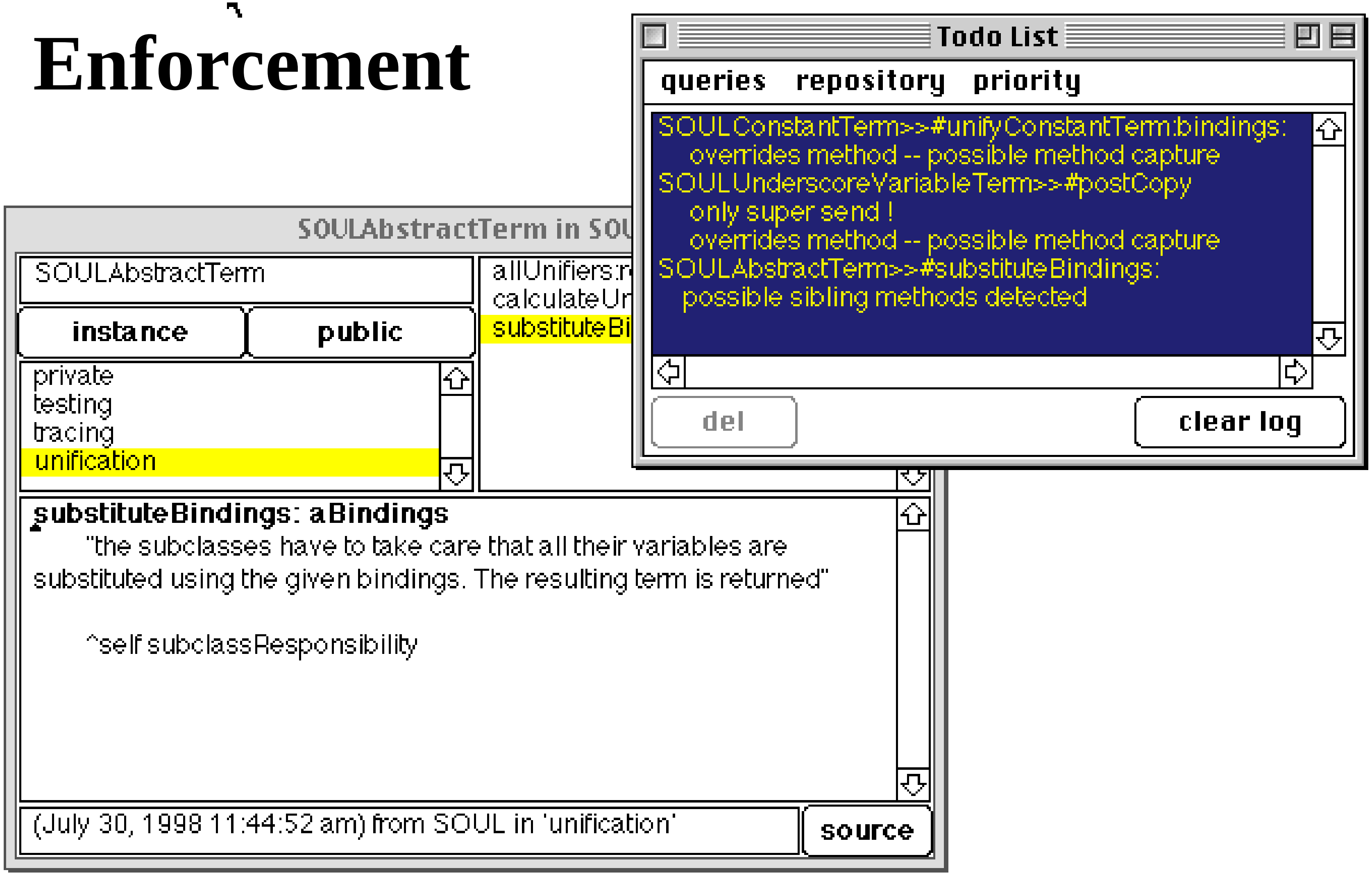
1. Introduction: Example

"detect candidates for possible refactoring of sibling methods for ?MyClass and ?myMethod"

```
Query
  hierarchy (?supers,?MyClass),
  not(selector(?supers,?myMethod)),
  hierarchy(?supers,?others),
  not(equals(?others,?MyClass)),
  selector(?others,?myMethod)
```

1. Introduction: Example

Enforcement



1. Introduction: Example

“find sibling method candidates, and compare their method bodies to find identical statements. These could be refactored to a method in a new common superclass”

Query

```
siblings(?MyClass,?myMethod,?c),
statements(?MyClass,?myMethod,?myStats),
statements(?c,?myMethod,?stats),
commons(?myStats,?stats,?commonStats)
```

Map

1. Introduction

2. Logic Programming

3. Implementation of SOUL

4. Declarative Framework

5. Future Work

6. Conclusion

7. Demonstration (System - Tools)

2. Logic Programming

- Declarative Programs:
  - Program = Data. (Control is general/implicit)
  - Specify *what*, not *how*

- Facts/Rules: State data (stored in repository)
  Queries: interrogate data

2. Logic Programming

- Example:

```
Fact class([Collection]).
Fact class([ArrayedCollection]).
Fact abstractMethod([Collection], [#add:]).
Rule abstractClass(?c) if
    class(?c),
    abstractMethod(?c, ?dummy).

Query abstractClass([Collection])
```

--> true

```
Query abstractClass([ArrayedCollection])
```

--> false

2. Logic Programming

- Fact
  - State information that is always true
  - Consist only of a head
- Example

```
Fact class([Collection]).
Fact super([Collection], [Object]).
```

2. Logic Programming

- Rules
  - derive new information
  - Have a head and a body
  - Allow recursion
- Example:

```
Rule hierarchy(?root,?c) if
     super(?root,?c).
 Rule hierarchy(?root,?c) if
    super(?root,?sub),
    hierarchy(?sub,?c)
```

2. Logic Programming

- Multi-way: Rule describes real relation in the mathematical sense

- Example: the same hierarchy-predicate can be used in 4 ways:

```
Query hierarchy([Object],[Set])
Query hierarchy([Object], ?subs)
Query hierarchy(?supers, [Set])
Query hierarchy(?root, ?subs)
```

2. Logic Programming

- Terms
  - constant `[Collection]`
  - variable `?var` `?X`
  - compound `super([Set], sub([Object]))`
  - Terms `?x, foo([Set]), [Collection]`
- Clauses
  - Fact `Fact` *`simpleTerm`*
  - Rule `Fact` *`headTerm`* `if` *`terms`*
  - Query Query *terms*

2. Logic Programming

- Unification
  - "Enhanced pattern matching"
  - Input: 2 terms
  - Output: bindings for variables such that subsitution of these variables in both terms results in identical terms

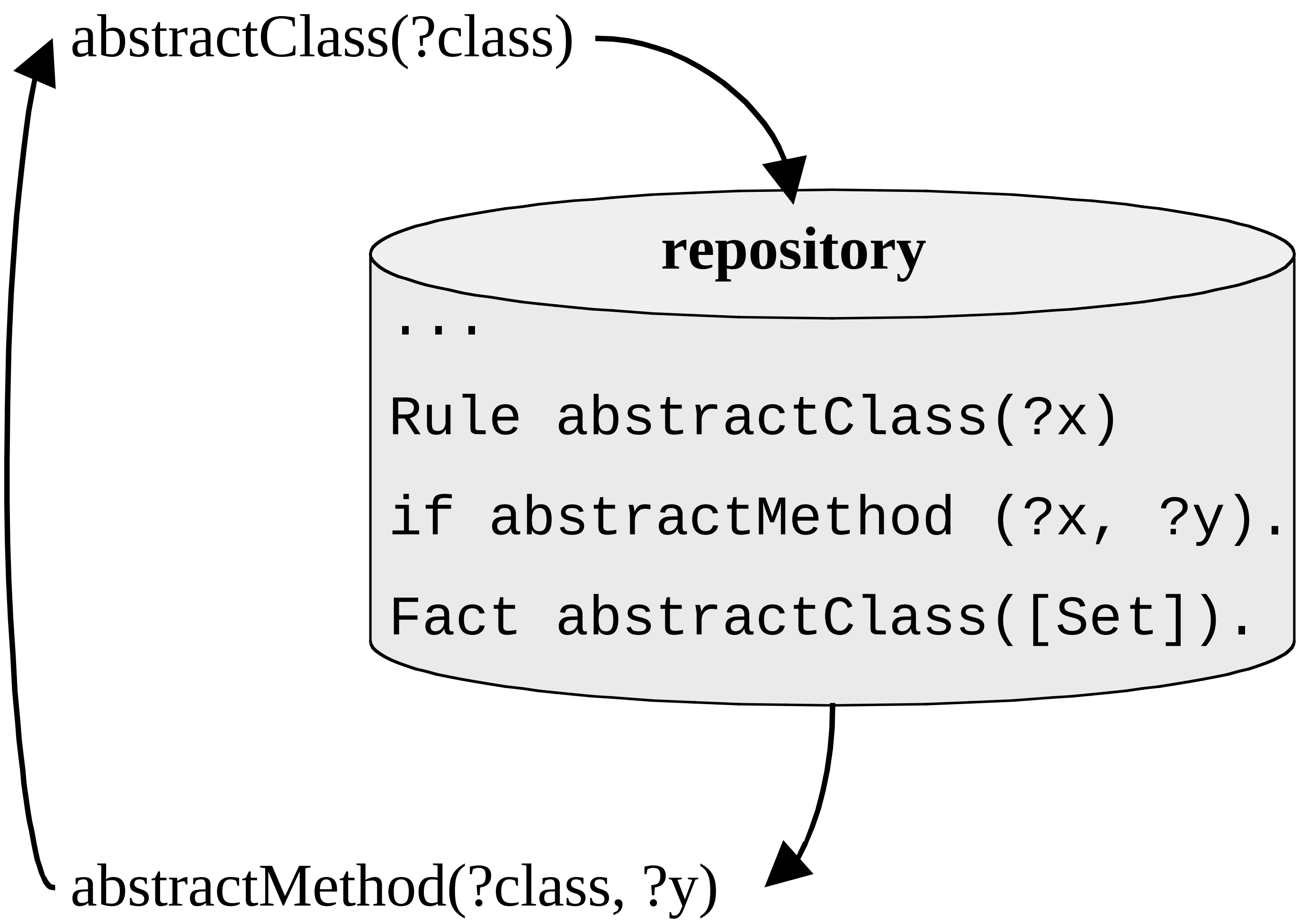
2. Logic Programming

- Unify: `class([Set])`
  `?x`
  Result: `{?x→ class([Set])}`

- Unify: `sel([Set], ?y, ?z)`
  `sel(?x,met([#add:]), ?t)`
  Result: `{?x→ [Set],?y→ met[#add:]),?z→ ?t}`

- Unify: `method(class([Set]), sel(?y))`
  `method(?x, met([#add:]))`
  Result: `fail`

2. Logic Programming



interprete

fetch

substitute

2. Logic Programming

- Declarative: Program = Data
- Positive:
  – real relations (no in- or output parameters)
  – powerfull: Turing equivalent
  – easy to learn and use
- Negative:
  – Sometimes slow execution, depending on the query to be solved

Map

1. Introduction
2. Logic Programming
3. **Implementation of SOUL**
4. Declarative Framework
5. Future Work
6. Conclusion
7. Demonstration (System - Tools)

3. SOUL: basics

- SOUL (Smalltalk Open Unification Language): reflective logic meta-language designed to reason about code/structure.
- Prolog-like, but
  - unification on general, user-definable elements
  - reflective
- ⇨ Smalltalk meta-language

3. SOUL: basics

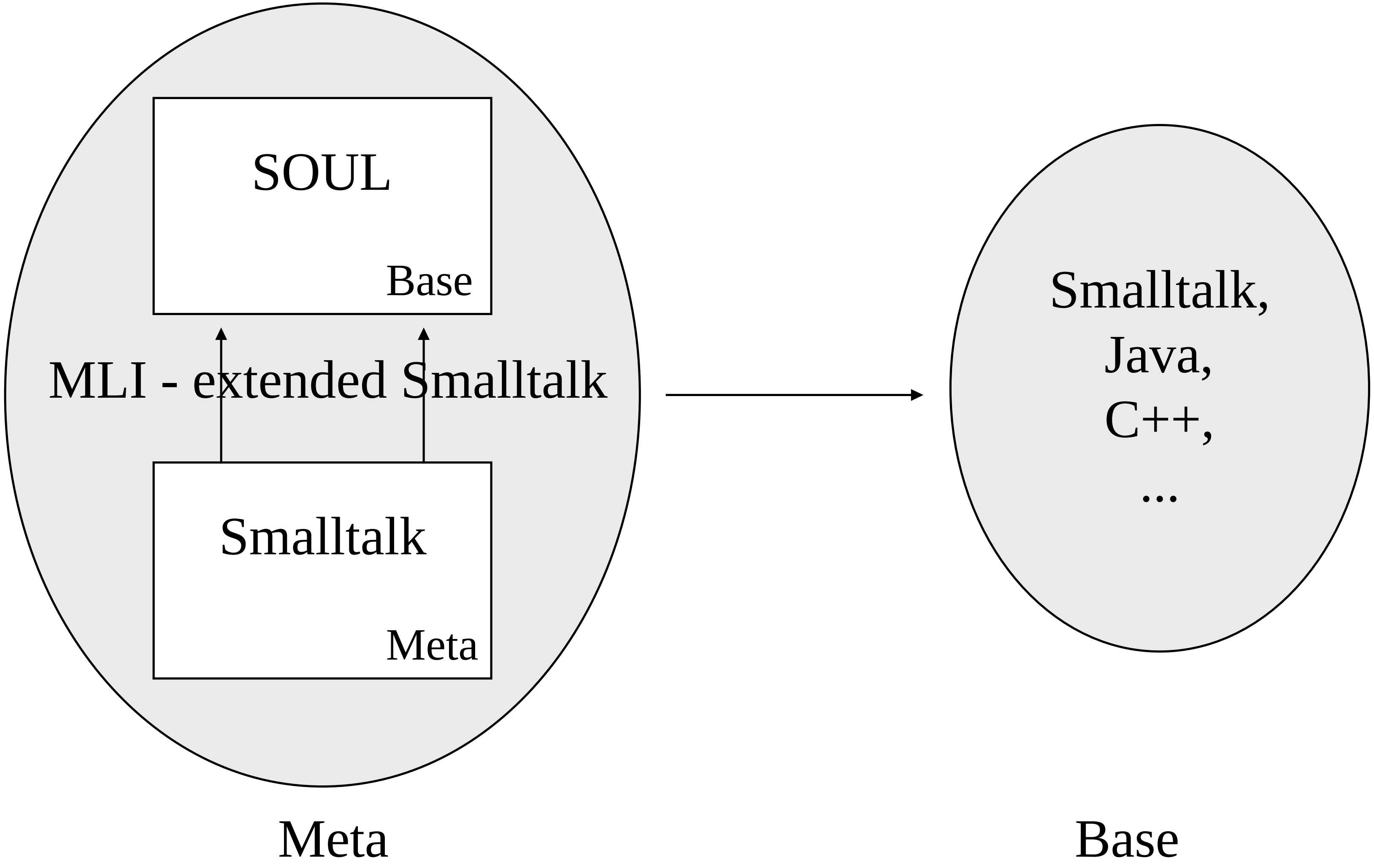
- ‘Smalltalk Term’: contains Smalltalk code extended with logic variables

> Smalltalk Term, checks ?c

```
Rule class(?c) if
    constant(?c),
    [Smalltalk includes: ?c name].

Rule class(?c) if
    variable(?c),
    generate(?c, [Smalltalk allClasses]).
```

3. SOUL: basics



3. SOUL: basics

SOUL represents object oriented systems by internal representation of *parsetrees*

⇨ reasoning about implementation on structural level

⇨ code and representation consistent

3. SOUL: implementation

- Smalltalk core
  - parser
  - basic logic elements (facts, rules, queries, constants, variables, Smalltalk terms, ...)
    - ⇨ unification strategy
  - Helper classes (bindings, repository, factory,…)
- SOUL extensions (reflective)
  - Lists, helper predicates, …
- SOUL Declarative Framework

3. SOUL: Smalltalk core

- Parser: made with the ParserCompiler
  – Straightforward
  – Problems with parsing Smalltalk Terms
  ⇨ Code between [and] is read as String !
  (see SOULParser>>scanUpTo:ignore:)
- As a result...
  – Syntax easy to change
  – Standard Browsers are used as editor
  – SOUL-code can be filed in/out

3. SOUL: Smalltalk Core

- Unification Strategy: Stream based, implemented with double dispatch
- Pro:
  - Clean & General
  - Calculates all solutions
  - Allows possibly infinite solutions (currently not used in SOUL)
- Contra
  - Difficult to have solutions one-by-one, or to implement some Prolog extensions like cut

3. SOUL: Smalltalk core

- Smalltalk Term: term containing Smalltalk extended with logic
- Is translated to block internally:

```
[?C includesSelector: ?M]
```

```
[:env | (env at: #C)
          includesSelector: (env at: #M)]
```

- Environment is filled in at runtime
- Fails if unbound variable

3. SOUL: Smalltalk core

- Generate predicate
  - generates bindings for a variable
  - 1st argument: variable to generate bindings for
  - 2nd argument: Smalltalk term describing what to generate
- Example:

```
generate(?c, [Smalltalk allClasses])
```

3. SOUL: Logic Layer

- Reflective part: extensions of SOUL written in SOUL
  - List predicates
  - System predicates (constant, variable, sound, equals, …)
- Use Smalltalk terms and Smalltalk meta-predicates (not discussed here)
- Implemented in class SOULLogicLayer

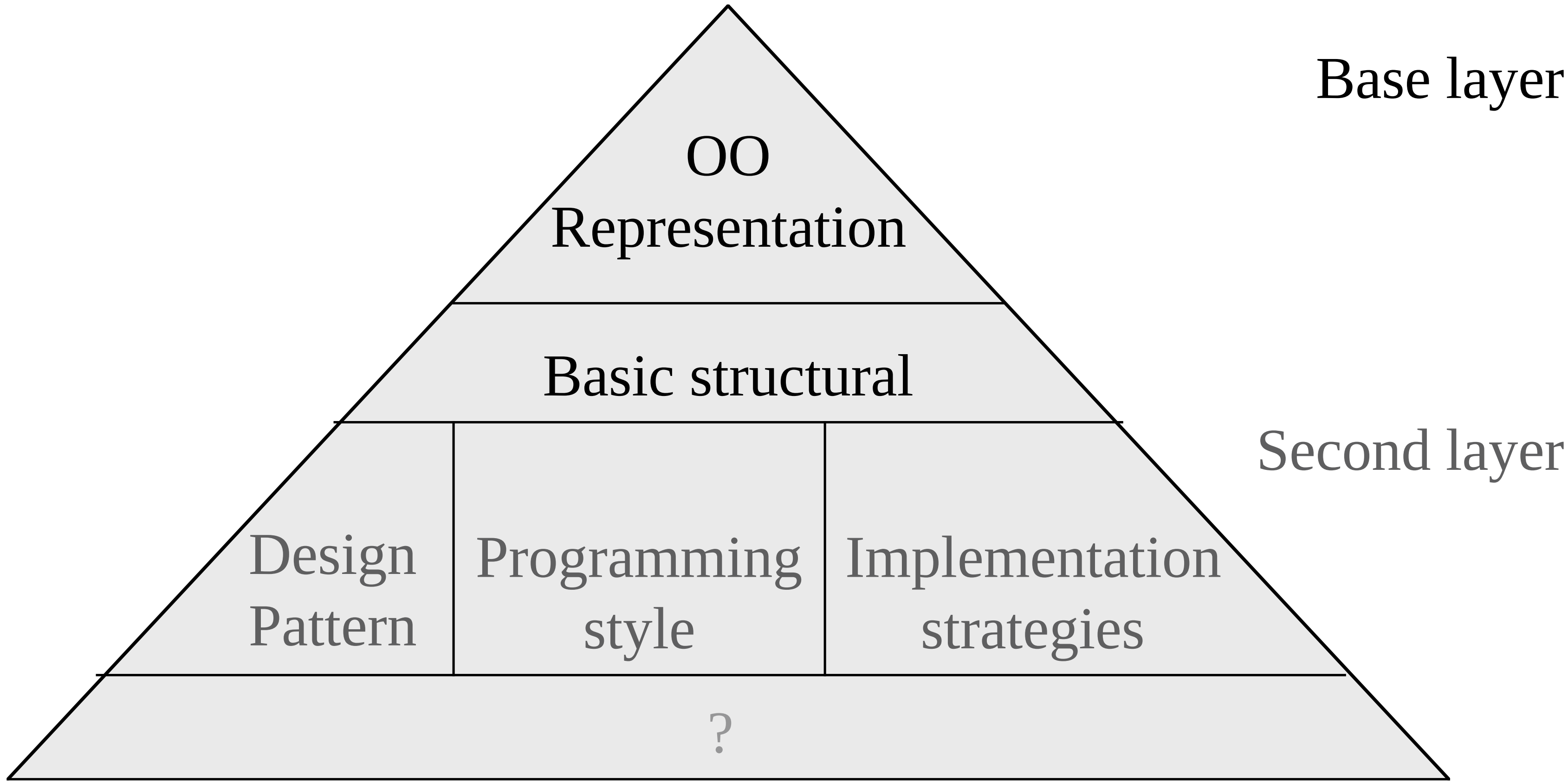
Map

1. Introduction
2. Logic Programming
3. Implementation of SOUL
4. **Declarative Framework**
5. Future Work
6. Conclusion
7. Demonstration (System - Tools)

4. Declarative Framework

- Groups facts and rules in different layers
- Will allow (< 2 weeks) overriding of rules
  - Real framework
  - General framework that allows plug-ins
- See the subclasses of SOULFramework

4. Declarative Framework



Map

1. Introduction
2. Logic Programming
3. Implementation of SOUL
4. Declarative Framework
5. **Future Work**
6. Conclusion
7. Demonstration (System - Tools)

5. Future Work

- Extend declarative framework
- Support other OO language (Java)
- Investigate MLI
- Generate code (structural find/replace)
- Build more Tools

Map

1. Introduction
2. Logic Programming
3. Implementation of SOUL
4. Declarative Framework
5. Future Work

6. Conclusion

7. Demonstration (System - Tools)

6. Conclusion

- Explicit link between design and implementation is needed
- Open, explicit, general system is needed to reason about the structure of OO systems
- Standalone Prolog is not enough
- We proposed SOUL, a reflective logic meta-language, and the declarative framework

Map

1. Introduction
2. Logic Programming
3. Implementation of SOUL
4. Declarative Framework
5. Future Work
6. Conclusion
7. **Demonstration (System - Tools)**

Coordinates

Roel Wuyts

Programming Technology Lab

Vrije Universiteit Brussel, Brussels, Belgium

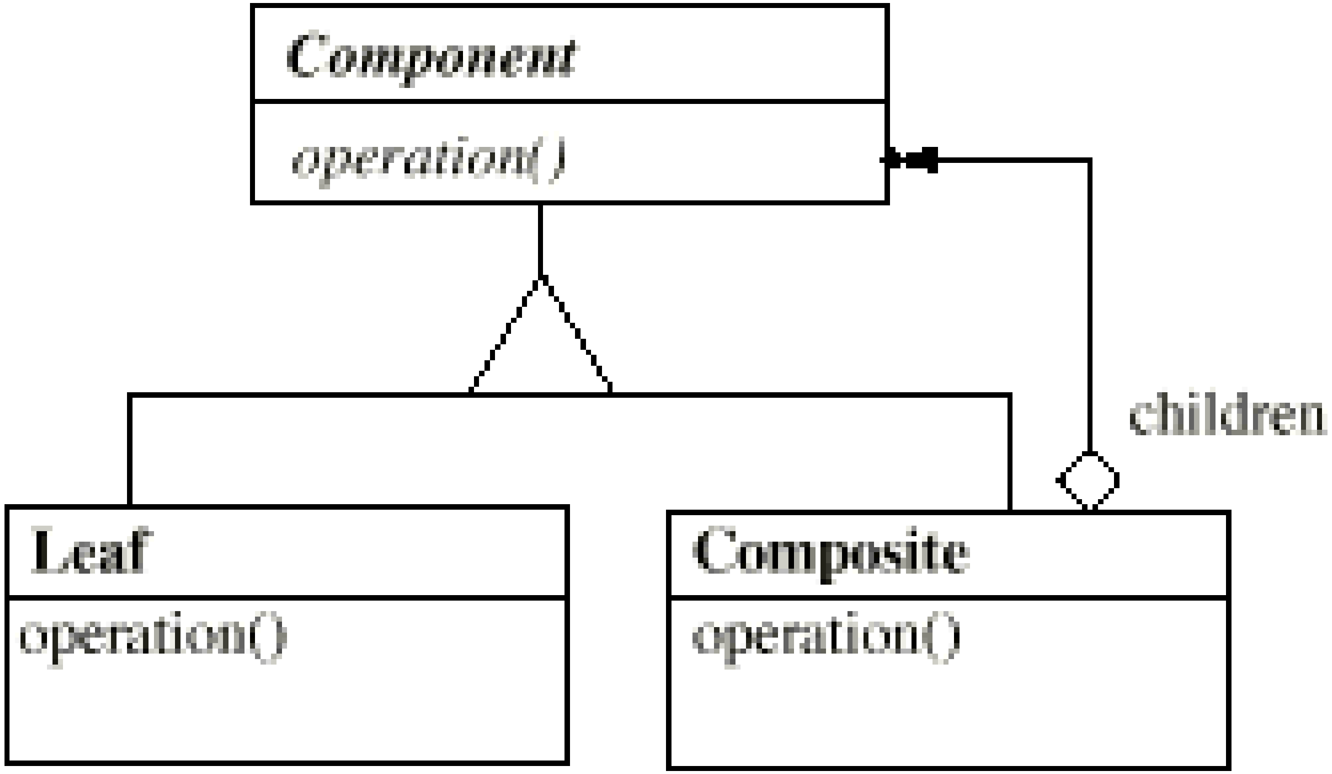
rwuyts@vub.ac.be

http://progwww.vub.ac.be/~rwuyts/

SOUL is free ! (VisualWorks 2.x, 3.x & Envy)

Composite Pattern Definition

Structure of Composite Design Pattern:



Composite Pattern Definition

```
Rule compositePattern(?comp,?composite,?op)
if
  compositeStructure(?comp,?composite),
  compositeAggregation(?comp,?composite,?op).

Rule compositeStructure(?comp,?composite)
if
  class(?comp),
  hierarchy(?comp,?composite).
```

Composite Pattern Definition

```
Rule
  compositeAggregation(?comp,?composite,?op)
if
  commonSelectors(?comp,?composite,?op),
  methodInClass(?composite,?m,?op),
  parseTree(?m,?tree),
  oneToManyStatement(?tree,?iv,?enumStat),
  isSend(?msg,?enumStat)
```

Composite Browser