Programming Languagages (Langages Evolués)

Roel Wuyts

Meta-Programming and Reflection

Meta-programming

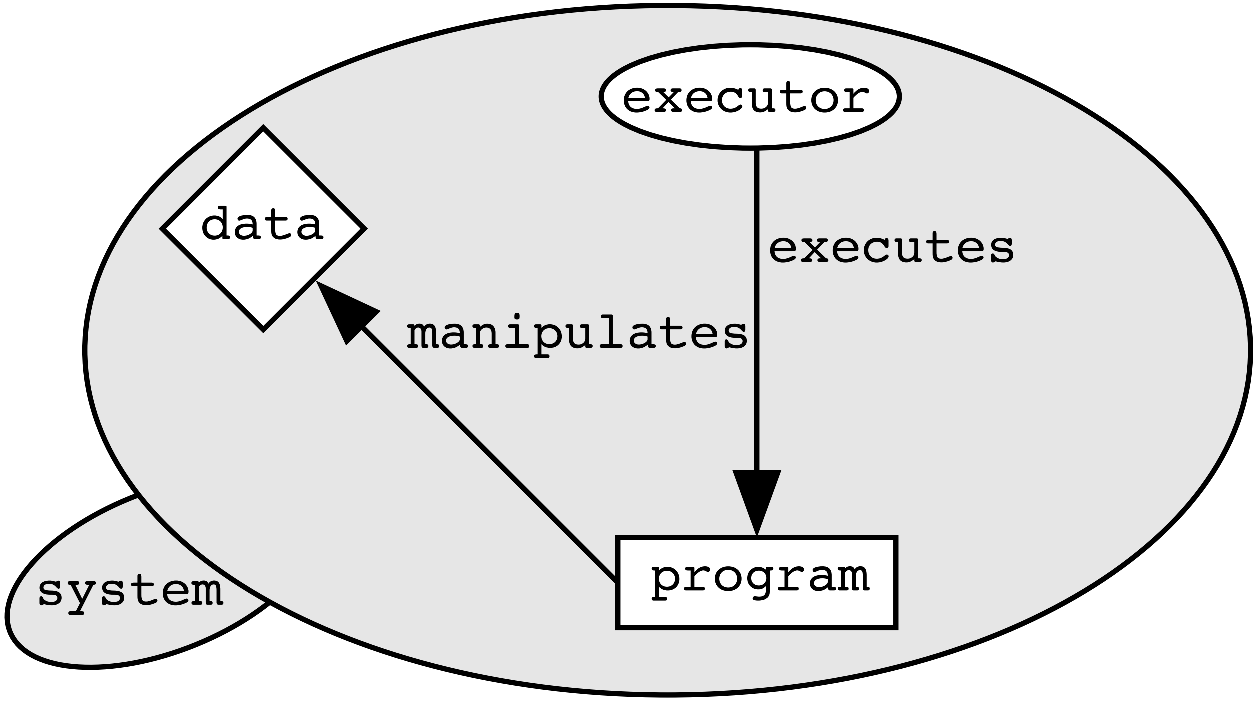
- In meta-programming one language (the meta language) reasons about a second language (the base language). Therefore both languages have to be integrated somehow. This is done by somehow representing the base language in the meta language [7].

Definitions

- Let's define:
  - computation system
  - programming language and program
  - reification & absorption
  - meta system
  - meta programming language and meta program
  - meta language and base language

Computational System

- A computational system is a system that reasons about and acts upon some part of the world, called the domain of that system.



Program and Language

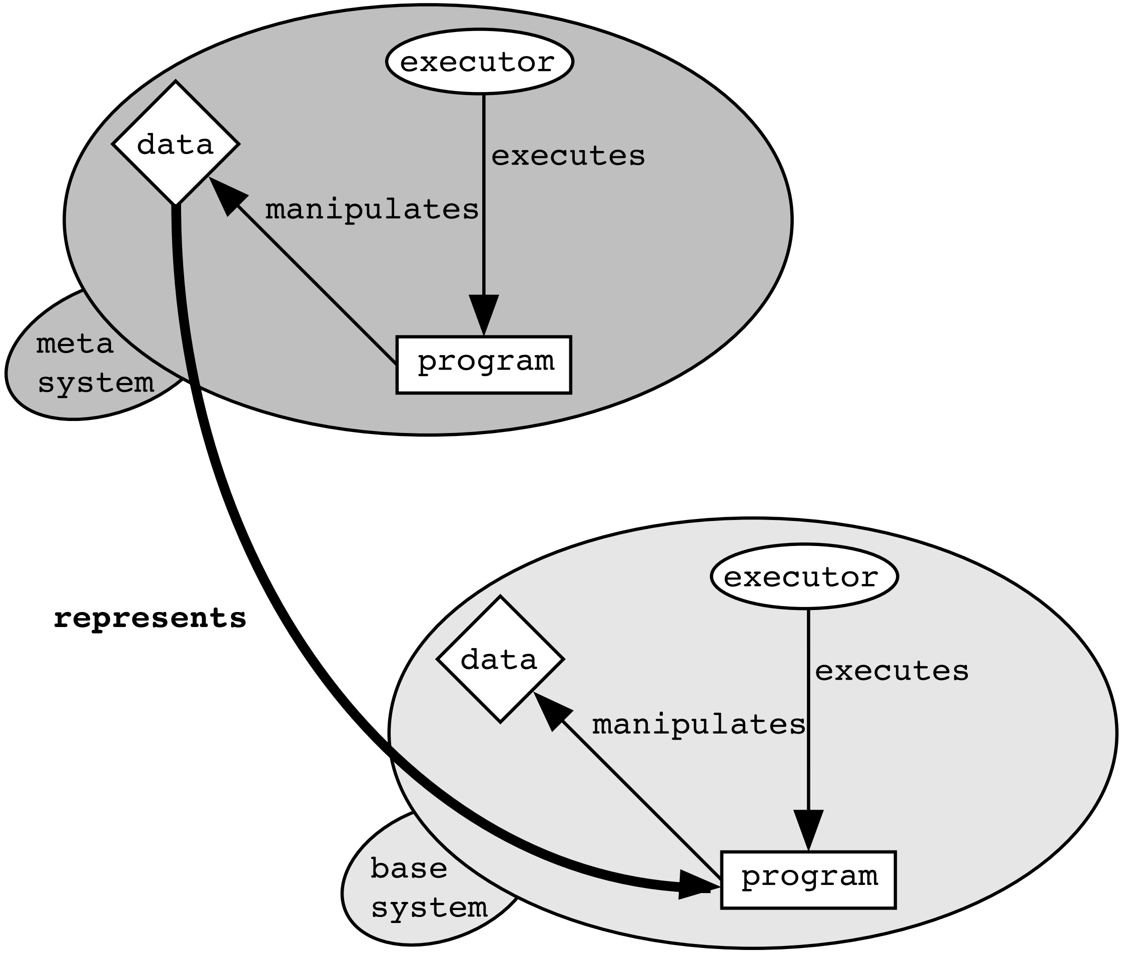
- A program is a formal, executable specification of a computational system.
- A programming language is a formalism that can be interpreted in an automatic manner in order to obtain the computational system specified by the program written in it.

Reification and absorption

- Every aspect of the internal workings of a computational system that has an explicit representation in the data of that system is said to be reified.
- Every aspect of the internal workings of a computational system that has no, or an implicit, representation in the data of that system is said to be absorbed.

Meta system

- A meta system is a system that has as its domain another computational system, called its base-system.



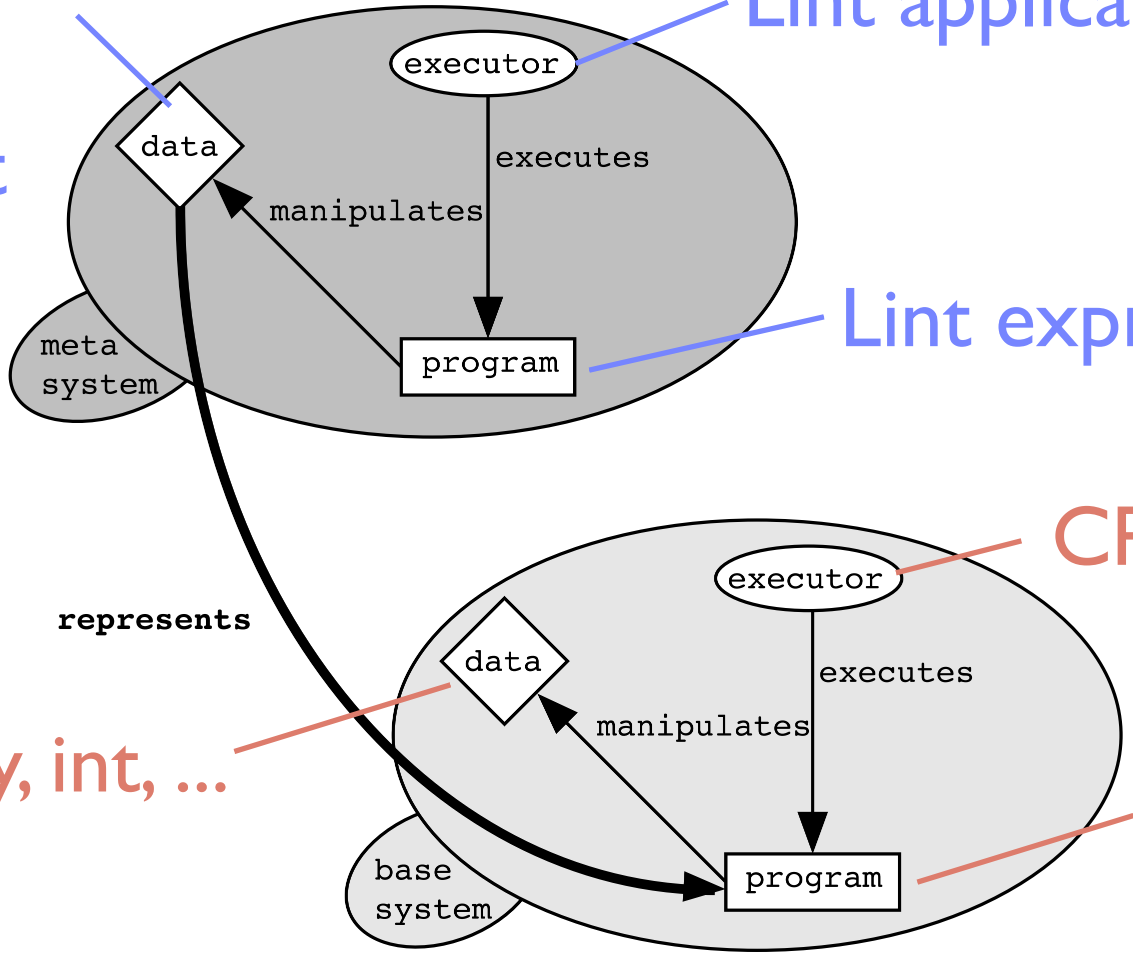
Meta program

- A meta program is the program specifying the meta system of a computational system.
    - The meta system does not directly manipulate its base-system; the meta system manipulates programs of the base-system.

Meta language and base language

- A meta language is a programming language specifically tuned for specifying meta programs.
- The base language for a given meta language is the programming language for which the meta language is specifically tuned.

Example: Lint as a meta system



C representation

Lint application

Lint

Lint expression

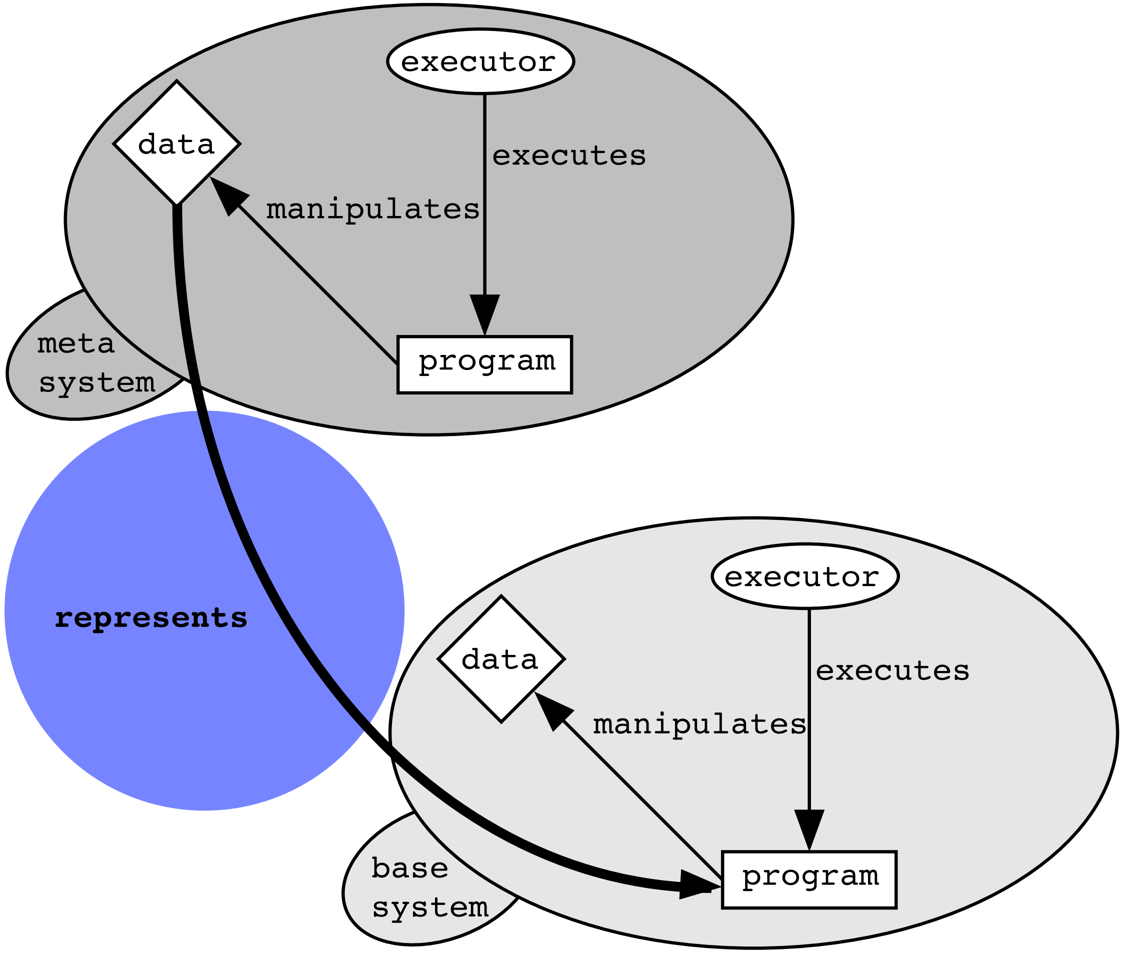
CPU

array, int, ...

C quicksort program

Representation

- Somehow the data of the meta system needs to represent programs of the base system



Example

- Lint originally represents the program under form of source
    - so kind of regular expressions over source code
- SmallLint: object representation
    - regular expressions over parse tree

Lint vs. SmallLint

- Choosing a representation is important!
  - as always...
- Lint
  - pro: easy: no work (source is there)
  - con: need to implement heuristic
- SmallLint
  - pro: real parse tree to use (scoping!)
  - con: need to have parser and tree walker

Reflection

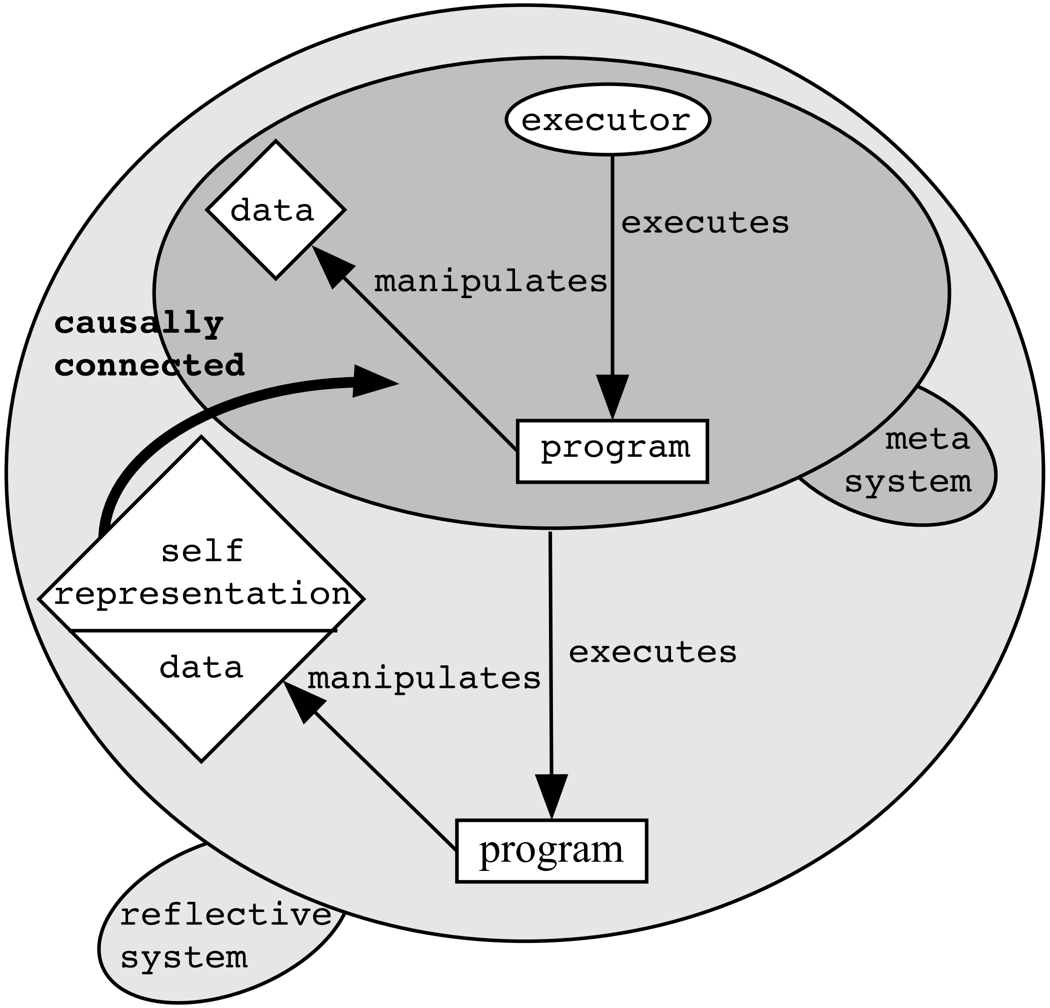
- Languages can be reflective
  - “reason about/manipulate” themselves
- Let’s see some definitions
  - causally connected
  - reflection
  - categorizing reflection
- Representation revisited

Causally connected

- A computational system is causally connected to *its* domain if the computational system is linked with its domain in such way that, if one of the two changes, this leads to an effect on the other.
- Example: robot arm
  - Domain: numbers indicating position of the arm.
  - Updating coordinates: robot arm moves.
  - Moving the robot arm: updates coordinates

Reflective system

- A reflective system is a causally connected meta system that has as base system itself



Introspective system

- Reflection as defined means that the system can inspect and change itself
  - Sometimes only inspection is possible
- An introspective system is a meta system that has as base-system itself.
  - So not necessarily causally connected

Categorizing Reflection

- *what* may be reflected
  - introspection
    - ex.: access representation, but do not modify
  - structural reflection
    - ex. : adding instance variables
  - computational (behavorial) reflection
    - ex.: modify method dispatching mechanism

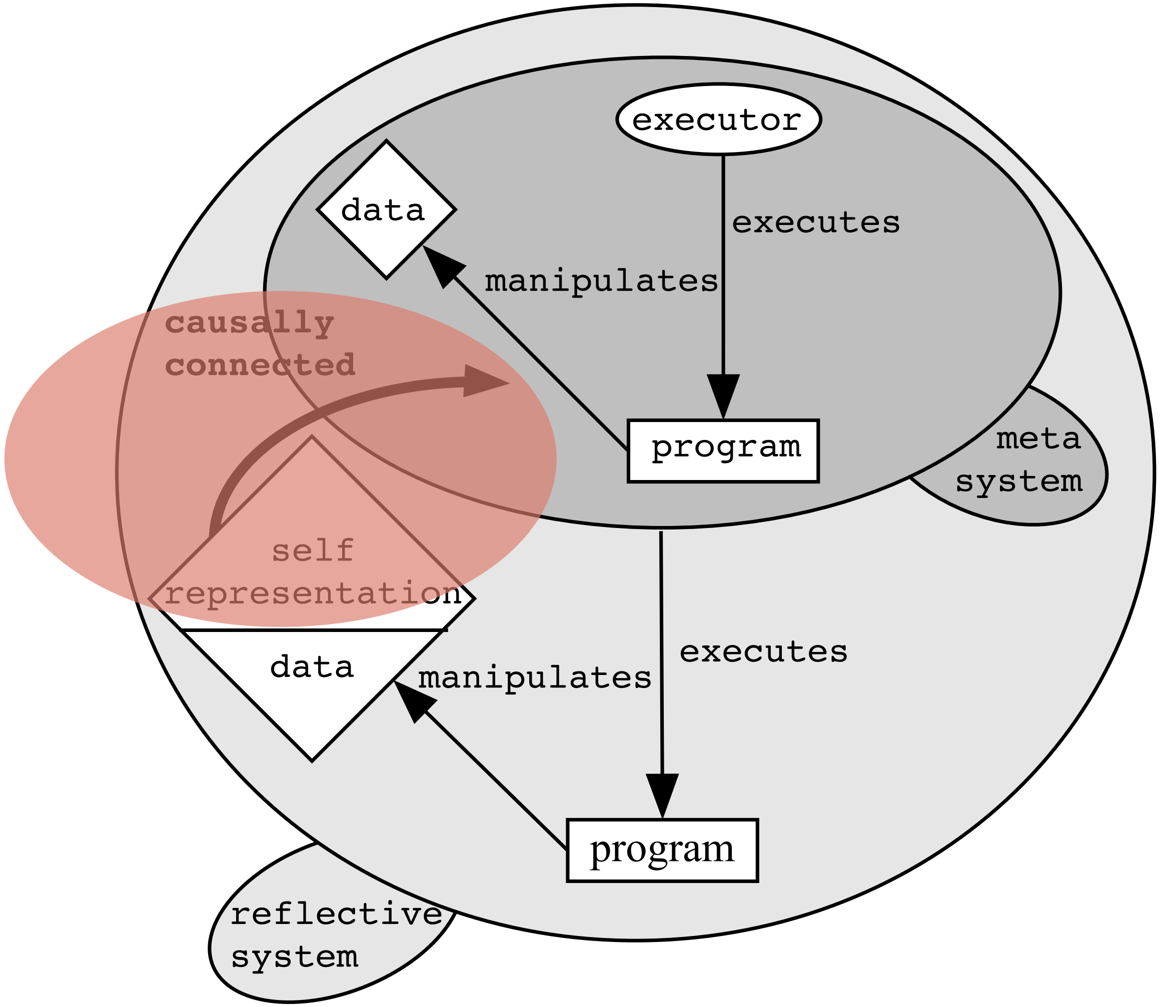
Categorizing Reflection (ctd)

- when does reflection takes place
  - Compile-time Reflection: customization takes place at compile-time.
    - Pro: runtime performance and the ability to adapt its own language (i.e., linguistic reflection).
  - Runtime Reflection: The system may be adapted at runtime, once it has been created and run.
    - Greater adaptability but performance penalties.
    - Typically used for computation reflection

Note

- With meta-programming, two languages are involved
  - one is meta for the other
- With reflection, one language is involved
  - the language is said to be *reflective* or not
  - or exhibits some reflective aspects...

Representation revisited



Self-representation Examples

- Smalltalk: objects and messages
  - Class, CompiledMethod, ParseNode
  - Context, Message
- Scheme: functors and lists
  - programs are represented as functors and lists
  - continuations
- Prolog: functors and lists

Smalltalk example

Smalltalk

- Some features are compiled away
  - e.g. send:withArgs:

Scheme example

Prolog example

Meta in OOP

- Meta composition problems
- show hierarchy in Smalltalk

Logic Meta Programming

- “Using a logic programming language to reason about an object-oriented base language”
  - Instance of Declarative Meta Programming
- Example (in Soul):

```
visitor(?Visitor,?Visited,?AcceptM,?VisitSelector) if
   classImplementsMethodNamed(?Visited,?AcceptM,?Meth),
   methodStatements(?Meth,

   <return(send(?V,?VisitSelector,?VisitArgs))>),
   member(variable([#self]),?VisitArgs),
   methodArguments(?Meth,?AccArgs),
   member(?V,?AccArgs),
   classImplements(?Visitor,?VisitSelector)
```

Soul

- Logic Programming Language in Smalltalk
  - ? for variables, < > for lists
- Example

```
append(< >,?Lst,?Lst).

append(<?F|?R>,?L1,<?F|?L2>) if
      append(?R,?L1,?L2).

if append(?L1,?L2,<1,2,3,4,5>)
```

Two Symbiotic Facilities

- Smalltalk wrapping

```
write(?text) if
      [Transcript show: (?text asString). true].

class(?C) if member(?C,[Smalltalk allClasses]).

smallerThan(?x,?y) if atom(?x), atom(?y), [?x < ?y].
```

Shorthands: `[4]` ⇔ `4`, `[#Symbol]` ⇔ `Symbol`

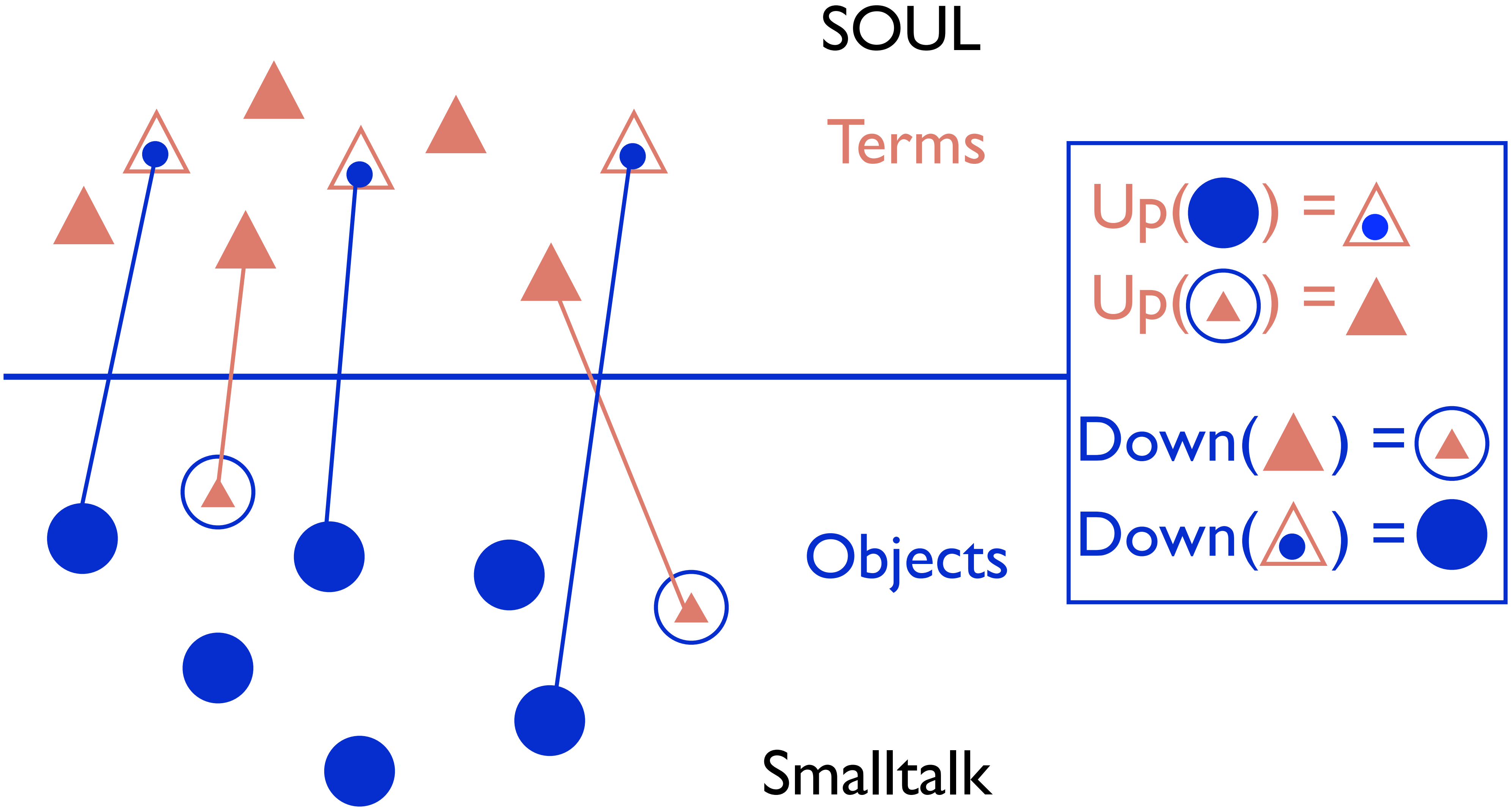
- Quasi-quoted code

```
{Array at: I put:?x}

{boolean ifTrue:[?trueC] ifFalse:[?falseC]}

{<html> ?htmlheader ?htmlbody </html>}
```

Soul Symbiosis



SOUL

Terms

Up(●) = ◬

Up(◉) = ▲

Down(▲) = ◉

Down(◬) = ●

Objects

Smalltalk

Stratification

- Two different levels are into play:
  - base level
  - meta level
- Stratification : meta-level facilities must be separated from base-level functionality
- Up and down is *the only way* to move between both
  - No implicit boundary crossing
  - Always clear which items belongs to what level

Why Symbiosis?

- Can do logic queries directly over objects
- For example:

```
class(?c) if member(?c, [Smalltalk allClasses]).

selector(?c, ?m) if
    class(?c),
    member(?m, [?c selectors]).


?- selector(?c1, ?m),
?- selector(?c2, ?m), not(?c1, ?c2)
```

Up/Down

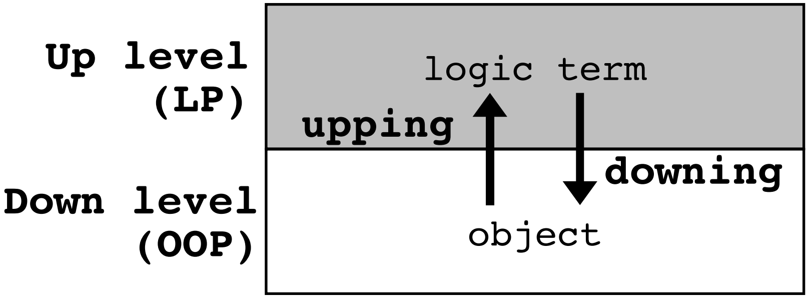
- How to realize symbiotic reflection?
  - ad-hoc: interpreter uses explicit tests:

    ```
    (argumentLanguage = X) {
    ...process as X...
    } else {
    ...process as Y...}
    ```

- clean with the *up/down* mechanism:
  - stratification clear in code
  - always evaluate expression in the same level
  - switch levels with up/down protocol

Up/Down rules

- T = logic terms, O = objects.



Up level (LP)

Down level (OOP)

logic term

upping

downing

object

- (1) $x \in T, x \notin O, up(down(x)) = x$

  For example in SOUL, $up(implementation(?c)) =?c$

- (2) $x \notin T, x \in O, up(x) = wrappedAsTerm(x)$

  For example in SOUL, $up(1) = [1] = wrappedAsTerm(1)$, where [1] is the logic representation of a term wrapping the integer 1.

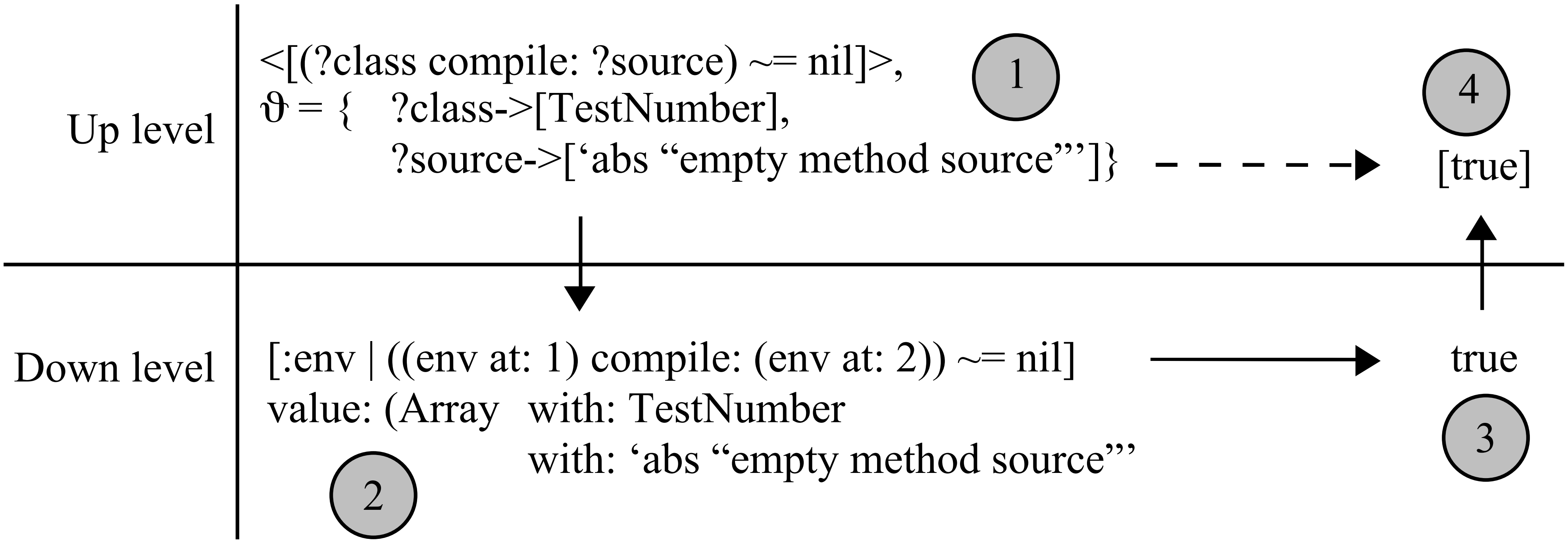
- (3) $x \in T, x \notin O, down(x) = implementationOf(x)$

  For example in SOUL, $down(?c) = aVariableTerm$, the smalltalk object representing the logic variable ?c.

- (4) $x \notin T, x \in O, down(up(x)) = x$

  For example in SOUL, $down([1]) = 1$, where [1] is the logic representation of a term wrapping the integer 1.

Up/Down in action



Up level

<[(?class compile: ?source) ~= nil]>,
ϑ = { ?class->[TestNumber],
?source->[‘abs “empty method source”’]}

[true]

Down level

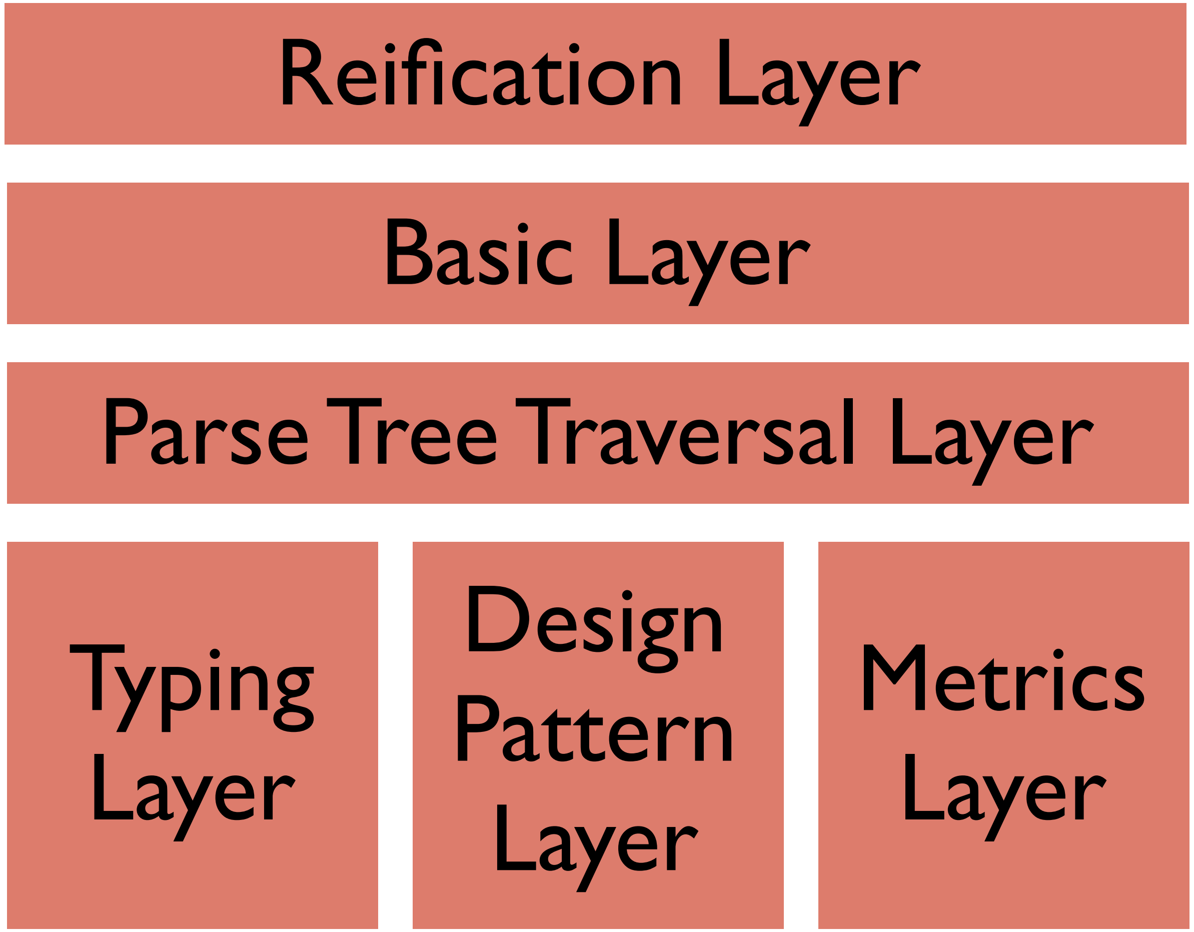
[:env | ((env at: 1) compile: (env at: 2)) ~= nil]
value: (Array with: TestNumber
with: ‘abs “empty method source”’

true

LiCoR

- *Li*brary for *Co*de *R*easoning
- Reifies OO code (Smalltalk)
- Provides
  - basic predicates
  - typing predicates, code convention predicates, design pattern predicates, UML predicates.
- Now also for Java (well, nearly...)

LiCoR Structure



Reification Layer

Basic Layer

Parse Tree Traversal Layer

Typing Layer

Design Pattern Layer

Metrics Layer

Reification

- LiCoR reifies structural information
- 4 concepts are reified
  - class, method, instance variable, inheritance
- Most of the rules use these
  - ... but some of them directly talk to Smalltalk
  - typically for performance reasons

Example: LiCoR Typing

- Smalltalk is dynamically typed.
- Use Soul to reconstruct types of instance variables of classes.
- Simplest version:
    - find all methods sent to the instance variable, and find all classes that understand those methods
- Uses the parse tree traversal predicates to do this

Example: LiCoR Typing (ctd)

```
• classWithInstvarOfTypeBySends(?Class, ?instvarName, ?Type) if
    instanceVariableInClass(?iVar, ?Class),
    findall(  ?messageSent,
              and( methodInClass(?Method, ?Class),
                   or(  methodWithSend(?Method, variable(? iVar),
                                                            ?messageSent, ?),
                        methodWithSend(?Method,send(variable(self),
                                                ?iVar,?), ?messageSent, ?))
                   ),
              ?messagesSent),
    noDups(?messagesSent, ?interface),
    classesUnderstanding(?sendTypes, ?interface),
    rootclassesOfClasses(?sendTypeRoots, ?sendTypes),
    member(?Type, ?sendTypeRoots)
```

Example: LiCoR Typing (ctd)

- More advanced versions
  - look at assignments, factory methods, ...
  - use “type snooping”

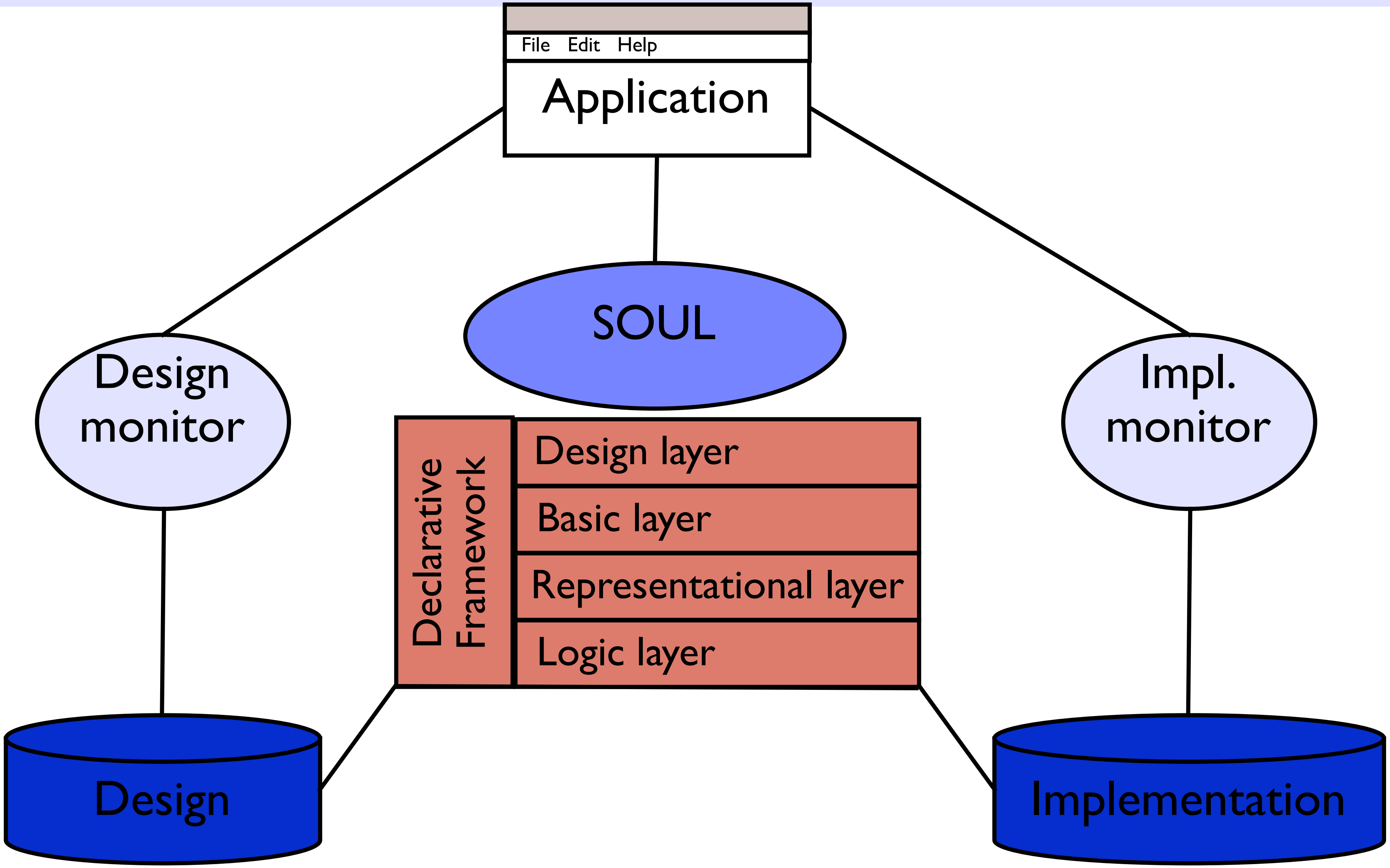
```
classWithInstvarOfTypeBySnooping(?class, ?instvarName, ?T)
if
     instanceVariableInClass(?instvarName, ?class),
     equals(?index,
         [?class instVarIndexFor: ?instvarName asString]),
     member(?T, [(?class allInstances collect: [:inst |
               (inst instVarAt: ?index) class]) asSet])
```

- Cons: heuristic based
- Pro: handles meta-programming, reflection, blocks,...

Co-Evolution

- Co-evolution of design and implementation:both design and implementation are subject to evolution, and they influence each other continuously.
- How to support co-evolution?
  - Synchronization Framework
  - Example of a LiCoR application
    - and of Soul

Synchronization Framework



Wrap-up

- Meta programming is writing a program that works on another program
- Reflection: reason about/manipulateyourself
  - lots of possibilities: *what* vs. *when*
- Application: Soul: logic language in Smalltalk
  - Symbiosis with Smalltalk
  - interpreter written in Smalltalk
  - uses reflective facilities of Smalltalk

References

- http://www.ulb.ac.be/di/rwuyts/INFO020_2003/
- Wuyts, Roel, *A Logic Meta-Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation*, Ph.D. Thesis, Vrije Universiteit Brussel, 2001.
- Maes, Patricia, *Computational Reflection*, Ph.D. Thesis, Vrije Universiteit Brussel, 87.