

Experiments with Pro-Active Declarative Meta-Programming

Verónica Uquillas Gómez

Andy Kellens

Kris Gybels

Theo D'Hondt



International Workshop on Smalltalk Technologies
31st of August 2009, Brest - France
ESUG 2009

Software Languages Lab
Vrije Universiteit Brussel
Belgium

Problem

```

# Mammal and Dog classes in Ruby
class Mammal
  def initialize name
    @name = name.capitalize
  end
  def breathe
    puts "inhale and exhale"
  end
  def display
    puts "#{@name}"
  end
end

class Dog < Mammal
  def initialize(name, breed)
    super
    @breed = breed
  end
  def speak
    puts 'ruff! ruff!'
  end
  def display
    puts "#{@name} is a/a #{@breed}"
  end
end

# Instantiation
d = Dog.new('lassie', 'alsatian')

# Invocation of display method
d.display
    
```

```

# Ruby implementation
class Employee
  def initialize(name, salary=0)
    @name = name
    @salary = salary
  end
  def giveRaise(percent)
    @salary += (@salary * percent / 100)
  end
  def work
    puts "#{@name} does work"
  end
end

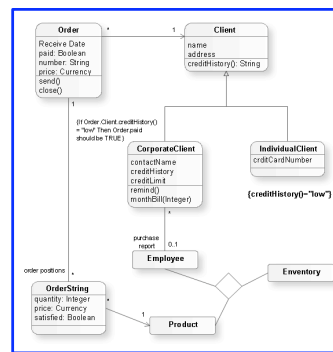
class Chef < Employee
  def initialize(name)
    super(name, 3500)
  end
  def work
    puts "#{@name} makes food"
  end
end
    
```

```

class PizzaRobot < Chef
  include Robot
  def initialize(name, cost, lifetime)
    super(name)
    newRobot(cost, lifetime)
    salary = 0
  end
  def work
    puts "#{@name} makes pizza"
  end
  def show
    puts info
  end
end
    
```

```

  def info
    return "Robot: cost=#{@cost}(e),
           lifetime=#{@lifetime}(y)"
  end
end
    
```



```

public class FlightMethod {
    public static void main(String[] args) {
        Client client = null;
        Order order = null;
        OrderShipping os = null;

        try {
            client = new Client("John");
            order = new Order(1000, "1000");
            os = new OrderShipping(1000, "1000");
            client.addOrder(order);
            client.addOrderShipping(os);
            client.showOrders();
            client.showOrderShipping();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
    
```

Co-evolution

SOUL

- ❑ Developer-driven ( pro-active) 
- ❑ Snapshot-based reasoning ( time) 

SOUL: Smalltalk Open Unification Language

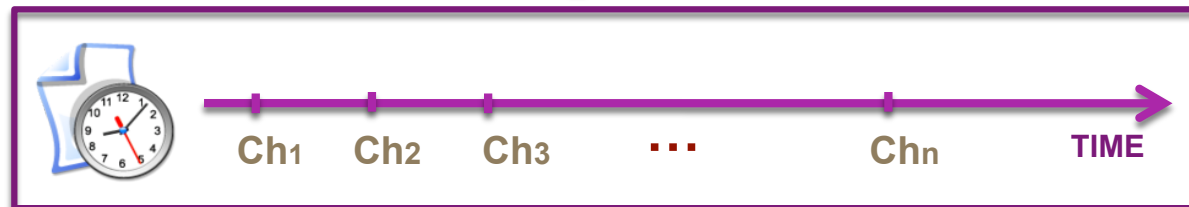
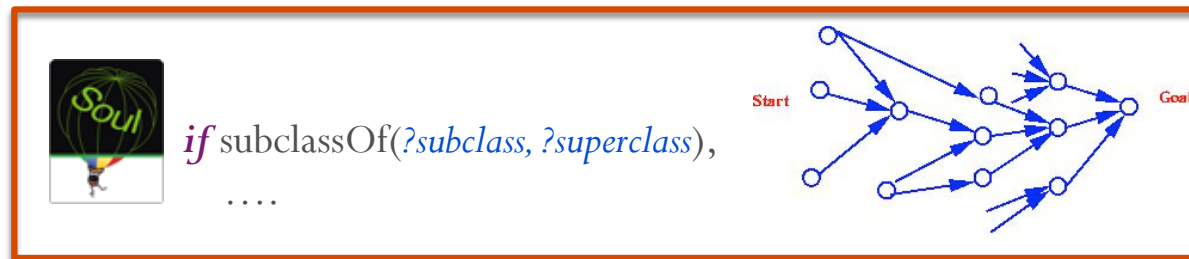
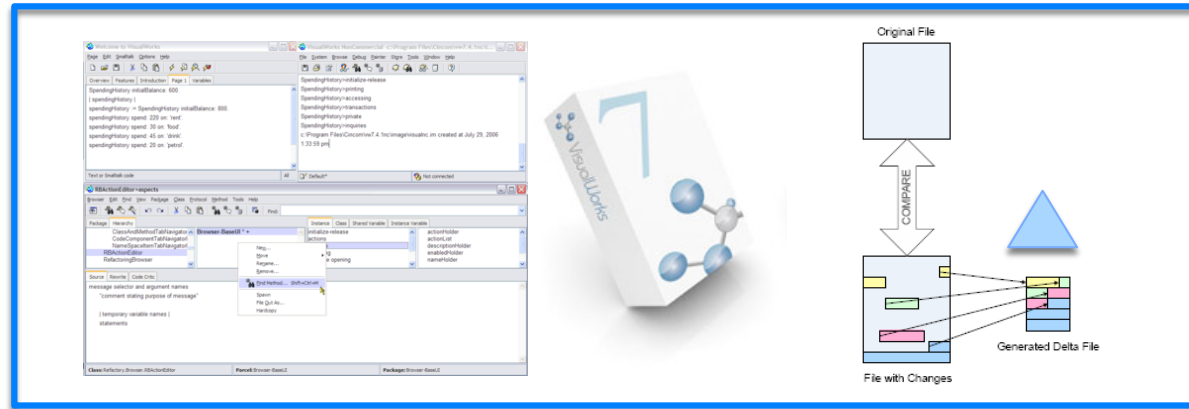


- Query language
- Prolog-like
- Smalltalk, Java, C(++), Cobol

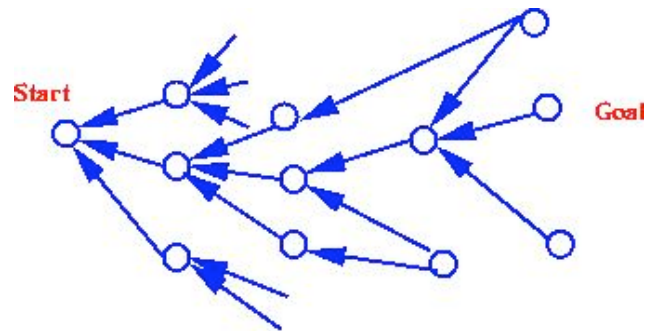
Declarative Framework	Design layer (programming conventions, design patterns)
	Basic layer (parse tree traversal, typing, flattening, code generation, accessing, auxiliary)
	Representational layer (base predicates)
	Logic layer (arithmetic, list handling, type checking, repository handling, pattern matching)

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

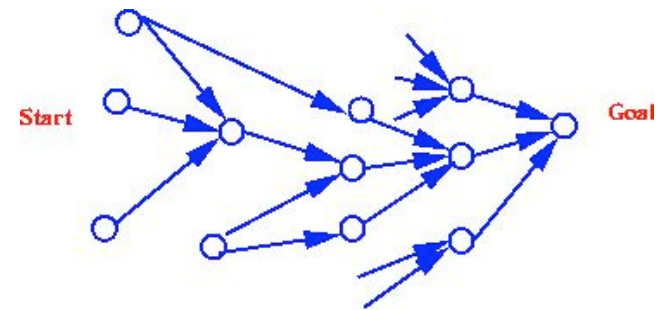
PARACHUT: Programming And Reasoning About CHanges Using Time



Backward Chaining vs. Forward Chaining



Goal-driven



Data-driven

- More possible start states or goal states?
- What kind of events triggers problem-solving?

Example: hash – equal rule

“re-implementing the = method implies re-implementing the hash method”

classNeedsHash(*?class*) *if*
methodInClass(*?class*, =),
not(methodInClass(*?class*, hash))

classNeedsEqual(*?class*) *if*
methodInClass(*?class*, hash),
not(methodInClass(*?class*, =))

- class
- instanceVariableInClass
- methodInClass
- methodCallsMethod
- classInPackage
- subclassOf
- classInHierarchyOf

PARACHUT: defining the declarative queries

The screenshot shows the SOUL Clause Browser interface. The window title is "SOUL Clause Browser". The menu bar includes "Tools", "Special", and "Help". The "Lookup:" dropdown is set to "default".

The left pane shows a tree view of the SOUL system structure:

- LogicStorage
 - QuotedParseLayer
 - LogicPrimitives
 - DeprecatedPredicatesL
 - TestQueriesLayer
 - MetalInterpretation
 - SmalltalkReasoning
 - SoulReteReasoning

The middle pane displays a list of clauses:

- actions
- entities
- evolutionMatrix
- hash_equal
- problemIndicators
- relationships

The right pane displays a list of clauses:

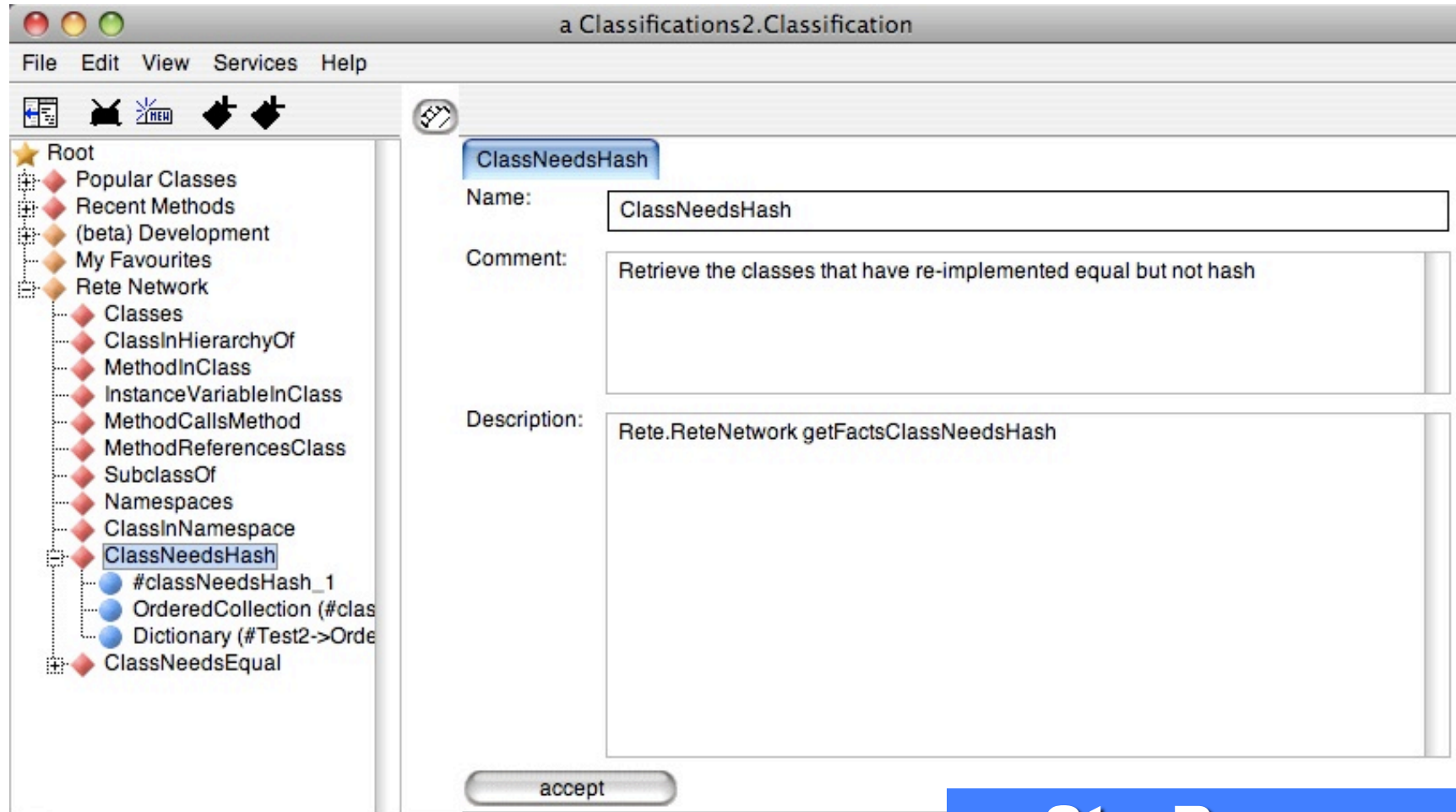
- allConsecutiveSizes/3
- allConsecutiveSizes/4
- canBecomeSupernova/1
- canBecomeWhiteDwarf/1
- consecutiveSizes/3
- deadClass/1
- decreasedSize/1
- increasedSize/1
- mayBeldle/1
- mayBeldle/2
- numberOfMethods/2
- pulsar/1
- pulsar/4
- redGiant/1
- supernova/1
- supernova/4
- whiteDwarf/1
- whiteDwarf/4

The bottom pane displays the definition of the `supernova` clause:

```
supernova(?class,?len3,?len2,?len1) if
  increasedSize(?class),
  mostrecent(canBecomeSupernova(?class)),
  allConsecutiveSizes(?class,?len3,?len2,?len1),
  escape(?val,[ ?len3 = (?len1 * 3)
             and:[ ?len2 = (?len1 * 2) ] ])
```

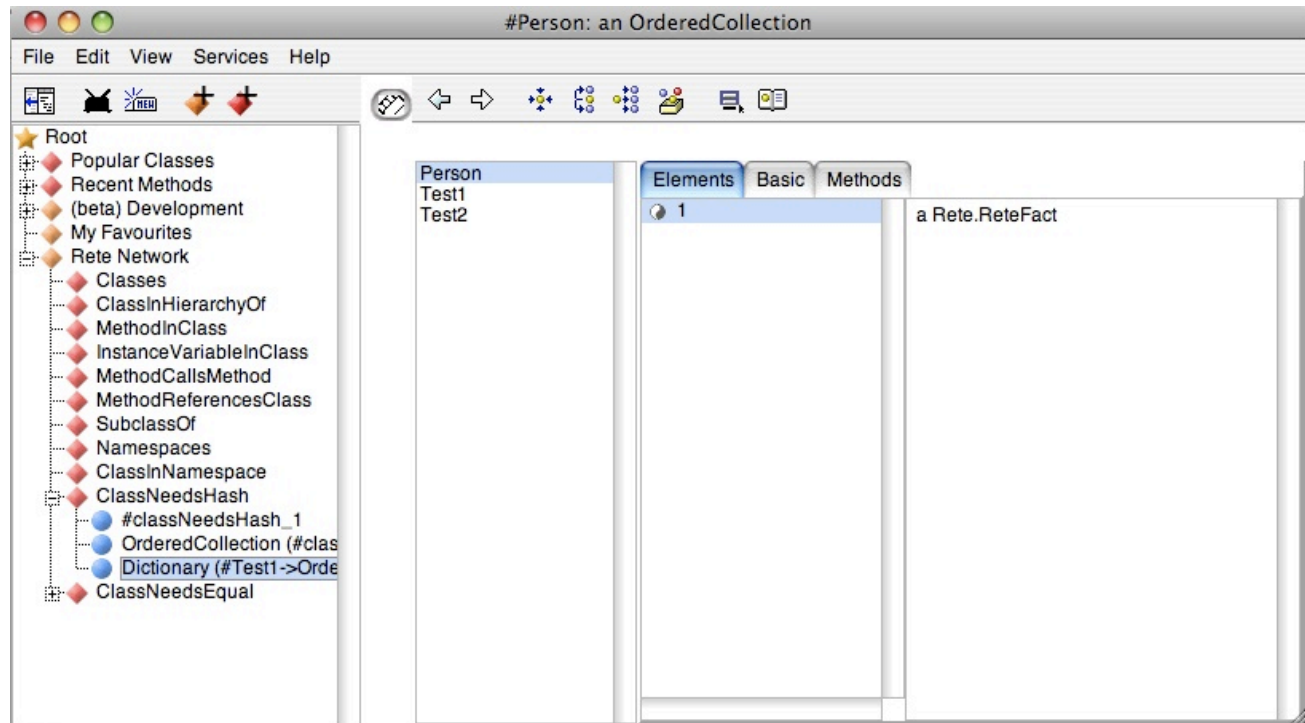
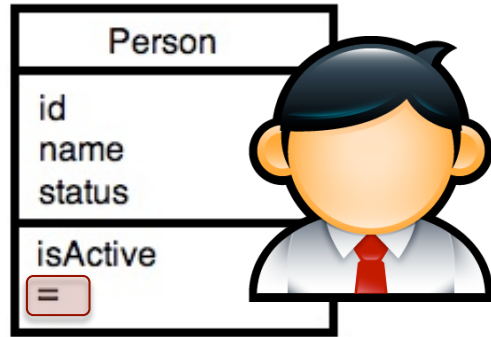
SOUL Clause Browser

PARACHUT: outputs to developers (I)



StarBrowser

PARACHUT: outputs to developers (II)

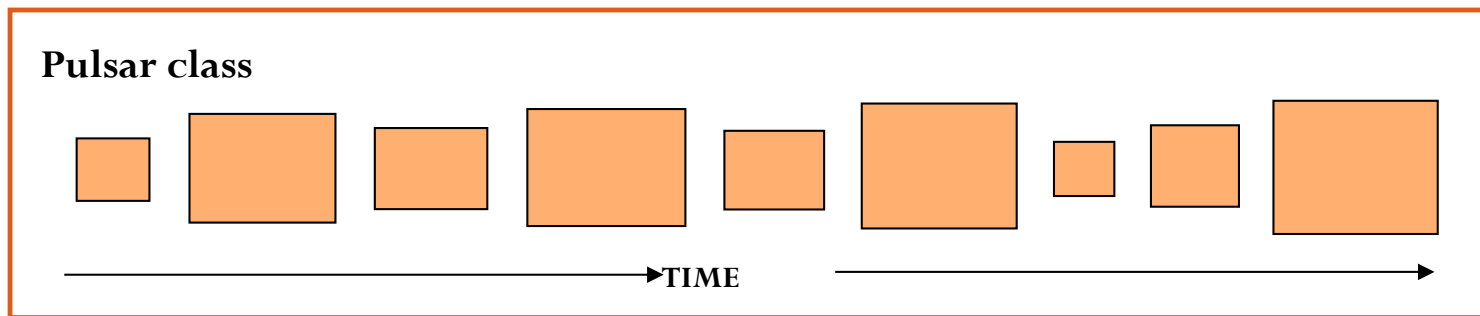


Other Examples



- Pulsar class pattern
- Possible refactoring opportunities

Pulsar Class Pattern



pulsar(*?class*) *if*

pulsar(*?class*, *?len3*, *?len2*, *?len1*)

?len3 at t3

?len2 at t2

?len1 at t1

t3 > t2 > t1

pulsar(*?class*, *?len3*, *?len2*, *?len1*) *if*

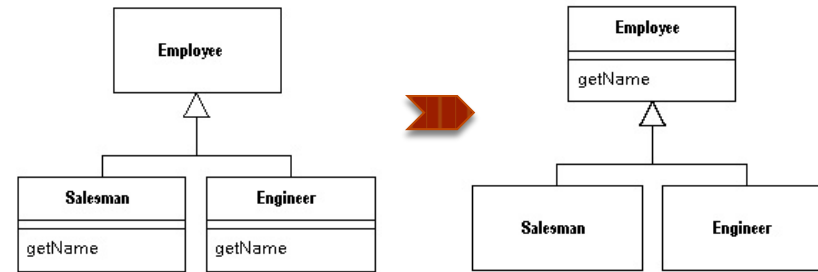
allConsecutiveSizes(*?class*, *?len3*, *?len2*, *?len1*),

escape(*?val*, [(*?len1* * 2) = *?len2*

and:[*?len1* = *?len3*]])

e.g. 10 – 20 – 10

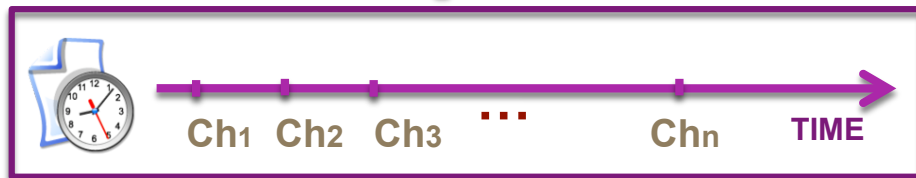
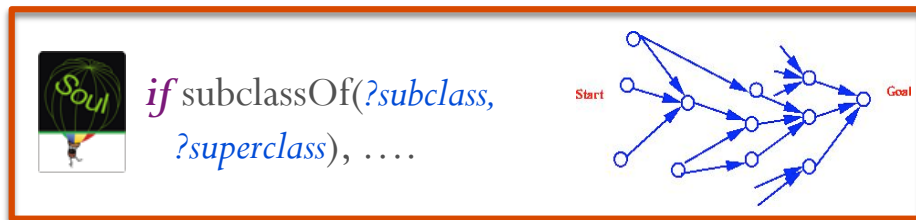
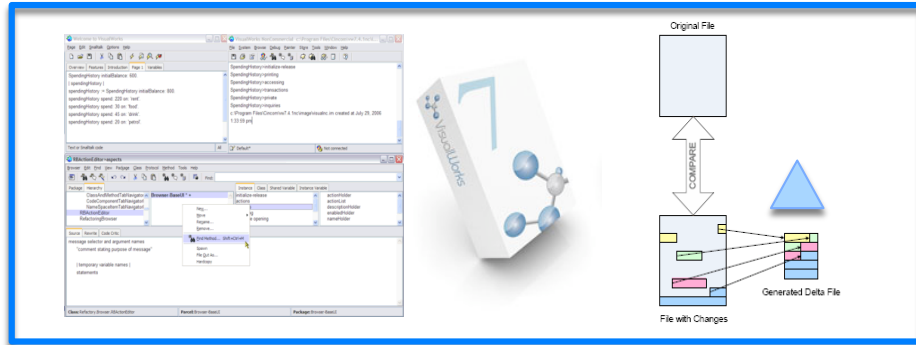
Detecting possible refactoring opportunities



`duplicateSelector(?classA, ?classB, ?method)` *if*
`addedMethodInClass(?classB, ?method),`
`mostrecent(addedMethodInClass(?classA, ?method))`

`refactoringOpportunity(?superclass, ?method)` *if*
`duplicatedSelector(?classA, ?classB, ?method),`
`subclassOf(?classA, ?superclass),`
`subclassOf(?classB, ?superclass),`
`not(addedMethodInClass(?superclass, ?method))`

Conclusions



PARACHUT



Verónica Uquillas Gómez
vuquilla@vub.ac.be