Computational Reflection and Context-Oriented Programming

Kim Mens
Sebastián González

Invited lecture at
Advanced Development Techniques
PhD School in Computer Science
University of Milan

3–9 July 2012
1. Reflection
   a. Principles
   b. In Smalltalk

2. Context-Oriented Programming
   a. Infrastructure
   b. Adaptation
   c. Composition
   d. Resolution

day 1
day 2
day 3
day 4
<table>
<thead>
<tr>
<th>Day</th>
<th>Theory</th>
<th>Practice</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Tuesday</strong>&lt;br&gt;July 3</td>
<td>Course introduction&lt;br&gt;Basics of reflection</td>
<td>Introduction to Smalltalk</td>
</tr>
<tr>
<td><strong>Wednesday</strong>&lt;br&gt;July 4</td>
<td>Reflection in Smalltalk</td>
<td></td>
</tr>
<tr>
<td><strong>Thursday</strong>&lt;br&gt;July 5</td>
<td>Context-oriented programming&lt;br&gt;Why? What? How?</td>
<td>Infrastructure&lt;br&gt;Adaptation</td>
</tr>
<tr>
<td><strong>Monday</strong>&lt;br&gt;July 9</td>
<td></td>
<td>Composition&lt;br&gt;Resolution</td>
</tr>
</tbody>
</table>
Reflection in Smalltalk

* partly based on slides by Roel Wuyts
Reflection (revisited)

- Reflection is the ability of a program to examine and control its own implementation.

- Computational reflection is good for:
  - extending a language
  - building programming environments
  - advanced software development tools
  - ...

- Smalltalk offers some interesting reflective capabilities:
  - Object, classes and metaclasses
Reflection in Smalltalk: classes

Classes in Smalltalk are first-class values

- **Point of reflection** to reason about objects
- **Group objects** with identical behaviour
  - Method `#allInstances` returns all instances of a given class
    - Transcript class allInstances
  - "Factory" for creating instances
    - Contain the instance creation methods (constructors)
  - **Describes the implementation of a set of objects**
    - Contain methods for reasoning about that implementation
      - `#compiledMethodAt:`
      - `#instVarNames`
      - `#compile:classified:notifying:`
Reflection in Smalltalk: classes

- Retrieve all instances of a class

  ```
  Transcript class
  ➡️ ThreadSafeTranscript
  
  (ThreadSafeTranscript allInstances at: 1) == Transcript
  ➡️ true
  ```

- In the above case:
  - Transcript is one of the instances of class ThreadSafeTranscript
Reflection in Smalltalk: classes

- Asking an instance to what class it belongs

(2@3) class ➞ Point
Reflection in Smalltalk: classes

- Inspecting the implementation details of a class:
  - names of instance variables defined by class
    ```
    OrderedCollection instVarNames
    ➞ #('array' 'firstIndex' 'lastIndex')
    ```
  - names of methods implemented by class
    ```
    OrderedCollection selectors
    ➞ #(#copyFrom:to: #with:collect: #grow #addAll: ...)
    ```
  - method with a given selector
    ```
    OrderedCollection compiledMethodAt: #size
    ➞ answers the compiled method associated with the message selector #size in the method dictionary of the class
    ```
  - source code of method with a given selector
    ```
    (OrderedCollection compiledMethodAt: #size) getSource
    ➞ gets the source code of the method named #size on class
    ```
Reflection in Smalltalk: classes

- Creating a new instance of a class

  ```smalltalk
  OrderedCollection new: 10
  ```

- Creates a new instance of `OrderedCollection`, with size 10
Reflection in Smalltalk: classes

- Dynamically adding a new method to some class:
  ```plaintext
  <class> compile: <text> classified: <category>
  ```

- Try this first:
  ```plaintext
  #(1 2) asOrderedCollection testMethod
  ➡ results in “method not understood” error
  ```

- Example:
  ```plaintext
  OrderedCollection
  compile:
    'testMethod
    Transcript show: ''it worked...''
  classified: 'testing'
  ```
Reflection in Smalltalk: classes

- Try it:
  
  `#(1 2) asOrderedCollection`
  `testMethod`
  ➡ prints “it worked...” on the transcript

- To remove the method again:
  
  `OrderedCollection removeSelector: #testMethod`
Reflection (revisited)

- Reflection is the ability of a program to examine and control its own implementation.

- Computational reflection is good for:
  - extending a language
  - building programming environments
  - advanced software development tools
  - ...

- Smalltalk offers some interesting reflective capabilities:
  - Object, classes and metaclasses
Reflection in Smalltalk

A particular class: Object

- is a class that models all instances
  - Every object is (indirectly) an instance of class Object
  - as in Java

- provides some methods to reason about instances
  - Understood by all instances (regardless their class)
    - #basicSize gives number of instance variables of any object
    - #instVarAt: and #instVarAt:put: access any instance variable
      - e.g., used by inspector

- all classes subclass from Object
  - and thus inherit these introspective methods
isKindOf: aClass

"Answer whether the class, aClass, is a superclass or class of the receiver."

self class == aClass

ifTrue: [true]

ifFalse: [self class inheritsFrom: aClass]
#doesNotUnderstand:

- Ability to intervene in method lookup mechanism
- Hook implemented by Smalltalk VM
  - Message #doesNotUnderstand: sent by VM to an object when a message is not understood by that object
  - Standard behaviour (defined on Object): error message!

- Mechanism to report type errors in dynamically typed Smalltalk language
  - Powerful: you can customize it!
doesNotUnderstand: aMessage

"Handle the fact that there was an attempt to send the given message to the receiver but the receiver does not understand this message (typically sent from the machine when a message is sent to the receiver and no method is defined for that selector)."

"Testing: (3 activeProcess)"
*fixed suggested by Eliot miranda to make sure

[Object new blah + 1]
  on: MessageNotUnderstood
  do: [:e | e resume: 1] does not loop indefinitely"

| exception resumeValue |
(exception := MessageNotUnderstood new)
  message: aMessage;
  receiver: self.
  if self is not the selector...
Example

```
Student

TrueType-Fonts
TrueType-Support
VB-Regex
VB-Regex-Exceptions

Student -- all -- as yet unclassified

doesNotUnderstand: aMessage
"comment stating purpose of message"

Transcript show: 'I do not understand: ', aMessage selector
```

```
Shout Workspace

Student new reflection

Transcript

I do not understand: reflection
```
Reflection (revisited)

- Reflection is the ability of a program to examine and control its own implementation.

- Computational reflection is good for:
  - extending a language
  - building programming environments
  - advanced software development tools
  - ...

- Smalltalk offers some interesting reflective capabilities:
  - Object, classes and metaclasses
MOP: Reflection in Smalltalk

- **Meta Object Protocol**
  - In Smalltalk, everything is an *object* and objects communicate using *messages*.
  - The meta/reflective protocol also uses objects

- We have already seen some part of the MOP
  - Examples on the previous slides
  - More to come...
Some example meta-objects

- **Class** represents classes
- **CompiledMethod** represents methods
- **Object** represents objects
- **Message** runtime messages
- **BlockClosure** represents smalltalk blocks
- **MethodContext** represents stack frames
Metaclass Mystery

- **Assumption**
  - Everything is an object
  - Every object has a class

- **Conclusion**
  - A class is an object
  - A class has a class

- **Mystery**
  - But what is the class of a class?

2@3
- an instance (a point)
- (2@3) class
- a class (Point)
- (2@3) class class
- a metaclass
Classes and meta classes

- Objects are instances of a class
- Classes are instances of meta classes
- Each class is the **sole** instance of a meta class (is a convention - Singleton design pattern)
- Meta classes describe class behaviour and state (subclasses, method dictionary, etc.)
Reflection in Smalltalk: metaclasses

- The class of a class is a metaclass.
- The metaclass of SmallInteger is SmallInteger class
- The name of SmallInteger class is ‘SmallInteger class’
Reflection in Smalltalk: metaclasses

- Messages understood by instances of SmallInteger
- Messages understood by SmallInteger
Reflection in Smalltalk: metaclass hierarchy

29 class
➡ SmallInteger
SmallInteger superclass ➪ Integer
SmallInteger class superclass ➪ Integer class
SmallInteger superclass superclass ➪ Number
SmallInteger class superclass superclass ➪ Number class
SmallInteger class superclass superclass ➪ SmallInteger superclass superclass class
➡ true
Reflection in Smalltalk: metaclass hierarchy

- Metaclasses form a hierarchy that mirrors the regular class hierarchy.
  - This allows subclasses to inherit class methods from superclasses (e.g., instance creation methods)
Reflection in Smalltalk

a particular metaclass : Class

- Just like Object is a class that models all instances,
  Class is a class that models all classes

- Provides some methods to reason about classes
  - Understood by all classes
    - addSubclass: create a subclass of receiver class
    - classVarNames: answers collection of names of class variables
    - subclasses: answer a set containing the receiver’s subclasses

- Class hierarchy:

  Behavior (‘superclass’ ‘methodDict’ ‘format’ ‘layout’)

  ClassDescription (‘instanceVariables’ ‘organization’)

  Class (‘subclasses’ ‘name’ ‘category’ ...)
allClassVarNames

"Answer a Set of the names of the receiver's class variables, including those defined in the superclasses of the receiver."

| aSet |
self superclass == nil
  ifTrue:
    [^self classVarNames asSet] "This is the keys so it is a new Set."
  ifFalse:
    [aSet := self superclass allClassVarNames.
     aSet addAll: self classVarNames.
     ^aSet]
Warning!

From this point on some slides are (still) based on VisualWorks Smalltalk
More Metaclass Mystery

- Just like `Object` is the root of all classes, `Object class` is the root of all metaclasses.

- But:
  - `Object class` superclass => `Class`
  - Indeed, since `Object` is a class, `Object` is an instance of `Class` or of a (direct or indirect) subclass of `Class`.

Therefore: `Object class` is a (direct or indirect) subclass of `Class`.

```
+-------------------+
| Class             |
+-------------------+
   ^                  ^
   |                  |
   |  Object class    |
   |                  |
   |                  v
+-------------------+
| Object            |
+-------------------+```
Do we need metaclasses?

- **Metaclasses are not strictly necessary.**
  - but we have to bootstrap somewhere...

- **Every class could be an instance of Class.**
  - Each class would then have exactly the same behaviour
  - So no special instance creation methods
Metaclass Madness

Behavior class ← -- Behavior

Class class ← -- Class

Object class ← -- Object
More Metaclass Madness

- Every metaclass is an instance of Metaclass.
The metaclass loop

- **Objects** are instances of a **class**
- **Classes** are instances of **meta classes**
- Each class is the **sole** instance of a meta class (is a convention - Singleton design pattern)
- **Meta classes** describe class behaviour and state (subclasses, method dictionary, etc.)
- **Meta classes** are instances of the **class Metaclass**

```
objects         classes         metaclasses
```

```
aPerson         Person          Person class
```
Examples of reflection in Smalltalk

- Let us get back to the essence and look at an interesting use of reflection in Smalltalk:
  - Scaffolding patterns
    - are programming idioms that support the rapid development of prototypes using Smalltalk
Example: scaffolding pattern

- We want to have a class that keeps some items in a collection
- and that allows to enumerate over those elements
- by using enumerators similar to those defined on the collection classes
  - basically by delegating to the appropriate enumeration method defined on the collection

<table>
<thead>
<tr>
<th>Collector</th>
</tr>
</thead>
<tbody>
<tr>
<td>items</td>
</tr>
<tr>
<td>do:</td>
</tr>
<tr>
<td>select:</td>
</tr>
<tr>
<td>inject:into:</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

```small
do: aBlock
  ^items do: aBlock
select: aBlock
  ^items select: aBlock
inject: aValue into: aBlock
  ^items inject: aValue into: aBlock
...```

38
Example: scaffolding pattern

- Solution 1: manual implementation
  - manually implement all the item enumeration methods
  - very repetitive: always the same pattern

<selector>
  ^items <selector>

where <selector> is an enumerator message like

do: aBlock
select: aBlock
inject: aValue into: aBlock
Example: scaffolding pattern

- Solution 2: static generation of methods
  - all methods exhibit exactly the same pattern
  - straightforward to statically generate the code for all the enumeration methods from a generic code template
  - using string replacement or macro expansion
Sidetrack: macro expansion

- `<ls>`
  ^items `<ls>`'

  expandMacrosWith: `do: aBlock`

  ➡ `do: aBlock`
  ^items `do: aBlock`

- `<ls>`<n><t>"This is generated code"<n><t>^items `<ls>`'

  expandMacrosWith: `do: aBlock`

  ➡ `do: aBlock`
  "This is generated code"
  ^items `do: aBlock`

- `<ls>, <2s>, ... = arguments to be replaced`
- `<n> = newline`
- `<t> = tab`
Example: scaffolding pattern

Solution 2: static generation of methods

```smalltalk
enumerationTemplate
^'<1n><t>'Generated automatically'"<n><n><t>'^items <1n>'.

compileEnumerationMethodFor: selector
    | codeTemplate code |
    codeTemplate := self enumerationTemplate.
    code := WriteStream on: String new.
    selector keywords with: (1 to: selector numArgs)
    do: [:keyword :nr | code nextPutAll: keyword;
        space; nextPutAll: 'arg';
        print: nr; space].
    self
    compile: (codeTemplate expandMacrosWith: code contents)
    classified: #enumerating

generateEnumerationMethods
    (Collection organization listAtCategoryNamed: #enumerating)
    do: [:selector | self compileEnumerationMethodFor: selector]
```

Method template with hole <1s> representing method signature and message to be sent.

Note that selectors can contain multiple keywords and arguments too.

Selectors are just copied from Collection hierarchy: all enumerating methods on Collection.
Example: scaffolding pattern

- Solution 2 at work: static generation of methods
Example: scaffolding pattern

Solution 3: Dynamic forwarding

```smalltalk
isEnumerationSelector: selector
| enumerationSelectors |

enumerationSelectors := Collection organization
    listAtCategoryNamed: #enumerating.
^enumerationSelectors includes: selector

doesNotUnderstand: aMessage

^(self isEnumerationSelector: aMessage selector)
    ifTrue: [items
        perform: aMessage selector
        withArguments: aMessage arguments
    ]
    ifFalse: [super doesNotUnderstand: aMessage]
```

inherently not possible in statically typed languages
Example: scaffolding pattern

Solution 4: On the fly code generation (by need)

doesNotUnderstand: aMessage
| selector |
selector := aMessage selector.
(self isEnumerationSelector: selector)
   ifFalse: [^super doesNotUnderstand: aMessage].
self compileEnumerationMethodFor: selector.
^self perform: selector withArguments: aMessage arguments