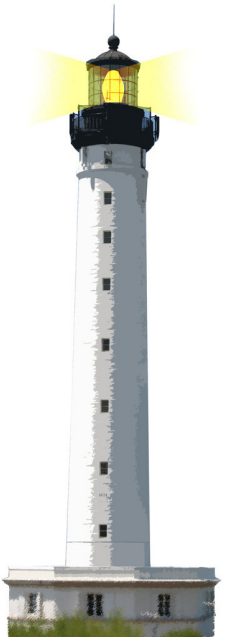
Reflective Operations for Live Programming

Damien Cassou, Stéphane Ducasse and Luc Fabresse

W7S04

Pharo

http://www.pharo.org



Behind the Scene

- What is happening when we recompile a class?
- What are the reflective operations that take place?

A Typical Scenario

- Define one class
- Define some methods
- Create some instances
- Add an instance variable to the class
- Existing instances got mutated
- Continue working

Operations Supporting Interactive Coding

- Dynamic class (re)definition
- Method recompilation
- Transparent instance migration
  - Collecting instances
  - Switching pointers from old to new instances

Getting All Instances

```
Dictionary allInstances size
```

```
Window allInstances first close
```

Getting All Pointers to an Object

```
anObject pointersTo
```

returns all objects that store a reference to anObject

Symmetric Pointer Swapping

```
anObject become: anotherObject
```

- All the pointers to anObject point now to anotherObject
- And "the inverse" atomically

Symmetric Pointer Swapping

```
| pt1 pt2 pt3 |
pt1 := 0@0.
pt2 := pt1.
pt3 := 100@100.
pt1 become: pt3.
self assert: pt2 = (100@100).
self assert: pt3 = (0@0).
self assert: pt1 = (100@100)
```

Asymmetric Pointer Swapping

Swap all the pointers from one object to the other (asymmetric)

```
anObject becomeForward: anotherObject
```

Example: Asymmetric Pointer Swapping

```
| pt1 pt2 pt3 |
pt1 := 0@0.
pt2 := pt1.
pt3 := 100@100.
pt1 becomeForward: pt3.
self assert: pt1 = (100@100).
self assert: pt1 == pt2.
self assert: pt2 == pt3.
```

Changing the Class of an Object

```
Class >> adoptInstance: anInstance
  "Change the class of anInstance to me. Returns the class
     rather than the modified instance"
```

- Limited reflective feature
- Target class should have the same format as the original one

Essence of a Class

1. A format i.e., a number of instance variables and types (named/indexed)
2. A superclass
3. A method dictionary

Class initialize

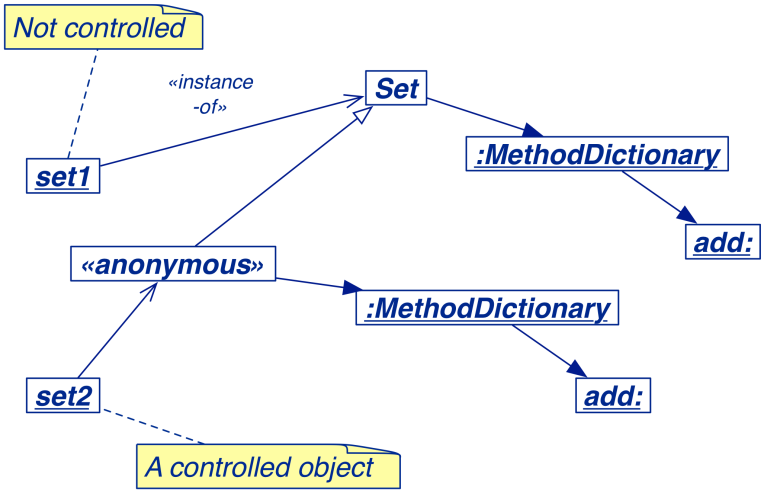
```
Behavior >> initialize

  super initialize.
  self superclass: Object.
  self methodDict: self emptyMethodDictionary.
  self setFormat: Object format.
```

Instance Specific Behavior

```
| behavior model newClass |
behavior := Behavior new.
behavior superclass: Model.
behavior setFormat: Model format.
model := Model new.
model primitiveChangeClassTo: behavior new.
self assert: model class = behavior.
self assert: model class superclass = Model.
behavior compile: 'foo ^ 2'.
self assert: model foo = 2.
self should: [Model new foo] raise: MessageNotUnderstood
```

Instance Specific Behavior



Anonymous Classes For Spying

```
| logClass set |
logClass := Behavior new.
logClass superclass: Set;
  setFormat: Set format.
logClass compile: 'add: anObject
    Transcript show: ''adding '', anObject printString; cr.
    ^ super add: anObject'.
set := Set new.
set add: 1.
set class.
set primitiveChangeClassTo: logClass basicNew.
set add: 2.
```

Conclusion

- Reflection is a solid foundation for innovation and language extensibility
- Avoid using reflective operations in domain code
- Understand when you absolutely need reflection

A course by

and

in collaboration with