4. Reflection

Oscar Nierstrasz

Selected material by Marcus Denker and Stéphane Ducasse
Reflection allows you to both *examine* and *alter* the meta-objects of a system.

Using reflection to modify a running system requires some care.
Roadmap

> Reification and reflection
> Reflection in Programming Languages
> Introspection
  — Inspecting objects
  — Querying code
  — Accessing run-time contexts
> Intercession
  — Overriding doesNotUnderstand:
  — Anonymous classes
  — Method wrappers
Roadmap

> Reification and reflection
> Reflection in Programming Languages
> Introspection
  — Inspecting objects
  — Querying code
  — Accessing run-time contexts
> Intercession
  — Overriding doesNotUnderstand:
  — Anonymous classes
  — Method wrappers
Why we need reflection

As a programming language becomes *higher and higher level*, its implementation in terms of underlying machine involves *more and more tradeoffs*, on the part of the implementor, about what cases to optimize at the expense of what other cases. … the *ability to cleanly integrate* something outside of the language’s scope *becomes more and more limited*

Kiczales, in Paepcke 1993
What is are Reflection and Reification?

> **Reflection** is the ability of a program to manipulate as data something representing the state of the program during its own execution.

  — **Introspection** is the ability for a program to observe and therefore reason about its own state.
  
  — **Intercession** is the ability for a program to modify its own execution state or alter its own interpretation or meaning.

> **Reification** is the mechanism for encoding execution state as data

  — Bobrow, Gabriel & White, 1993
Structural and behavioral reflection

> **Structural reflection** lets you reify and reflect on
  — the *program* currently executed
  — its *abstract data types*.

> **Behavioral reflection** lets you reify and reflect on
  — the language *semantics* and *implementation* (processor)
  — the data and implementation of the *run-time system*.

Malenfant et al., *A Tutorial on Behavioral Reflection and its Implementation*, 1996
Reflection and Reification

Metaobjects

Objects

reification

intercession (reflection)

introspection (reflection)
Roadmap

> Reification and reflection
> **Reflection in Programming Languages**
> Introspection
>   — Inspecting objects
>   — Querying code
>   — Accessing run-time contexts
> Intercession
>   — Overriding doesNotUnderstand:
>   — Anonymous classes
>   — Method wrappers
Metaprogramming in Programming Languages

> The meta-language and the language can be different:
  — Scheme and an OO language

> The meta-language and the language can be same:
  — Smalltalk, CLOS
  — In such a case this is a *metacircular architecture*
Introspection in Java

// Without introspection
World world = new World();
world.hello();

// With introspection
Class cls = Class.forName("World");
Method method = cls.getMethod("hello", null);
method.invoke(cls.newInstance(), null);
Reflection in Smalltalk


Instance  ?  Class

HelloWorld

-- all --
as yet unclassified

Browser  Hierarchy  Variables  Implementors  Inheritance  Senders  Versions  View

hello

^ 'hello'

"With reflection"
Object subclass: #HelloWorld
  instanceVariableNames: "
  classVariableNames: "
  poolDictionaries: "
  category: 'HelloWorld'.
(Smalltalk at: #HelloWorld) compile: 'hello ^ "hello"'.
((Smalltalk at: #HelloWorld) perform: #new) perform: #hello.

'hello'
Three approaches

1. Tower of meta-circular interpreters
2. Reflective languages
3. Open implementation
1. Tower of meta-circular interpreters

> Each level interprets and controls the next
   — 3-Lisp, Scheme

> “Turtles all the way down” [up]
   — In practice, levels are reified on-demand
2. Reflective languages

> Meta-entities control base entities
  – Smalltalk, Self
  – Language is written in itself
3. Open implementation

Meta-object protocols provide an interface to access and modify the implementation and semantics of a language — CLOS

More efficient, less expressive than infinite towers
Roadmap

> Reification and reflection
> Reflection in Programming Languages
> **Introspection**
  > Inspecting objects
  > Querying code
  > Accessing run-time contexts
> Intercession
  > Overriding doesNotUnderstand:
  > Anonymous classes
  > Method wrappers
The Essence of a Class

1. A format (e.g. a set of instance variables)
2. A superclass
3. A method dictionary
Behavior class>> new

> In Pharo:

```
Behavior class>>new
| classInstance |
classInstance := self basicNew.
classInstance methodDictionary:
    classInstance emptyMethodDictionary.
classInstance superclass: Object.
classInstance setFormat: Object format.
^ classInstance
```

**NB:** not to be confused with Behavior>>new!
The Essence of an Object

1. Class pointer
2. Values

> Can be special:
   — SmallInteger
   — Indexed rather than pointer values
   — Compact classes (CompiledMethod, Array ...)

Wednesday, 14 September 11
Metaobjects vs metaclasses

Need distinction between metaclass and metaobject!

- A metaclass is a class whose instances are classes
- A metaobject is an object that describes or manipulates other objects
  - Different metaobjects can control different aspects of objects
Some MetaObjects

> **Structure:**
>  — Behavior, ClassDescription, Class, Metaclass, ClassBuilder

> **Semantics:**
>  — Compiler, Decompiler, IRBuilder

> **Behavior:**
>  — CompiledMethod, BlockContext, Message, Exception

> **ControlState:**
>  — BlockContext, Process, ProcessorScheduler

> **Resources:**
>  — WeakArray

> **Naming:**
>  — SystemDictionary

> **Libraries:**
>  — MethodDictionary, ClassOrganizer
Meta-Operations

“Meta-operations are operations that provide information about an object as opposed to information directly contained by the object ... They permit things to be done that are not normally possible”

*Inside Smalltalk*
Accessing state

```plaintext
pt := 10@3.
pt instVarNamed: 'x'.
pt instVarNamed: 'x' put: 33.
pt
```

> `Object>>instVarNamed: aString`
> `Object>>instVarNamed: aString put: anObject`
> `Object>>instVarAt: aNumber`
> `Object>>instVarAt: aNumber put: anObject`
Accessing meta-information

> Object>>class
> Object>>identityHash

'hello' class
(10@3) class
Smalltalk class
Class class
Class class class
Class class class class

'hello' identityHash
Object identityHash
5 identityHash

ByteString
Point
SystemDictionary
Class class
Metaclass
Metaclass class

2664
2274
5
Changes

> `Object>>primitiveChangeClassTo: anObject`
   — Both classes should have the same format, *i.e.*, the same physical structure of their instances
   — “Not for casual use”

> `Object>>become: anotherObject`
   — Swap the object pointers of the receiver and the argument.
   — All variables in the entire system that used to point to the receiver now point to the argument, and vice-versa.
   — Fails if either object is a SmallInteger

> `Object>>becomeForward: anotherObject`
   — Like `become:` but only in one direction.
Implementing Instance Specific Methods

ReflectionTest>>testPrimitiveChangeClassTo
| behavior browser |

behavior := Behavior new. "an anonymous class"
behavior superclass: Browser.
behavior setFormat: Browser format.
browser := Browser new.

browser primitiveChangeClassTo: behavior new.
behavior compile: 'thisIsATest ^ 2'.

self assert: browser thisIsATest = 2.
self should: [Browser new thisIsATest]
raise: MessageNotUnderstood.
become:

> Swap all the pointers from one object to the other and back (symmetric)

```
ReflectionTest>>testBecome
    | pt1 pt2 pt3 |

pt1 := 0@0.
pt2 := pt1.
pt3 := 100@100.
pt1 become: pt3.

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

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

```
ReflectionTest>>testBecomeForward
  | pt1  pt2  pt3 |
    pt1 := 0@0.
    pt2 := pt1.
    pt3 := 100@100.
    pt1 becomeForward: pt3.

    self assert: pt1 = (100@100).
    self assert: pt1 == pt2.
```
Roadmap

> Reification and reflection
> Reflection in Programming Languages
> **Introspection**
>   — Inspecting objects
>   — **Querying code**
>   — Accessing run-time contexts
> > Intercession
>   — Overriding doesNotUnderstand:
>   — Anonymous classes
>   — Method wrappers
Code metrics

Collection allSuperclasses size. 2
Collection allSelectors size. 610
Collection allInstVarNames size. 0
Collection selectors size. 163
Collection instVarNames size. 0
Collection subclasses size. 9
Collection allSubclasses size. 101
Collection linesOfCode. 864
SystemNavigation
default browseAllImplementorsOf: #,
Recap: Classes are objects too

> Object
  — Root of inheritance
  — Default Behavior
  — Minimal Behavior

> Behavior
  — Essence of a class
  — Anonymous class
  — Format, methodDict, superclass

> ClassDescription
  — Human representation and organization

> Metaclass
  — Sole instance
Classes are Holders of CompiledMethods
Invoking a message by its name

> Asks an object to execute a message
— Normal method lookup is performed

5 factorial
5 perform: #factorial

120
Executing a compiled method

`CompiledMethod>>valueWithReceiver:arguments:`

No lookup is performed!

`(SmallInteger>>#factorial)
 valueWithReceiver: 5
 arguments: #()`

Error: key not found

`(Integer>>#factorial)
 valueWithReceiver: 5
 arguments: #() 120`
MethodReference

A MethodReference is a lightweight proxy for a CompiledMethod. Has methods for pointed to the CompileMethod's source statements, byte codes. Is heavily used by Tools.

Instance Variables
- `classIsMeta`: Boolean class vs. instance
- `classSymbol`: Symbol for method's class (without class keyword if meta)
- `methodSymbol`: Symbol for method's selector
- `stringVersion`: 'Class>>selector:' format
Finding super-sends within a hierarchy

class := Collection.
SystemNavigation default
  browseMessageList: (class withAllSubclasses gather: [:each |
    each methodDict associations
      select: [:assoc | assoc value sendsToSuper]
      thenCollect: [:assoc | MethodReference class: each
        selector: assoc key]])
  name: 'Supersends of ', class name, ' and its subclasses'
Roadmap

- Reification and reflection
- Reflection in Programming Languages
- **Introspection**
  - Inspecting objects
  - Querying code
  - **Accessing run-time contexts**
- Intercession
  - Overriding doesNotUnderstand:
  - Anonymous classes
  - Method wrappers
Accessing the run-time stack

> The execution stack can be *reified* and *manipulated* on demand

– thisContext is a pseudo-variable which gives access to the stack
What happens when a method is executed?

> We need space for:
  — The temporary variables
  — Remembering where to return to

> Everything is an Object!
  — So: we model this space with objects
  — Class MethodContext

```ruby
ContextPart variableSubclass: #MethodContext
  instanceVariableNames: 'method closureOrNil receiver'
  classVariableNames: ''
  poolDictionaries: ''
  category: 'Kernel-Methods'
```
MethodContext

> MethodContext holds all state associated with the execution of a CompiledMethod
  — Program Counter (pc, from ContextPart)
  — the Method itself (method)
  — Receiver (receiver) and the Sender (sender)

> The sender is the previous MethodContext
  — (or BlockContext)
  — The *chain of senders* is a stack
  — It grows and shrinks on activation and return
Contextual halting

> You can’t put a halt in methods that are called often
— e.g., OrderedCollection>>add:
— Idea: only halt if called from a method with a certain name

```
HaltDemo>>haltIf: aSelector
| context |
context := thisContext.
[context sender isNil]
whileFalse:
  [context := context sender.
   (context selector = aSelector)
   ifTrue: [ Halt signal ] ].
```

NB: Object>>haltIf: in Pharo is similar
HaltDemo

```smalltalk
HaltDemo>>foo
    self haltIf: #bar.
    ^ 'foo'

HaltDemo>>bar
    ^ (self foo), 'bar'
```
HaltDemo

HaltDemo>>foo
  
  ^ 'foo'

HaltDemo>>bar
  
  ^ (self foo), 'bar'

HaltDemo new foo
### HaltDemo

```
HaltDemo>>foo
    self haltIf: #bar.
    ^ 'foo'

HaltDemo>>bar
    ^ (self foo), 'bar'
```

```
HaltDemo new foo
```

```
'foo'
```

![Image showing halting debugger interface](attachment:image.png)
HaltDemo

HaltDemo>>foo
    self haltIf: #bar.
    ^ 'foo'

HaltDemo>>bar
    ^ (self foo), 'bar'

HaltDemo new foo

HaltDemo new bar

HaltDemo new bar
Roadmap

> Reification and reflection
> Reflection in Programming Languages
> Introspection
  — Inspecting objects
  — Querying code
  — Accessing run-time contexts
> Intercession
  — Overriding doesNotUnderstand:
  — Anonymous classes
  — Method wrappers
Overriding doesNotUnderstand:

> Introduce a *Minimal Object*
  – Wraps a normal object
  – Does not understand very much
  – Redefines doesNotUnderstand:
  – Superclass is `nil` or `ProtoObject`
  – Uses `becomeForward:` to substitute the object to control
Minimal Object at Work

old reference

anObj (3)

new reference

capsule doesNot Understand: aMessage (2)

a capsule or a spy controlled object

anObj m (1)

VM

MinimalObject

fooBar

doesNotUnderstand: #fooBar

Subject

fooBar

value
Logging message sends with a minimal object

ProtoObject subclass: #LoggingProxy
  instanceVariableNames: 'subject invocationCount'
  classVariableNames: '
  poolDictionaries: '
  category: 'PBE-Reflection'

LoggingProxy>>initialize
  invocationCount := 0.
  subject := self.

LoggingProxy>>doesNotUnderstand: aMessage
  Transcript show: 'performing ', aMessage printString; cr.
  invocationCount := invocationCount + 1.
  ^ aMessage sendTo: subject

Message>>sendTo: receiver
  ^ receiver perform: selector withArguments: args
Using become: to install a proxy

```plaintext
| point |
point := 1@2.

LoggingProxy new become: point.
self assert: point invocationCount = 0.
self assert: point + (3@4) = (4@6).
self assert: point invocationCount = 1.
```

NB: become: will swap the subject variable of the proxy
Limitations

> self problem
  — Messages sent by the object to itself are not trapped!

> Class control is impossible
  — Can’t swap classes

> Interpretation of minimal protocol
  — What to do with messages that are understood by both the MinimalObject and its subject?
Using minimal objects to dynamically generate code

DynamicAccessors>>doesNotUnderstand: aMessage
| messageName |
messageName := aMessage selector asString.
(self class instVarNames includes: messageName)
  ifTrue: [self class compile:
    messageName, String cr, ' ^ ' 
    , messageName.
    ^ aMessage sendTo: self].
super doesNotUnderstand: aMessage

A minimal object can be used to dynamically generate or lazily load code that does not yet exist.
Roadmap

> Reification and reflection
> Reflection in Programming Languages
> Introspection
  — Inspecting objects
  — Querying code
  — Accessing run-time contexts
> Intercession
  — Overriding doesNotUnderstand:
  — Anonymous classes
  — Method wrappers
Message control with anonymous classes

> Create an *anonymous class*
  - Instance of Behavior
  - Define controlling methods
  - Interpose it between the instance and its class
Selective control

- Not controlled
- «instance-of»
- Set
- «anonymous»
- :MethodDictionary
- add:
- set1
- add:
- set2
- A controlled object
Anonymous class in Pharo

| anonClass set |
anonClass := Behavior new.
anonClass superclass: Set;
    setFormat: Set format.

anonClass compile:
    'add: anObject
        Transcript show: ''adding '', anObject printString; cr.
        ^ super add: anObject'.

set := Set new.
set add: 1.

set primitiveChangeClassTo: anonClass basicNew.
set add: 2.
Evaluation

> Either instance-based or group-based
> Selective control
> No self-send problem
> Good performance
> Transparent to the user
> Requires a bit of compilation
  — (could be avoided using clone as in Method Wrapper)
Roadmap

> Reification and reflection
> Reflection in Programming Languages
> Introspection
  — Inspecting objects
  — Querying code
  — Accessing run-time contexts
> Intercession
  — Overriding doesNotUnderstand:
  — Anonymous classes
  — Method wrappers
Method Substitution

First approach:
> Add methods with mangled names
  — but the user can see them

Second approach:
> Wrap the methods without polluting the interface
  — replace the method by an object that implements run:with:in:
A MethodWrapper replaces an original CompiledMethod in the method dictionary of a class and wraps it by performing some before and after actions.

```plaintext
run: aSelector with: anArray in: aReceiver
...
^aReceiver withArgs: anArray executeMethod: method
```
A LoggingMethodWrapper

\[
\text{LoggingMethodWrapper}\triangleright\triangleright\text{initializeOn: aCompiledMethod}
\]
\[
\text{method} := \text{aCompiledMethod}.
\]
\[
\text{reference} := \text{aCompiledMethod methodReference}.
\]
\[
\text{invocationCount} := 0
\]

\[
\text{LoggingMethodWrapper}\triangleright\triangleright\text{install}
\]
\[
\text{reference actualClass methodDictionary}
\]
\[
\text{at: reference methodSymbol}
\]
\[
\text{put: self}
\]

\[
\text{LoggingMethodWrapper}\triangleright\triangleright\text{run: aSelector with: anArray in: aReceiver}
\]
\[
\text{invocationCount} := \text{invocationCount} + 1.
\]
\[
\wedge \text{aReceiver withArgs: anArray executeMethod: method}
\]

**NB:** Duck-typing also requires (empty) flushCache, methodClass:, and selector: methods
Installing a LoggingMethodWrapper

```
logger := LoggingMethodWrapper on: Integer>>#factorial.

logger invocationCount. 0
5 factorial.
logger invocationCount. 0

logger install.
[ 5 factorial ] ensure: [logger uninstall].
logger invocationCount. 6

10 factorial.
logger invocationCount. 6
```
Checking Test Coverage

```
TestCoverage>>run: aSelector with: anArray in: aReceiver
  self mark; uninstall.
  ^ aReceiver withArgs: anArray executeMethod: method

TestCoverage>>mark
  hasRun := true
```
Evaluation

> Class based:
  — all instances are controlled
> Only known messages intercepted
> A single method can be controlled
> Does not require compilation for installation/removal
What you should know!

> What is the difference between introspection and intercession?
> What is the difference between structural and Behavioral reflection?
> What is an object? What is a class?
> What is the difference between performing a message send and simply evaluating a method looked up in a MethodDictionary?
> In what way does thisContext represent the run-time stack?
> What different techniques can you use to intercept and control message sends?
Can you answer these questions?

> What form of “reflection” is supported by Java?
> What can you do with a metacircular architecture?
> Why are Behavior and Class different classes?
> What is the class ProtoObject good for?
> Why is it not possible to become: a SmallInteger?
> What happens to the stack returned by thisContext if you proceed from the self halt?
> What is the metaclass of an anonymous class?
You are free:

- to copy, distribute, display, and perform the work
- to make derivative works
- to make commercial use of the work

Under the following conditions:

**Attribution.** You must attribute the work in the manner specified by the author or licensor.

**Share Alike.** If you alter, transform, or build upon this work, you may distribute the resulting work only under a license identical to this one.

- For any reuse or distribution, you must make clear to others the license terms of this work.
- Any of these conditions can be waived if you get permission from the copyright holder.

Your fair use and other rights are in no way affected by the above.

[http://creativecommons.org/licenses/by-sa/3.0/](http://creativecommons.org/licenses/by-sa/3.0/)