Reflective Programming in Smalltalk

Stéphane Ducasse
Stephane.Ducasse@inria.fr
http://stephane.ducasse.free.fr/

M. Denker and S. Ducasse - 2005

License: CC-Attribution-ShareAlike 2.0

http://creativecommons.org/licenses/by-sa/2.0/

Why...

"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'92a]

Definition

"*Reflection* is the ability of a program to manipulate as data something representing the state of the program during its own execution. There are two aspects of such manipulation: introspection and intercession.

Introspection is the ability for a program to observe and therefore reason about its own state.

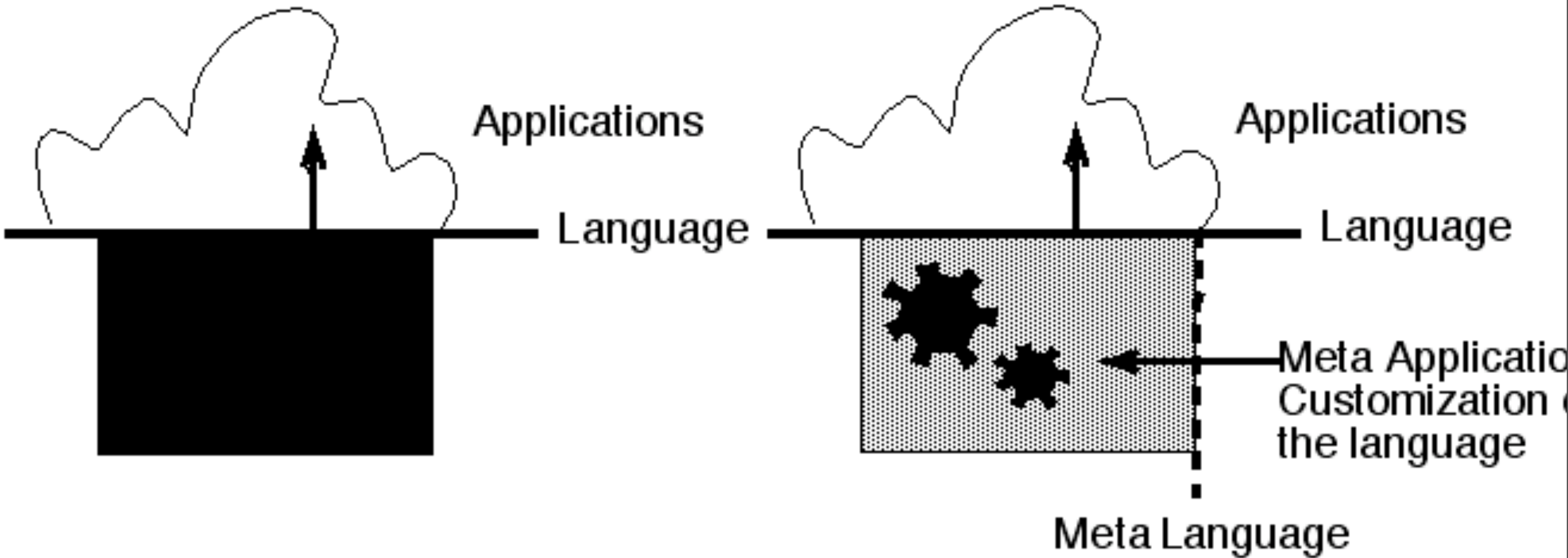
Intercessory is the ability for a program to modify its own execution state or alter its own interpretation or meaning.

Both aspects require a mechanism for encoding execution state as data: providing such an encoding is called reification." [Bobrow, Gabriel and White in Paepke'92]

Consequences

- A system having itself as application domain and that is causally connected with this domain can be qualified as a reflective system [Pattie Maes]
- A reflective system has an internal representation of itself.
- A reflective system is able to act on itself with the ensurance that its representation will be causally connected (up to date).
- A reflective system has some static capacity of self-representation and dynamic self-modification in constant synchronization

Meta Programming in Prog. Language



- 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

The Essence of a Class

- A format (number of instance variables and types)
- A superclass
- A method dictionary

Behavior >> new

In Squeak (3.8)

```
Behavior>>new

    | classInstance |
    classInstance := self basicNew.
    classInstance methodDictionary: classInstance
emptyMethodDictionary.
    classInstance superclass: Object.
    classInstance setFormat: Object format.
    ^ classInstance
```

The Essence of an Object

- class pointer
- values

- Can be special:
  - the pointer pointing to the object is the object itself
  - character, smallInteger (compact classes)

Some MetaObjects

- Structure: Behavior, ClassDescription, Class, Metaclass, ClassBuilder
- Semantics: Compiler, Decompiler, ProgramNode, ProgramNodeBuilder, IRBuilder
- Behavior: CompiledMethod, CompiledBlock, Message, Exception
- ControlState: Context, BlockContext, Process, ProcessorScheduler
- Resources: ObjectMemory, WeakArray
- Naming: SystemDictionary, Namespace
- Libraries: MethodDictionary, ClassOrganizer

Meta-Operations

- MetaOperations 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]"

Access

- Object>>instVarAt: aNumber
- Object>>instVarNamed: aString
- Object>>instVarAt: aNumber put: anObject

- Browser new instVarNamed: 'classOrganizer'
- | pt |
  pt := 10@3.
  pt instVarNamed: 'x' put: 33.
  pt
- > 33@3

Access

- Object>>class
- Object>>identityHash

Changes

- Object>>changeClassOfThat: anInstance
  in VW and Squeak both classes should have the same format, i.e., the same physical structure of their instances

- Object>>become: anotherObject
- Object>>becomeForward: anotherObject

Implementing Instance Specific Methods

In Squeak 3.8

```
| behavior browser |
behavior := Behavior new.
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: and oneWayBecome:

- become: is symmetric and swaps all the pointers
- oneWayBecome: (in VW) becomeForward: (Squeak) changes pointers only in one way

become:

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

- | 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).
```

becomeForward:

- Swap all the pointers from one object to the other one

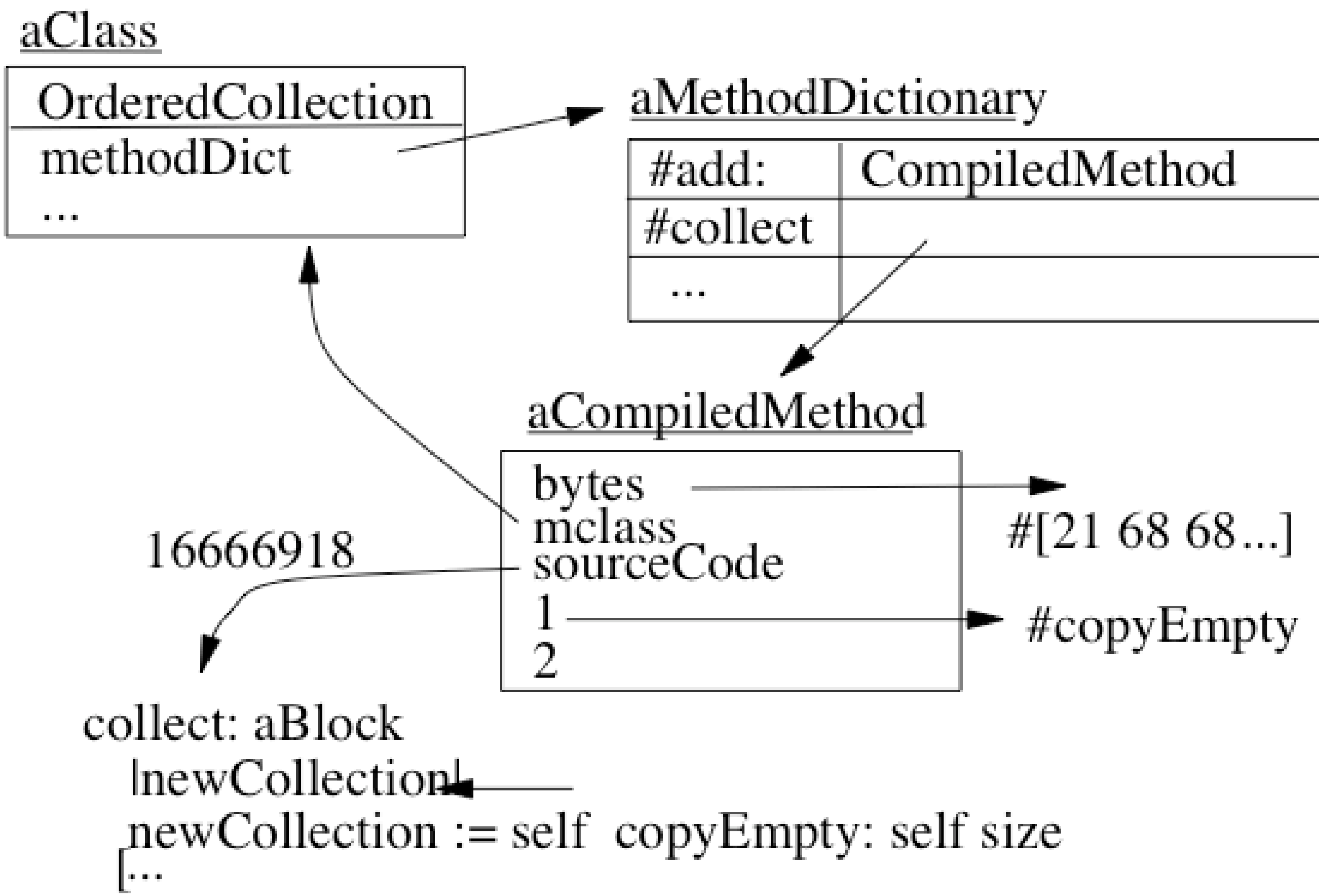
- | pt1 pt2 pt3 |

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

Structure

- Objects represent classes
- Object root of inheritance
  - default behavior
  - minimal behavior
- Behavior: essence of class
  - anymous class
  - format, methodDict, superclass
- ClassDescription:
  - human representation and organization
- Metaclass:
  - sole instance

CompiledMethod Holders



ClassBuilder

- Manages class creation
  - unique instance
  - format with superclass checking
  - changes of existing instance when class structure changes

Some Selected Protocols

- Illustrated by the tools of the IDE

- Class>>selectors
- Class>>superclass
- Class>>compiledMethodAt: aSymbol
- Class>>instVarNames
- Class>>compiler

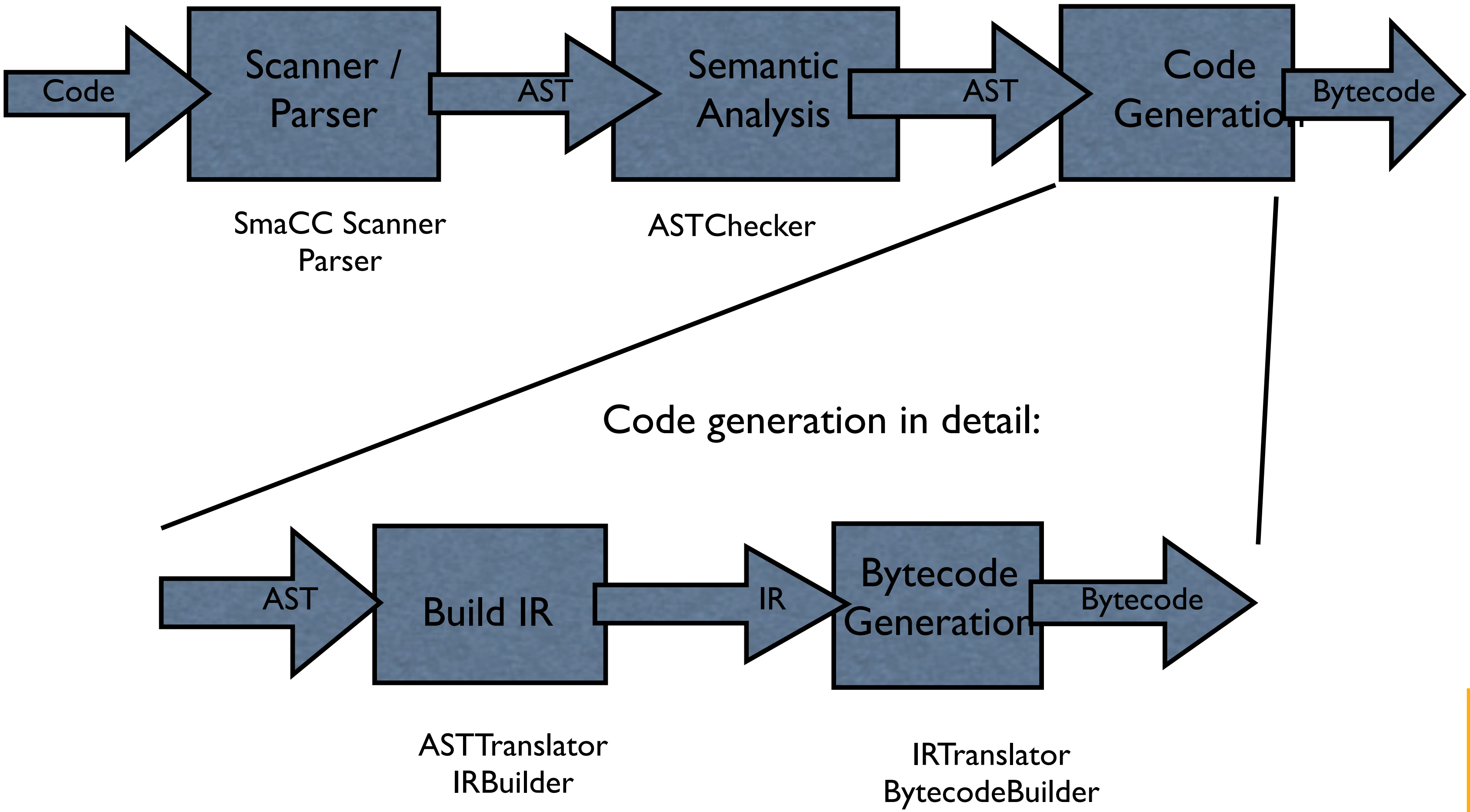
The Smalltalk Compiler

Compiler

- Fully reified compilation process:

- Scanner/Parser (build with SmaCC)
  - builds AST (from Refactoring Browser)
- Semantic Analysis: ASTChecker
  - annotates the AST (*e.g.*, var bindings)
- Translation to IR: ASTTranslator
  - uses IRBuilder to build IR (Intermediate Representation)
- Bytecode generation: IRTranslator
  - uses BytecodeBuilder to emit bytecodes

Compiler: Overview



Compiler: Syntax

- SmaCC: Smalltalk Compiler Compiler
- Like Lex/Yacc
- Input:
  - scanner definition: Regular Expressions
  - parser: BNF Like Grammar
  - code that build AST as annotation
- SmaCC can build LARL(1) or LR(1) parser
- Output:
  - class for Scanner (subclass SmaCCScanner)
  - class for Parser (subclass SmaCCParser)

Scanner

SmaCCParserGenerator: SqueakScanner/SqueakParser

Scanner Parser Compile Test Tutorial

```
<decimalNumber>:      [0-9]+ (\. [0-9]+)? ;
<radixNumber>:        [0-9]+ r [0-9A-Z]+ (\. [0-9A-Z]+)? ;
<exponentNumber>:     (<decimalNumber> | <radixNumber>) e \-? [0-9]+ ;
<number>:             <decimalNumber> | <radixNumber> | <exponentNumber> ;
<negativeNumber>:     \- <number> ;
<string>:             \'  [^\']* \' (\' [^\']* \')* ;
<name>:               [a-zA-Z] [a-zA-Z0-9]* ;
<keyword>:            <name> \: ;
<multikeyword>:       <name> \: (<name> \: )+ ;
<binarySymbol>:       [\~\!\@\%\&\*\-\+\=\\\|\?\/\>\<\,] [\~\!\@\%\&\*\-\+\=\\\|\?\/\>\<\,]* ;
<assignment>:         \: \= | \← ;
<alternateKeyword>: \: <name> \: (<name> \:)* ;
<whitespace>:         \s+ ;
<comment>:            \" [^\"]* \" ;
<character>:          \$ . ;
<period>:             \. ;
<variableAssignment>:     <name> \: \= ;

<anyChar>:            . ; # For VW literal arrays that handle #(;) -> #(#';')
```

Parser

SmaCCParserGenerator: SqueakScanner/SqueakParser

Scanner Parser Compile Test Tutorial

```
|    BinaryMessage                                {#first:}
|    KeywordMessage                               {#first:};

KeywordMessageSend:
     BinaryMessageSend KeywordMessage             {#messageSend:}
|    UnaryMessageSend KeywordMessage              {#messageSend:}
|    Primary KeywordMessage                       {#messageSend:};

KeywordMessage:
     <keyword> KeywordArgument                    {#messagePart:}
|    KeywordMessage <keyword> KeywordArgument {#addMessagePart:};

KeywordArgument:
     BinaryMessageSend                            {#first:}
|    UnaryMessageSend                             {#first:}
|    Primary                                      {#first:};

BinaryMessageSend:
     BinaryMessageSend BinaryMessage              {#messageSend:}
|    UnaryMessageSend BinaryMessage               {#messageSend:}
|    Primary BinaryMessage                        {#messageSend:};

BinaryMessage :
     <binarySymbol> BinaryArgument                {#messagePart:};
```

Calling Parser code

Compiler: AST

- AST: Abstract Syntax Tree
- Encodes the Syntax as a Tree
- No semantics yet!
- Uses the RB Tree:
  - visitors
  - backward pointers in ParseNodes
  - transformation (replace/add/delete)
  - pattern directed TreeRewriter
  - PrettyPrinter

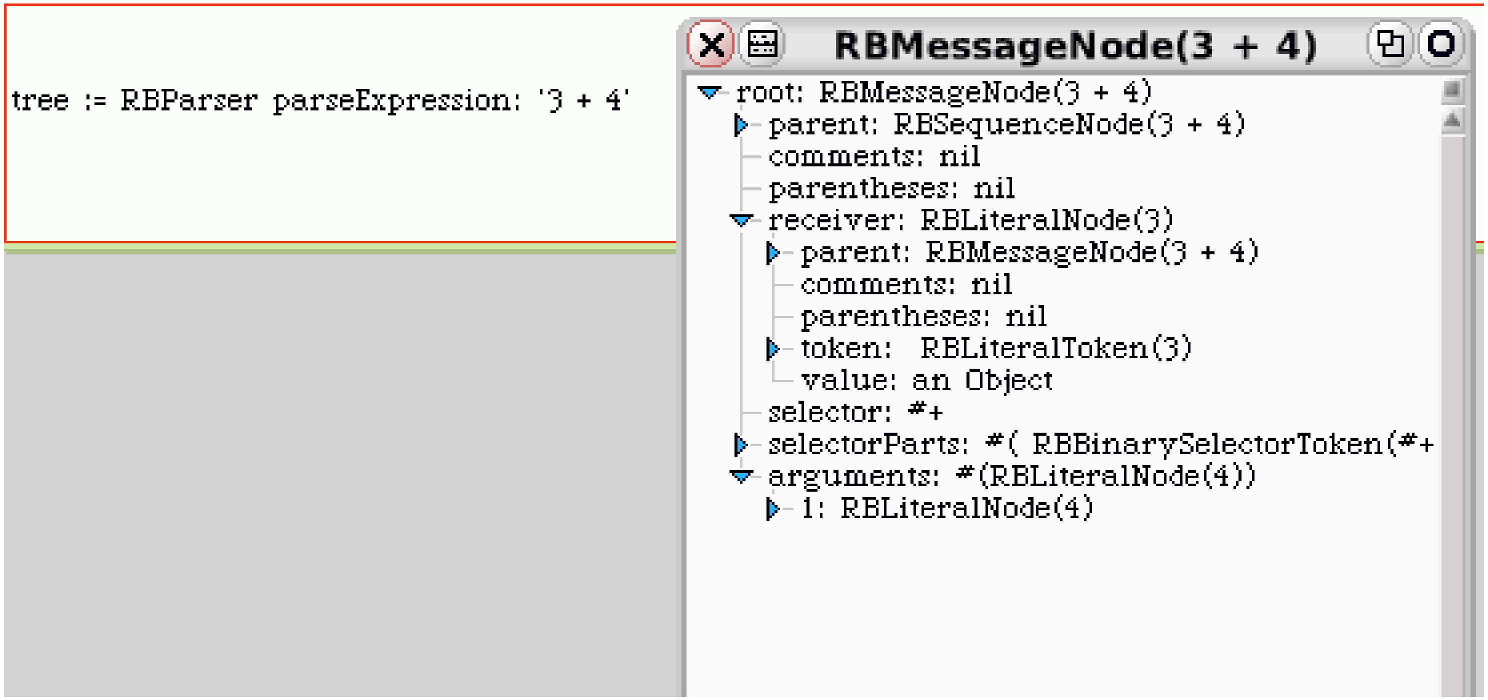
RBProgramNode
- RBDoItNode
- RBMethodNode
- RBReturnNode
- RBSequenceNode
- RBValueNode
  - RBArrayNode
  - RBAssignmentNode
  - RBBlockNode
  - RBCascadeNode
  - RBLiteralNode
  - RBMessageNode
  - RBOptimizedNode
  - RBVariableNode

Compiler: Semantics

- We need to analyse the AST
  - names need to be linked to the Variables according to the scoping rules

- ASTChecker implemented as a visitor
  - subclass of RBProgramNodeVisitor
  - visits the nodes
  - grows and shrinks Scope chain
  - method/Blocks are linked with the Scope
  - variable definitions and references are linked with objects describing the variables

A Simple Tree



A Simple Visitor

- RBProgramNodeVisitor new visitNode: tree.
- does nothing except walking throw the tree

LiteralGatherer

```
RBProgramNodeVisitor subclass: #LiteralGatherer
	instanceVariableNames: 'literals'
	classVariableNames: ''
	poolDictionaries: ''
	category: 'Compiler-AST-Visitors'

initialize
		literals := Set new.
literals
		^literals
acceptLiteralNode: aLiteralNode
		literals add: aLiteralNode value.

(TestVisitor new visitNode: tree) literals
#(3 4)
```

Compiler III: IR

- IR: Intermediate Representation
  - semantic like Bytecode, but more abstract
  - independent of the bytecode set
  - IR is a tree
  - IR nodes allow easy transformation
  - decompilation to RB AST

- IR build from AST using ASTTranslator:
  - AST Visitor
  - uses IRBuilder

Compiler 4: Bytecode

- IR needs to be converted to Bytecode
  - IRTranslator: Visitor for IR tree
  - Uses BytecodeBuilder to generate Bytecode
  - Builds a compiledMethod

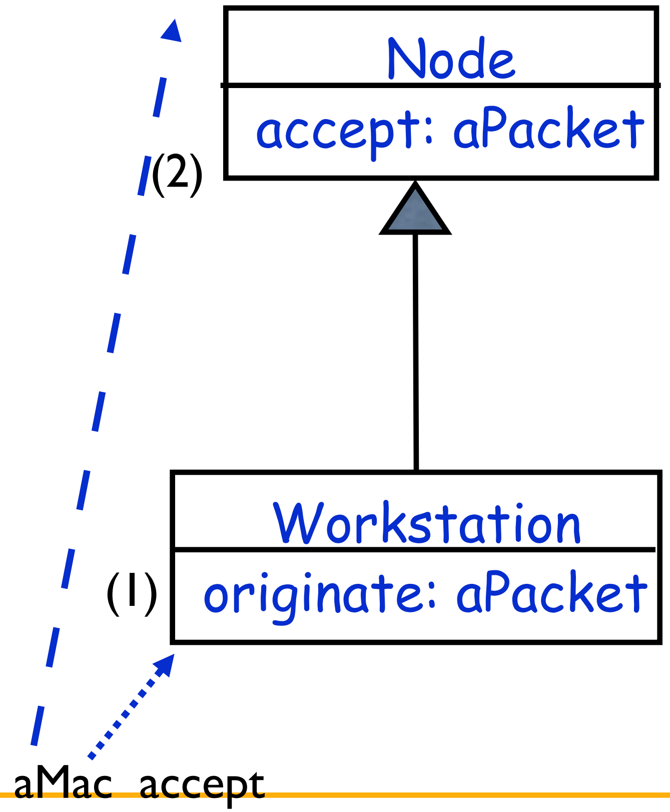
```
testReturn1
| iRMethod aCompiledMethod |
iRMethod := IRBuilder new
numRargs: 1;
addTemps: #(self);  "receiver and args declarations"
pushLiteral: 1;
returnTop;
ir.

aCompiledMethod := iRMethod compiledMethod.
self should: [(aCompiledMethod valueWithReceiver: nil arguments: #() ) = 1].
```

Behavior

- Method Lookup
- Method Application

The Essence

- Look on the *receiver class* (1)
- Follow *inheritance* link (2)



doesNotUnderstand:

- When the lookup fails
  - doesNotUnderstand: on the original message receiver
  - reification of the message
- 2 zork
- leads to
  - 2 doesNotUnderstand: aMessage
  - aMessage selector -> #zork

Invoking a message from its name

- Object>>perform: aSymbol
- Object>>perform: aSymbol with: arg
- ...

- Asks an object to execute a message
- The method lookup is done!

- 5 factorial
- 5 perform: #factorial

Executing a compiled method

CompiledMethod>>valueWithReceiver:arguments:

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

-> 120

No lookup is performed

Other Reflective Entities

- Execution stack can be reified and manipulated on demand
- thisContext is a pseudo variable which gives access to the stack

What happens on Method

- We need a space for
  - the temporary variables
  - remembering where to return to

- Everything is an Object!
- So: we model this space as Objects
- Class MethodContext

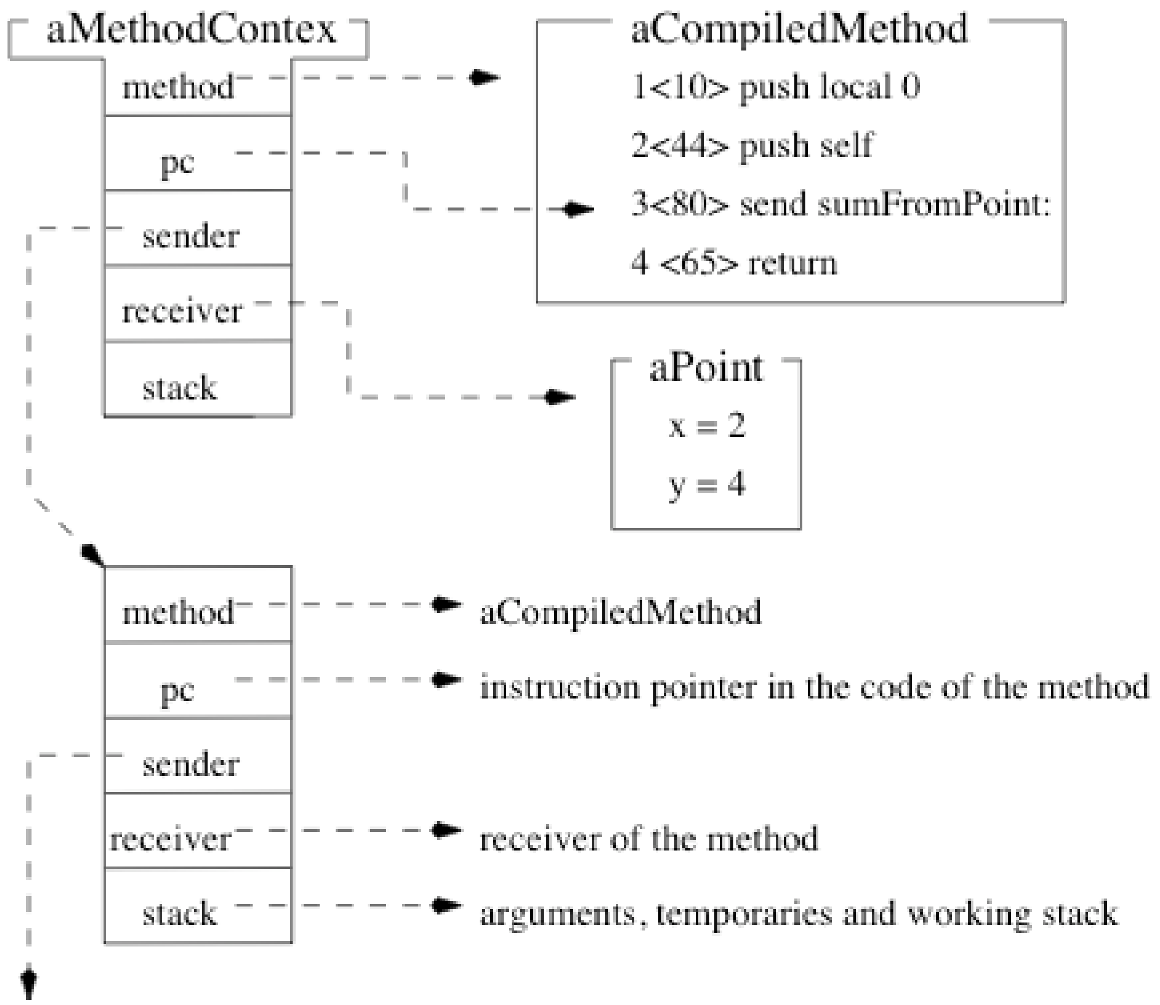
```
ContextPart variableSubclass: #MethodContext
    instanceVariableNames: 'method receiverMap receiver'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Kernel-Methods'
```

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 Context
- The chain of senders is a stack
- It grows and shrinks with activation/return

Contexts: Stack Reification



Example: #haltIf:

- 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

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

Controling Messages

Approaches to Control Message

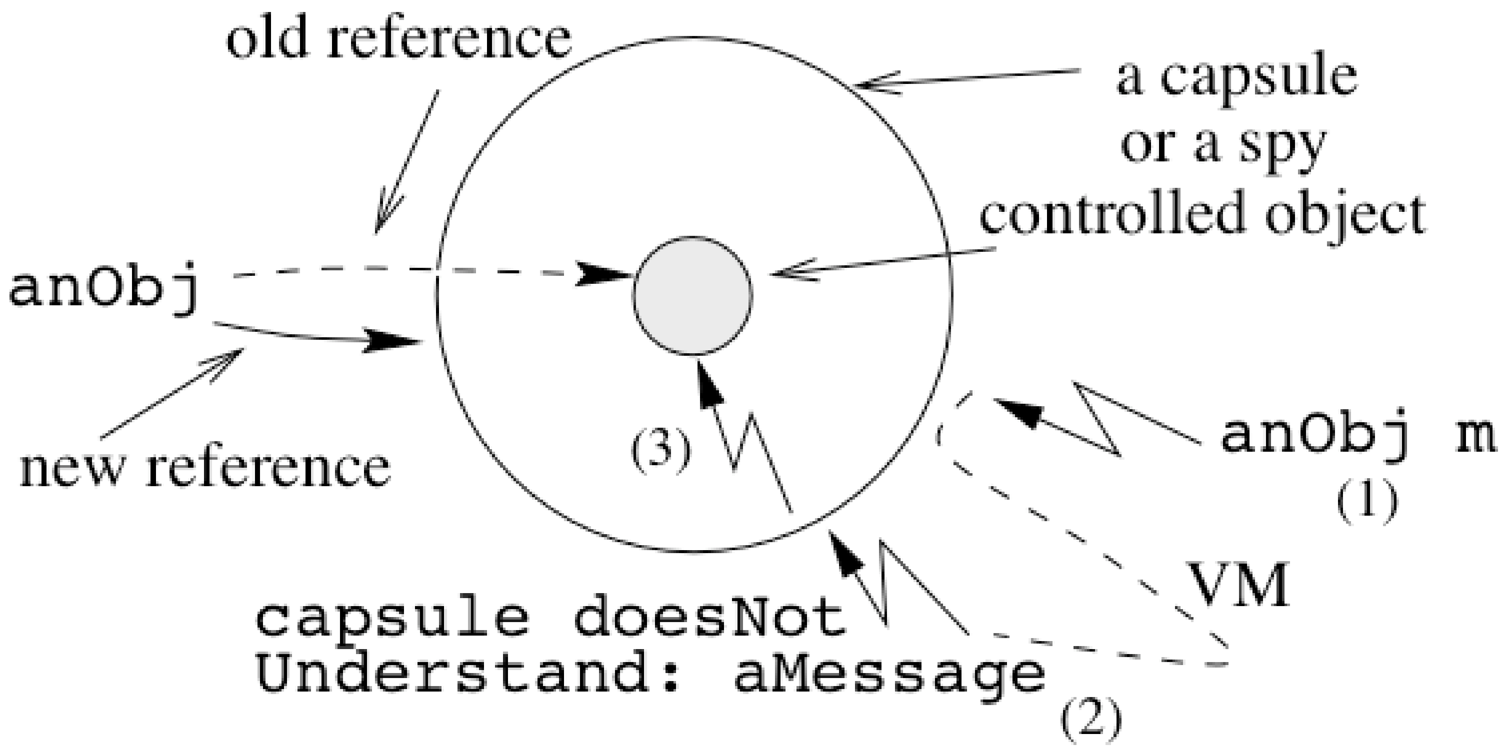
- **Error Handling Specialization**
  - Minimal Objects + doesNotUnderstand:
- Using Method Lookup
  - anonymous classes between instances and their classes
- Method Substitution
  - wrapping methods

- Control: instance-based/class/group
- Granularity: all/unknown/specific

Error Handling Specialization

- Minimal Object
  - do not understand too much
  - redefine doesNotUnderstand:
  - wrap normal object in a minimal object
- nil superclass or ProtoObject
- use becomeForward: to substitute the object to control

Minimal Object at Work



Control

- MinimalObject>>doesNotUnderstand: aMsg

```
...
originalObject perform: aMsg selector
        withArguments: aMsg arguments
....
```

Minimal Behavior in VW

```
MinimalObject class>>initialize
    superclass := nil.
    #(doesNotUnderstand: error:~~ isNil = ==
      printString printOn: class inspect basicInspect
      basicAt: basicSize instVarAt: instVarAt:put:)
          do: [:selector |
                 self recompile: selector from: Object]
```

Limits

- self problem:
  - messages sent by the object itself are not trapped
  - messages sent to a reference on it passed by the controlled object
- Class control is impossible
- Interpretation of minimal protocol:
  - message sent to the minimal object or to controlled object

Evaluation

- Simple
- In Squeak ProtoObject
- Some problems
- Instance-based
- All messages

Approaches to Control Message

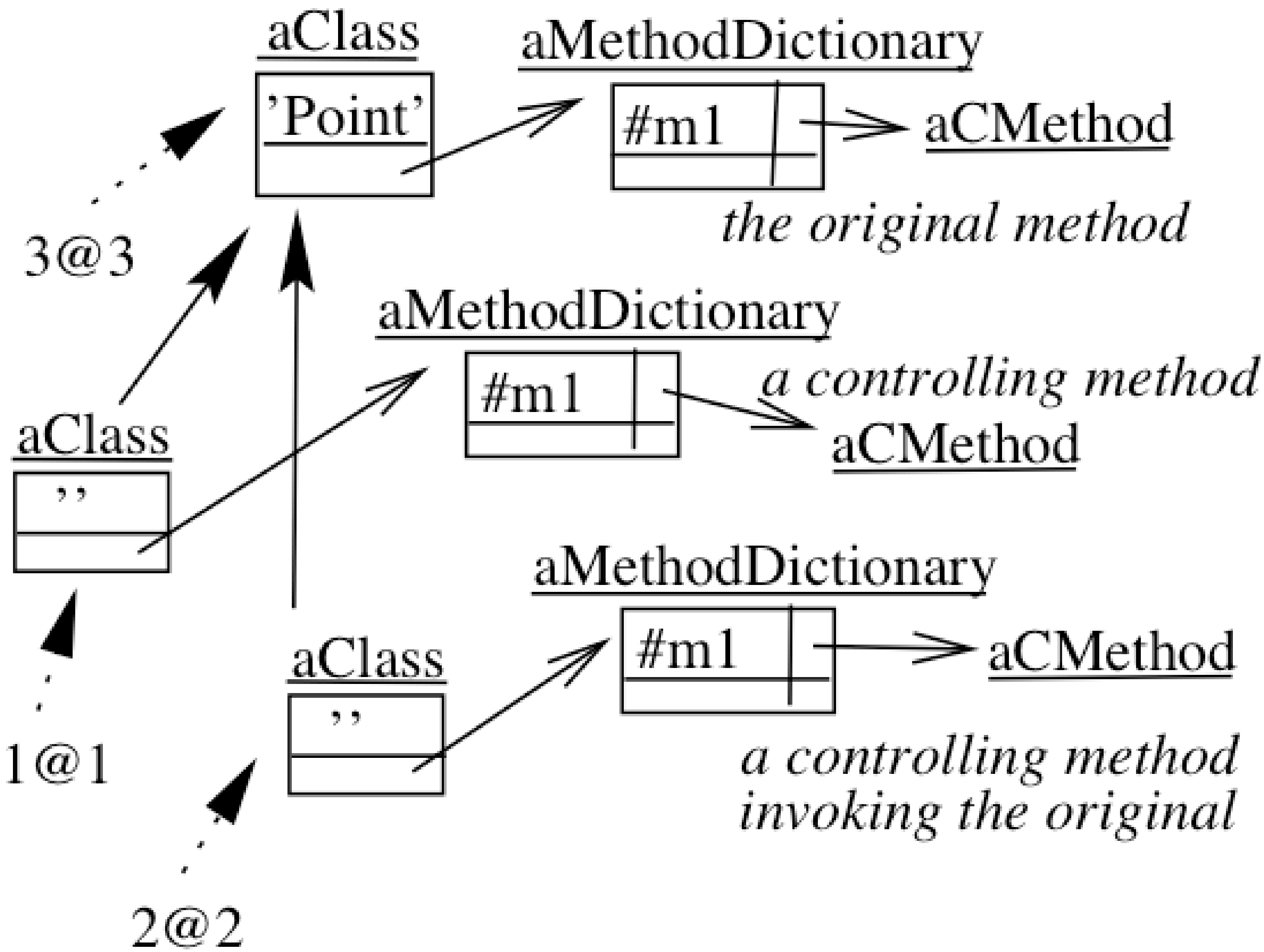
- Error Handling Specialization
  - Minimal Objects + doesNotUnderstand:
- **Using Method Lookup**
  - anonymous classes between instances and their classes
- Method Substitution
  - wrapping methods

Using VM Lookup

- Creation of a controlling class that is interposed between the instance and its class
- Definition of controlling methods
- Class change

- Hidding it from the developper/user using anonymous class

1@1, 2@2 are controlled, but not 3@3



Anonymous class in VW

```
Object>>specialize
            |nCl|
(1)         nCl :=Behavior new
(2)             setInstanceFormat: self class format;
(2)             superclass: self class;
                methodDictionary:MethodDictionary new.
(3)         self changeClassToThatOf: nCl basicNew
```

Control

```
anAnonymousClass>>setX:t1setY:t2
    ...before
    super setX:t1setY:t2
    ...after
```

The beauty in VisualWorks

```
AnonymousClass>>installEssentialMethods
    self compile: 'class ^ super class superclass'.
    self compile: 'isControlled ^ true'.
    self compile: 'anonymousClass ^ super class'
```

In Squeak class is not sent but optimized by the compiler

Evaluation

- instance-based or group-based
- selective control
- no identity problem
- good performance
- transparent to the user
- requires a bit of compilation (could be avoided using clone as in Method Wrapper)

Approaches to Control Message

- Error Handling Specialization
  - Minimal Objects + doesNotUnderstand:
- Using Method Lookup
  - anonymous classes between instances and their classes
- **Method Substitution**
  - wrapping methods

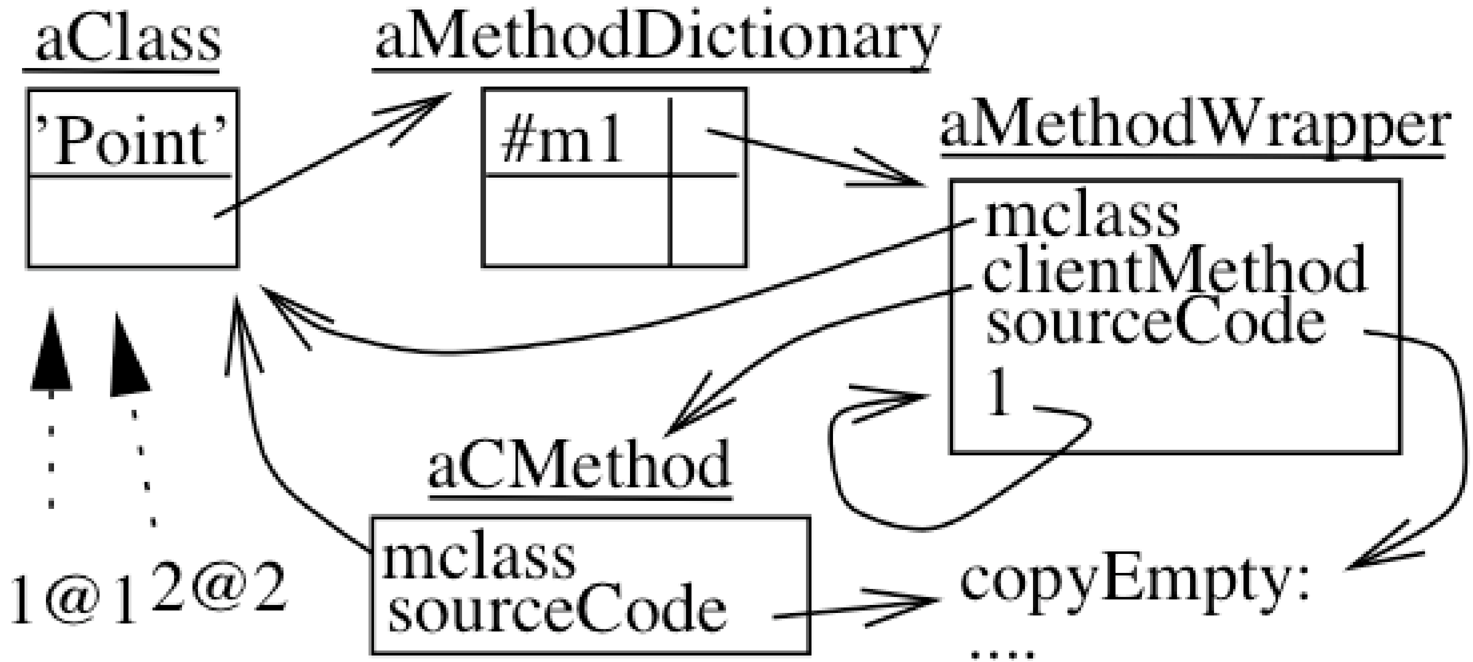
Method Substitution

- First approach: add methods with offucasted names
  - but the user can see them
- Wrapping the methods without poluting the interface

MethodWrapper Definition

```
CompiledMethod variableSubclass: #MethodWrapper
    instanceVariableNames: 'clientMethod selector'
    classVariableNames: ''
    poolDictionaries:''
    category: 'Method Wrappers'

(MethodWrapper on: #color inClass: Point) install
```

Method Wrappers: The Idea



Mechanics

```
WrapperMethod>>valueWithReceiver: anObject arguments: args
    self beforeMethod.
     ^ [clientMethod
          valueWithReceiver: object
          arguments: args]
             valueNowOrOnUnwindDo:
                 [self afterMethod]

aClass>>originalSelector: t1
    |t2|
    (t2 := Array new: 1) at: 1 put: t1.
    ^self valueWithReceiver: self arguments: t2
```

Evaluation

- Class based: all instances are controlled
- Only known messages
- Single method can be controlled
- Smart implementation does not require compilation for installation/removal

Scaffolding Patterns

- How to prototype applications even faster?
- Based on K. Auer Patterns

Patterns

- Extensible Attributes

- Artificial Delegation
  - How do you prepare for additional delegated operations?
- Cached Extensibility
- Selector Synthesis

Extensible Attributes

Context:

- multi person project + heavy version control
- other designers will want to add attributes to your class

How do you minimize the effort required to add additional attributes to the class?

Solution:

Add a dictionary attribute to your class

+ a dictionary access

Extensible Attributes

```
anExtensibleObject attributes at: #attName put: value

value := anExtensibleObject attributes at: #attName
```

Artificial Accessors

Context: you applied Extensible Attributes

How do you make it easier for other classes to access your extended attributes?

Solution: simulate the presence of accessor for the attributes by specializing doesNotUnderstand:

Artificial Accessors Code

anExtensibleObject widgets: 4

is converted to

self attributes at: #widgets put: 4

anExtensibleObject widgets

is converted to

^ self attributes at: #widgets

Consequences

Accessors do not exist therefore

- browsing can be a problem
- tracing also
- reflective queries (allSelectors, canUnderstand:....) will not work as with plain methods

Artificial Delegation

How do you make
^ self delegate anOperation
easier?

Solution: Override doesNotUnderstand: of the delegator to iterate through its attribute looking for an attribute that supports the method selector that was not understood

Cached Extensibility

Context: you used the previous patterns

How do you know which artificial accessors or artificial delegate have been used?

Solution: Specialize doesNotUnderstand: to create methods as soon as artificial ones are invoked

Selector Synthesis

How can you implement a state-dependent object with a minimal effort?

Solution: define state and event as symbols and given a pair synthesise a method selector

selector := 'handle', anEvent aString, 'In', aState asString.
self perform: selector asSymbol.

References

- [Ducasse'99] S. Ducasse, "Message Passing Control Techniques in Smalltalk", JOOP, 1999
- [Rivard'96] F. Rivard, Smalltalk : a Reflective Language, REFLECTION'96,1996
- [Bran'98] Wrappers To The Rescue, ECOOP'98, 1998
- [Auer] Scaffolding patterns, PLOD 3, Addison-Wesley, (http://www.rolemodelsoftware.com/moreAboutUs/publications/articles/scaffold.php)
- Smalltalk the Language, Golberg Robson, Addison-Wesley

Smalltalk Reflective Capabilities

- Both introspection and reflection
- Powerful
- Based on everything is an object approach