Elements of Design

- Instance initialization
- Enforcing the instance creation
- Instance / Class methods
- Instance variables / Class instance variables
- Class initialization
- Law of Demeter
- Factoring Constants
- Abstract Classes
- Template Methods
- Delegation
- Bad Coding Style

Instance Initialization

Instance Initialization

- How to ensure an instance is well initialized?
  - Automatic initialize
  - Lazy initialize
  - Proposing the right interface
  - Providing default value

A First Implementation of Packet

```
Object subclass: #Packet
  instanceVariableNames: 'contents addressee originator '
  ...
  category: 'Lan-Simulation'

One instance method
Packet>>printOn: aStream
  super printOn: aStream.
  aStream nextPutAll: ' addressed to: '; nextPutAll: self addressee.
  aStream nextPutAll: ' with contents: '; nextPutAll: self contents

Some Accessors
Packet>>addressee
  ^addressee
Packet>>addressee: aSymbol
  addressee := aSymbol
```

Packet CLASS Definition

- Packet class is Automatically defined

```
Packet class
        instanceVariableNames: ''
```

- Example of instance creation

```
Packet new
        addressee: # mac ;
        contents: 'hello mac'
```

Fragile Instance Creation

If we do not specify a contents, it breaks!

```
|p|
p := Packet new addressee: #mac.
p printOn: aStream -> error
```

- Problems of this approach:
  - responsibility of the instance creation relies on the clients
  - can create packet without contents, without address
  - instance variable not initialized -> error (for example, printOn:) -> system fragile

Fragile Instance Creation Solutions

- Automatic initialization of instance variables
- Proposing a solid interface for the creatio
- Lazy initialization

Assuring Instance Variable Initialization

- Kind of Smalltalk library mistake
- *Problem*: By default #new class method returns instance with uninitialized instance variables. Moreover, #initialize method is not automatically called by creation methods #new/new:
- How to initialize a newly created instance ?

The New/Initialize Couple

Define an instance method that initializes the instance variables and override #new to invoke it.

```
1       Packet class>>new                    Class Method
                ^ super new initialize

3       Packet>>initialize                   Instance Method
                super initialize.
4               contents := 'default message'
```

Packet new (1-2) -> aPacket initialize (3-4) -> returning aPacket but initialized!

Reminder: You cannot access instance variables from a class method like #new

The New/Initialize Couple

```
Object>>initialize
  "do nothing. Called by new my subclasses
  ovrride me if necessary"
  ^ self
```

Strengthen Instance Creation Interface

- *Problem*: A client can still create aPacket without address.

- *Solution*: Force the client to use the class interface creation.
- Providing an interface for creation and avoiding the use of #new

```
Packet send: 'Hello mac' to: #Mac
```

- *First try*:

```
Packet class>>send: aString to: anAddress
    ^ self new
        contents: aString ;
        addressee: anAddress
```

Example of Other Instance Initialization

- step 1.

```
SortedCollection sortBlock: [:a :b| a name < b name]
```

```
SortedCollection class>>sortBlock: aBlock
  "Answer a new instance of SortedCollection such that its
  elements are sorted according to the criterion specified
  in aBlock."
  ^self new sortBlock: aBlock
```

- step 2. self new = aSortedCollection
- step 3. aSortedCollection sortBlock: aBlock
- step 4. returning the instance aSortedCollection

Another Example

- step 1. OrderedCollection with: 1

```
Collection class>>with: anObject
   "Answer a new instance of a Collection
  containing anObject."
  | newCollection |
   newCollection := self new.
   newCollection add: anObject.
   ^newCollection
```

Lazy Initialization

- When some instance variables are:
  - not used all the time
  - consuming space, difficult to initialize because depending on other
  - need a lot of computation
- Use lazy initialization based on accessors
- Accessor access should be used consistently!

Lazy Initialization Example

- A lazy initialization scheme with default value

```
Packet>>contents
        contents isNil
                ifTrue: [contents := 'no contents']
        ^ contents

aPacket contents or self contents
```

- A lazy initialization scheme with computed value

```
Dummy>>ratioBetweenThermonuclearAndSolar
  ratio isNil
        ifTrue: [ratio := self heavyComputation]
  ^ ratio
```

Providing a Default Value

```
OrderedCollection variableSubclass: #SortedCollection
  instanceVariableNames: 'sortBlock '
  classVariableNames: 'DefaultSortBlock '

SortedCollection class>>initialize

  DefaultSortBlock := [:x :y | x <= y]

SortedCollection>>initialize
  "Set the initial value of the receiver's sorting
  algorithm to a default."

  sortBlock := DefaultSortBlock
```

Providing a Default Value

```
SortedCollection class>>new: anInteger
  "Answer a new instance of SortedCollection. The
  default sorting is a <= comparison on elements."

  ^(super new: anInteger) initialize

SortedCollection class>>sortBlock: aBlock
  "Answer a new instance of SortedCollection such that
  its elements are sorted according to the criterion
  specified in aBlock."

  ^self new sortBlock: aBlock
```

Invoking per Default the Creation Interface

```
OrderedCollection class>>new
        "Answer a new empty instance of
        OrderedCollection."

        ^self new: 5
```

Forbidding #new ?

- *Problem*: We can still use #new to create fragile instances
- *Solution*: #new should raise an error!

```
Packet class>>new
        self error: 'Packet should only be created using send:to:'
```

Forbidding #new Implications

But we still have to be able to create instance!

```
Packet class>>send: aString to: anAddress
        ^ self new
                contents: aString ;
                addressee: anAddress
```

-> raises an error

```
Packet class>>send: aString to: anAddress
        ^ super new
                contents: aString ;
                addressee: anAddress
```

-> BAD STYLE: link between class and superclass dangerous in case of evolution

Forbidding #new

- *Solution*: use #basicNew and #basicNew:

```
Packet class>>send: aString to: anAddress
        ^ self basicNew
                contents: aString ;
                addressee: anAddress
```

- Never override basic* methods

Different Self/Super

- Do not invoke a super with a different method selector. It's bad style because it links a class and a superclass. This is dangerous in case the software evolves.

```
Packet class>>new
        self error: 'Packet should only be created using send:to:'

Packet class>>send: aString to: anAddress
        ^ super new contents: aString ; addressee: anAddress
```

- Use basicNew and basicNew:

```
Packet class>>send: aString to: anAddress
        ^ self basicNew contents: aString ; addressee: anAddress
```

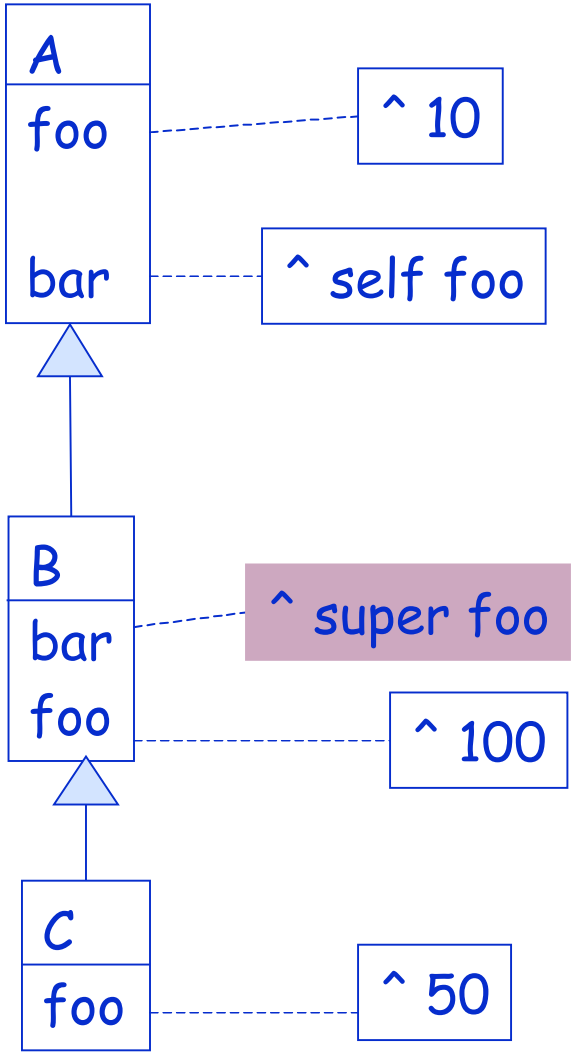
How to Reuse Superclass initialization?

```
A class>>new
  ^ super new blabla; andBlabla; andBlabla
B class>>forceClientInterface
  ^ self basicNew ???
```

=> Define the initialization behavior on the instance side

```
A>>blalaaa
  ^ self blabla; andBlabla; andBlabla
A class>>new
  ^ super new blalaaa
B class>>forceClientInterface
  ^ self basicNew blalaaa
```

Problem with m super n



With the pink box:

A new bar
-> 10
B new bar
-> 10
C new bar
-> 10

Without the pink box:

A new bar
-> 10
B new bar
-> 100
C new bar
-> 50

super shortcuts dynamic calls

Class Level Issues

Class Methods - Class Instance Variables

- Classes (Packet class) represents class (Packet).
- Class instance variables are instance variables of class
  -> should represent the state of class: number of created instances, number of messages sent, superclasses, subclasses....
- Class methods represent CLASS behavior: instance creation, class initialization, counting the number of instances....
- If you weaken the second point: class state and behavior can be used to define common properties shared by all the instances

Default value between class and instance

- Ex: If we want to encapsulate the way "no next node" is coded and shared this knowledge between class and instances.
- Instead of writing:

```
aNode nextNode isNil not =>     aNode hasNextNode
```

- Write:

```
Node>>hasNextNode
        ^ self nextNode = self noNextNode
Node>>noNextNode
        ^self class noNextNode
Node class>>noNextNode
        ^ #noNode
```

Class Initialization

- How do we know that all the class behavior has been loaded?
- At end !
- Automatically called by the system at load time or explicitly by the programmer.
- Used to initialize a classVariable, a pool dictionary or class instance variables.
- 'Classname initialize' at the end of the saved files in Squeak
- In postLoadAction: in VW

Example of class initialization

```
Magnitude subclass: #Date
  instanceVariableNames: 'day year'
  classVariableNames: 'DaysInMonth FirstDayOfMonth
        MonthNames SecondsInDay WeekDayNames'
  poolDictionaries: ''

Date class>>initialize
  "Initialize class variables representing the names of the months and
  days and the number of seconds, days in each month, and first day of each
  month. "

  MonthNames := #(January February March April May
  June July August September October November December ).
  SecondsInDay := 24 * 60 * 60.
  DaysInMonth := #(31 28 31 30 31 30 31 31 30 31 30 31 ).
  FirstDayOfMonth := #(1 32 60 91 121 152 182 213 244 274 305 335 ).
  WeekDayNames := #(Monday Tuesday Wednesday Thursday Friday Saturday
  Sunday )
```

Case Study

- Scanner

A Case Study: The Scanner class

```
Scanner new
  scanTokens: 'identifier keyword: 8r31 ''string''
  embedded.period key:word: .   '

  -> #(#identifier #keyword: 25 'string' 'embedded.period'
  #key:word: #'.')
```

- Class Definition

```
Object subclass: #Scanner
  instanceVariableNames: 'source mark prevEnd hereChar token
  tokenType saveComments currentComment buffer typeTable '
  classVariableNames: 'TypeTable '
  poolDictionaries: ''
  category: 'System-Compiler-Public Access'
```

Scanner enigma

- Why having an instance variable and a classVariable denoting the same object (the scanner table)?

- TypeTable is used to initialize once the table
- typeTable is used by every instance and each instance can customize the table (copying).

A Case Study: Scanner (II)

```
Scanner>>initialize
   "Scanner initialize"
   | newTable |
   newTable := ScannerTable new: 255 withAll: #xDefault. "default"
   newTable atAllSeparatorsPut: #xDelimiter.
   newTable atAllDigitsPut: #xDigit.
   newTable atAllLettersPut: #xLetter.
   newTable at: $_ asInteger put: #xLetter.
   '!%&*+,-/<=>?@\~' do: [:bin | newTable at: bin asInteger put: #xBinary].
   "Other multi-character tokens"
   newTable at: $" asInteger put: #xDoubleQuote.
   ...
   "Single-character tokens"
   newTable at: $( asInteger put: #leftParenthesis.
   ...
   newTable at: $^ asInteger put: #upArrow.  "spacing circumflex, formerly up
   arrow"
   newTable at: $| asInteger put: #verticalBar.
   TypeTable := newTable
```

A Case Study: Scanner (III)

- Instances only access the type table via the instance variable that points to the table that has been initialized once.

```
Scanner class>> new
        ^super new initScanner
Scanner>>initScanner
        buffer := WriteStream on: (String new: 40).
        saveComments := true.
        typeTable := TypeTable
```

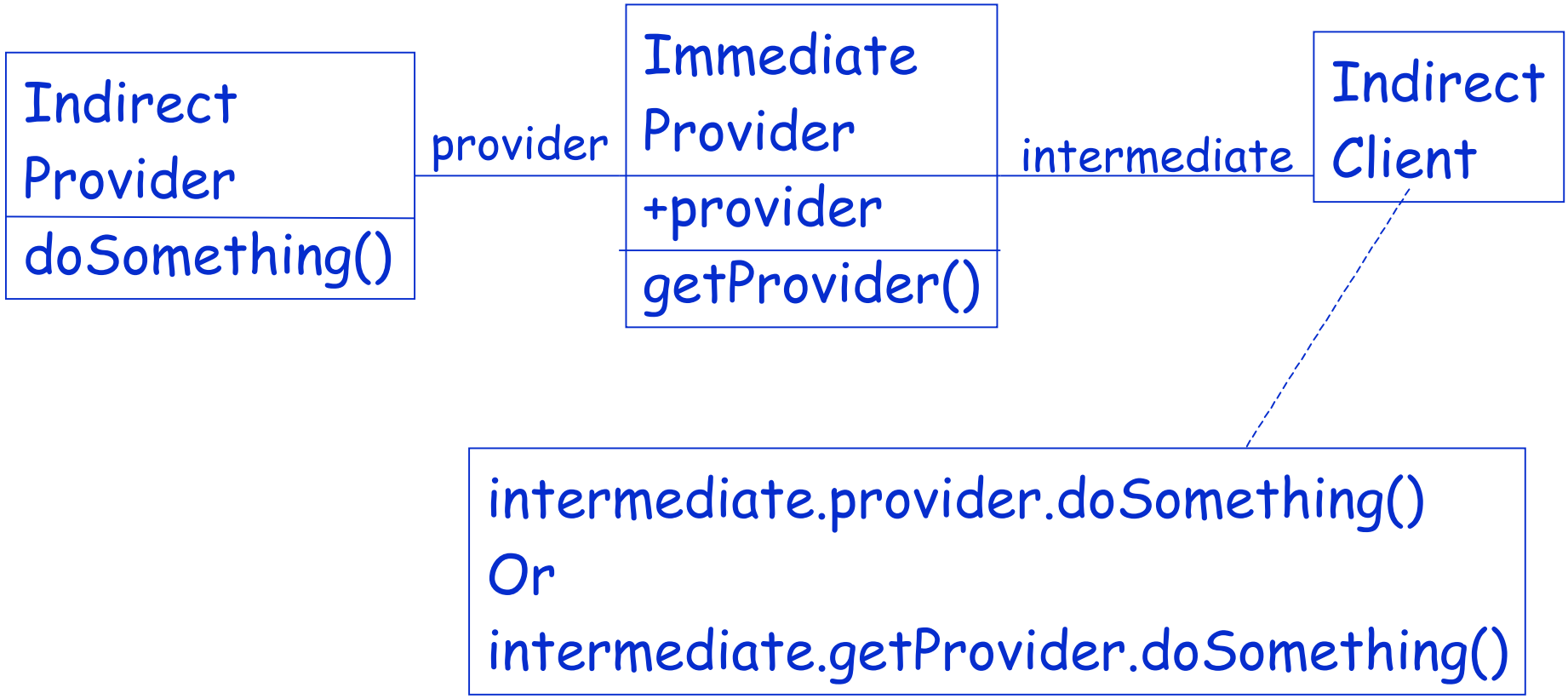
- A subclass just has to specialize initScanner without copying the initialization of the table

```
MyScanner>>initScanner
        super initScanner
        typeTable := typeTable copy.
        typeTable at: $( asInteger put: #xDefault.
        typeTable at: $) asInteger put: #xDefault.
```

Coupling

- Why coupled classes is fragile design?
- Law of Demeter
- Thoughts about accessor use

The Core of the Problem



Why are Coupled Classes bad?

```
Packet>>addressee
  ^addressee

Workstation>>accept: aPacket
        aPacket addressee = self name
                ifTrue:[ Transcript show: 'A packet  is accepted
  by the Workstation ', self name asString]
                ifFalse: [super accept: aPacket]
```

If Packet changes the way addressee is represented, Workstation, Node, PrinterServer have to be changed too

The Law ot Demeter

- You should only send messages to:
  - an argument passed to you
  - an object you create
  - self, super
  - your class
- Avoid global variables
- Avoid objects returned from message sends other than self

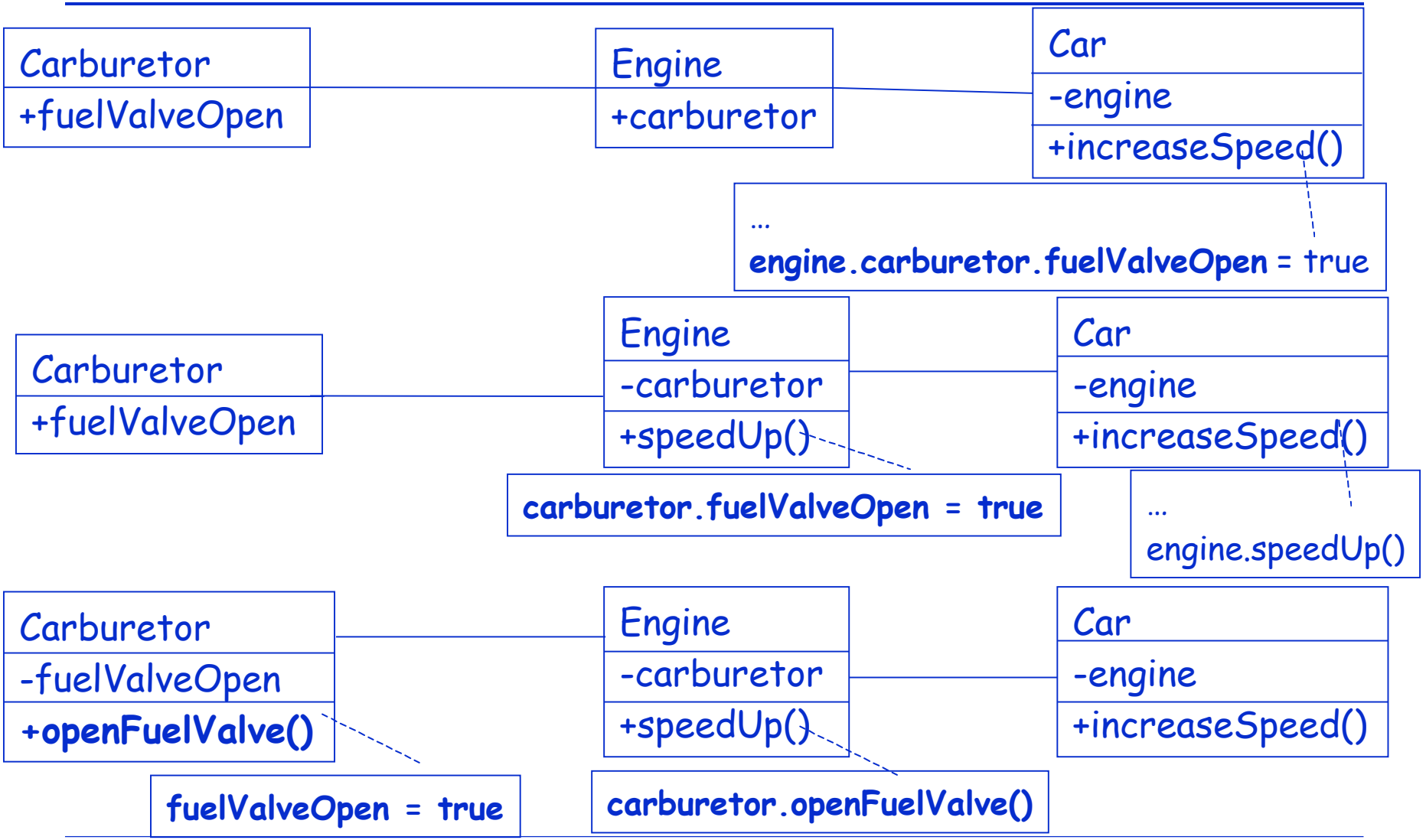
Correct Messages

```
someMethod: aParameter
  self foo.
  super someMethod: aParameter.
  self class foo.
  self instVarOne foo.
  instVarOne foo.
  self classVarOne foo.
  classVarOne foo.
  aParameter foo.
  thing := Thing new.
  thing foo
```

Law of Demeter by Example

```
NodeManager>>declareNewNode: aNode
  |nodeDescription|
  (aNode isValid)                    "Ok passed as an argument to me"
        ifTrue: [ aNode certified].
  nodeDescription := NodeDescription for: aNode.
  nodeDescription localTime.                    "I created it"
  self addNodeDescription: nodeDescription.
                                    "I can talk to myself"
  nodeDescription data                    "Wrong I should not know"
        at: self creatorKey               "that data is a dictionary"
        put: self creator
```

Transformation



Law of Demeter's Dark Side

```
Class A
  instVar: myCollection
A>>do: aBlock
  myCollection do: aBlock
A>>collect: aBlock
  ^ myCollection collect: aBlock
A>>select: aBlock
  ^ myCollection select: aBlock
A>>detect: aBlock
  ^ myCollection detect: aBlock
A>>isEmpty
  ^ myCollection isEmpty
.....................
```

About the Use of Accessors

- Literature says: "Access instance variables using methods"

But

- Be consistent inside a class, do not mix direct access and accessor use
- First think accessors as private methods that should not be invoked by clients
- Only when necessary put accessors in accessing protocol

```
Scheduler>>initialize
        self tasks: OrderedCollection new.
Scheduler>>tasks
        ^tasks
```

Accessors

- Accessors are good for lazy initialization

```
Schedule>>tasks
        tasks isNil ifTrue: [task := ...].
        ^tasks
```

- BUT accessors methods should be PRIVATE
by default at least at the beginning

- Return consistenly the receiver or the element but not the collection (otherwise people can look inside and modify it) or return a copy of it.

Accessors Open Encapsulation

- The fact that accessors are methods doesn't provide you with a good data encapsulation.
- You could be tempted to write in a client:

```
ScheduledView>>addTaskButton
        ...
        model tasks add: newTask
```

- What's happen if we change the representation of tasks? If tasks is now an array it will break
- Take care about the coupling between your objects and provide a good interface!

```
Schedule>>addTask: aTask
        tasks add: aTask
```

About the Use of Accessors (III)

"Never do the work somebody else can do!" Alan Knight

```
XXX>>m
  total := 0.
  aPlant bilings do: [:each |
        (each status == #paid and: [each date> self startDate])
              ifTrue: [total := total + each amount]].
```

- Instead write

```
XXX>m
  total := aPlant totalBillingsPaidSince: startDate

Plant> totalBillingsPaidSince: startDate
  total := 0
   bilings do: [:each |
        (each status == #paid and: [each date>startDate])
              ifTrue: [total := total + each amount]].
  ^ total
```

Provide a Complete Interface

```
Workstation>>accept: aPacket
        aPacket addressee = self name
                ...
```

- It is the responsibility of an object to propose a complete interface that propects itself from client intrusion.
- Shift the responsibility to the Packet object

```
Packet>>isAddressedTo: aNode
  ^ addressee = aNode name

Workstation>>accept: aPacket
        (aPacket isAddressedTo: self)
                ifTrue:[ Transcript show: 'A packet  is accepted by the
  Workstation ', self name asString]
                ifFalse: [super accept: aPacket]
```

Say once and only once

Factoring Out Constants

- Ex: We want to encapsulate the way "no next node" is coded. Instead of writing:

```
Node>>nextNode
        ^ nextNode
NodeClient>>transmitTo: aNode
        aNode nextNode = 'no next node'
                ...
```

- Write:

```
NodeClient>>transmitTo: aNode
        aNode hasNextNode
                ....
Node>>hasNextNode
        ^ (self nextNode = self class noNextNode) not

Node class>>noNextNode
        ^ 'no next node'
```

Initializing without Duplicating

```
Node>>initialize
        accessType := 'local'
        ...
Node>>isLocal
        ^ accessType = 'local'
```

- It's better to write

```
Node>>initialize
        accessType := self localAccessType

Node>>isLocal
        ^ accessType = self localAccessType

Node>>localAccessType
        ^ 'local'
```

Say something only once

- Ideally you could be able to change the constant without having any problems.
- You may have to have mapping tables from model constants to UI constants or database constants.

Constants Needed at Creation Time

- Previous solution works well for:

```
Node class>>localNodeNamed: aString
        |inst|
        inst := self new.
        inst name: aString.
        inst type: inst localAccessType
```

- If you want to have the following creation interface

```
Node class>>name: aString accessType: aType
        ^self new name: aString ; accessType: aType
Node class>>name: aString
        ^self name: aString accessType: self localAccessType
```

Constants Needed at Creation Time

- You need:

```
Node class>>localAccessType
        ^ 'local'
```

- -> Factor the constant between class and instance level

```
Node>>localAccessType
        ^self class localAccessType
```

- -> You could also use a ClassVariable that is shared between a class and its instances.

How to invoke a method

- Depending on **both** the receiver and an argument…
  - Type check are bad
  - Use Double Dispatch

Type Checking for Dispatching

- How to invoke a method depending on the receiver and an argument?
- A not so good solution:

```
PSPrinter>>print: aDocument
        ^ aDocument isPS
                ifTrue: [self printFromPS: aDocument]
                ifFalse: [self printFromPS: aDocument asPS]
PSPrinter>>printFormPS: aPSDoc
        <primitive>
PdfPrinter>>print: aDocument
        ^ aDocument isPS
                ifTrue: [self printFromPDF: aDocument asPDF]
                ifFalse: [self printFromPDF: aDocument]

PdfPrinter>>printFormPS: aPdfDoc
        <primitive>
```

Drawbacks of Typecheck

- Adding new kinds of documents requires changes everywhere
- Adding new documents requires changes everywhere
- No dynamic (without recompilation) possibilities

Double Dispatch

- Solution: use the information given by the single dispatch and redispatch with the argument (send a message back to the argument passing the receiver as an argument)

Double Dispatch

(a) *PSPrinter>>print: aDoc*

```
aDoc printOnPSPrinter: self
```

(b) *PdfPrinter>>print: aDoc*

```
aDoc printOnPdfPrinter: self
```

(c) PSDoc>>printOnPSPrinter: aPSPrinter

```
<primitive>
```

(d) PdfDoc>>printOnPdfPrinter: **aPSPrinter**

```
aPSprinter print: self asPS
```

(e) PSDoc>>printOnPSPrinter: **aPdfPrinter**

```
aPdfPrinter print: self asPdf
```

(f) PdfDoc>>printOnPdfPrinter:aPdfPrinter

```
<primitive>
```

- Some Tests:

```
psprinter print: psdoc =>(a->c)
pdfprinter print: pdfdoc => (b->f)
psprinter print: pdfdoc => (a->d->b->f)
pdfprinter print: psdoc => (b->e->b->f)
```

Let's Step Back

- Example: Coercion between Float and Integer
- Not a really good solution:

```
Integer>>+ aNumber
        (aNumber isKindOf: Float)
                ifTrue: [ aNumber asFloat + self]
                ifFalse: [ self addPrimitive: aNumber]

Float>>+ aNumber
        (aNumber isKindOf: Integer)
                ifTrue: [aNumber asFloat + self]
                ifFalse: [self addPrimitive: aNumber]
```

- Here receiver and argument are the same, we can coerce in both senses.

Double Dispatch on Numbers

(a) *Integer>>+ aNumber*

```
^ aNumber sumFromInteger: self
```

(b) *Float>>+ aNumber*

```
^ aNumber sumFromFloat: self
```

```
(c) Integer>>sumFromInteger: anInteger
        <primitive: 40>
(d) Float>>sumFromInteger: anInteger
        ^ anInteger asFloat + self
```

(e) Integer>>sumFromFloat: **aFloat**

```
^aFloat + self asFloat
```

(f) Float>>sumFromFloat: aFloat

```
<primitive: 41>
```

Some Tests:

1 + 1: (a->c)
1.0 + 1.0: (b->f)
1 + 1.0: (a->d->b->f)
1.0 + 1: (b->e->b->f)

Double Dispatching

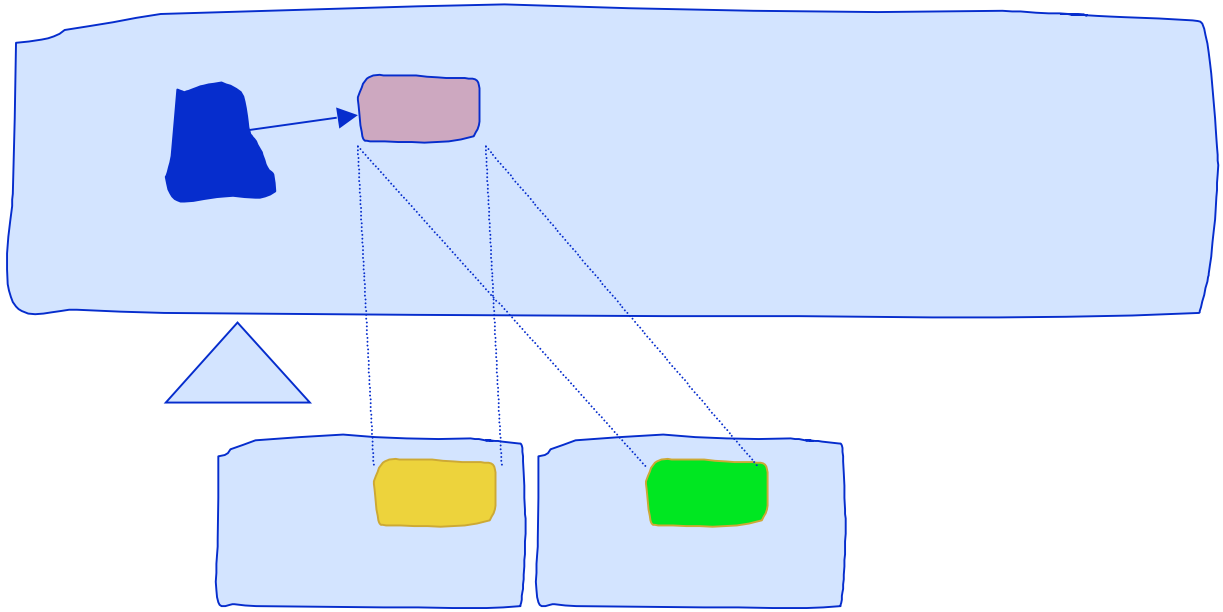
- Three Kinds of Messages
  - Primary operations
  - Double dispatching methods
  - Forwarding operations

Cost of Double Dispatching

- Adding a new class requires adding a new message to each of other classes
- Worst case is N*N methods for N classes.
- However, total *lines* of code is not much larger

Unit of Reuse

- Methods are Unit of Reuse



Methods are the Basic Units of Reuse

```
Node>>computeRatioForDisplay
  |averageRatio defaultNodeSize|
  averageRatio := 55.
  defaultNodeSize := self mainWindowCoordinate / maximiseViewRatio.
  self window add:
        (UINode new with:
              (self bandWidth * averageRatio / defaultWindowSize)
  ...
```

- We are forced to copy the method!

```
SpecialNode>>computeRatioForDisplay
  |averageRatio defaultNodeSize|
  averageRatio := 55.
  defaultNodeSize := self mainWindowCoordinate + minimalRatio /
  maximiseViewRatio.
  self window add:
        (UINode new with: (self bandWidth * averageRatio /
  defaultWindowSize)
  ...
```

Self sends: Plan for Reuse

```
Node>>computeRatioForDisplay
  |averageRatio defaultNodeSize|
  averageRatio := 55.
  defaultNodeSize := self defaultNodeSize.
  self window add:
        (UINode new with:
              (self bandWidth * averageRatio /
  defaultWindowSize)
  ...
  Node>>defaultNodeSize
        ^self mainWindowCoordinate / maximiseViewRatio

  SpecialNode>>defaultNodeSize
        ^self mainWindowCoordinate + minimalRatio /
  maximiseViewRatio
```

Do not Hardcode Constants

```
Node>>computeRatioForDisplay
   |averageRatio defaultNodeSize|
   averageRatio := 55.
   defaultNodeSize := self mainWindowCoordinate / maximiseViewRatio.
   self window add:
            (UINode new with:
                     (self bandWidth * averageRatio / defaultWindowSize).
   ...
```

- We are forced to copy the method!

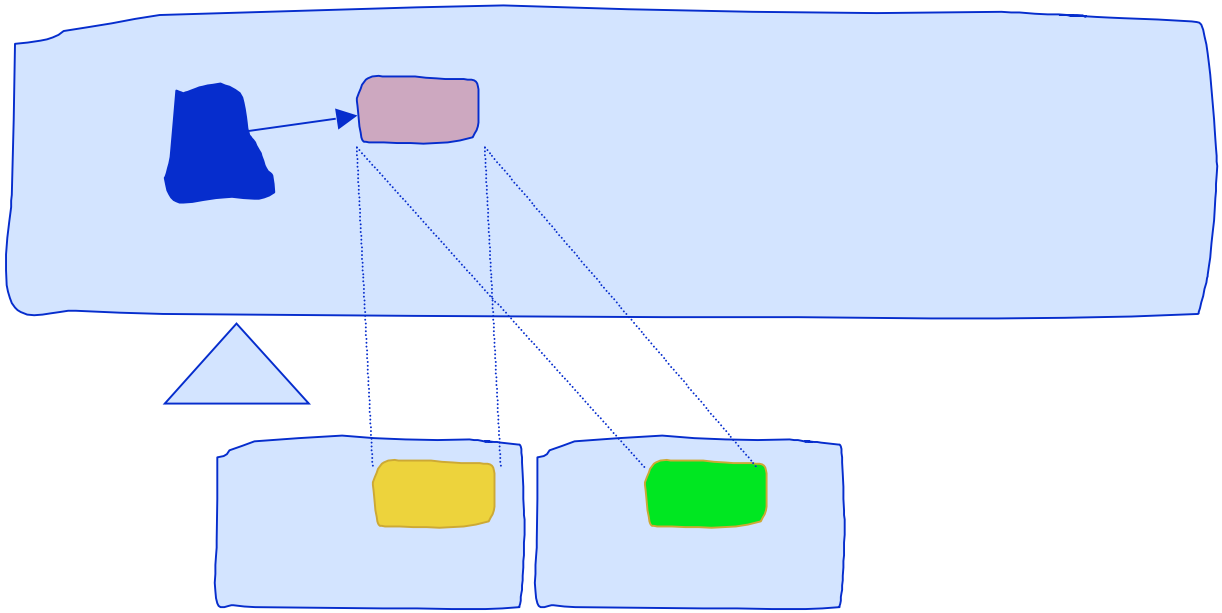
```
SpecialNode>>computeRatioForDisplay
   |averageRatio defaultNodeSize|
   averageRatio := 55.
   defaultNodeSize := self mainWindowCoordinate / maximiseViewRatio.
   self window add:
            (ExtendedUINode new with:
                     (self bandWidth * averageRatio / defaultWindowSize).
```

Class Factories

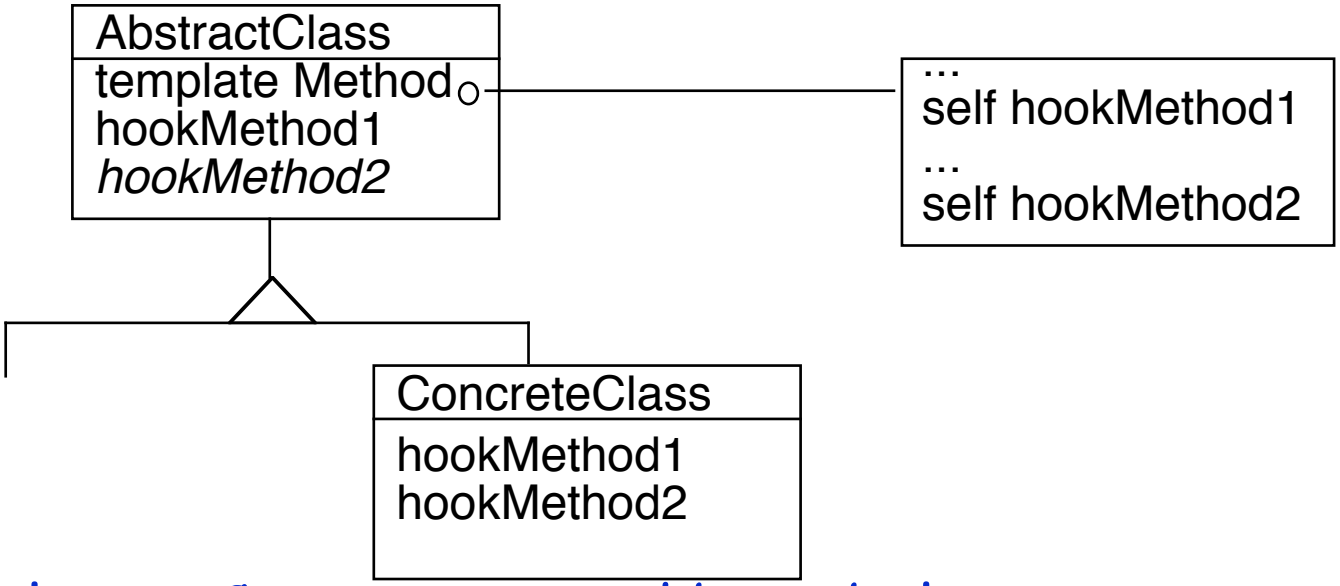
```
Node>>computeRatioForDisplay
  |averageRatio |
  averageRatio := 55.
  self window add:
        self UIClass new with:
              (self bandWidth * averageRatio / self
  defaultWindowSize)
  ...

  Node>>UIClass
        ^UINode

  SpecialNode>>UIClass
        ^ExtendedUINode
```

Hook and Template Methods

- Hooks: place for reuse
- Template: context for reuse



Hook and Template Methods



- Templates: Context reused by subclasses
- Hook methods: holes that can be specialized
- Hook methods do not have to be abstract, they may define default behavior or no behavior at all.
- This has an influence on the instantiability of the superclass.

Hook Example: Copying

```
Object>>copy
  " Answer another instance just like the receiver. Subclasses
  normally override the postCopy message, but some objects that
  should not be copied override copy. "

  ^self shallowCopy postCopy

Object>>shallowCopy
  "Answer a copy of the receiver which shares the receiver's
  instance
  variables."

  <primitive: 532>
  ....
```

postCopy

```
Object>>postCopy
  " Finish doing whatever is required, beyond a shallowCopy, to
  implement 'copy'. Answer the receiver. This message is only
  intended to be sent to the newly created instance.
  Subclasses may add functionality, but they should always do
  super postCopy first. "

  ^self
```

Hook Specialisation

```
Bag>>postCopy
  "Make sure to copy the contents fully."

  | new |
  super postCopy.
  new := contents class new: contents capacity.
  contents keysAndValuesDo:
        [:obj :count | new at: obj put: count].
  contents := new.
```

Hook and Template Example: Printing

```
Object>>printString
  "Answer a String whose characters are a description of the
  receiver."
  | aStream |
  aStream := WriteStream on: (String new: 16).
  self printOn: aStream.
  ^aStream contents

Object>>printOn: aStream
  "Append to the argument aStream a sequence of characters
  that describes the receiver."
  | title |
  title := self class name.
  aStream nextPutAll:
        ((title at: 1) isVowel ifTrue: ['an '] ifFalse: ['a ']).
  aStream print: self class
```

Overriding the Hook

```
Array>>printOn: aStream
  "Append to the argument, aStream, the elements of the Array
  enclosed by parentheses."
  | tooMany |
  tooMany := aStream position + self maxPrint.
  aStream nextPutAll: '#('.
  self do: [:element |
                  aStream position > tooMany
                          ifTrue:
                                  [aStream nextPutAll: '...(more)...)'.
                                  ^self].
                  element printOn: aStream]
          separatedBy: [aStream space].
  aStream nextPut: $)

False>>printOn: aStream
  "Print false."
  aStream nextPutAll: 'false'
```

Specialization of the Hook

- The class Behavior that represents a class extends the default hook but still invokes the default one.

```
Behavior>>printOn: aStream
  "Append to the argument aStream a statement of
  which
  superclass the receiver descends from."

  aStream nextPutAll: 'a descendent of '.
  superclass printOn: aStream
```

Guidelines for Creating Template Methods

- Simple implementation.
  - Implement all the code in one method.
- Break into steps.
  - Comment logical subparts
- Make step methods.
  - Extract subparts as methods
- Call the step methods
- Make constant methods, i.e., methods doing nothing else than returning.
- Repeat steps 1-5 if necessary on the methods created

Delegation of Responsibilities

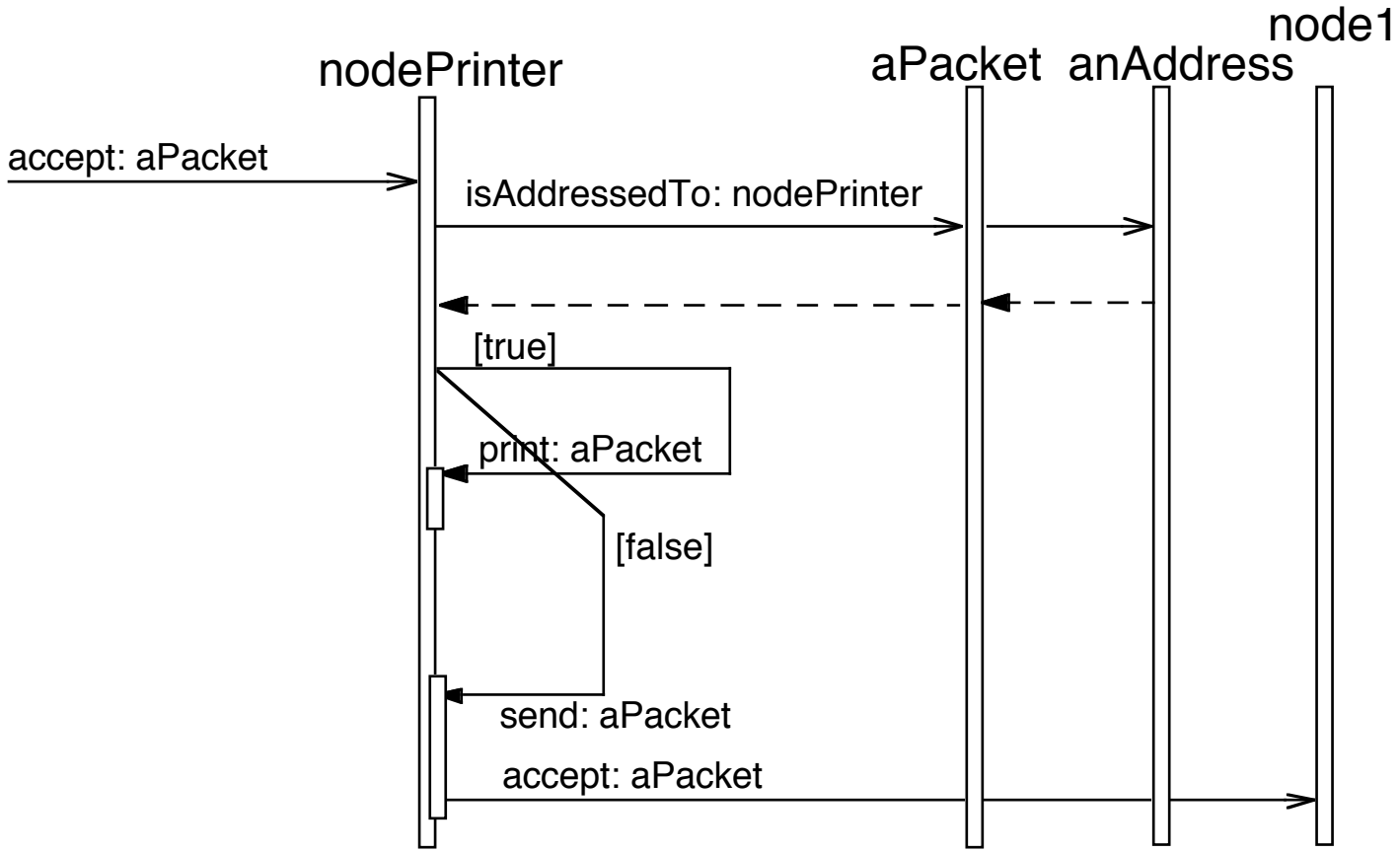
Towards Delegation: Matching Addresses

- New requirement: A document can be printed on different printers for example lw100s or lw200s depending on which printer is first encountered.
- -> Packet need more than one destination
- Ad-hoc Solution:

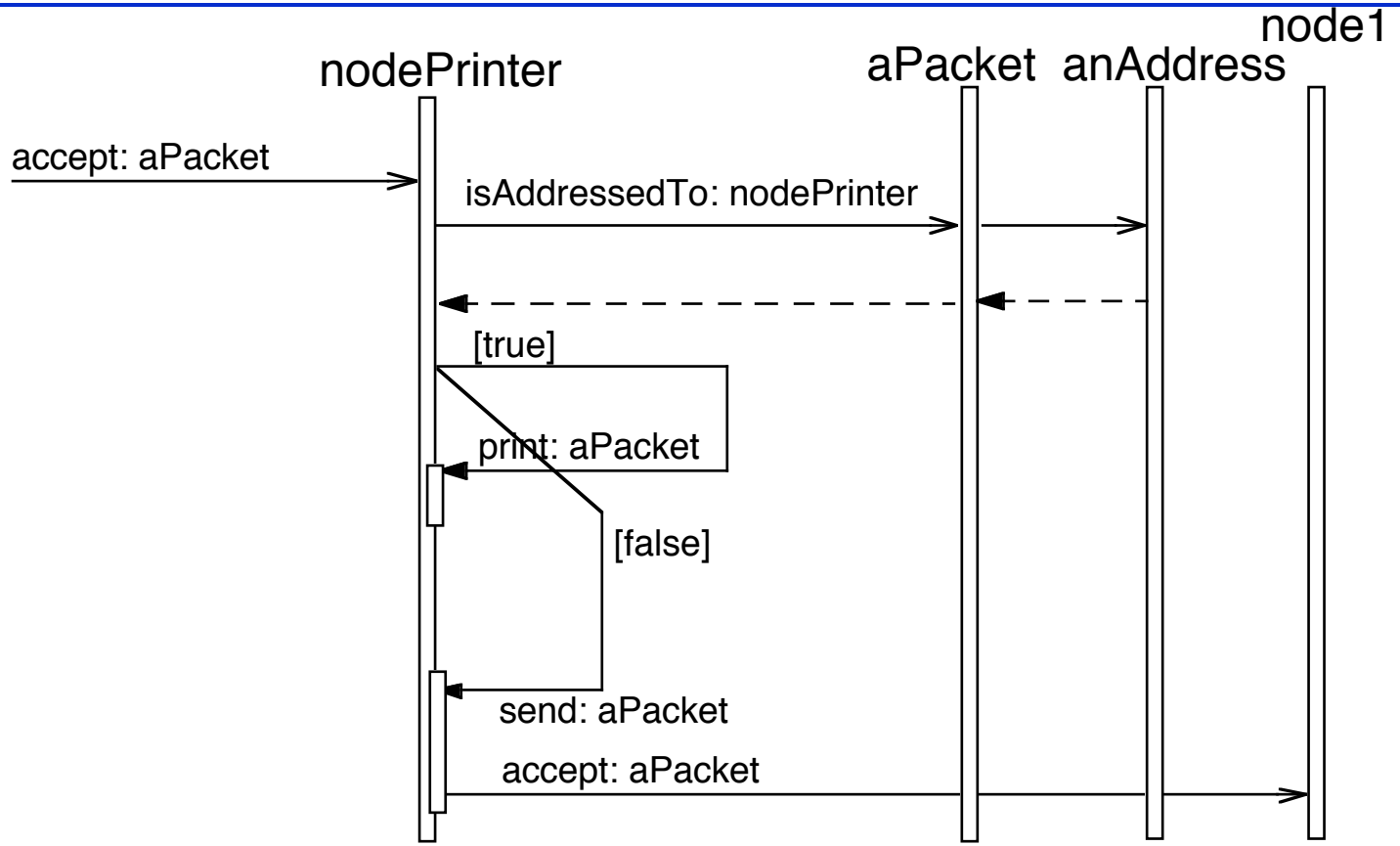
```
LanPrinter>>accept: aPacket
  (thePacket addressee = #*lw*)
        ifTrue: [ self print: thePacket]
        ifFalse: [ (thePacket isAddressedTo: self)
                        ifTrue: [self print: thePacket]
                        ifFalse: [super accept: thePacket]]
```

- Limits:
  - not general
  - brittle because based on a convention
  - adding a new kind of address behavior requires editing the class Printer

Create Object and Delegate



Create Object and Delegate



- An alternative solution: isAddressedTo: could be sent directly to the address
- With the current solution, the packet can still control the process if needed

Reifying Address

Reify: v. making something an object (philosophy)

NodeAddress is responsible for identifying the packet receivers

```
Object subclass: #NodeAddress
        instanceVariableNames: 'id'

NodeAddress>>isAddressedTo: aNodeAddress
        ^ self id = aNodeAddress id

Packet>>isAddressedTo: aNode
        ^ self addressee isAddressedTo: aNode name
```

Having the same name for packet and for address is not necessary but the name is meaningful!

Refactoring Remark: name was not a good name anyway, and now it has become an address -> we should rename it.

Matching Address

```
Address subclass: #MatchingAddress
        instanceVariableNames: ''
NodeAddress>>isAddressedTo: aNodeAddress
        ^ self id match: aNodeAddress id
```

- Works for packets with matchable addresses

```
Packet send: 'lulu' to: (MatchingAddress with: #*lw*)
```

- Does not work for nodes with matchable addresses because the match is directed. But it corresponds to the requirements!

```
Node withName: (MatchingAddress with: #*lw*)
```

```
Packet>>isAddressedTo: aNode
        ^ self addressee isAddressedTo: aNode name
```

- Remarks

inheritance class relationship is not really good because we can avoid duplication (coming soon)

Creation interfaces could be drastically improved

Addresses

```
Object subclass: #Address
  instanceVariableNames: 'id'

Address>>isAddressedTo: anAddress
  ^self subclassResponsibility

Address subclass: #NodeAddress
  instanceVariableNames: ''

Address subclass: #MatchingAddress
  instanceVariableNames: ''
```

Trade-Off

- Delegation Pros
  - No blob class: one class one responsibility
  - Variation possibility
  - Pluggable behavior without inheritance extension
  - Runtime pluggability

- Delegation Cons
  - Difficult to follow responsibilities and message flow
  - Adding new classes = adding complexities (more names)
  - New object

Designing Classes for Reuse

- Encapsulation principle: minimize data representation dependencies
- Complete interface
- No overuse of accessors
- Responsibility of the instance creation
- Loose coupling between classes
- Methods are units of reuse (self send)
- Use polymorphism as much as possible to avoid type checking
- Behavior up and state down
- Use correct names for class
- Use correct names for methods

Minor Stuff

Do not overuse conversions

```
nodes asSet
```

- removes all the duplicated nodes (if node knows how to compare). But a systematic use of asSet to protect yourself from duplicate is not good

```
nodes asSet asOrderedCollection
```

- returns an ordered collection after removing duplicates

- Look for the real source of duplication if you do not want it!

Hiding missing information

Dictionary>>at: aKey

- This raises an error if the key is not found

Dictionary>>at: aKey ifAbsent: aBlock

- This allows one to specify action <aBlock> to be done when the key does not exist. Do not overuse it:

nodes at: nodeId ifAbsent:[]

- This is bad because at least we should know that the nodeId was missing