Elements of Design

Stéphane Ducasse
Stephane.Ducasse@univ-savoie.fr
http://www.iam.unibe.ch/~ducasse/

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

- Automatic initialize
- Lazy initialize
- Proposing the right interface
- Providing default value

Provider Responsibility

- This is the responsibility of the class to provide well-formed object
- The client should not make assumptions or been responsible to send specific sequence of messages to get a working object

Instance Initialization

- How to ensure that 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 ‘

Packet>>printOn: aStream
        super printOn: aStream.
        aStream nextPutAll:‘ addressed to:‘; nextPutAll: self
   addressee.
        aStream nextPutAll:‘ with contents: ’; nextPutAll: self contents

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***

A client 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 creation
- Lazy initialization

Assuring Instance Variable Initialization

- ***Problem:*** By default ***new*** class method returns instance with uninitialized instance variables.
- Moreover, initialize method is not automatically called by creation methods new/new:.
    - Note that since Squeak 3.7 initialize is called automatically at creation time (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&2)         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
   override 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
```

Examples of 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: anAddres
	^ 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
```

Conclusion: Never override basic* methods else you will not be able to invoke them later

How to Reuse Superclass Initialization?

```
A class>>new
	^ super new doThat; andThat; end
```

```
B class>>forceClientInterface
	^ self basicNew ???
```

Solution: *Define the initialization behavior on the instance side*

```
A>>doThatAndThatEnd
	^ self doThat; andThat; end
A class>>new
```

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.

Example

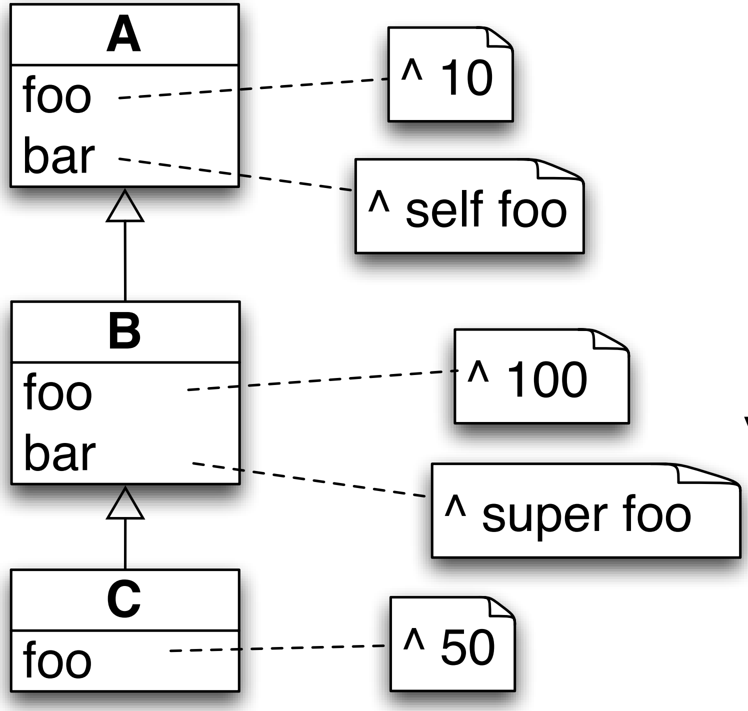
```
Packet class>>new
	self error: 'Packet should be created using send:to:'
```

```
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
```

Super is static!



With the super foo:

A new bar

-> 10

B new bar

-> 10

C new bar

-> 10

Without the super foo:

A new bar

-> 10

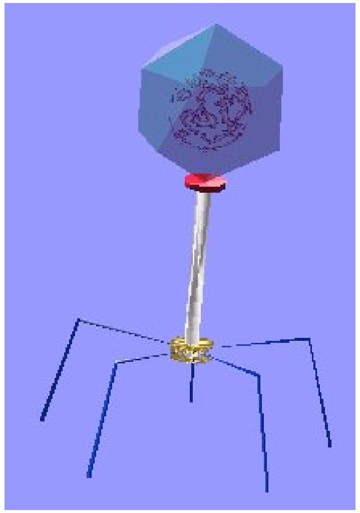
B new bar

-> 100

C new bar

-> 50

Basic Design Mistakes



A Class should have

```
Class Person {
    String getName();
    void setName(String name);
    int getAge();
    void setAge(int age);
    Car getCar();
    void setCar(Car car);
}
```

What do we see ?

- A class should have one main responsibility and some behavior not just holding state
- Minimal access to its data!

Confusing

Class City extends Place { … }
Class Jerusalem extends City implements Capital { … }
Class TelAviv extends City { … }

What is wrong here?

Confusing inheritance and instantiation
Too much inheritance?

Do not expose implementation

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

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

isNil

Avoid to return special results as nil

```
messages := self fetchMessages.
messages isNil
    ifFalse: [ messages dispatchFrom: self ]
```

What if we would simply return an empty collection in fetchMessages instead of nil?

Less conditional and ugly tests!!

Say once and only once

- No Magic Number Duplicated
- Extract method
- Remove duplicated code

Factorize Magic Numbers

Ideally you should be able to change your constants without having any impact on the code!

For that

- define a constant only once via accessor
- provide testing method (hasNextNode)
- default value using the constant accessor

Factoring Out Constants

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’
             ...
```

Factoring Out Constants

Write:

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

Node class>>noNextNode
        ^ ‘no next node’
```

Default value between class and instance

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
```

Write:

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

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
```

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

```
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.

Elements of Design

- Class initialization

Class Methods - Class Instance Variables

- Classes (Packet class) represents class (Packet).
- Class instance variables are instance variables of class
- They 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

Class Initialization

- How do we know that all the class behavior has been loaded?
- At the 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’
```

Date class>>initialize

```
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 )
```

Sharing or not

- How can I share state and prepare for instance specific state?

Case Study: Scanner

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

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

A Case Study: The Scanner class

Class Definition

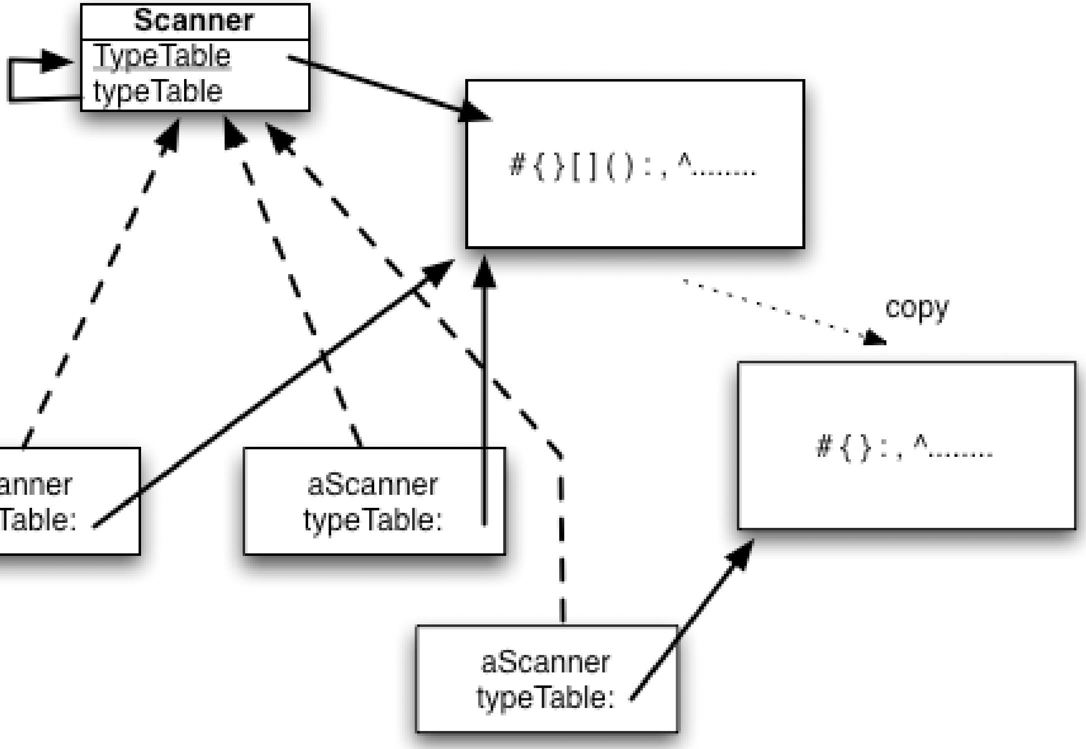
```
Object subclass: #Scanner
        instanceVariableNames: 'source mark prevEnd
    hereChar token tokenType saveComments
    currentComment buffer typeTable '
        classVariableNames: 'TypeTable '
        poolDictionaries: ''
        category: 'System-Compiler-Public Access'
```

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).

Clever Sharing



Scanner

TypeTable

typeTable

#{}[]():,^........

copy

#{}:,^........

aScanner
typeTable:

aScanner
typeTable:

aScanner
typeTable:

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.
        '!%&*+,-/<=>?@\~' 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"
```

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.
```

A Simple Case...

- Introducing parametrization

Parametrization Advantages

```
DialectStream>>initializeST80ColorTable
        "Initialize the colors that characterize the ST80 dialect"
        ST80ColorTable _ IdentityDictionary new.
        #((temporaryVariable blue italic)
           (methodArgument blue normal)
           …
           (setOrReturn black bold)) do:
              [:aTriplet |
                  ST80ColorTable at: aTriplet first put: aTriplet allButFirst]
```

- Problems:
  - Color tables ***hardcoded*** in method
  - Changes Require compilation
  - Client responsible of initialize invocation
  - No run-time changes

One Step

```
DialectStream>>initializeST80ColorTable
    ST80ColorTable := IdentityDictionary new.
    self defaultDescription do:
                [:aTriplet |
                    ST80ColorTable at: aTriplet first put: aTriplet
    allButFirst]

DialectStream>>defaultDescription
  ^ #((temporaryVariable blue italic)
      (methodArgument blue normal)
       ...
      (setOrReturn black bold))
```

Still requires subclassing and recompilation

Composition-based Solution

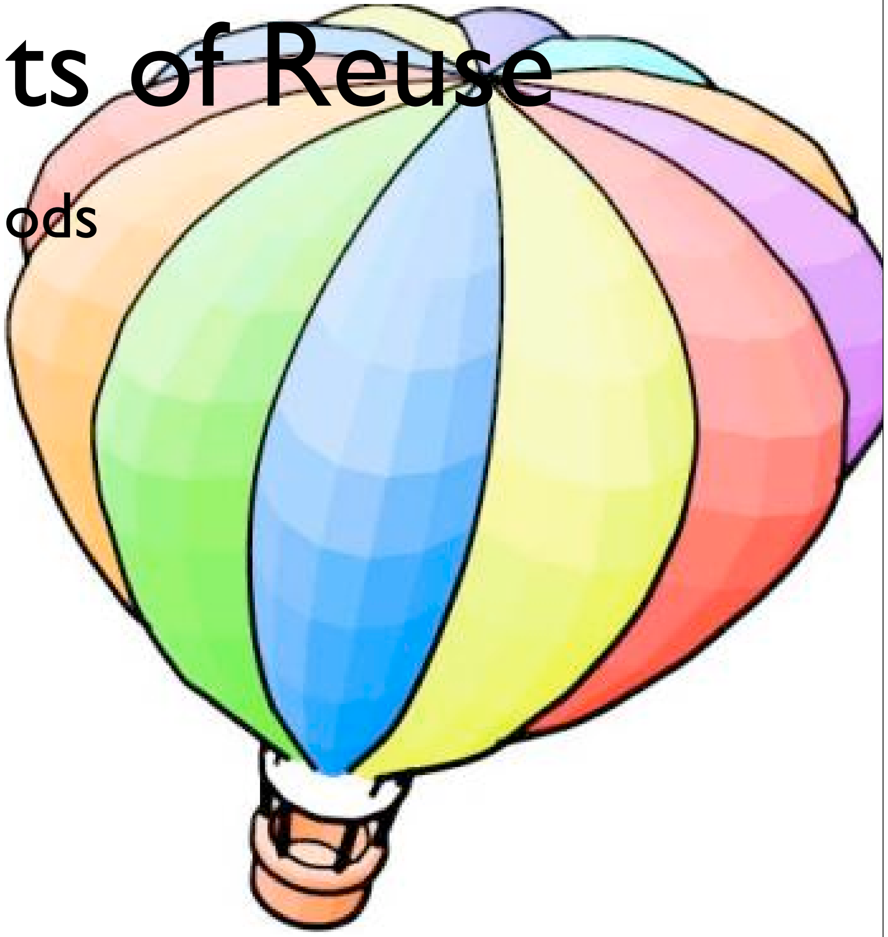
```
DialectStream>>initializeST80ColorTableWith: anArray

    ST80ColorTable := IdentityDictionary new.
    anArray
        do: [:aTriplet | ST80ColorTable at: aTriplet first
                                                put: aTriplet allButFirst].
  self initialize
```

- **In a Client**

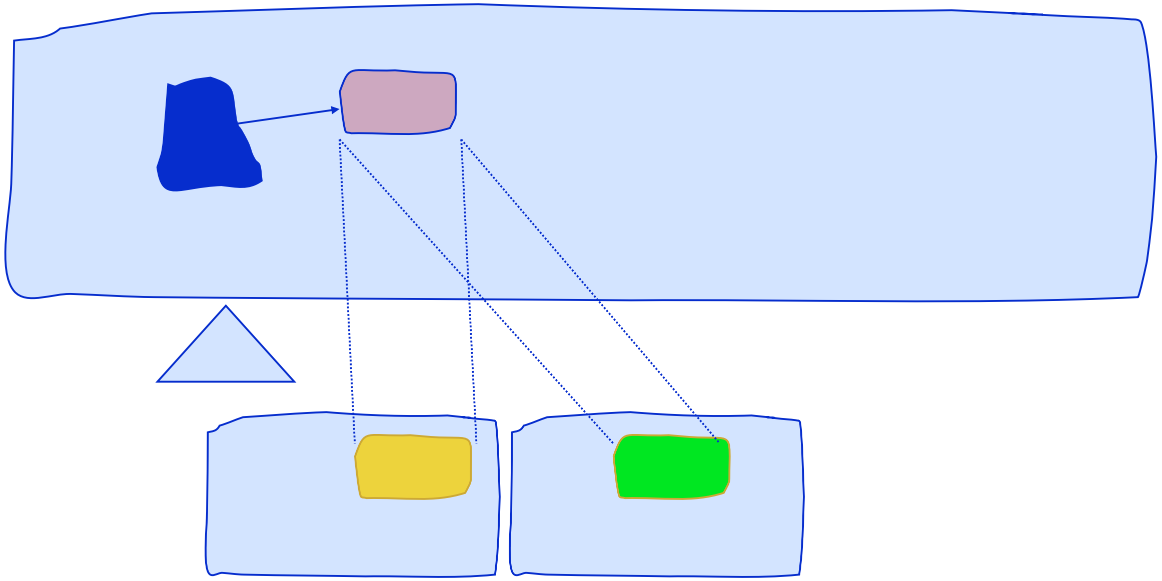
```
DialectStream initializeST80ColorTableWith:
 #(#(#temporaryVariable #blue #normal) ...
   #(#prefixKeyword #veryDarkGray #bold)
   #(#setOrReturn #red #bold) )
```

Methods are Units of Reuse

- Dynamic binding and methods
  = reuse in subclasses



Methods are Unit of Reuse



Example: Forced to Duplicate!

```
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 complete 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 / maxiViewRatio
```

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).
```

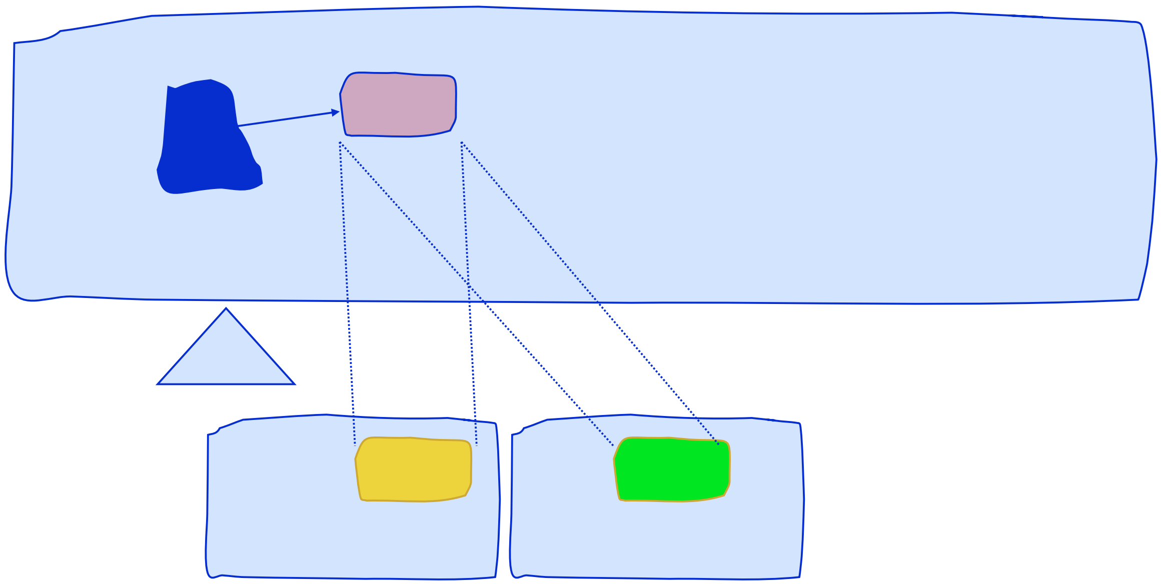
Class Factories

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

Node>>UIClass
	^ UINode

SpecialNode>>UIClass
	^ ExtendedUINode
```

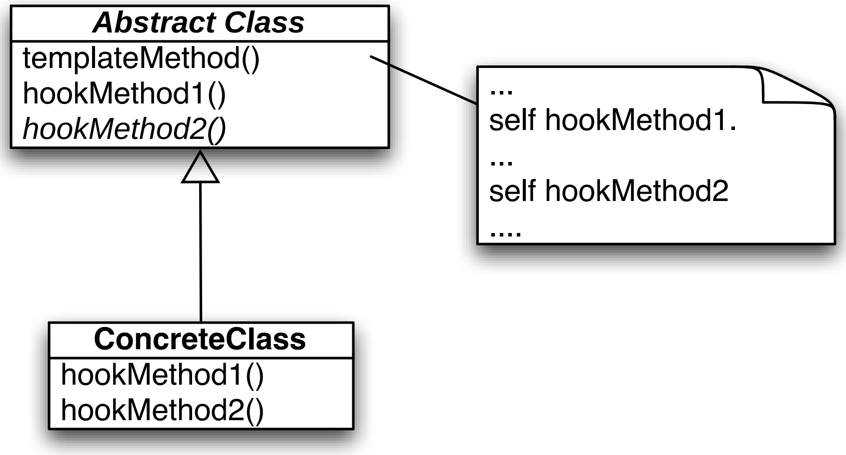
Hook and Template

Hook and Template Methods

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



Hook and Template Methods



Abstract Class
templateMethod()
hookMethod1()
hookMethod2()

...
self hookMethod1.
...
self hookMethod2
....

ConcreteClass
hookMethod1()
hookMethod2()

- ***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 / 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
```

Hook

```
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: $)
```

Overriding

```
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
```

Another Example: Copying

Complex (deepCopy, veryDeepCopy...)

Recursive objects

Graph of connected objects

Each object wants a different copy of itself

No up-front solution

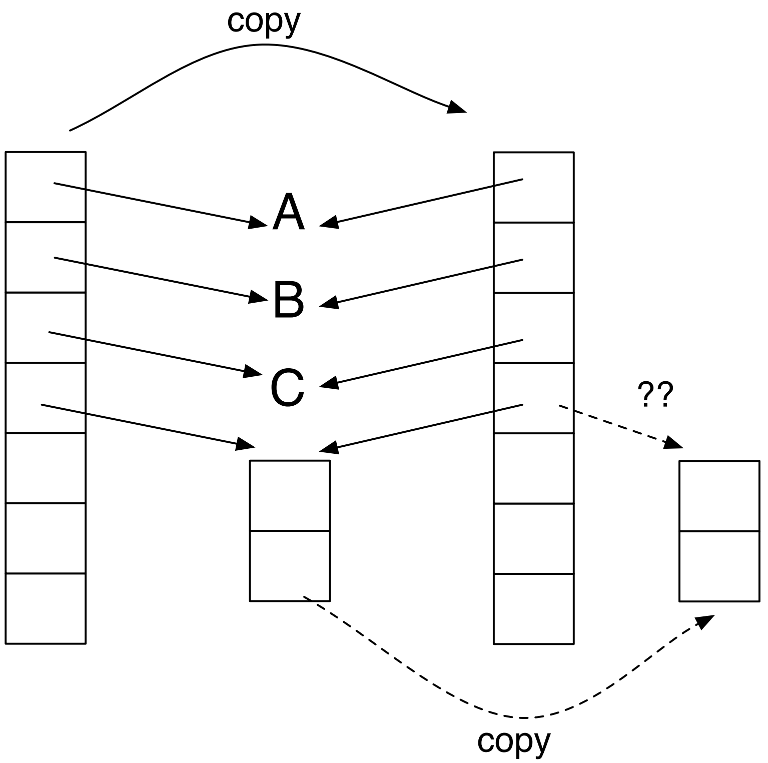
Hook Example: Copying

```
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
```

Sounds Trivial?



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.
```

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

Inheritance vs. Composition

Delegation of Responsibilities

New requirement: A document can be printed on different printers for example lw100s or lw200s depending on which printer is first encountered.

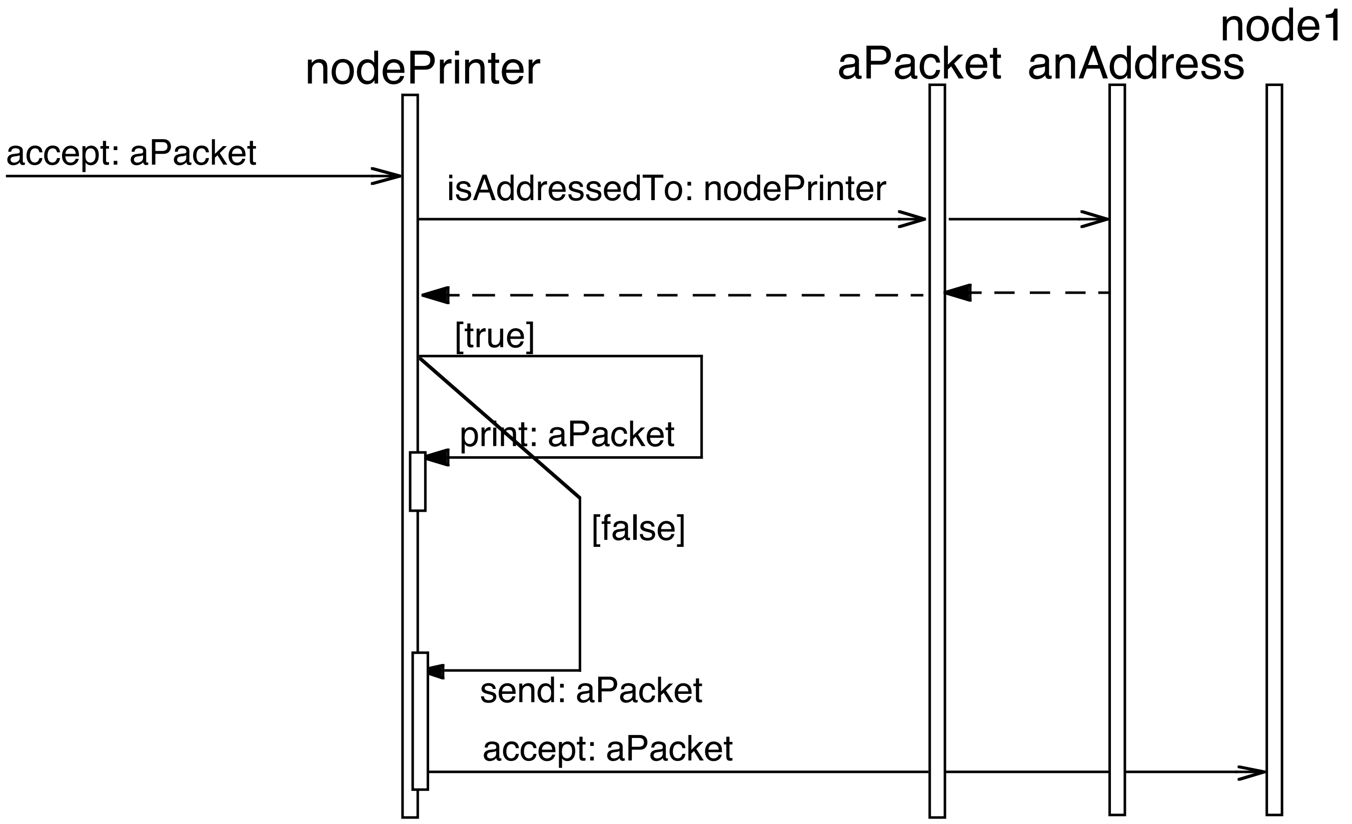
Ad-hoc Solution

```
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



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

NodeAddress

NodeAddress is responsible for identifying the packet receivers

```
Packet>>isAddressedTo: aNode
	^ self address isAddressedTo: aNode address  “was name”

Object subclass: #NodeAddress
	instanceVariableNames: ‘id‘

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

Matching Address

For packets with matchable addresses

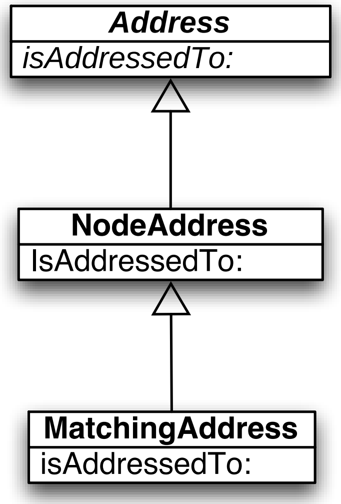
```
    Packet send: ‘lulu’ to: (MatchingAddress with: #*lw*)
```

```
Address subclass: #MatchingAddress
        instanceVariableNames: “
```

```
MatchingAddress>>isAddressedTo: aNodeAddress
        ^ self id match: aNodeAddress id
```

Addresses

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

Address>>isAddressedTo: anAddress
    ^self subclassResponsibility

Address subclass: #NodeAddress
    instanceVariableNames: “

Address subclass: #MatchingAddress
    instanceVariableNames: “
```



Trade-Off

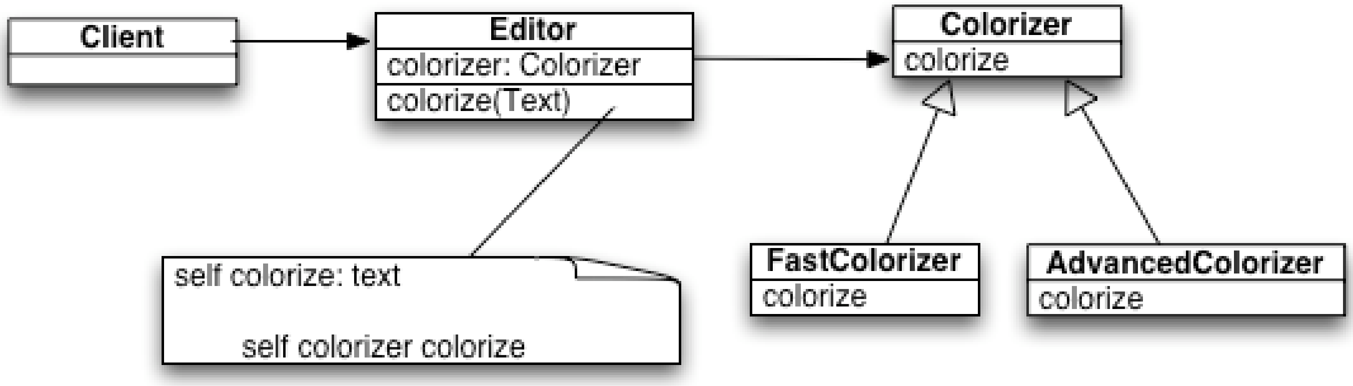
- Delegation Pros
  - No blob class: one class one responsibility
  - Variation possibility
  - Pluggable behavior without inheritance extension
  - Runtime pluggability

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

Inheritance vs. Composition

- Inheritance is not a panacea
  - Require class definition
  - Require method definition
  - Extension should be prepared in advance
  - No run-time changes
- Ex: editor with spell-checkerS, colorizerS, mail-readerS….
  - No clear responsibility
  - Code bloated
  - Cannot load a new colorizers

Delegating to other Objects



myEditor setColorizer: FastColorizer new.

myEditor setColorizer: AdvancedColorizer new.

Strategy design pattern

Composition Analysis

- Pros
  - Possibility to change at run-time
  - Clear responsibility
  - No blob
  - Clear interaction protocol
- Cons
  - New class
  - Delegation
  - New classes

Designing Classes...

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)

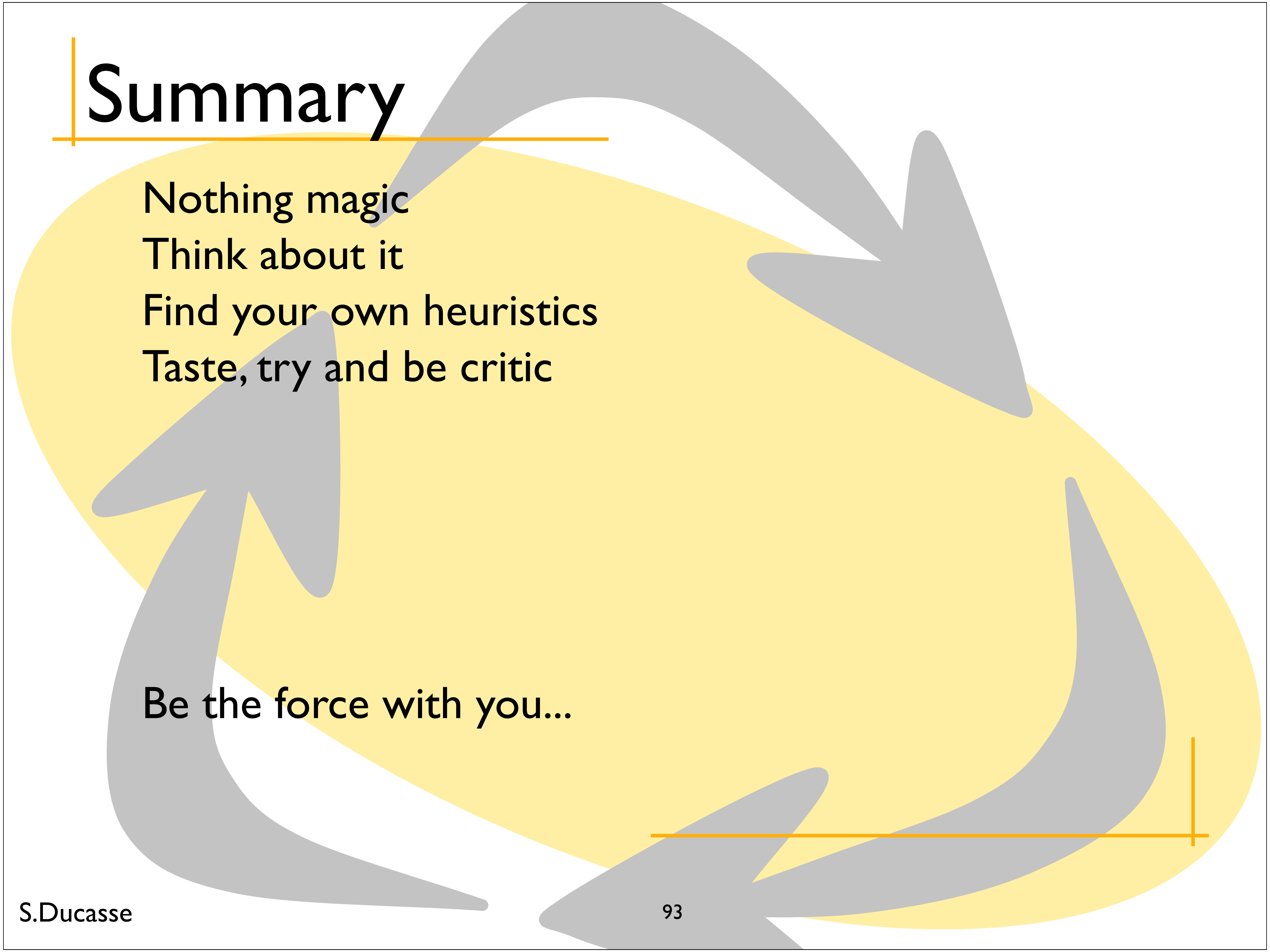
Use polymorphism as much as possible to avoid type checking

Behavior up and state down

Use correct names for class

Use correct names for methods

Summary



Nothing magic
Think about it
Find your own heuristics
Taste, try and be critic

Be the force with you...