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

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

Packet class Definition
Packet class is automatically defined
instanceVariableNames: "
Example of instance creation
Packet new
  addressee: #mac ;
  contents: 'hello mac'

Instance initialization
• Automatic initialize
• Lazy initialize
• Proposing the right interface
• Providing default value

A First Implementation of Packet
Object subclass: #Packet
instanceVariableNames: 'contents addressee originator ;'
Packet>>addressee
  super printOn: aStream
  aStream nextPutAll: 'address: '; nextPutAll: self addressee.
  aStream nextPutAll: ' with contents: '; nextPutAll: self contents.
Packet>>addressee:
  aSymbol
  Packet>>addressee: aSymbol
  addresses := aSymbol

Fragile Instance Creation
If we do not specify a contents, it breaks!
[p] printOn: aStream . -> error

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/initialize`.
- 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
   * super new initialize
   (3) `Packet>>initialize" "Instance Method"
   super initialize.
   contents := 'default message'
   (4) `Packet new (1-2) => aPacket initialize (3-4) =>`
   returning `aPacket` but initialized!
   Reminder: You cannot access instance variables from a class method like `new`

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 ; addresses: anAddress

Examples of Instance Initialization

step 1. `SortedCollection sortBlock: [:a :b| a name < b name]`
  SortedCollection class>>sortBlock: aBlock
  "Answer a new instance of a Collection containing elements 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

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

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

A lazy initialization scheme with computed value

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

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

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

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

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

```java
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

```java
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

```java
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

```java
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

```java
Dictionary>>at: aKey
```

This raises an error if the key is not found

```java
Dictionary>>at: aKey ifAbsent: aBlock
```

Allows one to specify action aBlock to be done when the key does not exist.

Do not overuse it:

```java
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:

```smalltalk
Node>>nextNode
^ self nextNode
```

```smalltalk
NodeClient>>transmitTo: aNode
aNode nextNode = 'no next node'
```

Instead of writing:

```smalltalk
Node>>hasNextNode
^ (self nextNode = self class noNextNode) not
```

Instead of writing:

```smalltalk
Node class>>name:
^ self name:
```

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

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

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

```
Node class>>noNextNode
^ #noNode
```

Instead of writing:

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

```
Node class>>noNextNode
^ self noNextNode
```

### Initializing without Duplicating

```smalltalk
Node>>initialize
accessType := 'local'
...accessType := self localAccessType
```

It's better to write:

```smalltalk
Node>>initialize
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

You need:

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

```smalltalk
Node class>>name:
^ self name:
```

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

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

### Elements of Design

- Class initialization

```
Node>>name:
^ self name:
```

### 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

Sharing or not
- How can I share state and prepare for instance specific state?

Example of class initialization
Magnitude subclass: #Date
   instanceVariableNames: 'day year'
   classVariableNames: 'DaysInMonth FirstDayOfMonth MonthNames SecondsInDay WeekDayNames'

A 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 char 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>>initialize
  "Scanner initialize"
  newTable | newTable = ScannerTable new 255 withAll: #default. "default"
  newTable addCharacterPunctuation: #default.
  newTable addDigitRules: #default.
  newTable addIdentifierRules: #default.
  "Identifiers and comments"
  newTable at: $i$ put: i. "identifier"
  newTable at: $s$ put: s. "string"
  newTable at: $. put: #verticalBar. "period"
  "Other multi-character tokens"
  newTable at: $u$ put: #spacedChar. "up arrow"
  "Pair delimiters"
  newTable at: $l$ put: #leftPar. "left parenthesis"
  "other delimiters"
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.

```small
Scanner class>> new
    super new initScanner
Scanner>> initScanner
    buffer :=ReadStream on (String new:40).
    sawComments := 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.
```

Example: Forced to Duplicate!

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

Node>> defaultNodeSize
    |defaultWindowSize := self defaultNodeSize.
    |window add:
        ( UNNode new with:
            self bandWidth * averageRatio / defaultWindowSize)
```

Methods are Unit of Reuse

• Introducing parametrization

Composition-based Solution

```small
DialectStream>> initializeST80ColorTable
    with: onArray
        ST80ColorTable := IdentityDictionary new.
        onArray
        do: [xTriplet | ST80ColorTable at: xTriplet first put: xTriplet allButFirst].

DialectStream>> defaultDescription
    "|(temporaryVariable blue italic)"
    |methodArgument blue normal|...;
    "(setParameter Return black bold)"

In a Client:

```small
DialectStream initializeST80ColorTableWith:
    "#: threadIdx #blue #normal"
    "#: #pref #keyword #everyDarkGray #bold"
    "#: #set #Return #lined #bold"
```

Parametrization Advantages

DialectStream>> initializeST80ColorTable
    "Initializes the colors that characterize the ST80 dialog"
    ST80ColorTable := IdentityDictionary new.
    (temporaryVariable blue italic)
    "(setParameter Return black bold)"
    "#: xTrplet | ST80ColorTable at: xTriplet first put: xTriplet allButFirst"

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

One Step

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

Still requires subclassing and recompilation

Methods are Units of Reuse

• Dynamic binding and methods = reuse in subclasses
**Do not Hardcode Constants**

```plaintext
Node>>computeRatioForDisplay
|averageRatio| defaultNodeSize
averageRatio := 55;
defaultNodeSize := self mainWindowSize / maxRatioViewRatio.
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 mainWindowSize / maxRatioViewRatio.
self window add:
  (ExtendedUINode new with:
    (self bandWidth + averageRatio / defaultWindowSize).
  )

**Hook and Template Methods**

- Hooks: place for reuse
- Templates: context for reuse

**Hook and Template**

- 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 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 and Template Example: Printing**

Object>>printString

```
"Answer a string whose characters are a description of the receiver."

| aStream |
aStream := WriteStream on: (String new: 16).
sel printOn: aStream.
^aStream contents
```

**Hook**

Object>>printOn: aStream

```
"Append to the argument aStream a sequence of characters that describes the receiver."

| title |
title := self className.
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 + sel maxPrint.
aStream nextPutAll: '\n'.
sel do: [element |
  aStream position > tooMany
  ifTrue: [aStream nextPutAll: '..('more)...'].
  ifFalse: [aStream print: element printOn: aStream]
  separatedBy: [aStream space].
aStream nextPutAll: $]
```

**Overriding**

False>>printOn: aStream

```
"Print false."

| aStream |
aStream := WriteStream on: (String new: 16).
sel printOn: aStream.
^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.

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 shallowCopy postCopy.

Hook Specialisation

Bag>>postCopy:
   "Make sure to copy the contents fully."
   | new |
   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

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

Sounds Trivial?

Another Example: Copying

Complex (deepCopy, veryDeepCopy...) Recursive objects Graph of connected objects Each object wants a different copy of itself No up-front solution

Delegation of Responsibilities
Ad-hoc Solution

Ad-hoc Solution
LanPrinter>>accept: aPacket
(thePacket addressTo: #id)
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

Matching Address

For packets with matchable addresses
Packet send: 'lulu' to: ('MatchingAddress with: #id')
Address subclass: #MatchingAddress
instanceVariableNames: '
MatchingAddress>>isAddressedTo: anNodeAddress
^ self id match: anNodeAddress 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

Composition Analysis

Pros
Possibility to change at run-time
Clear responsibility
No blob
Clear interaction protocol

Cons
New class
Delegation
New classes

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

Create Object and Delegate

NodeAddress

NodeAddress is responsible for identifying the packet receivers
Packet>>isAddressedTo: aNode
^ self id isAddressedTo: aNode address "was name"

Object subclass: #NodeAddress
instanceVariableNames: 'id'
NodeAddress>>isAddressedTo: aNodeAddress
^self id = aNodeAddress id

Delegating to other Objects

myEditor setColorizer: FastColorizer new.
myEditor setColorizer: AdvancedColorizer new.
Strategy design pattern
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...