

# Elements of Design - Simple Smells

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

S.Ducasse

1

## 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

S.Ducasse

3

## A Simple Case...

Introduce parametrization  
Avoid recompilation



S.Ducasse

2



## 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))
```

S.Ducasse

4

## 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))
```

S.Ducasse

5

## Good Coding Practices



Good coding practices  
promote good design  
Encapsulation  
Level of decomposition  
Factoring constants

S.Ducasse

6



## The Object Manifesto



Be lazy and be private  
Never do the job that you can delegate to another one  
Never let someone else plays with your private data

S.Ducasse

7

## The Programmer Manifesto



Say something only once  
Don't ask, tell!

S.Ducasse

8

## Designing Classes for Reuse



- Complete interface
- 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

S.Ducasse

9

## Behavior up State down



- Abstract class

- Concrete subclasses

S.Ducasse

10

## Say once and only once



- No duplicated magic number
- Extract method
- Remove duplicated code

S.Ducasse

11

## 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

S.Ducasse

12

## 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’  
  ...

S.Ducasse

13

## Factoring Out Constants



Write:

**NodeClient>>transmitTo: aNode**

  aNode hasNextNode

....

**Node>>hasNextNode**

  ^ (self nextNode = self class noNextNode) not

**Node class>>noNextNode**

  ^ ‘no next node’

S.Ducasse

14

## 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

S.Ducasse

15

## 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

S.Ducasse

16

## Good Signs of OO Thinking

- Short methods
- No dense methods
- No super-intelligent objects
- No manager objects
- Objects with clear responsibilities
- State the purpose of the class in one sentence
- Not too many instance variables



S.Ducasse

17



## Composed Methods

How do you divide a program into methods?

Messages take time

Flow of control is difficult with small methods

But:

Reading is improved

Performance tuning is simpler (Cache...)

Easier to maintain / inheritance impact



S.Ducasse

18

## Composed Methods



Divide your program into methods that perform one identifiable task. Keep all of the operations in a method at the same level of abstraction.

```
Controller>>controlActivity
    self controlInitialize.
    self controlLoop.
    self controlTerminate
```

S.Ducasse

19

## Do you See the Problem?

```
initializeToStandAlone
```

```
super initializeToStandAlone.
self borderWidth: 2.
self borderColor: Color black.
self color: Color blue muchLighter.
self extent: self class defaultTileSize * (self columnNumber @ self rowNumber).
self initializeBots.
self running.
area := Matrix rows: self rowNumber columns: self columnNumber.
area indicesDo: [:row :column | area at: row at: column
put: OrderedCollection new].
self fillWorldWithGround.
self firstArea.
self installCurrentArea
```



S.Ducasse

20

## Do you See the Problem?



```
initializeToStandAlone
```

```
super initializeToStandAlone.  
self initializeBoardLayout.  
self initializeBots.  
self running.  
self initializeArea.  
self fillWorldWithGround.  
self firstArea.  
self installCurrentArea
```

S.Ducasse

21

## With code reuse...



```
initializeArea
```

```
area := self matrixClass  
rows: self rowNumber  
columns: self columnNumber.  
area indicesDo: [:row :column | area  
at: row  
at: column  
put: OrderedCollection new]
```

initializeArea can be invoke **several** times

S.Ducasse

22

## About Methods



- Avoid long methods
- A method: one task
- Avoid duplicated code
- Reuse Logic

S.Ducasse

23

## Class Design



S.Ducasse

24

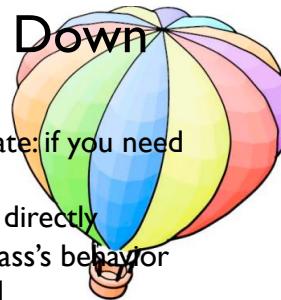
## Behavior Up and State Down

Define classes by behavior, not state

Implement behavior with abstract state: if you need state do it indirectly via messages.

Do not reference the state variables directly

Identify message layers: implement class's behavior through a small set of kernel method



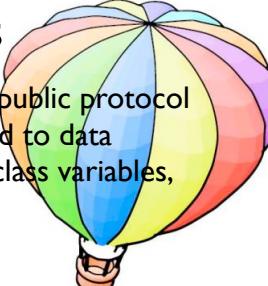
S.Ducasse

25



## Behavior-Defined Class

When creating a new class, define its **public protocol** and specify its behavior without regard to data structure (such as instance variables, class variables, and so on).



For example:

  Rectangle

Protocol:

  area

  intersects:

  contains:

  perimeter

S.Ducasse

27

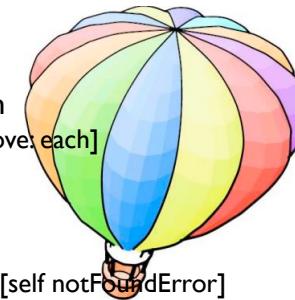


## Example

```
Collection>>removeAll: aCollection
```

```
  aCollection do: [:each | self remove: each]
```

```
^ aCollection
```



```
Collection>>remove: oldObject
```

```
  self remove: oldObject ifAbsent: [self notFoundError]
```

```
Collection>>remove: anObject ifAbsent:
```

```
  anExceptionBlock
```

```
  self subclassResponsibility
```

S.Ducasse

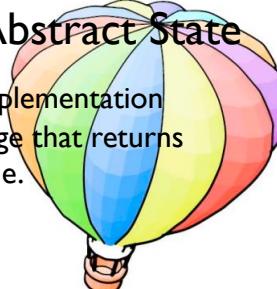
26



## Implement Behavior with Abstract State

If state is needed to complete the implementation

Identify the state by defining a message that returns that state instead of defining a variable.



For example, use

  Circle>>area

```
  ^self radius squared * self pi
```

not

  Circle>>area

```
  ^radius squared * pi.
```



S.Ducasse

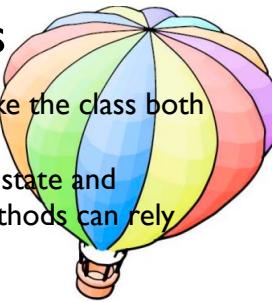
28



## Identify Message Layers

How can methods be factored to make the class both efficient and simple to subclass?

Identify a small subset of the abstract state and behavior methods which all other methods can rely on as kernel methods.



```
Circle>>radius  
Circle>>pi  
Circle>>center  
Circle>>diameter  
    ^self radius * 2  
Circle>>area  
    ^self radius squared * self pi
```

