Elements of Design - Simple Smells

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

A Simple Case...

Introduce parametrization
Avoid recompilation

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 hard-coded in method
Changes Require compilation
Client responsible of initialize invocation

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)}
Composition-based Solution

```smalltalk
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) )
```

Good Coding Practices

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

The Object Manifesto

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

The Programmer Manifesto

Say something only once
Don’t ask, tell!
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

Behavior up State down

Abstract class
Concrete subclasses

Say once and only once

No duplicated magic number
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:

\[\text{Node}>>\text{nextNode} \quad \wedge \text{nextNode} \]

\[\text{NodeClient}>>\text{transmitTo: aNode} \quad \text{aNode nextNode = 'no next node'} \]

\[\ldots\]

... Write:

\[\text{NodeClient}>>\text{transmitTo: aNode} \quad \text{aNode hasNextNode} \]

... \text{Node}>>\text{hasNextNode} 
\[\wedge (\text{self nextNode} = \text{self class noNextNode}) \not\]

\text{Node class}>>\text{noNextNode} 
\[\wedge \text{'no next node'} \]

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:

\[\text{aNode nextNode isNil not} \]

Write:

\[\text{Node}>>\text{hasNextNode} \quad \wedge \text{self nextNode} = \text{self noNextNode} \]

\[\text{Node}>>\text{noNextNode} \quad \wedge \text{self class noNextNode} \]

Initializing without Duplicating

It's better to write:

\[\text{Node}>>\text{initialize} \quad \text{AccessType} := \text{'local'} \]

\[\ldots\]

\[\text{Node}>>\text{isLocal} \quad \wedge \text{AccessType} = \text{'local'} \]

\[\text{Node}>>\text{initialize} \quad \text{AccessType} := \text{self localAccessType} \]

\[\text{Node}>>\text{isLocal} \quad \wedge \text{AccessType} = \text{self localAccessType} \]
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

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

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

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
Do you See the Problem?

initializeToStandAlone

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

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

About Methods

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

Class Design
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.

Example

```Smalltalk
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
```

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
```

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.
```
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