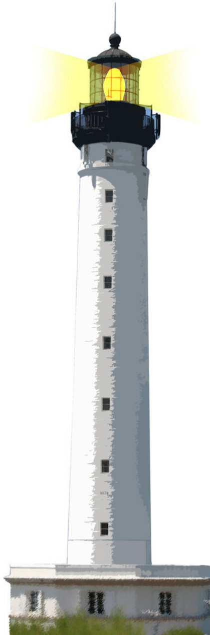
Learning Object-Oriented
Programming and Design with TDD

Avoid Null Checks

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



http://www.pharo.org

W7S06

What You Will Learn

- AntiIf Campaign
- If you do not want to be forced to test nil, do not generate nil
- NullObject Design Pattern

Anti If Campaign

```
Main >> showHappiness: animal
  animal isDog
    ifTrue: [ animal shakeTail ].
  animal isDuck
    ifTrue: [ animal quack ].
  animal isCat ifTrue: [ ... ].
```

Branching (with if) based on the type of an object is bad:

- adding a new type requires modifying all such code
- methods will become very long and full of details

Instead, send messages!

Anti-IF Campaign

Anti If Campaign

```
Dog >> showHappiness
  self shakeTail
```

```
Duck >> showHappiness
  self quack
```

```
Cat >> showHappiness
  ...
```

nil or anObject?

When you get a variable that can be nil or anObject

- Forced to check before doing anything
- Every access should be controlled

How to Avoid nil?

- Initialize well your objects (see Lectures Instance Initialization)
- When possible, do not return nil!
- When possible, apply NullObject Design Pattern

Example: Do Not Return Nil

```
Inferencer >> rulesForFact: aFact
    self noRule ifTrue: [ ^ nil ]
    ^ self rulesAppliedTo: aFact
```

`ifTrue: [ ^ nil ]` forces every client to check for `nil`!

```
(inferencer rulesForFact: 'a')
    ifNotNil: [ :rules |
      rules do: [ :each | ... ]
```

Return Polymorphic Objects

When possible, replace `if` by polymorphic objects:

- when returning a collection, return an empty one
- when returning a number, return 0

Example: Return Polymorphic an Empty Collection

```
Inferencer >> rulesForFact: aFact
    self noRule ifTrue: [ ^ #() ]
    ^ self rulesAppliedTo: aFact
```

Advantages:

- Your clients can just iterate and manipulate the returned value
- No check needed

```
(inferencer rulesForFact: 'a')
  do: [:each | ... ]
```

For Exceptional Cases, Use Exceptions

For exceptional cases, replace `nil` by exceptions:

- avoid error codes because they require `if` in clients
- exceptions may be handled by the client, or the client's client, or ...

```
FileStream >> nextPutAll: aByteArray
  canWrite ifFalse: [ self cantWriteError ].
  ...
```

```
FileStream >> cantWriteError
  (CantWriteError file: file) signal
```

```
Client >> handle...
  [ ... ] on: CantWriteError do: [:ex | ...handle exceptional case... ]
```

Initialize Your Object State

Avoid nil checks by initializing your variables

- by default instance variables are initialized with nil

```
Archive >> initialize
  super initialize.
  members := OrderedCollection new
```

See Lecture Initialize Instances

Use Lazy Initialization when Necessary

You can defer initialization of a variable to its first use:

```
FreeTypeFont >> descent
  ^ cachedDescent ifNil: [
      cachedDescent := (self face descender * self pixelSize //
                        self face unitsPerEm) negated ]
```

Be careful to systematically use the lazy accessor and not direct access

Sometimes you have to check...

- Sometimes you have to check before doing an action
- Solution: if you can, turn the default case into an object (null object)

Example

```
SelectionTool >> attachHandles
   ^ ... something complex...
```

```
SelectionTool >> detachHandles
   ^ ... something complex...
```

If tool can be nil, clients have to check!

```
ToolPalette >> nextAction
   self selectedTool
     ifNotNil: [ :tool | tool attachHandles ]
```

```
ToolPalette >> previousAction
  self selectedTool
    ifNotNil: [ :tool | tool detachHandles ]
```

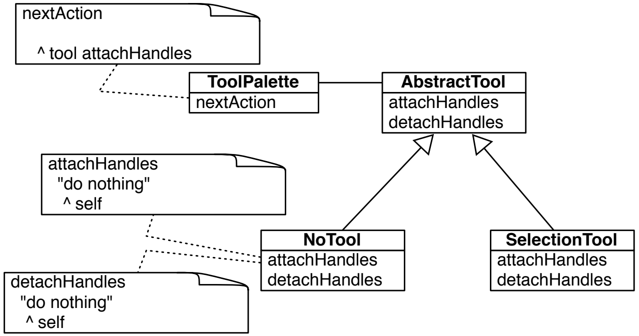
Use NullObject

```
NoTool >> attachHandles
    ^ self
```

```
NoTool >> detachHandles
    ^ self
```

The NullTool does nothing but offers a compatible API!

Use NullObject



- a null object proposes a polymorphic API and embeds default actions/values
- Woolf, Bobby (1998). "Null Object". In Pattern Languages of Program Design 3. Addison-Wesley.

Clients do not have to check anymore

```
ToolPalette >> initialize
    self selectedTool: NoTool new
```

```
ToolPalette >> nextAction
    self selectedTool attachHandles
```

```
ToolPalette >> previousAction
   self selectedTool detachHandles
```

Conclusion

- A message acts as a better `if`
- Avoid null checks, return polymorphic objects instead
- Initialize your variables
- If you can, create objects representing default behavior

Anti-IF Campaign

Resources

- Pharo mooc - Videos W7S07: `http://mooc.pharo.org`

A course by Stéphane Ducasse
http://stephane.ducasse.free.fr

Reusing some parts of the Pharo Mooc by

Damien Cassou, Stéphane Ducasse, Luc Fabresse
http://mooc.pharo.org