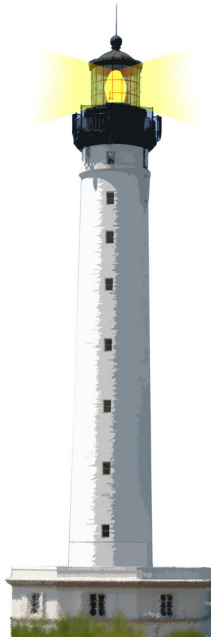
Advanced Object-Oriented Design

Avoid Null Checks

S. Ducasse, G. Polito, P. Tesone and L. Fabresse



Pharo

http://www.pharo.org

Goal

- Understanding the implication behind returning nil
- Looking at provider side
- Object initialization avoids nil propagation
- Looking at client side
- Null Object

nil?

- Unique instance of the class UndefinedObject
- In Pharo, a real object, as anybody else
- Default value of uninitialized instance variables
- Still we should be careful when to use it

Looking at provider side

- What is the impact of code generating nil?

Example

Imagine an inferencer that looks for rules that correspond to a fact.

```
| inf |
inf := Inferencer new.
inf
  addRule: #sunny –> #'sunglasses';
  addRule: #sunny –> #'solar cream';
  addRule: #rainy –> #'umbrella'.

inf rulesForFact: #sunny
> #(Fact (sunglasses) Fact(umbrella))

inf rulesForFact: #cloudy
> nil
```

Example code

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

- Here rulesForFact: returns nil to indicate that there is no rules for a fact.
- What are the consequences?

Consequences!

- Returning nil (e.g., ifTrue: [^ nil]) forces **EVERY** client to check for nil:

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

- Code ends up full of nil checks

Solution: Return polymorphic objects

When possible, return polymorphic objects:

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

Solution: Return polymorphic objects

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

Your clients can just iterate and manipulate the returned value

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

About nil

Limit the propagation of nil

- by having method returning nil
- avoid storing nil

Initialize your object state

Avoid nil checks by initializing your variables:

- By default instance variables are initialized with nil
- The responsibility of an object is to **correctly initialize** its state

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

Sometimes you have to check...

- Sometimes you have to check some conditions before doing an action
- When possible, you can turn the default case into an object, a Null Object.

Example

From the perspective of the client

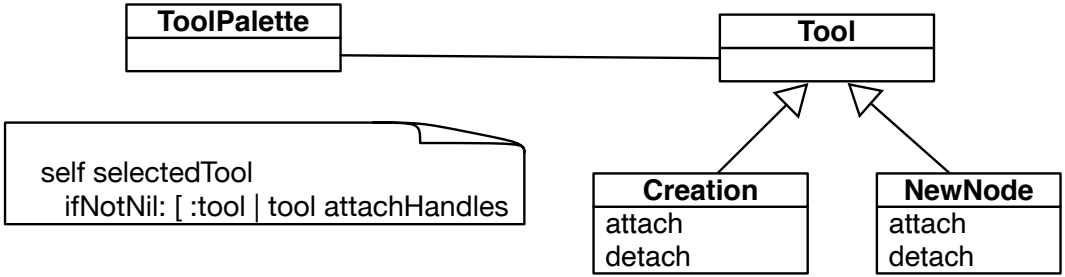
```
ToolPalette >> nextAction
  self selectedTool
    ifNotNil: [ :tool | tool attachHandles ]

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

Here we are forced to check that there is a selected tool.

- Why not having always one selected?
- Even one doing nothing?

Example


ToolPalette
Tool
self selectedTool
ifNotNil: [:tool | tool attachHandles
Creation
attach
detach
NewNode
attach
detach

Solution: 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.

Let us create a NoTool class whose behavior is to do nothing.

Solution: NoTool

```
AbstractTool << #NoTool
```

```
NoTool >> attachHandles
  ^ self

NoTool >> detachHandles
  ^ self
```

Solution: Use NullObject

Initialize the ToolPalette with a NoTool instance.

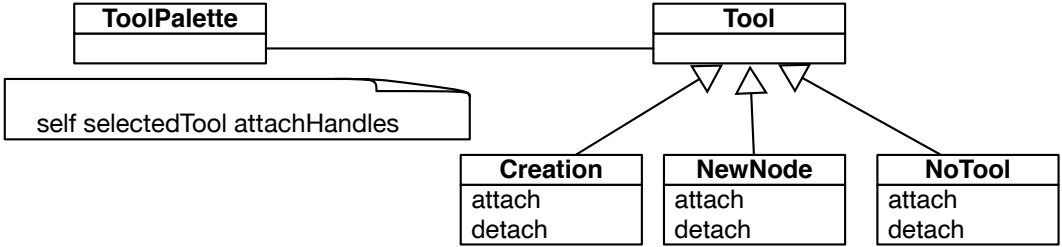
```
ToolPalette >> initialize
  self selectedTool: NoTool new
```

And we get no forced ifNil: tests anymore

```
ToolPalette >> nextAction
  self selectedTool attachHandles

ToolPalette >> previousAction
  self selectedTool detachHandles
```

Solution: With initialization and NoTool



Difficulty with NullObject

Sometimes it is difficult to apply the NullObject

- Too large API
- Or would need too many NullObjects
- Unclear default "no behavior"

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
```

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**

A course by

S. Ducasse, G. Polito, and Pablo Tesone