

# Avoid Null Checks

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



# Goals

- Understand the implication behind returning nil
- Analyze at provider side
- Object initialization avoids nil propagation
- Look 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 nils?



# 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  
> { Rule(sunny - sunglasses) . Rule(sunny - solar cream) }  
  
inf rulesForFact: #cloudy  
> nil
```



# Example code

```
Inferencer >> rulesForFact: aFact  
  (self noRule: aFact) 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 | ... ]
```

- Client code ends up full of nil checks (ifNil:, ifNotNil, isNil)



# 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: aFact) 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

- Methods should not return nil
- Avoid storing nil in variable
- **Initialize** well your object instance variables!



# Initialize your object state

Remember by default instance variables are initialized with `nil`

- This is the developer responsibility to produce well-initialized objects
- Avoid `nil` checks by initializing your variables:



# Initialization example

The responsibility of an object is to **correctly initialize** its state

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

- When default values are not enough, provide a constructor method



# 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)



# An example calling for a Null Object

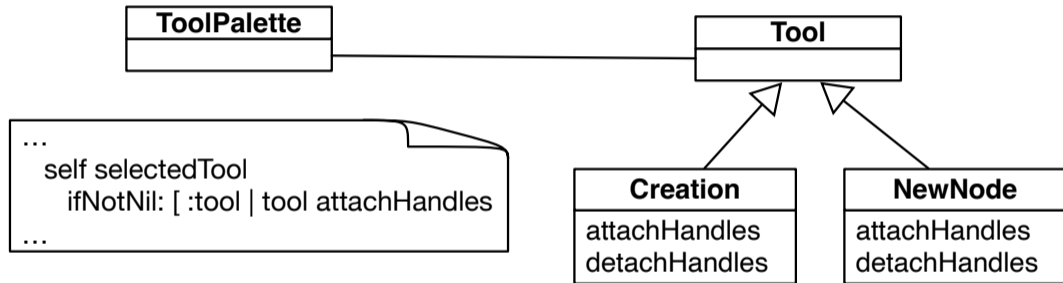
- Imagine a palette manipulates tools
- Palette has a selected tool

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

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



# Example



# Analysis

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

Forced to check that there is a selected tool

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





# Solution: Apply NullObject Design Pattern

- 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
- Read it!



# Solution: NoTool

Create a NoTool class whose behavior is to do nothing

```
AbstractTool << #NoTool
```

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

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



# Solution: Use NilObject

Initialize the ToolPalette with a NoTool instance.

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

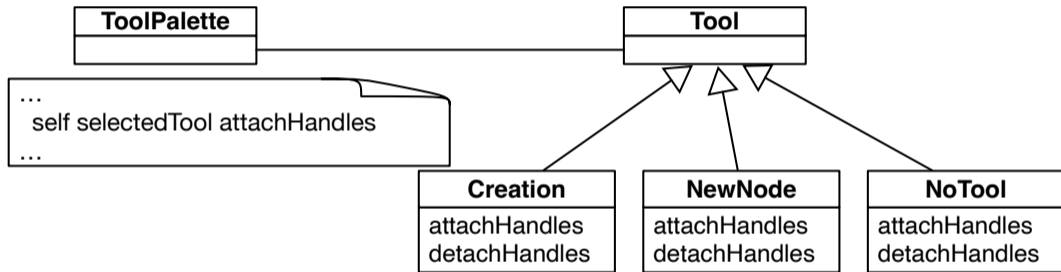
Not forced to use ifNil: tests anymore

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

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



# Solution: With initialization and NoTool



# NullObject pros

- Simplifies client code: real collaborators and null objects **offer the same API**
- Encapsulates do **nothing behavior**
- Makes do nothing behavior **reusable**



# NullObject drawback

- Encapsulate null values: may be difficult to mix with real objects
- A NullObject is not mutable into a real object
- All clients should **agree on the same do-nothing** behavior



# Difficulty applying NullObject

Sometimes it is difficult to apply the NullObject

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



# null object vs. NullObject

Sometimes it is possible to get a specific instance initialized with null values

- `NullTimeZone` is instance of `TimeZone` but represents a null object
- Null values could be good default values: empty collections, zeros...





# For exceptional cases, use exceptions

For exceptional cases, replace `nil` by exceptions:

- **avoid** error codes because they require `if` in clients
- exceptions are handled in the correct layer
- i.e., 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**



Produced as part of the course on <http://www.fun-mooc.fr>

# Advanced Object-Oriented Design and Development with Pharo

A course by

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



Inria  
LearningLab



Except where otherwise noted, this work is licensed under CC BY-NC-ND 3.0 France  
<https://creativecommons.org/licenses/by-nc-nd/3.0/fr/>