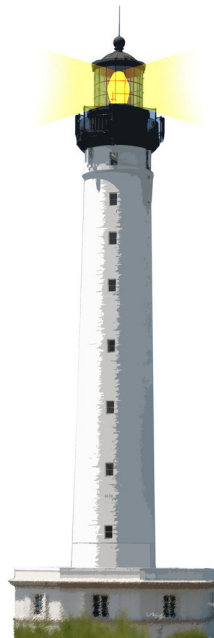


Avoid Null Checks

S. Ducasse



Goal

- Understanding the implication behind returning nil
- Object initialization avoids nil propagation
- 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



Example

Imagine an inferencer that looks for rules that can be applied to a fact.

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

- Here `rulesForFact:` returns `nil` to indicate that there is no rules for a fact.



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 | ... ]
```



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
```


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
```



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 ]
```

- This is only when the method `descent` is executed that `cachedDescent` will be initialized.



Solution: Use Lazy Initialization when Necessary

- Lazy initialization trades time vs runtime cost (ifNil: check)
- You should always use this accessor
- Pay attention you should NOT access directly an instance variable used in a lazy setting
- Else you can get exposed to nil value



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

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

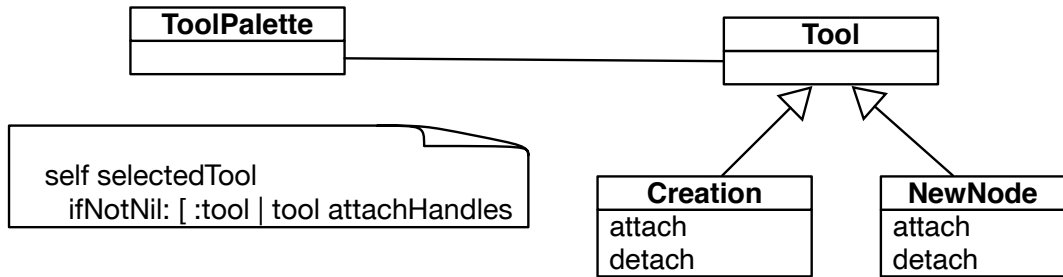
```
ToolPalette >> previousAction  
  self selectedTool  
  ifNotNil: [ :tool | tool detachHandles ]
```

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

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



Example



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 NilObject

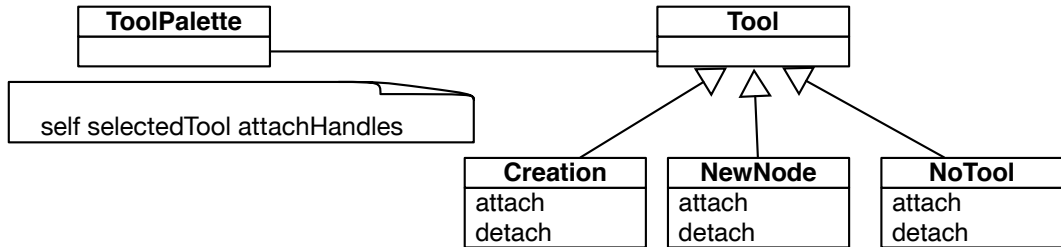
Initialize the ToolPalette with a NoTool instance.

```
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"



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



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/>