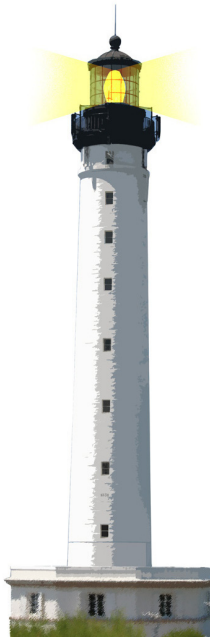
Singleton

a highly misunderstood pattern



Outline

- Singleton
- Singleton discussions
- Singleton misunderstanding

Singleton intent

- **From the book**: Ensure that a class has only one instance, and provide a global point of access to it
- **Better**: Ensure that a class has only one instance available at the same time

Problem/Solution

- **Problem:** We need a class with a unique instance.
- **Solution:** Store the first time an instance is created and return it each time a new instance is requested.

Most of the time think twice because you probably do not need it!

Example

```
db := DBConnect uniqueInstance.
db2 := DBConnect uniqueInstance.

db2 == db2
> true
```

Yes we get only one instance of the database connection

Possible implementation

```
Object < #BDConnect
  sharedVariables: { UniqueInstance }
```

```
BDConnect class >> uniqueInstance
  UniqueInstance isNil
      ifTrue: [ UniqueInstance := self basicNew initialize].
  ^ UniqueInstance
```

Kinds of Singleton

- **Persistent Singleton**: only one instance exists and its identity does not change
- **Transient Singleton**: only one instance exists at any time, but that instance changes
- **Single Active Instance Singleton**: a single instance is active at any point in time, but other dormant instances may also exist.

About name

```
DBConnect class >> new
  ^ self uniqueInstance
```

- The intent (uniqueness) is not clear anymore!
- new is normally used to return newly created instances
- new potentially means to
- get a new object and initialize that object
- uniqueInstance don’t

Method name variation

uniqueInstance

- Pure singleton ensuring a single global instance
- new should better be blocked

```
Author class >> uniqueInstance
  ^ uniqueInstance ifNil: [ uniqueInstance := self basicNew initialize ]
Author class >> new
  self error: 'Author is a singleton -- send uniqueInstance instead'
```

default

- Some meaningful default instance, but there is no reason to bar the user from creating more instances

current

- Keep the same instance system-wide, but we also want to change it under some circumstances

Discussion

- Even if the language supports global variables avoid to store a Singleton in a global
- A class is already acting as a global and it can manage the Singleton (one single entry point)

Shared variable vs class instance variable

In Pharo we have

- **SharedVariable**: shared between all the class of a hierarchy
- **class instance variable**:specific to a single class

Holding a singleton with

- **Shared variable**: One singleton for a complete hierarchy
- **Class instance variable**:
  - One singleton per class
  - Each subclass has its own singleton

Singleton misunderstanding

- Singleton is about time: one instance available at the same time is possible
- Singleton is not about access: don’t use a singleton because it is easier to access one instance!

Singleton misunderstanding

If you can add one instance variable to your object and suddenly you do not need a singleton then it was not a singleton but an ugly disguised global variable! Sometimes you cannot add an instance variable.

How to test singletons

- Singletons are global variables so this makes them more difficult to test
- Should be careful about not breaking the current singleton

Example: RPackageOrganizer

- RPackageOrganizer is a singleton: should not be destroyed when tests are run
- It uses withOrganizer: aNewOrganizer do: aBlock for testing behavior

```
withOrganizer: aNewOrganizer do: aBlock
  "Perform an action locally to aNewOrganizer. Does not impact any other organizers."

  | old |
  [ old := self organizer.
  old unregister.
  self organizer: aNewOrganizer.
  aNewOrganizer register.
  aBlock cull: aNewOrganizer ] ensure: [
    self organizer: old.
    old register.
    aNewOrganizer unregister]
```

Conclusion

- Having only one instance at a time
- Avoid Singleton as a global
- Avoid Singleton because it acts as a global

A course by

S. Ducasse, G. Polito, and Pablo Tesone