Command Pattern

CS 342 Fall 16

Design Problem

You are designing a new frontend framework called SlickUI. It is certainly going to be the next Bootstrap, and change the way designers create web pages. However, you need to assign actions to each of your components, like buttons.

Example Button class:

```
class SlickButton
    #lots of beautiful code
    def onButtonPush
        #do something
    end
    #lots more beautiful code
end
```

Using your SlickUI

Team # 1

- Will be using your GUI to build an online word processor.
- Button objects will need to create and save documents.

Team # 2

- Will be using your GUI to build a network utility that establishes a secure, anonymous network with remote machines.
- Button objects will need to initialize the network connection and disconnect from the network.

How can we design a button that commits an action, without knowing what the action will be?

Solution 1

- Make SlickButton an abstract class and the subclass can overwrite the functionality.
  - As the complexity of the program grows, so do the number of Classes. We could end up with hundreds of classes for each component.
  - Cannot change the button functionality without a conditional statement.
    - What if the button acts like a switch, does one thing when a file is open, but another when a file is closed.

Separate the Noun form the Verb

- The better solution to to separate the thing from the action.

- Every possible action is a command which makes up the **Command Pattern**.

Command Pattern

- The Command Pattern is an instruction to do something specific, usually, at a later date.

- Define the command patterns separately from the components that execute those commands.
  - The button HAS-A operation (command), not IS-A operation (command)

SlickButton again

- We can design a SaveCommand class that encapsulates the save action.
  - We then pass this command to the constructor of the button.

```
class SaveCommand
    def execute
        #brilliant save code
    end
end

save_button = SlickButton.new (SaveCommand.new)
```

Changing Behavior

- We can now change the behavior of the button during runtime
  - save_button.changeCommand(CopyCommand.new)

Procs as Commands

- A Command is a wrapper around a piece of code that knows how to perform some operation
  - It only exists to run a chunk of code during runtime
    - sound familiar?
- Proc also encapsulates a chunk of code
  - No need to define new classes, just use the prebuilt Proc
  - command = Proc.new { puts ‘Hello’}
    button = SlickButton.new(command)

Classwork

Installer

```
#Sounds like composite and commands
class Installer
	def intialize
		commands = []
	end
	def addCommand(command)
		@commands << command
	end
	def installationOptions()
		#define Procs for installing files, installing
		#extras, installing documentation. moving or
		#removing old files

		#ask user preferences
		#add to command queue
	end

	def call
		@commands.each {|c| c.call}
	end
end
```

Queueing up Commands

- A common operation is to ask for a set of preferences, then execute a series of commands based on those preferences
  - Installation
  - database transactions (Migrations)
    - database connections are slow, queueing commands allows you to not only commit commands at once, but also roll back commands

Are Command Classes Unnecessary in Ruby?

- Not exactly
  - depends on the complexity of your design, if you need to save state
- A command that records state requires a class
  - such as the data base transactions previously mentioned

Class Bloat

- The command pattern is not for simple commands
  - Use the command pattern for commands that need to be remembered and executed later
- Example or poor use of command: delete a file
  -
```
class FileDeleteCommand
        def initialize(path)
                @path = path
        end
        def execute
                File.delete(@path)
        end
end
```

```
fdc = FileDeleteCommand.new('foo.dat')
fdc.execute
```

-or just-

```
File.delete(@path)
```

Classwork

Undo

```
def TextProcessor
        def intialize
                @commands = []
        end
        def addCommand(command)
                @commands << command
        end
        def undo()
                #
        end
        def redo()
                #
        end
```

```
class command
        def intialize
                @state_info = []
        end

        def execute
                raise “abstract”
        end

        def unexecute
                raise “abstract”
        end
end
```

Classwork

Mobile Menu

```
#iterator
class MenuIterator
        def hasNext()
                #implement
        end
        def next()
                #implement
        end
        ...
end
```