Strategy Pattern

CS342 Fall 2016

File Reading in Ruby

```
fileObj = File.new($fileName, "r")
fileObj.each_line do |line|
    puts line
end
fileObj.close
```

Classwork - Pt 1: Rewrite a File

Assume Extensibility

- Did you design a base class?
  - When designing software, always assume it will be extended
- Assume you will always need a base class
  - This is always the starting assumption, even if you don’t think you’ll need it.
    - *It is easier to not extend something that was designed to be extended, than vice versa*

Design Principle of Delegation

- What would we need a base class for?
  - In other words, what could be generalized?
    - *The text transformations*
- ***Design Principle:*** Prefer delegation of tasks over centralization

Checking words from a file

```
class CheckStrategy
    def check(toCheck)
        raise “trying to call an abstract method”
    end
end
```

Checking words from a file (2)

```
class StartWithTheWordDesign < CheckStrategy
        def check(toCheck)
                if(toCheck.downcase.include? “design”)
                        puts(toCheck)
                end
        end
end
```

Checking words from a file (3)

```
class GreaterThan7 < CheckStrategy
        def check(toCheck)
                if(toCheck.length > 7)
                        puts(toCheck)
                end
        end
end

// can generalize this to GreaterThanN
// pass N in the constructor
```

Checking words from a file (4)

```
class IsPalindrome < CheckStrategy
        def check(toCheck)
                if(toCheck == toCheck.reverse)
                        puts(toCheck)
                end
        end
end

// can generalize this to GreaterThanN
// pass N in the constructor
```

Checking words from a file (4)

```
def printStuff(filename, strategy)
        infile = File.new(fileName, "r")
        while((buffer = infile.gets).empty?)
                tokens = buffer.split
                tokens. each do |t|
                        // CLASSWORK: What goes here?
                end
        end
end
```

Using A Strategy

```
def printStuff(filename, strategy)
        infile = File.new(fileName, "r")
        while(not (buffer = infile.gets).empty?)
                tokens = buffer.split
                tokens. each do |t|
                        if (strategy.check(word))
                                puts(word);
                        end
                end
        end
end
# method that never changes, because it is programmed to an interface
```

Classwork - Pt 2: Rewriting a File

Writing the Driver Code

```
designPrefix = StartsWithDesign.new()
printStuff('cs342.txt', designPrefix);
```

Strategy Pattern

Defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it

Modular Algorithms

- The algorithms can be used interchangeably to alter application behavior without changing its architecture
  - A new strategy can be added with only a slight change in the driver code
- By encapsulating the algorithm separately, new algorithms complying with the same interface can be easily introduced

Strategy Pattern

- Strategy enables the clients to choose the required algorithm, without using a "switch" statement or a series of "if-else" statements.
  - Imagine a sort class that uses a switch statement to select between bubblesort, mergesort, and quicksort switch statements
    - *How could we fix it using Strategy?*

Strategy Cons

- The application must be aware of all the strategies to select the right one for the right situation
    - There must be some logic when choosing between strategies
        - *this means *sigh* if statements*
- Driver code must be refactored whenever new strategies are introduced

Comparing Strategy to others

- What is the difference between Strategy and Template method pattern?
  - Hint: when is the concrete class (subclass incase of Template method) chosen?
    - *Compile time: Template Method*
    - *Run-time: Strategy*
  - Strategy represents a single, abstracted logical operation
  - Template method is a series of steps that do not vary

Classwork: Addition Performance

Classwork:
Vacation Planner

Duck Typing

- “If it acts like a duck and looks like a duck, then it is probably a duck”
- Why use an interface in ruby?
  - Duck typing makes a super-class that is just an interface unnecessary
    - *make it a module instead*
- In duck typed languages, the interface acts as documentation only, so is the interface completely unnecessary?

lambda methods

- An anonymous function (lambda method) is a function definition that is not bound to an identifier.
  - i.e. has no constant name
- lambdas are useful for:
  - passing functions around like data objects
  - when the result of a higher-order function is a process rather than a value

lambdas in Ruby

- uses the keyword lambda
  - example
    - *hello = lambda {*

      puts('Hello')

      }
  - usage
    - *hello.call*
      - yup, it's an object that has a method, call()

Proc Objects

- Proc wraps a lambda function
  - A wrapper class for lambda objects
- Proc picks up the local environment
  - name = ‘John’

    func = Proc.new {

        name = ‘Mary’

    }
    - *there is only one **name** object here*

Proc Parameters

- fullname = Proc.new{ |fname, lname|
  puts(fname + ‘ ‘ + lname)
  }
- first = ‘John’
  last = ‘Doe’
  fullname.call(first, last)

Proc v lambda scope

```
def lambda_test
  lam = lambda { return }
  lam.call
  puts "Hello world"
end #prints Hello world
```

```
def proc_test
  proc = Proc.new { return }
  proc.call
  puts "Hello world"
end #prints nothing
```

Proc vs lambda

- both are instances of Proc class
  - parameters
    - *lambdas require an absolute set number of parameters*
    - *Procs ignore extra parameters, and return nil on not enough parameters*
  - returns
    - *lambda returns from the lambda scope*
    - *Proc returns from the calling scope*