Builder Pattern

CS 342 Fall '16

Problem

Imagine that you are writing a system that will support a small computer manufacturing business. Each machine is custom made to order, so you need to keep track of the components for each machine. To keep things simple, suppose each computer is made up of a display, a motherboard, and some drives:

- The display is either :fourK or :ten_eighty
- The motherboard is an object and contains:
    - a standard CPU object or Turbo CPU object
    - Memory object which can be DDR3 or DDR4
    - Ports which can be:
        - usb, hdmi, vga, dvi
- Hard drive which can be Mechanical or Solid State

Driver code to build a computer

```
# Build a fast computer with lots of memory...
ports = [:usb, :usb, :usb, :hdmi]
motherboard = Motherboard.new(Turbo.new, DDR4.new(4096),
ports)
drives = []
drives << Drive.new(:hd, 2048) #size in GB's
drives << Drive.new(:ssd, 120)
computer = Computer.new(:lcd, motherboard, drives)
```

PROBLEM: **We have to write this code for 1000 computers**

Encapsulate the Object Creation

- What if we encapsulate the build process?
- Each builder has an interface that lets you specify the configuration of your new object step by step.
  - sort of like a multipart new method
- The Builder class factors out all of the details involved in creating an instance of an Object.
  - builder = ComputerBuilder.new
    pc = builder.turbo.display(:FourK).add_usb(4).add_ssd(120).buildComputer

Basic Builder Code

Builder

```
class Builder
    def initialize
        @product = Product.new
    end

    def addOptionA
        @product.optionA = OptionA.new
        self
    end

    ...
end
```

Product

```
class Product
    @optionA

    ...
end
```

Director

```
builder = Builder.new

prod = builder.addOptionA.addOptionB(x).buildProduct
```

Builder Structure

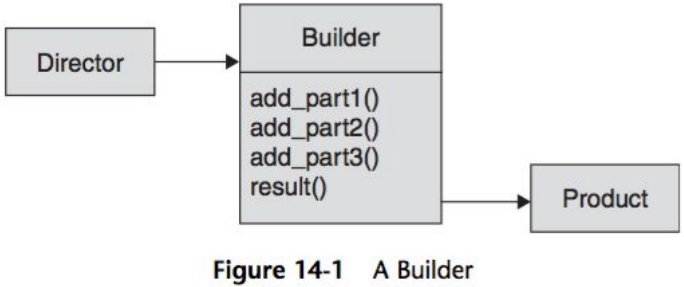

Figure 14-1 A Builder

The client of the builder object is called the **director** because it directs the builder in the construction of the new object (called the **product**)

Configuring Complex Objects

- Builders not only ease the burden of creating complex objects, but also hide the implementation details.
  - The director does not have to know the specifics of what goes into creating the new object.
- builders are also incredibly convenient spots to make those “which class” decisions.
  - SSD class or HD class for storage?
  - similar to the factory method

Building SubTypes

- Imagine that our computer business expands into producing laptops along with traditional desktop machines.
- The components of a laptop computer are not the same as the ones you find in a desktop computer.
    - Laptops have laptopMemory, batteries, and no display option. Desktops have desktopMemory.
- How could we refactor your builder into a base class and two subclasses to take care of these differences?

Abstract Base Builder

- The abstract base builder deals with all of the details that are common to the two kinds of computers
  - The DesktopBuilder knows how to build desktop computers
    - It knows to create instances of the DesktopComputer class and it is aware that desktop computers use DesktopMemory
  - The laptop builder knows how to create instances of LaptopComputer
    - It knows to populate that object with instances of the special LaptopMemory
- Why not write a single builder class that creates a laptop or a desktop system depending on a parameter.
  - Separate those things that change from those that stay the same. If we start making smartphones, we have to change the class again

Invalid Objects

- How can we use builders can make object construction safer?
  - This is especially helpful in a language like Ruby
- The final “give me my object” method is a great place to perform a sanity check on the configuration
  - For example, we can make sure our computer has a sane configuration:
    - def buildComputer

      raise "Not enough memory" unless @computer.motherboard.memory_size > 250

      raise "Too many drives" unless @computer.drives.size < 4

      raise "No hard disk." unless @computer.drives

      @computer

Builder vs Factory

- The Builder pattern is less concerned about picking the right class and more focused on helping you configure your object.
  - The final object is always the same, just configured differently internally
- The factory helps you pick the right kind of class in a family of classes without duplicating code shared by both
  - The final object could be of multiple subtypes
- YAGNI principle applies to both
  - “You ain’t going to need it”

Builder vs Decorator

- Builder is essentially the decorator applied to object creation
- Difference between builder and Decorator?
  - Builder uses methods instead of Classes, why?
    - Decorator classes take one parameter on initialization, the next chain class
    - Builder has many parameters that can configure the result
  - Builder has an option of methods, letting the director control the creation of the object
    - Decorator creates the object(s) and calls the single method that makes all the magic happen
    - Builder handles the construction of the object after it has been configured through method calls

Magic Methods

- Problem: You still have to create the builder and then call any number of methods to configure the new computer.
- Ruby Magic Methods to the rescue
  - The idea behind a magic method is to let the caller make up a method name on the fly according to a specific pattern
- Used extensively in Rails and other Frameworks
  - Magic Methods are a common technique to make using a class dynamic and simple

Return of the Method Missing

- Implement using method_missing
  - catch unexpected methods calls with method_missing, then parse the method names to see if it matches the pattern of your method names
    - builder.add_turbo_usb_hd
      - def method_missing(name, *args)

```
words = name.to_s.split("_")
return super(name, *args) unless words.shift == 'add'
words.each do |word|
    add_usb if word == 'usb'
    add_hard_disk(100000) if word == 'hd'
```

Classwork:

NYC Vacation

```
class Day
	def dinnerReservation(restaurant)
		self
	end
	def attraction(type)
		#super complex code
		self
	end
	def subwayTicket()
		#super complex code
		self
	end
end
class Vacation
	@days = []
	@hotel
	@num_days = 0
end
class VacationPackage
	def intialize
		@vacation = Vacation.new
	end
	def addDay(restaurant, attraction)
		@num_days += 1
		day = Day.new
		day.attraction(attraction).dinnerReservation(restaurant).subwayTicket
		@vacation.days << day
	end
	def addHotelStay(duration)
		#super complex code
		@vacation.hotel = hotel
		@vacation
	end
	def vacation
		self.addHotelStay(num_days)
	end
end

vacation = VacationPackage.new.addDay(“someRestaurant”, :met).addDay(“someOtherRestaurant”, :zoo).vacation
```