# Chapter 02 - The Builder Pattern

September 18, 2021

## 0.1 Overview

- imagine that we want to create an object that is composed of multiple parts and the composition needs to be done step by step
- the object is not complete unless all its parts are fully created
- the `builder desgin` pattern seprates the construction of a complex object from its representation
- by keeping the construction seprate from the representation, the same construction can be used to create several different representations

### 0.1.1 Example

- suppose that we want to create an HTML page generator
- the basic structure of an HTML page is always the same
  - starts with `<html>` and ends with `</html>`
  - inside can be `<title>` elements
- the representation of the page can differ however
- each page can have its own headings and a different body
- important to note that the page is usually built in steps
  - one function adds the title, another adss the main heading, another the footer, etc.
- only after the whole structure of a page is compelte can it be shown to the client using a final render function

- the HTML page generation problem can be solved using the builder pattern
- in this pattern, there are two main participants

- **the builder**: the component responsible for creating the various parts of a complex object. in this example, these parts are the title, heading, body, and the footer of the page
- **the director**: the component that controls the building process using a `builder` instance. It calls the builder's functions for setting the title, the heading and so on. And, using a different `builder` instance allows us to create a different HTML page without touching any of the code of the director

## 0.2 Builder vs Factory Pattern

- builder is useful when you need to do alot of things to build an object, such as the `DOM`
- a factory is used when the factory can easily create the entire object within one method call
- builder creates objects in multiple steps through a director
- factory pattern returns a created object immediately, in the builder pattern the client code explicitly asks the director to return the final object when its needed

### 0.3 Real-World Examples

- builder design pattern is used in fast-food restaurants
- the same procedure is always used to prepare a burger and the packaging (box and paper bag), even if there are many different kinds of burgers and different packages
- the difference between a classic burger and a cheeseburger is the representation, and not in the construction procedure
- in this case, the `director` is the cashier who gives instructions about what needs to be prepared to the crew, and the `builder` is the person from the crew that takes care of the specific order
- the `django CMS` uses the builder pattern

### 0.4 Use Cases

- use the builder pattern when we know that an object must be created in multiple steps and different representations of the same construction are required
- sine resources mention that the builder pattern can also be used as a solution to the telescopic construction problem
- the telescopic constructor problem occurs when we are forced to create a new constructor for supporting different ways of creating an object
- the problem is that we end up with many constructors and long parameter lists, which are hard to manage
- this problem does not exist in python because of named variable and argument list unpacking

**New Computer Analogy**: - assume that you want to buy a new computer, if you decide to by a specific, preconfigured computer model, for example, the latest Apple 1.4 GHz Mac Mimi, you use the factory pattern - all hardware specifications are already predefined by the manufacture, who knows what to do without consulting you

[13]:
```
MINI14 = '1.4GHz Mac mini'

class AppleFactory:
    class MacMini14:
        def __init__(self):
            self.memory = 4 # in gigabytes
            self.hdd = 500 # in gigabytes
            self.gpu = 'Intel HD Graphics 5000'

        def __str__(self):
            info = (f'Model: {MINI14}',
                f'Memory: {self.memory}GB',
                f'Hard Disk: {self.hdd}GB',
                f'Graphics Card: {self.gpu}')
            return '\n'.join(info)

    def build_computer(self, model):
            if model == MINI14:
```

```
                return self.MacMini14()
            else:
                msg = f'I dont know how to build {mode}'
                print(msg)

# main part of the program, which uses the AppleFactory CLass
if __name__ == '__main__':
    afac = AppleFactory()
    mac_mini = afac.build_computer(MINI14)
    print(mac_mini)
```

```
Model: 1.4GHz Mac mini
Memory: 4GB
Hard Disk: 500GB
Graphics Card: Intel HD Graphics 5000
```

- another option would be to buy a custom PC
- in this care, you use the builder pattern
- you are the director that gives orders to the manufacturer (`builder`) about your ideal computer

[15]:
```
# define the computer class
class Computer:
    def __init__(self, serial_number):
        self.serial = serial_number
        self.memory = None # in gigabytes
        self.hdd = None # in gigabytes
        self.gpu = None

    def __str__(self):
        info = (f'Memory: {self.memory}GB',
        f'Hard Disk: {self.hdd}GB',
        f'Graphics Card: {self.gpu}')
        return '\n'.join(info)

# define the ComputerBuilder class
class ComputerBuilder:
    def __init__(self):
        self.computer = Computer('AG23385193')

    def configure_memory(self, amount):
        self.computer.memory = amount

    def configure_hdd(self, amount):
        self.computer.hdd = amount

    def configure_gpu(self, gpu_model):
        self.computer.gpu = gpu_model
```

```
# define the HardwareEngineer
class HardwareEngineer:
    def __init__(self):
        self.builder = None

    def construct_computer(self, memory, hdd, gpu):
        self.builder = ComputerBuilder()
        steps = (self.builder.configure_memory(memory),
                 self.builder.configure_hdd(hdd),
                 self.builder.configure_gpu(gpu))
        [step for step in steps]

    @property
    def computer(self):
        return self.builder.computer

# we end the code with the main() function followed by the trick to call
# it when the file is called from the commandline

def main():
    engineer = HardwareEngineer()
    engineer.construct_computer(hdd=500,
                                memory=8,
                                gpu='GeForce GTX 650 Ti')
    computer = engineer.computer
    print(computer)

if __name__ == '__main__':
    main()
```

```
Memory: 8GB
Hard Disk: 500GB
Graphics Card: GeForce GTX 650 Ti
```

- the basic changes are the introduction of a builder (`ComputerBuilder`), a director (`HardwareEngineer`) and the step-by-step construction of a computer, which now supports different configurations (notice that `memory`, `hdd` and `gpu` are parameters and are not preconfigured

## 0.5 Implementation

- we will be be building a pizza-ordering application
- pizza is a good example because things in pizza need to be built in order
- also each pizza requires different thickness of its dough and toppings used

- we start by importing the required modules and declaring a few `Enum` parameters plus a constant that is used many times in the application
- the `STEP_DELAY` constant is used to add a time delay between the different steps of preparing

a pizza (prepare the dough, add the sauce, and so on)

[19]:
```
import time
from enum import Enum


PizzaProgress = Enum('PizzaProgress', 'queued preparation baking ready')
PizzaDough = Enum('PizzaDough', 'thin thick')
PizzaSauce = Enum('PizzaSauce', 'tomato creme_fraiche')
PizzaTopping = Enum('PizzaTopping', 'mozzarella double_mozzarella bacon ham␣
↪mushrooms red_onion oregano')

STEP_DELAY = 3 # in seconds for the sake of the example
```

- our end product is a pizza, which is described by the `Pizza` class
- when using the builder pattern, then end product does not have many responsibilities, since it is not supposed to be instantiated directly
- the builder creates an instance of the end product and makes sure that it is properly prepared
- thats why the `Pizza` class is so minimal
- it basically initializes all data to sane default values
- an exception is the `prepare_dough()` method

- the `prepare_dough()` method is defined in the `Pizza` class instead of a builder for two reasons
  - firstly, to clarify the fact that the end product is typically minimal, which does not mean that you should never assign it any responsibilities
  - secondly, to promote code reuse through composition

[ ]:
```
class Pizza:
    def __init__(self, name):
        self.name = name
        self.dough = None
        self.sauce = None
        self.topping = []

    def __str__(self):
        return self.name

    def prepare_dough(self, dough):
        self.dough = dough
        print (f'preparing the {self.dough.name} dough of your {self}...')
        time.sleep(STEP_DELAY)
        print(f'done with the {self.dough,name} dough')
```

- there are two builders: one for creating a margarita pizza (`MargaritaBuilder`) and another for creating a creamy bacon pizza (`CreamyBaconBuilder`)
- each builder creates a `Pizza` instance and contains methods that follow the pizza-making procedures:
  - `prepare_dough()`, `add_sauce()`, `add_toppings()` and `bake()`
  - to be precise, `prepare_dough()` is just a wraper to the `Pizza` class

- notice how each builder takes care of all the pizza-specific details
- for example, the topping of the margarita pizza is double mozzarella and oregano, while the topping of the creamy bacon pizza is mozzarella, bacon, ham, mushroom, red onion, and oregano

[22]:
```python
class MargaritaBuilder:
    def __init__(self):
        self.pizza = Pizza('margarita')
        self.progress = PizzaProgress.queue
        self.backing_time = 5 # in seconds for the sake of this example

    def prepare_dough(self):
        self.progress = PizzaProgress.preparation
        self.pizza.prepare_dough(PizzaDough,thin)

    def add_sauce(self):
        print('adding the tomato sauce to your margarita...')
        self.pizza.sauce = PizzaSauce.tomato
        time.sleep(STEP_DELAY)
        print('done with the tomato sauce')

    def add_topping(self):
        topping_desc = 'double mozzarella, oregano'
        topping_items = (PizzaTopping.double_mozzarella, PizzaTopping.oregano)
        print(f'adding the topping ({topping_desc}) to your margarita')
        self.pizza.topping.append([t for t in topping_items])
        time.sleep(STEP_DELAY)
        print(f'done with the topping ({topping_desc})')

    def bake(self):
        self.progress = PizzaProgress.baking
        print(f'baking your margarita for {self.baking_time} seconds')
        time.sleep(self.baking_time)
        self.progress = PizzaProgress.ready
        print('your margarita is ready')
```

[25]:
```python
class CreamyBaconBuilder:
    def __init__(self):
        self.pizza = Pizza('creamy bacon')
        self.progress = PizzaProgress.queued
        self.baking_time = 7 # in seconds for the sake of the example

    def prepare_dough(self):
        self.progress = PizzaProgress.preparation
        self.pizza.prepare_dough(PizzaDough.thick)
```

```
    def add_sauce(self):
        print('adding the crème fraîche sauce to your creamy bacon')
        self.pizza.sauce = PizzaSauce.creme_fraiche
        time.sleep(STEP_DELAY)
        print('done with the crème fraîche sauce')

    def add_topping(self):
        topping_desc = 'mozzarella, bacon, ham, mushrooms, red onion, oregano'
        topping_items = (PizzaTopping.mozzarella,
        PizzaTopping.bacon,
        PizzaTopping.ham,
        PizzaTopping.mushrooms,
        PizzaTopping.red_onion,
        PizzaTopping.oregano)
        print(f'adding the topping ({topping_desc}) to your creamy bacon')
        self.pizza.topping.append([t for t in topping_items])
        time.sleep(STEP_DELAY)
        print(f'done with the topping ({topping_desc})')
    def bake(self):
        self.progress = PizzaProgress.baking
        print(f'baking your creamy bacon for {self.baking_time} seconds')
        time.sleep(self.baking_time)
        self.progress = PizzaProgress.ready
        print('your creamy bacon is ready')
```

- the director in this example is the waiter
- the core of the `Waiter` class is the `constuct_pizza()` method which acecepts a builder as a parameter and executes allthe pizza-preparation steps in the right order
- choosing the apporpriate builder, which can be even done at runtime, gives us the ability to create a different pizza style withoug modifying any of the code of the director (`Waiter`)
- the `Waiter` class also contains the `pizza()` method, which returns the end product (prepared pizza) as a variable to the caller

[26]:
```
class Waiter:
    def __init__(self):
        self.builder = None

    def construct_pizza(self, builder):
        self.builder = builder
        steps = (builder.prepare_dough,
                 builder.add_sauce,
                 builder.add_topping,
                 builder.bake)
        [step() for step in steps]

    @property
    def pizza(self):
```

```
        return self.builder.pizza
```

- validate_stype() function is similar to the validate_age() function described in the Factory Pattern
- it is used to make sure that the user gives valid input, which in this case is a character that is mapped to a pizza builder
- the m character uses the MargaritaBuilder class and the c character uses the CreamyBaconBuilder class
- these mappings are the builder parameter
- a tuple is returned, with the first element set to True id the input is valid, or False if it is invalid

[27]:
```
def validate_style(builders):
    try:
        input_msg = 'What pizza would you like, [m]argarita or [c]reamy bacon? '
        pizza_style = input(input_msg)
        builder = builders[pizza_style]()
        valid_input = True
    except KeyError:
        error_msg = 'Sorry, only margarita (key m) and creamy bacon (key c) are␣
↪available'
        print(error_msg)
        return (False, None)
    return (True, builder)
```

- the last part is the main() function
- the main() fucntion contains code for instantiating a pizza builder
- the pizza builder is then used byt the Waiter director for preparing the pizza
- the created pizza can be delivered to the client at any later point

[28]:
```
def main():
    builders = dict(m=MargaritaBuilder, c=CreamyBaconBuilder)
    valid_input = False
    while not valid_input:
        valid_input, builder = validate_style(builders)
    print()
    waiter = Waiter()
    waiter.construct_pizza(builder)
    pizza = waiter.pizza
    print()
    print(f'Enjoy your {pizza}!')
```

**Summary**: 1. we imported time and Enum 2. we declared the variables for a few constants: PizzaProgress, PizzaDough, PizzaSauce, PizzaTopping, STEP_DELAY 3. we define our Pizza class 4. we define the classes for two builders, MargaritaBuilder and CreamyBaconBuilder 5. we define our Waiter class 6. we add the validate_style() function to improve things regarding exception handling 7. finally we have the main() function followed by the snippet for calling it when the program is run - we make it possible to choose the pizza builder based on the users

input, after validation via the `validate_style()` function - the pizza builder is used by the waiter preparing the pizza - the created pizza is then delivered

**Builder Pattern Variation** - an intresting variation of the builder pattern is on where calls to builder methods are chained - this is accomplished by defining the builder itself as an inner class and returning itself from each of the setter-like methods on it - the `build()` method returns the final object - the pattern is called `fluent builder`

[29]:
```
class Pizza:
    def __init__(self, builder):
        self.garlic = builder.garlic
        self.extra_cheese = builder.extra_cheese

    def __str__(self):
        garlic = 'yes' if self.garlic else 'no'
        cheese = 'yes' if self.extra_cheese else 'no'
        info = (f'Garlic: {garlic}', f'Extra cheese: {cheese}')
        return '\n'.join(info)

    class PizzaBuilder:
        def __init__(self):
            self.extra_cheese = False
            self.garlic = False

        def add_garlic(self):
            self.garlic = True
            return self

        def add_extra_cheese(self):
            self.extra_cheese = True
            return self

        def build(self):
            return Pizza(self)

if __name__ == '__main__':
    pizza = Pizza.PizzaBuilder().add_garlic().add_extra_cheese().build()
    print(pizza)
```

```
Garlic: yes
Extra cheese: yes
```