

Creational

Creational design patterns are used to

- Create objects effectively
- Add flexibility to software design

Many designs start by using [Factory Method](#) (less complicated and more customizable via subclasses) and evolve toward [Abstract Factory](#), [Prototype](#), or [Builder](#) (more flexible, but more complicated).

Factory Pattern

- **Object Creation without Exposing Concrete Classes:**
 - Factories provide a centralized way to create objects, allowing for better encapsulation and abstraction.
- **Dynamic Object Creation:**
 - Factories are useful when the exact class of the object to be created may vary at runtime based on certain conditions or configurations.
 - Factories can select the appropriate subclass or implementation to create based on these conditions.
- **Polymorphic Object Creation:**
 - Allowing clients to create objects without knowing the specific subclass or implementation being instantiated.
 - This promotes loose coupling and simplifies client code.
- **Difference to Dependency Injection:**
 - **When using a factory your code is still actually responsible for creating objects. By DI you outsource that responsibility to another class or a framework, which is separate from your code.**

```
class Burger:
    def __init__(self, ingredients):
        self.ingredients = ingredients

    def print(self):
        print(self.ingredients)

class BurgerFactory:
```

```
def createCheeseBurger(self):
    ingredients = ["bun", "cheese", "beef-patty"]
    return Burger(ingredients)

def createDeluxeCheeseBurger(self):
    ingredients = ["bun", "tomatoe", "lettuce", "cheese", "beef-patty"]
    return Burger(ingredients)

def createVeganBurger(self):
    ingredients = ["bun", "special-sauce", "veggie-patty"]
    return Burger(ingredients)

burgerFactory = BurgerFactory()
burgerFactory.createCheeseBurger().print()
burgerFactory.createDeluxeCheeseBurger().print()
burgerFactory.createVeganBurger().print()
```

Copy

Output

```
["bun", 'cheese', 'beef-patty']

['bun', 'tomatoe', 'lettuce', 'cheese', 'beef-patty']

['bun', 'special-sauce', 'veggie-patty']
```

Builder Pattern

1. Builder Pattern:

- **Complex Object Construction:**
 - Create complex objects with multiple optional or mandatory parameters, and the number of parameters makes constructor overloading impractical or confusing.
- **Fluent Interface:**
 - Allowing for a more expressive and readable way to construct objects, especially when chaining multiple method calls together.
- **Flexible Object Creation:**

- Create objects in a step-by-step manner, allowing for more flexibility in the construction process.
- When different parts of the object need to be configured independently or when the construction process is dynamic.

- **Parameter Omission:**

- Allow parameters to be omitted or set to default values, providing more control over object creation and reducing the need for multiple overloaded constructors

```
class Burger:
```

```
    def __init__(self):
```

```
        self.buns = None
```

```
        self.patty = None
```

```
        self.cheese = None
```

```
    def setBuns(self, bunStyle):
```

```
        self.buns = bunStyle
```

```
    def setPatty(self, pattyStyle):
```

```
        self.patty = pattyStyle
```

```
    def setCheese(self, cheeseStyle):
```

```
        self.cheese = cheeseStyle
```

```
class BurgerBuilder:
```

```
    def __init__(self):
```

```
        self.burger = Burger()
```

```
    def addBuns(self, bunStyle):
```

```
        self.burger.setBuns(bunStyle)
```

```
        return self
```

```
    def addPatty(self, pattyStyle):
```

```
        self.burger.setPatty(pattyStyle)
```

```
        return self
```

```
    def addCheese(self, cheeseStyle):
```

```
        self.burger.setCheese(cheeseStyle)
```

```
        return self
```

```
    def build(self):
```

```
return self.burger
```

```
burger = BurgerBuilder() \
    .addBuns("sesame") \
    .addPatty("fish-patty") \
    .addCheese("swiss cheese") \
    .build()
```

Singleton

Singleton Pattern is considered unpythonic

<https://www.youtube.com/watch?v=Rm4JP7JfsKY&t=634s>

Why it is bad:

- If you inherit from it, you can get multiple instances, which shouldn't be allowed.
- Testing code is hard with singleton because you cannot create multiple fresh instances for testing
- Does not work well with multi threaded applications because raise condition of

```
class ApplicationState:
    instance = None

    def __init__(self):
        self.isLoggedIn = False

    @staticmethod
    def getAppState():
        if not ApplicationState.instance:
            ApplicationState.instance = ApplicationState()
        return ApplicationState.instance

appState1 = ApplicationState.getAppState()
print(appState1.isLoggedIn)

appState2 = ApplicationState.getAppState()
appState1.isLoggedIn = True
```

```
print(appState1.isLoggedIn)
print(appState2.isLoggedIn)
```

```
>> False
```

```
>> True
```

```
>> True
```

Better Methodologies than Singleton:

- Object Pool Pattern manages a fixed number of instances instead of one,
 - ex. Managing database connections of graphics objects with a lot of data drawn over and over again
- Global Object pattern, as in the Constant pattern, a module instantiates an object at import time and assigns it a name in the module's global scope.
 - But the object does not simply serve as data; it is not merely an integer, string, or data structure. Instead, the object is made available for the sake of the methods it offers — for the actions it can perform.
- Python's module system and the fact that modules are imported only once make it easy to implement singleton behavior using modules.

Revision #16

Created 14 February 2024 22:49:01 by victor

Updated 2 April 2024 05:22:55 by victor