Other Design Patterns

Instructors **Battista Biggio**, **Angelo Sotgiu** and **Leonardo Regano**

M.Sc. in Computer Engineering, Cybersecurity and Artificial Intelligence
University of Cagliari, Italy

The Decorator Pattern

Until now, we often used Python **decorators**. They allowed us to add some functionality to an existing function or method, without modifying its implementation or leveraging subclassing.

The main concept of Python decorators is to wrap a function inside another function (the *wrapper*).
The wrapper, which will be called in place of the decorated function, is then responsible to perform the required operations (typically, additional operations are performed before or after the original function).

The Decorator Pattern

Until now, we often used Python **decorators**. They allowed us to add some functionality to an existing function or method, without modifying its implementation or leveraging subclassing.

The main concept of Python decorators is to wrap a function inside another function (the *wrapper*).
The wrapper, which will be called in place of the decorated function, is then responsible to perform the required operations (typically, additional operations are performed before or after the original function).

A more general approach to obtain a similar result, that can be implemented in any OOP programming language, is defined in the **Decorator Pattern**

The Decorator Pattern

An object of a class is wrapped inside an object (the *wrapper*) of a decorator class.

```
obj = SomeClass()
wrapped_obj = DecoratorClass(obj)
```

The Decorator Pattern

An object of a class is wrapped inside an object (the *wrapper*) of a decorator class.

```
obj = SomeClass()
wrapped_obj = DecoratorClass(obj)
```

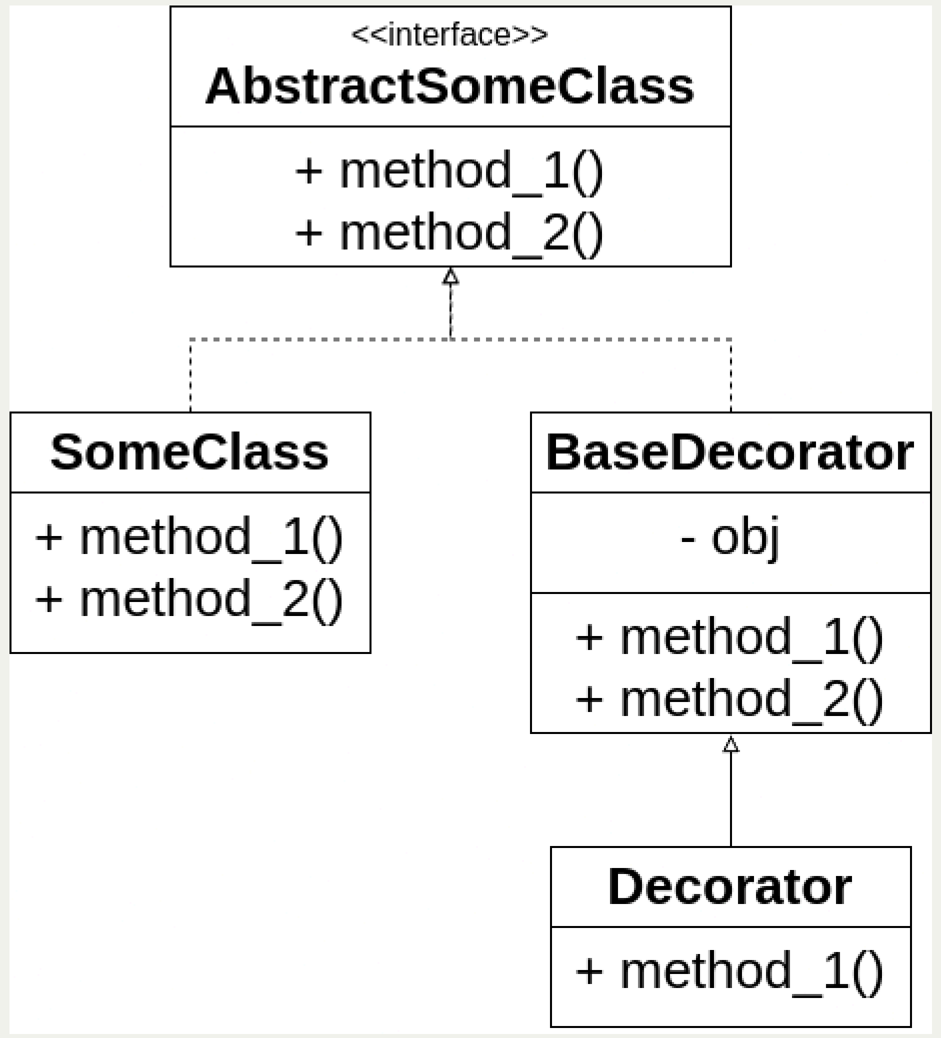
The wrapper object stores a reference of the wrapped object inside an attribute.

```
class DecoratorClass:

  def __init__(self, obj):
    self._obj = obj
```

The decorator can then be used in place of the original object, by exposing a similar interface.

```python
class SomeClass:

  def method_1(self):
    print("original method_1 of SomeClass")


class DecoratorClass:

  def __init__(self, obj):
    self._obj = obj

  def method_1(self):
    print("decorator operations")
    self._obj.method_1()
```

If we want to force the decorator classes to expose the **same** interface of the wrapped classes, we can rely on this architecture



```python
from abc import ABC, abstractmethod


class AbstractSomeClass(ABC):

  @abstractmethod
  def method_1(self):
    pass

  @abstractmethod
  def method_2(self):
    pass


class SomeClass(AbstractSomeClass):

  def method_1(self):
    print("original method_1 of SomeClass")

  def method_2(self):
    print("original method_2 of SomeClass")
```

```python
class BaseDecorator(AbstractSomeClass):

  def __init__(self, obj):
    self._obj = obj

  def method_1(self):
    self._obj.method_1()

  def method_2(self):
    self._obj.method_2()


class Decorator(BaseDecorator):

  def method_1(self):
    print("decorator operations")
    self._obj.method_1()
```

A more Pythonic implementation:

```
class SomeClass:

  def method_1(self):
    print("original method_1 of SomeClass")

  def method_2(self):
    print("original method_2 of SomeClass")


class Decorator:

    def __init__(self, obj):
        self._obj = obj

    def __getattr__(self, item):
        return getattr(self._obj, item)

    def method_1(self):
        print("decorator operations")
        self._obj.method_1()
```

Creational Patterns

The **Factory Pattern** is a creational design pattern that provides an interface for creating objects in a superclass, but allows clients to alter the type of objects that will be created.

Consider this class hierarchy

```python
# Define a product interface
class Animal:
  def speak(self):
    pass


# Concrete product classes
class Dog(Animal):
  def speak(self):
    return "Woof!"


class Cat(Animal):
  def speak(self):
     return "Meow!"
```

The concept aims to enable clients to instantiate objects from a class hierarchy **without directly referencing the class name** (see code on the right).
Complete the code writing the class **AnimalFactory**

```
dog = Dog()
print(dog.speak())  # Output: Woof!

cat = Cat()
print(cat.speak())  # Output: Meow!
```

```
factory = AnimalFactory()

dog = factory.create_animal("dog")
print(dog.speak())  # Output: Woof!

cat = factory.create_animal("cat")
print(cat.speak())  # Output: Meow!
```

The concept aims to enable clients to instantiate objects from a class hierarchy **without directly referencing the class name** (see code on the right).
Complete the code writing the class **AnimalFactory**

```python
dog = Dog()
print(dog.speak())  # Output: Woof!

cat = Cat()
print(cat.speak())  # Output: Meow!
```

```python
factory = AnimalFactory()

dog = factory.create_animal("dog")
print(dog.speak())  # Output: Woof!

cat = factory.create_animal("cat")
print(cat.speak())  # Output: Meow!
```

Solution

```python
# Factory class
class AnimalFactory:

  def create_animal(self, animal_type):
    if animal_type.lower() == "dog":
      return Dog()
    elif animal_type.lower() == "cat":
      return Cat()
    else:
      raise ValueError("Invalid animal type")
```

In this example, **Animal** is the interface, **Dog** and **Cat** are concrete classes implementing the interface, and **AnimalFactory** is the factory class responsible for creating instances of `Dog` or `Cat` based on the provided type.

The client code uses the factory to create objects without specifying their concrete classes directly.

We can avoid explicit conditional structures by using a mapping approach.

```python
class AnimalFactory:

  def __init__(self ):
    self._animal_types = {}

  def register_animal(self, animal_type, animal_class):
    self._animal_types[animal_type.lower()] = animal_class

  def create_animal(self, animal_type):
    animal_class = self._animal_types.get(animal_type.lower())
    if animal_class:
      return animal_class()
    else:
      raise ValueError("Invalid animal type")


# Register concrete classes
factory = AnimalFactory()
factory.register_animal("dog", Dog)
factory.register_animal("cat", Cat)
```

In this example, the **AnimalFactory** class has a `_animal_types` dictionary that maps animal types to their corresponding classes.
The `register_animal()` method is used to register these mappings, and the `create_animal()` method looks up the type in the dictionary to create the appropriate object.

This way, you can avoid explicit if-else conditions and make the code more extensible.

Builder Pattern

The **Builder Pattern** is a creational design pattern that **separates the construction of a complex object from its representation.**

It allows you to create different representations of an object by using the same construction process.
This pattern is particularly useful when you have an object with a large number of parameters or configuration options.

Computer is the product being constructed.

```python
class Computer:

  def __init__(self):
    self.parts = {}

  def add_part(self, part_name, part_specification):
    self.parts[part_name] = part_specification

  def display(self):
    print("Computer Parts:")
    for part, specification in self.parts.items():
      print(part, ":" ,specification)
```

`ComputerBuilder` is the builder interface declaring the construction steps. It can be either an abstract class or a concrete class, depending on the requirements and the desired level of flexibility.

```python
from abc import ABC, abstractmethod


class ComputerBuilder(ABC):

  def __init__(self):
    self.computer = Computer()

  @abstractmethod
  def build_cpu(self):
    self.computer.add_part("CPU", "Default CPU")

  @abstractmethod
  def build_memory(self):
    self.computer.add_part("Memory", "Default Memory")

  @abstractmethod
  def build_storage(self):
    self.computer.add_part("Storage", "Default Storage")

  def get_computer(self):
    return self.computer
```

`GamingComputerBuilder` and `WorkstationComputerBuilder` are concrete builders implementing the builder interface.

```python
class GamingComputerBuilder(ComputerBuilder):

  def build_cpu(self):
    self.computer.add_part("CPU", "High-end Gaming CPU")

  def build_memory(self):
    self.computer.add_part("Memory", "16GB RAM")

  def build_storage(self):
    self.computer.add_part("Storage", "1TB SSD")
```

```python
class WorkstationComputerBuilder(ComputerBuilder):

  def build_cpu(self):
    self.computer.add_part("CPU", "Professional Workstation CPU")

  def build_memory(self):
    self.computer.add_part("Memory", "32GB ECC RAM")

  def build_storage(self):
    self.computer.add_part("Storage", "2TB HDD + 512GB NVMe SSD")
```

`ComputerDirector` is responsible for directing the construction process.

```
class ComputerDirector:

  def construct(self, builder):
    builder.build_cpu()
    builder.build_memory()
    builder.build_storage()
    return builder.get_computer()
```

The client code creates different builders (for gaming and workstation computers) and instructs the director to construct the computers using these builders.

```python
gaming_builder = GamingComputerBuilder()
workstation_builder = WorkstationComputerBuilder()

director = ComputerDirector()

gaming_computer = director.construct(gaming_builder)
print("Gaming Computer:")
gaming_computer.display()

workstation_computer = director.construct(workstation_builder)
print("\nWorkstation Computer:")
workstation_computer.display()

# WARNING: workstation_computer2 is the same computer!
# workstation_computer2 = director.construct(workstation_builder)
```

Other (key concept of some) Design Patterns

- **The Flyweight Pattern**: allows different object to share common data in order to minimize the memory usage. Usually, this data is stored in a separate data structure (*e.g.*, a dedicated class) and passed by reference to the objects when they need it.

Other (key concept of some) Design Patterns

- **The Flyweight Pattern**: allows different object to share common data in order to minimize the memory usage. Usually, this data is stored in a separate data structure (*e.g.*, a dedicated class) and passed by reference to the objects when they need it.
- **The Adapter Pattern**: converts the interface of a class into another interface that is convenient for the developing task. It is useful when we want to link already existing code portions that were not thought to work together.

Other (key concept of some) Design Patterns

- **The Flyweight Pattern**: allows different object to share common data in order to minimize the memory usage. Usually, this data is stored in a separate data structure (*e.g.*, a dedicated class) and passed by reference to the objects when they need it.
- **The Adapter Pattern**: converts the interface of a class into another interface that is convenient for the developing task. It is useful when we want to link already existing code portions that were not thought to work together.
- **The Facade Pattern**: provides a unified high-level interface to a complex software architecture (*e.g*, a library), masking its inner interactions. This makes the software easier to understand and use.