CE259 PIP

PYTHON DESIGN PATTERNS

20CE076 – PATEL ADITYA
20CE081 – PATEL GAURANG

The design pattern is a technique which used by the developer to solve the commonly occurring software design. In simple word, it is a predefine pattern to solve a recurring problem in the code. These patterns are mainly designed based on requirements analysis.

A design pattern provides a general reusable solution for the common problems that occur in software design. The pattern typically shows relationships and interactions between classes or objects. The idea is to speed up the development process by providing well-tested, proven development/design paradigms. Design patterns are programming language independent strategies for solving a common problem. That means a design pattern represents an idea, not a particular implementation. By using design patterns, you can make your code more flexible, reusable, and maintainable.

It is a general repeatable solution for the potential problem in software development. the patterns details and We can follow apply a solution which suits our code.

It's not mandatory to always implement design patterns in your project. Design patterns are not meant for project development. Design patterns are meant for common problem-solving. Whenever there is a need, you have to implement a suitable pattern to avoid such problems in the future. To find out which pattern to use, you just have to try to understand the design patterns and their purposes. Only by doing that, you will be able to pick the right one.

ADVANTAGES

- All design patterns are language neutral.
- Patterns offer programmers to select a tried and tested solution for the specific problems.
- It consists of record of execution to decrease any technical risk to the projects.
- Patterns are easy to use and highly flexible.

There are mainly three types of design patterns

1. Creational Design Pattern
2. Structural Design Patterns
3. Behavioral Design Pattern

Creational Design Pattern

Creational patterns provides essential information regarding the Class instantiation or the object instantiation. Class Creational Pattern and the Object Creational pattern is the major categorization of the Creational Design Patterns. While class-creation patterns use inheritance effectively in the instantiation process, object-creation patterns use delegation effectively to get the job done.

Classification of Creational Design Patterns

- Factory Method
- Abstract Factory Method
- Builder Method
- Prototype Method
- Singleton Method

Structural Design Patterns

Structural design patterns are about organizing different classes and objects to form larger structures and provide new functionality while keeping these structures flexible and efficient. Mostly they use Inheritance to compose all the interfaces. It also identifies the relationships which led to the simplification of the structure.

Classification of Structural Design Patterns

- Adapter Method
- Bridge Method
- Composite Method
- Decorator Method
- Facade Method
- Proxy Method
- FlyWeight Method

Behavioral Design Pattern

Behavioral patterns are all about identifying the common communication patterns between objects and realize these patterns. These patterns are concerned with algorithms and the assignment of responsibilities between objects.

Classification of Behavioral Design Patterns

- Chain of Responsibility Method
- Command Method
- Iterator Method
- Mediator Method
- Memento Method
- Observer Method
- State Method
- Strategy Method
- Template Method
- Visitor Method

Adapter Method

Adapter method is a **Structural Design Pattern** which helps us in making the incompatible objects adaptable to each other. The Adapter method is one of the easiest methods to understand because we have a lot of real-life examples that show the analogy with it. The main purpose of this method is to create a bridge between two incompatible interfaces. This method provides a different interface for a class. We can more easily understand the concept by thinking about the Cable Adapter that allows us to charge a phone somewhere that has outlets in different shapes.

CODE

```
class MotorCycle:

    """Class for MotorCycle"""

    def __init__(self):
        self.name = "MotorCycle"

    def TwoWheeler(self):
        return "TwoWheeler"


class Truck:

    """Class for Truck"""

    def __init__(self):
        self.name = "Truck"

    def EightWheeler(self):
        return "EightWheeler"


class Car:

    """Class for Car"""

    def __init__(self):
        self.name = "Car"

    def FourWheeler(self):
        return "FourWheeler"

class Adapter:
    """
    Adapts an object by replacing methods.
```

```
    Usage:
    motorCycle = MotorCycle()
    motorCycle = Adapter(motorCycle, wheels = motorCycle.TwoWheeler)
    """

    def __init__(self, obj, **adapted_methods):
        """We set the adapted methods in the object's dict"""
        self.obj = obj
        self.__dict__.update(adapted_methods)

    def __getattr__(self, attr):
        """All non-adapted calls are passed to the object"""
        return getattr(self.obj, attr)

    def original_dict(self):
        """Print original object dict"""
        return self.obj.__dict__


""" main method """
if __name__ == "__main__":

    """list to store objects"""
    objects = []

    motorCycle = MotorCycle()
    objects.append(Adapter(motorCycle, wheels = motorCycle.TwoWheeler))

    truck = Truck()
    objects.append(Adapter(truck, wheels = truck.EightWheeler))

    car = Car()
    objects.append(Adapter(car, wheels = car.FourWheeler))

    for obj in objects:
    print("A {0} is a {1} vehicle".format(obj.name, obj.wheels()))
```

Advantages

- Principle of Single Responsibility: We can achieve the principle of Single responsibility with Adapter Method because here we can separate the concrete code from the primary logic of the client.
- Flexibility: Adapter Method helps in achieving the flexibility and reusability of the code.

- Less complicated class: Our client class is not complicated by having to use a different interface and can use polymorphism to swap between different implementations of adapters.
- Open/Closed principle: We can introduce the new adapter classes into the code without violating the Open/Closed principle.

Disadvantages

- Complexity of the Code: As we have introduced the set of new classes, objects and interfaces, the complexity of or code definitely rises.
- Adaptability: Most of the times, we require many adaptations with the adaptee chain to reach the compatibility what we want.

Mediator Method

Mediator Method is a **Behavioral Design Pattern** that allows us to reduce the unordered dependencies between the objects. In a mediator environment, objects take the help of mediator objects to communicate with each other. It reduces coupling by reducing the dependencies between communicating objects. The mediator works as a router between objects and it can have it's own logic to provide a way of communication.

CODE

```
class Course(object):
    """Mediator class."""

    def displayCourse(self, user, course_name):
        print("[{}'s course ]: {}".format(user, course_name))


class User(object):
    '''A class whose instances want to interact with each other.'''

    def __init__(self, name):
        self.name = name
        self.course = Course()

    def sendCourse(self, course_name):
        self.course.displayCourse(self, course_name)

    def __str__(self):
```

```
        return self.name

"""main method"""

if __name__ == "__main__":

    mayank = User('Mayank') # user object
    lakshya = User('Lakshya') # user object
    krishna = User('Krishna') # user object

    mayank.sendCourse("Data Structures and Algorithms")
    lakshya.sendCourse("Software Development Engineer")
    krishna.sendCourse("Standard Template Library")
```

Advantages

- Single Responsibility Principle: Extracting the communications between the various components is possible under Mediator Method into a single place which is easier to maintain.
- Open/Closed Principle: It's easy to introduce new mediators without disturbing the existing client code.
- Allows Inheritance: We can reuse the individual components of the mediators as it follows the Inheritance
- Few Sub-Classes: Mediator limits the Sub-Classing as a mediator localizes the behavior that otherwise would be disturbed among the several objects.

Disadvantages

- Centralization: It completely centralizes the control because the mediator pattern trades complexity of interaction for complexity in the mediator.
- God Object: A Mediator can be converted into a God Object (an object that knows too much or does too much).
- Increased Complexity: The structure of the mediator object may become too much complex if we put too much logic inside it.