Rahbord Shomal Institute of Higher Education
Bachelor's Thesis - Computer Engineering

Design Patterns in Object-Oriented Programming

Mehdi Nikzamir

Relevant Professor
Dr. Marzieh Faridi

February 2022

Table of Contents

Abstract

,The term 'design patterns' was first used by Erich Gamma, Richard Helm and Ralph Johnson in 1994. Design patterns are solutions to common software design problems. In the book 'Design Patterns: Elements of Reusable Object-Oriented Software', patterns are categorized into three groups: creational, structural, and behavioral. Creational patterns provide solutions for problems related to object creation, structural patterns deal with object composition, and behavioral patterns focus on class or object communication. The mentioned book describes 23 patterns. With the passage of time and advancements in programming languages, some patterns have lost their relevance. This article defines, examines, and implements 19 important and fundamental patterns in Python, including ,Factory Method, Abstract Factory, Prototype, Singleton, Builder, Adapter Bridge, Composite, Decorator, Facade, Proxy, Flyweight, Chain of Responsibility, Command, Observer, Iterator, State, Strategy, and Template Method. Efforts have also been made to simplify concepts as much as possible and ensure that the examples are practical and up-to-.date

Chapter One: Introduction

One of the most important topics in software engineering is understanding the principles and standards of programming, among which design patterns hold special significance. One of the challenges faced by programmers who have recently been hired to work on a project is understanding how the current system functions and identifying the various components of the project. This process is very time-consuming and often new programmers do not fully grasp the system being discussed. Several solutions are proposed to address this issue. One of these solutions is standardizing programming languages. Another approach is to use standard libraries. However, the most important and .best approach is understanding and implementing design patterns

Design patterns in object-oriented programming are very important from :two dimensions

.They facilitate communication among different developers -1
If two developers both know the design pattern 'observer', there is no need for hours of discussion and code review, and both can easily .understand how the system works or a part of it

Ready and easy solutions are available for solving multiple -2 .problems
If you are proficient in programming patterns, when faced with design and programming problems, you know what solution to take. For example, you want to design a system where information from multiple objects or multiple customer objects is received but you do not want to increase coupling between classes. If you are proficient in design .patterns, you know that you should use the Observer pattern

The author's goal in this article is to simplify the concepts of design patterns and their implementation in the Python language so that programmers can benefit from it as a comprehensive and concise .resource

:Keywords

– Design patterns – Object-oriented – Design elements – Programming Creational patterns – Structural patterns – Behavioral patterns

Chapter Two: Research Background

In 1995, Eric Gamma, Richard Helm, Ralph Johnson, and John Vlissides published a book titled "Design Patterns: Elements of Reusable Object-Oriented Software". This book is known as the father of design patterns and has been the driving force behind this idea. 23 design patterns are mentioned in this book. The authors of the book presented these 23 patterns based on their own experience as solutions to common problems of their time. After 26 years since the publication of this book, various patterns have been invented, but these 23 patterns still remain the most [54321].important programming patterns

"One of the other good books in this field, "Head First Design Patterns written by Eric Freeman, Elisabeth Robson, Bert Bates, and Kathy Sierra was published in 2004. The second edition of this book was also [109876] .published in 2020. This book examines 13 important patterns

[1] Eric Gamma
[2] Richard Helm
[3] Ralph Johnson
[4] John Vlissides
[5] Design Patterns: Elements of Reusable Object-Oriented Software
[6] Head first Design Patterns
[7] Eric Freeman
[8] Elisabeth Robson
[9] Bert Bates
[10] Kathy Sierra

Chapter Three: Research Method

This article is a narrative review article that discusses 23 design patterns in object-oriented programming and tries to simplify the definitions of this .topic as much as possible and help facilitate understanding of the issue In this article, each pattern is first defined and explained, then that pattern [11] .is explained in an example, and then it is implemented in Python

[11] Narrative Review Article

Chapter Four: Analysis of Findings

?What is Design Pattern

Christopher Alexander, a famous Austrian architect, describes design :patterns as follows

Each pattern addresses a common problem and then provides the core" of the solution in such a way that you can use this solution a million times "over, without ever doing it the same way twice

It is true that Alexander has mentioned this about architecture, but his statement about object-oriented design patterns is also true. The only difference is that in design patterns, we use the terms object and interface .instead of wall and door

:In general, each pattern has four elements

1- ,Pattern name: It is a title used to describe the problem, solution .and consequences of that pattern in one or two words

2- Problem: The problem that a pattern solves. Sometimes it includes a list of conditions that must be met in order to use the desired .pattern

3- ,Solution: The solution describes the elements of the pattern relationships, responsibilities, and collaborations. This solution is not definitive and completely clear, and it can be implemented in .various ways

4- .Outcome: Outcomes and side effects of using patterns are Knowing the outcome of each pattern can help us make better .pattern choices

:In general, three categories of design patterns are defined

1- Creational patterns
2- Structural patterns
3- Behavioral patterns

[12]Creational patterns

,Creational patterns summarize the instantiation process. With their help ,a system can be designed independently of how objects are created composed, and represented. A creational pattern of the class type uses inheritance to modify the class that is instantiated from it. While a creational pattern of the object type delegates the instantiation process .to another object

The importance of design patterns becomes more pronounced when a system relies more on compositions of objects rather than class inheritances. For this reason, the focus of these patterns is on defining sets of fundamental behaviors rather than coding a set of rules, which themselves can form the basis of complex behaviors. Therefore, creating .objects with specific behaviors is more than just instantiating a class

These patterns have two main principles; firstly, they encapsulate the information of the classes that the system uses. Secondly, they also hide .the creation and arrangement of instances of these classes

Creational patterns give us a lot of flexibility in what to build, who builds it, how and when to build it. These patterns allow us to configure a system .with ready-made objects that are very different in structure and behavior .This configuration can be static (at compile time) or dynamic (at runtime) [14][13]

[12] Creational Patterns
[13] Static
[14] Dynamic

Factory Method Pattern

Definition

A pattern that uses an interface to create objects and allows the subclass .to decide which object to instantiate

Objective

We want to define an abstract class or an interface that a class inheriting from it can perform the process of instantiation of other classes and based on the input we provide, return an object of the specific concrete class to us. Having an abstract class allows us to define different processes for instantiation. The method in which the instantiation task is performed in .that class is called the factory method

Use Cases

- A class of object-oriented programming that creates an object with .awareness is not
- A class does not want to determine the type of object it creates as a .subclass
- The class has delegated its responsibility to one of several subclasses and you want to locate the information on which subclass the work has .been delegated to

Structure

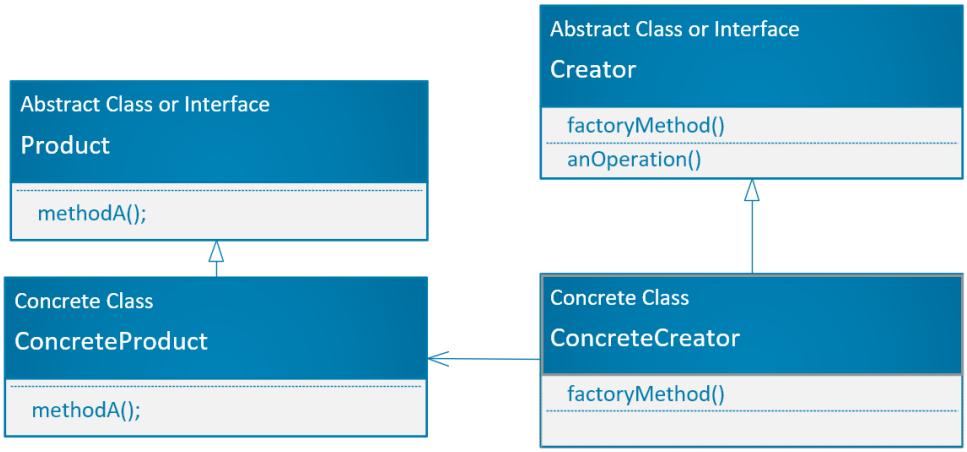


Figure 11 Class diagram in Factory Method Design Pattern -

Abstract class or interfaceCreator :

It is an abstract class or an interface that declares a method for creating .the appropriate object. It can also declare other necessary methods

ConcreteCreator class:

It is a concrete class that inherits from the Creator class and defines and implements the methods declared in it. The object creation method is implemented in this class and returns an instance of the ConcreteProduct class, which is of type Product class. So when we need to identify and create the appropriate object type, we should create an instance of this .class

ClassProduct :

An abstract class or an interface that declares the necessary methods and variables common to each type of product in it. For example, the class Product can be the class of mammals and the class ConcreteProduct can be a .type of mammal like a cow

ClassConcreteProduct :

A class that inherits from the classProduct and defines and implements .all the necessary methods and variables

Example

Assume there is a car manufacturing company named Medi .in Canada This company currently offers three cars M1 ,M2 and ,M2X to its customers. Each of these cars has different specifications. The M1 car is a crossover with a 2-liter engine producing 190 horsepower and 252 .Newton-meters of torque

The M2 car is a sedan with a 1.5-liter engine producing 140 horsepower .and 150 Newton-meters of torque

The car M2X is also a hatchback version M2 These cars are sold for . $26,000, $20,000, and $22,000 respectively. The sales program of this company is that first the customer orders the desired car, and then the car is built for them. Furthermore, this company plans to open a branch in Iran and modify its product specifications according to the needs of the .society

Now we want to design a system for this company. First, we need to pay :attention to the key features of this company

1. .Each car has its own specific features, but they are all cars
2. The company does not know in advance which car the customer .will order, and the customer chooses their car at the moment
3. Each car of this company is sold in Iran with changes in technical .specifications but in the same way

First, we create the class Medi:which is the main company class

```
class Medi:
    registerationNumber = 598632
    name = "Medi Motors .Co"

    def order(carName):
        pass
```

Company class – Figure 22

Then, we design the abstract class MediCars which the car classes must inherit from and implement its methods and properties. The constructor ,method of this class receives and stores the engine size, torque horsepower, and price in order. Additionally, this class implements the specs method which prints out the car specifications. There are two abstract methods build and deliver which are responsible for building a .car with the desired specifications and delivering it, respectively

```python
class MediCars(ABC):
    type: str
    engineSize: float
    torque: int
    horsepower: int
    price: int

    def __init__(self, engineSize: float, torque: int, horsepower: int, price: int):
        self.engineSize = engineSize
        self.torque = engineSize
        self.horsepower = horsepower
        self.price = price

    def specs(self):
        print("Specs: \n engineSize:{0} \n horsepower:{1} \n torque:{2} \n price:{3}".format(
            self.engineSize, self.horsepower, self.torque, self.price))

    @abstractmethod
    def build(self):
        return

    @abstractmethod
    def deliver(self):
        return
```

Figure 3- Abstract class implementation of carMediCars 3

```python
class M1(MediCars):
    type: "Crossover"

    def build(self):
        print("Your M1 is being built")

    def deliver(self):
        print("Here's your brand new M1 ^^")
```

```python
class M2(MediCars):
    type: "Sedan"

    def build(self):
        print("M2 is being built")

    def deliver(self):
        print("Here's your brand new M2 ^^")
```

```python
class M2X(MediCars):
    type: "Hatchback"

    def build(self):
        print("M2X is being built")

    def deliver(self):
        print("Here's your brand new M2X ^^")
```

Figure 44- Implementation of car classesM1, M2, M2X

Now we need to implement the order method in the Medi class in such a way that the customer enters the car name and an object of that car is created and returned (the company builds that car for him). For this purpose, we need to write a conditional statement and check that for example if the entered car name by the customer is M1 the object ,M1 is .created and returned

```
class Medi:
    registerationNumber = 598632
    name = "Medi Motors .Co"

    def order(self, carName):
        if(carName == "M1"):
            self.car = M1(2, 252, 180, 26000)
        elif(carName == "M2"):
            self.car = M2(1.5, 150, 140, 20000)
        else:
            self.car = M2X(1.5, 150, 140, 22000)

    self.car.build()
    self.car.deliver()
```

Implementation of the order method for cars in the class – Figure 55Medi

```
medi = Medi()
car1 = medi.order("M2")
```

Figure 66 Car order –

```
M2 is being built
Here's your brand new M2 ^^
```

Figure 77 Output of the above code -

:However, this approach comes with some drawbacks

1. If the number of cars increases, we are forced to repeatedly edit the order .method
2. Implementing the branch of this company in Iran will be troublesome. For example, the user must enter the car name as"M1-iran" and we have to check each of these strings in the conditional statement in the order method. This approach has two major drawbacks: 1- It makes code maintenance very difficult. 2- Due to repeated and excessive checks, we increase the number of .conditional statement executions

[15] .Our code is open for extension but closed for modification

The solution is to use the Factory Method pattern. This pattern .encapsulates the process of object instantiation in one or more classes In our example, the instantiation process of cars is encapsulated in several classes because there are two factories. One factory in Canada and one factory in Iran, each producing cars with different technical specifications. For example, each of these factories produces a specific model, such as M1, with suitable specifications for that country. In addition to their differences, there is also a similarity; both factories produce cars. Therefore, we need an abstract factory class that both the .Canadian and Iranian factories inherit from

```
class MediCarFactory(ABC):
    @abstractmethod
    def createCar(self, carName):
        return


class CanadaFactory(MediCarFactory):
    def createCar(self, carName):
        if(carName == "M1"):
            return M1(2, 252, 180, 26000)
        elif(carName == "M2"):
            return M2(1.5, 150, 140, 20000)
        else:
            return M2X(1.5, 150, 140, 22000)


class IranFactory(MediCarFactory):
    def createCar(self, carName):
        if(carName == "M1"):
            return M1(1.5, 180, 140, 56000000)
        elif(carName == "M2"):
            return M2(1.5, 150, 100, 46000000)
        else:
            return M2X(1.5, 135, 150, 50000000)
```

Shape 8Classes of factories in Iran and Canada -

Now we have two classesCanadaFactory and IranFactory which their method createCar creates and returns a suitable object. Now we need to rewrite the methodorder in the class Medi :

[15]open for extension, closed for modification

```python
class Medi:
    registerationNumber = 598632
    name = "Medi Motors .Co"

    def __init__(self, factory: MediCarFactory):
        self.factory = factory

    def order(self, carName):
        self.car = self.factory.createCar(carName)
        self.car.build()
        self.car.deliver()
        self.car.specs()
```

Shape 99 Rewrite the method -order

In the constructor method of classMedi first an object of type ,MediCarFactory is given, which based on the current scenario, will be an object of class IranFactory or CanadaFactory and stored in the variable factory In the method . order using polymorphism, the method ,createCar from the class stored in the variablefactory is called, which returns the desired object and stored in the variablecar Then three methods .build ,deliver and , specs are called from the object stored incar .

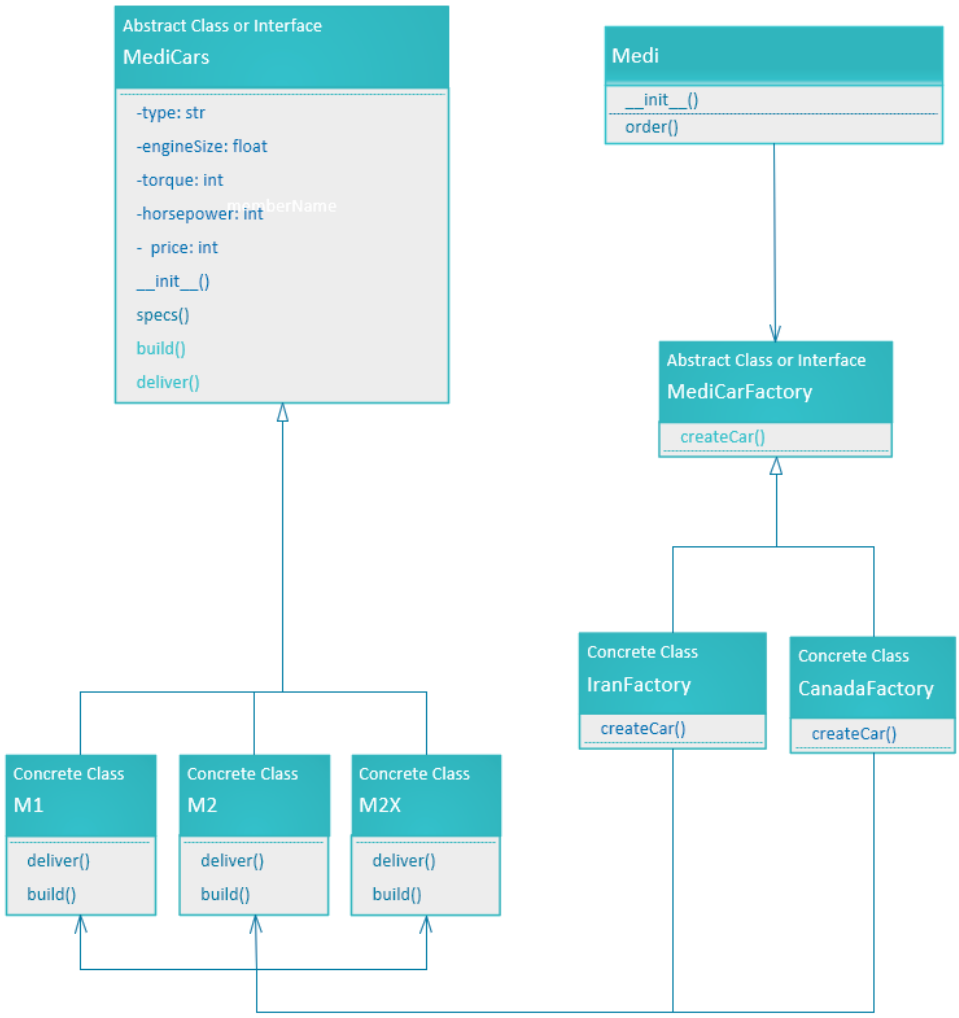
```python
IranFactory = Medi(IranFactory())
IranFactory.order("M1")
```

Shape 1010 - Order CarM1 from Iran Factory

```
Your M1 is being built
Here's your brand new M1 ^^
Specs:
 engineSize:1.5
 horsepower:140
 torque:1.5
 price:56000000
```

Shape 1111 - Output of the above code

:The new structure of this company's system will be as follows



– Shape 1212class diagram *New system design*

Abstract Factory Pattern

Definition

A pattern that provides an interface for creating families of related or .dependent objects without specifying their concrete classes

Objective

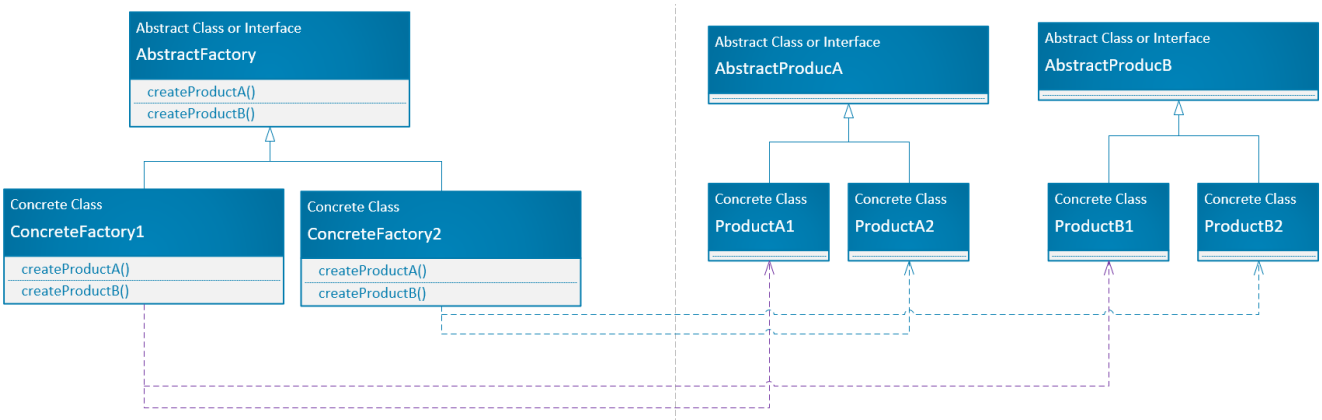
We want to define an abstract class or interface that the class inheriting from it can perform the process of instantiating other classes. The difference between this pattern and the Factory Method pattern is that this time a group of objects are created, and these objects are related to each other and together form a product. For example, objects like ,steering wheel, wheel, engine, etc. can form the product of a car. In fact .in the Abstract Factory pattern, there are several factory methods

Use Cases

- ,The system should not be dependent on how it produces, combines .and displays its products
- .The system should be configured with one of several product families
- A family of objects must be used together, and you must enforce this .constraint
- You want the information about the construction of a product and the combination of its components to be encapsulated, and you to be .unaware of it

Structure

Abstract Factory Pattern:



Shape 13 - Abstract Factory Pattern13

An abstract class or interface defined for operations that produce objects AbstractProduct .

ConcreteFactory Class:
A concrete class that implements operations to create objects AbstractProduct .

AbstractProduct Class:
An abstract class or interface defined for a type of product objectproduct .

ClassProductA1 :
It is a class that inherits from the classAbstractProduct.

Example
We want to create a date picker for two web and android platforms that developers can easily use. Each date picker consists of two modules: a [16] box and a scroller. Each of these modules is written once injava and once injavascript We need to design a system for developers to use the . .date picker of their desired platform

:First, we need to pay attention to the key features of this system

1. .A selector consists of two modules, a box and a scroller
2. The box and scroller modules are written injava for Android and in javascript .for the web
3. .The box and scroller modules are not used independently

Based on the above explanations, our selected pattern isAbstract Method . First, we create an abstract class namedDatePickerFactory which has two methodscreateDateBox() and createDateScroller() .

[16] Date Picker

```python
class DatePickerFactory(ABC):
    @abstractmethod
    def createDateBox(self):
        return

    @abstractmethod
    def createDateScroller(self):
        return
```

Figure 14– Implementation of the DatePickerCreator class 14

Then, we create two abstract classesDateScrollerer and DateBox These . .two classes define the general structure of the scroller and box modules

```python
class DateBox(ABC):
    @abstractmethod
    def dateBox(self):
        return


class DateScroller(ABC):
    @abstractmethod
    def dateScroller(self):
        return
```

15Figure 15 - Implementation of the DateBox and DateScroller classes

Now, it's time for concrete classesJSDateBox ,JavaDateBox , JSDateScroller and , JavaDateScroller where the first two classes inherit from the , DateBox class and the second two classes inherit from theDateScroller .class

```python
class JSDateBox(DateBox):
    def dateBox(self):
        print("a datebox is created in javascript")


class JavaDateBox(DateBox):
    def dateBox(self):
        print("a datebox is created in java")


class JSDateScroller(DateScroller):
    def dateScroller(self):
        print("a date Scroller is created in javascript")


class JavaDateScroller(DateScroller):
    def dateScroller(self):
        print("a date Scroller is created in java")
```

16Figure 16 - Implementation of classesJSDateBox, JavaDateBox, JSDateScroller, and JavaDateScroller

Then we implement two concrete classesJSDatePickerFactoryandJavaDatePickerFactory. These two classes inherit from the classDatePickerFactory. It is in these classes that .objects of typeDateScrollerandDateBoxare created

```
class JSDatePickerFactory(DatePickerFactory):

    def createDateBox(self):
        self.DateBox = JSDateBox()

    def createDateScroller(self):
        self.DateBox = JSDateScroller()


class JavaDatePickerFactory(DatePickerFactory):

    def createDateBox(self):
        self.DateBox = JavaDateBox()
        self.DateBox.dateBox()

    def createDateScroller(self):
        self.DateBox = JavaDateScroller()
        self.DateBox.dateScroller()
```

Implementation of classes - Figure 1717JSDatePickerFactory and JavaDatePickerFactory

:And finally, we create the Java date picker

```
javadatepicker = JavaDatePickerFactory()
javadatepicker.createDateBox()
javadatepicker.createDateScroller()
```

Creating a Java Date Picker - Final Code - Figure 1818

```
a datebox is created in java
a date Scroller is created in java
```

Figure 19 Output of the above code -

:So the structure of this system will be as follows

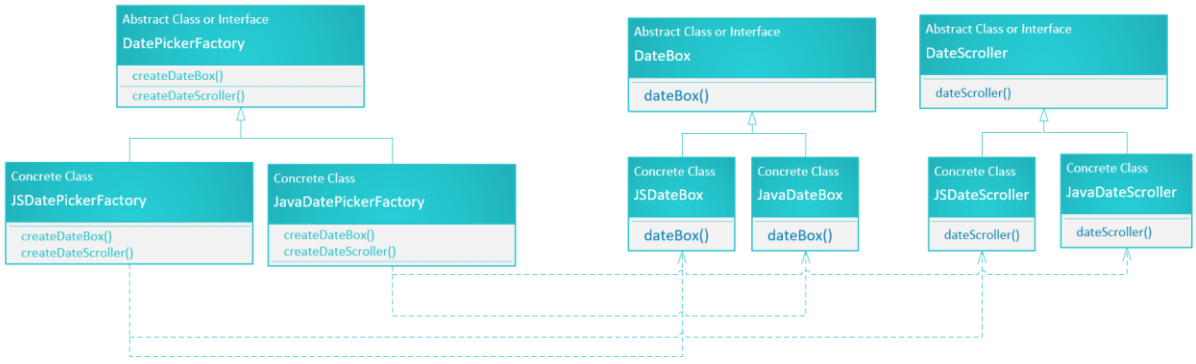


Figure 20 – class class Final design

Prototype Pattern

Definition

The Prototype pattern specifies the kinds of objects to create using a .prototypical instance, and creates new objects by copying this prototype [17]

Objective

Assume we want to completely copy a object. First create a new object of the same class. Then copy all the desired properties of the target object into the new object. But this may not be possible as the properties of that object may be private and we may not have access to them. Also we need to identify the class of the target object and make our code dependent on that class. Even knowing the interface of that class can help us because .we are aware of the method implementations as well

The main idea of the Prototype design pattern is that it allows copying an object without tying us to the class of that object. The responsibility of copying is delegated to the object we want to copy from, and this is done by providing a common interface for all objects that are capable of being copied. This interface allows you to perform an action on an object without being dependent on the class of that object. Usually, there is only one .copy method in this interface

.The implementations of copy method in different classes are very similar In a way that a copy of the current class is created and all the old attributes ;are transferred to the new object. You can even copy specific properties as most programming languages allow access to private fields of the .same class instance

To support copying, the first example should be named. When your objects have dozens of fields and hundreds of configurations, copying .them can be a replacement for subclassing

Use cases

[17] Clone

Use the Prototype pattern when you want to create, compose, and display a system independent of its products. You can also benefit from this :pattern in the following cases

- When classes that need to be instantiated are determined at runtime. For example, dynamic loading

or

- When you want to avoid creating a hierarchy of factories that .parallels a hierarchy of products

or

- When instances of a class can have a state from only a limited combination of states. Perhaps creating some initial corresponding instances and simulating them, instead of manually instantiating a .class each time with the appropriate state, is easier

Structure

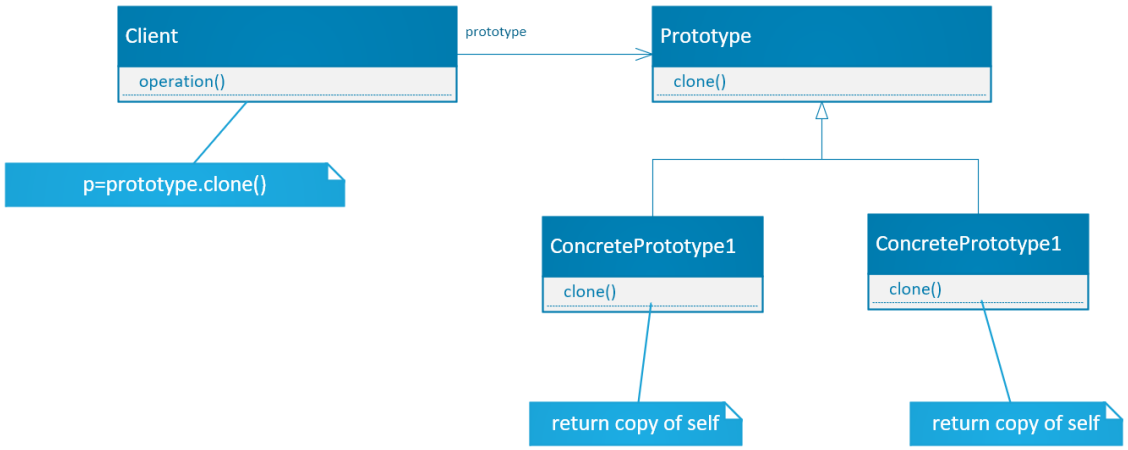


Figure 21 - *class diagram of Prototype pattern*

Prototype

Declares an interface for simulating itself

ConcretePrototype

Implements the necessary operations for simulating itself

Client

Creates a new object by requesting simulation of an initial instance

Example

Assume we have a class ScreenSaver that provides an interface for creating various screen savers. This class accepts an array of colors and a method for designing it. We have created a screen saver with a minimalist design and green and yellow colors (we created an object with these specifications from the ScreenSaver class). Now we want to design another screen saver with the same design but with green, yellow, and blue colors. So we need to make a copy of the initial screen saver object and then add the blue color to the existing colors. Perhaps we can say that the reason we do this without using class inheritance is that we only intend to create an object from this screen saver and do not want to unnecessarily expand the system with class inheritance. Based on the given explanations, we should use the Prototype. pattern

In Python, to copy an object, a copy module is provided and we can easily use it.

:The implementation of the system will be as follows

```python
class Prototype():
    def clone(self):
        return copy.copy(self)


class ScreenSaver(Prototype):
    _design = None
    _colors: List[str] = None

    def __init__(self, designFunction, colors):
        self._design = designFunction
        self._colors = colors

    def addColor(self, color):
        self._colors.append(color)

    def draw(self):
        self._design(self._colors)
```

Figure 22 Implementing the classes -Prototype and ScreenSaver

:Testing and running the program

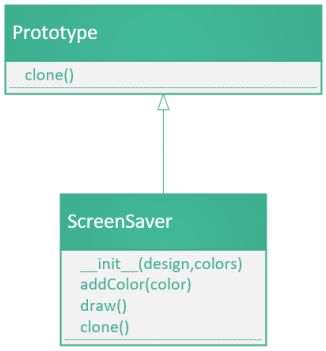
```python
screenSaver1 = ScreenSaver(lambda colors: [print(
    f"The design is made with the color {i}") for i in colors], ["green", "yellow"])
screenSaver1.draw()
print("-------------")
screenSaver2 = screenSaver1.clone()
screenSaver2.addColor("blue")
screenSaver2.draw()
```

23 Figure23- Testing and running the program

```
The design is made with the color green
The design is made with the color yellow
-------------
The design is made with the color green
The design is made with the color yellow
The design is made with the color blue
```

Figure 2424- Output of the above code

:The program structure is as follows



Shape 25 – *class diagram program*

Singleton Pattern

Definition

A pattern that allows only one instance of a class to be created and [18] .provides global access to it

Objective

We want to implement a class in such a way that it can only be instantiated once and if instantiation is repeated, it returns the same previous instance. This is done by privatizing the class constructor [2019] .function and using a static method

Use Cases

- There should only be one instance of a class and that instance should .be accessible from a well-known access point
- When a single instance is extensible and clients can use a developed .instance without changing their code

Structure

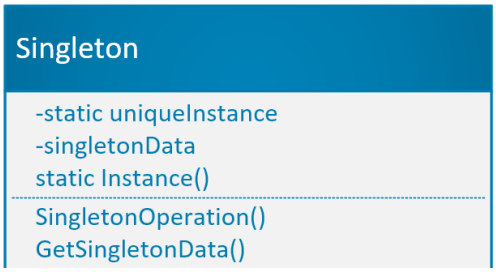


Figure 26 - class diagram of Singleton Design Pattern26

Singleton class:

.A class that can only have one instance of it

Example

Suppose we have a computer that only has a CD drive. We want to design a system that can read a CD or a DVD from the CD drive. To read the .next CD, the CD drive must be empty

:First, we need to pay attention to the key features of this system

[18] Public
[19] Private
[20] Constructor

1. .There is only one CD drive available
2. We need to check if the CD drive is empty when inserting the next .CD
3. .An optical drive should be available to everyone

Based on the above explanations, we can use the Singleton pattern. The reality is that this pattern has many opponents. One of the criticisms of this pattern is that the programmer may misunderstand the system requirements and in the future have more than one instance of the mentioned class. However, we focus on this pattern for educational .purposes

Since it is not possible to privatize the constructor method in Python, we need to create a metaclass for this purpose. There are two ways to implement this pattern in Python. The first way is easier but does not work .correctly in parallel processing or multiprocessing

The first way

__First, we create a metaclass SingletonMeta that checks in its __call method whether an instance of this class has been created or not. If an instance exists, return it, and if an instance does not exist, create one and return it. Then we create the OpticalDrive class, which has a variable .IsEmpty and two methods read and eject

```
class SingletonMeta(type):
    _classInstances = {}

    def __call__(cls, *args, **kwargs):
        if(cls not in cls._classInstances):
            instance = super().__call__(*args, **kwargs)
            cls._classInstances[cls] = instance

        return cls._classInstances[cls]


class OpticalDrive(metaclass=SingletonMeta):
    isEmpty = True

    def read(self, CD):
        if(self.isEmpty):
            self.isEmpty = False
            print(CD)
        else:
            print("The drive is full.")

    def eject():
        self.isEmpty = True
```

27Figure 27– Implementation of the SingletonMeta classandOpticalDrive

:Now we create two instances of this class

```
drive1 = OpticalDrive()
drive1.read("Ahang ghadimi")

drive2 = OpticalDrive()
drive2.read("Ahang Jadid")
```

Figure 28– Creating two instances of the OpticalDrive class 28

```
Ahang ghadimi
The drive is full.
```

Output of the above code – Figure 2929

As can be seen, only one object has been created. Because when we tried to create another object and read another CD, the message 'Drive is full' was displayed. If we print the two variablesdrive1anddrive2, we see .that both point to the same object

```
<__main__.OpticalDrive object at 0x00000259529C1130>
<__main__.OpticalDrive object at 0x00000259529C1130>
```

*Shape 3030 – Output printing two variables*drive1 *and* drive2

But the problem with this implementation is that if we instantiate this class in different threads, it is possible for the instances to be different. Because it is possible for two threads to run simultaneously and produce two [21].different instances

```
def createObject():
    drive = OpticalDrive()
    print(f"\n{drive}")


if(__name__ == "__main__"):
    thread1 = Thread(target=createObject, args=())
    thread2 = Thread(target=createObject, args=())
    thread1.start()
    thread2.start()
    thread1.join()
    thread2.join()
```

Running the above code in a multi-threaded manner – Figure 3131

Second way

Therefore, we need to implement this pattern in a way that it works correctly in different threads. To do this, we need to define a lock so that .only one thread can enter the instantiation section at a time

[21] Threads

```python
class SingletonMeta(type):
    _classInstances = {}
    _lock: Lock = Lock()

    def __call__(cls, *args, **kwargs):
        with cls._lock:
            if(cls not in cls._classInstances):
                instance = super().__call__(*args, **kwargs)
                cls._classInstances[cls] = instance

        return cls._classInstances[cls]
```

Figure 32– Implementation of the Singleton pattern in a Thread-proof manner 32

:Also, the new system structure is as follows

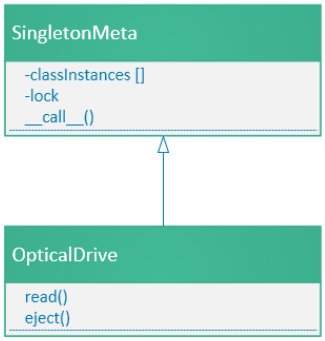


Figure 33–class diagram of the implemented system 33

Builder pattern

Definition

Builder pattern is a design pattern that separates the construction process of a complex object from its representation so that the same construction .process can create different representations as well

Objective

We want to build a complex object step by step by creating smaller objects so that we have more control over how that object is built. We can also decide in the moment what type of object that complex object should .be and how it should be constructed

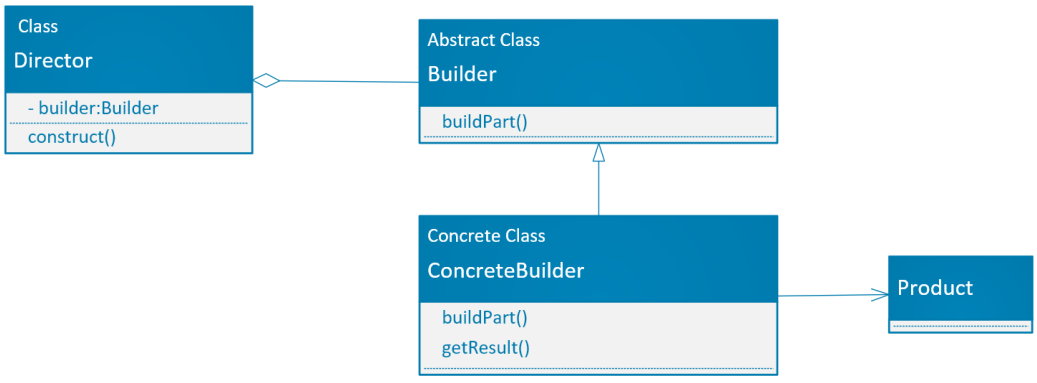
Use Cases

- The algorithm for creating a complex object should be separate from other parts of the code that are responsible for displaying and .assembling objects

- The construction process should allow for creating different .representations for the created object

.

Structure



– Figure 3434class diagram of Builder Pattern

The Director class

A class that holds an instance of the Builder within itself and produces a complex object using it. The existence of this class is not necessary but .it improves the readability of the code and system design

BuilderAbstract Class
An abstract class that defines a framework for creating product objects

ConcreteBuilderClass
A class responsible for building, assembling, and displaying objects

ProductClass
Final objects created by the ConcreteBuilder class

Example

Suppose we want to design an application that takes text as input and :translates it into English. This program performs several tasks

- .Translate the text into Persian or Italian and display it as text
- .Read the translated text and output an audio file

We know that there are two target languages, and the above two operations must be implemented separately for each language. So we .need an abstract class that each language inherits from

```
class TargetLanguage(ABC):
    @abstractmethod
    def translate():
        pass

    @abstractmethod
    def narrate():
        pass
```

Shape 35 *Class- TargetLanguage*

The output of the two above methods is text and sound, for which we also .need to create separate classes

```
class Text:
    text: str

    def __init__(self, txt):
        self.text = txt

    def download(self):
        return self.text


class Audio:
    filename: str
    fileformat: str
    narrator: str

    def __init__(self, name, fileformat, narrator):
        self.filename = name
        self.fileformat = fileormat
        self.narrator = narrator

    def download(self):
        return self.text + self.fileformat + "\n Narration by:" + self.narrator
```

Shape 36 *Implementation of classes- Audio and Text*

Now we need to implement classes for Italian and Persian languages and two methods translate and narrate .in them

```
class ToFarsi:
    def translate(self, txt):
        self.translation = Text("متن ترجمه شده!")
        return self.translation.download()

    def narrate():
        pasself.narration = Audio("Farsi-narration", "mp3", "Hootan Shakiba")
        return self.narration.download()


class ToItalian:
    def translate(self, txt):
        self.translation = Text("testo tradotto")
        return self.translation.download()

    def narrate():
        pasself.narration = Audio("Italian-narration", "mp3", "Carlo Sabatini")
        return self.narration.download()
```

Figure 3737 - Implementation of classes ToFarsi and ToItalian

:Now we design and implement the class for this application

```
class TranslatorApp:
    builder: TargetLanguage

    def __init__(self, builder):
        self.builder = builder

    def translate(self, txt):
        print(self.builder.translate(txt))

    def narrate(self, txt):
        print(self.builder.narrate(txt))

    def changeBuilder(self, builder):
        self.builder = builder
```

38 *Figure38- Implementing the class TranslationApp*

:We run the program

```
FarsiTranslationBuilder = ToItalian()
translationDirector = TranslatorApp(FarsiTranslationBuilder)
translationDirector.translate("translated text")
```

Figure 39 *Running the final code -*

```
testo tradotto
```

Figure 40 *Output of the above code -*

:So the structure of this application will be as follows

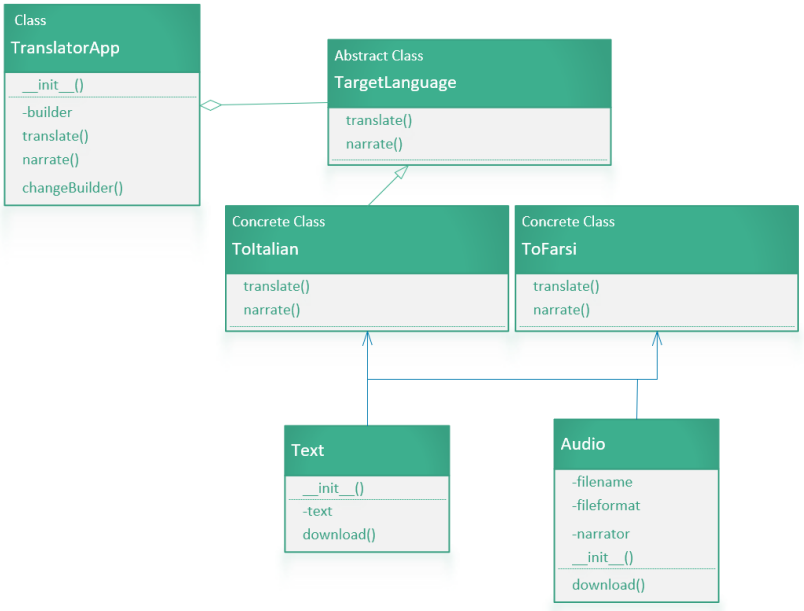


Figure 41- *class diagram of translator app*

Structural Patterns

Structural patterns are related to how objects and classes are combined to create larger structures. Class structural patterns use inheritance to combine interfaces or implementations. For example, consider how multiple inheritance merges two or more classes into one that combines the features of the parent classes. This pattern is very useful when we want to create libraries of independent classes. Another example could be the class structure in the Adapter pattern. In general, an adapter makes one interface like another interface. Therefore, a uniform .abstraction is created for all interfaces

But object-oriented design patterns introduce different ways of combining objects to achieve new functionality. The added flexibility in object composition allows for changing the composition at runtime, which is not possible in class composition. The Composite pattern is one of these patterns. This pattern describes how to build a hierarchical class structure that is made up of classes for two types of objects: the first object and the composite object. Composite objects allow you to combine first objects [2322] .and other composite objects in a complex structure of your choice

PatternAdapter

Definition

The Adapter pattern converts the structure of a class into the shape[2524] that the client needs. This pattern allows classes with different shapes to

[22] Primitive
[23] Composite
[24] Interface
[25] Client

work together. There are two types of Adapter pattern: Class Adapter and .Object Adapter

Objective

Sometimes a class cannot be used as it is and we need to change its shape to be able to use it. If we want to explain the Adapter pattern with a real-world example, converting a two-pronged plug can be a good example. We know that plugs in different countries have different standards and shapes. In some countries, like Iran, power cables have .two prongs and in some three prongs, with different shapes and distances

Types of sockets and series of power cables – Shape 4242

Now, imagine an American citizen has traveled to Iran. He realizes that the shapes of sockets in Iran are different from those in America. So, he can't use his laptop cable alone. To solve this problem, he needs to use a three-prong to two-prong adapter. This is exactly what the Adapter .pattern does

Shape 4343- Three-prong to Two-prong Adapter

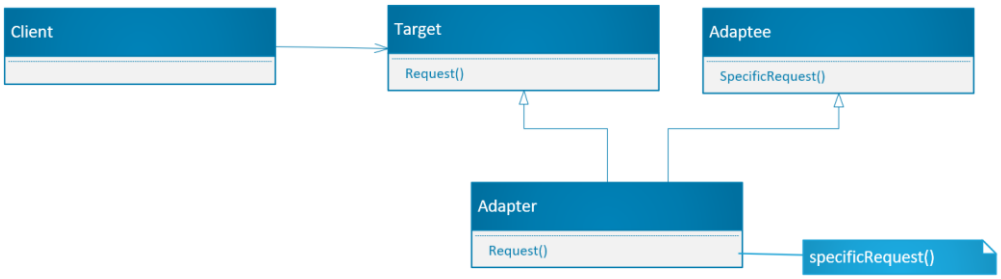
Use cases

- When you want to use a class but its shape is not compatible with the .class you need
- When you want to design a class that is reusable and can work with [26] .any other class, regardless of shape compatibility
- (Only for Adapter pattern) You need to use multiple subclasses, but adapting the shape of these classes to subclassing is difficult and .inefficient. An adapter object can take the shape of the parent class

[26] Reusable

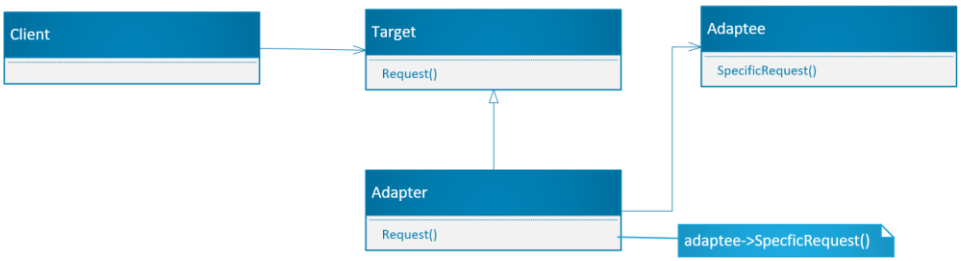
Structure

:The structure of the Adapter pattern class



44 Figure 44 - class diagram of Adapter(Class)

:The structure of the Adapter pattern object



45Figure 45 - class diagram of Adapters(object)

Class Target:

Specifies a structure that the client uses

Class Client:

Interacts with objects that are compatible with the Target

Class Adaptee:

A class that implements an operation in it and the Adapter class inherits .from it or creates an object of it

Class Adapter:

A class that inherits from the Target and Adaptee classes or creates an object of the Adaptee class. This class, when the request() method is .called, executes the specificRequest() method from the Adaptee class

Example

We want to design a system in which, during the registration process, a verification code is sent to the user's mobile number. We use a SMS panel that supports both SOAP and REST methods. We have decided to use SOAP in phase one, and if its performance is not satisfactory, we will switch to REST. For each of these methods, there is a separate script that we should use. The method for sending SMS in these two scripts are respectively sendSmsBySoapByNumber() and sendSmsRestNumbePN(). We can instantiate each of the classes of these two methods in our code and then call the desired method, but if ,we want to make a change in the method of connecting to the SMS panel we have to edit all the main parts of the program that have called the relevant method, which goes against the principle of package closure for editing. So, we want to design a class that is reusable and can work with any SMS panel connection script. The solution is to use the Adapter .design pattern

If we want to implement the Adapter class pattern, first we create a TargetSMS interface that defines the SendSms() method in it. Then we create an adapter class called SMSAdapter that inherits from the SoapSms and TargetSms .classes

We also create a class calledAPPLogin in which we create an object of SMSAdapter .

```python
class TargetSMS(ABC):
    @abstractmethod
    def SendSms(self, number):
        pass


class SoapSms():
    def sendSmsBySoapByNumber(self, number):
        print(f"sending {randint(1000,9999)} to {number} ")


class SMSAdapter(TargetSMS, SoapSms):
    def SendSms(self, number):
        self.sendSmsBySoapByNumber(number)


class APPLogin:
    # ....
    # some codes here
    def __init__(self):
        self.SMSAdapter = SMSAdapter()
        self.SMSAdapter.SendSms("09305667042")

    # some codes there


login = APPLogin()
```

ImplementationAdapter(class) – Shape 4646

```
sending 2009 to 09305667042
```

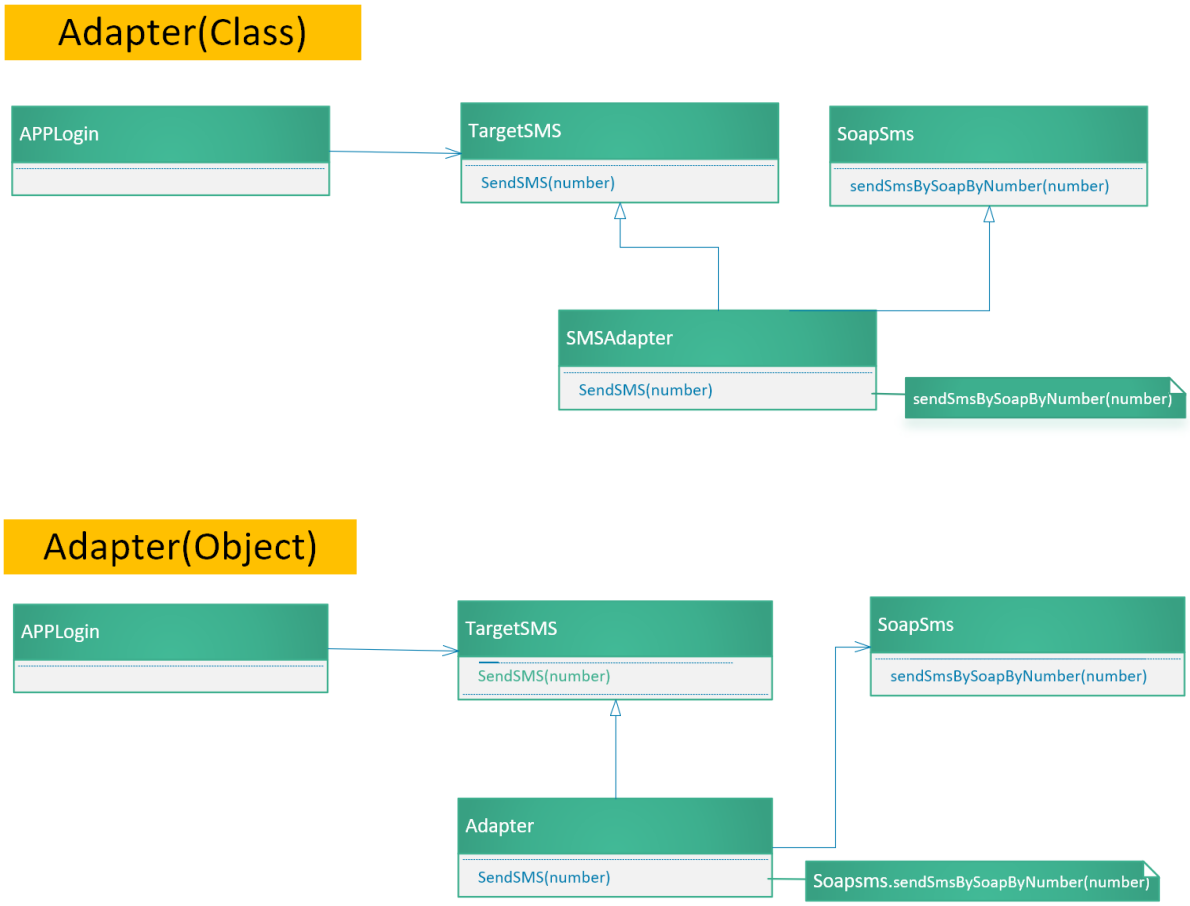
Shape 47 Output of the above code -

And if we want to implement the Adapter pattern for an object, we need .to create an object of SoapSms in the SMSAdapter class

```python
class SMSAdapter(TargetSMS):
    def SendSms(self, number):
        self.smsAproach = SoapSms()
        self.smsAproach.sendSmsBySoapByNumber(number)
```

Adapter(Object) Implementation – Shape 4848

:So our program structure will be as follows



Shape 49 - *class diagrams of Adapter(class)& Adapter(object)*

Bridge pattern

Definition

The Bridge pattern separates abstraction from implementation so that [28][27] .they can vary independently of each other

Goal

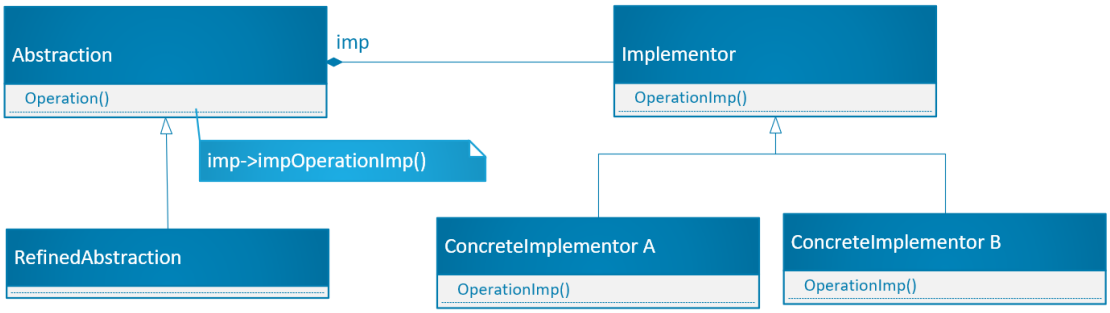
When an abstraction can have multiple implementations, the usual way to do this is inheritance. An abstract class defines the abstract structure and concrete classes implement it. But this approach is not always flexible enough. Inheritance permanently binds an implementation to an abstraction, making it difficult to edit, extend, and reuse abstractions and implementations separately. The Bridge pattern acts like a Cartesian product and creates various combinations of abstractions and .implementations

Use Cases

- When you do not want implementations and abstractions to be permanently limited and dependent on each other. For example, the system may need to be designed so that the implementation must be .selected or changed at runtime
- Through subclassing, both abstractions and their implementations .should be extendable
- Changes in the implementation of an abstraction should not affect the .clients. This means that their code should not need to be recompiled
- .The number of classes is increasing
- [29] You want to share an implementation among several different objects (possibly using the technique of reference counting) and this .should remain hidden from the clients

Structure

[27] Abstraction
[28] Implementation
[29] Reference Counting Technique



– Shape 5050class diagram of Bridge Pattern

Abstraction

This class defines an interface for abstraction and also has an instance of an object of type Implementor

RefinedAbstraction

This class extends the Abstraction interface.

Implementor

Defines an interface for implementing classes. This interface does not need to be fully compatible with the Abstraction interface. In fact, these two can be completely different from each other. Typically, the Implementor interface only covers several initial operations and the Abstraction defines higher-level operations from these several initial [30] .operations

ConcreteImplementor

واسطهImplementor implements

[30] Higher-level

Example

Assume that we want to design a program similar to Spotify. We want to have three types of content: book, album, and podcast. When we click on .them, we enter the specific page of each content and see its information We also want to have three display modes: comprehensive, simple, and minimal. In the comprehensive display, cover, artist name, title, number of listens, description, and ... In the simple display, cover, title, artist .name, and in the minimal display, only the title and artist name are shown We want each type of content, book, album, and podcast, to have these three displays and be displayed appropriately to the user based on the screen size. So far, we know that we need to create three classes for display, three classes for content type, and three classes for the main page information of each content type because the information of each content varies depending on its type. For example, book information includes book title, front and back cover, narrator, author biography, and ... But album information includes singer, composer, discography, and ... Note that the reason we have created a separate class for the book ,information page is the existence of attributes such as biography narrator, and ... That are not part of the book but are about that book. So [31] .our classes will be as follows

[31] Spotify

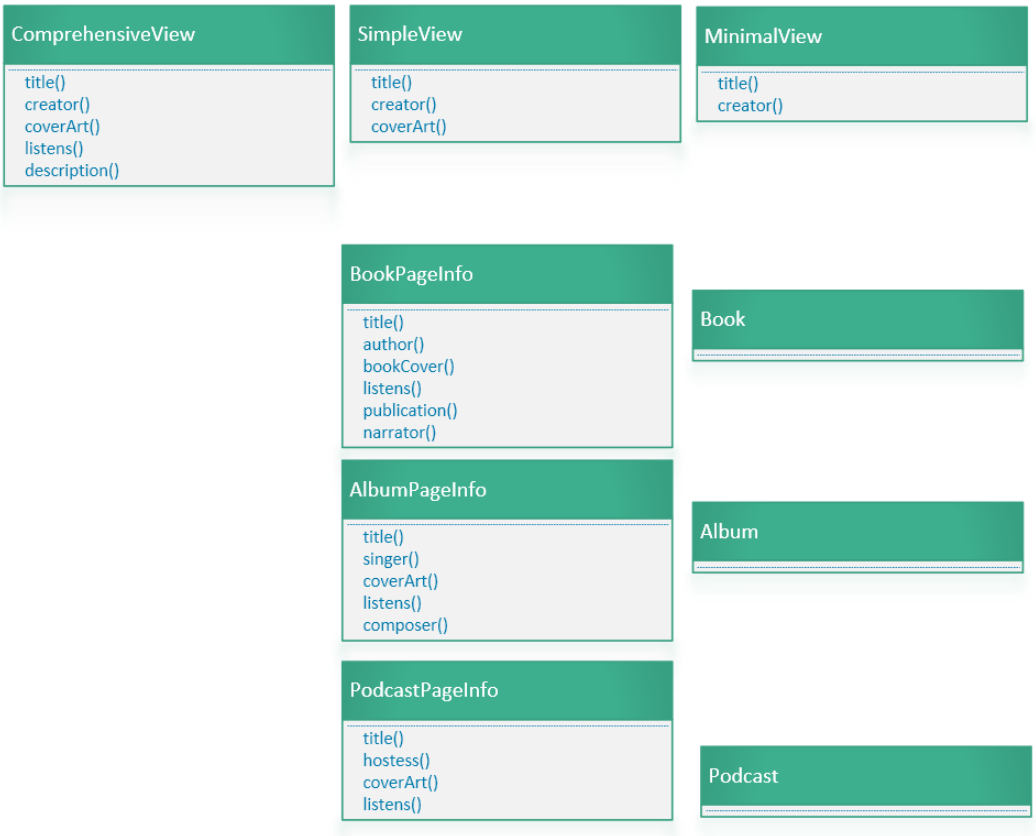


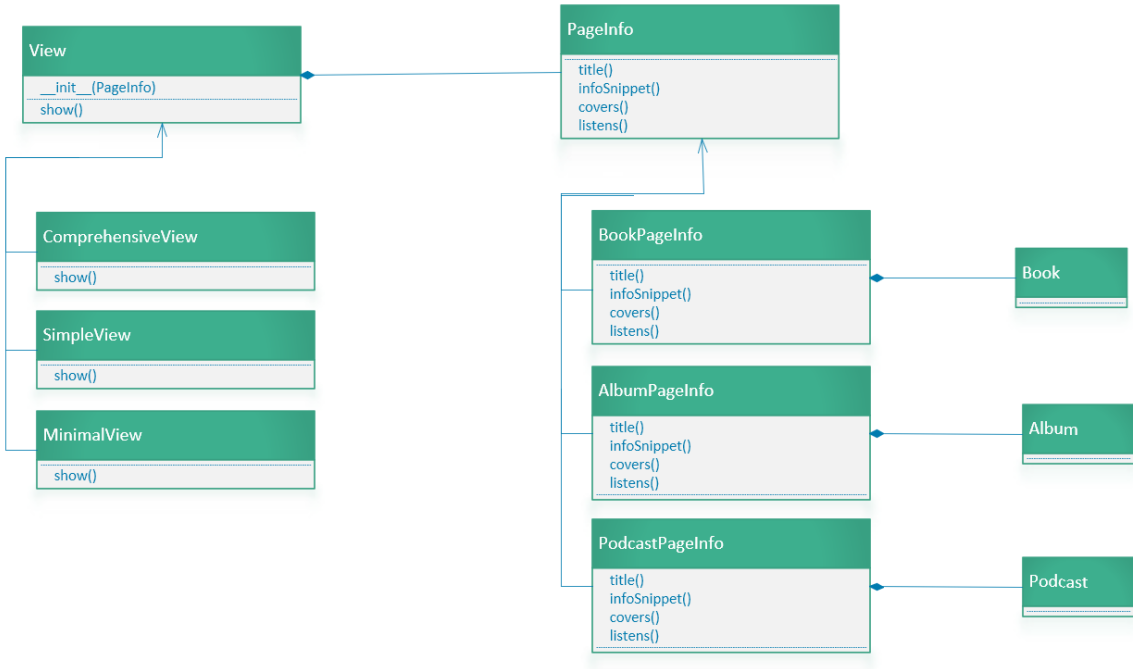
Figure 5151Primary Classes –

The first way that comes to mind for implementing different displays for different contents is to create a class for each combination of content information pages and displays. Because different content information pages differ from each other, and we cannot display the correct information with a simple display class. For example, if we have a method in the simple display class that shows the reader's name, this method does not work well for book and podcast content. So, each information page class should be combined with each display class. For a better understanding, consider the following two sets. SetS1 displays and set S2 .information pages of each content

S1 = {ComprehensiveView, SimpleView, MinimalistView}
S2={BookPageInfo, AlbumPageInfo, PodcastPageInfo}

Now, if we want to follow the method mentioned above and combine these two assemblies, we will have 9 classes! However, this approach may not be suitable because the number of classes increases ,significantly with the increase in types of content and display types .making it extremely difficult to maintain a system

The solution is to use the Bridge pattern. We need to separate the levels of abstraction and implementation from each other. So, first we need to separate the presentation and information page from the implementation



for the new system class diagram – Shape 5252

:of each

First, we create a class namedPageInfo that all classesBookPageInfo, AlbumPageInfo, PodcastPageInfo inherit from. Then, we create an abstract class namedView that has two constructor methods andshow() method. This class in its constructor method has an object ofPageInfo. Next, we create classesComprehensiveView, SimpleView, MinimalistView that inherit from classView. Now it's time to implement the classes. (To reduce complexities, we simplify the classes)

```
class Book:
    def __init__(self, title, author, narrator, fCover, bCover):
        self.title = title
        self.author = author
        self.narrator = narrator
        self.fCover = fCover
        self.bCover = bCover


class Album:
    def __init__(self, title, singer, composer):
        self.title = title
        self.singer = singer
        self.composer = composer


class Podcast:
    def __init__(self, title, hostess, episode, sponser):
        self.title = title
        self.hostess = hostess
        self.episode = episode
        self.sponser = sponser
```

Figure 5353 –Implementation of classes Book, Album, and Podcast

```python
class PageInfo:
    def title():
        pass

    def creator():
        pass

    def metaInfo():
        pass

    def cover():
        pass


class BookPageInfo(PageInfo):
    def __init__(self, book: Book):
        self.book = book

    def title(self):
        print(f"***{self.book.title}***")

    def creator(self):
        print(f"\nAuthor:{self.book.author}")

    def metaInfo(self):
        self.narrator()
        self.biography()

    def cover(self):
        print(
            f"\nfront cover:{self.book.fCover} | back cover: {self.book.bCover}")

    def biography(self):
        print("\nHere comes the biography!")

    def narrator(self):
        print(f"\The narrator is {self.book.narrator}")
```

```python
class AlbumPageInfo(PageInfo):
    def __init__(self, album: Album):
        self.album = album

    def title(self):
        print(f"***{self.album.title}***")

    def creator(self):
        print(f"\nSinger:{self.album.singer}")

    def metaInfo(self):
        self.discography()
        self.composer()

    def cover(self):
        print(f"\nAlbum cover:{self.album.cover} ")

    def discography(self):
        print("\nHere comes the discography!")

    def composer(self):
        print(f"\nThe composer is {self.album.composer}")


class PodcastPageInfo(PageInfo):
    def __init__(self, podcast: Podcast):
        self.podcast = podcast

    def title(self):
        print(f"***{self.podcast.title}***")

    def creator(self):
        print(f"\nHostess:{self.book.hostess}")

    def metaInfo(self):
        self.sponser()
        self.episode()

    def cover(self):
        print(f"\nPodcast cover:{self.podcast.cover} ")

    def sponser(self):
        print(f"\nToday's podcast's sponser is {self.podcast.sponser}")

    def episode(self):
        print(f"\nToday's episode is about {self.podcast.episode}")
```

Implementation of classes- Figure 5454 PageInfo ,BookPageInfo,AlbumPageInfo and PodcastPageInfo

```python
class View(ABC):
    def __init__(self, pageInfo: PageInfo):
        self.pageInfo = pageInfo

    @abstractmethod
    def show(self):
        pass


class ComprehensiveView(View):
    def show(self):
        self.pageInfo.title()
        self.pageInfo.creator()
        self.pageInfo.cover()
        self.pageInfo.metaInfo()


class SimpleView(View):
    def show(self):
        self.pageInfo.title()
        self.pageInfo.creator()
        self.pageInfo.cover()


class MinimalView(View):
    def show(self):
        self.pageInfo.title()
        self.pageInfo.creator()
```

Implementation of classes - Figure 5555View. ComprehensiveView, SimpleView,MinimalView

:Finally, we test the program

```
myBook = BookPageInfo(Book("Book Title", "Book Author",
                           "Book Narrator", "Front Cover", "Back Cover"))
bigscreen = ComprehensiveView(myBook)
print("Comprehensive View---------------------------------")
bigscreen.show()
smallscreen = SimpleView(myBook)
print("Simple View----------------------------------")
smallscreen.show()
myAlbum = AlbumPageInfo(Album("Album Title", "Album Singer", "Song Composer"))
minimalistic = MinimalView(myAlbum)
print("Minimal View---------------------------------")
minimalistic.show()
```

Testing the program - Figure 5656

```
Comprehensive View---------------------------------
***Book Title***
Author:Book Author
front cover:Front Cover | back cover: Back Cover
The narrator is Book Narrator
Here comes the biography!

Simple View----------------------------------
***Book Title***
Author:Book Author
front cover:Front Cover | back cover: Back Cover

Minimal View---------------------------------
***Album Title***
Singer:Album Singer
```

Output of the above code - Figure 5757

As shown above, we were able to reduce the number of classes and dependencies between them by using the bridge pattern and separating .the abstraction levels and implementing classes View and PageInfo

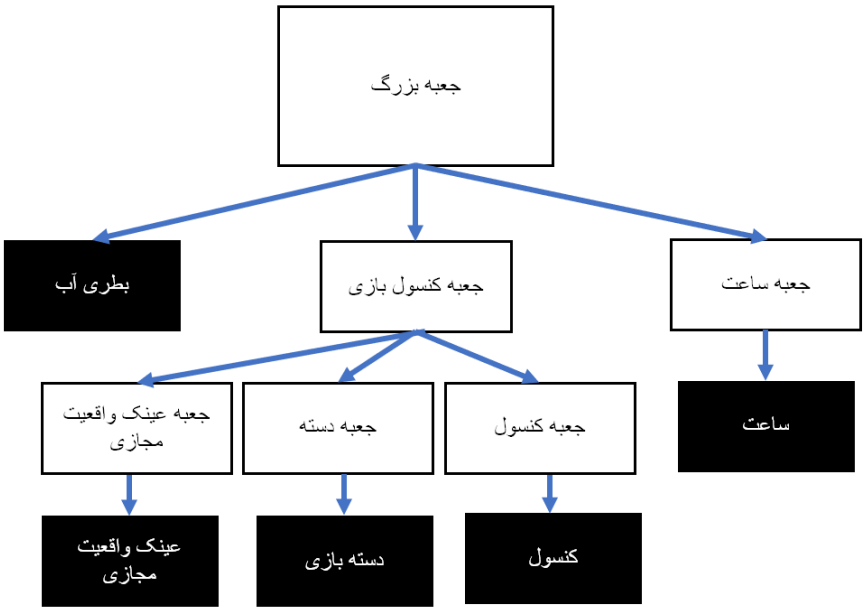
PatternComposite

Definition

The Composite pattern creates a tree structure of objects that shows the part-whole hierarchy. This pattern allows the client to treat individual .objects and compositions of objects uniformly

Objective

We have a system where each part is made up of other parts, and performing an operation on an object requires performing that operation on all of its child objects. For example, imagine you have made an online purchase and a large box containing your items arrives. Your items are a watch, a game console, and a water bottle placed in this large box. Your watch is inside another box. The water bottle does not have a box. The ,game console is inside a box that contains the console, controller charger, and virtual reality glasses, each placed in a separate box (Figure Let's say we want to check the prices of the items. So we need to .(53 first open the large box and then continue this process for each box until we reach the items. Therefore, the operation of checking the price should be applied to each box and item. This is exactly what the Composite .pattern does

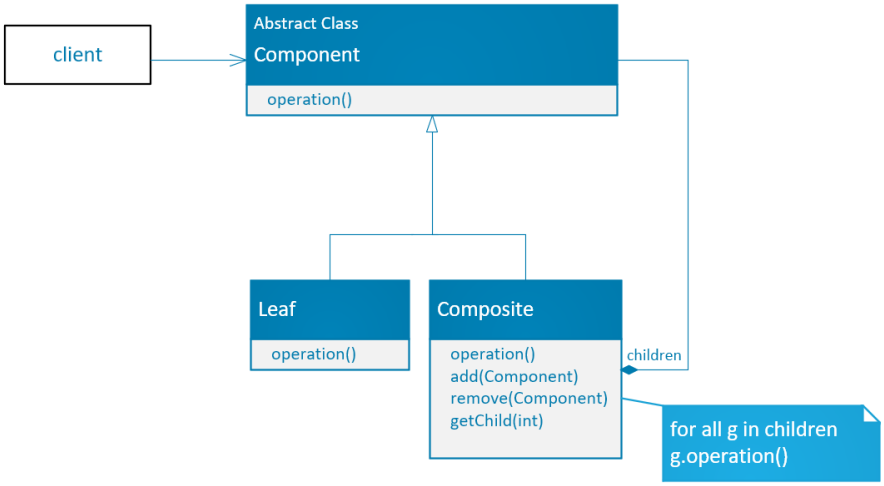


Displaying a tree-like structure of boxes and items - Figure 5858

Use cases

- .When you want to display the hierarchy of parts and whole objects
- .When your program can have a tree-like structure
- When you want the client to be able to ignore the difference between a combination of objects and individual objects. In this structure, the .client can behave the same way with all combinations of objects

Structure



class diagram of Composite Pattern - Figure 5959

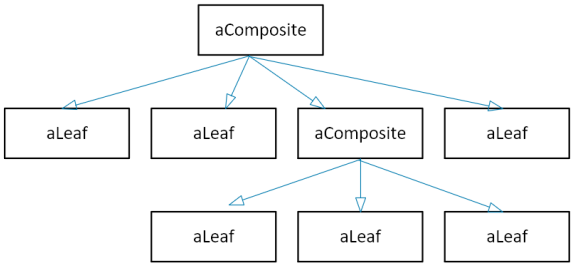


Figure 6060 – Displaying the tree of a composition of objects

Component

Defines an interface for objects in the composition, through which each object can access and manage its child objects. It also implements common behaviors among classes. In specific cases, this class can .define an interface for a child object to access the parent as well

Leaf

Represents a leaf object. A leaf object has no children. In this class, only .the primary behaviors are implemented in composition

Composite

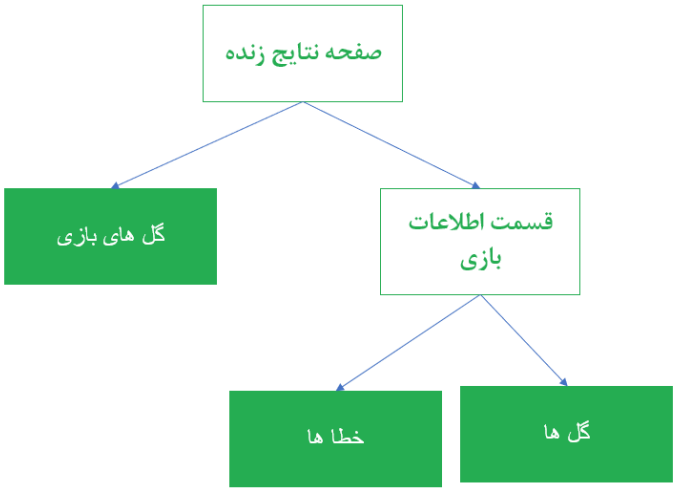
Defines behaviors of objects that have children. It also stores children within itself and implements operations related to children in the .Component class

Client

Works with objects present in the composition through the Component .interface

Example

We want to design a program that displays live football match results. On ,each match page, the names of the two teams, goals, goal scorers, fouls and videos of the goals are displayed. In each update, all information except the names of the two teams will be updated. We have decided to design the live information section in a modular way so that we can add new modules in the future or allow the user to choose the sections they :want. The system tree display will be as follows



Display System Tree - Figure 6161

.Update operations in each section above are different from each other For example, to update flowers, data should be retrieved from the flowers table, and for errors, from the errors table. But they all have an update .method

```python
class Component(ABC):
    @abstractmethod
    def refresh(self):
        pass


class Tab(Component):
    def __init__(self, tabName, modules: list[Component]):
        self.tabName = tabName
        self.modules = modules

    def add(self, module: Component):
        self.modules.append(module)

    def remove(self, module: Component):
        self.modules.remove(module)

    def refresh(self):
        list(map(lambda module: module.refresh(), self.modules))

class GoalsStats(Component):
    def refresh(self):
        print("Update Goals ⚽")


class FoulsStats(Component):
    def refresh(self):
        print("Update Fouls 🔴")


class HighlightVideo(Component):
    def refresh(self):
        print("Update Highlight videos 💥⚽")
```

Implementation of Live Results System - Figure 6262

As seen, the classTab which inherits from the ,Component class, implements therefresh method in a way that the refresh method is called for all its child .objects

:Now we test the above code

```python
print("LiveScore - Juventus vs Inter ")
goalModule = GoalsStats()
fouldModule = FoulsStats()
highlight = HighlightVideo()
StatsTab = Tab("StatsTab", [goalModule, fouldModule])

StatsTab.refresh()
highlight.refresh()
```

Running and testing the above code -Figure 6363

```
LiveScore - Juventus vs Inter
Update Goals ⚽
Update Fouls 🔴
Update Highlight videos 💥⚽
```

Output of the above code -Figure 6464

.The program works correctly

:Also, the structure of this program is as follows

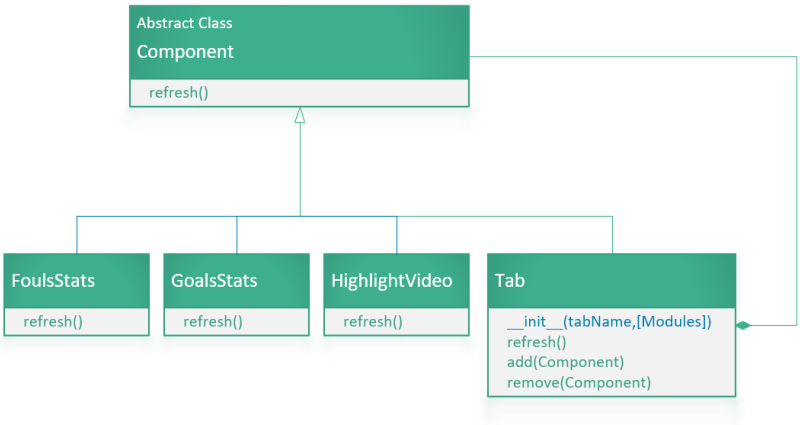


Figure 65- Class diagram of the designed program 65

Decorator Pattern

Definition

.The Decorator pattern dynamically adds new responsibilities to an object This pattern provides a more flexible and alternative way, compared to .inheritance, to enhance functionality

Objective

Sometimes we want to add new responsibilities to one or a group of objects, not a class. One way to add these responsibilities is to use inheritance and a better way is to use the combination and the Decorator pattern. Imagine you received a pair of socks as a gift for someone, but you don't want to give it to them without any decoration. You want to first put it in a box. Then wrap it around gift paper and put a ribbon on it. Using inheritance for this is like going back to the store and buying a pair of socks that comes in a box with gift paper and a ribbon. But this can be done by yourself. The box, gift paper, and ribbon are all considered gifts but they add something to the previous gift and encompass it. That's why another name for this pattern is Wrapper. The following diagram helps to :better understand the subject

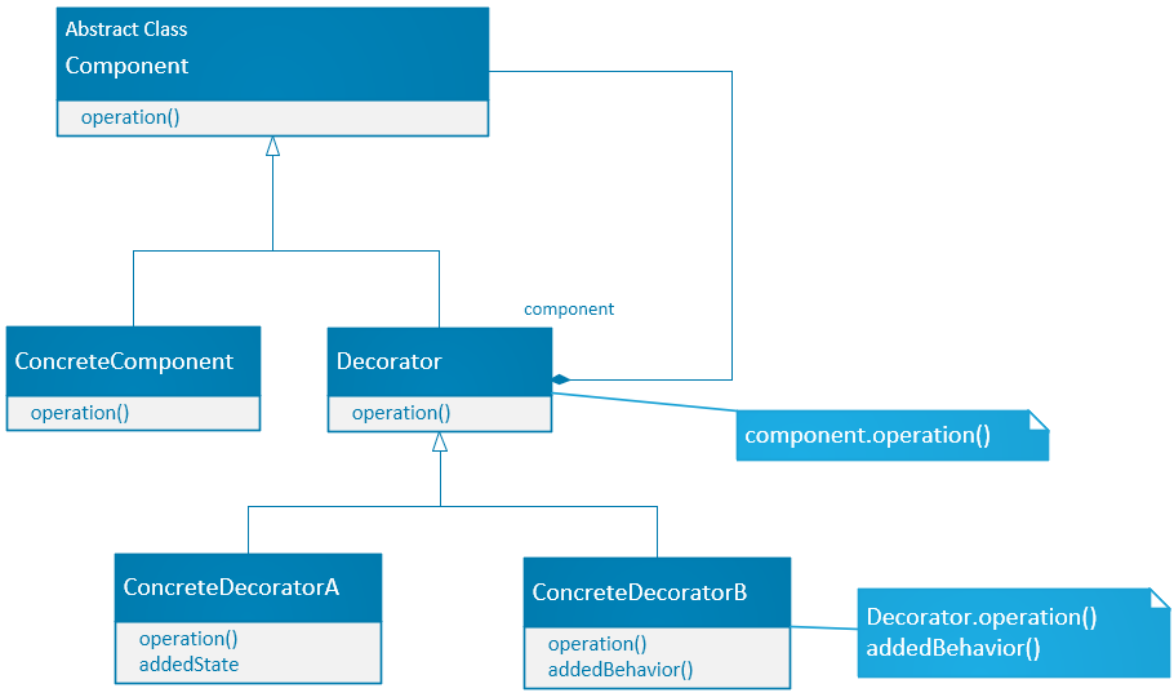
Visual representation of the above example -Shape 6666

Use Cases

- When you want to add new responsibilities to multiple objects dynamically and transparently, meaning without changing the other [3332] .objects
- .Those responsibilities can be removed from the object
- .When increasing efficiency through inheritance is impractical

[32] Dynamic
[33] Transparent

Structure



67Shape 67 - class diagram of Decorator Pattern

Component

.It provides an interface for objects that can take on new responsibilities

ConcreteComponent

.Defines an object that can have new responsibilities added to it

Decorator

An interface that inherits from the Component class and also contains an .object of it

ConcreteDecorator

Implements the Decorator interface and adds new responsibilities to the .Compneent object

Example

We want to write a browser that can display text and images. First, we create a class ElementView that any view such as image view or text view

should inherit from it and implement the show() method, then we create .two classes TextView and PictureView

```
class ElementView(ABC):
    @abstractmethod
    def show(self):
        pass


class TextView(ElementView):
    def show():
        print("Text is being shown")


class PictureView(ElementView):
    def show():
        print("Picture is being shown")
```

Shape 6868 - ,Initial implementation of ElementView TextView *and* PictureView

Now we want these views to have margins and scrolling. The simplest way is to add four methods to each display class, with the output of each being simple display, margin display, scroll display, and margin-scroll display respectively. This approach is not optimal because each time a feature is added, such as the ability to select, multiple new display ,methods need to be added. The next approach is to use inheritance which is also not optimal because for each possible combination, such as methods for example, a class must be created. For example, the class TextViewWithBorder . ... and

But using theDecorator pattern, we can design each feature as a class and then pass the displays as arguments to them. So first, we create an interface calledDecorator that inherits from the ElementView class. Then I ,implement two classesScroll and Border .

```
class Decorator(ElementView, ABC):
    def __init__(self, elementView):
        self.elementView = elementView


class Border(Decorator):
    def show(self):
        self.border()

    def border(self):
        print("-----------------------------------------")
        self.elementView.show()
        print("----------------------------------------- \n")


class Scroll(Decorator):
    def show(self):
        self.scroll()

    def scroll(self):
        print("Scroller Added ++++++++++++++++++++++++++++++++")
        self.elementView.show()
        print("\n")
```

Decorator Figure 69 - Implementing the 69, Scroll , and *Border* classes

```
textWithBorder = Border(TextView())
textWithBorder.show()

textwithScroll = Scroll(TextView())
textwithScroll.show()

pictureWithBorderAndScroll = Scroll(Border(PictureView()))
pictureWithBorderAndScroll.show()
```

Shape 70 Running the program -

```
---------------------------------------------
Text is being shown
---------------------------------------------

Scroller Added ++++++++++++++++++++++++++++++++++
Text is being shown


Scroller Added ++++++++++++++++++++++++++++++++++
---------------------------------------------
Picture is being shown
---------------------------------------------
```

Output of the above code -Shape 7171

As seen, we were able to add scrolling and margins to displaying text and .images without creating additional classes using the Decorator pattern

Facade pattern

Definition

This pattern defines a unified interface for a set of interfaces in a .subsystem. This high-level interface facilitates the use of this subsystem

Objective

Dividing the structure of a system into multiple subsystems reduces complexity. A common goal in design is to reduce communications and dependencies between subsystems. One way to achieve this goal is to use a Facade that provides a simple interface for the overall features of .a subsystem

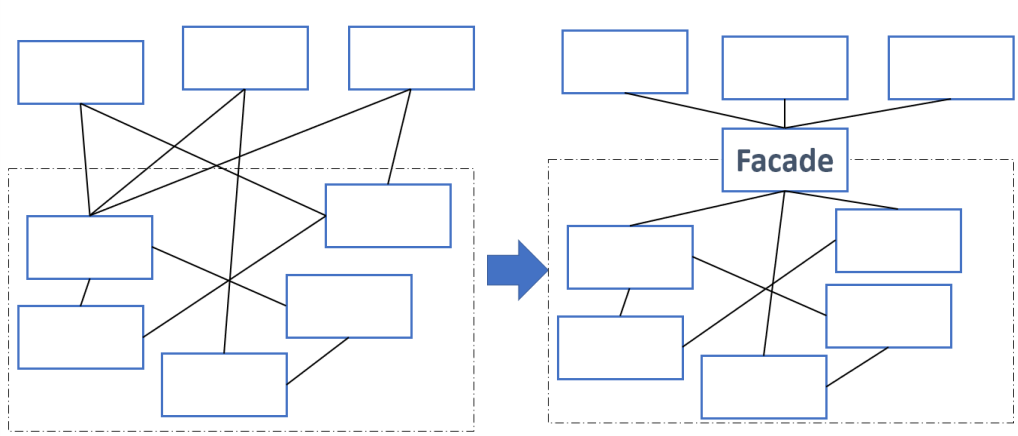


Figure 72 - Reducing the complexity of class communications using the Facade pattern 72

It is also better to pay attention to the minimum knowledge rule in designing subsystems. This rule states that a method of an object should [34]:only be used in four cases

1. .The method is part of the object itself
2. The method belongs to an object that is passed as an input .parameter to the method
3. .The method belongs to any object that the current method creates
4. The method belongs to any part of the object (such as calling a .(method of an object stored in a class

Use Cases

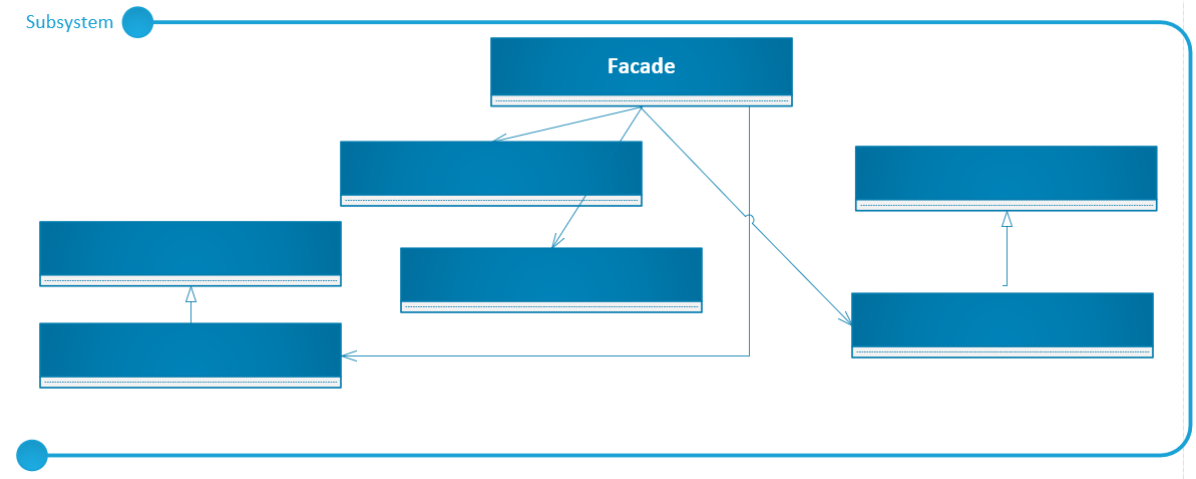
[34] Principle of Least Knowledge

- .When you want to provide a simple interface for a complex subsystem .As subsystems are developed further, their complexity increases Design patterns often result in smaller and more classes. This increases the reusability of the subsystem and makes customization easier, but it complicates the work for a customer who does not need customization. A facade is a simple default view of a subsystem that is sufficient for most customers. Only customers who need more .customization should use something more than a facade

- When the number of dependencies between the client and implementation classes of an abstraction is high, a facade should be ,used to separate subsystems from clients and other subsystems .improving the independence and replaceability of subsystems

- When you want to layer your subsystem, use a facade to create an entry point for each level of the subsystem. If subsystems are interdependent, you can easily simplify this dependency by restricting [35] .the communication between them only through their facades

Structure



- Shape 7373class diagram of Façade Pattern

Facade

This interface knows which classes of a subsystem are responsible for .what tasks and delegates client requests to them

Subsystem classes

[35] Entry Point

.These are the classes that implement the functionality of a subsystem They can both directly perform client work and work delegated to them ,by the Facade object. However, they have no knowledge of the Facade [36] .so they do not hold any reference to it within themselves

Example

,We want to design a program that captures input data from a camera processes it, edits it, and outputs it (in jpg format). It is clear that our customer is a camera program. We want to design the system in a way that the camera program itself decides whether it wants to go through all .these steps or just wants the processed image to be displayed to the user But most of the time, all three cases above need to be done. We also .want to be able to add new features to it in the future

From the explanations of this system, we understand that the desired pattern is Facade. The design and implementation of our system is as .follows

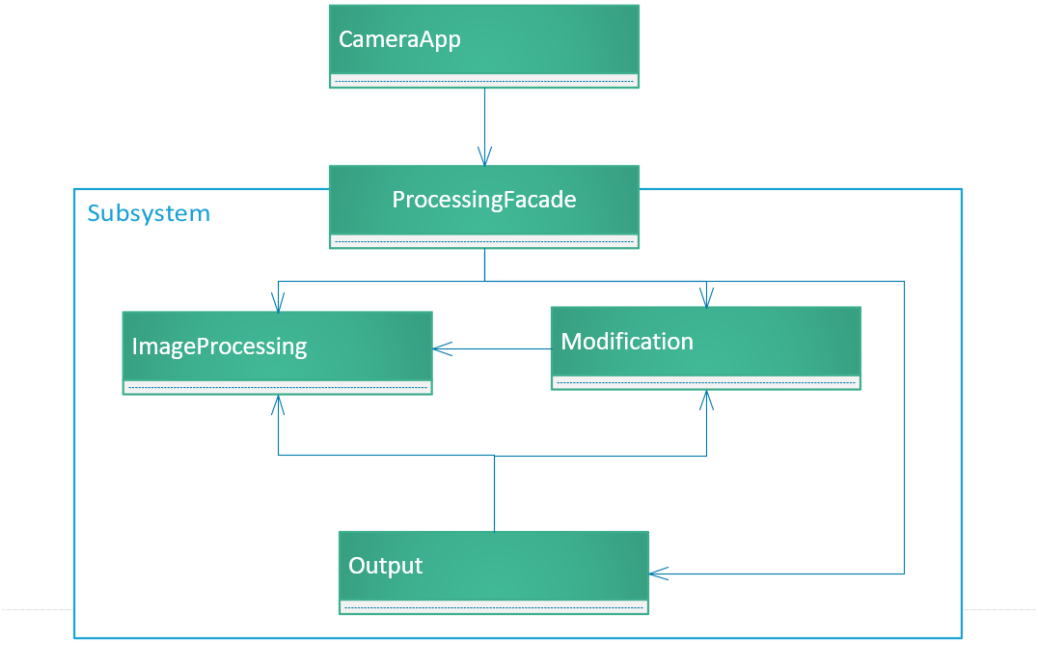


Figure 74 - class diagram of the designed system

[36] Reference

```python
class CameraApp:
    def __init__(self, data):
        self.data = data

    def takePicture(self):
        self.facade = ProcessingFacade(self.data)
        print(self.facade.output())


class ImageProcessing:
    def __init__(self, rawData):
        self.rawData = rawData

    def process(self):
        self.result = f"Processed Data+-{self.rawData}"
        return self


class Modification:
    def __init__(self, imageProcessing):
        self.imageProcessing = imageProcessing

    def modify(self):
        self.result = f"Modified Data**{self.imageProcessing.result}"
        return self


class Output:
    def __init__(self, finalData):
        self.finalData = finalData
    def export(self):
        return "Image Ready! output.jpg"


class ProcessingFacade:
    def __init__(self, rawData):
        self.processed = ImageProcessing(rawData).process()
        self.modified = Modification(self.processed).modify()
        self.result = Output(self.modified).export()

    def output(self):
        return self.result


camera = CameraApp("10101001111101110011110110010110111101010110")
camera.takePicture()


onlyProcessedImage = ImageProcessing(
    "10101001111101110011110110010110111101010110").process().result

print(onlyProcessedImage)
```

Figure 75 Implementation of the above system -

```
Image Ready! output.jpg
Processed Data+-10101001111101110011110110010110111101010110
```

Output of the above code - Figure 7676

PatternProxy

Definition

PatternProxy provides a substitute for another object in order to control .access to it

Objective

One of the reasons we want to control access to an object is to delay the cost of creating that object until we actually need it. For example, suppose ,we want to design a document and editor. Creating some graphic objects such as raster images, takes a long time, but we want the program to open quickly, so we should avoid creating expensive objects at the beginning of the work. In any case, creating all graphic objects at the beginning is not reasonable because not all of those objects are [37] .supposed to be displayed at the same time

Given these constraints, we should only create an object when it needs to be displayed. The solution is to use a proxy - Proxy - that acts as a substitute for the real object. This proxy behaves just like an image and .instantiates it when needed

Use Cases

The Proxy pattern is used when we need a complex or multi-functional reference instead of a simple reference. Some situations where this :pattern is useful are listed below

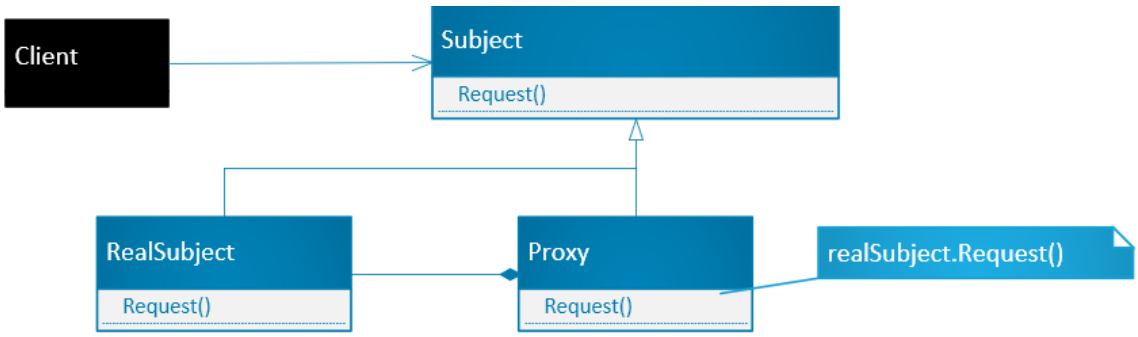
- A proxy, a local representation of an object in a different address [38] .space, provides a surrogate or a placeholder for another object
- A virtual proxy creates expensive objects on demand. The proxy in [39] .the above example is of the same type
- [40]A protective proxy controls access to the original object. Protective proxies are most useful when objects need different access levels. The kernel proxy in the Choices operating system provides protected access to operating system objects.

[37] Document Editor
[38] Remote Proxy
[39] Virtual Proxy
[40] Protection Proxy

- A smart pointer is a replacement for a pointer that performs additional operations when accessing an object. Some common [41]:uses include

  - Counting the number of pointers pointing to the original object so that when there are no more references, it can free it from .memory
  - Loading a permanent object into memory when it is first referenced
  - Checking if the main object is locked before accessing it, so that .another object cannot modify it

Structure



77Figure 77- Class diagram of Proxy Pattern

78Figure 78- Object diagram of Proxy Pattern

Proxy

.It holds a reference to the main object that allows access to the object Proxymay refer to a Subject if RealSubject and Subject are the same. It is also an intermediary that is consistent with theSubject interface, so a proxy can be a substitute for the real object. This class also controls access to the .main object and is responsible for creating and deleting it

:Other responsibilities of this class can be as follows

[41] Smart reference

- Remote proxies handle the encryption of a request and its arguments, and send the encrypted request to the real object in a different address space.[42]
- Virtual proxies must store additional information about the real object in the cache memory to delay access to it. In the example mentioned in the 'Objective' section, the proxy stores the dimensions of a real image in the cache memory.[43]
- Protective proxies check whether the caller has permission to execute a request or not.

Subject

Defines an interface for RealSubject and Proxy so that a Proxy can be used anywhere instead of RealSubject.

RealSubject

Defines the real object that the proxy represents.

Example

As mentioned in the 'Intent' section, we want to write a text editor that loads raster images whenever it encounters them. A proxy for an image must have its dimensions so that the editor can properly handle spacing and pagination when loading the document.

To start the implementation, first we create an abstract class Element in which the function render() is defined.

```
class Element(ABC):
    @abstractmethod
    def render():
        pass
```

Figure 79- Implementation of the Element class[79]

Then we implement the classes Text ,Picture and , PictureProxy:

[42] Encrypt
[43] Cache

```
class Text(Element):
    def __init__(self, text):
        self.text = text

    def render(self):
        print(f"Rendered Text: **** \n {self.text}")


class Picture(Element):
    def __init__(self, pic, height, width):
        self.pic = pic
        self.height = height
        self.width = width

    def render(self):
        print(
            f"\nRendered Picture:***  \n {self.pic}  -{self.height} * {self.width}")


class PicutreProxy(Element):
    def __init__(self, picture):
        self.picture = picture

    def render(self, xFactor):
        if(self.check_access(xFactor)):
            self.picture.render()
        else:
            print(
                f"\nProxy ! : \n there's a picture with the title {self.picture.pic} and the dimentions of {self.picture.height} * {sel
                picture.width} ")

    def check_access(self, xFactor):
       # Proxy: Checking access prior to firing a real request
        if(xFactor):
            return True
        else:
            return False
```

Figure 80 *Classes Text, Picture, and PictureProxy -*

The method checkAccess in the class PictureProxy checks whether the original picture should be rendered or its proxy. The argument xFactor determines .this condition xFactor ,can be any determining factor such as page number approval requirement for the picture, etc. For simplicity, we considered a [44] .binary value for this factor

:Now we test and run the program

```
someText = Text(
    "hey this sis a sample text. you can ignore it but i wouldn't recommend ou to do so.")

aPicture = Picture("Picture of a Cow", 1080, 720)

aPictureProxy = PicutreProxy(aPicture)

someText.render()
aPictureProxy.render(False)
aPictureProxy.render(True)
```

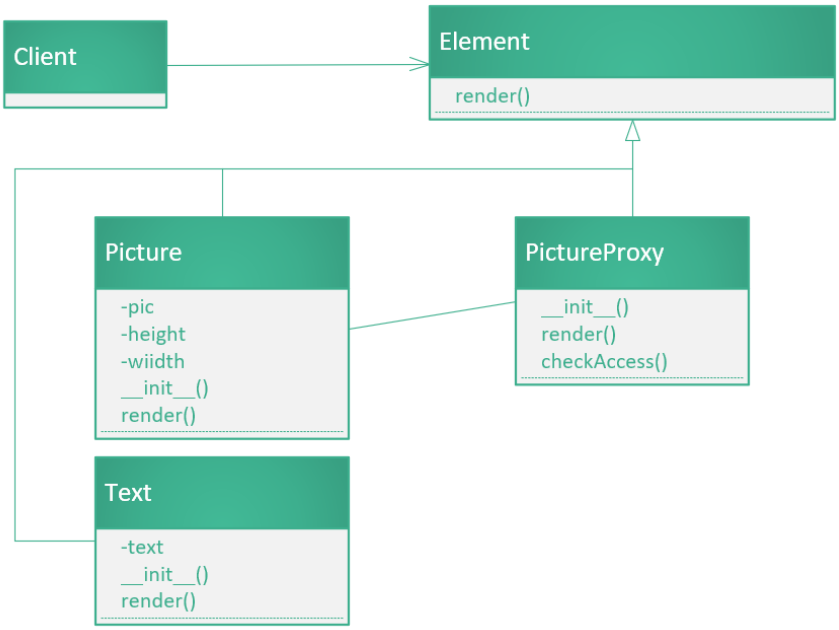
81 *Figure81- Testing and running the program*

Output of the above code -Figure 8282

```
Rendered Text: ****
 hey this sis a sample text. you can ignore it but i wouldn't recommend ou to do so.

Proxy ! :
 there's a picture with the title Picture of a Cow and the dimentions of 1080 * 720

Rendered Picture:***
 Picture of a Cow  -1080 * 720
```

:The designed program structure is as follows

[44] Boolean



83 *Figure83- Program class diagram*

Flyweight Pattern

Definition

The Flyweight pattern uses sharing to support the efficient use of a large .number of objects with extensive details

Objective

In some applications, there is a need to use objects throughout the program, but a simplistic implementation can become cumbersome. For example, in a document editor, each character may be considered as an object. Although this approach provides great flexibility to the program, it is very costly to implement. The Flyweight pattern explains how to share [45].objects for their use at different data levels without heavy costs

A lightweight object - flyweight - is a shared object that can be used in multiple contexts simultaneously. A flyweight object acts as an independent object in each context - like an instance of an object that is not shared. A flyweight object cannot predict the context in which it is used. The key point here is the difference between internal and external state. Internal state is information shared among all objects, while external state is unique information for each object. Internal state is stored in the flyweight object. This state includes information independent of the flyweight context, making it shareable. External state depends on the flyweight context and therefore cannot be shared. Client objects are responsible for transferring external state to the flyweight object when [4746] .needed. These cases are explained in detail in the example section

Use Cases

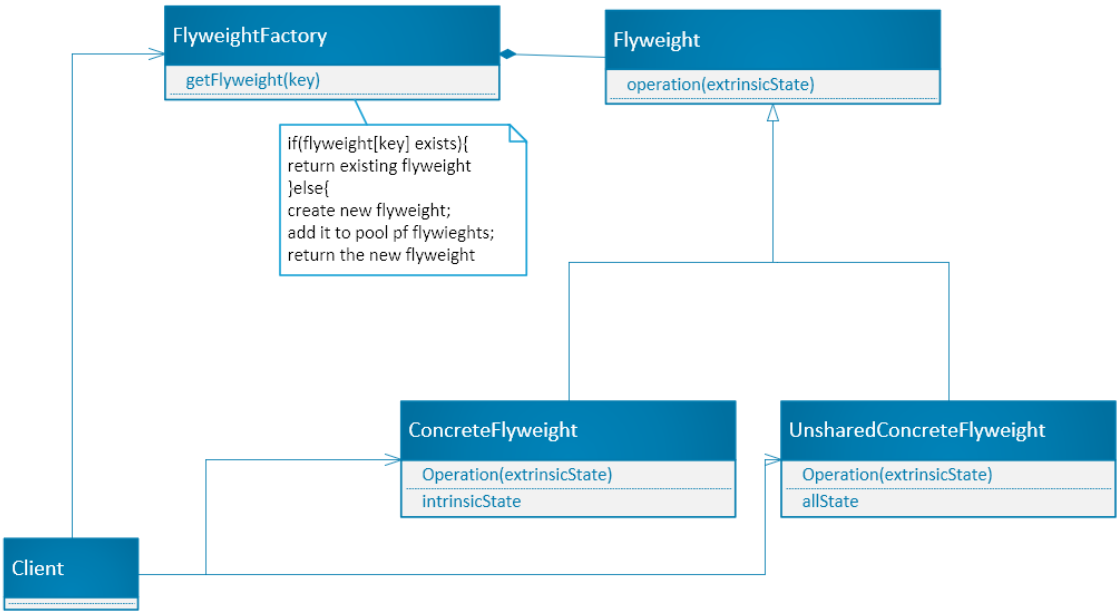
The efficiency of the Flyweight pattern largely depends on where and how :it is used. Only use this pattern when all 5 conditions below are met

1. .A program consists of a large number of objects
2. .The storage cost is high due to the large number of objects
3. .Most objects can be extrinsic
4. A large group of objects, when their extrinsic state is removed, can .be replaced by multiple shared objects

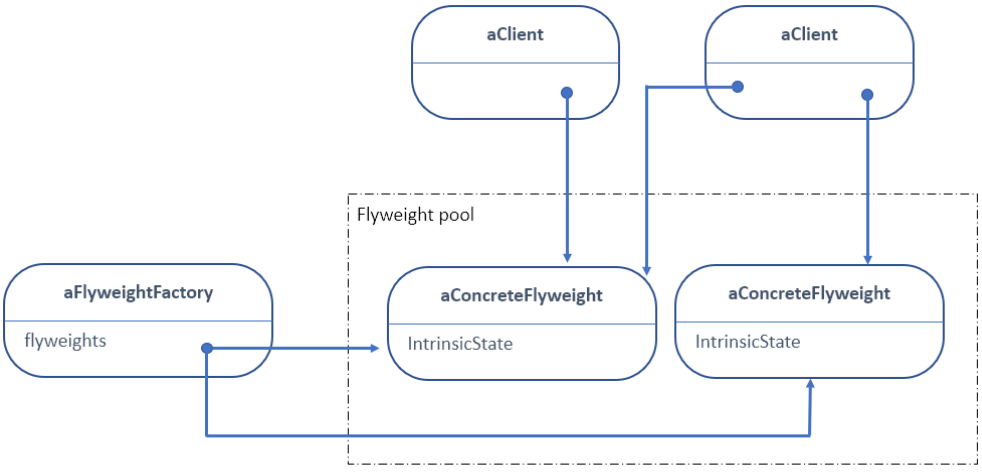
[45] Granularities
[46] Intrinsic
[47] Extrinsic

5. .The program should not be dependent on the identity of the object Since lightweight objects need to be shared, identity tests always .return True for different objects

Structure



84Figure 84 - class diagram of Flyweight pattern



85Figure 85 - object diagram of Flyweight pattern

Flyweight

Defines an interface through which flyweights can receive external state .and perform operations on it

ConcreteFlyweight

The Flyweight pattern implements sharing and provides the potential for space optimization of internal state. A ConcreteFlyweight object must be shareable, and any state it stores must be intrinsic, meaning that this state .must be independent of the object's position

UnsharedConcreteFlyweight

Not all subclasses of Flyweight need to be shared. The Flyweight interface does not mandate this. Having a ConcreteFlyweight object as a child for UnsharedConcreteFlyweight at some structural levels is .common

FlyweightFactory

It creates and manages lightweight objects. It also ensures the sharing of lightweight objects. When a client requests a lightweight object, the FlyweightFactory provides an existing instance, otherwise a new instance.

Client

It has a reference to lightweight objects. It also calculates and stores their .external state

Example

[48] We want to write a text editor in which each letter is an object that stores information such as ASCII code, typography, color, size, and coordinates .within itself. Suppose we want to open a file that has 100,000,000 words Assume that each word is on average composed of 5 letters. So with these conditions, we have 5000 letters for which we need to create an .object for each one. We have simulated this using a for loop

[48] ASCII code

```
class Letter:
    def __init__(self, ASCIICode, typography, color, size, coordinate):
        self.ASCIICode = ASCIICode
        self.typography = typography
        self.color = color
        self.size = size
        self.coordinate = coordinate

    def draw(self):
        pass


start_time = time.time()
for i in range(500000):
    globals()[f"object1{i}"] = Letter("69", "A", "black", "12", [i, i])

process = psutil.Process(os.getpid())
print(
    colored(f"--- CPU TIME ----{(time.time() - start_time)} seconds", "red"))

# in Megabytes
print(
    colored(f"---- Memory Used---- {process.memory_info().rss/1048576} MB", "red"))
```

Sample simulation of characters using a – Figure 8686for loop

```
CPU TIME
1.5897488594055176 seconds
Memory Used
 220.6484375 MB
```

Figure 87 Resources used to run this code –

As can be seen, creating this number of objects is very expensive and occupies 220 megabytes of RAM. To solve this problem, first we need to ,note that many of the data of these objects, such as typography, color size, and ASCII code, are the same. However, some data like variable coordinates are different. What we need to do is consider a set of objects that have a lot of similar data as a new object. For example, consider the letterA with the color black, size 14, and typography Times New Roman as a sample. The only data that is different for members of this set is their coordinates. So, the properties of ASCII code, typography, size, and color are internal state, and the coordinates are external state. Now we need to design a lightweight object that stores examples like this example and whenever we need to call and use that example, we transfer the external ,state from the argument shape to the appropriate method of that class which in this case is thedraw .method

First, we create a lightweight class namedGlyph that stores internal states and defines a methoddraw .in it

```python
class Glyph(ABC):
    def __init__(self, ASCIICode, typography, color, size):
        self.ASCIICode = ASCIICode
        self.typography = typography
        self.color = color
        self.size = size

    @abstractmethod
    def draw(self, coordinate):
        pass
```

Shape 8888 - Implementing the class Glyph

:Implementing the class Charcater

```python
class Charcater(Glyph):
    def draw(self, coordinate):
        #print(f"{chr(int(self.ASCIICode))} at {coordinate} ({self.typography})")
        pass
```

Shape 8989 Implementing the class - Charcater

Now it's time to design the class GlyphFactory. This class, when the software needs to display, for example, the black letter B with the font Times and size 14 at coordinates (1,1), first checks if an object with such specifications exists or not. If it exists, it calls the draw method and passes the coordinates to it. But if an object with these specifications does not exist, it first creates a Glyph object, stores it in glyphContext, and returns .that object

```python
class GlyphFactory:
    _glyphContext: Dict[str, Charcater] = {}

    def getCharacter(self, ASCIICode, typography, color, size):

        if self._glyphContext.get(f"{ASCIICode}{typography}{color}{size}"):
            return self._glyphContext.get(f"{ASCIICode}{typography}{color}{size}")
        else:
            newCharacter = Charcater(ASCIICode, typography, color, size)
            self._glyphContext[f"{ASCIICode}{typography}{color}{size}"] = newCharacter
            return newCharacter
```

Shape 9090 – Implementation of the class GlyphFactory

At the end, we define a function called draw that uses the defined glyphFactory and each time it is called, it stores an object, whether it is a new object or a duplicate, in a variable whose name changes based on the value of i .

```python
glyphFactory = GlyphFactory()


def draw(ASCIICode, typography, color, size, coordinate, i):
    globals()[f"object1{i}"] = glyphFactory.getCharacter(
        ASCIICode, typography, color, size)
    globals()[f"object1{i}"].draw(coordinate)
```

Shape 9191 – Implementation of the function draw

:Now, using a for loop, we simulate the execution

```
start_time = time.time()
for i in range(500000):
    draw("69", "Times", "black", "14", (i, i), i)

process = psutil.Process(os.getpid())
print(
    colored(f"CPU TIME \n{(time.time() - start_time)} seconds", "red"))

# in Megabytes
print(
    colored(f"Memory Used \n {process.memory_info().rss/1048576} MB", "red"))
```

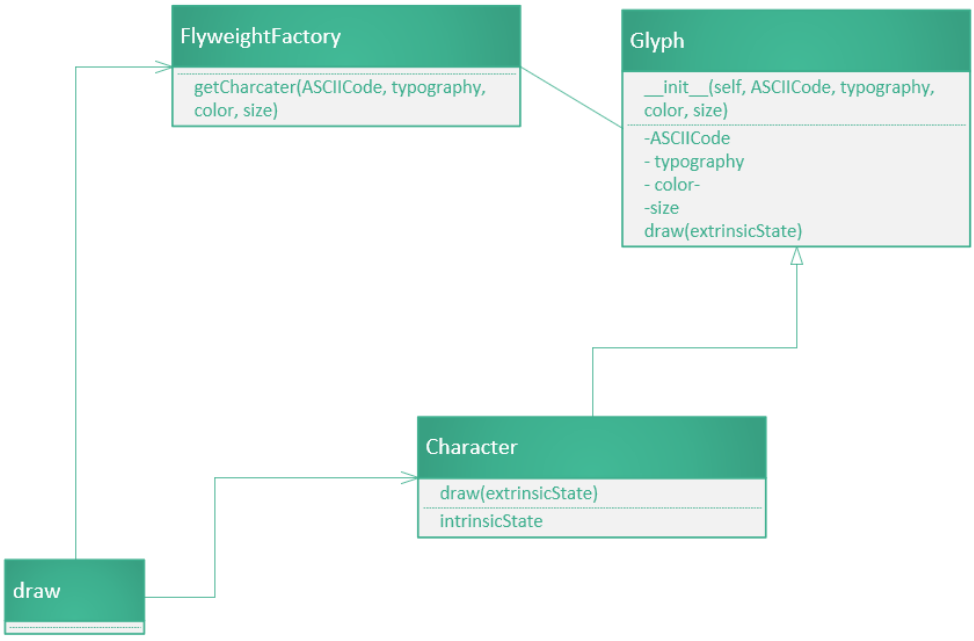
Figure 92- Implementation of the for loop 92

```
CPU TIME
1.881305456161499 seconds
Memory Used
 89.0 MB
```

93*Figure 93 - Resources used for running this code*

As can be seen, the memory consumed in the new program is three times .less than the memory consumed in the initial program

:The structure of the new program is as follows



94*Figure 94 - Class diagram of the new program*

Behavioral Patterns

Behavioral patterns are related to algorithms and the assignment of responsibilities between objects. Behavioral patterns describe not only patterns of objects and classes but also patterns of relationships between them. These patterns describe complex control flows that are difficult to follow at runtime. They remove your focus from control flow and allow you .to concentrate only on how objects interact

Class behavioral patterns use inheritance for distributing behavior among classes. Object behavioral patterns leverage object composition for this purpose. Some of them describe how objects collaborate to perform .operations that could not be done by any single object alone

Chain Of Responsibility Pattern

Definition

.This pattern prevents coupling the sender of a request with its receiver This is done by allowing multiple objects to handle a request. Objects are chained to each other and pass the request along the chain. This [49] .continues until an object in the chain can handle the request

Objective

Consider a sensitive assistant for text in the graphical user interface. By clicking on any part of this interface, the user can obtain information about that part. These descriptions vary in different sections. For example, a button in a dialog box has different descriptions than a similar button on the main page. If that part has no descriptions, more general information about the environment the user is in should be displayed. (In this [50](example, the main page

Therefore, sorting descriptions based on their generality - from the most specific to the most general ones – is a natural thing. Furthermore, it is clear that a help request is managed by one of several user interface objects; a request that depends on the specificity and extent of the .available descriptions

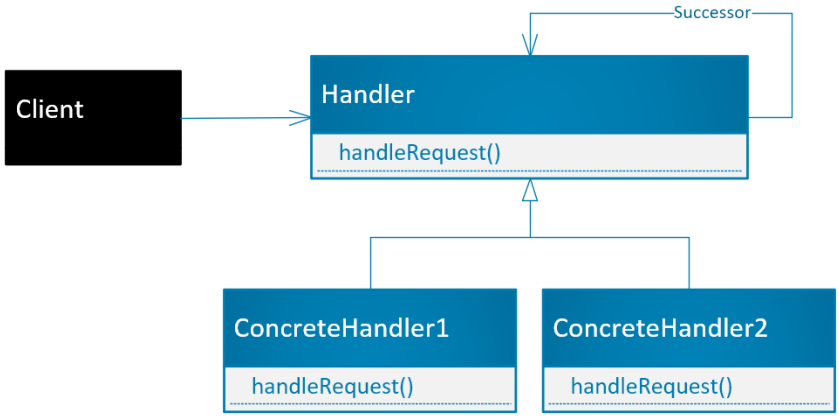
But the problem here is that the object providing the help descriptions is not recognized by the object initiating the help request (like a button). So we need a way to separate the button that sends the request for help descriptions and the object that provides those descriptions. TheChain of Responsibility.pattern explains how to do this

Use Cases

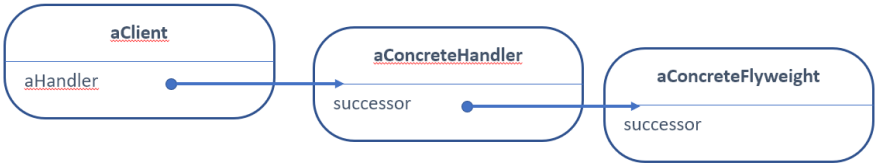
- When more than one object may handle a request and the handler .is not known. The handler should be automatically determined
- When you want to send a request to one of several objects without .specifying the receiver explicitly
- When a set of objects can handle a request, they must be .dynamically identified

[49] Coupling
[50] Context-sensitive

Structure



-Figure 9595class diagram of Chain Of Responsibility Pattern



-Figure 9696Object diagram of Chain Of Responsibility Pattern

Handler

Creates an interface for handling requests. It can also implement the .successor relationship

ConcreteHandler

Manages the request that is its responsibility. It can also have access to its successors. If the ConcereteHandler can handle the request, it will do .so. Otherwise, it will pass the request to its successor

Client

.Creates a request forConcreteHandler

Example

As mentioned in the 'Objective' section, we want to design a text-sensitive .assistant that provides explanations when the user clicks on any object An object can have different explanations in different positions. For

example, a button on the main page has a different explanation than a button on the comment submission page. Also, if an object does not have an explanation, the parent's explanation should be displayed, which in .this example can be the main page or the comment submission page
So we have a chain (tree) structure in which a request is sent sequentially .to the next nodes and executed and managed in the appropriate node

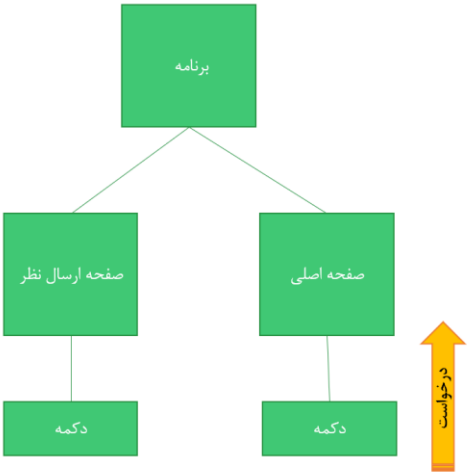


Figure 9797- Software Tree Structure

Each of the above components is considered a request manager. So we :need a class that all of these components, classes, inherit from it

```
class HelpHandler(ABC):
    _successor: HelpHandler

    def __init__(self, successor=None):
        self._successor = successor

    @abstractmethod
    def showInfo(self):
        pass

    def handleHelpInfo(self):
        if(self.condition()):
            self.showInfo()
        else:
            if(self._successor):
                self._successor.showInfo()
            return False

    def condition(self):
        if(hasattr(self, 'info')):
            return True
        else:
            return False

    def setInfo(self, info):
        self.info = info
```

Shape 98 *Implementation of the class -HelpHandler*

The method handleHelpInfo is responsible for managing the request. In a way that if theinfo attribute does not exist in an object, the showInfo method of its parent will be called instead. This method is implemented in each of the subclasses of this class, which in this example are

Application ,CommentPage and ,Button The .setInfo method is implemented to assign descriptions to the desired object in theHelpInfo .class

The implementation of classesApplication ,CommentPage and ,Button is :as follows

```
class Application(HelpHandler):

    def showInfo(self):
        print(f"Application: {self.info}")


class CommentPage(HelpHandler):
    def showInfo(self):
        print(f"CommentPage: {self.info}")


class Button(HelpHandler):
    def showInfo(self):
        print(f"Button: {self.info}")
```

99Figure 99 - Implementation of classes Application, CommentPage, and Button

Now we test the program. First, we create an object of the program that has no parent. Then we create a comments page and two buttons on the comments page, with the first button having no descriptions and the .second button having descriptions

```
application = Application()
application.setInfo("This is just a test application")
commentPage = CommentPage(application)
commentPage.setInfo("You can send your comment in this page")
button1 = Button(commentPage)
button2 = Button(commentPage)
button2.setInfo("this is the second button which includes help info")
button1.handleHelpInfo()
button2.handleHelpInfo()
```

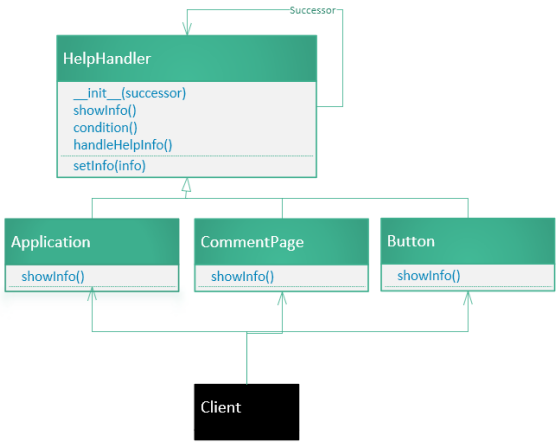
Test and run program -Shape 100100

```
CommentPage: You can send your comment in this page
Button: this is the second button which includes help info
```

Shape 101 *Output of the above code -*

,As expected, when we click on the first button, the parent's description which is the comments page, is displayed, and when we click on the .second button, the button's own description is displayed

:The program structure is as follows



of the program - class diagramShape 102102

PatternCommand

Definition

The Command pattern encapsulates a request as an object. This allows ,you to encapsulate clients with requests, queue or log requests [51] .parameterize requests, and support undoable operations

Objective

Sometimes it is necessary to send requests to objects without the ,requester or the receiver having any information about it. For example user interface toolboxes contain objects such as buttons and menus that execute a request based on user input. However, the toolbox cannot explicitly implement the request in the button or menu; only the program that uses that toolbox is aware of the task of that button or menu. As designers of this toolbox, we have no way of knowing the information [52] .about the request and its receiver

The Command pattern allows the toolbox to generate requests from unspecified programs. In this approach, the request is transformed into an object. This object can be stored or transferred. The main idea of this pattern is an abstract class Command that defines an interface for executing operations. In its simplest form, this interface has an abstract method execute. Concrete Command subclasses specify a receiver-action pair, which is done by storing the receiver in an instance variable and implementing execute to call the request. The receiver has the .necessary knowledge to execute the request

Use Cases

- When you want to parameterize objects with an executable operation. You can parameterize such operations in a procedural language using a recursive function, which is a function that is registered to be called at a later time. Commands in object-oriented .programming are equivalent to recursive functions

[51] Parameterize
[52] User interface toolkits

- [53] When you want to schedule, queue, and execute requests at different times. A Command object can have a life independent of the original request. If the recipient of a request can be displayed as an address independent of distance, you can transfer a Command object for that request to another process and perform .the request there

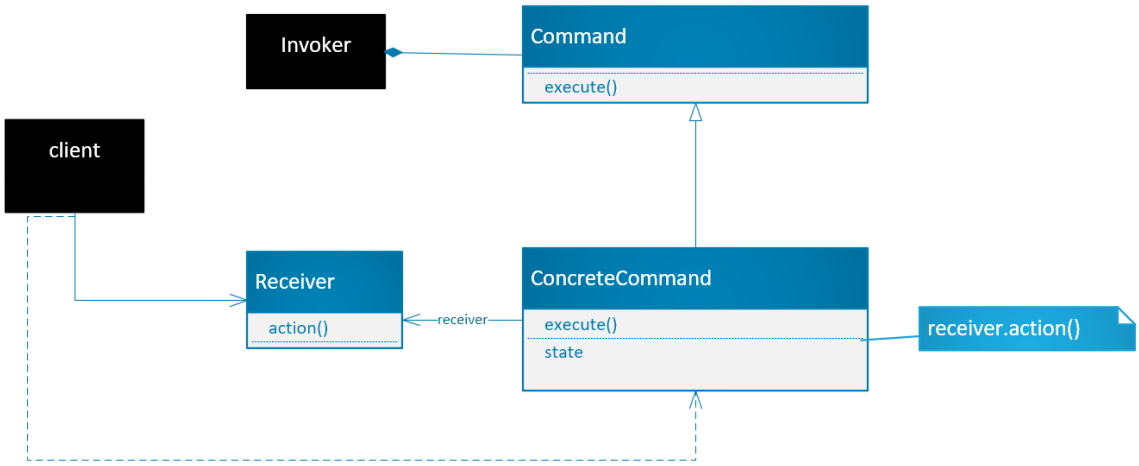
- [54] When you want to support undo, The Execute operation in Command can store the state for undoing its effects in the Command object. The Command interface must have an Unexecute operation that reverses the effects of a previous Execute invocation. Executed commands are stored in a history list. By moving forward and backward in this list, unlimited undo and redo .can be achieved

- When you want to support logging changes so that they can be reapplied in case of system failure. You can maintain a permanent report of the changes made with the reliability of the Command interface by loading and storing operations. The recovery process after a failure involves reloading the recorded commands from disk .and re-executing them with the Execute operation

- When you want to structure a high-level operation system based on the first operation, structure it. Several structures in information systems that support transactions are important. A transaction involves a set of changes in data. The Command pattern provides a way to model transactions. Commands have a common interface that allows you to invoke all transactions in a uniform manner. This pattern also facilitates the extension of the system with separate .transactions

[53] Lifetime
[54] Undo

Structure



Shape 103103- class diagram of Command Pattern

Command
Defines an interface for executing an operation

ConcreteCommand
Defines a binding between a receiver object and an action. It also [55] implements the execute method by calling the corresponding operation .on the receiver object

Client
Creates a ConcreteCommand object and specifies its receiver

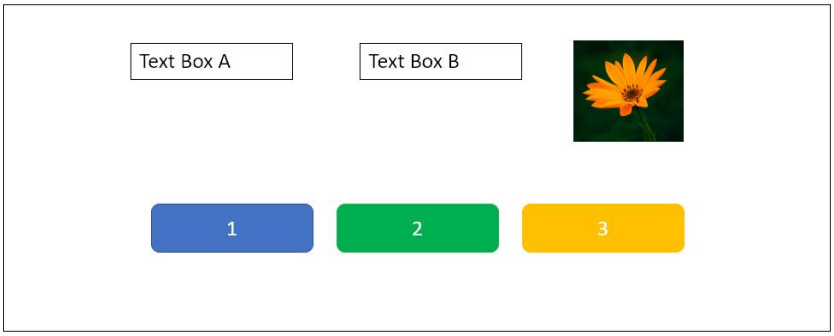
Invoker
Asks the Command. to execute the request

Receiver
The way operations related to executing a request are executed is .important. Any class can act as an operation receiver

Example
Suppose we want to design a website where users can design their own website using it. In a part of this website, the user can add buttons and define the operations that each button performs. Also, the user must .specify on which element these operations will be applied

[55] Binding

Assume the user creates three buttons. There are two text boxes and an image on the page. (Figure 99) The operations available on our website currently include zooming, highlighting, blurring, and deleting the recipient element. The user can attach one or more operations to each of these .buttons



Shape 104 *Implementation page designed by the user -*

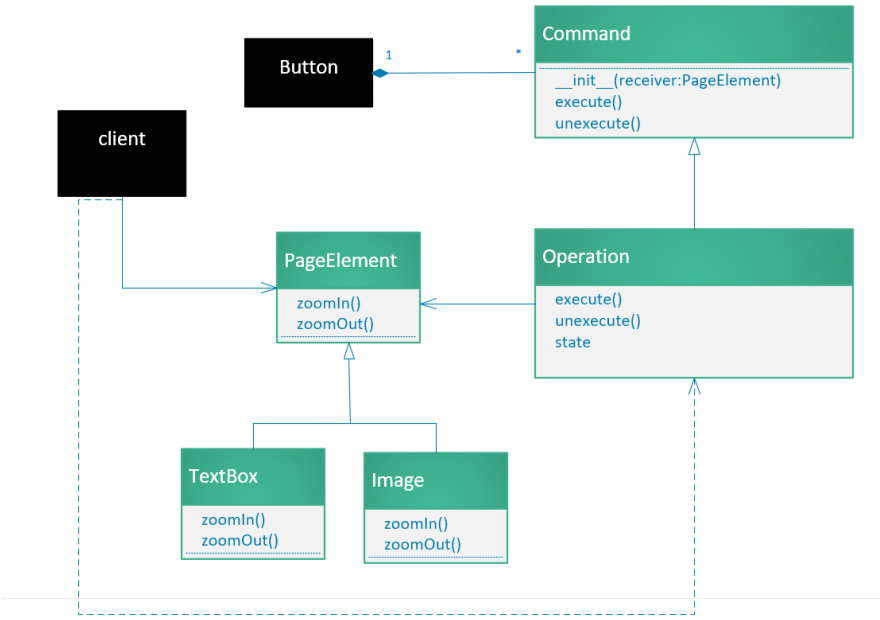
We need to design our website in a way that the user can perform any action on any button and on any element on the page. This means that the action invoker, buttons, have no information about the action. But all of these operations are inherently actions. So there should be an abstract classCommand Also, these actions have no information about the . elements on which they should be executed, the elements that receive .the action

We also want the user to be able to toggle these buttons. This means that if clicked again, the previous action will be reversed. For example, if button 1 performs the action of blurring and the user has clicked on it .once, clicking on it again should perform the action of transparency

Based on the system requirements and the above explanations, our desired pattern is theCommand pattern, where each of the four mentioned actions is aCommand The page elements are also . receiver and the buttons areinvoker Note that if we want to develop this program without using the . Command pattern, considering that the page elements are different, the implementation of the mentioned operations will also be different, and we must directly call the appropriate methods of each element object wherever we intend to use these operations. For example, if we add or remove a method in the elements class, we must change our code in all .buttons

The benefits of using the Command design pattern include decoupling the caller from the operation, objectifying the operation, and leveraging .the benefits of object orientation

:The structure of the described program is as follows



105*Figure105- Program class diagram*

:Implementing the program

```
class PageElement(ABC):
    @abstractmethod
    def zoomIn(self):
        pass

    @abstractmethod
    def zoomIn(self):
        pass


class Command(ABC):
    def __init__(self, receiver: PageElement):
        self.receiver = receiver

    @abstractmethod
    def execute(self):
        pass

    @abstractmethod
    def execute(self):
        pass
class TextBox(PageElement):

    def zoomIn(self):
        print("font size ++ ")

    def zoomOut(self):
        print("font size -- ")


class Image(PageElement):
    def zoomIn(self):
        print("Zoomed In ")

    def zoomOut(self):
        print("Zoomed Out ")
```

```
class Button():
    do = True

    def setAction(self, operation: Command):
        self.operation = operation

    def click(self):
        if(self.do):
            self.operation.execute()
        else:
            self.operation.unexecute()

        self.do = not(self.do)


class ZoomCommand(Command):
    def execute(self):
        self.receiver.zoomIn()

    def unexecute(self):
        self.receiver.zoomOut()

textBox = TextBox()
image = Image()
imageZoom = ZoomCommand(image)
textZoom = ZoomCommand(textBox)
btn1 = Button()
btn2 = Button()
btn1.setAction(imageZoom)
btn2.setAction(textZoom)
btn1.click()
btn1.click()
btn2.click()
btn2.click()
```

Figure 106- Implementing and running the program 106

```
Zoomed In
Zoomed Out
font size ++
font size --
```

Figure 107107- Output of the above code

Observer design pattern

Definition

Pattern that creates a one-to-many dependency between objects, so that when one object changes state, all its dependents are notified and .updated automatically

Objective

One of the side effects of dividing a system into related classes is the need to maintain compatibility between related objects, and this .compatibility should not lead to high coupling as it reduces reusability For example, in Excel software, data is displayed both in the form of charts and spreadsheets. Spreadsheets and charts do not know each other, but their behavior is contrary to this because when a user makes changes to a spreadsheet, the data changes and as a result, the chart also changes. In the Observer pattern, this data is called the subject and each observer, such as a spreadsheet or chart in the Excel example, is [585756].called an Observer

Use Cases

- When an abstraction has two aspects that are dependent on each other. Encapsulating these aspects in different objects allows us to .increase diversity and reusability
- ,When a change in one object means a change in other objects as well and we are unaware of the number of objects that will change as a .result
- When an object needs to notify other objects of changes it has made without knowing which objects they are. In other words, these objects .should have low coupling with each other

[56] Tightly coupled
[57] Reusability
[58] Spread sheet

Structure

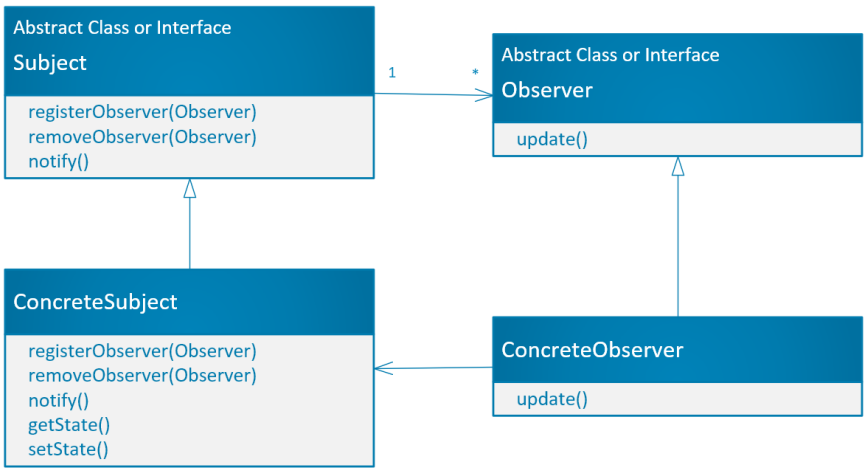


Figure 108108- class diagram in Observer Pattern

:Abstract class Subject

An abstract class that knows its observers. Observer objects can be any number. This class provides an interface for adding or removing Observer .objects

:Observer Pattern

.An interface that defines a way to update desired objects

:ConcreteSubject class

A concrete class that stores the relevant data and notifies all observers .when they change

:ConcreteObserver class

A class that holds a reference to a ConcreteObject and stores the .information obtained from this object, updating it using the update method

Example

Imagine there is a weather station that receives and stores meteorological data using its own sensors. So far, several weather software programs have been connected to this station and use its data. Each software

displays specific information. SoftwareCurrentWeather displays current temperature, air pressure, and humidity, SoftwareAverageWeather shows average temperature, pressure, and humidity, and SoftwareWeatherForecast provides a weather forecast for the next three days. It is also expected that due to the high accuracy of this weather station, the software WeatherHistory will also be connected to this station, which will display the temperature history in the region. When the data of the weather station changes, these software programs must be informed of these changes and their information updated. So in this example, the weather station is aSubject and the software programs are Observers .

We want to design a system for this weather station. First, we need to :pay attention to the key features of this station

1. Several software are connected to this station, but the station is .unaware of their number
2. In the future, some software may start or stop using the data from .this station
3. Any changes to the weather station data must be communicated to .all connected software

Incorrect Design

```
class WeatherStation(ABC):
    temperature:float;
    barometricPressure: float;
    humidity: float;

    def setData():
        #aquires the new data from the sensors
        pass;

    def notify():
        CurrentWeather.update(temperature,barometricPressure,humidity);
        AverageWeather.update(temperature,barometricPressure,humidity);
        WeatherForecast.update(temperature,barometricPressure,humidity);
```

Initial and incorrect system design -Shape 109109

This design and implementation have two flaws:

1. To add or remove an observer, we need to modify the code and .manually add each observer to our code
2. Sending data through method arguments in the update method can be problematic. Because this station may add new sensors and generate new data (e.g. UV levels). In this case, we need to modify our code in both notify and observers, which complicates code .maintenance

So we need to redesign the system. First, we create an abstract class or an interface named Subject that defines three methods registerObserver(), removeObserver(), and notify() which are responsible .for registering, removing, and notifying an Observer respectively

```python
class Subject(ABC):
    @abstractmethod
    def registerObserver(self, observer: Observer):
        return

    @abstractmethod
    def removeObserver(self, observer: Observer):
        return

    @abstractmethod
    def notifyObserver(self, observer: Observer):
        return
```

Figure 110 - Implementing the abstract class Subject 110

Then we define a concrete class named WeatherStation that inherits from the Subject class and implements its methods.Additionally, there are four methods setData(), getTemperature(), getPressure(), and getHumidity() which are responsible for providing data to the weather station (getting data from sensors), returning temperature, pressure, and .humidity respectively

```python
class WeatherStation(Subject):
    __temperature: float
    __barometricPressure: float
    __humidity: float

    def __init__(self):
        self.__observers = []

    def registerObserver(self, observer: Observer):
        self.__observers.append(observer)

    def removeObserver(self, observer: Observer):
        self.__observers.remove(observer)

    def notifyObserver(self):
        for obj in self.__observers:
            obj.update()

    def setData(self, temperature: float, barometricPressure: float, humidity: float):
        self.__temperature = temperature
        self.__barometricPressure = barometricPressure
        self.__humidity = humidity
        self.notifyObserver()

    def getTemperature(self):
        return self.__temperature

    def getPressure(self):
        return self.__barometricPressure

    def getHumidity(self):
        return self.__humidity
```

Implementation of the class – Shape 111111WeatherStation

.Now we need to implement abstract classes Observer and DisplayPage

```
class Observer(ABC):
    @abstractmethod
    def update():
        return


class displayPage(ABC):
    @abstractmethod
    def display():
        pass
```

*Shape 112112 – Implementation of classes*Observer *and* DisplayPage

The classDisplayPage is an interface that declares the behavior of display .and concrete classes must implement it

Now for each of the software, we create a class that inherits from classes .Observer and DisplayPage

```
class CurrentWeatherSoftware(Observer, displayPage):
    __temperature: float
    __barometricPressure: float
    __humidity: float
    __weatherStation: WeatherStation

    def __init__(self, weatherStationObj):
        self.__weatherStation = weatherStationObj
        self.__weatherStation.registerObserver(self)

    def update(self):
        self.__temperature = self.__weatherStation.getTemperature()
        self.__barometricPressure = self.__weatherStation.getPressure()
        self.__humidity = self.__weatherStation.getHumidity()

    def display(self):
        print("The current Weather Info: \nTemperature:{0} \nPressure:{1} \nHumidity:%{2}".format(
            self.__temperature, self.__barometricPressure, self.__humidity))
```

```
class AverageWeatherSoftware(Observer, displayPage):
    __temperature: float
    __barometricPressure: float
    __humidity: float
    __weatherStation: WeatherStation

    def __init__(self, weatherStationObj):
        self.__weatherStation = weatherStationObj
        self.__weatherStation.registerObserver(self)

    def update(self):
        self.__temperature = self.__weatherStation.getTemperature()
        self.__barometricPressure = self.__weatherStation.getPressure()
        self.__humidity = self.__weatherStation.getHumidity()

    def display(self):
        print("displays the average Temperature, Pressure and Humidity")
```

```
class WeatherForecast(Observer, displayPage):
    __temperature: float
    __barometricPressure: float
    __humidity: float
    __weatherStation: WeatherStation

    def __init__(self, weatherStationObj):
        self.__weatherStation = weatherStationObj
        self.__weatherStation.registerObserver(self)

    def update(self):
        self.__temperature = self.__weatherStation.getTemperature()
        self.__barometricPressure = self.__weatherStation.getPressure()
        self.__humidity = self.__weatherStation.getHumidity()

    def display(self):
        print("displays the weather forecast")
```

Shape 113Implementing software classes -

As shown in the above figure, the method update no longer takes another argument as input. Instead, software classes receive an instance of WeatherStation when instantiating an object, and during updates, they directly receive the data from that object by calling the relevant methods. And finally, we get the output from the designed system:

```
RashtWeatherStation = WeatherStation()
CurrentWeatherSoftware = CurrentWeatherSoftware(RashtWeatherStation)
RashtWeatherStation.setData(9, 1021, 70)
CurrentWeatherSoftware.display()
RashtWeatherStation.setData(7, 1018.5, 80)
CurrentWeatherSoftware.display()
```

Figure 114114 -Executing the final code

```
The current Weather Info:
Temperature:9
Pressure:1021
Humidity:%70
The current Weather Info:
Temperature:7
Pressure:1018.5
Humidity:%80
```

Output of the above code - Figure 115115

The structure of the new program will be as follows:

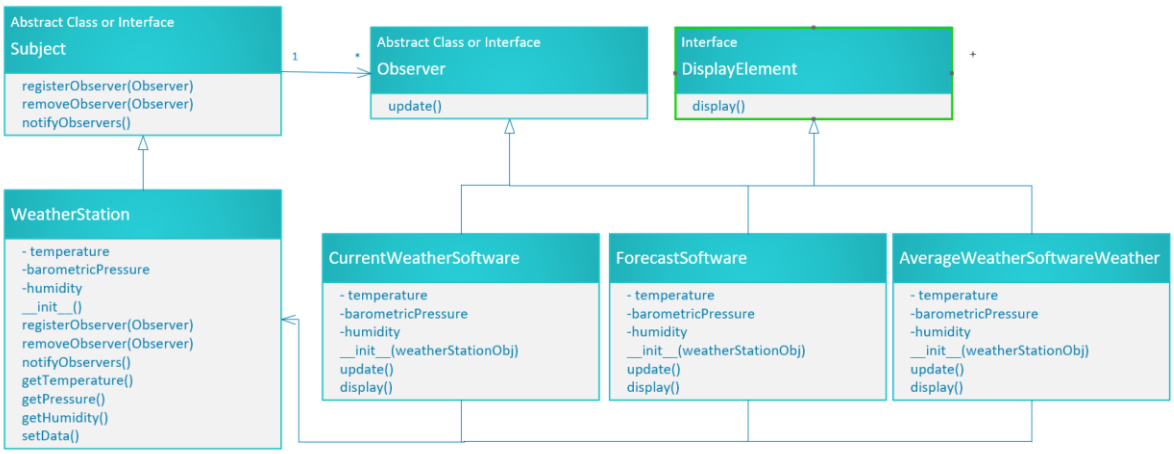


Figure 116116 - class diagram of the designed system

Iterator pattern

Definition

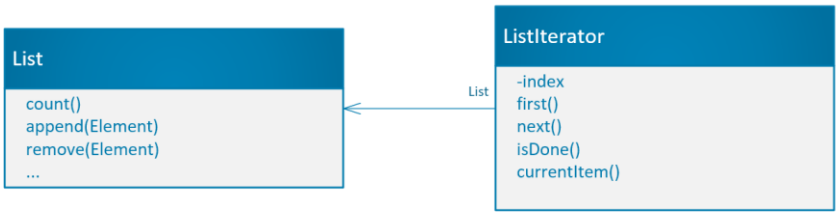
The Iterator pattern provides a way to access elements of a collection [59] .sequentially without exposing its underlying structure

Objective

A collection like a list provides a way to access its elements without revealing its internal structure to you. Additionally, depending on what you want to do, you may want to traverse the list in different ways. But even if you can anticipate your needs, you probably don't want to fill the List interface with operations for various traversals. You may also need to .have more than one traversal waiting on a list

The pattern of Iterator allows you to perform all the tasks. The main idea in this pattern is that responsibility for accessing and traversing the list is removed and placed in a repeater - Iterator. The Iterator class defines an interface for accessing list elements. A repeatable object is responsible for tracking the current element. Meaning it knows which .elements it has previously traversed

For example, a class List has a ListIterator, which calls the subsequent :relationship between them



Relationship between -Figure 117117List and ListIterator

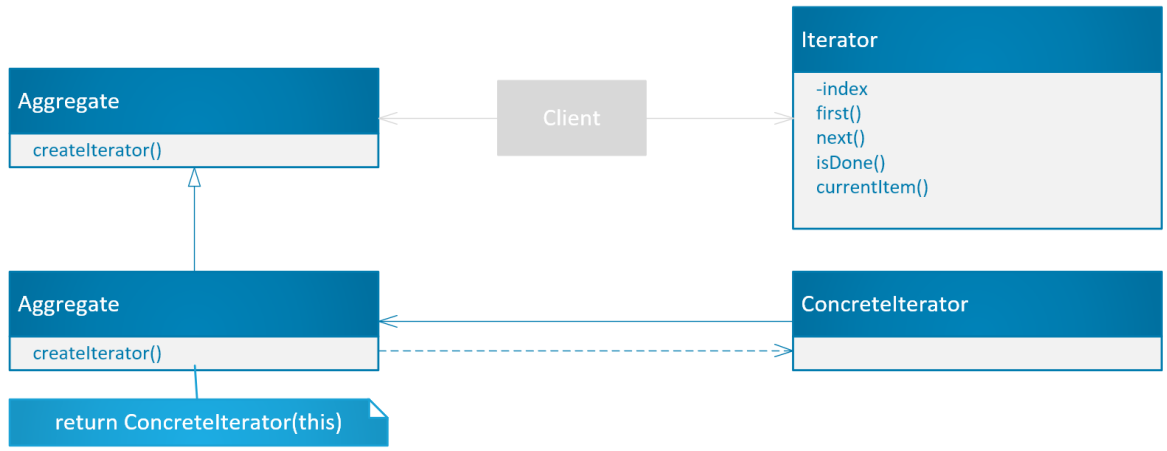
Before creating an object ofListIterator you must create a ,List .for traversal Then you can traverse the list elements in order with aListIterator .object TheCurrentItem method returns the current element and the First method returns the first element. TheisDone method also checks whether we have .reached the end of the list or not

[59] Aggregate object

Use Cases

- When you want to access the contents of a collection without revealing .it, you need to
- When you want your program to support multiple views of aggregated .objects
- When you need a uniform interface for traversing different aggregate [60].structures (i.e., for supporting multiple traversal strategies)

Structure



Shape 118118- class diagram of Iterator Pattern

Iterator

.It provides an interface for accessing and traversing elements

ConcreteIterator

Implements the Iterator interface and keeps track of the current position .in the aggregate object being traversed

Aggregate

.Provides an interface for creating an Iterator object

ConcreteAggregate

Implements the Aggregate interface and creates an object of the desired iterator.

[60] Polymorphic iteration

Example

We want the default traversal of a set of words to be such that one word .is returned at a time

In Python, there are two classes Iterator and Iterable which are used respectively for implementing an iterator and iterable. The iterator class must implement the method __next__ and the iterable class must implement the method __iter__.

:Our program will be as follows

```
class SkipTraverse(Iterator):
    _position = 0

    def __init__(self, collections):
        self.collections = collections

    def __next__(self):
        try:
            value = self.collections[self._position]
            self._position += 2
        except IndexError:
            raise StopIteration()

        return value


class WordsCollection(Iterable):
    def __init__(self, collection: List[str]):
        self._collection = collection

    def __iter__(self):
        return SkipTraverse(self._collection)


words = WordsCollection(["this", "doesn't", "make", "any", "sense"])
print("\n".join(words))
```

Figure 119 *Implementing and running the program -*

```
this
make
sense
```

Figure 120120- Output of the above code

PatternState

Definition

The State pattern allows an object to change its behavior when its internal .state changes

Objective

[61][62] .Consider a TCPConnection class representing a network connection :A TCP-Connection object can be in one of several different states established, listening, and closed. When a TCPConnection receives requests from other objects, it responds differently based on its current state. For example, the response to a close request depends on whether the connection is closed or established. The State pattern explains how .TCPConnection can exhibit different behaviors in each of its states

The main idea in this pattern is to define an abstract class named TCPState to represent network connection states. The TCPState class declares a common interface for all classes that represent different operational states, announcing them. Subclasses of TCPState execute specific behavior for a state. For example, the classes TCPEstablished and TCPClosed implement specific behavior in the Established and .Closed states of TCPConnection

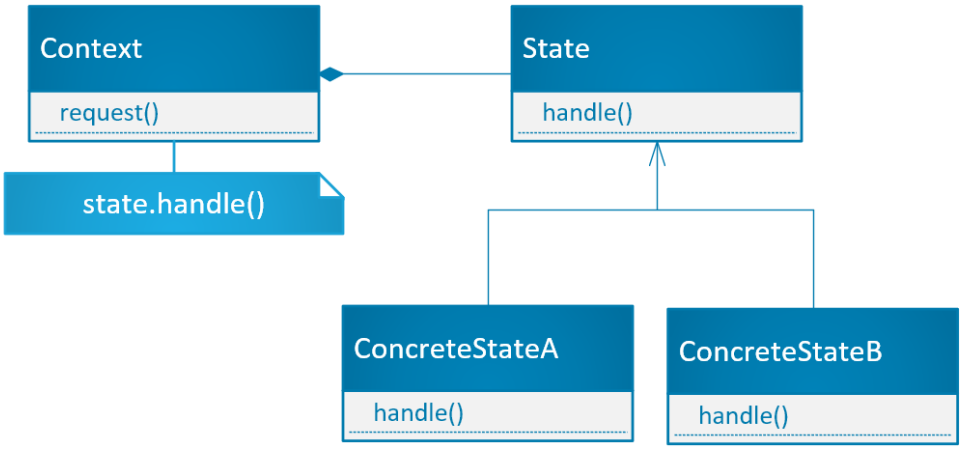
Use Cases

When the behavior of an object is dependent on its state and it changes• .its behavior at runtime based on that state

When an operation consists of large conditional expressions and• [63] multiple parts that depend on the state. This state is usually represented by one or more enumeration constants. Often, multiple operations have the same conditional structure. The State pattern separates each branch of the condition into a separate class. This allows you to consider the state as a single entity that can be independent and different from other .objects

Structure

[61] Established
[62] Open request
[63] Enumerate constant



Shape 121121- class diagram of State Pattern

Context

Define the favorite customer interface.It also holds an instance of a .ConcreteState subclass that defines the behavior of the active state

State

Defines an interface for encapsulating the behavior associated with a .particular state of the Context

ConcreteState

Each subclass implements behavior associated with a state of the .Context

Example1

Suppose we are in the development team of WhatsApp and we want to upload photos and videos based on the connection status. We have three statesoffline, wifi and data In the future, we may also add .lowData and lowWifi states. Uploading a video and uploading a photo are methods that need to be implemented. In each of the mentioned states, the implementation .of these methods is different

:The implementation and execution of this program is as follows

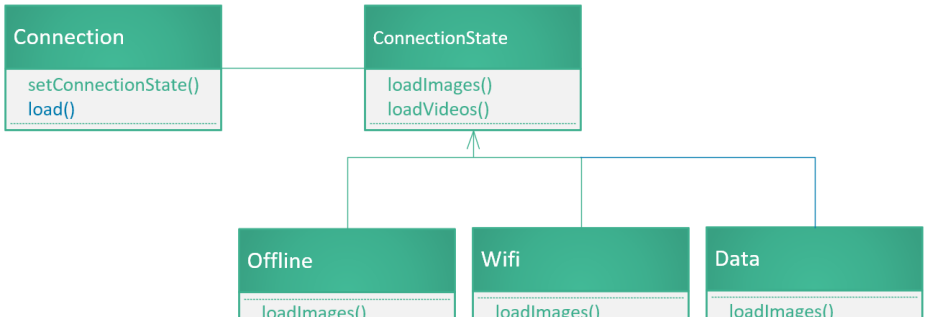
```python
class ConnectionState(ABC):
    @abstractmethod
    def loadImages(self):
        pass

    @abstractmethod
    def loadVideos(self):
        pass


class Offline(ConnectionState):
    def loadImages(self):
        print("don't even try")

    def loadVideos(self):
        print("don't even try")


class Wifi(ConnectionState):
    def loadImages(self):
        print("download all images ")

    def loadVideos(self):
        print("download all videos ")

class Data(ConnectionState):
    def loadImages(self):
        print("download half of images ")

    def loadVideos(self):
        print("download no videos ")


class Connection:
    _connectionState = Offline()

    def setConnectionState(self, connectionState: ConnectionState):
        self._connectionState = connectionState

    def load(self):
        self._connectionState.loadImages()
        self._connectionState.loadVideos()


connection = Connection()
connection.load()
data = Data()

connection.setConnectionState(data)
connection.load()
```

Figure 122 *Implementation and execution of the program -*

```
don't even try
don't even try
download half of images
download no videos
```

Figure 123 *Output of the above code -*

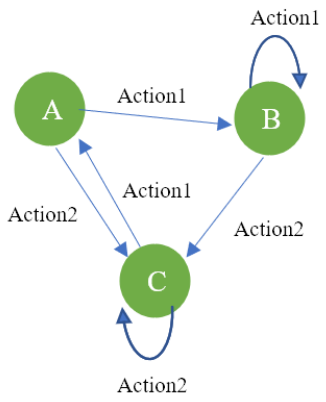
:The program structure is as follows



of the program - class diagram *Figure 124* 124

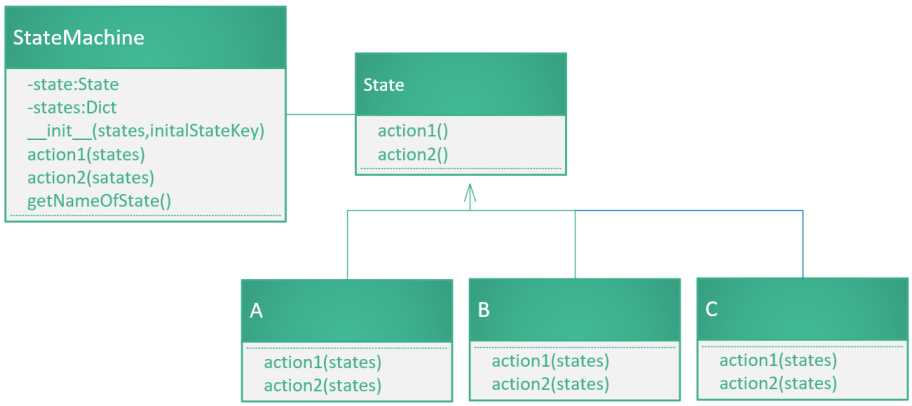
Example 2

This time we want to implement an abstract but slightly more difficult [64]:example. Assume we have a state machine as follows



Sample State Machine - Figure 125125

There are two actions for each state. So in the abstract class State we define these two actions which classes A, B and C implement. Executing each of these actions changes the state of the device. So each time a state object must be returned. To prevent the creation of additional and duplicate instances of states, we must first store all state objects somewhere. In the StateMachine class, we design the constructor function in such a way that it first receives a dictionary of state objects and then ,the key of the initial state. And each time a method of that state is called for example action1 a dictionary containing objects is also sent to that , method. The method is executed and the state object is returned from the same dictionary stored in the state variable in the StateMachine.class



Shape 126 -class diagram designed program

:Implement the program as follows

[64] State machine

```python
class State(ABC):

    @abstractmethod
    def action1(self, states):
        pass

    @abstractmethod
    def action2(self, states):
        pass


class A(State):
    def action1(self, states):
        return states["B"]

    def action2(self, states):
        return states["C"]


class B(State):
    def action1(self, states):
        return states["B"]

    def action2(self, states):
        return states["C"]

class C(State):
    def action1(self, states):
        return states["A"]

    def action2(self, states):
        return states["C"]


class StateMachine():
    _states: Dict[State]
    _state: State = None

    def __init__(self, states: Dict[State], initalStateKey: str):
        self._states = states
        self._state = states[initalStateKey]

    def action1(self):
        print(self.getNameofState()+"----action1---->", end="")
        self._state = self._state.action1(self._states)
        print(self.getNameofState())

    def action2(self):
        print(self.getNameofState()+"----action1---->", end="")
        self._state = self._state.action2(self._states)
        print(self.getNameofState())

    def getNameofState(self):
        name = str(type(self._state))
        return (name[name.find(".")+1:name.find(".")+2])
```

Figure 127127 – Program implementation

```python
myStateMachine = StateMachine({"A": A(), "B": B(), "C": C()}, "A")

myStateMachine.action1()
myStateMachine.action1()
myStateMachine.action2()
myStateMachine.action2()
myStateMachine.action1()
myStateMachine.action2()
```

Figure 128 *Testing and running the program -*

```
A----action1---->B
B----action1---->B
B----action2---->C
C----action2---->C
C----action1---->A
A----action2---->C
```

Figure 129129- Output of the above code

As can be seen, performing actionsaction11 and action2 in each case, leads .us to our expected state. So the program works correctly

The advantage of the State pattern over conditional statements is easier maintenance and extensibility. Assume if we wanted to implement the above example using only conditional statements, we would need 6 conditional statements that would increase with more cases, making .maintenance and extension very difficult

The Strategy pattern

Definition

The Strategy pattern defines a family of algorithms, encapsulates each one and allows them to be interchangeable. This pattern allows algorithms to be independent of clients.

Objective

There are many algorithms for converting a stream of text into different lines. Implementing them in classes that need them is not a suitable option for the following reasons:

- Customers who need line breaking should bear more complexity in developing code for line breaking. This makes maintaining customers, especially when supporting multiple line-breaking algorithms, more difficult[65].

- The appropriate algorithm varies at different times. We also do not want to support linear breaking algorithms that we do not need.

- Adding new algorithms and modifying existing algorithms at the time when linear breaking is implemented in the client is very difficult.

- We can avoid these problems by defining classes that encapsulate various linear breaking algorithms. An algorithm that is encapsulated in this way has a strategy.

Use Cases

- When classes differ only in their behavior. Strategies provide a way to configure a class with one of multiple behaviors.

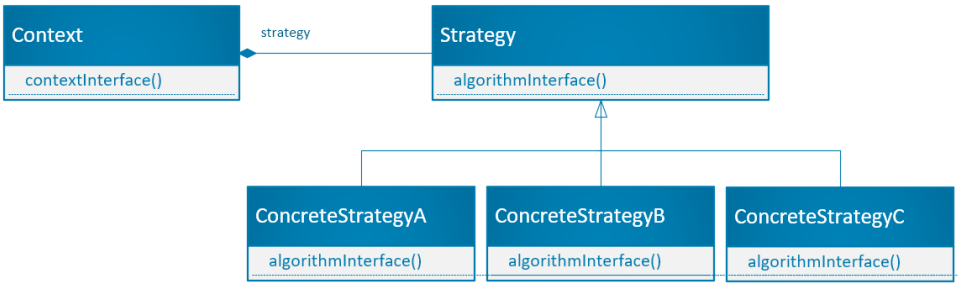
- When you need different variations of an algorithm. For example, you may define algorithms that reflect different space/time trade-offs. Timely, strategies can be used to encapsulate a family of algorithm classes.

[65] Line breaking

- When an algorithm uses data that customers should not know about. Use the Strategy pattern to encapsulate complex data .structures and specific algorithms

- When a class defines too many behaviors, and they appear as multiple conditional statements in its operations. Instead of many conditionals, move related conditional branches to your Strategy .class

Structure



Shape 130 *-class diagram of Strategy pattern*

Strategy

Declares a common interface for all supported algorithms. Contextuses .this interface to call an algorithm defined by a ConcreteStrategy

ConcreteStrategy

Implements the algorithm using theStrategy.interface

Context

Configured with a one ConcreteStrategy. Also has a reference to a Strategyobject. This class may provide an interface for Strategyto access .its data

Example

We want to write a program for an equalizer. This equalizer supports two styles, classic and rock. In this program, the user can choose the music style being played and press the play button. They can also change the music style at any time. Implementing this program using the methods of music styles is possible, but for adding a new style or editing the settings of a style, we are forced to change the code in the program class. Given the explanations provided, we need to implement the Strategy pattern, so :our program will look like this

```
class EqualizerApp():
    equalizerGenre: EqualizerGenre = None

    def __init__(self, defaultGenre: EqualizerGenre):
        self.equalizerGenre = defaultGenre

    def play(self):
        self.equalizerGenre.adjust()
        name = str(type(self.equalizerGenre))
        name = name[name.find(".")+1:name.rfind("'")]
        print(f"the music is playing. Equalizer On:{name}\n")

    def changeGenre(self, genre: EqualizerGenre):
        self.equalizerGenre = genre
        self.play()

class EqualizerGenre(ABC):
    @abstractmethod
    def adjust(self):
        pass


class Classical(EqualizerGenre):
    def adjust(self):
        print("Tuned for classical genre")


class Rock(EqualizerGenre):
    def adjust(self):
        print("Tuned for Rock genre")


class Pop(EqualizerGenre):
    def adjust(self):
        print("Tuned for Pop genre")
```

131 *Shape131- Implementation of the program*

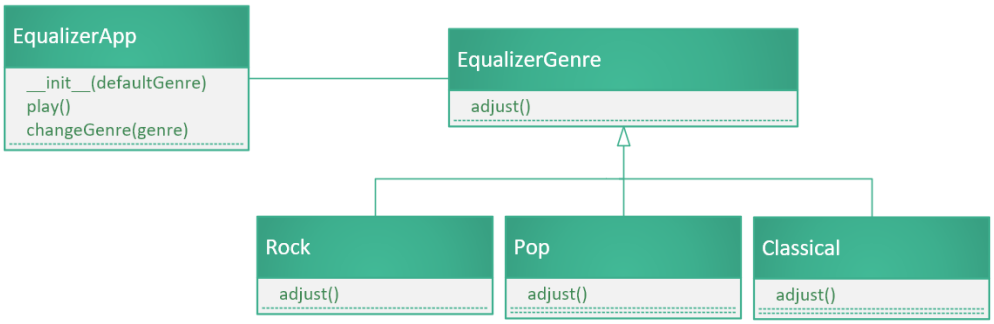
```
rock = Rock()
classical = Classical()
pop = Pop()
app = EqualizerApp(pop)
app.play()
app.changeGenre(rock)
```

Shape 132 *Testing and running the program -*

```
Tuned for Pop genre
the music is playing. Equalizer On:Pop

Tuned for Rock genre
the music is playing. Equalizer On:Rock
```

Shape 133 *Output of the above code -*

:The program structure is as follows



134*Shape134- Class diagram of the program*

Template Method Pattern

Definition

The Template Method pattern defines the skeleton of an algorithm in an operation, deferring some steps to subclasses. This pattern allows subclasses to redefine certain steps of an algorithm without changing the .overall structure of the algorithm

Objective

The Template Method pattern is used to define a structural framework that we are completely sure we will never change. The common use of this pattern is in frameworks. Where the structure of a framework is fixed and will not change. If we want to give a real-world example for this pattern, we can liken it to opening a bank account form. The form has its own default and fixed texts and places to enter user information. When the user enters their information, they submit the form, which includes printed texts and their information, to the clerk. The clerk then enters their information and signature into the form. Thus, the process of opening an .account is completed

Use Cases

- When we want to implement the immutable parts of an algorithm at one time and delegate the implementation of the variable parts to .subclasses

- When common behaviors among subclasses should be placed in a common local class to prevent duplicate code. First identify the differences in the existing code and then separate the differences into new operations. Finally, replace the different code with a .template method that calls one of these new operations

- When you need to control the extension of subclasses. You can define a template method that calls the hook operations at specific .times. So extending subclasses is only possible at these times

Structure

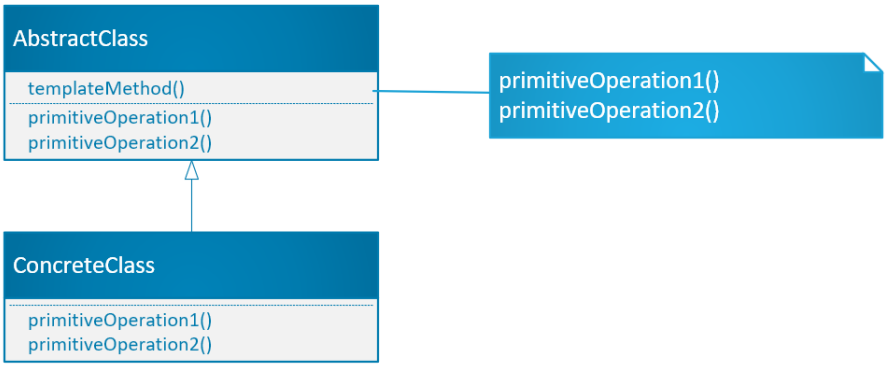


Figure 135135- class diagram of Template Method pattern

AbstractClass

Defines the first abstraction operation that concrete subclasses implement to define the steps of an algorithm. It also defines the template method. The template method calls the first operation and the operations .defined in AbstractClass and operations that are in other objects

ConcreteClass

It performs preliminary operations to implement various stages of the .algorithm

Example

Imagine we are developing a framework for creating a survey form. Each survey consists of at least one text field and a submit button. The customer can also add other fields to this form. So the text field and .submit button are always constant, and customer fields are variable Given the circumstances, our choice is the Template Method pattern. So first, we create an abstract class SurveyForm that implements three methods: create, reviewBox, and submitButton, and defines :anotherTextBox method in it

```
class SurveyForm(ABC):

    def create(self):
        self.anotherTextBox()
        self.reviewBox()
        self.submitButton()

    def reviewBox(self):
        print("a review box□")

    def submitButton(self):
        print("a submit button□")

    @abstractmethod
    def anotherTextBox(self):
        pass
```

Then we create the class ... which also has a name field

SurveyForm Implementation of the class-Figure 136136

```
class MySurvey(SurveyForm):
    def anotherTextBox(self):
        print("name box 🆎")
```

Figure 137 *Implementation of the class -MySurvey*

:Run and test the program

```
mySurvey = MySurvey()
mySurvey.create()
```

Figure 138 *Testing and running the program -*

```
name box 🆎
a review box⬜
a submit button🅾
```

Figure 139 *Output of the above code -*

:The program structure is as follows

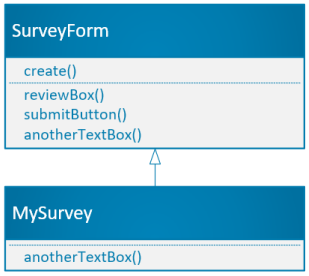


Figure 140 - *class diagram of the program*

Chapter Five: Resources

1.Gamma, Eric: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional; 1st edition (November 10, 1994)

2.Freeman, Eric et al: Headfirst Design Patterns. O'Reilly Media; 1st edition (October 1, 2004)

3. Alexander Shvets, Design Patterns, https://refactoring.guru/design-patterns

4.Alexander Shvets, Design Patterns in Python
https://refactoring.guru/design-patterns/python

5. Christopher Okhravi, Design Patterns Series, https://www.youtube.com/c/ChristopherOkhravi