# Advanced Software Engineering


**Student Names**

Mennat-Allah Yousri Rabeh
Sohila Ayman Lashien
Mennat-Allah Medhat Mostafa


17/4/2024

—

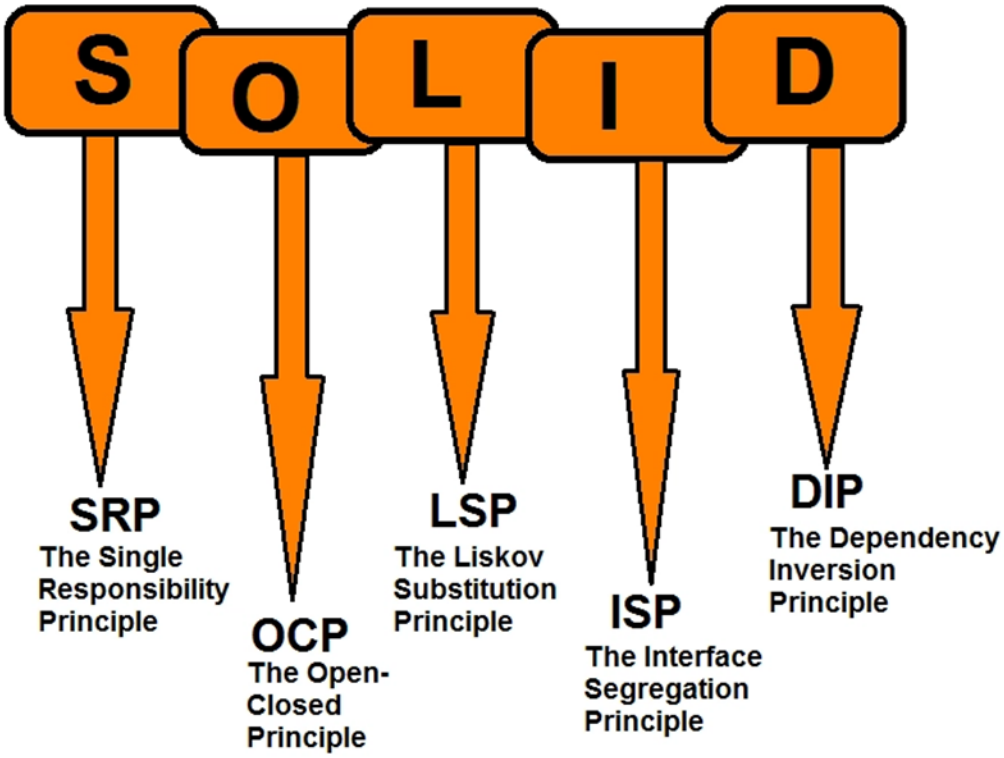

Dept. Machine Intelligence


—

Dr. Mahmoud Sakr

# Table of Contents

# SOLID Principles




- **SOLID is a mnemonic acronym** of 5 acronyms themselves: **SRP** - Single Responsibility Principle, **OCP** - Open-Closed Principle, **LSP** - Liskov Substitution Principle, ISP - Interface Segregation Principle, **DIP** - Dependency Inversion Principle.
- Best practice to solve common software problems.
- Solution in the form of templets that may be applied to real-world problems.
- Our job as a developer is to create code and programs that are easy to read and thereby easy to maintain and extend.

## 1. Single Responsibility Principle (SRP)

- It is the first of the SOLID principles and encourages to give classes only **a single** and definite **reason to change,** meaning that the class should have only one job. If you can think of more than a single reason for the class to change, the class has more than one responsibility.
- SRP is often confused with the "Do one thing and do it will" rule.

Imagine a rectangle class that has the two public methods draw and area. The draw method should return coordinates or any graphical representation. The area method returns the area of the current rectangle instance. The rectangle class thereby has two responsibilities. It's responsible for calculating the area and responsible to draw itself.

```
class Rectangle(object):

  def __init__(self, height: float, width: float):
    self.height = height
    self.width = width

  def draw(self) -> VisualRepresentation:
    return visual_representation(self)

  def area()-> float:
    return self.height*self.width
```

This separation leads to a single responsibility for each geometric rectangle and the rectangle class. Changes in the draw method now can no longer affect the way they are calculated.

```
class GeometricRectangle(object):

  def __init__(self, height: float, width: float):
    self.height = height
    self.width = width

  def area()-> float:
    return height*width


class Rectangle(GeometricRectangle):

  def draw(self) -> VisualRepresentation:
    return visual_representation(self)
```

## 2. Open-Closed Principle (OCP)

- The open-closed principle states that the software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification. Meaning, once you created your class, it **shouldn't change** any more. However, it could change by simply creating a child class that thereby **extends its behavior**.
- Any new functionality should be done by new classes instead of changing the existing one.
- How to implement OCP:
  - ➢ One may achieve this by adding new functionality to the derived class.
  - ➢ Or allow client to access the original class with abstract interface.

The key message of the **open-closed principle**, in this case, is that your classes should be open for extension but closed for modification. Meaning, once you created your class, it **shouldn't change** any more. However, it could change by simply creating a child class that thereby **extends its behavior**. Imagine you had a user class that holds a name and an age.

```
class User:

  def __init__(self, username: str, age: int):
    self.username = username
    self.age = age

  def __repr__(self):
    return f"User: {self.username}, {self.age} years old"
```

Now imagine you want to extend this class by an attribute that saves the user's favorite game. A naive solution to this problem would be to simply add an attribute "favorite_game" to the user class:

```
class User:

  def __init__(self, username: str, age: int, favorite_game: Game):
      self.username = username
      self.age = age
      self.favorite_game = favorite_game

  def __repr__(self):
      return f"User: {self.username}, {self.age} years old, favorite game: {self.favorite_game}"
```

This might work if your system is small or in development. But if you want to change this in a productive system, things are going to break. Not only did the signature change because the constructor now expects a favorite_game, but also the __repr__ method changed and might break things further. This violates Meyer's postulate. A possible solution to this again could be inheritance:

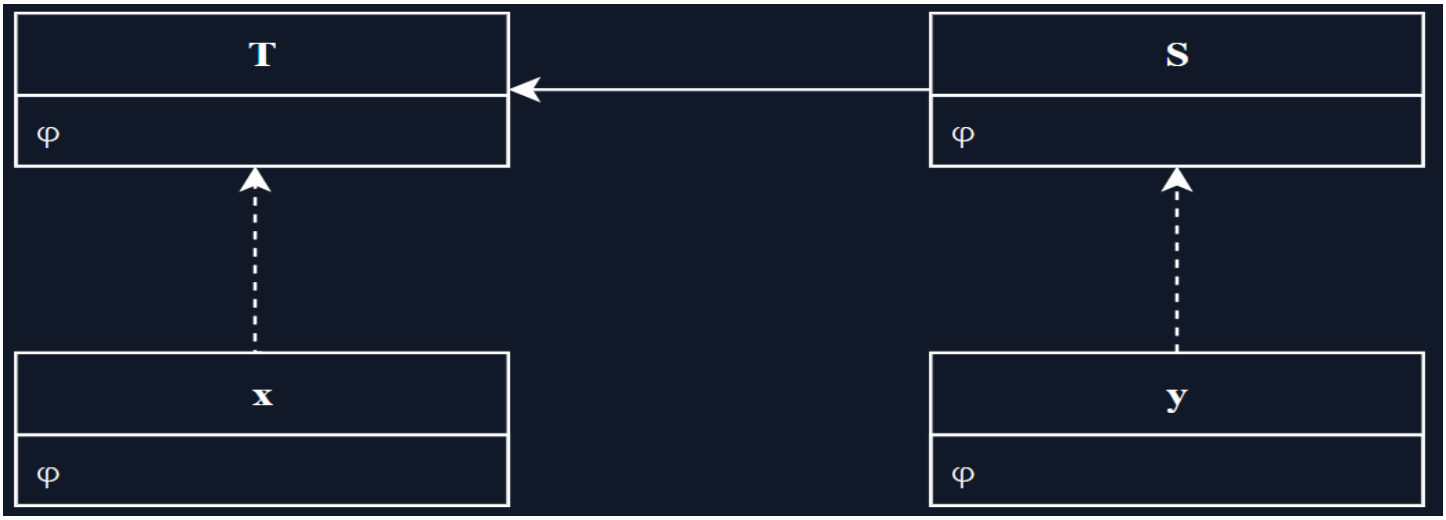
```python
class User:

  def __init__(self, username: str, age: int):
      self.username = username
      self.age = age

  def __repr__(self):
      return f"User: {self.username}, {self.age} years old"


class Gamer(User):

  def __init__(self, username: str, age: int, favorite_game: Game):
      super().__init__(username, age)
      self.favorite_game = favorite_game

  def __repr__(self):
    return f"User: {self.username}, {self.age} years old, favorite game: {self.favorite_game}"
```

By using this, your **functionality is extended**, but you don't apply any changes to the original class.

## 3. Liskov Substitution Principle (LSP)

- If you have class S inherits from class T, then class T should be replaced by class S without any changes.
- Subtype Requirement: Let ϕ(x) be a property provable about objects x of type T. Then ϕ(y) should be true for objects y of type S where S is a subtype of T.

Visualizing Liskov's definition leads to the following class diagram:




We have a type T and a subtype S, and objects of x that are of type T and objects y of type S. Also, all the elements possess an attribute ϕ. A representation in Python code could look like this:

```python
class T:

  def __init__(self, phi: list):
    self.phi = phi

class S(T):
  pass

if __name__ == "__main__":
  x = T(phi=["a", "b"])
  y = S(phi=["c", "d"])
```

And this indeed fulfills the Liskov substitution principle. Any object of type S could replace its parent class. How could you possibly violate this?

```python
class T:

  def __init__(self, phi: list):
    self.phi = phi

class S(T):

  def __init__(self, phi: str):
    self.phi = phi

if __name__ == "__main__":
  x = T(phi=["a", "b"])
  y = S(phi="c, d")
```

If you look closely, you will see that the subtype S implements its own class attribute phi of type string instead of list. This violates Liskov's theory. You can no longer replace T with objects of type S. One becomes aware of this concept's meaningfulness when you think about implementing a method "print_phis" that should solely print all elements in the class attribute phi. Either instances of class T or S would run into a runtime error or lead to a higher degree of complexity in the "print_phis" method due to additional conditionals.

## 4. Interface Segregation Principle (ISP)

- Clients should not be forced to depend on methods they don't use.
- Avoid fat interface.
- Client must not implement unnecessary methods.

Thereby it naturally supports loose coupling and maintainability. The main message behind ISP is that large Interfaces should be split into multiple ones. Functions and classes should not depend on methods they don't use. Look at the following example for a "fat" interface:

```
class GeometricInterface(ABC):

  @abstractmethod
  def get_area() -> float:
    raise NotImplementedError

  @abstractmethod
  def get_diameter() -> float:
    raise NotImplementedError

class Square(GeometricInterface):

  def __init__(self, height: float, width: float):
    self.height = height
    self.width = width

  def get_area():
    return self.height*self.width

  def get_diameter() -> float:
    raise NotImplementedError

class Circle(GeometricInterface):

  def __init__(self, radius: float):
    self.radius = radius

  def get_area():
    return self.radius * PI **2

  def get_diameter() -> float:
    return self.radius*2
```

In the previous example as you can see, the Geometric Interface has two abstract methods of which get_diameter() is not used by its subtype square. Thereby the interface segregation rule is violated. There are many possible solutions to this. We could, for instance, segregate the Geometric Interface into three Interfaces:

```
class GeometricInterface(ABC):

  @abstractmethod
  def get_area() -> float:
    raise NotImplementedError

class ElipseInterface(GeometricInterface, ABC):

  @abstractmethod
  def get_diameter() -> float:
    raise NotImplementedError

class RectangleInterface(GeometricInterface, ABC):
  pass


class Square(Rectangle):

    def __init__(self, height: float, width: float):
        self.height = height
        self.width = width

    def get_area(self):
        return self.height * self.width

class Circle(Elipse):

    def __init__(self, radius: float):
        self.radius = radius

    def get_area(self):
        return self.radius * PI ** 2

    def get_diameter(self) -> float:
        return self.radius * 2
```

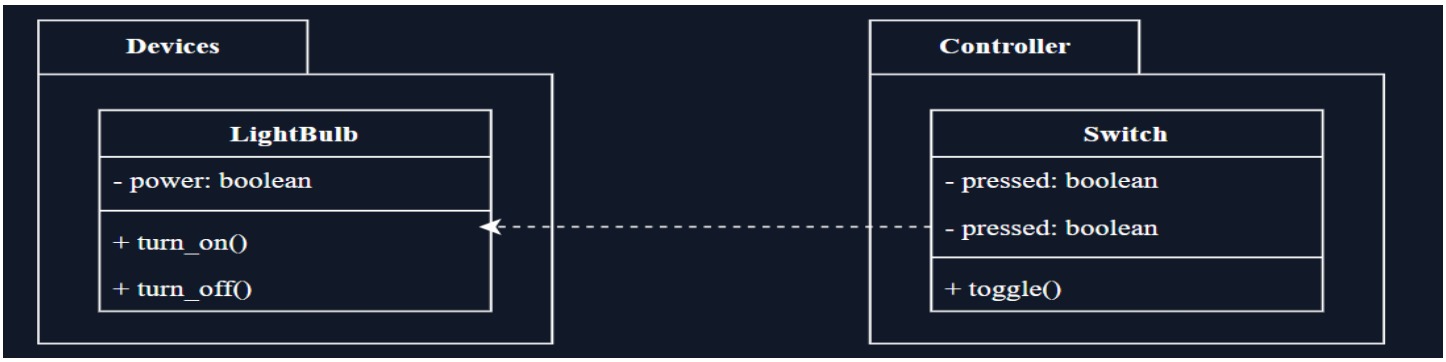
## 5. Dependency Inversion Principle (DIP)

A. High level modules should not depend upon low level Modules. Both should depend upon abstractions.
B. Abstractions should not depend upon details. Details should depend upon abstractions.

the DIP states that modules should not rely on modules that belong to a subordinate concept and should not rely on generalizations. A quick piece of code that is often referenced:

```python
class LightBulb:

  def __init__(self, initial_state: bool=False):
    self.power = initial_state

  def turn_on(self):
    self.power = True

  def turn_off(self):
    self.power = False

class Switch:

  def __init__(self, light_bulb: LightBulb, pressed: bool=False):
    self.light_bulb = light_bulb
    self.pressed = pressed

  def toggle(self):
    self.pressed = not self.pressed # Toggle
    if self.pressed:
      self.light_bulb.turn_on()
    else:
      self.light_bulb.turn_off()
```

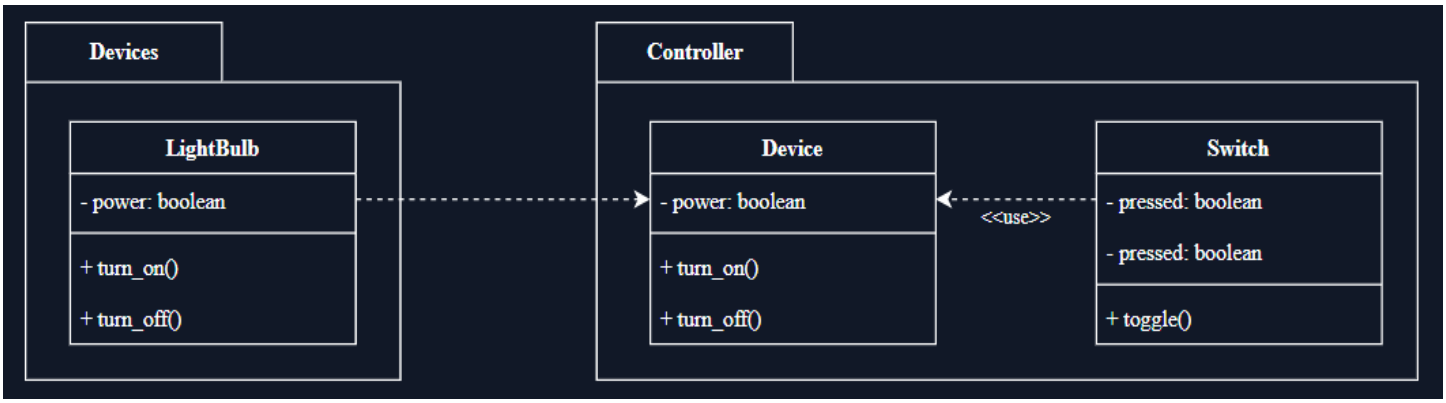
The **DIP violation** here is that a switch is a concept that is logically in a layer above the light bulb, and the switch relies on it. This will lead to poor extensibility or even circular imports that prevent the program from being interpreted or compiled.



Instead of the light bulb telling the switch how the bulb should be handled, the switch should tell the light bulb how to implement it. The naive approach would be to define an interface that tells the light bulb how it should behave to be used with a switch.

```
class Device(ABC):
  power: boolean

  def __init__(self, initial_state: bool=False):
    self.power = initial_state

  def turn_on(self):
    raise NotImplementedError

  def turn_off(self):
    raise NotImplementedError

class Switch:

  def __init__(self, device: Device, pressed: bool=False):
    self.device = device
    self.pressed = pressed

  def toggle(self):
    self.pressed = not self.pressed # Toggle
    if self.pressed:
      self.device.turn_on()
    else:
      self.device.turn_off()

class LightBulb(Device):

  def turn_on(self):
    self.power = True

  def turn_off(self):
    self.power = False
```

Visualized as a class diagram, this source code would lead to the object-oriented design:




The dependency has been inverted. Instead of the switch relying on the light bulb, the light bulb now relies on an interface in a higher module. Also, both rely on abstractions, as required by the DIP. Last but not least, we also fulfilled the requirement "Abstractions should not depend upon details. Details should depend upon abstractions" - The details of how the device behaves rely on the abstraction (Device interface).

# Behavioral Design Patterns

Behavioral design patterns are concerned with algorithms and the assignment of responsibilities between objects.

## 1. Strategy

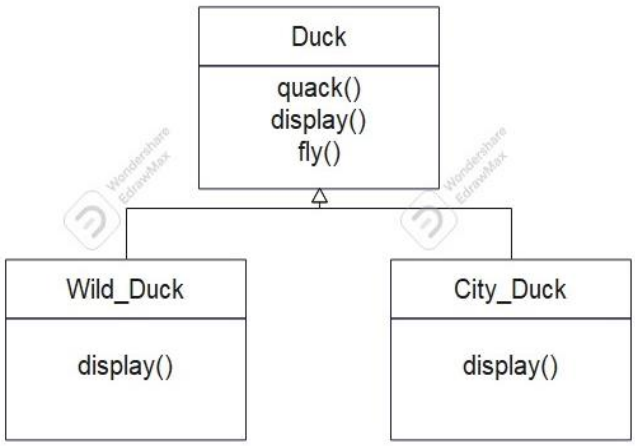
- **Intent**

  **Strategy** is a behavioral design pattern that lets you define a family of algorithms, put each of them into a separate class, and make their objects interchangeable.

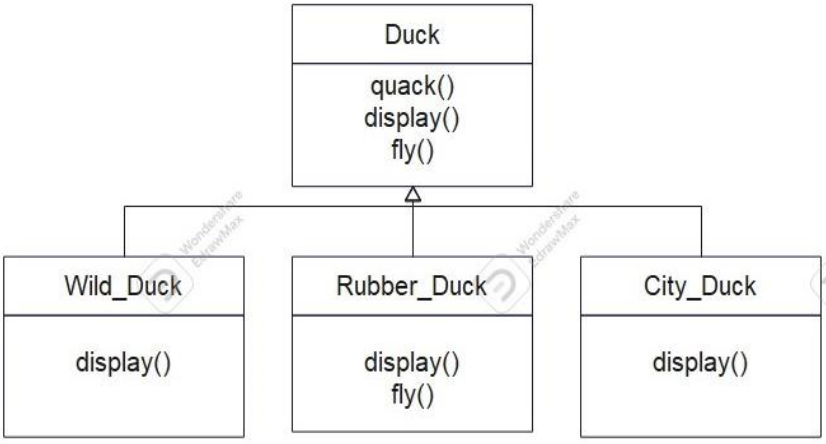
- **Problem**

  It started with a simple SimUDuck app. Joe works for a company that makes a highly successful duck pond simulation game, SimUDuck. The game can show a large variety of duck species swimming and making quacking sounds. The initial designers of the system used standard OO techniques and created one Duck superclass from which all other duck types are inherited.

## The Original Diagram

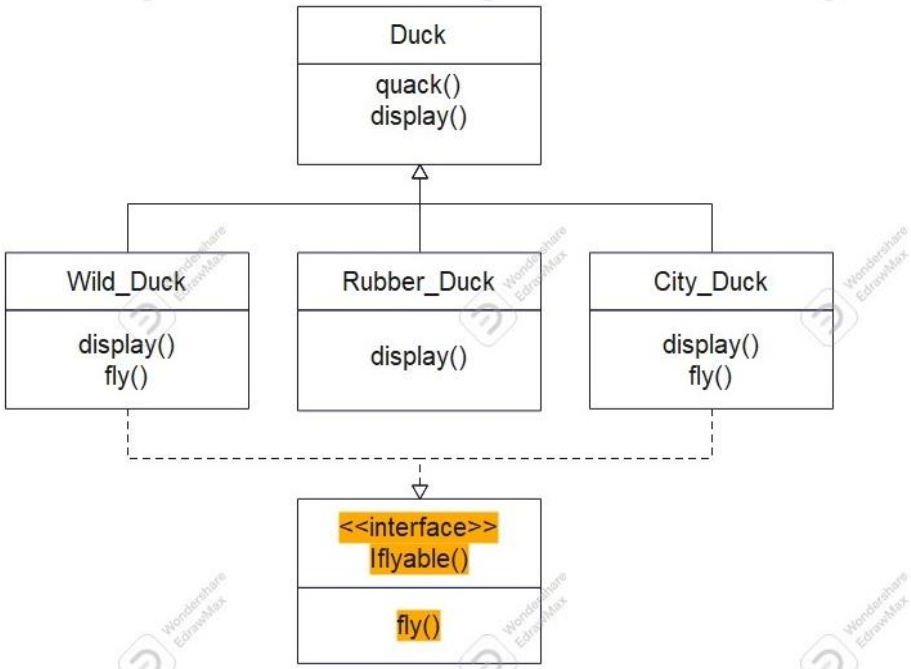



## First Suggested Solution: Inheretance with overriding the required methods

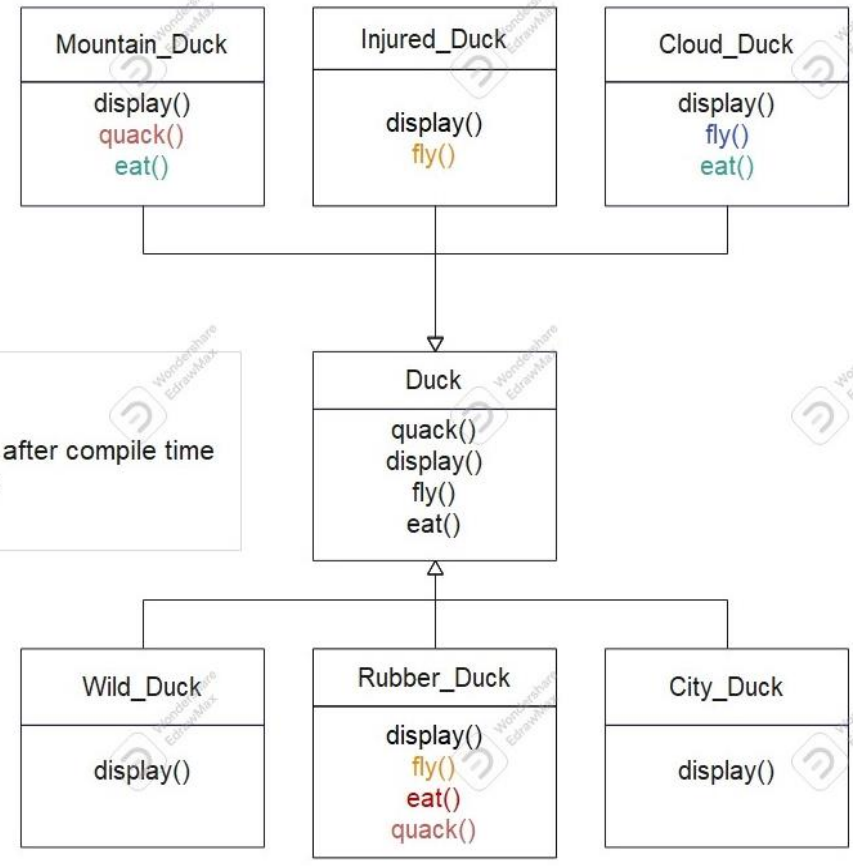



Remember:
DRY: “Don’t Repeat Yourself” principle advocates for avoiding code duplication by creating reusable components.
We need our code to be high cohesion correlates with loose coupling.

## Second Suggested Solution: Inheretance with implementing an interface when required




## Why Inherentance is not the best solution?




Disadvantages:
(1) No modification after compile time
(2) Class explosion

```python
class Duck:
    def quack(self):
        print("Quack!")

    def fly(self):
        print("Flying...")

    def eat(self):
        print("Eating...")

    def display(self):
        print("Displaying duck")


class WildDuck(Duck):
    def display(self):
        print("Displaying wild duck")


class CityDuck(Duck):
    def display(self):
        print("Displaying city duck")


class RubberDuck(Duck):
    def fly(self):
        print("Can't fly!")

    def quack(self):
        print("Squeak!")

    def display(self):
        print("Displaying rubber duck")

    def eat(self):
        print("Rubber ducks can't eat!")
```

```python
class InjuredDuck(Duck):
    def fly(self):
        print("Can't fly!")

    def display(self):
        print("Displaying injured duck")


class MountainDuck(Duck):
    def eat(self):
        print("Fishes!")

    def display(self):
        print("Displaying mountain duck")


class CloudDuck(Duck):
    def eat(self):
        print("Fishes!")

    def display(self):
        print("Displaying cloud duck")
```

```
wild_duck = WildDuck()
city_duck = CityDuck()
rubber_duck = RubberDuck()
injured_duck = InjuredDuck()
mountain_duck = MountainDuck()
cloud_duck = CloudDuck()

wild_duck.quack()
wild_duck.fly()
wild_duck.eat()
wild_duck.display()

city_duck.quack()
city_duck.fly()
city_duck.eat()
city_duck.display()

rubber_duck.quack()
rubber_duck.fly()
rubber_duck.display()

injured_duck.quack()
injured_duck.fly()
injured_duck.eat()
injured_duck.display()

mountain_duck.quack()
mountain_duck.fly()
mountain_duck.eat()
mountain_duck.display()

cloud_duck.quack()
cloud_duck.fly()
cloud_duck.eat()
cloud_duck.display()
```

The output:

```
Quack!
Flying...
Eating...
Displaying wild duck
Quack!
Flying...
Eating...
Displaying city duck
Squeak!
Can't fly!
Displaying rubber duck
Quack!
Can't fly!
Eating...
Displaying injured duck
Quack!
Flying...
Fishes!
Displaying mountain duck
Quack!
Flying...
Fishes!
Displaying cloud duck
```

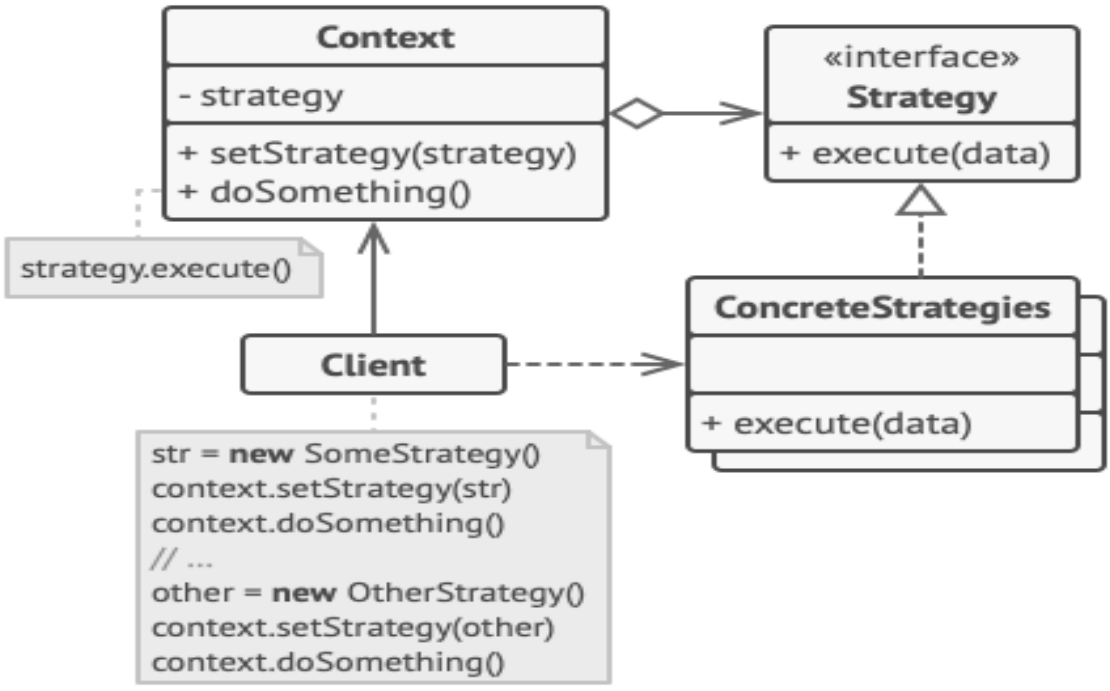
- **Solution**

  The Strategy pattern suggests that you take a class that does something specific in a lot of different ways and extract all these algorithms into separate classes called strategies.

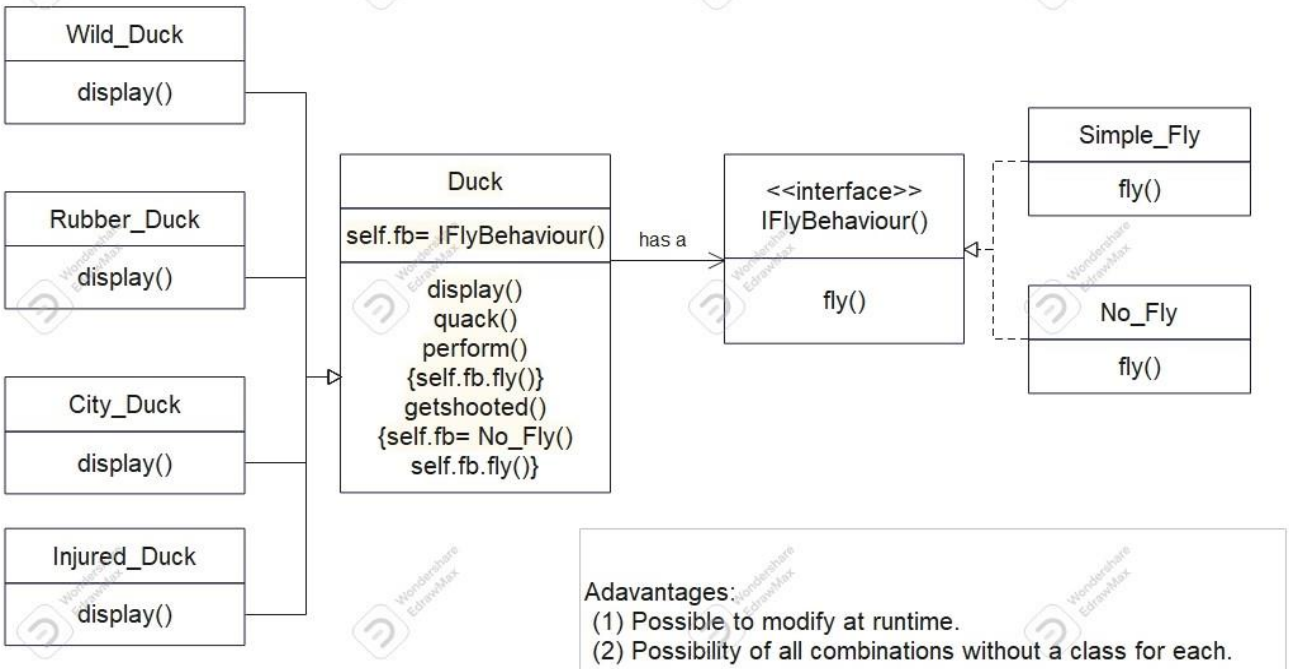
  The original class, called context, must have a field for storing a reference to one of the strategies. The context delegates the work to a linked strategy object instead of executing it on its own.

  The context isn't responsible for selecting an appropriate algorithm for the job. Instead, the client passes the desired strategy to the context. In fact, the context doesn't know much about strategies. It works with all strategies through the same generic interface, which only exposes a single method for triggering the algorithm encapsulated within the selected strategy.

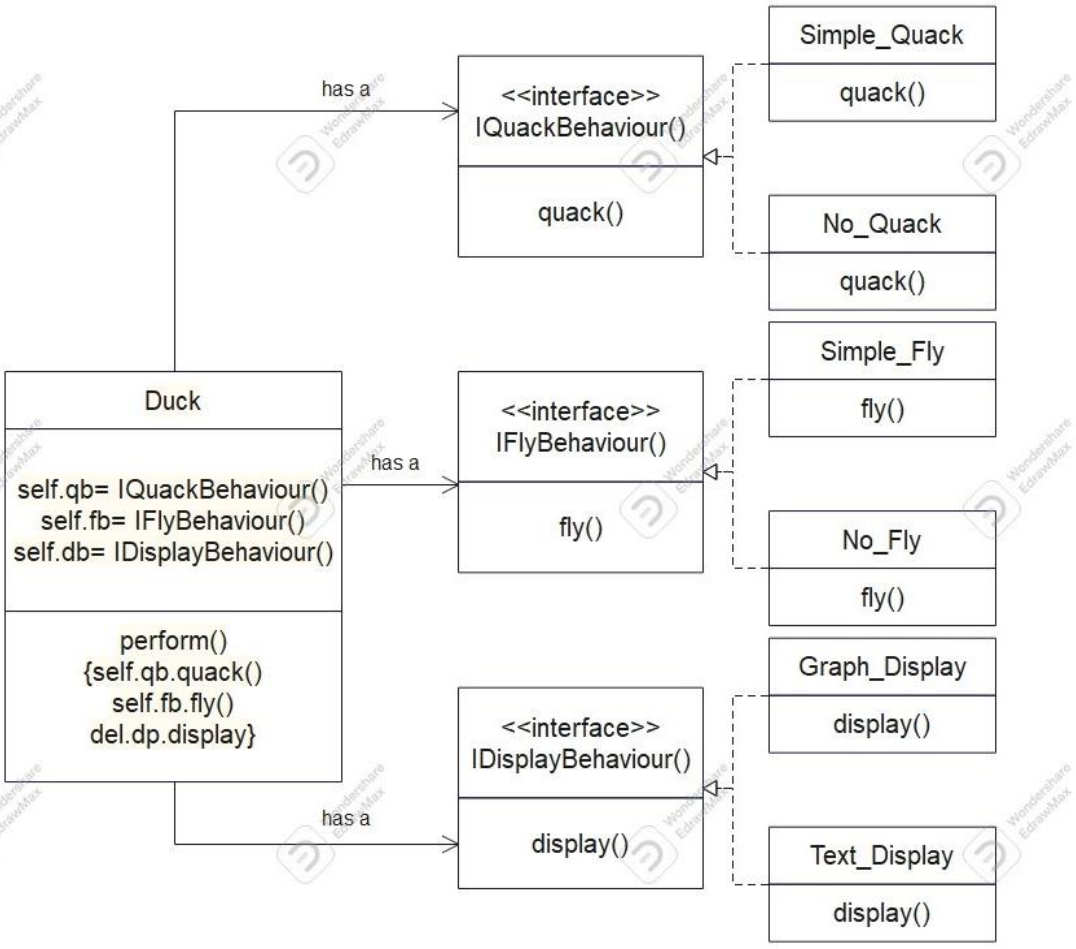
  This way the context becomes independent of concrete strategies, so you can add new algorithms or modify existing ones without changing the code of the context or other strategies.



Strategy Pattern is the solution!




Adavantages:
(1) Possible to modify at runtime.
(2) Possibility of all combinations without a class for each.



```python
class IFlyBehavior:
    def fly(self):
        pass

class CanFly(IFlyBehavior):
    def fly(self):
        print("Flying...")

class CannotFly(IFlyBehavior):
    def fly(self):
        print("Can't fly!")


class IQuackBehavior:
    def quack(self):
        pass

class Quack(IQuackBehavior):
    def quack(self):
        print("Quack!")

class Squeak(IQuackBehavior):
    def quack(self):
        print("Squeak!")

class IDisplayBehavior:
    def display(self):
        pass

class Graph(IDisplayBehavior):
    def display(self):
        print("Graph display!")
```

```
class Text(IDisplayBehavior):
    def display(self):
        print("Text display!")

class Duck:
    def __init__(self, fly_behavior, quack_behavior, display_behavior):
        self.fly_behavior = fly_behavior
        self.quack_behavior = quack_behavior
        self.display_behavior = display_behavior

    def perform(self):
        self.fly_behavior.fly()
        self.quack_behavior.quack()
        self.display_behavior.display()

noFly= CannotFly()
quack= Quack()
text= Text()
injuredDuck= Duck(noFly, quack, text)
injuredDuck.perform()
```

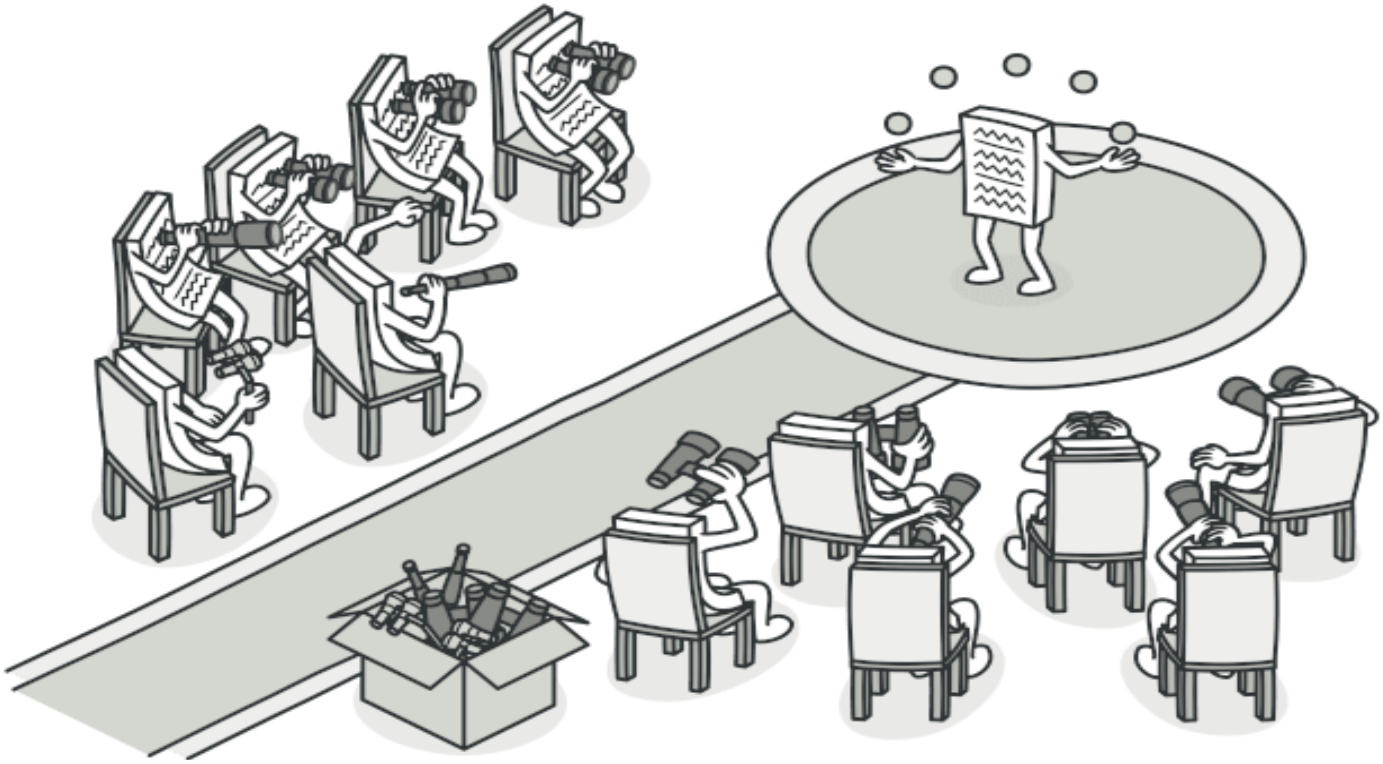
The output:

```
Can't fly!
Quack!
Text display!
```

## 2. Observer

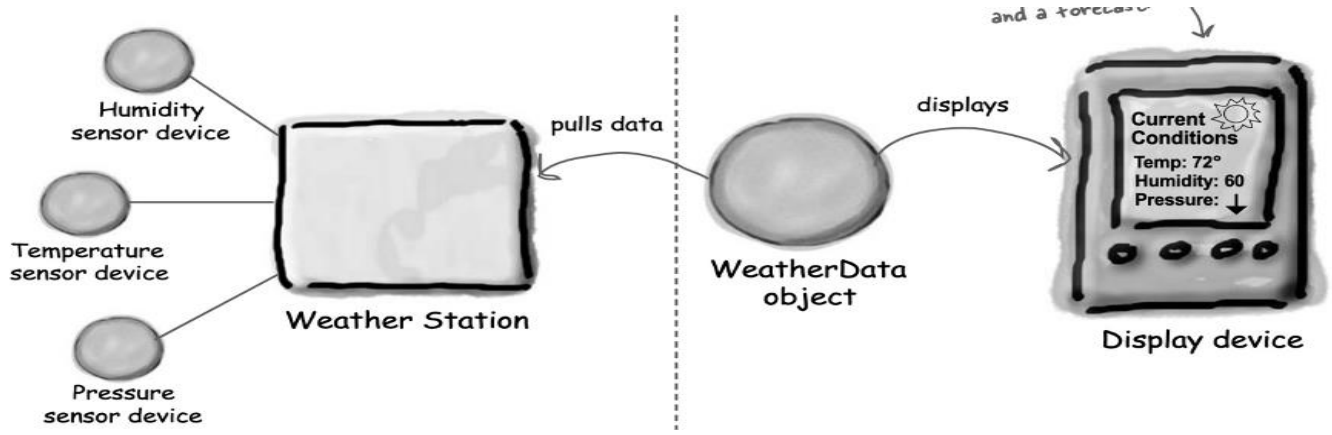
- **Intent**

  **Observer** is a behavioral design pattern that lets you define a subscription mechanism to notify multiple objects about any events that happen to the object they're observing.



- **Problem**

  The Weather Monitoring application overview. The three players in the system are the weather station (the physical device that acquires the actual weather data), the WeatherData object (that tracks the data coming from the Weather Station and updates the displays), and the display that shows users the current weather conditions.



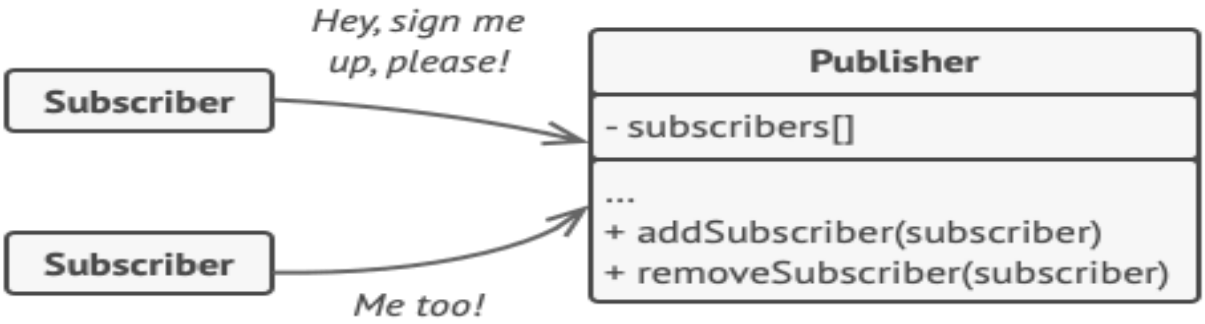
- **Solution**

The object that has some interesting state is often called subject, but since it's also going to notify other objects about the changes to its state, we'll call it publisher. All other objects that want to track changes to the publisher's state are called subscribers.

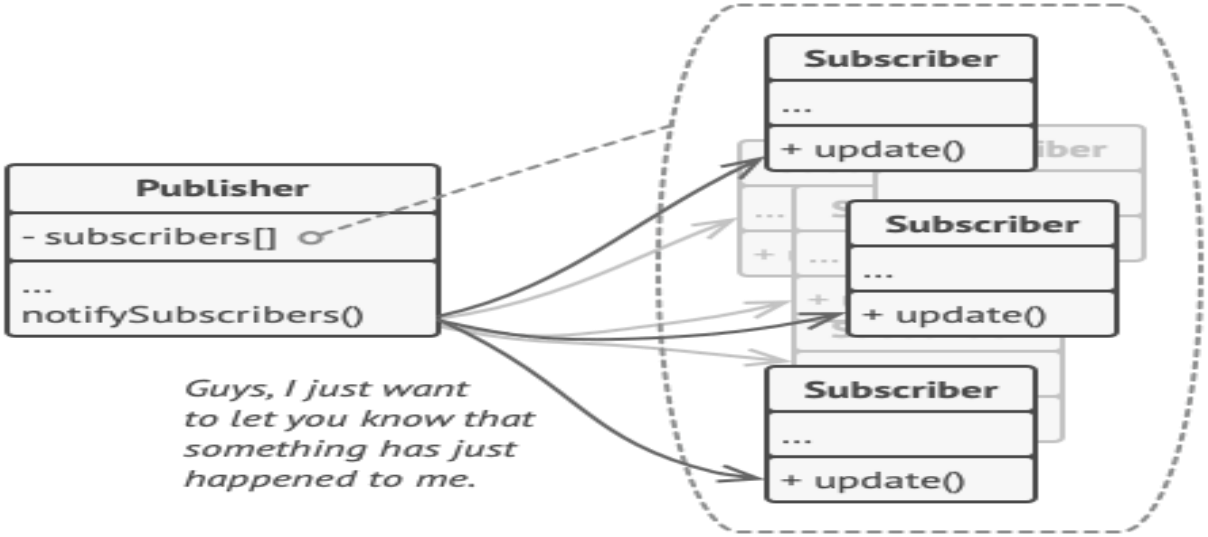
The Observer pattern suggests that you add a subscription mechanism to the publisher class so individual objects can subscribe to or unsubscribe from a stream of events coming from that publisher. Fear not! Everything isn't as complicated as it sounds. This mechanism consists of:

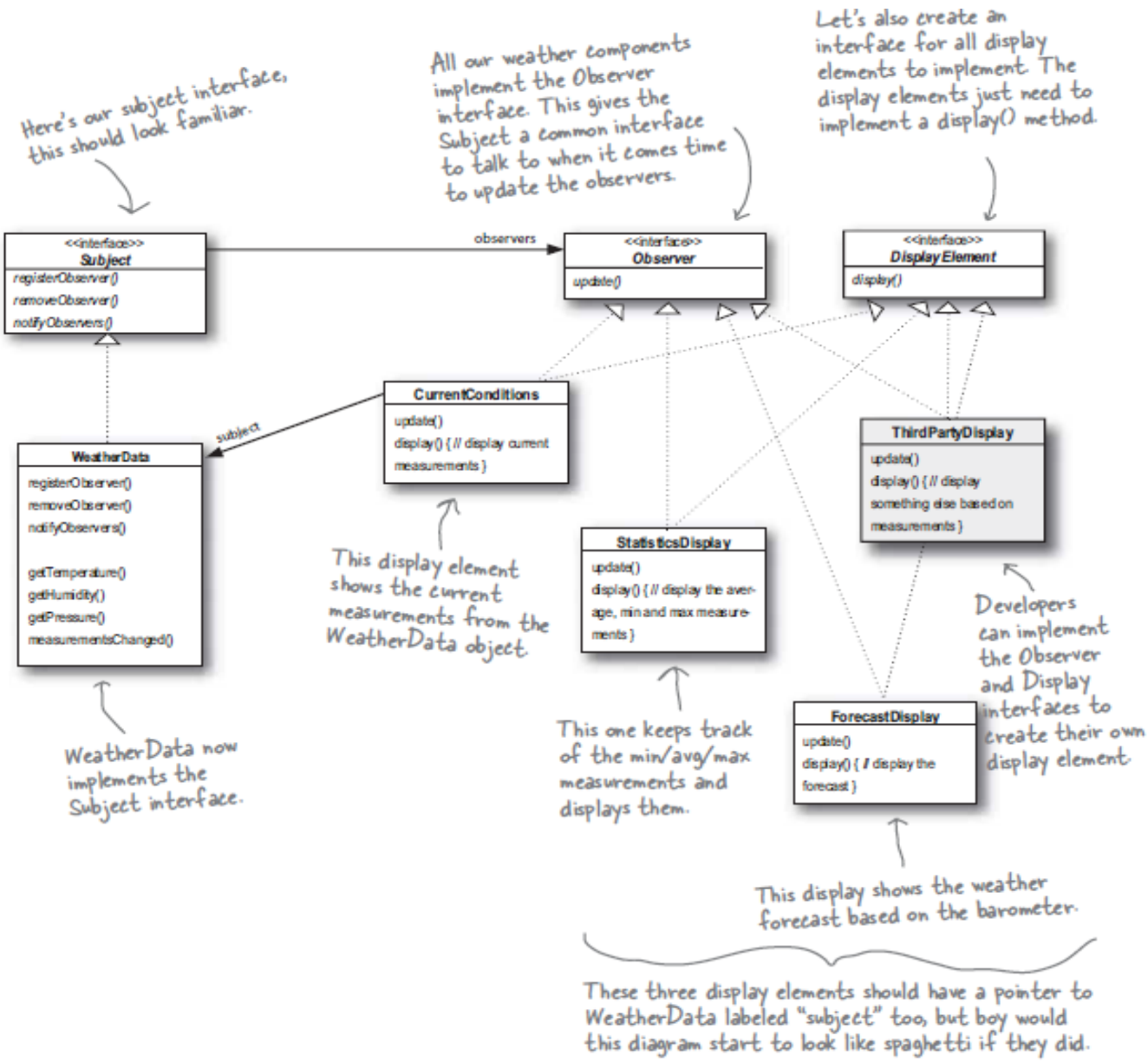
1) an array field for storing a list of references to subscriber objects and

2) several public methods which allow adding subscribers to and removing them from that list.




Now, whenever an important event happens to the publisher, it goes over its subscribers and calls the specific notification method on their objects.






```python
class ISubject:
    def register_observer(self, o):
        pass

    def remove_observer(self, o):
        pass

    def notify_observers(self):
        pass


class IDisplayElement:
    def display(self):
        pass
```

```python
class WeatherData(ISubject):
    def __init__(self):
        self.observers = []
        self.temperature = 0.0
        self.humidity = 0.0
        self.pressure = 0.0

    def register_observer(self, o):
        self.observers.append(o)

    def remove_observer(self, o):
        if o in self.observers:
            self.observers.remove(o)

    def notify_observers(self):
        for observer in self.observers:
            observer.update(self.temperature, self.humidity, self.pressure)

    def measurements_changed(self):
        self.notify_observers()

    def set_measurements(self, temperature, humidity, pressure):
        self.temperature = temperature
        self.humidity = humidity
        self.pressure = pressure
        self.measurements_changed()

class CurrentConditionsDisplay:
    def __init__(self, weather_data):
        self.temperature = 0.0
        self.humidity = 0.0
        self.weather_data = weather_data
        self.weather_data.register_observer(self)

    def update(self, temperature, humidity, pressure):
        self.temperature = temperature
        self.humidity = humidity
        self.display()

    def display(self):
        print("Current Conditions")
        print("  Temperature:", self.temperature)
        print("  Humidity:", self.humidity)
```

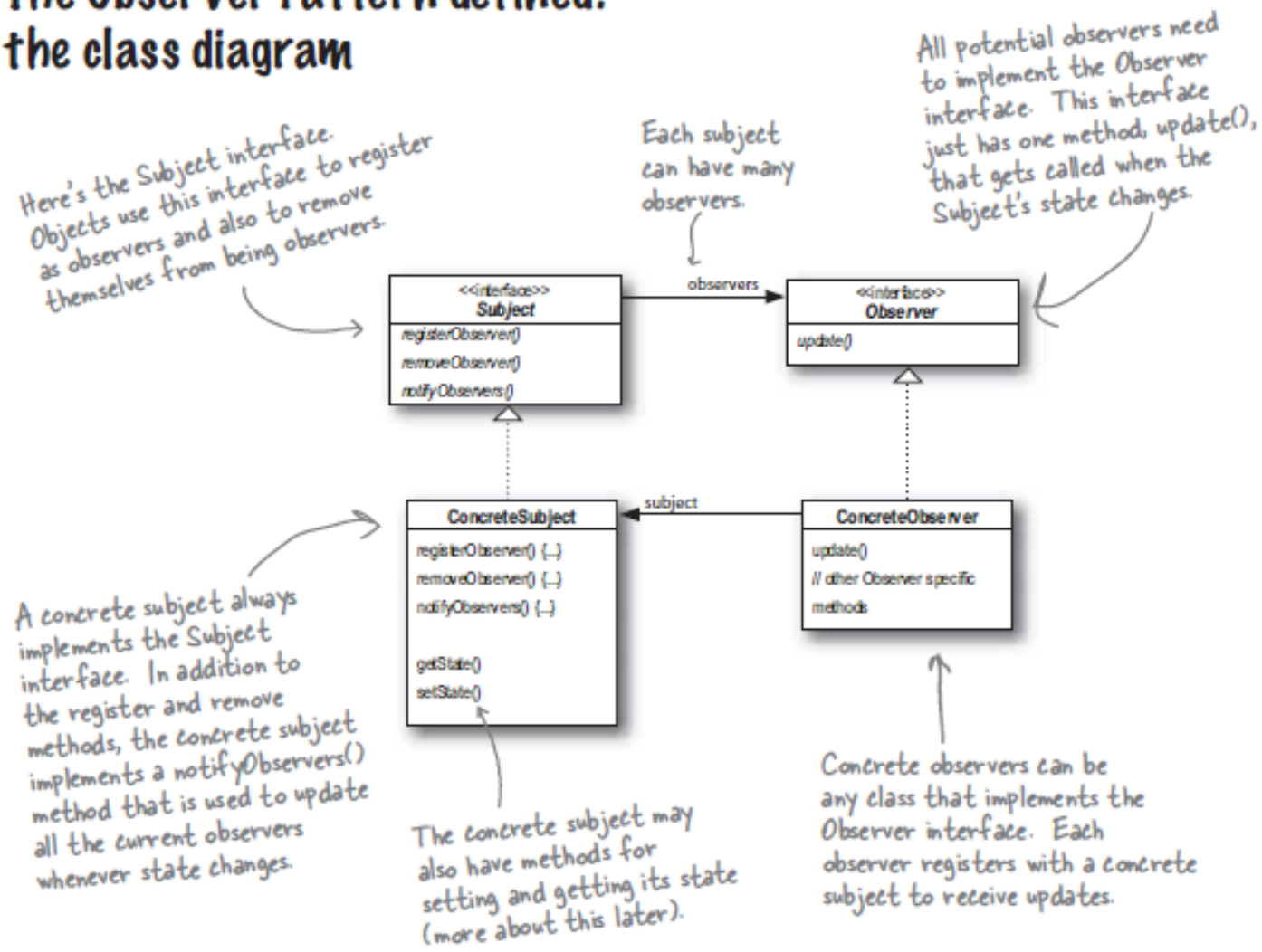
```python
class PhoneDisplay:
    def __init__(self, weather_data):
        self.temperature = 0.0
        self.humidity = 0.0
        self.weather_data = weather_data
        #self.weather_data.register_observer(self)

    def update(self, temperature, humidity, pressure):
        self.temperature = temperature
        self.humidity = humidity
        self.display()

    def display(self):
        print("Phone Display")
        print("  Temperature:", self.temperature)
        print("  Humidity:", self.humidity)


weather_station = WeatherData()
phone_display = PhoneDisplay(weather_station)
weather_station.register_observer(phone_display)
current_conditions_display= CurrentConditionsDisplay(weather_station)
weather_station.set_measurements(25, 15, 1)
```

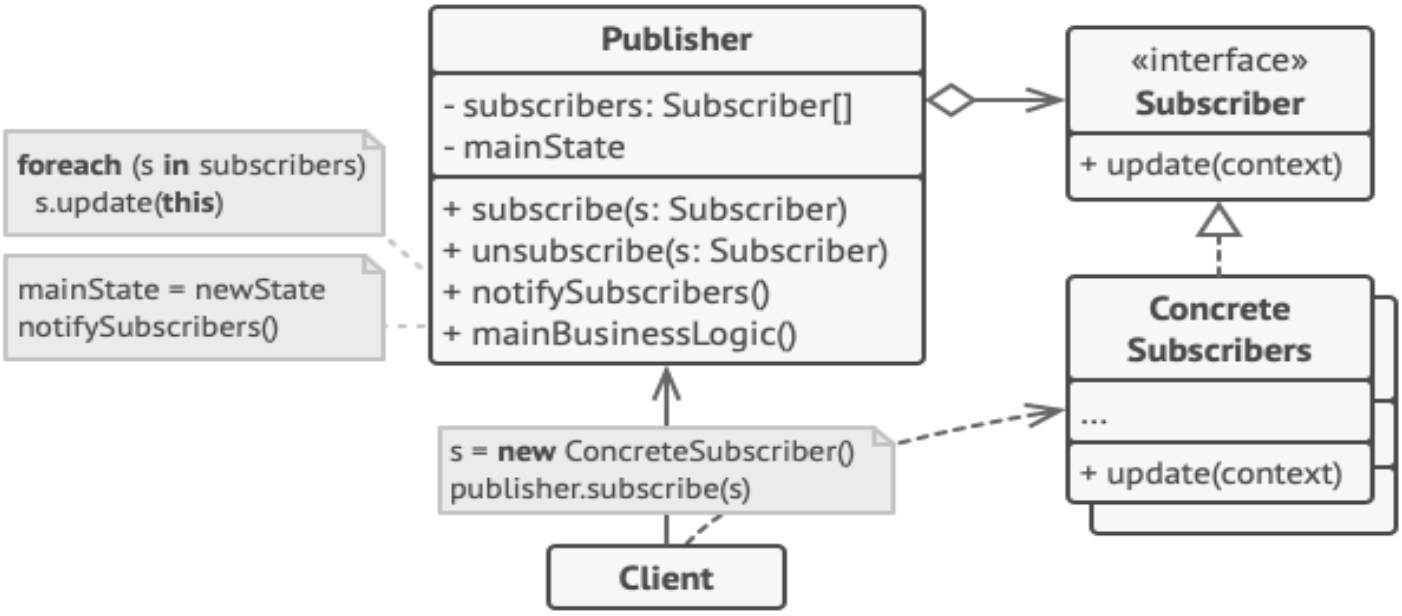
The output:

```
Phone Display
  Temperature: 25
  Humidity: 15
Current Conditions
  Temperature: 25
  Humidity: 15
```

➢ **Structure**

# The Observer Pattern defined: the class diagram

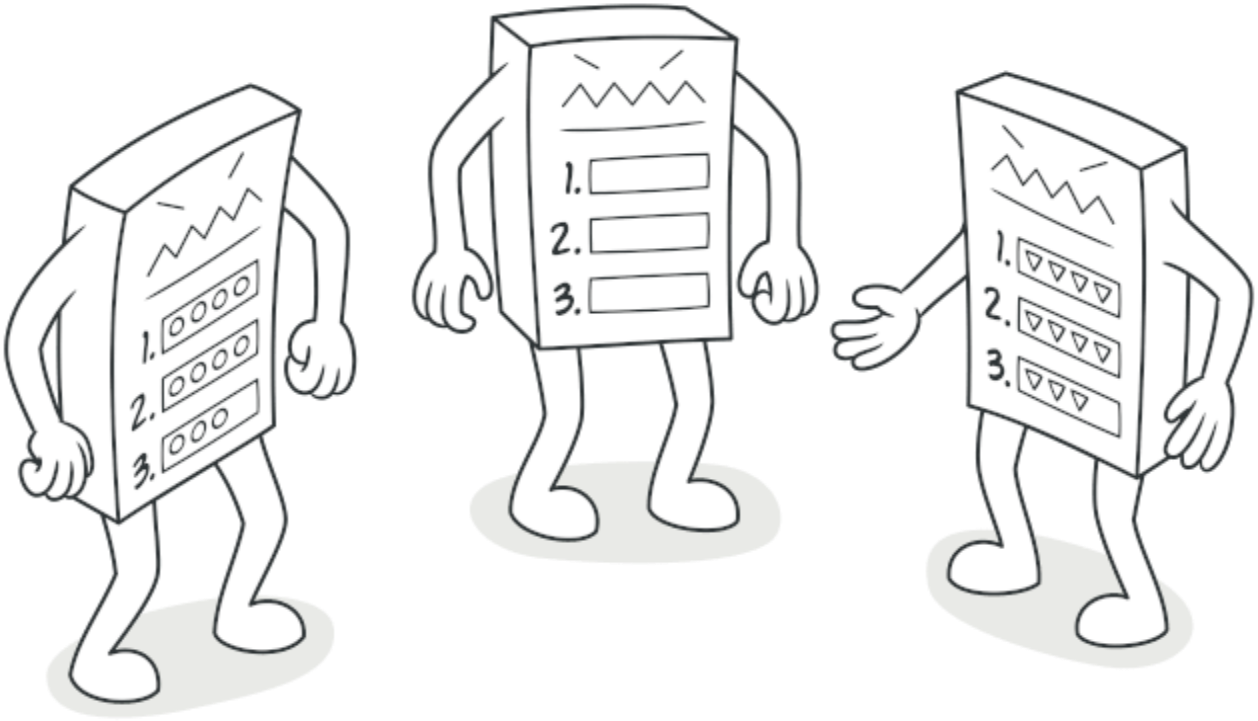


## 3. Template Method

- **Intent**

  **Template Method** is a behavioral design pattern that defines the skeleton of an algorithm in the superclass but lets subclasses override specific steps of the algorithm without changing its structure.




- **Problem**

  It’s time for some more caffeine. Some people can’t live without their coffee; some people can’t live without their tea. The common ingredient? Caffeine of course! But there’s more; tea and coffee are made in very similar ways.

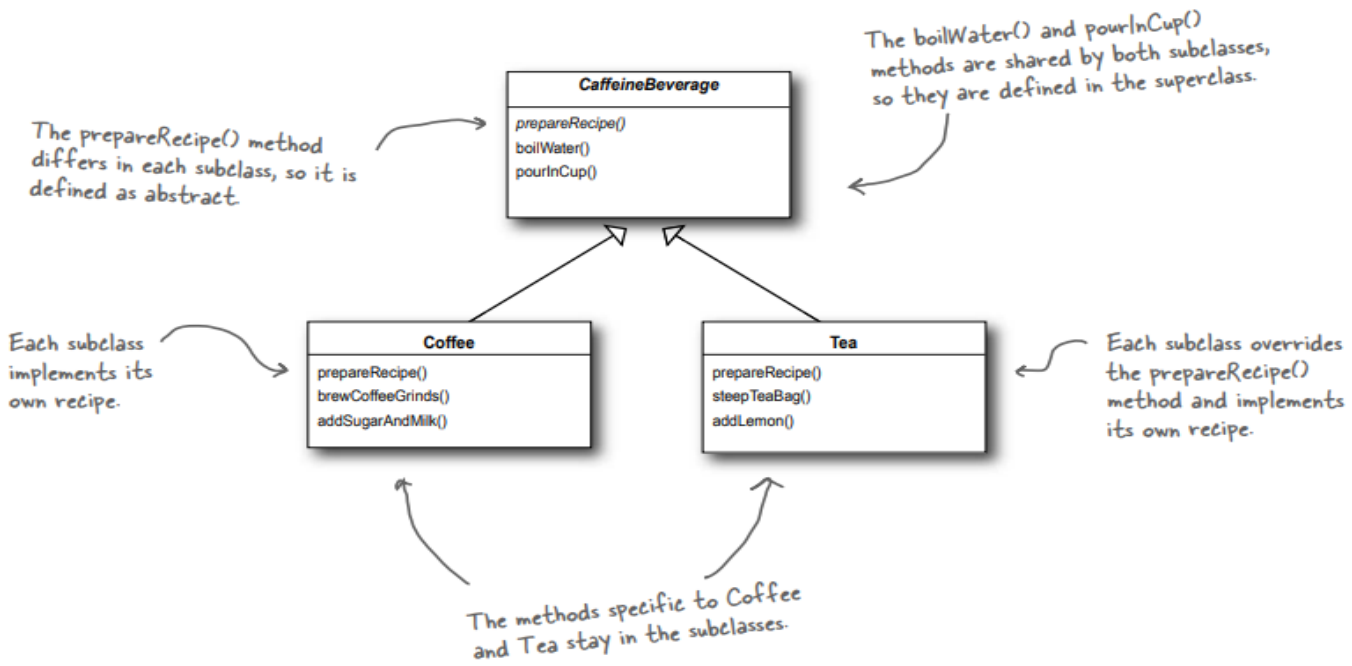
```python
class Tea:
    def prepare(self):
        self.boil_water()
        self.brew()
        self.pour_in_cup()
        self.add_condiments()

    def boil_water(self):
        print("Boiling water")

    def brew(self):
        print("Steeping the tea")

    def pour_in_cup(self):
        print("Pouring water into the cup")

    def add_condiments(self):
        print("Adding lemon")
class Coffee:
    def prepare(self):
        self.boil_water()
        self.brew()
        self.pour_in_cup()
        self.add_condiments()

    def boil_water(self):
        print("Boiling water")

    def brew(self):
        print("Dripping coffee through filter")

    def pour_in_cup(self):
        print("Pouring water into the cup")

    def add_condiments(self):
        print("Adding sugar and milk")

if __name__ == "__main__":
    print("Preparing Tea:")
    tea = Tea()
    tea.prepare()

    print("\nPreparing Coffee:")
    coffee = Coffee()
    coffee.prepare()
```

The output:

```
Preparing Tea:
Boiling water
Steeping the tea
Pouring water into the cup
Adding lemon

Preparing Coffee:
Boiling water
Dripping coffee through filter
Pouring water into the cup
Adding sugar and milk
```

- **Solution**

  The Template Method pattern suggests that you break down an algorithm into a series of steps, turn these steps into methods, and put a series of calls to these methods inside a single template method. The steps may either be abstract or have some default implementation. To use the algorithm, the client is supposed to provide its own subclass, implement all abstract steps, and override some of the optional ones if needed (but not the template method itself).



```python
from abc import ABC, abstractmethod

class BeverageMaker(ABC):
    # Template method defining the overall process
    # Template method defines the skeleton of an algorithm.
    def make_beverage(self):
        self.boil_water()
        self.brew()
        self.pour_in_cup()
        self.add_condiments()

    # Abstract methods to be implemented by subclasses
    @abstractmethod
    def brew(self):
        pass

    @abstractmethod
    def add_condiments(self):
        pass

    # Common methods
    def boil_water(self):
        print("Boiling water")

    def pour_in_cup(self):
        print("Pouring into cup")

class Tea(BeverageMaker):
    def brew(self):
        print("Steeping the tea")

    def add_condiments(self):
        print("Adding lemon")
```

```python
class Coffee(BeverageMaker):
    def brew(self):
        print("Dripping coffee through filter")

    def add_condiments(self):
        print("Adding sugar and milk")

if __name__ == "__main__":
    print("Preparing Tea:")
    tea = Tea()
    tea.make_beverage()

    print("\nPreparing Coffee:")
    coffee = Coffee()
    coffee.make_beverage()
```

The output:

```
Preparing Tea:
Boiling water
Steeping the tea
Pouring into cup
Adding lemon

Preparing Coffee:
Boiling water
Dripping coffee through filter
Pouring into cup
Adding sugar and milk
```

Hooks provide additional extension points in some crucial places of the algorithm. Subclasses may override them, but it's not mandatory since the hooks already have default (but empty) implementation. Here is the code withing hooks:

```python
from abc import ABC, abstractmethod

class BeverageMaker(ABC):
    def __init__(self, use_condiments=True):
        self.use_condiments = use_condiments

    # Template method defining the overall process
    # Template method defines the skeleton of an algorithm.
    def make_beverage(self):
        self.boil_water()
        self.brew()
        self.pour_in_cup()
        # These are "hooks." Subclasses may override them, but it's not mandatory
        # since the hooks already have default (but empty) implementation. Hooks
        # provide additional extension points in some crucial places of the
        # algorithm.
        if self.use_condiments:
            self.add_condiments()

    # Abstract methods to be implemented by subclasses
    @abstractmethod
    def brew(self):
        pass

    @abstractmethod
    def add_condiments(self):
        pass

    # Common methods
    def boil_water(self):
        print("Boiling water")

    def pour_in_cup(self):
        print("Pouring into cup")
```

```python
class Tea(BeverageMaker):
    def brew(self):
        print("Steeping the tea")

    def add_condiments(self):
        print("Adding lemon")

class Coffee(BeverageMaker):
    def brew(self):
        print("Dripping coffee through filter")

    def add_condiments(self):
        print("Adding sugar and milk")

if __name__ == "__main__":
    print("Preparing Tea:")
    tea = Tea(True)
    tea.make_beverage()

    print("\nPreparing Tea (without condiments):")
    tea = Tea(False)
    tea.make_beverage()
```

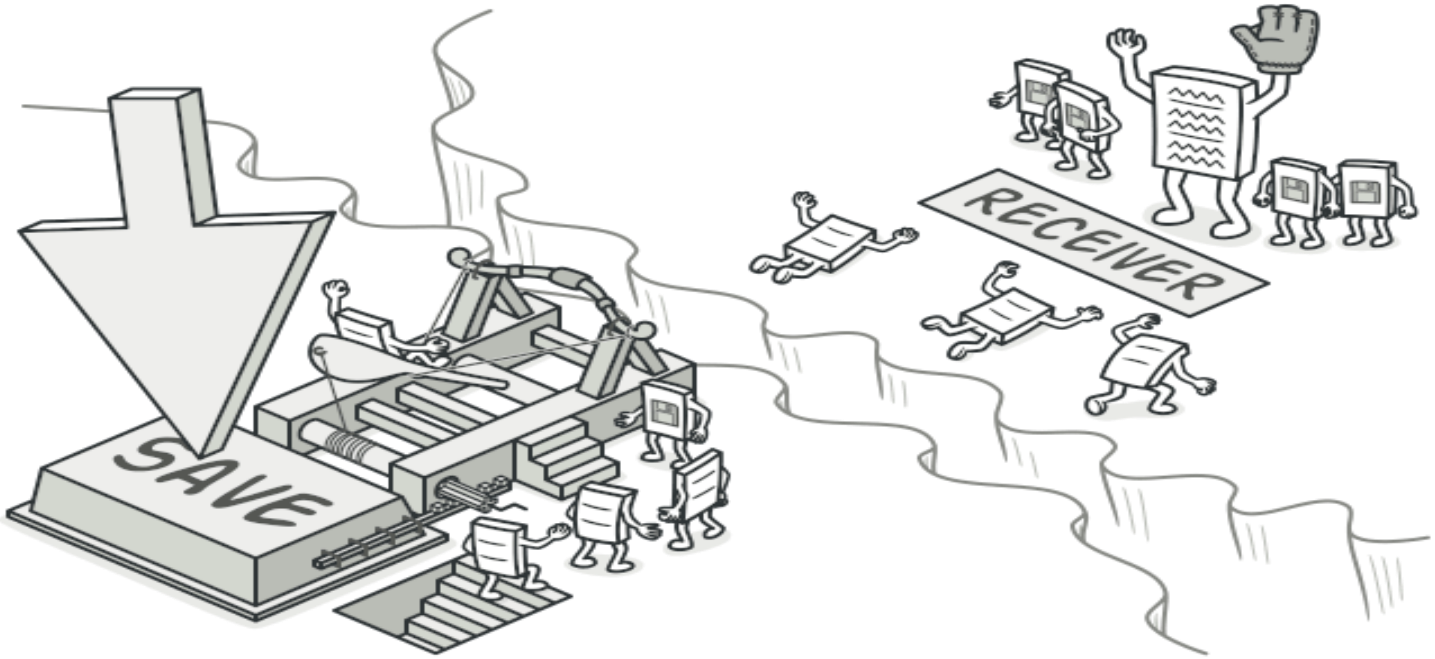
The output:

```
Preparing Tea:
Boiling water
Steeping the tea
Pouring into cup
Adding lemon

Preparing Tea (without condiments):
Boiling water
Steeping the tea
Pouring into cup
```

# 4. Command

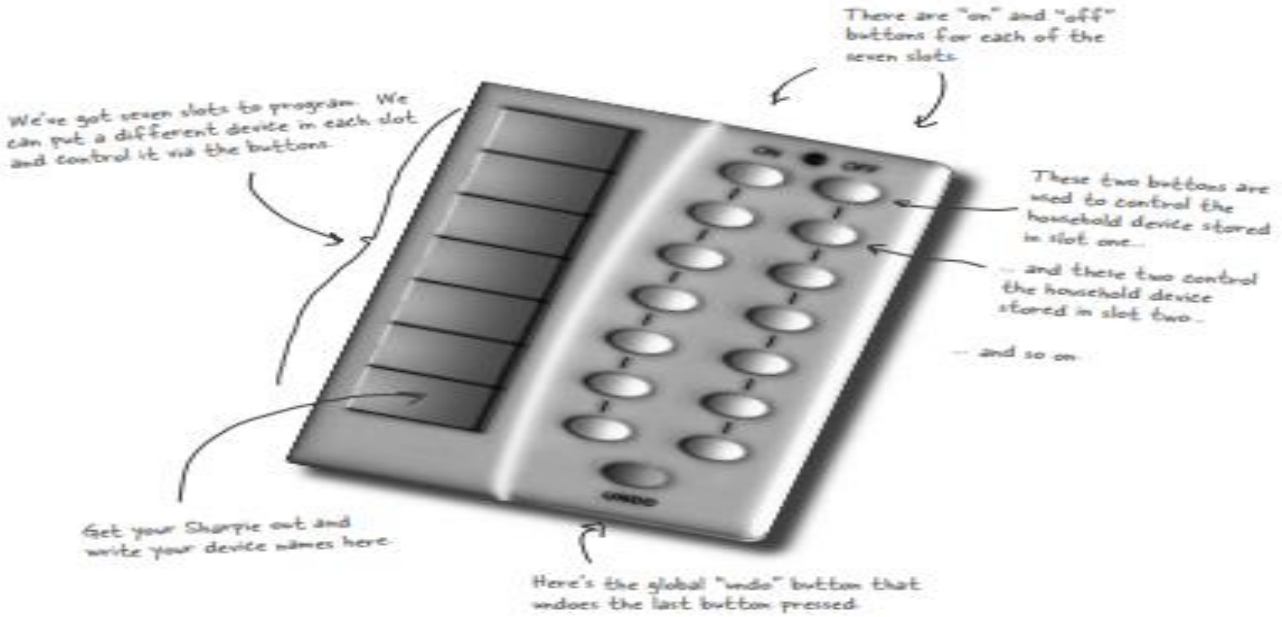
- **Intent**

  **Command** is a behavioral design pattern that turns a request into a stand-alone object that contains all information about the request. This transformation lets you pass requests as a method argument, delay or queue a request's execution, and support undoable operations.




- **Problem**

  Free hardware! Let's check out the Remote Control...

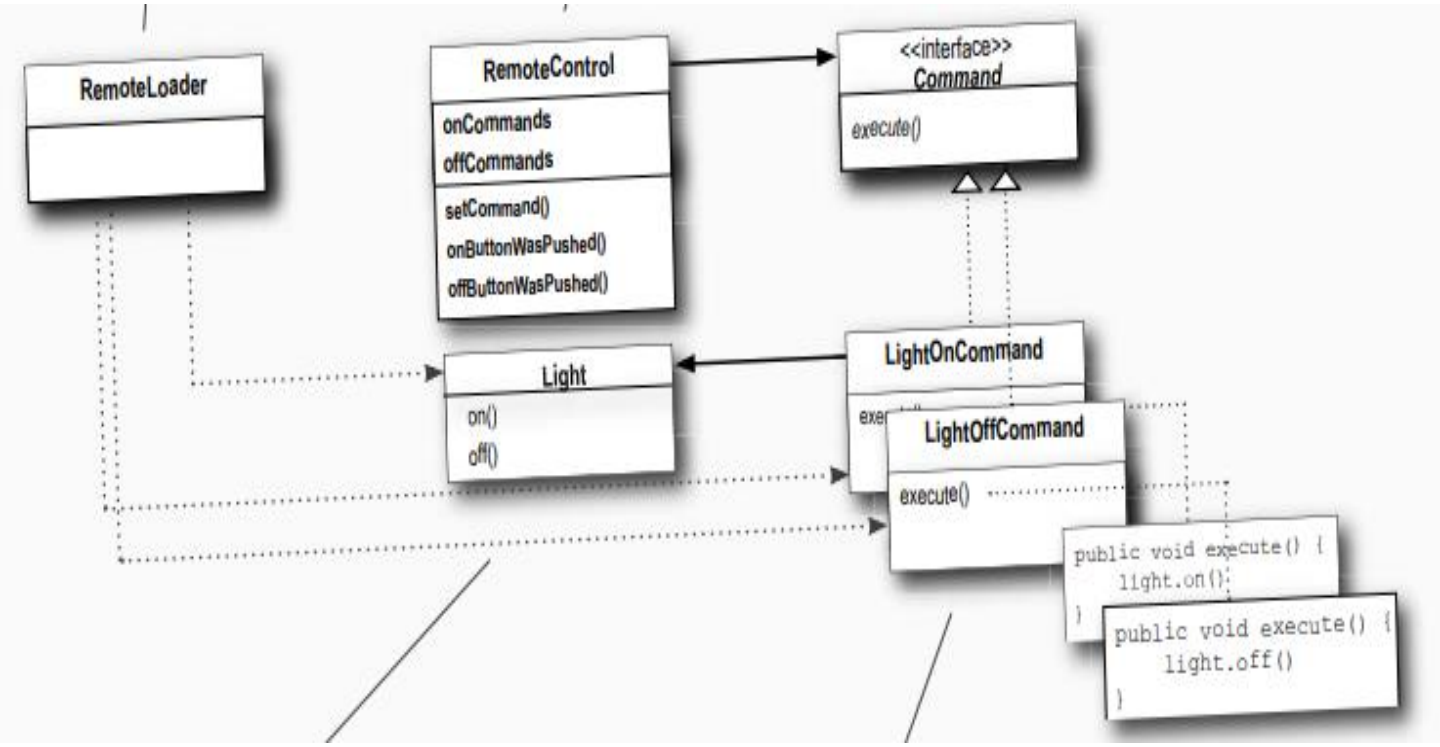


- **Solution**

  The Command pattern suggests that GUI objects shouldn't send these requests directly. Instead, you should extract all of the request details, such as the object being called, the name of the method and the list of arguments into a separate command class with a single method that triggers this request.

  Command objects serve as links between various GUI and business logic objects. From now on, the GUI object doesn't need to know what business logic object will receive the request and how it'll be processed. The GUI object just triggers the command, which handles all the details.

  The next step is to make your commands implement the same interface. Usually, it has just a single execution method that takes no parameters. This interface lets you use various commands with the same request sender, without coupling it to concrete classes of commands. As a bonus, now you can switch command objects linked to the sender, effectively changing the sender's behavior at runtime.



```python
# Command interface
class Command:
    def execute(self):
        pass

    def unexecute(self):
        pass

# Concrete command for turning a device on
class TurnOnCommand(Command):
    def __init__(self, device):
        self.device = device
        self.previous_state = None

    def execute(self):
        self.previous_state = self.device.get_state()
        self.device.turn_on()

    def unexecute(self):
        if self.previous_state:
            self.device.set_state(self.previous_state)

# Concrete command for turning a device off
class TurnOffCommand(Command):
    def __init__(self, device):
        self.device = device
        self.previous_state = None

    def execute(self):
        self.previous_state = self.device.get_state()
        self.device.turn_off()

    def unexecute(self):
        if self.previous_state:
            self.device.set_state(self.previous_state)
```

```python
# Receiver interface
class Device:
    def turn_on(self):
        pass

    def turn_off(self):
        pass

    def get_state(self):
        pass

    def set_state(self, state):
        pass
```

```python
# Concrete receiver for a TV
class TV(Device):
    def __init__(self):
        self.is_on = False

    def turn_on(self):
        print("TV is now on")
        self.is_on = True

    def turn_off(self):
        print("TV is now off")
        self.is_on = False

    def get_state(self):
        return "on" if self.is_on else "off"

    def set_state(self, state):
        if state == "on":
            self.turn_on()
        else:
            self.turn_off()
```

```python
# Concrete receiver for a stereo
class Stereo(Device):
    def __init__(self):
        self.is_on = False

    def turn_on(self):
        print("Stereo is now on")
        self.is_on = True

    def turn_off(self):
        print("Stereo is now off")
        self.is_on = False

    def get_state(self):
        return "on" if self.is_on else "off"

    def set_state(self, state):
        if state == "on":
            self.turn_on()
        else:
            self.turn_off()
```

```python
# Invoker
class RemoteControl:
    def __init__(self):
        self.on_commands = [None] * 4
        self.off_commands = [None] * 4
        self.undo_stack = []

    def set_command(self, slot, on_command, off_command):
        self.on_commands[slot] = on_command
        self.off_commands[slot] = off_command

    def press_on_button(self, slot):
        if self.on_commands[slot]:
            self.on_commands[slot].execute()
            self.undo_stack.append(self.on_commands[slot])

    def press_off_button(self, slot):
        if self.off_commands[slot]:
            self.off_commands[slot].execute()
            self.undo_stack.append(self.off_commands[slot])

    def press_undo_button(self):
        if self.undo_stack:
            last_command = self.undo_stack.pop()
            last_command.unexecute()
```

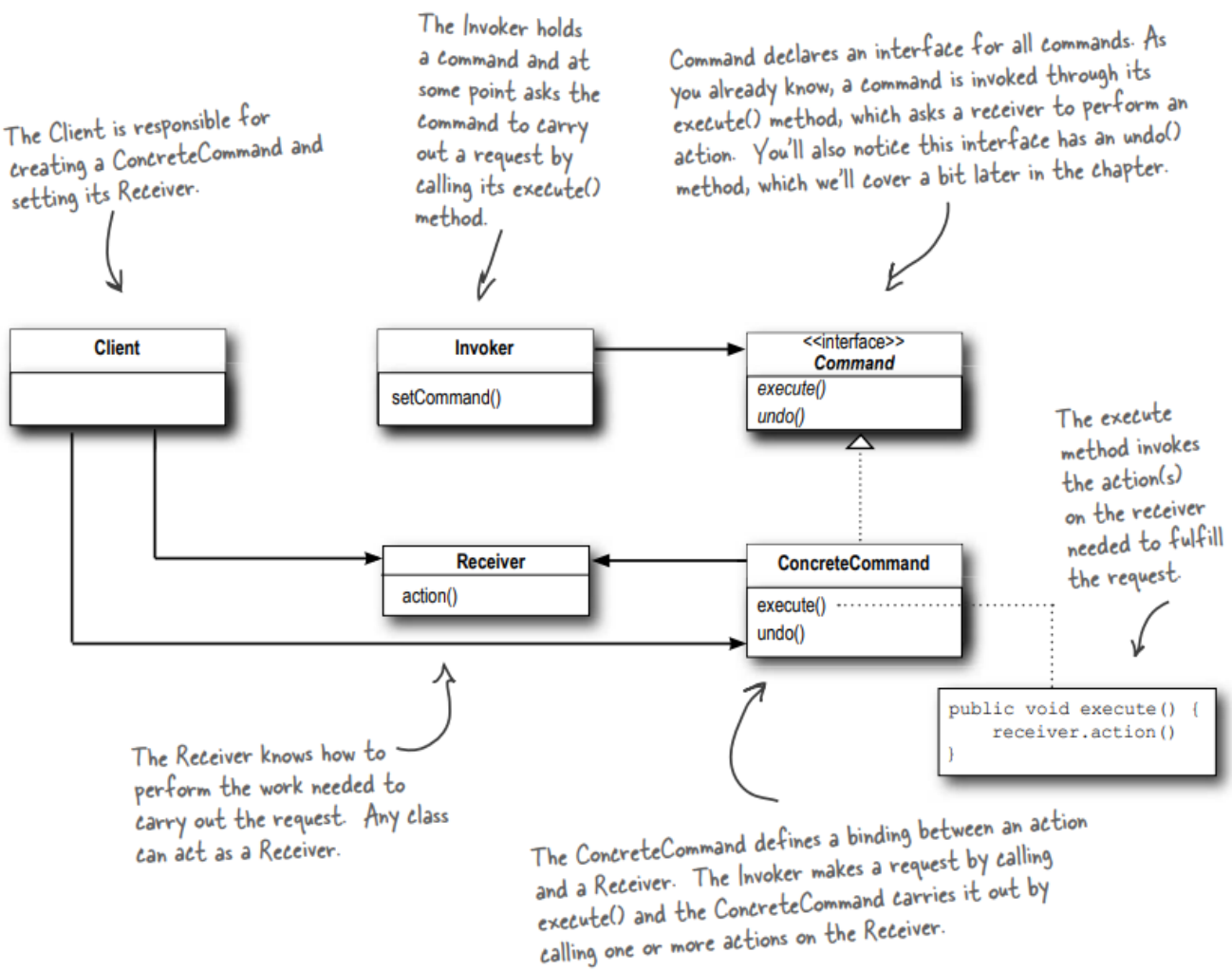
```python
if __name__ == "__main__":
    # Create devices
    tv = TV()
    stereo = Stereo()

    # Create command objects
    turn_on_tv_command = TurnOnCommand(tv)
    turn_off_tv_command = TurnOffCommand(tv)
    turn_on_stereo_command = TurnOnCommand(stereo)
    turn_off_stereo_command = TurnOffCommand(stereo)

    # Create remote control
    remote = RemoteControl()

    # Set commands for slots
    remote.set_command(0, turn_on_tv_command, turn_off_tv_command)
    remote.set_command(1, turn_on_stereo_command, turn_off_stereo_command)

    # Press buttons
    remote.press_on_button(0)  # Outputs: TV is now on
    remote.press_off_button(0)  # Outputs: TV is now off
    remote.press_on_button(1)  # Outputs: Stereo is now on
    remote.press_off_button(1)  # Outputs: Stereo is now off

    # Undo last action
    remote.press_undo_button()  # Outputs: Stereo is now on (undo last action)
    remote.press_undo_button()
    remote.press_undo_button()
    remote.press_undo_button()
```

The output:

```
TV is now on
TV is now off
Stereo is now on
Stereo is now off
Stereo is now on
Stereo is now off
TV is now on
TV is now off
```

➢ **Structure**

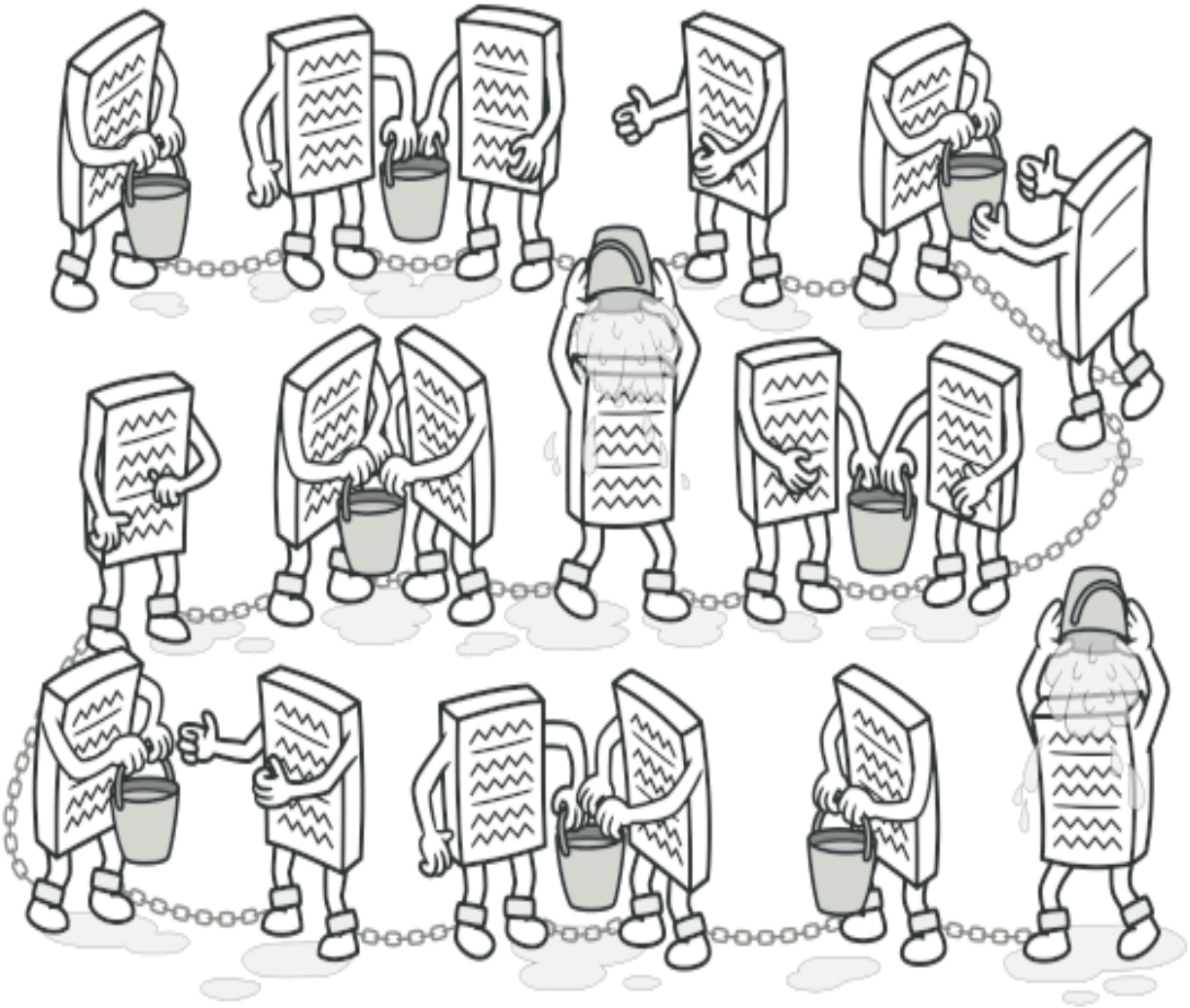
# The Command Pattern defined: the class diagram



## 5. Chain of Responsibility

- **Intent**

  **Chain of Responsibility** is a behavioral design pattern that lets you pass requests along a chain of handlers. Upon receiving a request, each handler decides either to process the request or to pass it to the next handler in the chain.



- **Problem**

  Create a Mathematical Handler

➢ **Solution**

Like many other behavioral design patterns, the **Chain of Responsibility** relies on transforming particular behaviors into stand-alone objects called *handlers*. In our case, each check should be extracted to its own class with a single method that performs the check. The request, along with its data, is passed to this method as an argument.

The pattern suggests that you link these handlers into a chain. Each linked handler has a field for storing a reference to the next handler in the chain. In addition to processing a request, handlers pass the request further along the chain. The request travels along the chain until all handlers have had a chance to process it.

Here's the best part: a handler can decide not to pass the request further down the chain and effectively stop any further processing.

```
from abc import ABC , abstractmethod


class Chain(ABC):
    @abstractmethod
    def setNextChain(self , nextChain):
        pass

    @abstractmethod
    def calculate(self , request):
        pass


class Numbers:
    def __init__(self , newNumber1 , newNumber2 , calcWanted):
        self.number1 = newNumber1
        self.number2 = newNumber2
        self.calculationWanted = calcWanted

    def getNumber1(self):
        return self.number1

    def getNumber2(self):
        return self.number2

    def getCalcWanted(self):
        return self.calculationWanted
```

```python
class AddNumbers(Chain):
    def setNextChain(self , nextChain):
        self.nextInChain = nextChain

    def calculate(self , request):
        if request.getCalcWanted() == "add":
            print(f"{request.getNumber1()} + {request.getNumber2()} = {request.getNumber1() + request.getNumber2()}")
        else:
            self.nextInChain.calculate(request)


class SubtractNumbers(Chain):
    def setNextChain(self , nextChain):
        self.nextInChain = nextChain

    def calculate(self , request):
        if request.getCalcWanted() == "sub":
            print(f"{request.getNumber1()} - {request.getNumber2()} = {request.getNumber1() - request.getNumber2()}")
        else:
            self.nextInChain.calculate(request)


class MultNumbers(Chain):
    def setNextChain(self , nextChain):
        self.nextInChain = nextChain

    def calculate(self , request):
        if request.getCalcWanted() == "mult":
            print(f"{request.getNumber1()} * {request.getNumber2()} = {request.getNumber1() * request.getNumber2()}")
        else:
            self.nextInChain.calculate(request)
```

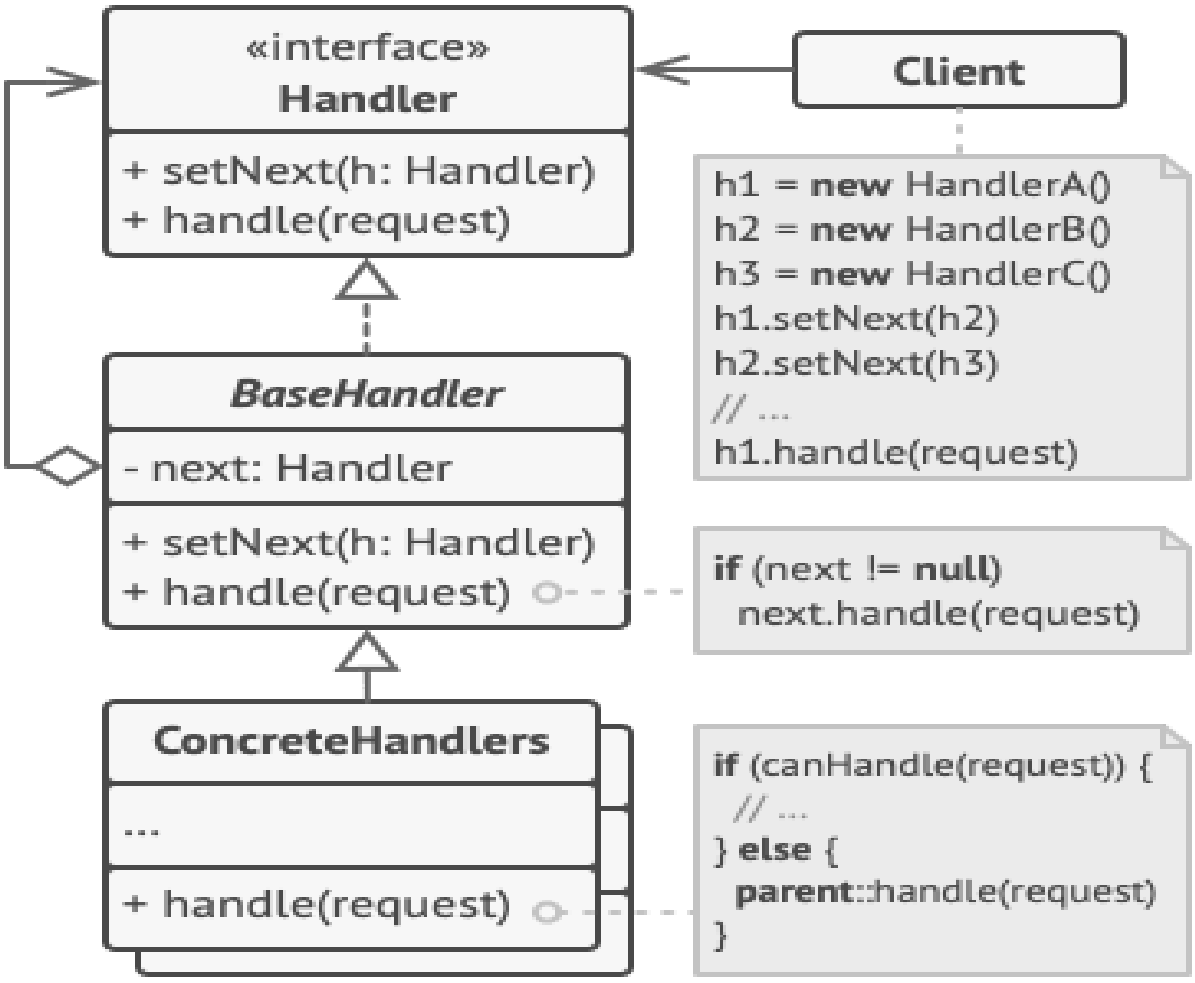
```python
class DivideNumbers(Chain):
    def setNextChain(self , nextChain):
        self.nextInChain = nextChain

    def calculate(self , request):
        if request.getCalcWanted() == "div":
            print(f"{request.getNumber1()} / {request.getNumber2()} = {request.getNumber1() / request.getNumber2()}")
        else:
            print("Only works for add, sub, mult, and div")
```

```
class TestCalcChain:
    @staticmethod
    def main():
        # Here I define all of the objects in the chain
        chainCalc1 = AddNumbers()
        chainCalc2 = SubtractNumbers()
        chainCalc3 = MultNumbers()
        chainCalc4 = DivideNumbers()

        # Here I tell each object where to forward the
        # data if it can't process the request
        chainCalc1.setNextChain(chainCalc2)
        chainCalc2.setNextChain(chainCalc3)
        chainCalc3.setNextChain(chainCalc4)

        # Define the data in the Numbers Object
        # and send it to the first Object in the chain
        request = Numbers(4 , 2 , "add")
        chainCalc1.calculate(request)

        request = Numbers(4 , 2 , "sub")
        chainCalc1.calculate(request)

        request = Numbers(4 , 2 , "mult")
        chainCalc1.calculate(request)

        request = Numbers(4 , 2 , "div")
        chainCalc1.calculate(request)

# Test Drive
if __name__ == "__main__":
    TestCalcChain.main()
```

The output:

```
4 + 2 = 6
4 - 2 = 2
4 * 2 = 8
4 / 2 = 2.0
```

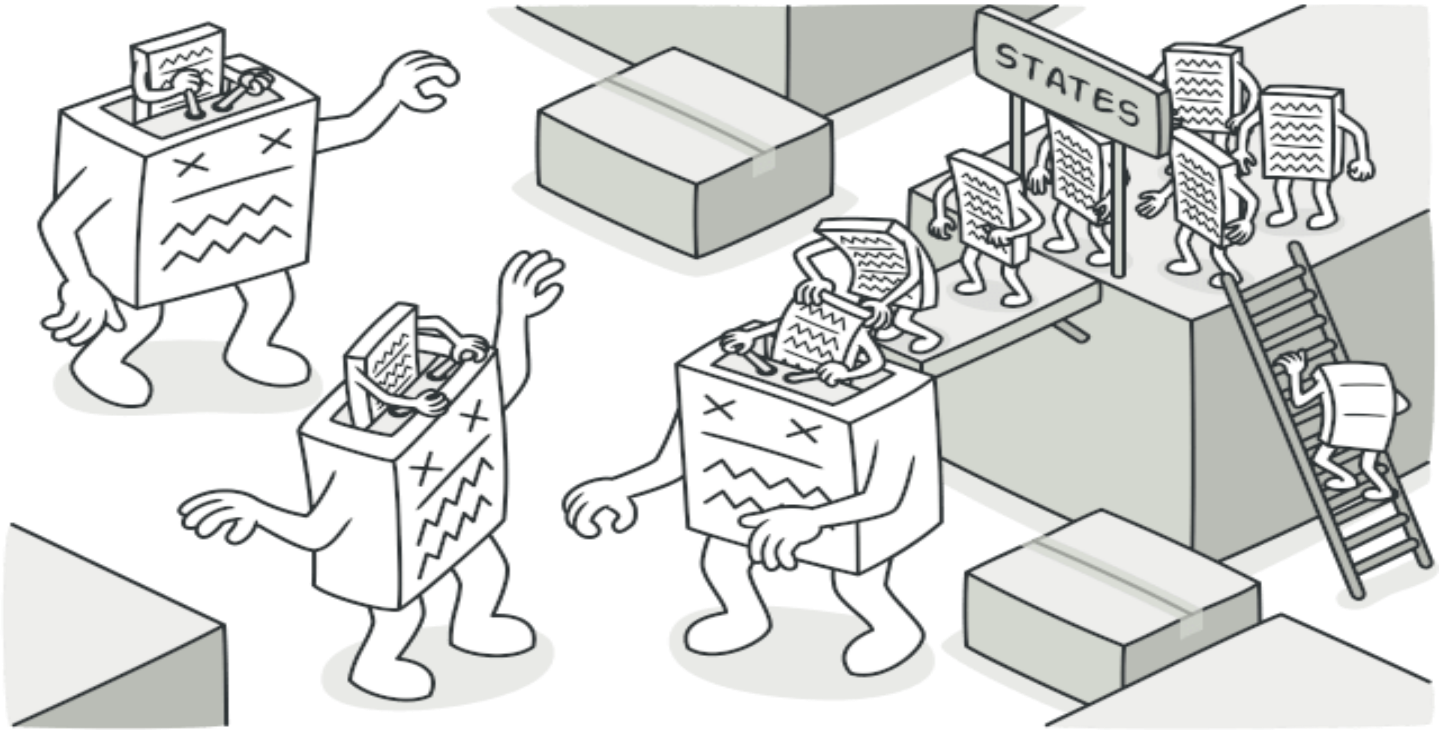
➢ **Structure**


«interface»
Handler
+ setNext(h: Handler)
+ handle(request)
Client
h1 = new HandlerA()
h2 = new HandlerB()
h3 = new HandlerC()
h1.setNext(h2)
h2.setNext(h3)
// ...
h1.handle(request)
BaseHandler
- next: Handler
+ setNext(h: Handler)
+ handle(request)
if (next != null)
next.handle(request)
ConcreteHandlers
...
+ handle(request)
if (canHandle(request)) {
// ...
} else {
parent::handle(request)
}

## 6. State

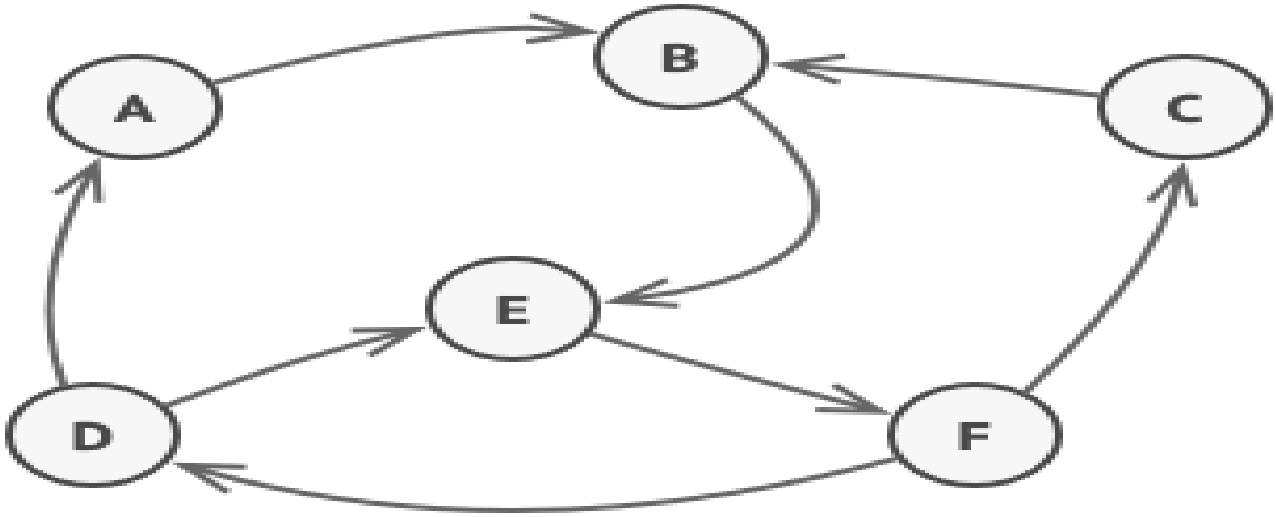
- **Intent**

  **State** is a behavioral design pattern that lets an object alter its behavior when its internal state changes. It appears as if the object changed its class.




- **Problem**

The State pattern is closely related to the concept of a *Finite-State Machine*

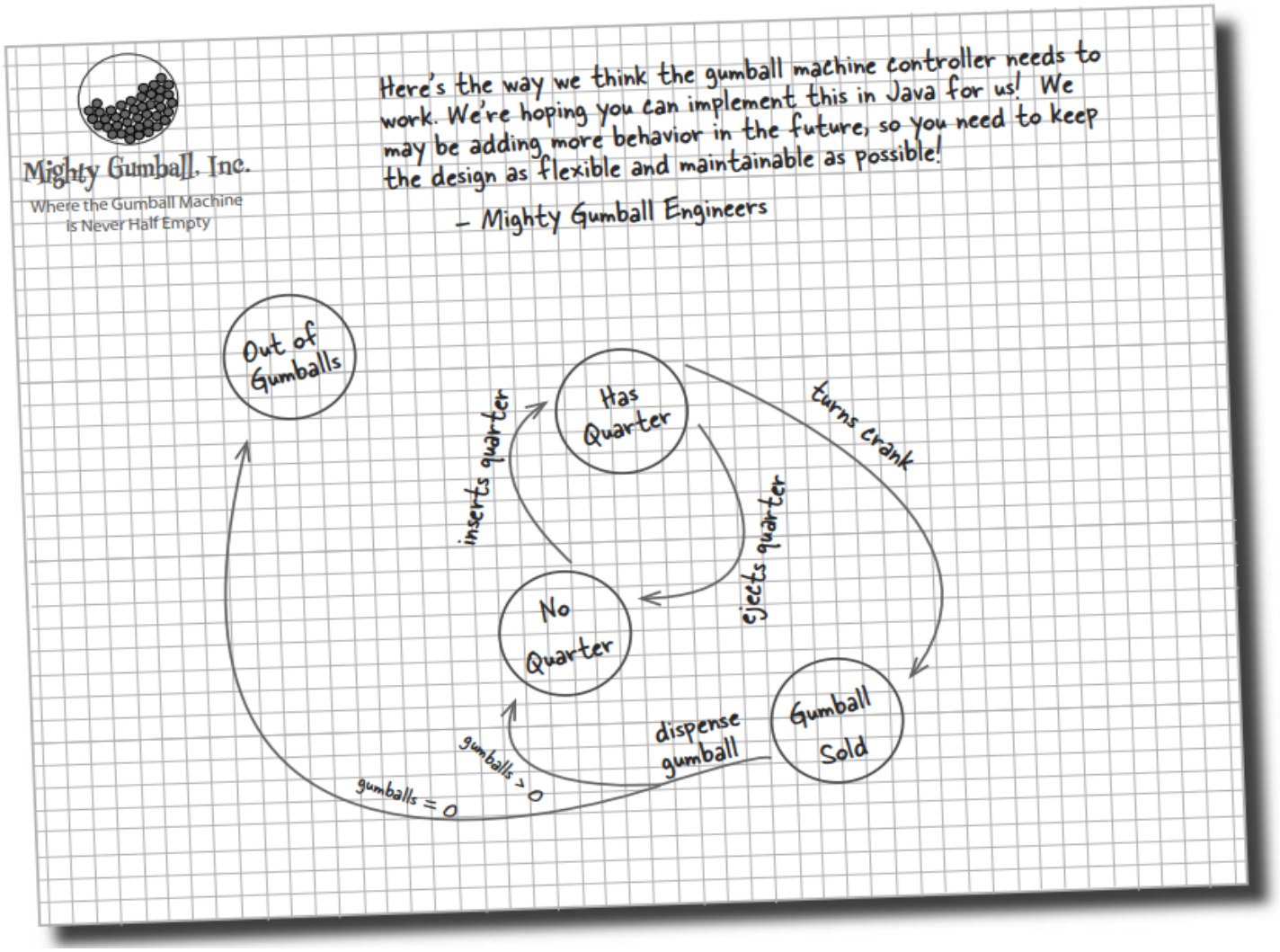



The main idea is that, at any given moment, there's a *finite* number of *states* which a program can be in. Within any unique state, the program behaves differently, and the program can be switched from one state to another instantaneously.

However, depending on a current state, the program may or may not switch to certain other states. These switching rules, called *transitions*, are also finite and predetermined.

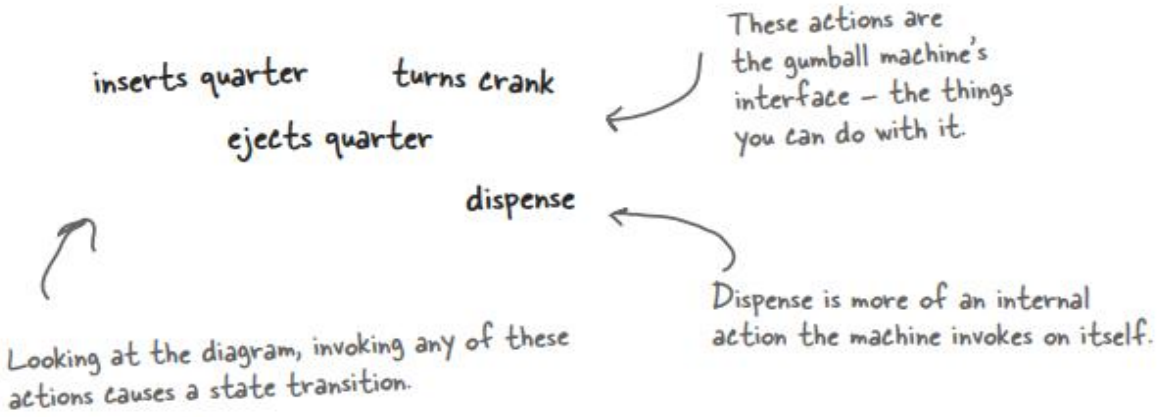
State machines are usually implemented with lots of conditional statements (if or switch) that select the appropriate behavior depending on the current state of the object. The biggest weakness of a state machine based on conditionals reveals itself once we start adding more and more states and state-dependent behaviors to the class. Most methods will contain monstrous conditionals that pick the proper behavior of a method according to the current state. Code like this is very difficult to maintain because any change to the transition logic may require changing state conditionals in every method.

The problem tends to get bigger as a project evolves. It's quite difficult to predict all possible states and transitions at the design stage.



I. First, gather up your states:


II. Now we gather up all the actions that can happen in the system:




III. Writing the code

```
class GumballMachine:
    SOLD_OUT = 0
    NO_QUARTER = 1
    HAS_QUARTER = 2
    SOLD = 3

    Menna Yousri
    def __init__(self, count):
        self.state = self.SOLD_OUT
        self.count = count
        if count > 0:
            self.state = self.NO_QUARTER

    Menna Yousri
    def insertQuarter(self):
        if self.state == self.HAS_QUARTER:
            print("You can't insert another quarter")
        elif self.state == self.NO_QUARTER:
            self.state = self.HAS_QUARTER
            print("You inserted a quarter")
        elif self.state == self.SOLD_OUT:
            print("You can't insert a quarter, the machine is sold out")
        elif self.state == self.SOLD:
            print("Please wait, we're already giving you a gumball")
```

```
Menna Yousri
def ejectQuarter(self):
    if self.state == self.HAS_QUARTER:
        print("Quarter returned")
        self.state = self.NO_QUARTER
    elif self.state == self.NO_QUARTER:
        print("You haven't inserted a quarter")
    elif self.state == self.SOLD:
        print("Sorry, you already turned the crank")
    elif self.state == self.SOLD_OUT:
        print("You can't eject, you haven't inserted a quarter yet")

Menna Yousri
def turnCrank(self):
    if self.state == self.SOLD:
        print("Turning twice doesn't get you another gumball!")
    elif self.state == self.NO_QUARTER:
        print("You turned but there's no quarter")
    elif self.state == self.SOLD_OUT:
        print("You turned, but there are no gumballs")
    elif self.state == self.HAS_QUARTER:
        print("You turned...")
        self.state = self.SOLD
        self.dispense()
```

```
def dispense(self):
    if self.state == self.SOLD:
        print("A gumball comes rolling out the slot")
        self.count -= 1
        if self.count == 0:
            print("Oops, out of gumballs!")
            self.state = self.SOLD_OUT
        else:
            self.state = self.NO_QUARTER
    elif self.state == self.NO_QUARTER:
        print("You need to pay first")
    elif self.state == self.SOLD_OUT:
        print("No gumball dispensed")
    elif self.state == self.HAS_QUARTER:
        print("No gumball dispensed")

Menna Yousri
def refill(self, count):
    self.count += count
    print("The gumball machine was just refilled; its new count is:", self.count)
    if self.state == self.SOLD_OUT:
        self.state = self.NO_QUARTER

Menna Yousri
def __str__(self):
    result = "Inventory: " + str(self.count) + " gumball"
    if self.count != 1:
        result += "s"
    result += "\n"
    result += "Machine is " + str(self.state) + "\n"
    return result
```

```python
if __name__ == "__main__":
    gumballMachine = GumballMachine(2)

    print(gumballMachine)

    gumballMachine.insertQuarter()
    gumballMachine.turnCrank()

    print(gumballMachine)

    gumballMachine.insertQuarter()
    gumballMachine.turnCrank()
    gumballMachine.insertQuarter()
    gumballMachine.turnCrank()

    gumballMachine.refill(5)
    gumballMachine.insertQuarter()
    gumballMachine.turnCrank()

    print(gumballMachine)
```

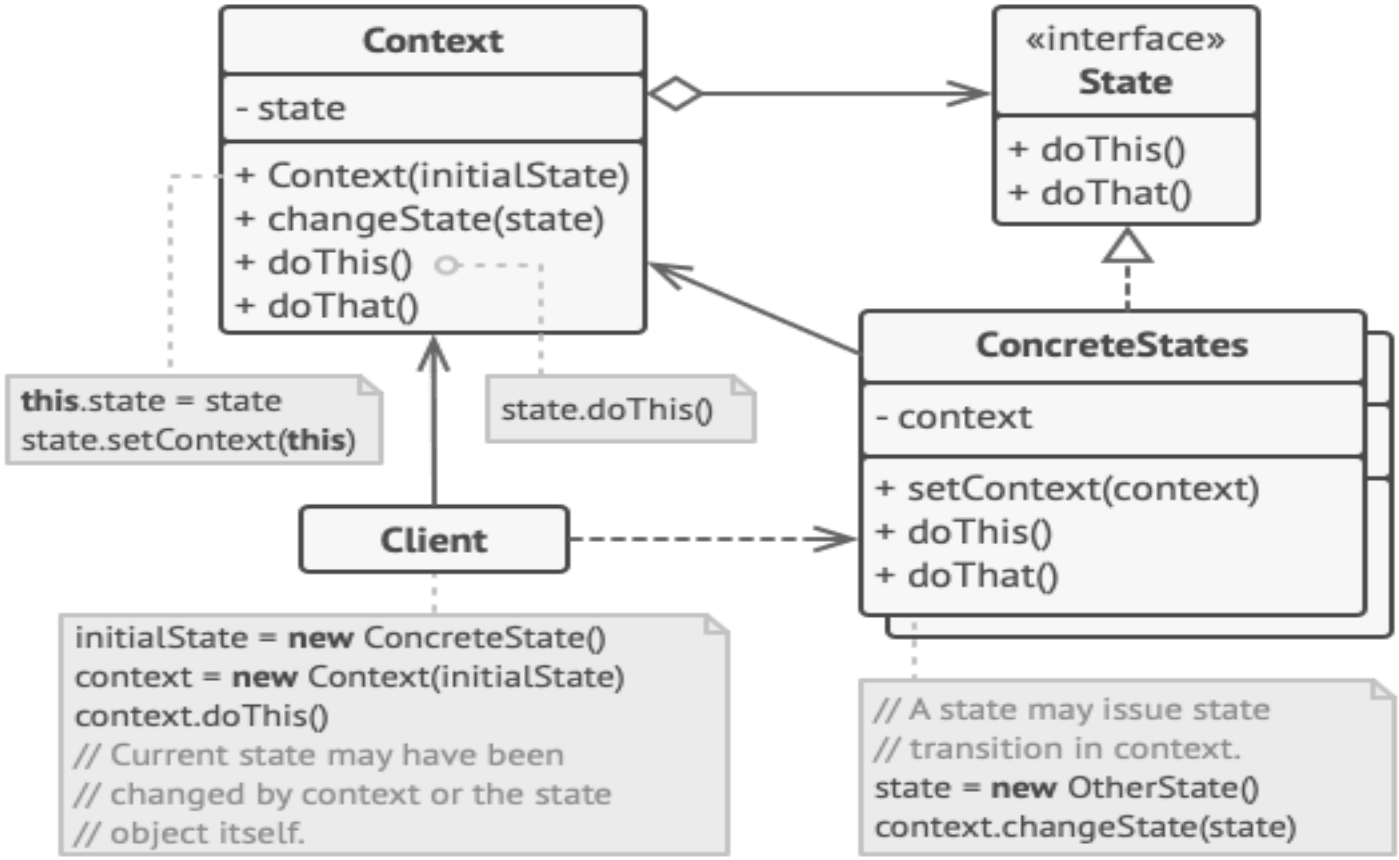
The output:

```
Inventory: 2 gumballs
Machine is 1

You inserted a quarter
You turned...
A gumball comes rolling out the slot
Inventory: 1 gumball
Machine is 1

You inserted a quarter
You turned...
A gumball comes rolling out the slot
Oops, out of gumballs!
You can't insert a quarter, the machine is sold out
You turned, but there are no gumballs
The gumball machine was just refilled; its new count is: 5
You inserted a quarter
You turned...
A gumball comes rolling out the slot
Inventory: 4 gumballs
Machine is 1
```

➢ **Solution**

The State pattern suggests that you create new classes for all possible states of an object and extract all state-specific behaviors into these classes.

Instead of implementing all behaviors on its own, the original object, called *context*, stores a reference to one of the state objects that represents its current state, and delegates all the state-related work to that object.




To transition the context into another state, replace the active state object with another object that represents that new state. This is possible only if all state classes follow the same interface and the context itself works with these objects through that interface.

This structure may look similar to the Strategy pattern, but there's one key difference. In the State pattern, the particular states may be aware of each other and initiate transitions from one state to another, whereas strategies almost never know about each other.

**The new design**

It looks like we've got a new plan: instead of maintaining our existing code, we're going to rework it to encapsulate state objects in their own classes and then delegate to the current state when an action occurs. We're following our design principles here, so we should end up with a design that is easier to maintain down the road. Here's how we're going to do it:

I. First, we're going to define a state interface that contains a method for every action in the Gumball Machine.

II. Then we're going to implement a state class for every state of the machine. These classes will be responsible for the behavior of the machine when it is in the corresponding state.

III. Finally, we're going to get rid of all of our conditional code and instead delegate to the state class to do the work for us.

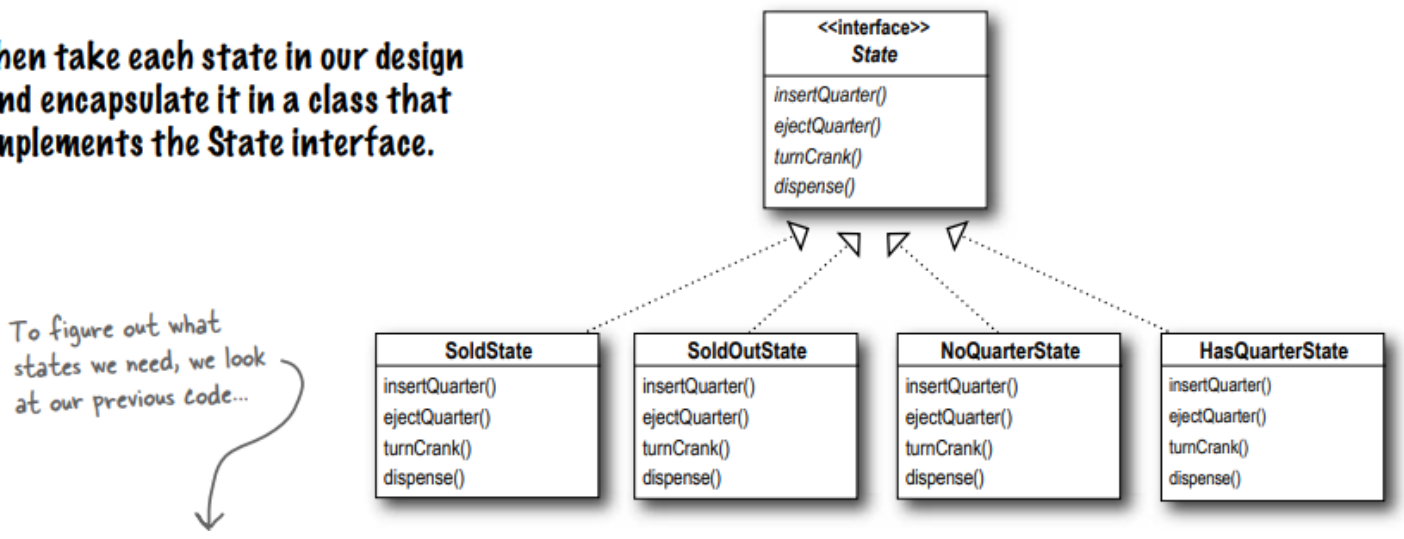
Not only are we following design principles, as you'll see, we're actually implementing the State Pattern.

## Defining the State interfaces and classes

**First let's create an interface for State, which all our states implement:**

Here's the interface for all states. The methods map directly to actions that could happen to the Gumball Machine (these are the same methods as in the previous code).

**Then take each state in our design and encapsulate it in a class that implements the State interface.**




To figure out what states we need, we look at our previous code...

```python
from abc import ABC, abstractmethod


Menna Yousri
class State(ABC):
    Menna Yousri
    @abstractmethod
    def insertQuarter(self):
        pass

    Menna Yousri
    @abstractmethod
    def ejectQuarter(self):
        pass

    Menna Yousri
    @abstractmethod
    def turnCrank(self):
        pass

    Menna Yousri
    @abstractmethod
    def dispense(self):
        pass

    Menna Yousri
    @abstractmethod
    def refill(self):
        pass
class NoQuarterState(State):
    Menna Yousri
    def __init__(self, gumballMachine):
        self.gumballMachine = gumballMachine

    Menna Yousri
    def insertQuarter(self):
        print("You inserted a quarter")
        self.gumballMachine.setState(self.gumballMachine.getHasQuarterState())

    Menna Yousri
    def ejectQuarter(self):
        print("You haven't inserted a quarter")

    Menna Yousri
    def turnCrank(self):
        print("You turned, but there's no quarter")

    Menna Yousri
    def dispense(self):
        print("You need to pay first")

    Menna Yousri
    def refill(self):
        pass

    Menna Yousri
    def __str__(self):
        return "waiting for quarter"
```

```python
class HasQuarterState(State):
    Menna Yousri
    def __init__(self, gumballMachine):
        self.gumballMachine = gumballMachine

    Menna Yousri
    def insertQuarter(self):
        print("You can't insert another quarter")

    Menna Yousri
    def ejectQuarter(self):
        print("Quarter returned")
        self.gumballMachine.setState(self.gumballMachine.getNoQuarterState())

    Menna Yousri
    def turnCrank(self):
        print("You turned...")
        self.gumballMachine.setState(self.gumballMachine.getSoldState())

    Menna Yousri
    def dispense(self):
        print("No gumball dispensed")

    Menna Yousri
    def refill(self):
        pass

    Menna Yousri
    def __str__(self):
        return "waiting for turn of crank"
class SoldState(State):
    Menna Yousri
    def __init__(self, gumballMachine):
        self.gumballMachine = gumballMachine

    Menna Yousri
    def insertQuarter(self):
        print("Please wait, we're already giving you a gumball")

    Menna Yousri
    def ejectQuarter(self):
        print("Sorry, you already turned the crank")

    Menna Yousri
    def turnCrank(self):
        print("Turning twice doesn't get you another gumball!")

    Menna Yousri
    def dispense(self):
        self.gumballMachine.releaseBall()
        if self.gumballMachine.getCount() > 0:
            self.gumballMachine.setState(self.gumballMachine.getNoQuarterState())
        else:
            print("Oops, out of gumballs!")
            self.gumballMachine.setState(self.gumballMachine.getSoldOutState())

    Menna Yousri
    def refill(self):
        pass

    Menna Yousri
    def __str__(self):
        return "dispensing a gumball"
```

```python
class GumballMachine:
    Menna Yousri
    def __init__(self, numberGumballs):
        self.soldOutState = SoldOutState(self)
        self.noQuarterState = NoQuarterState(self)
        self.hasQuarterState = HasQuarterState(self)
        self.soldState = SoldState(self)

        self.count = numberGumballs
        if numberGumballs > 0:
            self.state = self.noQuarterState
        else:
            self.state = self.soldOutState

    Menna Yousri
    def insertQuarter(self):
        self.state.insertQuarter()

    Menna Yousri
    def ejectQuarter(self):
        self.state.ejectQuarter()

    Menna Yousri
    def turnCrank(self):
        self.state.turnCrank()
        self.state.dispense()

    Menna Yousri
    def releaseBall(self):
        print("A gumball comes rolling out the slot...")
        if self.count != 0:
            self.count -= 1

    Menna Yousri
    def getCount(self):
        return self.count
    Menna Yousri
    def refill(self, count):
        self.count += count
        print("The gumball machine was just refilled; its new count is:", self.count)
        self.state.refill()

    Menna Yousri
    def setState(self, state):
        self.state = state

    Menna Yousri
    def getState(self):
        return self.state

    Menna Yousri
    def getSoldOutState(self):
        return self.soldOutState

    Menna Yousri
    def getNoQuarterState(self):
        return self.noQuarterState

    Menna Yousri
    def getHasQuarterState(self):
        return self.hasQuarterState

    Menna Yousri
    def getSoldState(self):
        return self.soldState

    Menna Yousri
    def __str__(self):
        result = "Inventory: " + str(self.count) + " gumball"
        if self.count != 1:
            result += "s"
        result += "\n"
        result += "Machine is " + str(self.state) + "\n"
        return result
```

```python
if __name__ == "__main__":
    gumballMachine = GumballMachine(2)

    print(gumballMachine)

    gumballMachine.insertQuarter()
    gumballMachine.turnCrank()

    print(gumballMachine)

    gumballMachine.insertQuarter()
    gumballMachine.turnCrank()
    gumballMachine.insertQuarter()
    gumballMachine.turnCrank()

    gumballMachine.refill(5)
    gumballMachine.insertQuarter()
    gumballMachine.turnCrank()

    print(gumballMachine)
```

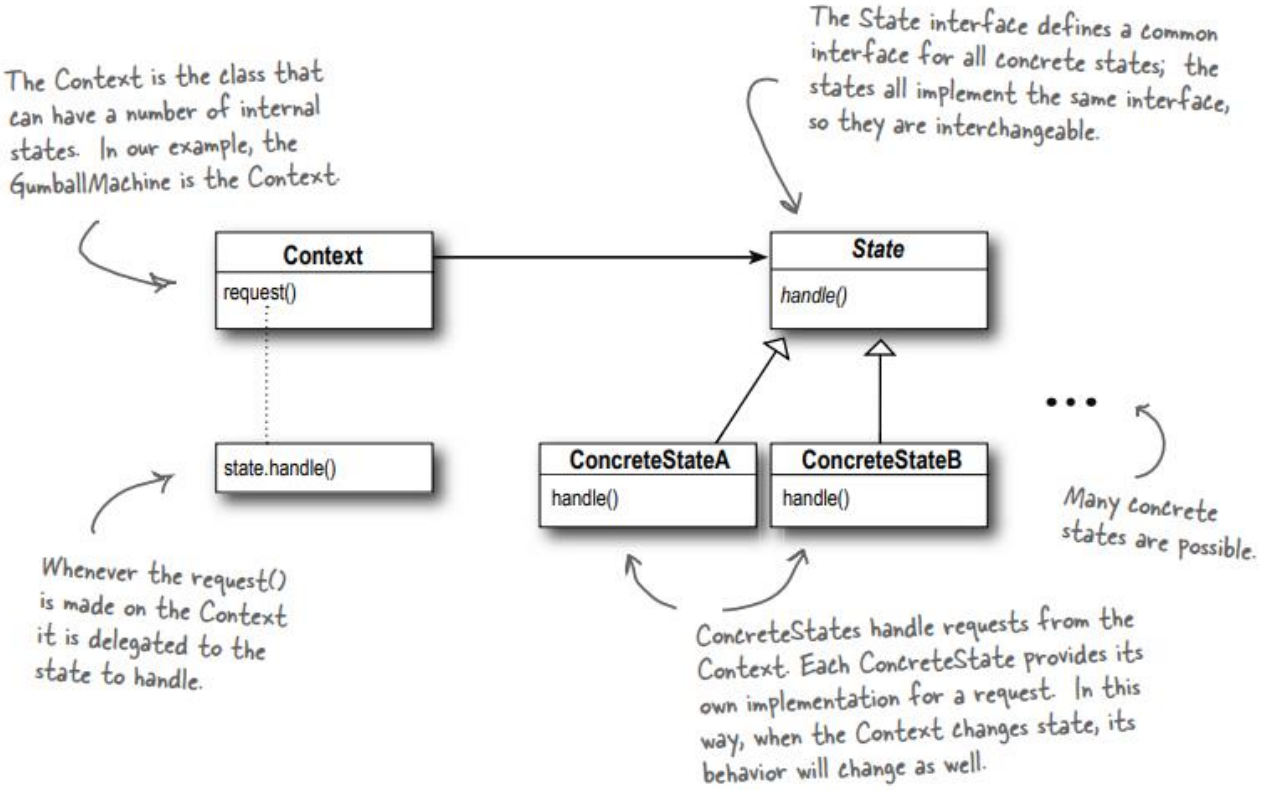
The output:

```
Inventory: 2 gumballs
Machine is waiting for quarter

You inserted a quarter
You turned...
A gumball comes rolling out the slot...
Inventory: 1 gumball
Machine is waiting for quarter

You inserted a quarter
You turned...
A gumball comes rolling out the slot...
Oops, out of gumballs!
You can't insert a quarter, the machine is sold out
You turned, but there are no gumballs
No gumball dispensed
The gumball machine was just refilled; its new count is: 5
You inserted a quarter
You turned...
A gumball comes rolling out the slot...
Inventory: 4 gumballs
Machine is waiting for quarter
```

**Let's take a look at what we've done so far...**

For starters, you now have a Gumball Machine implementation that is structurally quite different from your first version, and yet functionally it is exactly the same. By structurally changing the implementation you've:

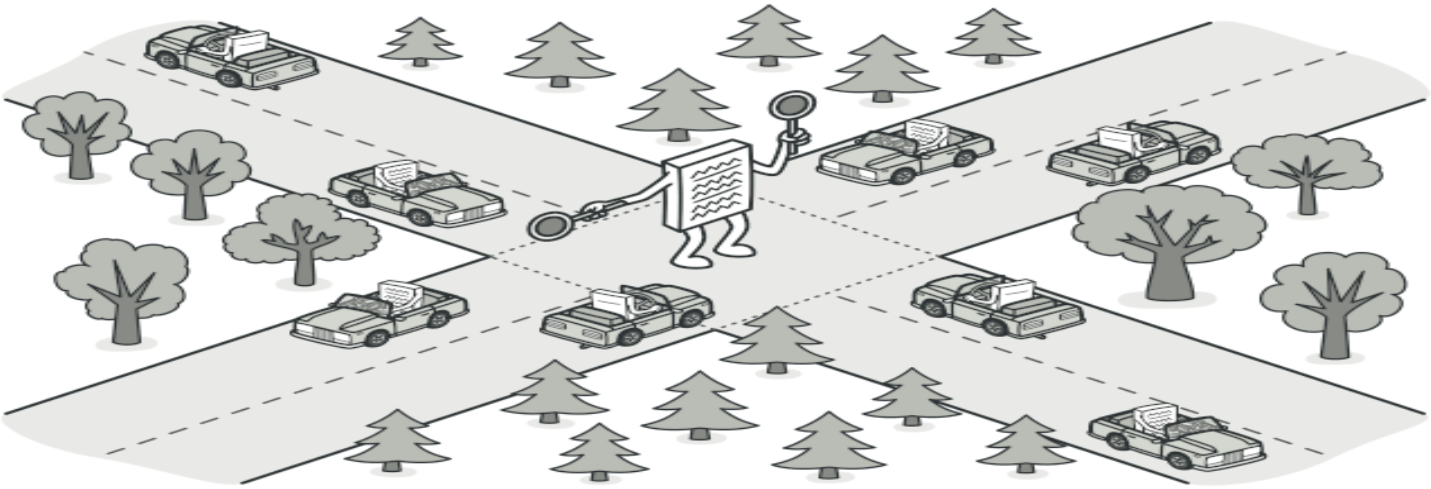
- Localized the behavior of each state into its own class.
- Removed all the troublesome if statements that would have been difficult to maintain.
- Closed each state for modification, and yet left the Gumball Machine open to extension by adding new state classes.
- Created a code base and class structure that maps much more closely to the Mighty Gumball diagram and is easier to read and understand.



## 7. Mediator

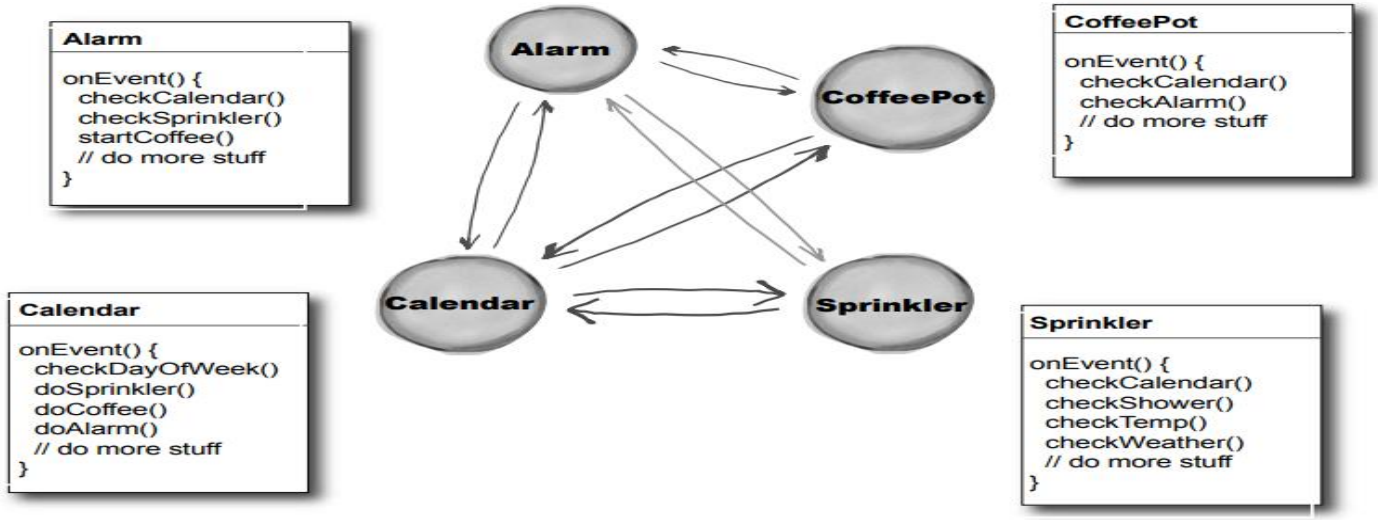
- **Intent**

**Mediator** is a behavioral design pattern that lets you reduce chaotic dependencies between objects. The pattern restricts direct communications between the objects and forces them to collaborate only via a mediator object.



- **Problem**

Bob has an auto-house, thanks to the good folks at HouseOfTheFuture. All of his appliances are designed to make his life easier. When Bob stops hitting the snooze button, his alarm clock tells the coffee maker to start brewing. Even though life is good for Bob, he and other clients are always asking for lots of new features: No coffee on the weekends...Turn off the sprinkler 15 minutes before a shower is scheduled...




HouseOfTheFuture's dilemma, it's getting really hard to keep track of which rules reside in which objects, and how the various objects should relate to each other.

```python
from datetime import datetime


class MyCalendar:
    def get_time(self):
        cal = datetime.now()
        day_of_week = cal.weekday()  # Monday is 0 and Sunday is 6
        return day_of_week


class Alarm:
    def __init__(self , my_calendar , my_coffee_machine):

        self.my_calendar = my_calendar
        self.my_coffee_machine = my_coffee_machine

    def snooze(self):
        day = self.my_calendar.get_time()
        if day != 5 and day != 6:  # Assuming Monday is 0 and Sunday is 6
            self.my_coffee_machine.start()

    def ring(self):
        print("RINGGG..")
class CoffeeMachine:
    def __init__(self , moving_robot , my_calendar):
        self.moving_robot = moving_robot
        self.my_calendar = my_calendar

    def start(self):
        print("Preparing Coffee")
        print("Finished Preparing Coffee")
        day = self.my_calendar.get_time()
        if day == 2:  # Assuming Wednesday is 2
            print("Adding Sugar!")
        self.moving_robot.transport()


class MovingRobot:
    def __init__(self , alarm=None , smart_window=None):
        if smart_window is None:
            pass
        else:
            self.alarm = alarm
            self.smart_window= smart_window

    def transport(self):
        print("Robot Transporting!")
        print("Reached Destination!")
        self.alarm.ring()
        if self.smart_window:
            self.smart_window.open()
```

```
class SmartWindow:
    def open(self):
        print("Opening Window")

    def close(self):
        print("Closing Window")


class MediatorTester:
    @staticmethod
    def main():
        c = MyCalendar()
        window = SmartWindow()
        mr = MovingRobot()
        cm = CoffeeMachine(mr , c)
        alarm = Alarm(c , cm)
        mr.alarm = alarm
        mr.smart_window= window
        alarm.snooze()


if __name__ == "__main__":
    MediatorTester.main()
```

The output:

```
Preparing Coffee
Finished Preparing Coffee
Robot Transporting!
Reached Destination!
RINGGG..
Opening Window
```

➢ **Solution**

The Mediator pattern suggests that you should cease all direct communication between the components which you want to make independent of each other. Instead, these components must collaborate indirectly, by calling a special mediator object that redirects the calls to appropriate components. As a result, the components depend only on a single mediator class instead of being coupled to dozens of their colleagues.

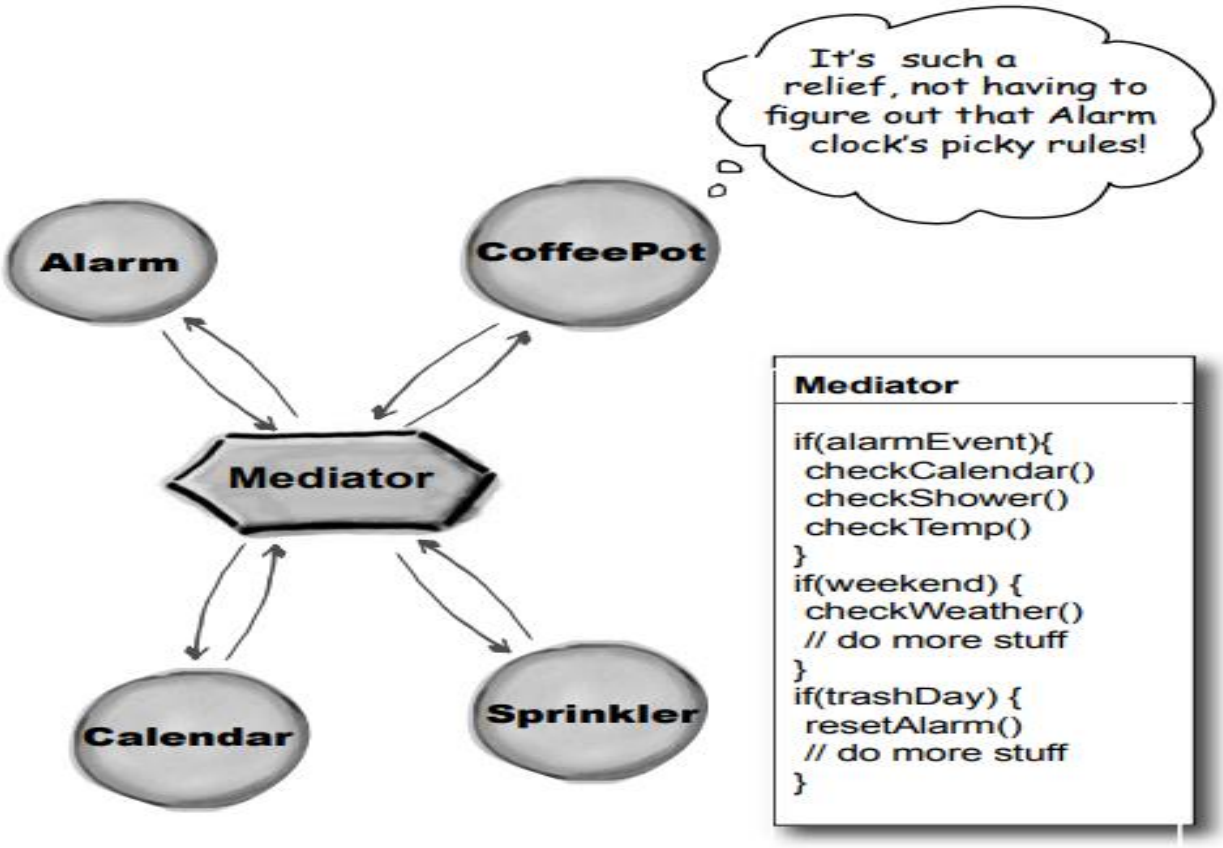
Mediator in action...

With a Mediator added to the system, all of the appliance objects can be greatly simplified.

- They tell the Mediator when their state changes.
- They respond to requests from the Mediator.

Before adding the Mediator, all of the appliance objects needed to know about each other... they were all tightly coupled. With the Mediator in place, the appliance objects are all completely decoupled from each other.

Mediator contains all of the control logic for the entire system. When an existing appliance needs a new rule, or a new appliance is added to the system, you'll know that all of the necessary logic will be added to the Mediator.



```python
from datetime import datetime
Menna Yousri
class MyCalendar:
    Menna Yousri
    def get_time(self):
        cal = datetime.now()
        day_of_week = cal.weekday()  # Monday is 0 and Sunday is 6
        return day_of_week

Menna Yousri
class Mediator:
    Menna Yousri
    def __init__(self, my_calendar, alarm, coffee_machine, moving_robot, smart_window=None):
        self.my_calendar = my_calendar
        self.alarm = alarm
        self.coffee_machine = coffee_machine
        self.moving_robot = moving_robot
        self.smart_window = smart_window

    Menna Yousri
    def trigger_alarm(self):
        self.alarm.ring()

    Menna Yousri
    def prepare_coffee(self):
        self.coffee_machine.start()

    Menna Yousri
    def transport_robot(self):
        self.moving_robot.transport()

    Menna Yousri
    def open_window(self):
        if self.smart_window:
            self.smart_window.open()
Menna Yousri
class Alarm:
    Menna Yousri
    def __init__(self, mediator):
        self.mediator = mediator

    Menna Yousri
    def snooze(self):
        day = self.mediator.my_calendar.get_time()
        if day != 5 and day != 6:  # Assuming Monday is 0 and Sunday is 6
            self.mediator.prepare_coffee()


    Menna Yousri
    def ring(self):
        print("RINGGG..")
```

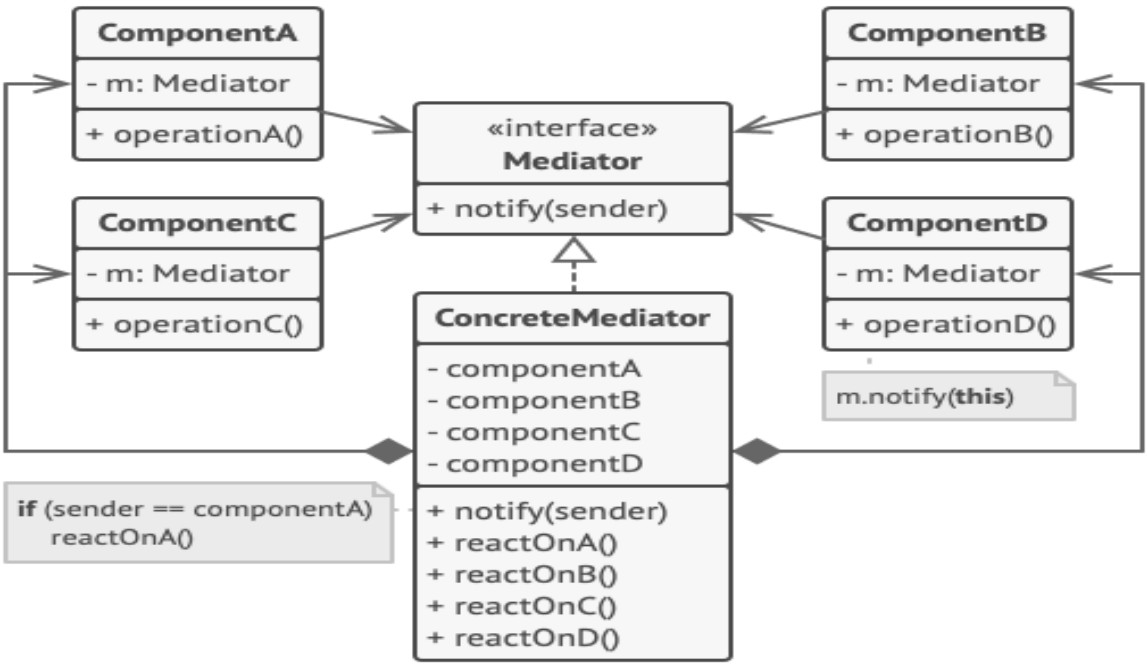
```python
class CoffeeMachine:
    Menna Yousri
    def __init__(self, mediator):
        self.mediator = mediator

    Menna Yousri
    def start(self):
        print("Preparing Coffee")
        print("Finished Preparing Coffee")
        day = self.mediator.my_calendar.get_time()
        if day == 2:  # Assuming Wednesday is 2
            print("Adding Sugar!")
        self.mediator.transport_robot()


Menna Yousri
class MovingRobot:
    Menna Yousri
    def __init__(self, mediator):
        self.mediator = mediator

    Menna Yousri
    def transport(self):
        print("Robot Transporting!")
        print("Reached Destination!")
        self.mediator.trigger_alarm()
        self.mediator.open_window()
Menna Yousri
class MovingRobot:
    Menna Yousri
    def __init__(self, mediator):
        self.mediator = mediator

    Menna Yousri
    def transport(self):
        print("Robot Transporting!")
        print("Reached Destination!")
        self.mediator.trigger_alarm()
        self.mediator.open_window()


Menna Yousri
class SmartWindow:
    Menna Yousri
    def open(self):
        print("Opening Window")

    Menna Yousri
    def close(self):
        print("Closing Window")
```

```
Menna Yousri
class MediatorTester:
    Menna Yousri
    @staticmethod
    def main():
        c = MyCalendar()
        window = SmartWindow()
        alarm = Alarm(c)
        mr = MovingRobot(c)
        cm = CoffeeMachine(c)
        mediator = Mediator(c, alarm, cm, mr, window)
        alarm.mediator = mediator
        mr.mediator = mediator
        cm.mediator = mediator
        alarm.snooze()


if __name__ == "__main__":
    MediatorTester.main()
```

The output:

```
Preparing Coffee
Finished Preparing Coffee
Robot Transporting!
Reached Destination!
RINGGG..
Opening Window
```

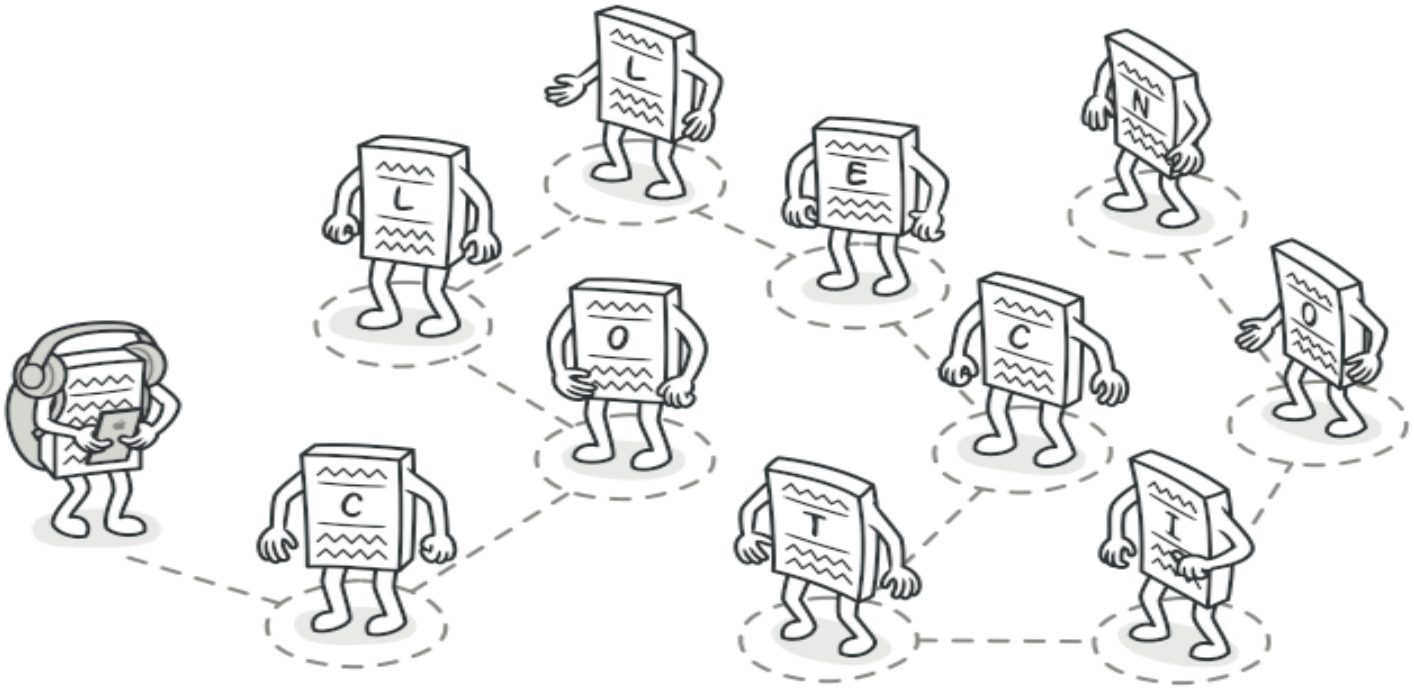
- **Structure**



## 8. Iterator

- **Intent**

  **Iterator** is a behavioral design pattern that lets you traverse elements of a collection without exposing its underlying representation (list, stack, tree, etc.).




- **Problem**

Breaking News: Objectville Diner and Objectville Pancake House Merge. That's great news! Now we can get those delicious pancake breakfasts at the Pancake House and those yummy lunches at the Diner all in one place. But there seems to be a slight problem...

The problem with having two different menu representations.

- The Waitress needs to know how each menu represents its internal collection of menu items; this violates encapsulation.
- We have duplicate code: the printMenu() method needs two separate loops to iterate over the two different kinds of menus. And if we added a third menu, we'd have yet another loop.

```python
class PancakeHouseMenu:
    Menna Yousri
    def __init__(self , menu):
        self.menu = menu

    Menna Yousri
    def get_menu(self):
        return self.menu


Menna Yousri
class DinerMenu:
    Menna Yousri
    def __init__(self , menu):
        self.menu = menu

    Menna Yousri
    def get_menu(self):
        return self.menu
```

```python
class Waitress:
    Menna Yousri
    def __init__(self , restaurant1 , restaurant2):
        self.restaurant1 = restaurant1
        self.restaurant2 = restaurant2

    Menna Yousri
    def print_menus(self):
        print("Menu of Pan Restaurant:")
        for dish in self.restaurant1.get_menu():
            print("-" , dish)

        print("\nMenu of Diner Restaurant:")
        for index in self.restaurant2.get_menu():
            print("-",self.restaurant2.get_menu().get(index))


# Example usage
pan_menu = [ "Spaghetti" , "Pizza" , "Salad" , "Soup" ]
diner_menu = {0: "Burger" , 1: "Sandwich" , 2: "Fries" , 3: "Milkshake"}

pan_restaurant = PancakeHouseMenu(pan_menu)
diner_restaurant = DinerMenu(diner_menu)

waitress = Waitress(pan_restaurant , diner_restaurant)
waitress.print_menus()
```

The output:

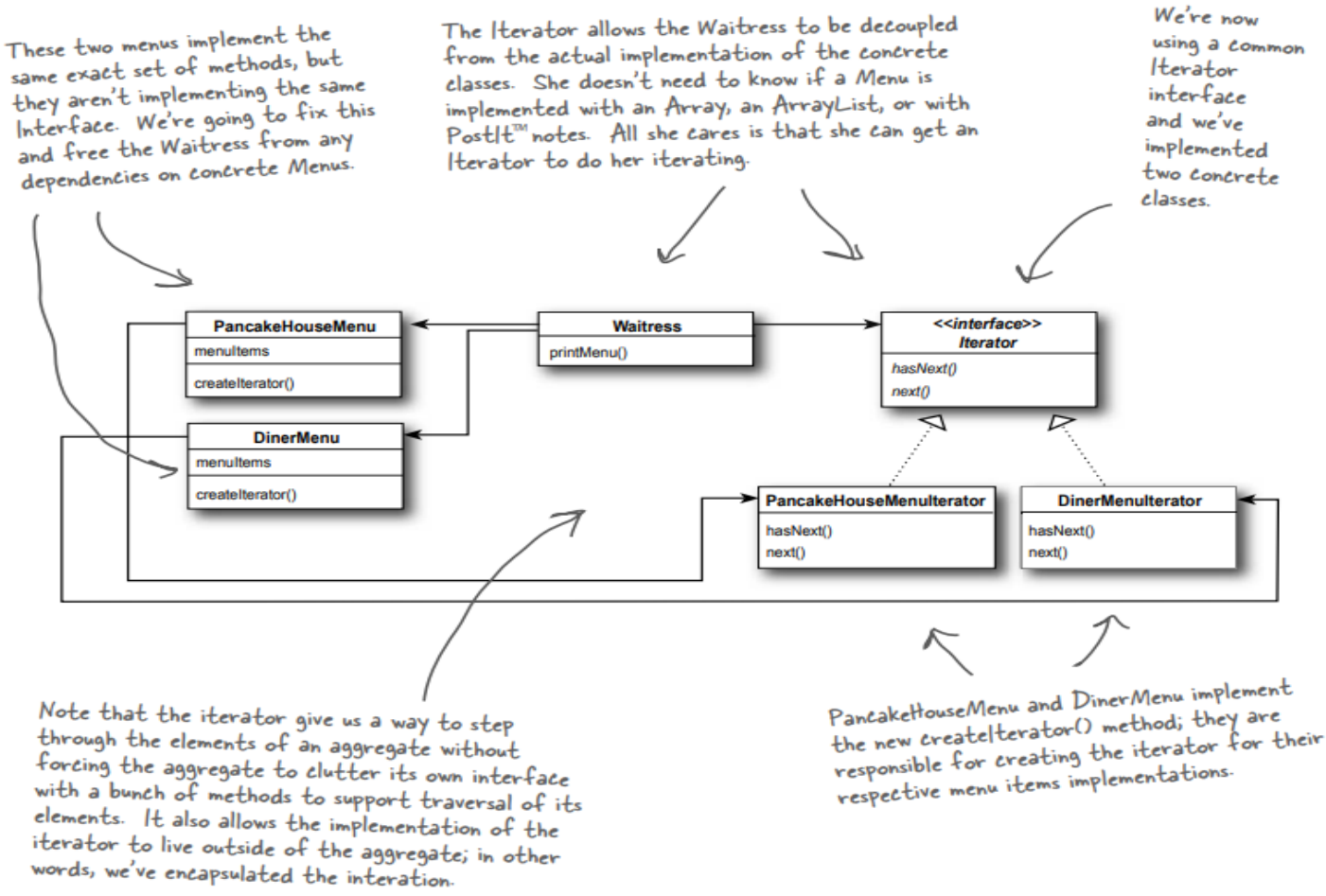
```
Menu of Pan Restaurant:
- Spaghetti
- Pizza
- Salad
- Soup

Menu of Diner Restaurant:
- Burger
- Sandwich
- Fries
- Milkshake
```

➢ **Solution**

Can we encapsulate the iteration?

We need to encapsulate what varies. It's obvious what is changing here: the iteration caused by different collections of objects being returned from the menus. But can we encapsulate this?

The main idea of the Iterator pattern is to extract the traversal behavior of a collection into a separate object called an iterator.



```python
from abc import ABC , abstractmethod

class Iterator(ABC):
    @abstractmethod
    def has_next(self):
        pass
    @abstractmethod
    def next(self):
        pass


class PancakeHouseMenu:
    def __init__(self, menu):
        self.menu_items = menu

    def create_iterator(self):
        return PancakeHouseMenuIterator(self.menu_items)
```

```python
class PancakeHouseMenuIterator(Iterator):
    def __init__(self, menu_items):
        self.menu_items = menu_items
        self.position = 0

    def has_next(self):
        return self.position < len(self.menu_items)

    def next(self):
        if self.has_next():
            item = self.menu_items[self.position]
            self.position += 1
            return item
        else:
            raise StopIteration


class DinerMenu:
    def __init__(self, menu):
        self.menu_items = menu

    def create_iterator(self):
        return DinerMenuIterator(self.menu_items)
```

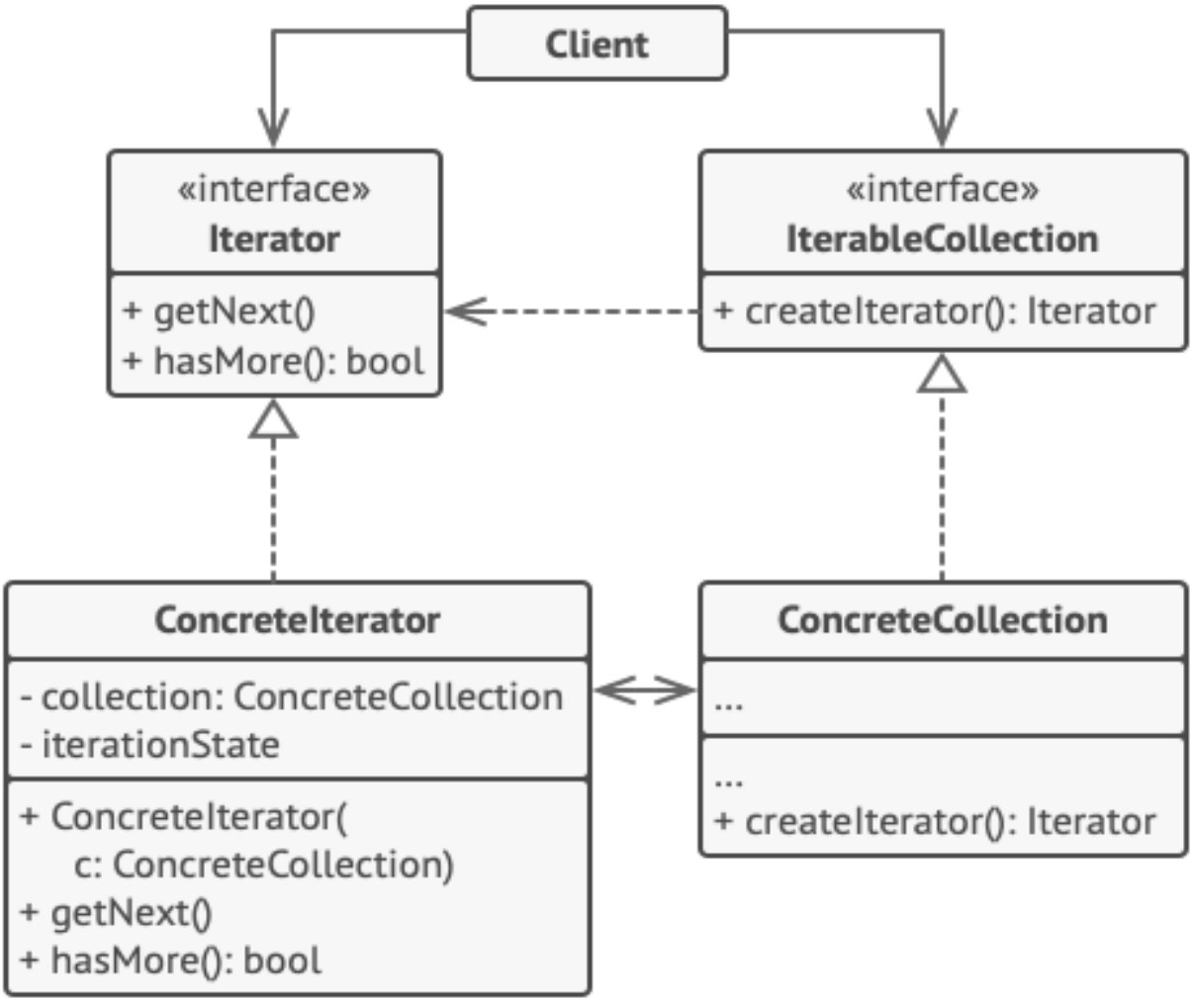
```python
Menna Yousri
class DinerMenuIterator(Iterator):
    Menna Yousri
    def __init__(self, menu_items):
        self.menu_items = menu_items
        self.position = 0

    Menna Yousri
    def has_next(self):
        return self.position < len(self.menu_items)

    Menna Yousri
    def next(self):
        if self.has_next():
            item = self.menu_items[self.position]
            self.position += 1
            return item
        else:
            raise StopIteration
```

```python
Menna Yousri
class Waitress:
    Menna Yousri
    def __init__(self, restaurant1, restaurant2):
        self.restaurant1 = restaurant1
        self.restaurant2 = restaurant2

    Menna Yousri
    def print_menus(self):
        print("Menu of Pan Restaurant:")
        pan_iterator = self.restaurant1.create_iterator()
        while pan_iterator.has_next():
            print("-", pan_iterator.next())

        print("\nMenu of Diner Restaurant:")
        diner_iterator = self.restaurant2.create_iterator()
        while diner_iterator.has_next():
            print("-", diner_iterator.next())


# Example usage
pan_menu = ["Spaghetti", "Pizza", "Salad", "Soup"]
diner_menu = {0: "Burger", 1: "Sandwich", 2: "Fries", 3: "Milkshake"}

pan_restaurant = PancakeHouseMenu(pan_menu)
diner_restaurant = DinerMenu(diner_menu)

waitress = Waitress(pan_restaurant, diner_restaurant)
waitress.print_menus()
```

The output:

```
Menu of Pan Restaurant:
- Spaghetti
- Pizza
- Salad
- Soup

Menu of Diner Restaurant:
- Burger
- Sandwich
- Fries
- Milkshake
```

- **Structure**

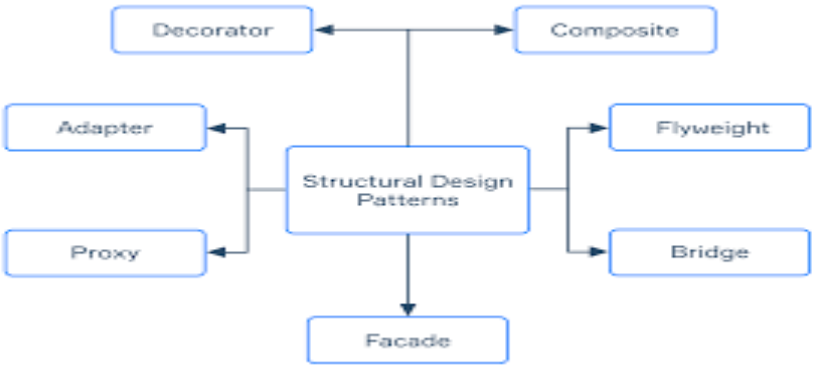


# Structural Design Patterns

## Structural Design Patterns:-

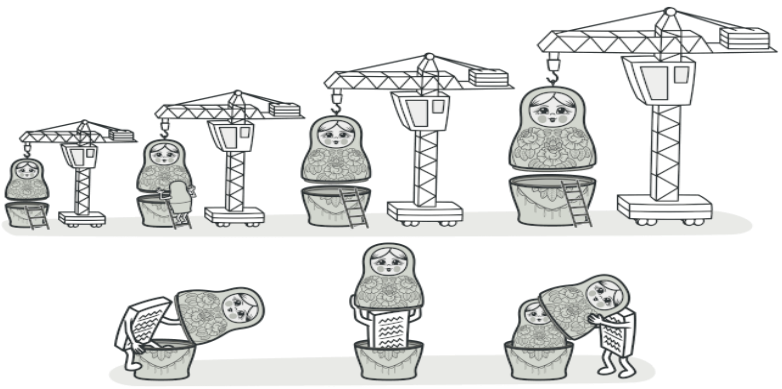
explain how to assemble objects and classes into larger structures, while keeping these structures flexible and efficient and are concerned with how classes and objects are composed to form larger structures.

These patterns help ensure that if one part of a system changes, the entire system does not need to do so. They focus on simplifying the structure of the code and making it easier to understand




## ➢ Decorator Pattern:-

- **Also known as:** Wrapper
- **Decorator** is a structural design pattern that lets you attach new behaviors to objects by placing these objects inside special wrapper objects that contain the behaviors.



## - When we use it:-

- **Use the Decorator pattern when you need to be able to assign extra behaviors to objects at runtime without breaking the code that uses these objects.**
- **Use the pattern when it's awkward or not possible to extend an object's behavior using inheritance.**

## - Advantages :-

- ✓ You can extend an object's behavior without making a new subclass.
- ✓ You can add or remove responsibilities from an object at runtime.
- ✓ You can combine several behaviors by wrapping an object into multiple decorators.
- ✓ *Single Responsibility Principle*. You can divide a monolithic class that implements many possible variants of behavior into several smaller classes.

## - Disadvantage:-

I. Removing a specific wrapper from the stack of wrappers is challenging.
II. Ensuring that a decorator's behavior does not depend on its position in the stack is difficult.
III. The initial setup code for layers can be complex and messy.

## - Implementation:-

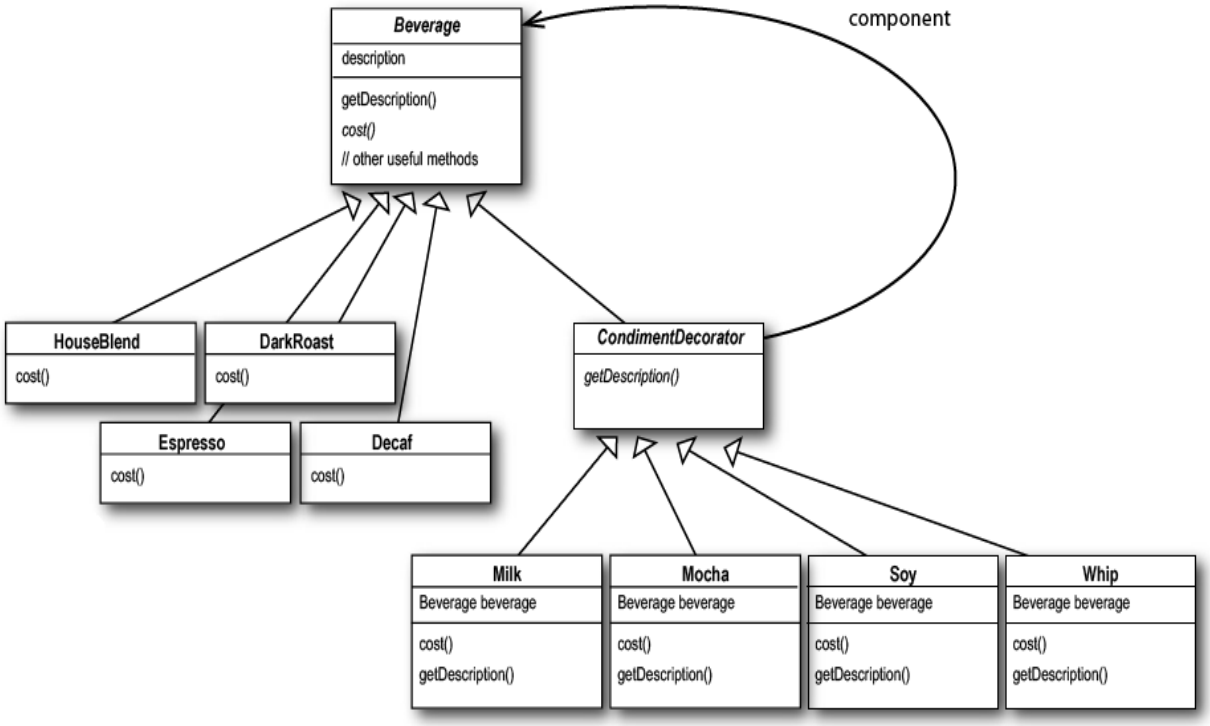
```
from abc import ABC, abstractmethod

# Component interface
class Coffee(ABC):
    @abstractmethod
    def cost(self) -> float:
        pass

    @abstractmethod
    def description(self) -> str:
        pass

# Concrete component
class SimpleCoffee(Coffee):
    def cost(self) -> float:
```

```
        return 1.0

    def description(self) -> str:
        return "Simple coffee"

# Decorator
class CoffeeDecorator(Coffee):
    def __init__(self, coffee: Coffee):
        self._coffee = coffee

    def cost(self) -> float:
        return self._coffee.cost()

    def description(self) -> str:
        return self._coffee.description()

# Concrete decorators
class Milk(CoffeeDecorator):
    def cost(self) -> float:
        return self._coffee.cost() + 0.5

    def description(self) -> str:
        return self._coffee.description() + ", milk"

class Sugar(CoffeeDecorator):
    def cost(self) -> float:
        return self._coffee.cost() + 0.2

    def description(self) -> str:
        return self._coffee.description() + ", sugar"

# Client code
def main():
    coffee = SimpleCoffee()
    print(f"Cost: ${coffee.cost()}, Description: {coffee.description()}")

    coffee_with_milk = Milk(coffee)
    print(f"Cost: ${coffee_with_milk.cost()}, Description:
{coffee_with_milk.description()}")

    coffee_with_milk_and_sugar = Sugar(coffee_with_milk)
    print(f"Cost: ${coffee_with_milk_and_sugar.cost()}, Description:
{coffee_with_milk_and_sugar.description()}")

if __name__ == "__main__":
```

```
main()
```

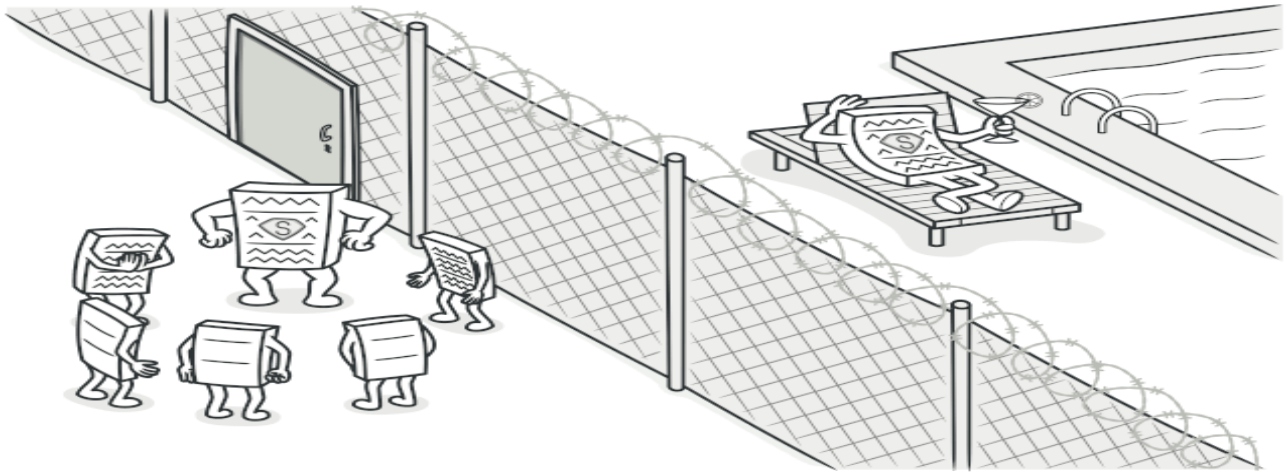
- **Output:-**

**Cost: $1.0, Description: Simple coffee**
**Cost: $1.5, Description: Simple coffee, milk**
**Cost: $1.7, Description: Simple coffee, milk, sugar**




## ➢ Proxy Pattern:-

- **Proxy** is a structural design pattern that lets you provide a substitute or placeholder for another object. A proxy controls access to the original object, allowing you to perform something either before or after the request gets through to the original object.



## - When we use it:-

- **Lazy initialization (virtual proxy).**
- **Access control (protection proxy).**
- **Local execution of a remote service (remote proxy).**
- **Logging requests (logging proxy).**
- **Caching request results (caching proxy).**

## - Advantages :-

- ✓ You can control the service object without clients knowing about it.
- ✓ You can manage the lifecycle of the service object when clients don't care about it.
- ✓ The proxy works even if the service object isn't ready or is not available.
- ✓ *Open/Closed Principle*. You can introduce new proxies without changing the service or clients.

## - Disadvantage:-

I. The code may become more complicated since you need to introduce a lot of new classes.
II. The response from the service might get delayed.

## - Implementation:-

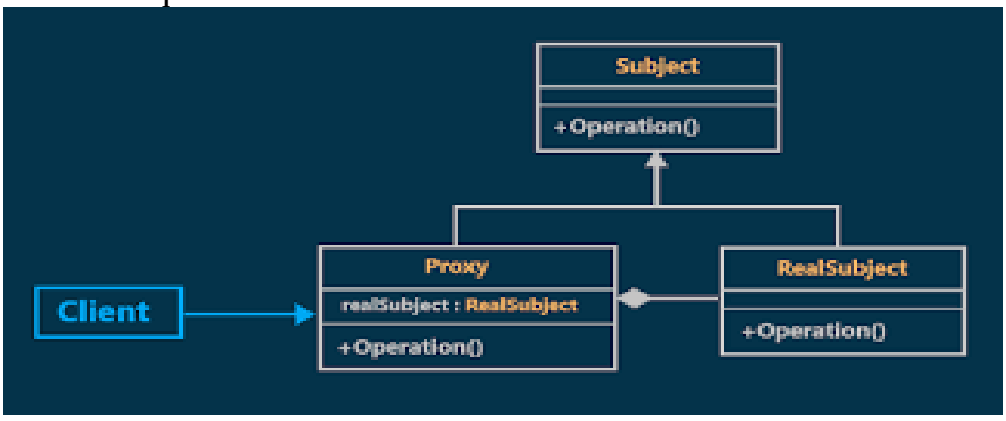
```
from abc import ABC, abstractmethod

# Subject interface
class Subject(ABC):
    """
    Subject is an interface that both RealSubject and Proxy implement.
```

```
    It defines a request method that must be implemented by both classes.
    """
    @abstractmethod
    def request(self) -> None:
        pass

# RealSubject class
class RealSubject(Subject):
    """
    RealSubject is a class that contains the actual implementation of the
request method.
    This class performs the core business logic.
    """
    def request(self) -> None:
        print("RealSubject: Handling request.")

# Proxy class
class Proxy(Subject):
    """
    Proxy is a class that acts as a proxy for RealSubject.
    It has a reference to a RealSubject object and controls access to it.
    The request method in Proxy first checks access,
    then delegates the request to the RealSubject if access is allowed.
    """
    def __init__(self, real_subject: RealSubject) -> None:
        self._real_subject = real_subject

    def request(self) -> None:
        """
        The request method of Proxy checks access before delegating the request
to the RealSubject
        and logs the access time.
        """
        if self.check_access():
            self._real_subject.request()
            self.log_access()

    def check_access(self) -> bool:
        """
        Check access before allowing the request to be forwarded to the real
subject.
        """
        print("Proxy: Checking access prior to firing a real request.")
        return True
```

```
    def log_access(self) -> None:
        """
        Log the time of request.
        """
        print("Proxy: Logging the time of request.", end="")

# Client code
def client_code(subject: Subject) -> None:
    """
    The client code demonstrates how the client can work with both RealSubject
    and Proxy objects interchangeably through the Subject interface.
    """
    subject.request()

if __name__ == "__main__":
    print("Client: Executing the client code with a real subject:")
    real_subject = RealSubject()
    client_code(real_subject)

    print("")

    print("Client: Executing the same client code with a proxy:")
    proxy = Proxy(real_subject)
    client_code(proxy)
-
```

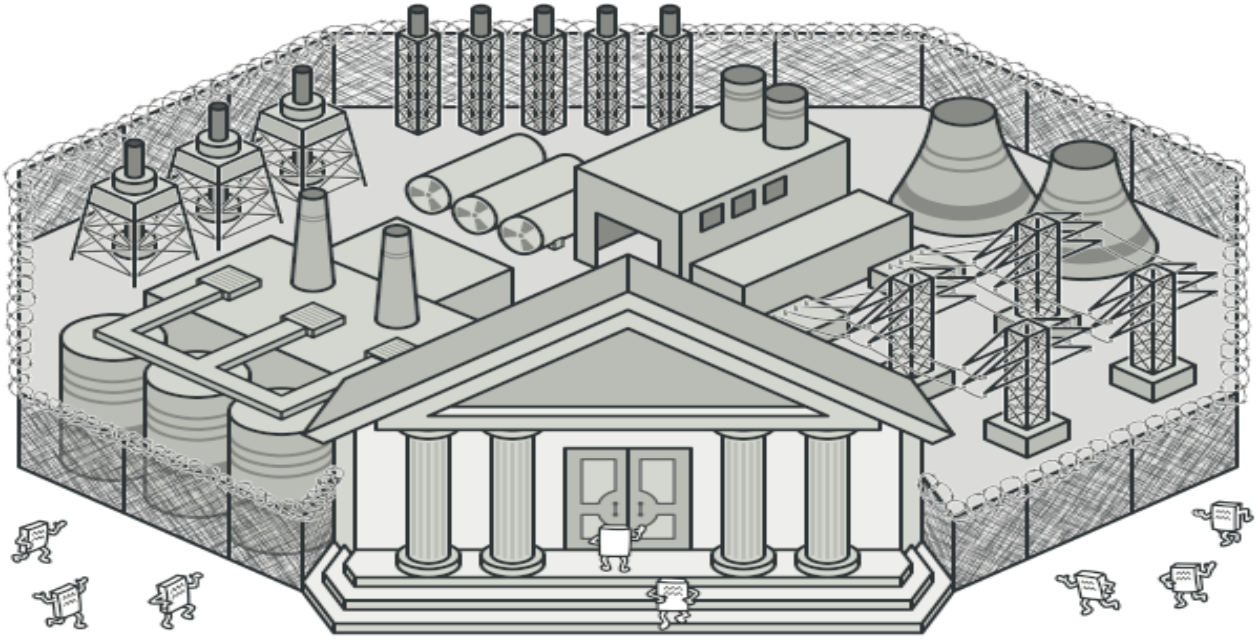
- **Output**:-

Client: Executing the client code with a real subject:
RealSubject: Handling request.
Client: Executing the same client code with a proxy:
Proxy: Checking access prior to firing a real request.
RealSubject: Handling request.
Proxy: Logging the time of request.



## ➢ Facade Pattern:-

**Facade** is a structural design pattern that provides a simplified interface to a library, a framework, or any other complex set of classes.



## - When we use it:-

- **Use the Facade pattern when you need to have a limited but straightforward interface to a complex subsystem.**
- **Use the Facade when you want to structure a subsystem into layers**

## - Advantages :-

✓ You can isolate your code from the complexity of a subsystem.

## - Disadvantage:-

I. A facade can become **a god object** coupled to all classes of an app.

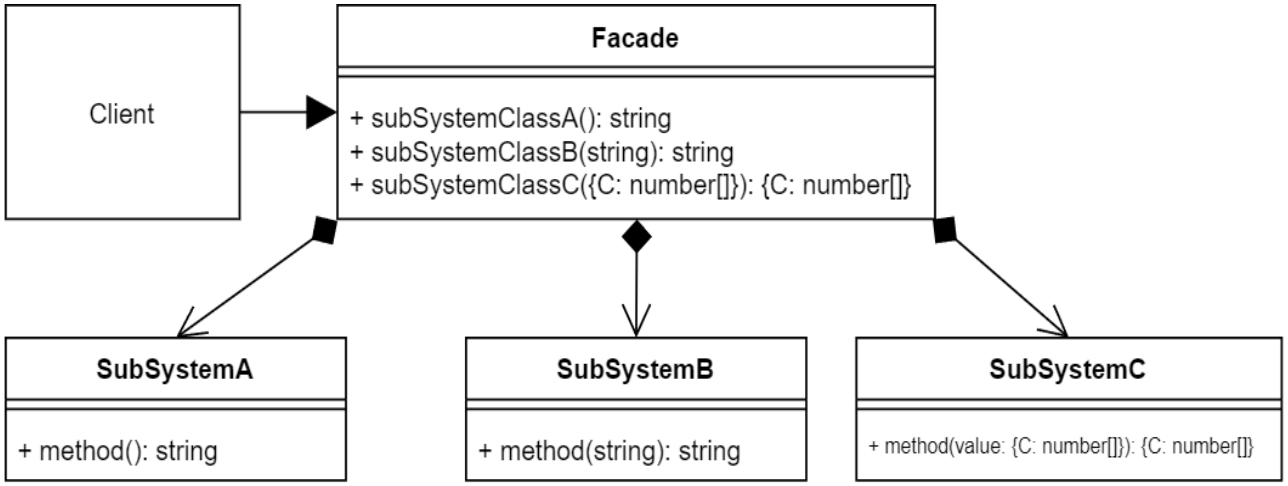
- **Implementation:-**

```
# Subsystems
class Engine:
    def __init__(self):
        self.speed = 0

    def start(self):
        print("Engine started")
        self.speed = 100

    def stop(self):
        print("Engine stopped")
        self.speed = 0

class Lights:
    def turn_on(self):
        print("Lights on")

    def turn_off(self):
        print("Lights off")

# Facade
class CarFacade:
    def __init__(self):
        self.engine = Engine()
        self.lights = Lights()

    def start_car(self):
        self.lights.turn_on()
        self.engine.start()

    def stop_car(self):
        self.engine.stop()
        self.lights.turn_off()

# Client code
def main():
    car = CarFacade()
    car.start_car()
    # Do something
    car.stop_car()
if __name__ == "__main__":
```

```
main()
```

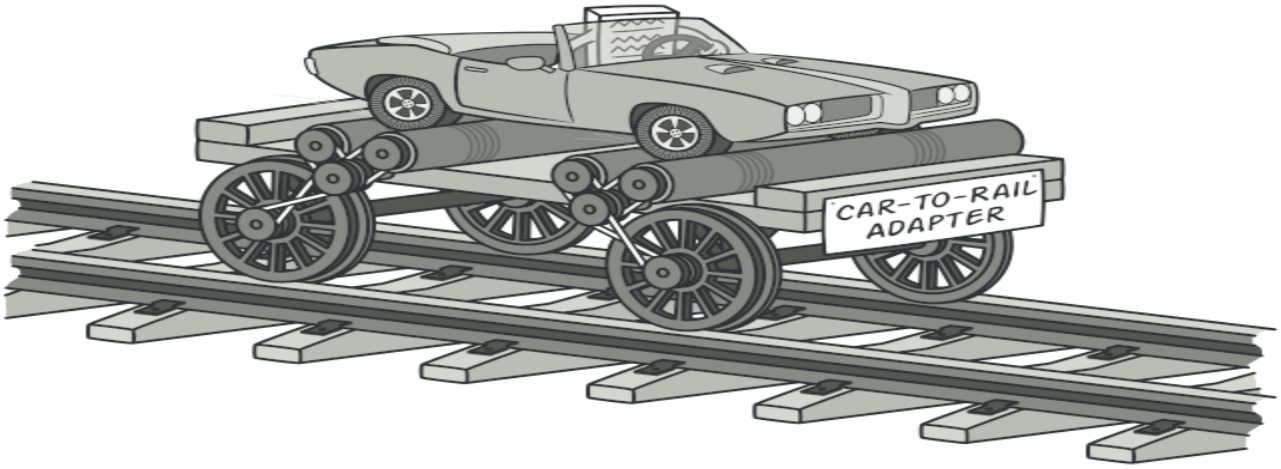
- **Output**:-

Lights on
Engine started
Engine stopped
Lights off




## ➢ Adapter Pattern:-

**Adapter** is a structural design pattern that allows objects with incompatible interfaces to collaborate.



## - When we use it:-

- **Use the Adapter class when you want to use some existing class, but its interface isn't compatible with the rest of your code.**
- **Use the pattern when you want to reuse several existing subclasses that lack some common functionality that can't be added to the superclass.**

## - Advantages :-

- ✓ *Single Responsibility Principle*. You can separate the interface or data conversion code from the primary business logic of the program.
- ✓ *Open/Closed Principle*. You can introduce new types of adapters into the program without breaking the existing client code, as long as they work with the adapters through the client interface.

## - Disadvantage:-

II. The overall complexity of the code increases because you need to introduce a set of new interfaces and classes. Sometimes it's simpler just to change the service class so that it matches the rest of your code.

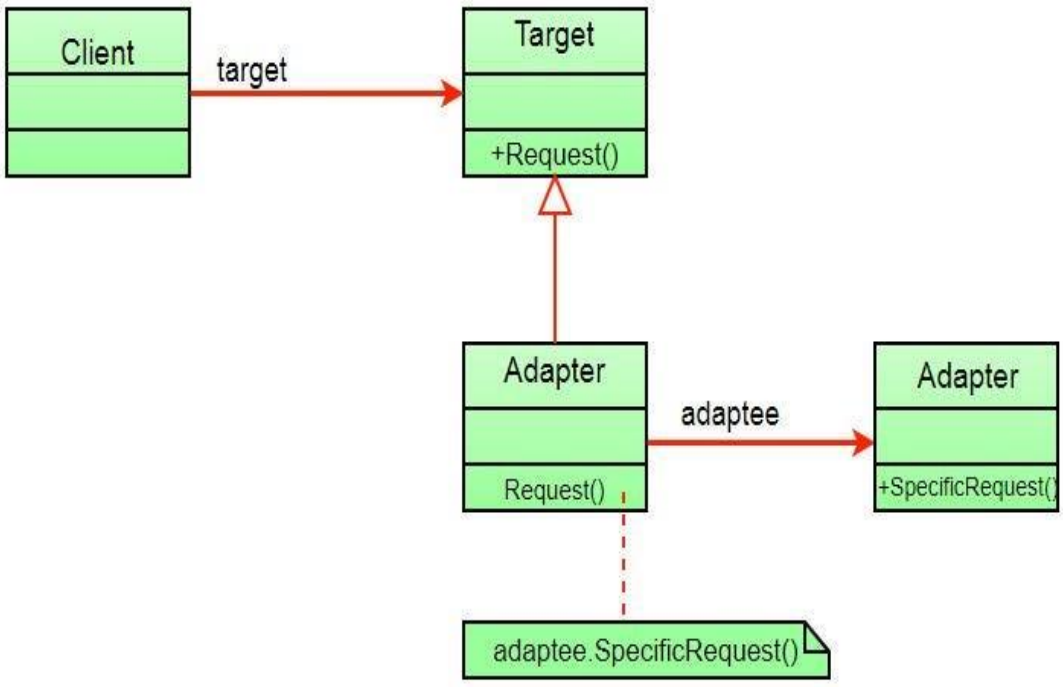
## - Implementation:-

```
- # Adaptee class (incompatible interface)
- class CelsiusTemperature:
-     def get_temperature(self):
-         return 25
-
- # Target interface (expected by the client)
- class FahrenheitTemperatureInterface:
-     def get_temperature(self):
-         pass
-
- # Adapter class which implements the FahrenheitTemperatureInterface and
  wraps an instance of CelsiusTemperature
- class Adapter(FahrenheitTemperatureInterface):
-     def __init__(self, celsius_temperature):
```

```
-           self.celsius_temperature = celsius_temperature
-
-       def get_temperature(self):
-           # Convert Celsius to Fahrenheit
-           return (self.celsius_temperature.get_temperature() * 9/5) + 32
-
-   # Client code
-   def display_temperature(temperature):
-       print(f"Temperature: {temperature.get_temperature()}°F")
-
-   if __name__ == "__main__":
-       celsius_temperature = CelsiusTemperature()
-       adapter = Adapter(celsius_temperature)
-       display_temperature(adapter)
```

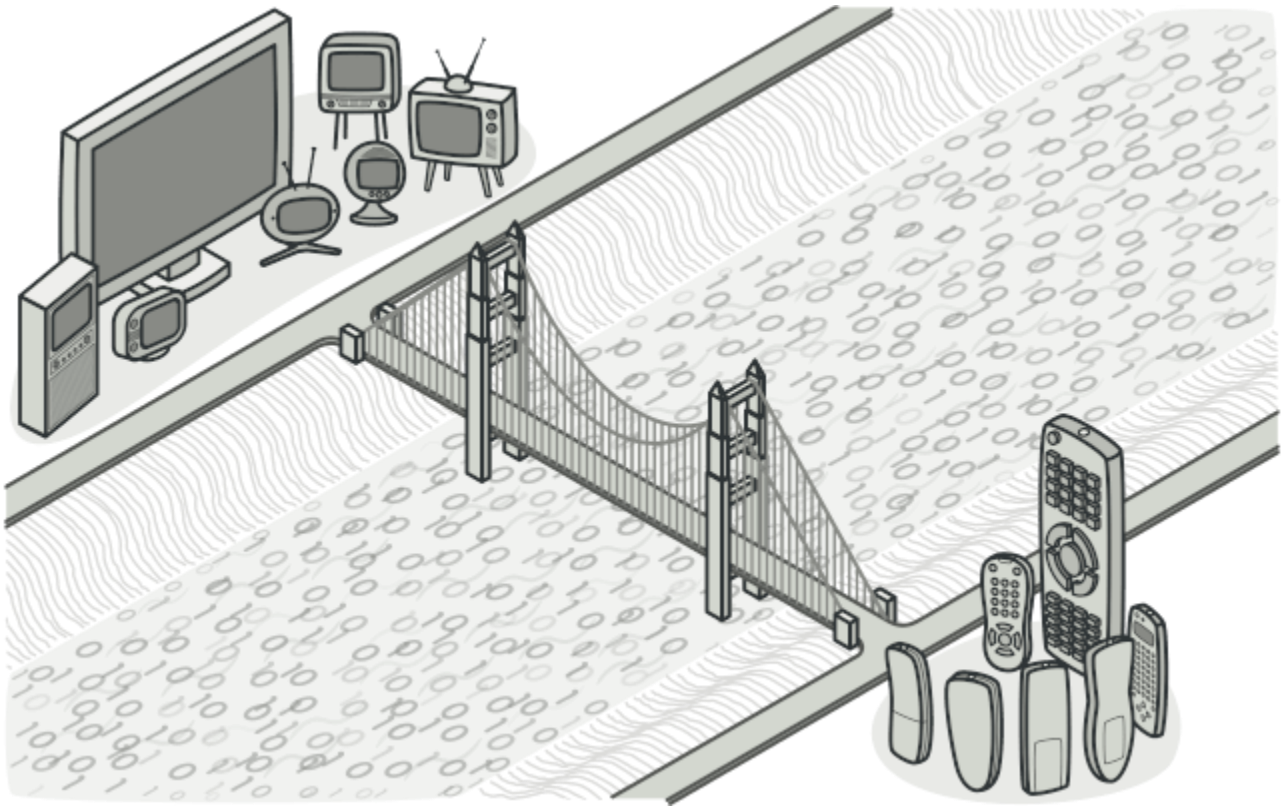
- **Output:-**

```
Temperature: 77.0°F
```



## ➢ Bridge Pattern:-

**Bridge** is a structural design pattern that lets you split a large class or a set of closely related classes into two separate hierarchies—abstraction and implementation—which can be developed independently of each other.



## - When we use it:-

- **Use the Bridge pattern when you want to divide and organize a monolithic class that has several variants of some functionality (for example, if the class can work with various database servers).**
- **Use the pattern when you need to extend a class in several (independent) dimensions.**
- **Use the Bridge if you need to be able to switch implementations at runtime.**

## - Advantages :-

- ✓ You can create platform-independent classes and apps.
- ✓ The client code works with high-level abstractions. It isn't exposed to the platform details.

- ✓ *Open/Closed Principle*. You can introduce new abstractions and implementations independently from each other.
- ✓ *Single Responsibility Principle*. You can focus on high-level logic in the abstraction and on platform details in the implementation.

## - Disadvantage:-

III. You might make the code more complicated by applying the pattern to a highly cohesive class.

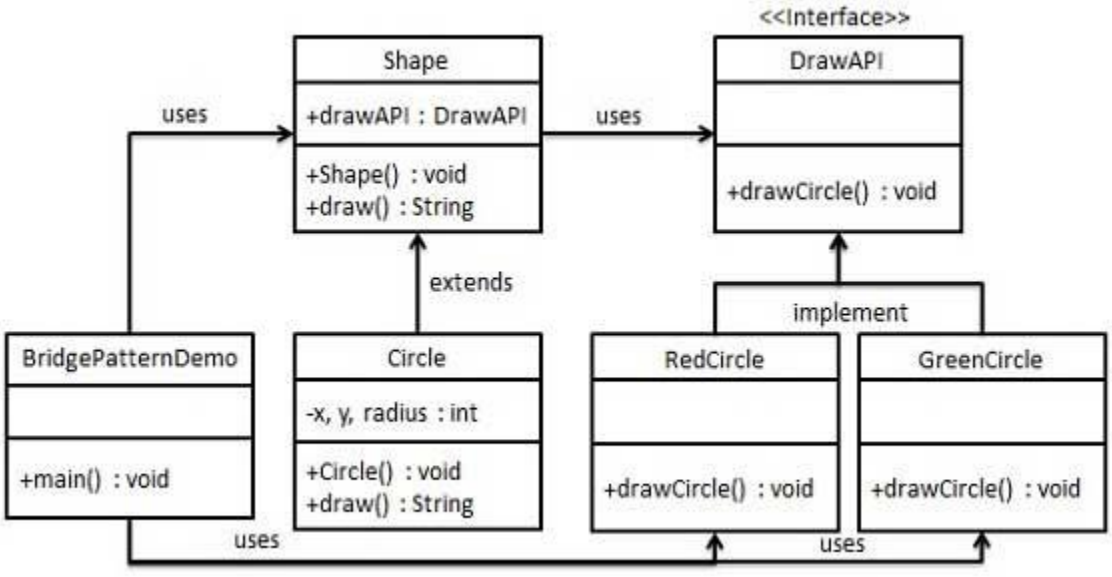
## - Implementation :-

```
from abc import ABC, abstractmethod

# Implementor interface
class DrawingAPI(ABC):
    @abstractmethod
    def draw_circle(self, x, y, radius):
        pass

# Concrete Implementor 1
class DrawingAPI1(DrawingAPI):
    def draw_circle(self, x, y, radius):
        print(f"API1.circle at {x}:{y} radius {radius}")

# Concrete Implementor 2
class DrawingAPI2(DrawingAPI):
    def draw_circle(self, x, y, radius):
        print(f"API2.circle at {x}:{y} radius {radius}")

# Abstraction
#Shape is the Abstraction class, which has a reference to a DrawingAPI object
class Shape(ABC):
    def __init__(self, drawing_api):
        self.drawing_api = drawing_api

    @abstractmethod
    def draw(self):
        pass

# Refined Abstraction
class CircleShape(Shape):
# that implements the draw method using the DrawingAPI object.
```

```
    def __init__(self, x, y, radius, drawing_api):
        super().__init__(drawing_api)
        self.x = x
        self.y = y
        self.radius = radius

    def draw(self):
        self.drawing_api.draw_circle(self.x, self.y, self.radius)

# Client code
#This demonstrates how the Bridge pattern allows us to switch implementations at
runtime and extend classes
if __name__ == "__main__":
    shapes = [
        CircleShape(1, 2, 3, DrawingAPI1()),
        CircleShape(5, 7, 11, DrawingAPI2())
    ]

    for shape in shapes:
        shape.draw()
```

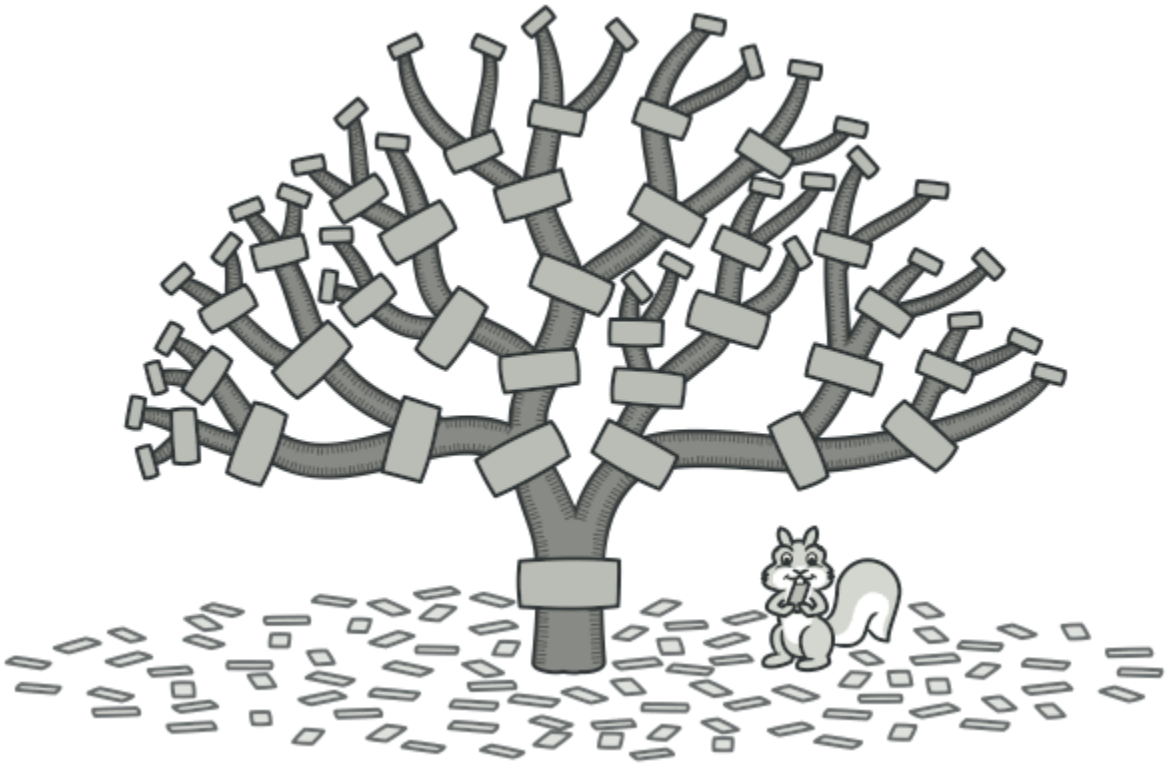
- **Output:-**

**-API1.circle at 1:2 radius 3**
**-API2.circle at 5:7 radius 11**



## ➢ Composite Pattern:-

**Composite** is a structural design pattern that lets you compose objects into tree structures and then work with these structures as if they were individual objects.



## - When we use it:-

- **Use the Composite pattern when you have to implement a tree-like object structure.**
- **Use the pattern when you want the client code to treat both simple and complex elements uniformly.**

## - Advantages :-

- ✓ You can work with complex tree structures more conveniently: use polymorphism and recursion to your advantage.
- ✓ *Open/Closed Principle*. You can introduce new element types into the app without breaking the existing code, which now works with the object tree

- ## Disadvantage:-

IV. It might be difficult to provide a common interface for classes whose functionality differs too much. In certain scenarios, you'd need to overgeneralize the component interface, making it harder to comprehend.

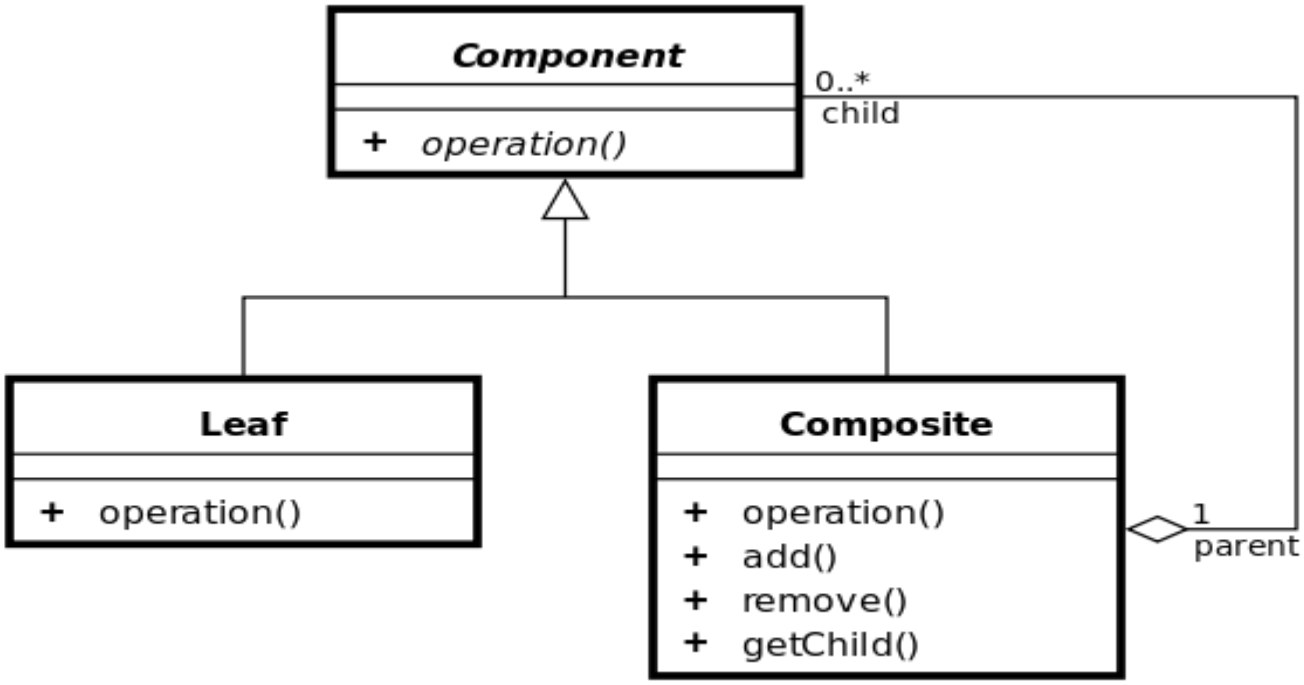
- ## Implementation :-

```python
  from abc import ABC, abstractmethod

# Component interface
class FileSystemComponent(ABC):
    @abstractmethod
    def list_contents(self):
        pass

# Leaf class
class File(FileSystemComponent):
    def __init__(self, name):
        self.name = name

    def list_contents(self):
        print(f"File: {self.name}")

# Composite class
class Directory(FileSystemComponent):
    def __init__(self, name):
        self.name = name
        self.contents = []

    def add(self, component):
        self.contents.append(component)

    def list_contents(self):
     #then recursively call list_contents on its contents.
        print(f"Directory: {self.name}")
        for component in self.contents:
            component.list_contents()

# Client code
if __name__ == "__main__":
    file1 = File("file1.txt")
    file2 = File("file2.txt")
```

```
file3 = File("file3.txt")

directory1 = Directory("Folder 1")
directory1.add(file1)
directory1.add(file2)

directory2 = Directory("Folder 2")
directory2.add(file3)

root = Directory("Root")
root.add(directory1)
root.add(directory2)

root.list_contents()
```

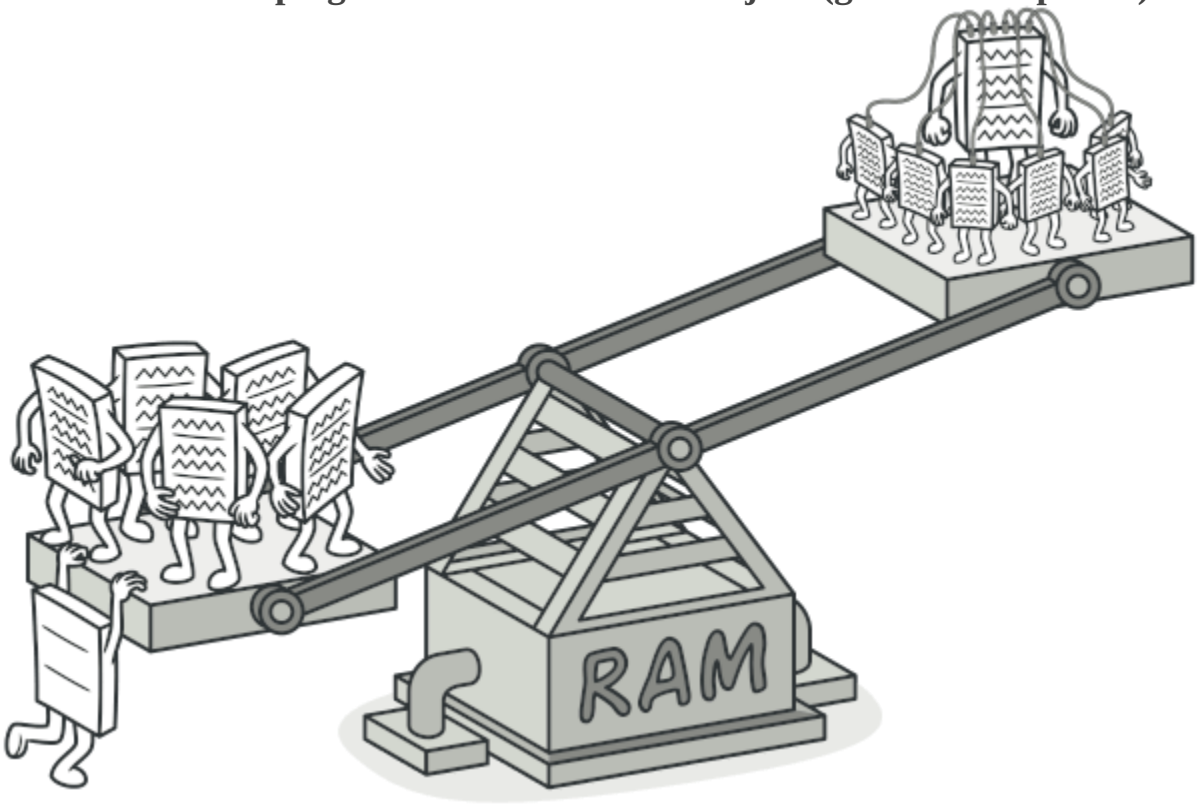
- **Output:-**

**Directory: Root**
**Directory: Folder 1**
**File: file1.txt**
**File: file2.txt**
**Directory: Folder 2**
**File: file3.txt**



## ➢ FlyweightPattern:-

Flyweight **is a structural design pattern that lets you fit more objects into the available amount of RAM by sharing common parts of state between multiple objects instead of keeping all of the data in each object. (game Development)**




## - When we use it:-

- **Use the Flyweight pattern only when your program must support a huge number of objects which barely fit into available RAM**

## - Advantages :-

- ✓ **You can save lots of RAM, assuming your program has tons of similar objects.**

- ## Disadvantage:-

V. You might be trading RAM over CPU cycles when some of the context data needs to be recalculated each time somebody calls a flyweight method.

VI. The code becomes much more complicated. New team members will always be wondering why the state of an entity was separated in such a way.

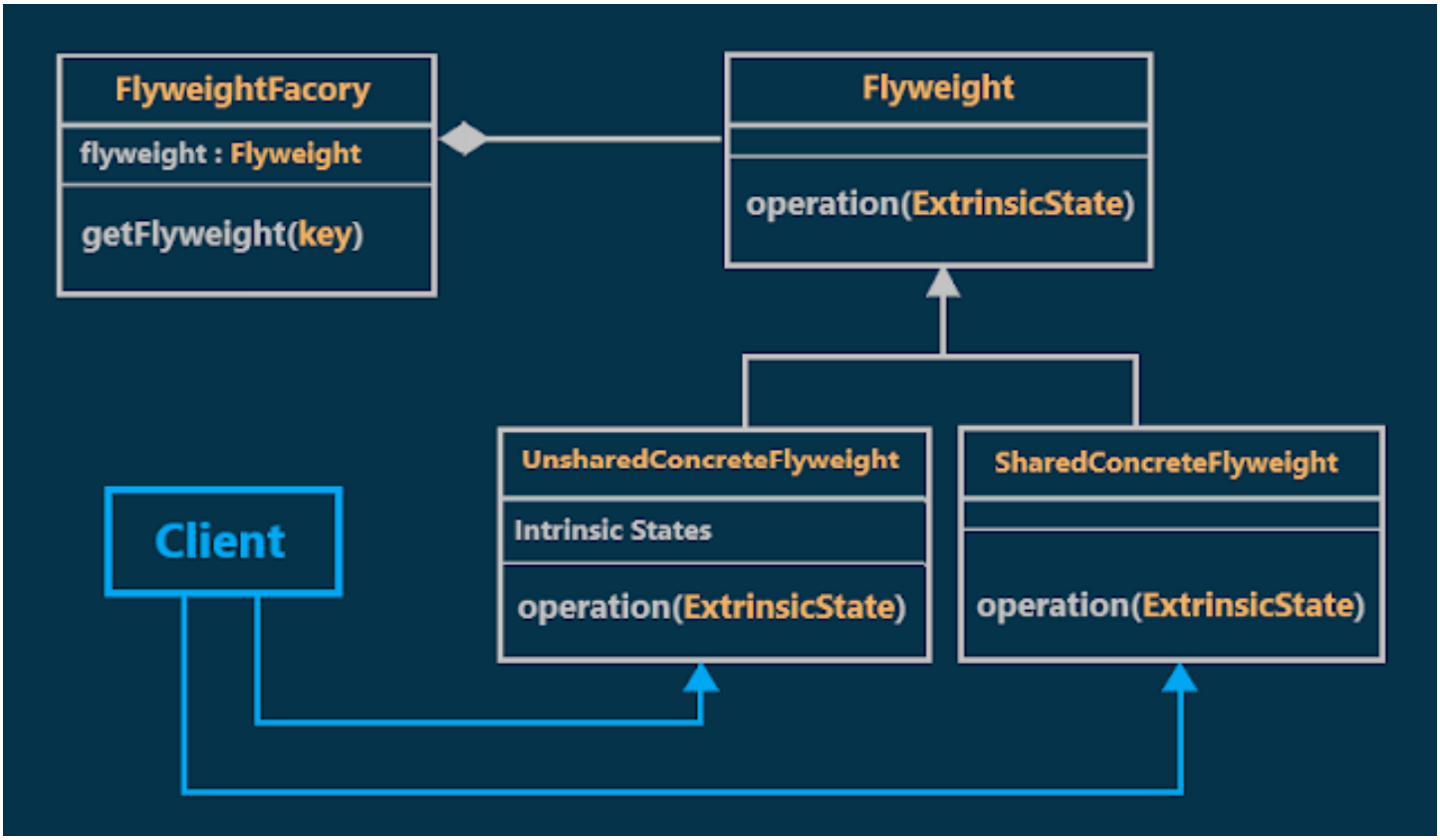
- ## Implementation :-

```
class Flyweight:
# Flyweight: This is the base class that defines the interface for flyweight objects. It has an __init__ method to initialize the shared state and an operation method that concrete flyweights will override to include unique state.
    def __init__(self, shared_state):
        self.shared_state = shared_state

    def operation(self, unique_state):
        pass

class ConcreteFlyweight(Flyweight):
# ConcreteFlyweight: This is a subclass of Flyweight that implements the operation method. It includes both the shared state (from the base class) and unique state passed to it.
    def operation(self, unique_state):
        print(f"Shared: {self.shared_state}, Unique: {unique_state}")

class FlyweightFactory:
    def __init__(self):
        self.flyweights = {}

    def get_flyweight(self, shared_state):
        if shared_state not in self.flyweights:
            self.flyweights[shared_state] = ConcreteFlyweight(shared_state)
        return self.flyweights[shared_state]

def main():
     # when we request a flyweight with the same shared state, we get the same instance
```

```
    factory = FlyweightFactory()

    flyweight1 = factory.get_flyweight("shared_state_1")
    flyweight1.operation("unique_state_1")

    flyweight2 = factory.get_flyweight("shared_state_1")
    flyweight2.operation("unique_state_2")

    flyweight3 = factory.get_flyweight("shared_state_2")
    flyweight3.operation("unique_state_3")

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

- **Output:-**

**Shared: shared_state_1, Unique: unique_state_1**
**Shared: shared_state_1, Unique: unique_state_2**
**Shared: shared_state_2, Unique: unique_state_3**



**Software architecture pattern**

# Architecture Style:-

The architectural style shows how do we organize our code, or how the system will look like from 10000 feet helicopter view to show the highest level of abstraction of our system design. Furthermore, when building the architectural style of our system we focus on layers and modules and how they are communicating with each other. There are different types of architectural styles, and moreover, we can mix them and produce a hybrid style that consists of a mix between two and even more architectural styles. Below is a list of architectural styles and examples for each category:

**Structure architectural styles:**such as layered, pipes and filters and component-based styles.

**Messaging styles:**such as Implicit invocation, asynchronous messaging and publish-subscribe style.

**Distributed systems:**such as service-oriented, peer to peer style, object request broker, and cloud computing styles.

**Shared memory styles:**such as role-based, blackboard, database-centric styles.

**Adaptive system styles:**such as microkernel style, reflection, domain-specific language styles.

- **Model-View-Controller (MVC):**

**MVC is a software architecture pattern commonly used for developing user interfaces. It divides an application into three interconnected components: the model, the view, and the controller.**

- **Model: Represents the data and business logic of the application. It interacts with the database and performs operations on the data.**
- **View: Represents the user interface. It displays the data from the model to the user and sends user commands to the controller.**
- **Controller: Acts as an intermediary between the model and the view. It receives user input from the view, processes it (possibly updating the model), and updates the view accordingly.**

**MVC helps in separating the concerns of an application, making it easier to maintain and test the code**

- **Event-Driven Architecture:**

**EDA is a software architecture pattern where the flow of the system is determined by events.**

**Events can be user actions, sensor outputs, messages from other systems, etc**

**In event-driven programming, the program typically has an event loop that waits for events to occur and then triggers the appropriate event handler to respond to the event. This programming style is commonly used in graphical user interfaces (GUIs) and network programming.**

## ➢ Microservice Architecture:

**Microservice architecture is an architectural style that structures an application as a collection of small, loosely coupled services. Each service is self-contained and implements a single business function.**

**Microservices communicate with each other over a network using lightweight protocols such as HTTP or message queues. This approach allows for easier development, deployment, and scaling, making the system more flexible and agile than traditional monolithic architectures.**

**.**

## ➢ Pipe and Filter Architectures:

**Pipe and filter architecture is a design pattern where data is processed through a series of components (filters) connected by pipes.**

**Each filter performs a specific transformation or processing on the data and passes it to the next filter through the pipe.**

**This architecture is useful for designing systems where data needs to undergo multiple processing steps in a structured and modular manner.**
**It promotes reusability and maintainability by allowing filters to be easily added, removed, or modified without affecting the overall system.**

## ➢ Model View Template (MVT):

**Used in: Django web framework (specifically in the context of Django's architecture).**

**Similar to: MVC.**

**Components:**

**Model: Represents the data structure (e.g., database schema).**

**View: Represents the presentation layer (e.g., HTML templates).**

**Template: Acts as a bridge between the model and the view, handling the presentation logic.**

**MVT is essentially Django's take on MVC. The main difference is that in MVT, the controller is replaced by the template engine, which manages the presentation logic and generates the final output (HTML).**

## ➢ Model View Presenter (MVP):

**Used in: GUI development, web development.**

**Similar to: MVC.**

**Components:**

**Model: Represents the data and business logic.**

**View: Represents the user interface.**

**Presenter: Acts as an intermediary that retrieves data from the model and formats it for the view.**

**MVP is like MVC but focuses more on separating concerns. The presenter manages most of the application logic, keeping the view passive.**

## ➢ Model View View Model (MVVM):

**Used in: GUI development, especially with frameworks like WPF (Windows Presentation Foundation) .**

**Similar to: MVC AND MVP**

**Components:**

**Model: Represents the data and business logic.**

**View: Represents the user interface.**

**View Model: Acts as a mediator between the view and the model, exposing data and handling user interactions.**

**MVVM is like MVP but adds a view model to manage the view's state and behavior. This makes testing and maintaining the view's logic easier.**