

Design Pattern



- ▶ Design patterns represent the best practices used by experienced object-oriented software developers.
- ▶ Design patterns are solutions to general problems that software developers faced during software development.
- ▶ These solutions were obtained by trial and error by numerous software developers over quite a substantial period of time.

What is Gang of Four (GOF)?

In 1994, four authors Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides published a book titled **Design Patterns - Elements of Reusable Object-Oriented Software** which initiated the concept of Design Pattern in Software development.

These authors are collectively known as Gang of Four (GOF). According to these authors design patterns are primarily based on the following principles of object oriented design.

- Program to an interface not an implementation
- Favor object composition over inheritance

Usage of Design Pattern



Design Patterns have two main usages in software development.

Common platform for developers

Design patterns provide a standard terminology and are specific to particular scenario. For example, a singleton design pattern signifies use of single object so all developers familiar with single design pattern will make use of single object and they can tell each other that program is following a singleton pattern.

Best Practices

Design patterns have been evolved over a long period of time and they provide best solutions to certain problems faced during software development. Learning these patterns helps inexperienced developers to learn software design in an easy and faster way.

Types of Design Patterns

As per the design pattern reference book **Design Patterns - Elements of Reusable Object-Oriented Software**, there are **23 design patterns** which can be classified in three categories:

1. **Creational**
2. **Structural**
3. **Behavioral**

Creational



- Talks about object creations.
- To make design more readable and less complexity.
- Talks about efficiently object creation.
- Example: Singleton Pattern

Structural



- Talks about how classes and objects can be composed.
- Parent-child relationship like inheritance used for structuring more than one classes.
- Deals with simplicity in identifying relationships.
- Example: Adapter Pattern

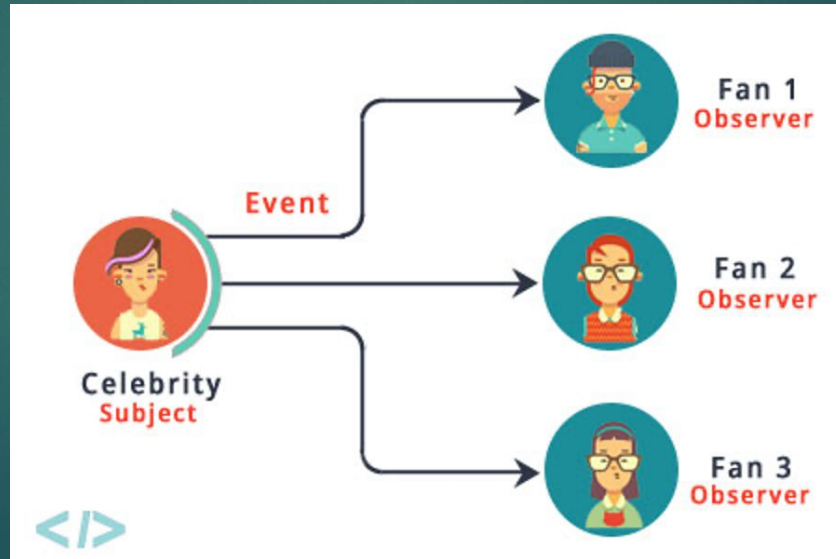
Behavioral



- Deals with interactions and responsibility of objects.
- Used for identifying and setting up common communication patterns among objects.
- Example: Observer Pattern

Observer Pattern

Intent: Define a one-to-many dependency between objects so that when one object change state, all its dependents are notified and updated automatically.



Colab File for Examples



[Design Patterns Example.ipynb](#)

Student-Teacher Example

Class Teacher:

```
def __init__(self, name):
    self.students=[]
    self.name=name

def attach_student(self, student):
    self.students.append(student)
    print(f"{student.name} has been attached to {self.name}")
```

We also need remove methods in both classes.

Class Student:

```
def __init__(self, name):
    self.teachers=[]
    self.name=name

def add_teachers(self, teacher):
    self.teachers.append(teacher)
    teacher.attach_student(self)
```

driver code

```
teacher_fzn = Teacher("FZN")
student_ryan = Student("Ryan")
student_ryan.add_teacher(teacher_fzn)
```

Fan-Celebrity Example

```
# Subject (Celebrity)
class Celebrity:
    def __init__(self):
        self._fans = []
        self._state = None
    def attach(self, fan):
        self._fans.append(fan)
    def detach(self, fan):
        if fan in self._fans:
            self._fans.remove(fan)
    def notify(self):
        for fan in self._fans:
            fan.update(self)
    def set_state(self, state):
        self._state = state
        self._notify()
    def get_state(self):
        return self._state
```

```
# Observer (Fan)
class Fan:
    def __init__(self):
        self._celebrities = []
    def update(self, celebrity):
        state = celebrity.get_state()
    def add_celebrity(self, celebrity):
        self._celebrities.append(celebrity)
        celebrity.attach(self)
    def remove_celebrity(self, celebrity):
        self._celebrities.remove(celebrity)
        celebrity.detach(self)
```

```
celebrity = Celebrity()
fan = Fan()
#fan starts following...
fan.add_celebrity(celebrity)
#celeb changes status and #notification is received by
fan
celebrity.set_state('New state')
#fan can stop following...
fan.remove_celebrity(celebrity)
```

Observer Pattern in Java

```
public class Celebrity{
    private List<Fan> fans = new ArrayList<Fan>();
    private int state;

    void attach(Fan f){
        fans.add(f);
    }
    void remove(Fan f){
        fans.remove(f);
    }
    void notify(){
        foreach( Fan f: fans)
            f.update(this);
    }
    void setState(int newState){
        state = newState;
        notify();
    }
    int getState(){
        return state;
    }
}
```

*Celebrity class may look like this. **Now add the Fan class***

Observer Pattern in Java

```
public class Celebrity{
    private List<Fan> fans=new
ArrayList<Fan>();
    private int state;

    void attach(Fan f){
        fans.add(f);
    }
    void remove(Fan f){
        fans.remove(f);
    }
    void notify(){
        foreach( Fan f: fans)
            f.update(this);
    }
    void setState(int newState){
        state = newState;
        notify();
    }
    int getState(){
        return state;
    }
}
```

```
public class Fan{
    private List<Celebrity>
celebrities=new ArrayList<Celebrity>();

    void update(Celebrity c){
        c.getState();
    }
    void addCelebrity(Celebrity c){
        celebrities.add(c);
        c.attach(this);
    }
    void removeCelebrity(Celebrity c){
        celebrities.remove(c);
        c.remove(this);
    }
}
```

From main method -
Fan f1 = new Fan();
Fan f2 = new Fan();

Celebrity c1 = new Celebrity();
Celebrity c2 = new Celebrity();

f1.addCelebrity(c1);
f1.addCelebrity(c2);
f2.addCelebrity(c1);

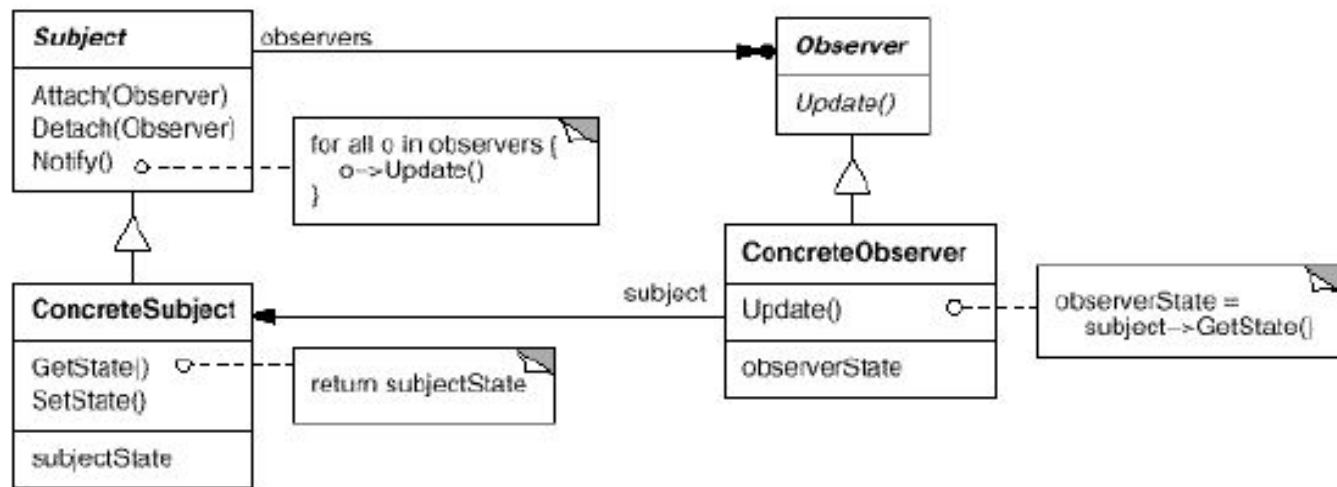
c1.setState(5); //new State
c2.setState(2); //newState

Participants

- **Subject:** knows its observers. Any number of Observer objects may observe a subject. It sends a notification to its observers when its state changes. (ex: Celebrity)
- **Observer:** defines an updating interface for objects that should be notified changes in a subject. (ex: Fans)
- **ConcreteSubject:** (ex: FilmCelebrity, FashionCelebrity)
- **ConcreteObserver** (ex: FilmFan, FashionFan)

Structure

15



Advanced Observer with Concrete subjects and observers

16

```
Public FilmCelebrity extends Celebrity{  
    private int state;  
    void setState(int newState){  
        state = newState;  
        notify();  
    }  
  
    int getState(){  
        return state;  
    }  
}
```

Advanced Observer with Concrete subjects and observers

```
Public FilmCelebrity extends Celebrity{  
    private int state;  
  
    void setState(int newState){  
        state = newState;  
        notify();  
    }  
  
    int getState(){  
        return state;  
    }  
}
```

```
public class FilmFan extends Fan{  
    private List<FilmCelebrity> filmCelebrities=  
        new ArrayList<FilmCelebrity>();  
  
    void update(FilmCelebrity fc){  
        if(filmCelebrities.contains(fc){  
            fc.getState();  
        }  
    }  
  
    void addCelebrity(FilmCelebrity fc){  
        celebrities.add(fc);  
        fc.attach(this);  
    }  
  
    void removeCelebrity(FilmCelebrity fc){  
        celebrities.remove(fc);  
        fc.remove(this);  
    }  
}
```

Singleton Pattern

```
public class HelpDesk{  
    public void getService(){  
        // Implement the service  
    }  
}  
  
public class Student{  
    HelpDesk hd = new HelpDesk();  
    hd.getService();  
}  
  
public class Teacher{  
    HelpDesk hd = new HelpDesk();  
    hd.getService();  
}
```

In reality one Single HelpDesk is serving all. No need for multiple objects.

BASIC

```
class MyClass:
    counter = 0
    def __init__(self):
        type(self).counter += 1 # Accessing class variable via the instance

    @classmethod
    def instances_created(cls):
        return cls.counter # Accessing class variable via cls
```

Singleton Pattern(EX-1)

20

class Singleton:

```
    _instance = None
```

```
    def __new__(cls):
```

```
        if cls._instance is None:
```

```
            print(' instance creating')
```

```
            cls._instance = super(Singleton, cls).__new__(cls)
```

```
        return cls._instance
```

```
    def get_service(self):
```

```
        print("Service has been provided by the Singleton instance.")
```

The `__init__` method is called after the object is created by the `__new__` method, and it initializes the object attributes with the values passed as arguments.

`__new__` is responsible for creating and returning a new instance, while `__init__` is responsible for initializing the attributes of the newly created object.

```
# Usage
```

```
if __name__ == "__main__":
```

```
    # Let's assume that multiple clients need the service
```

```
    client1 = Singleton()
```

```
    client2 = Singleton()
```

```
    client3 = Singleton()
```

```
    client1.get_service()
```

```
    client2.get_service()
```

```
    client3.get_service()
```

```
    print(client1 is client2 is client3)
```

OUTPUT=>

instance creating

Service has been provided by the Singleton instance.

Service has been provided by the Singleton instance.

Service has been provided by the Singleton instance.

True

BASIC

Instantiation: `__new__()` is responsible for creating a new instance of the class. It allocates memory and prepares the object.

Initialization: After `__new__()` returns the new instance, the `__init__()` method is called to initialize the object's state.

Example:

```
class MyClass:
    def __new__(cls, *args, **kwargs):
        instance = super(MyClass, cls).__new__(cls)
        print("Instance created")
        return instance

    def __init__(self, value):
        self.value = value
        print("Instance initialized")

obj = MyClass(10)
# Output:
# Instance created
# Instance initialized
```

Singleton Pattern(EX-2)

22

```
class HelpDesk:
```

```
    _instance = None
```

```
    def __new__(cls):
```

```
        if cls._instance is None:
```

```
            print('Creating the HelpDesk instance')
```

```
            cls._instance = super(HelpDesk,cls).__new__(cls)
```

```
        return cls._instance
```

```
    def get_service(self):
```

```
        print("Service provided by the HelpDesk.")
```

```
# Usage
```

```
if __name__ == "__main__":
```

```
    student_help_desk = HelpDesk()
```

```
    teacher_help_desk = HelpDesk()
```

```
    student_help_desk.get_service()
```

```
    teacher_help_desk.get_service()
```

```
    print(student_help_desk is teacher_help_desk)
```

SingletonPattern(EX-2-cont)

23

OUTPUT=>

Creating the HelpDesk instance

Service provided by the HelpDesk.

Service provided by the HelpDesk.

True

```

public class HelpDesk{
    public void getService(){
        // Implement the service
    }
    private static HelpDesk helpDesk = Null;
    private HelpDesk(){
        // No other class will be able to create
instance
    }
    public static HelpDesk getInstance(){
        if(helpDesk == null){
            helpDesk = new HelpDesk(); // Lazy
instance
        }
        return helpDesk;
    }
}

```

```

public class Teacher{
    //HelpDesk hd = new
HelpDesk();
    HelpDesk hd =
HelpDesk.getInstance();
    hd.getService();
}

```

```

public class Student{
    //HelpDesk hd = new HelpDesk();
    HelpDesk hd = HelpDesk.getInstance();
    hd.getService();
}

```

Singleton Pattern

Intent: Ensure a class only has one instance, and provide a global point of access to it.

Motivation: The reason behind is

- ▶ More than one instance will result in incorrect program behaviour. (thread specific)
- ▶ More than one instance will result the overuse of resources. (ex: database connection string)
- ▶ Some classes should have only one instance throughout the system for (ex: printer spooler windows 11)

Classification: Classified as one of the most known Creational Pattern

Singleton Pattern Structure

```
public class Singleton{  
    private static Singleton singletonInstance;  
  
    private Singleton(){  
        //nothing to do as object initiation will be done  
once  
    }  
    public static Singleton getInstance(){  
        if(singletonInstance == null){  
            singletonInstance = new Singleton(); // Lazy  
instance  
        }  
        return singletonInstance;  
    }  
}
```

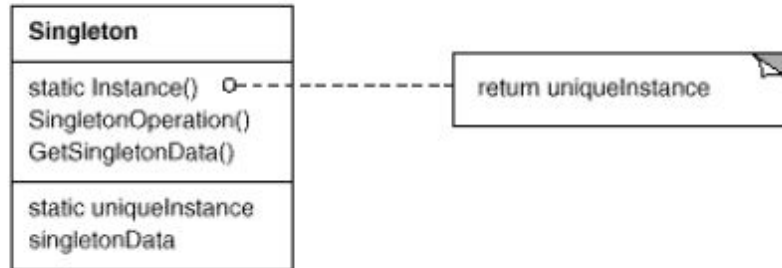
From main method call –
Singleton instance =
Singleton.getInstance();

Singleton

Participants:

- ❑ **Singleton**-defines an Instance operation that lets clients access its unique instance. Instance is a class operation .It may be responsible for creating its own unique instance. (ex: Singleton)

Structure:



Singleton

Lazy instance: Singleton make use of lazy initiation of the class. It means that its creation is deferred until it is first used. It helps to improve performance, avoid wasteful computation and reduce program memory requirement.

```
if(singleInstance == null){  
    singleInstance = new Singleton(); // Lazy initialization  
}
```

Read more:

<https://stackoverflow.com/questions/978759/what-is-lazy-initialization-and-why-is-it-useful>

Adapter Pattern

In the real world...

we are very familiar with adapters and what they do

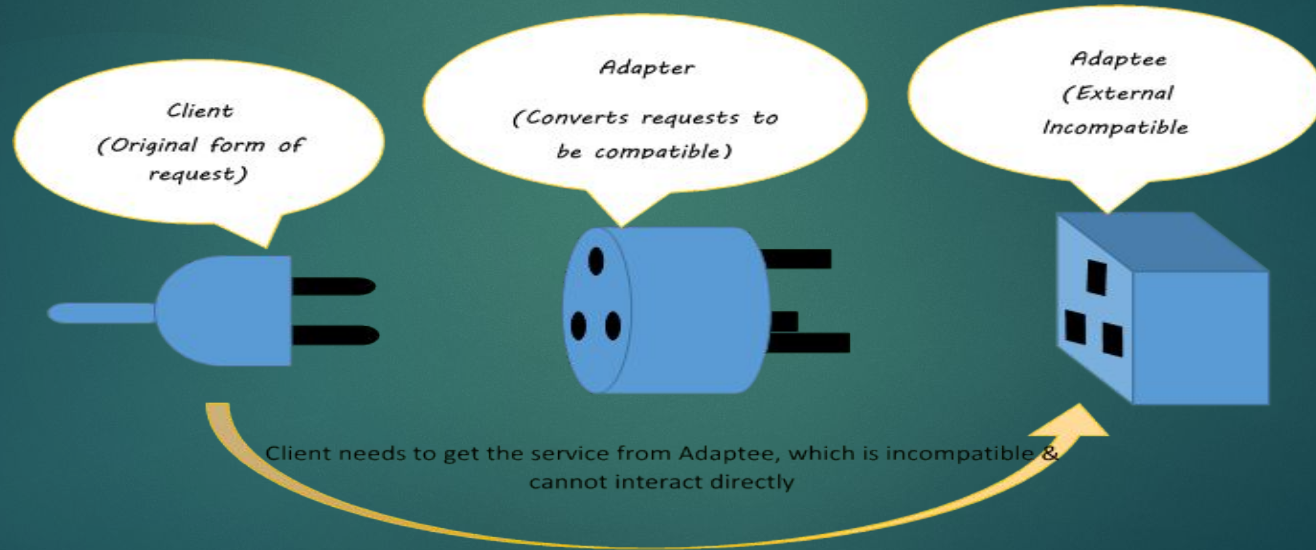


Figure 1-Adapter Pattern Concept

What about object oriented adapters?

Intent:

Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

Classified as:

A Structural Pattern

(Structural patterns are concerned with how classes and objects are composed to form larger structures.)

Also Known As:

Wrapper

Adapter Pattern

Adapter pattern can be solved in one of two ways:

- ▶ **Class Adapter:** its Inheritance based solution
- ▶ **Object Adapter:** its Object creation based solution

Class Adapter

32

Scenario: *I have a pizza making store that creates different pizzas based on the choices of people of different locations. For example – people of Dhaka like DhakaStylePizza, people of Sylhet like SylhetStylePizza.*



Class Adapter

Scenario: *I have a pizza making store that creates different pizzas based on the choices of people of different locations. For example – people of Dhaka like DhakaStylePizza, people of Sylhet like SylhetStylePizza.*

Solution: To meet the scenario, we can declare a Pizza interface and different location people can make their own style pizza by implementing the same interface.

Class Adapter

```
Public Interface Pizza{  
    abstract void toppings();  
    abstract void bun();  
}
```

```
Public class DhakaStylePizza implements  
    Pizza{  
        public void toppings(){  
            print("Dhaka cheese toppings");  
        }  
        public void bun(){  
            print("Dhaka bread bun");  
        }  
    }
```

Python Example

<https://refactoring.guru/design-patterns/adapter/python/example>

Class Adapter

- ▶ Now we want to support ChittagongStylePizza.
- ▶ The customer of Chittagong are rigid. They want to use the authentic **existing class, ChittagongPizza**
- ▶ **But we can not call it directly**, as it's not name same as **our Pizza interface**.

```
public class ChittagongPizza{  
    public void sausage(){  
        print("Ctg pizza");  
    }  
    public void bread(){  
        print("Ctg bread");  
    }  
}
```

```
Public Interface Pizza{  
    abstract void toppings();  
    abstract void bun();  
}
```

Class Adapter

- ▶ We want to adapt the existing ChittagongPizza, so it's a Adaptee.
- ▶ To do so, **introduce a** Class Adapter, **ChittagongClassAdapter**

Public class ChittagongClassAdapter extends ChittagongPizza implements Pizza{

```
    public void toppings(){  
        this.sausage();  
    }
```

```
    public void bun(){  
        this.bread();  
    }  
}
```

public class ChittagongPizza{

```
    public void sausage(){  
        print("Ctg pizza");  
    }
```

```
    public void bread(){  
        print("Ctg bread");  
    }  
}
```

Class Adapter

- ▶ We want to adapt the existing ChittagongPizza, so it's a Adaptee.
- ▶ To do so, **introduce a** Class Adapter, **ChittagongClassAdapter**
- ▶ Customer use the adapter to adapt the adaptee.

Public class ChittagongClassAdapter extends ChittagongPizza implements

Pizza{

public void toppings(){
 this.sausage();
}

public void bun(){
 this.bread();
}

}

public class ChittagongPizza{
 public void sausage(){
 print("Ctg pizza");
 }

public void bread(){
 print("Ctg bread");
}

From main method, customer call –

```
Pizza adaptedPizza = new ChittagongClassAdapter();
adaptedPizza.toppings();
adaptedPizza.bun();
```

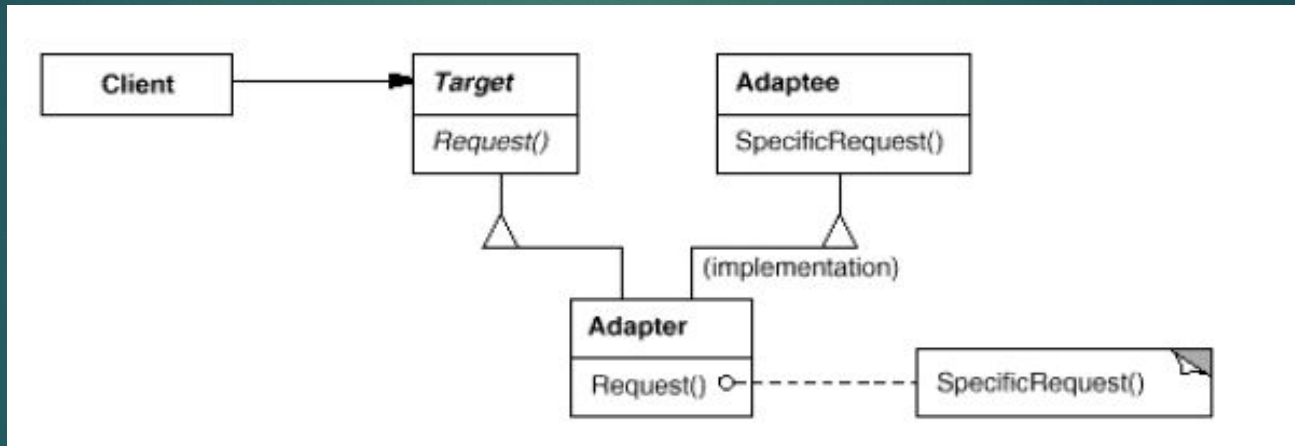

Adapter Pattern

Participants:

- ▶ **Target:** defines the domain-specific interface that Client uses. (ex: Pizza)
- ▶ **Client:** collaborates with objects conforming to the Target interface.
- ▶ **Adaptee:** defines an existing interface that needs adapting (ex: ChittagongPizza)
- ▶ **Adapter:** adapts the interface of Adaptee to the Target interface. (ex: ChittagongClassAdapter)

Adapter Pattern

► Structure:



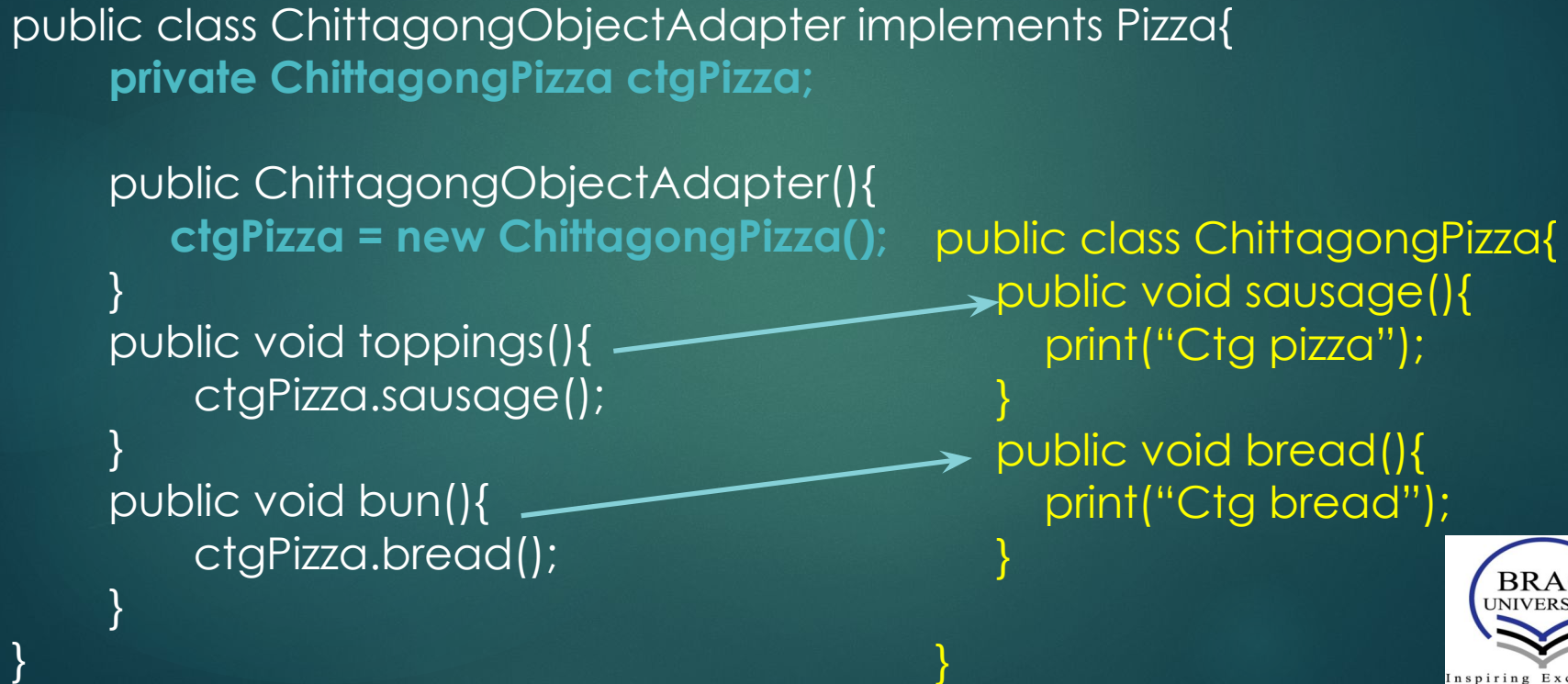
Object Adapter

40

We have the existing adaptee.

Now, create a object adapter for adapting the same ChittagongPizza **existing class**.

```
public class ChittagongObjectAdapter implements Pizza{  
    private ChittagongPizza ctgPizza;  
  
    public ChittagongObjectAdapter(){  
        ctgPizza = new ChittagongPizza();  
    }  
    public void toppings(){  
        ctgPizza.sausage();  
    }  
    public void bun(){  
        ctgPizza.bread();  
    }  
}  
  
public class ChittagongPizza{  
    public void sausage(){  
        print("Ctg pizza");  
    }  
    public void bread(){  
        print("Ctg bread");  
    }  
}
```



Pros and Cons

- ▶ **Class Adapter:** in this case, as it extends the adaptee, it can override the adaptee's methods. But, It can not use the adaptee's subclasses.
- ▶ **Object Adapter:** As we use object, a parent class object can store subclass object. So, It can adapt the subclasses as well. However, it can not override any behaviour of adaptee.
- ▶ Check the colab file to get the python examples - [Design Patterns Example.ipynb](#)

Read more:

<https://refactoring.guru/design-patterns/adapter>

<https://medium.com/@akshatsharma0610/adapter-design-pattern-in-java-fa20d6df25b8>