CSE470 – Software Engineering

SINGLETON AND ADAPTER PATTERN

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)


```
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.")
```

```
# 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

Singleton Pattern(EX-2)


```
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)


```
# 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;

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

```
public class Teacher{
      //HelpDesk hd = new
HelpDesk();
HelpDesk hd =
HelDesk.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)

Classification: Classified as one of the most known Creational Pattern

Singleton PatternStructure


```
public class Singleton{
    private static Singleton singleInstance;

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

From main method call –

```
Singleton instance =
Singleton.getInstance();
```

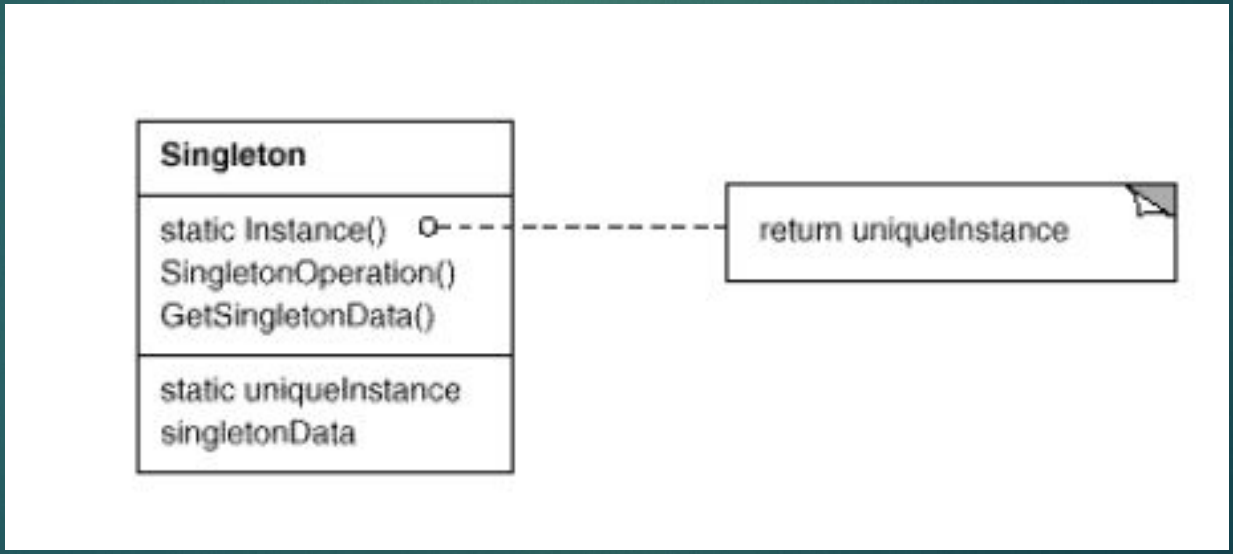
BRAC UNIVERSITY
Inspiring Excellence

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 Singletong(); // Lazy initialization
}
```

Adapter Pattern

In the real world...

we are very familiar with adapters and what they do

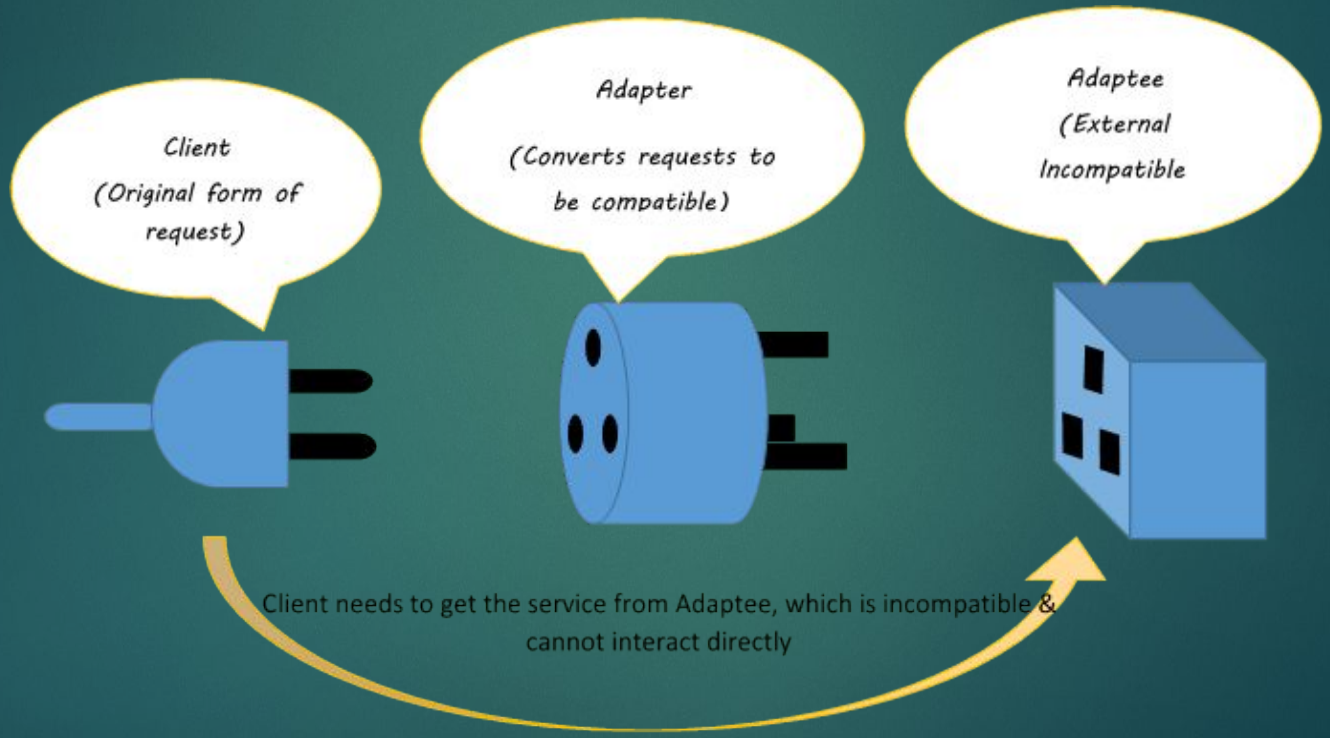


Figure 1-Adapter Pattern Concept

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

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 chesse toppings");
    }
    public void bun(){
        print("Dhaka bread bun");
    }
}
```

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 its 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 Adapater**, 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 Adapater**, 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();
```

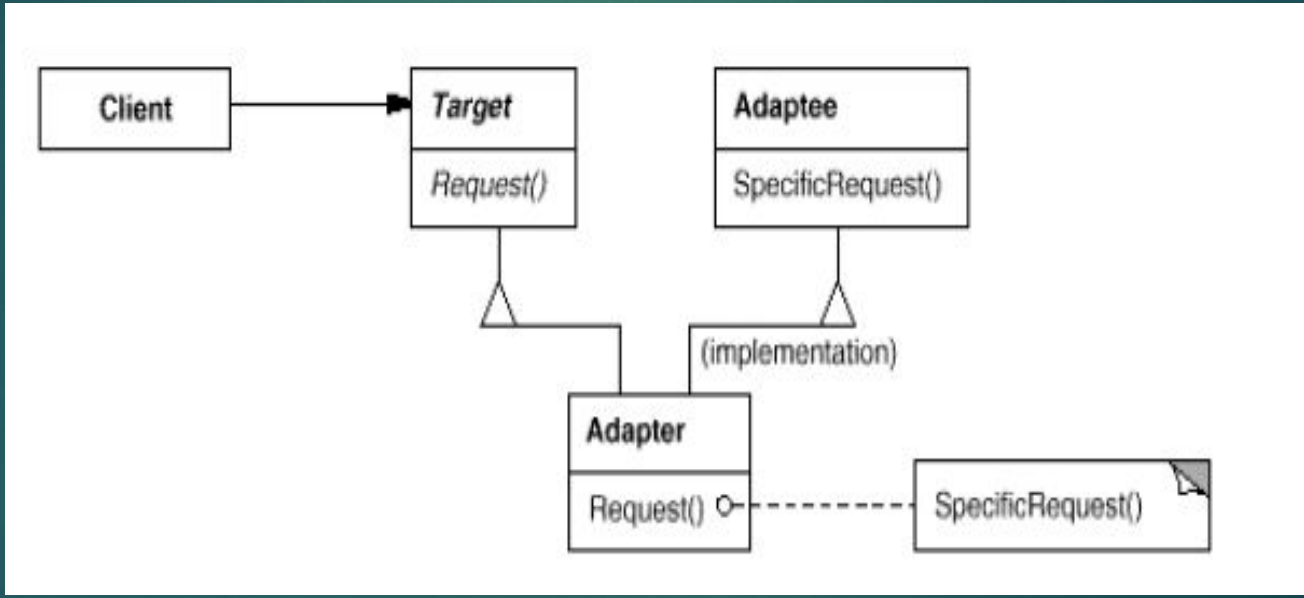
Adapter Pattern

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

We have the existing adpatee.

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

```
public class ChittagongObjectAdapter implements Pizza{
    private ChittagongPizza ctgPizza;
    public ChittagongStylePizzaObjectAdapter(){
        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");
    }

}
```

BRAC UNIVERSITY
Inspiring Excellence

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.