CSE 470
Software Engineering
Software Design Patterns

Imran Zahid
Lecturer
Computer Science and Engineering, BRAC University

BRAC UNIVERSITY
Inspiring Excellence

Recap

Why do we need Design Patterns

Designing Reusable software is difficult.

- finding good objects and abstraction
- flexibility, modularity, elegance reuse
- takes time to emerge, trial and error

Successful design does exist

- exhibit recurring class and object structure

How to describe these recurring structure?

Design Patterns

- 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.

Types of Design Patterns

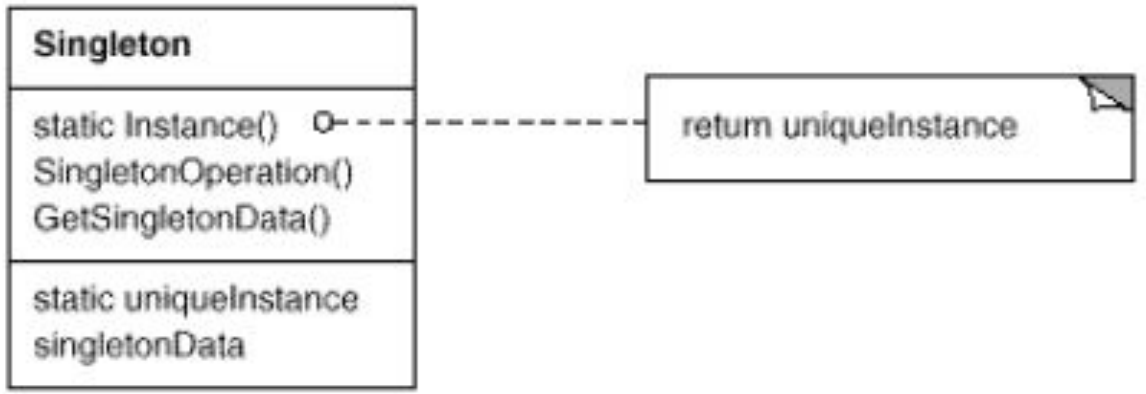
- As per the design pattern reference book Design Patterns - Elements of Reusable Object-Oriented Software, there are 23 design patterns which can be classified into three categories: Creational, Structural and Behavioral patterns.
- We'll also discuss another category of design pattern: J2EE design patterns.

Singleton Pattern

- Ensures a class has only one instance and provide a global point of access to it.
- Define the instance operation that lets clients access its unique instance. Instance is a class operation (static member function).

Singleton Pattern

Structure:



Classification: Classified as one of the most known Creational Pattern

Structure of Singleton (Python)

```
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.")
```

Structure of Singleton Cotd. - 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)
```

Structure of Singleton Cotd. - Output

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

Lazy Instance

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
}
```

BRAC UNIVERSITY
Inspiring Excellence

Observer Pattern

Observer Pattern

- Models a 1-to-many dependency between objects
- Connects the state of an observed object, the subject with many observing objects, the observers

Observer Pattern

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

Motivation:

- There may be many observer
- Each observer may react differently to the same notification
- The subject should be as decoupled as possible from the observers to allow observers to change independently of the subject.

Observer Pattern

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)

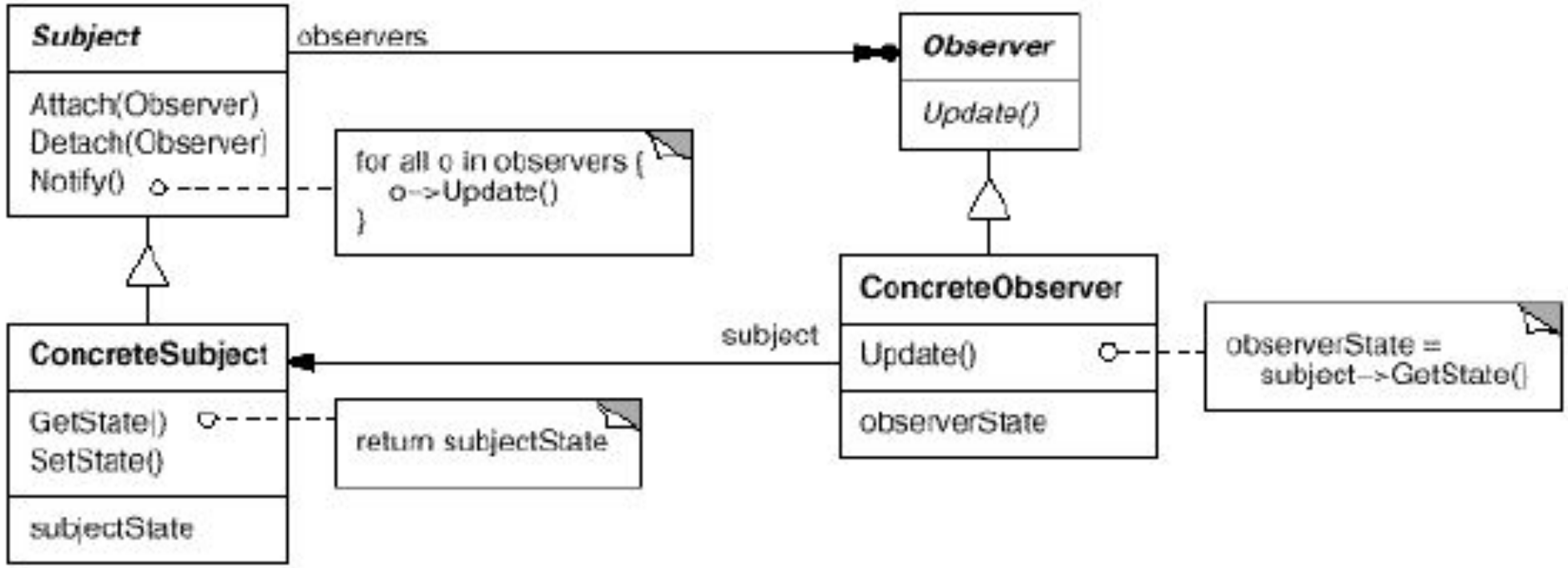
ConcreteObserver (ex: FilmFan, FashsionFan)

Observer Pattern

- The Subject (“Publisher”) represents the entity object
- Observers (“Subscribers”) attach to the Subject by calling subscribe()
- Each Observer has a view of the state of the entity object
    - The state is contained in the subclass ConcreteSubject
    - The state can be obtained and set by subclasses of type ConcreteObserver.

Observer Pattern

Structure:



Example (Python)

```
class Fan: # Observer (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)
```

```
class Celebrity:  # Subject (Celebrity)
        def __init__(self):
                self._fans = []
                self._state = None
        def attach(self, fan):
                self._fans.append(fan)
        def detach(self, fan):
                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
```

Example (Python)

```
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)
```

Example (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) {
```

```
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);
        }
}
```

Example (Java)

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
```

Example (Java)

```
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.contain(fc){
                                fc.getState();
                }
           }
        void addCelebrity(FilmCelebrity fc){
                celebrities.add(fc);
                fc.attach(this);
        }
        void removeCelebrity(FilmCelebrity fc){
                celebrities.remove(fc);
                fc.remove(this);
        }
}
```

BRAC UNIVERSITY
Inspiring Excellence

Adapter Pattern

Adapter Pattern

In the real world, we are very familiar with adapters and what they do

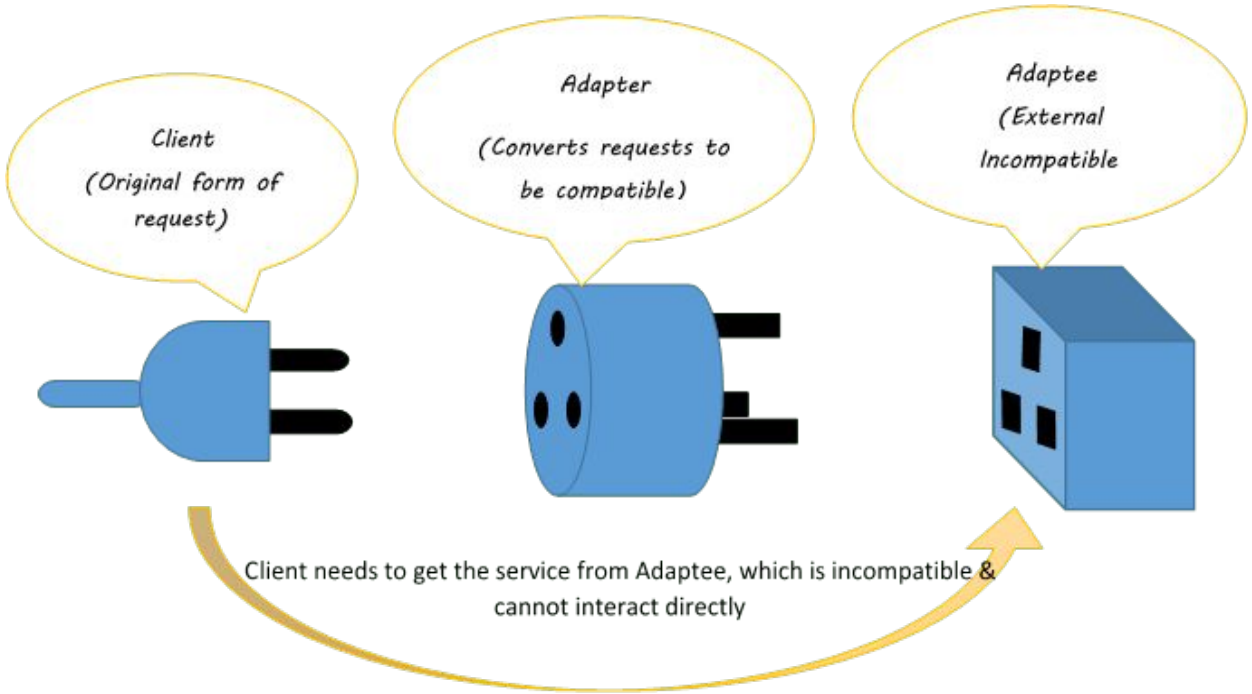


Figure 1-Adapter Pattern Concept

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

Participants:

Target: defines the domain-specific interface that Client uses.

Client: collaborates with objects conforming to the Target interface.

Adaptee: defines an existing interface that needs adapting.

Adapter: adapts the interface of Adaptee to the Target interface.

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 - Example

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 - Example

```
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”);

    }

}
```

Class Adapter - Example

```
class Pizza:
    def toppings(self):
        raise NotImplementedError()

    def bun(self):
        raise NotImplementedError()
```

```
class DhakaStylePizza(Pizza):
    def toppings(self):
        print("Dhaka cheese toppings")

    def bun(self):
        print("Dhaka bread bun")
```

Class Adapter - Example

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 the same name as our Pizza interface.

Class Adapter - Example

```
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 - Example

```
class ChittagongPizza:
    def sausage(self):
        print("Ctg pizza")

    def bread(self):
        print("Ctg bread")
```

```
class Pizza:
    def toppings(self):
        raise NotImplementedError()

    def bun(self):
        raise NotImplementedError()
```

Class Adapter - Example

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.

Class Adapter - Example

```
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();
```

Class Adapter - Example

```
class ChittagongClassAdapter(ChittagongPizza, Pizza):

    def toppings(self):

        self.sausage()



    def bun(self):

        self.bread()
```

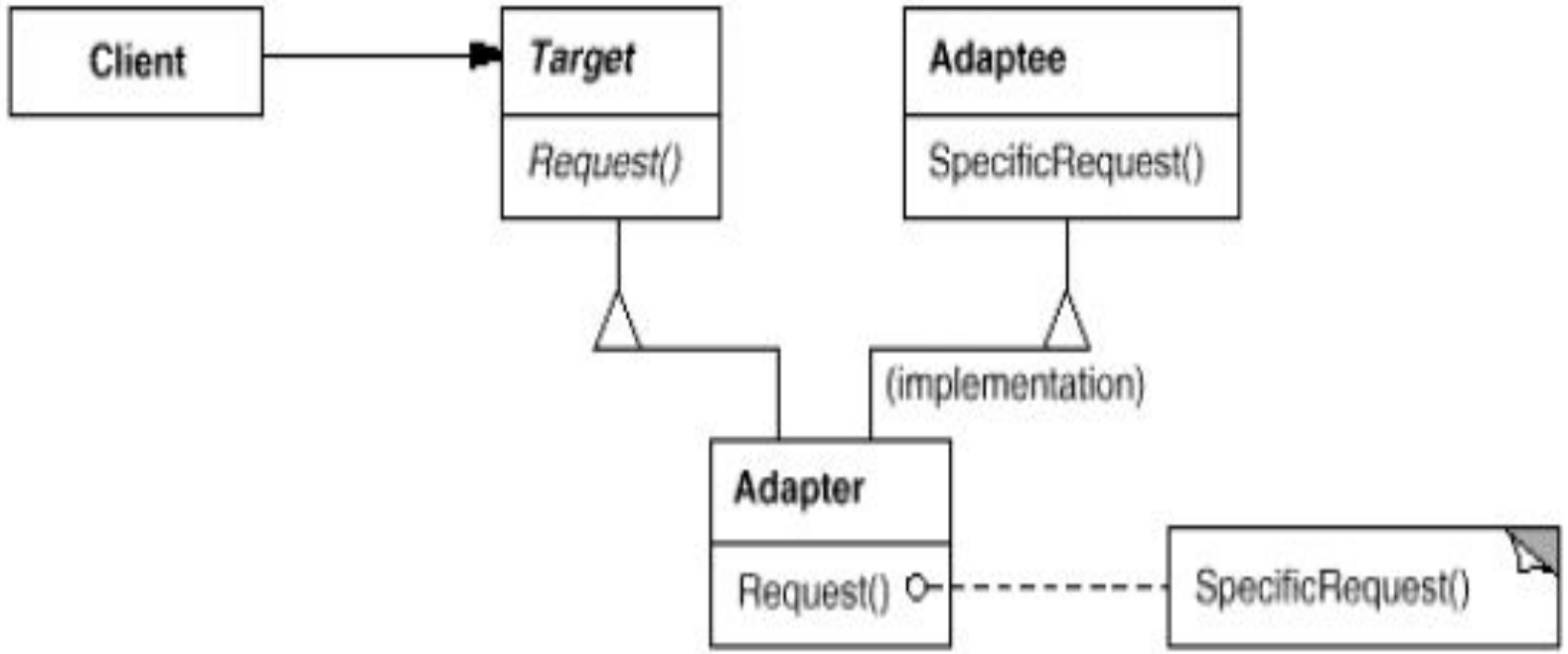
```
class ChittagongPizza:
    def sausage(self):
        print("Ctg pizza")

    def bread(self):
        print("Ctg bread")
```

```
From main method, customer call -
adaptedPizza = ChittagongClassAdapter();
adaptedPizza.toppings();
adaptedPizza.bun();
```

Class Adapter

Structure:



Object Adapter

We have the existing adpatee.

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

Object Adapter - Example

```
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”);
    }

}
```

Object Adapter - Example

```
class ChittagongObjectAdapter(Pizza):
    def __init__(self):
        self.ctg_pizza = ChittagongPizza()

    def toppings(self):
        self.ctg_pizza.sausage()

    def bun(self):
        self.ctg_pizza.bread()
```

```
class ChittagongPizza:
    def sausage(self):
        print("Ctg pizza")

    def bread(self):
        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.

Thank you