Atypon Training
Software Design Patterns
(Structural Patterns)

Instructor: Dr. Fahed Jubair

fjubair@atypon.com

ATYPON WILEY

1

Structural Patterns

- Describe how objects are connected with each other
- Often rely on Decomposition and Generalization to create new functionalities
- Example of structural patterns:
  - Façade pattern
  - Adaptor pattern
  - Proxy pattern
  - Decorator pattern

2

Façade Pattern

- Aims to hide the complexity of a software system that consists of multiple classes by creating an interface which clients use to access the software system
- Façade Pattern is done using three steps:

1. Create all classes of the software system
2. Create a class, called the façade class, that wraps the software system's classes
3. Client classes must use the façade class to access the software system

3

Façade Pattern Example

- Assume we want to design a software system that describes hotel services
- There are two types of hotels: budget hotels and luxury hotels
- Customers desire to access services such as booking rooms or inquiring room prices
- Our goal is to use the Façade pattern to design this software
- Doing so avoids the need to expose customers (i.e., clients) to the underlying complexity of having different hotel types

4

```java
public abstract class Hotel {
    private int numOfAvailableRooms;

    protected Hotel(int numOfRooms){
        if(numOfRooms < 0)
            throw new IllegalArgumentException();
        this.numOfAvailableRooms = numOfRooms;
    }

    public boolean book() {
        if(numOfAvailableRooms == 0)
            return false;
        numOfAvailableRooms--;
        return true;
    }

    public void checkOut() { numOfAvailableRooms++; }

    public int getNumOfAvailableRooms() {
        return numOfAvailableRooms;
    }

    public abstract int getRoomPrice();
    public abstract int getNumberOfStars();
}
```

Façade Pattern Example Step1: Create System Classes

5

```java
public class LuxuryHotel extends Hotel {

    public LuxuryHotel(int numOfRooms){
        super(numOfRooms);
    }

    @Override
    public int getRoomPrice() { return 1000; }

    @Override
    public int getNumberOfStars() {
        return 5;
    }
}


public class BudgetHotel extends Hotel {

    public BudgetHotel(int numOfRooms){
        super(numOfRooms);
    }

    @Override
    public int getRoomPrice() { return 200; }

    @Override
    public int getNumberOfStars() { return 1; }
}
```

Façade Pattern Example Step1: Create System Classes

6

Façade Pattern Example Step2: Create The Façade Class

```
import java.util.HashMap;
public class HotelManager {
    private HashMap<String,Hotel> hotels;
    public HotelManager() { hotels = new HashMap<String,Hotel> (); }
    public void createHotel(String hotelName, String hotelType, int numOfRooms){
        if(hotelName == null || hotelType == null || numOfRooms < 0)
            throw new IllegalArgumentException();
        if(!hotels.containsKey(hotelName)) {
            if(hotelType.equalsIgnoreCase( anotherString: "luxury"))
                hotels.put(hotelName, new LuxuryHotel(numOfRooms));
            else if(hotelType.equalsIgnoreCase( anotherString: "budget"))
                hotels.put(hotelName, new BudgetHotel(numOfRooms));
        }
    }
    public boolean book(String hotelName){
        if(hotelName == null || !hotels.containsKey(hotelName))
            return false;
        return hotels.get(hotelName).book();
    }
    public void checkOut(String hotelName){
        if(hotelName == null || !hotels.containsKey(hotelName))
            throw new IllegalArgumentException();
        hotels.get(hotelName).checkOut();
    }
    public int getRoomPrice(String hotelName){
        if(hotelName == null || !hotels.containsKey(hotelName))
            throw new IllegalArgumentException();
        return hotels.get(hotelName).getRoomPrice();
    }
    public int getNumOfAvailableRooms(String hotelName){
        if(hotelName == null || !hotels.containsKey(hotelName))
            throw new IllegalArgumentException();
        return hotels.get(hotelName).getNumOfAvailableRooms();
    }
    public int getNumberOfStars(String hotelName){
        if(hotelName == null || !hotels.containsKey(hotelName))
            throw new IllegalArgumentException();
        return hotels.get(hotelName).getNumberOfStars();
    }
}
```

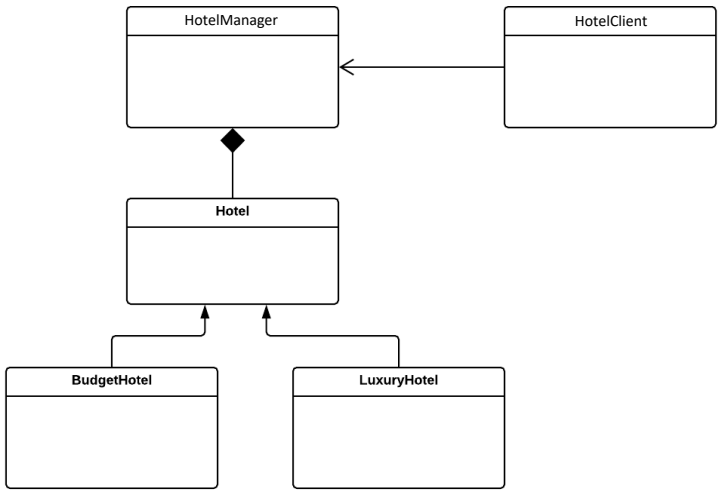

7

Façade Pattern Example Step3: Clients

Clients Access The System Using The Façade Class

```
public class HotelClient {

    public static void main(String[] args){
        HotelManager hotelManager = new HotelManager();

        hotelManager.createHotel( hotelName: "Sheraton", hotelType: "luxury",  numOfRooms: 200);
        hotelManager.createHotel( hotelName: "Marriot",  hotelType: "luxury",  numOfRooms: 300);
        hotelManager.createHotel( hotelName: "Amman",    hotelType: "budget",  numOfRooms: 20);
        hotelManager.createHotel( hotelName: "Zarqa",    hotelType: "budget",  numOfRooms: 25);

        System.out.println(hotelManager.getNumberOfStars( hotelName: "Sheraton"));
        System.out.println(hotelManager.getNumberOfStars( hotelName: "Marriot"));
        System.out.println(hotelManager.getNumberOfStars( hotelName: "Amman"));
        System.out.println(hotelManager.getNumberOfStars( hotelName: "Zarqa"));

        System.out.println(hotelManager.book( hotelName: "Marriot"));
        System.out.println(hotelManager.book( hotelName: "Zarqa"));
        System.out.println(hotelManager.book( hotelName: "Zarqa"));
        System.out.println(hotelManager.book( hotelName: "Sheraton"));
        System.out.println(hotelManager.book( hotelName: "Sheraton"));
        System.out.println(hotelManager.book( hotelName: "Sheraton"));
        System.out.println(hotelManager.book( hotelName: "Sheraton"));

        System.out.println(hotelManager.getNumOfAvailableRooms( hotelName: "Sheraton"));
        System.out.println(hotelManager.getNumOfAvailableRooms( hotelName: "Marriot"));
        System.out.println(hotelManager.getNumOfAvailableRooms( hotelName: "Amman"));
        System.out.println(hotelManager.getNumOfAvailableRooms( hotelName: "Zarqa"));
    }
}
```


8



Façade Pattern Example Step3: UML Diagrams

9

Remarks About The Façade Pattern

- hides the complexity of a software system by encapsulating it behind a unifying wrapper called a façade class
- Removes the need for client classes to manage a software system on their own, resulting in less coupling between the software system's classes and the client classes
- Acts simply as a point of entry to a software system and does not add any new functionality to the software system
- One obvious issue is what if Façade class becomes too big?

10

Adaptor Pattern

- Not all software systems have compatible interfaces
- In other words, the output of one system software may not conform to the expected input format by another software system
- This frequently occurs when trying to incorporate third-party libraries in your code
- Solution: the adapter design pattern facilitates communication between two existing systems by providing a compatible interface

11

Adaptor Pattern Example

- Consider you wrote software that describes TVs
- In your code, you have an interface that is called, TV, which describes TVs in general
- You have two concrete TV classes: StandardTV and HDTV that both implement TV interface
- You have a class, called TVplayer, that represents the client in your software
- Imagine you found third-party code that contains a new kind of TV, called UltraTV, that you want to incorporate in your software
- Show how to use the adaptor pattern to allow the client class to also use the new TV?

12

Adaptor Pattern Example TV Software

```
public interface TV {
   public void playMovie();
}
public class HDTV implements TV {
   @Override
   public void playMovie() {
      System.out.println("play movie in 1920 × 1080");
   }
}
public class StandardTV implements TV {
   @Override
   public void playMovie() {
      System.out.println("play movie in 1280 × 720");
   }
}
public class TVPlayer {   // client class
   public static void main(String[] args){
      TV tv1 = new StandardTV();
      TV tv2 = new HDTV();
      tv1.playMovie();
      tv2.playMovie();
   }
}
```


13

Adaptor Pattern Example Third-party Class

```
public class UltraTV {

   public void play4k(){
      System.out.println("play video in 3840 x 2160");
   }

}
```

- Third-party code uses different interface than the software system we have
- Therefore, this class cannot be accessed directly by our software system's client, TVplayer class
- Solution: access it indirectly using the Adpator pattern, which introduces a new class

14

Adaptor Pattern Example The Adaptor Class

```
public class AdaptorTV implements TV {

   private UltraTV ultraTV;

   public AdaptorTV(UltraTV ultraTV){
      this.ultraTV = ultraTV;
   }

   public void playMovie(){
      this.ultraTV.play4k();
   }
}
```

Implement the expected interface by the client

Wrap the third-party class inside the adaptor class

The expected interface indirectly invokes the interface of the third-party class

15

Adaptor Pattern Example Client Class

```
public class TVPlayer {

   public static void main(String[] args){
      TV tv1 = new StandardTV();
      TV tv2 = new HDTV();
      TV tv3 = new AdaptorTV(new UltraTV());

      tv1.playMovie();
      tv2.playMovie();
      tv3.playMovie();
   }
}
```

Third-part class now conforms to the expected interface by wrapping it inside the adaptor class

Same interface for all TVs is used

Exercise: Draw the UML diagrams

16

Remarks About The Adaptor Pattern

- Indirectly changes the third-party's interface into one that the client is expecting (achieved by implementing a target interface)
- Indirectly translates the client's request into one that the third-party code is expecting
- Final outcome: reuse an existing third-party software with an incompatible interface

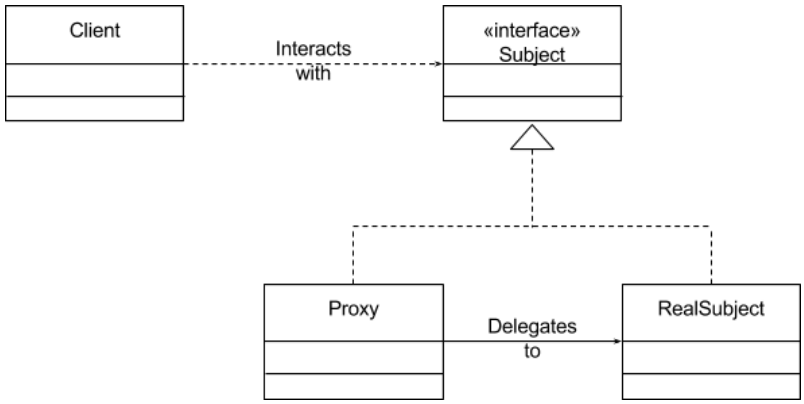
17

Proxy Pattern

- Introduces a proxy class that represents a simplified, lightweight version of another class
- A proxy class performs the same API as an original class, but may delegate requests to the original class to achieve them
- Proxy class implements the same interface as the original class
- Using the proxy pattern is useful in many scenarios, such as:
  1. Controlling access to a class with sensitive information
  2. Avoiding instantiating a class that is resource-intensive
  3. Speeding up the access to an class that exists remotely

18

Proxy Pattern
UML Diagram



19

Proxy Pattern Example

The example contains four classes:

- Image interface, which provides a general description for an image
- RealImage class, which provides concrete implementation
- ProxyImage class, which provides a proxy implementation
- ImageClient class, which represents the client

20

Proxy Pattern Example Image Interface and RealImage Class

```
public interface Image {

   public String getName();
   public void display();

}

public class RealImage implements Image {
    private String imageName;
    public RealImage(String imageName){
       this.imageName = imageName;
       System.out.println("loading image from disk");
    }

    @Override
    public String getName() {
       return imageName;
    }

    @Override
    public void display() {
       System.out.println("displaying image");
    }
}
```


21

Proxy Pattern Example ProxyImage Class

```
public class ProxyImage implements Image {
   private String imageName;
   private RealImage realImage;   // Proxy class wraps original class

   public ProxyImage(String imageName){
      this.imageName = imageName;
      realImage = null;
   }

   @Override
   public String getName() {
      return imageName;     // handle light requests by proxy
   }

   @Override
   public void display() {
      if(realImage == null){
         realImage = new RealImage(imageName);   // delegate to original class
      }
      realImage.display();
   }
}
```

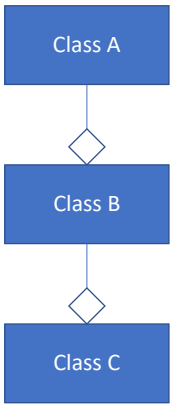

22

Decorator Pattern

- Using aggregation, the decorator pattern allows to dynamically add new functionalities to an existing object without the need to modify its code
- Create a decorator class that aggregates the original class
- Aggregation allows the decorator to augment the methods of the original class, while adding new function

23

Aggregation Stack



Object A acts as a base class

Object B aggregates Objects A and adds new functionality

Object C aggregates Objects B and adds new functionality

24

Decorator Pattern Example

- Consider writing classes that describe different kinds of beverages
- Let Beverage be an abstract class that describes beverages in general
- Let EspressoBeverage and AmericanoBeverage be two concrete classes of Beverage
- What if we decided to add condiments to beverages (such as milk, caramel or chocolate)?
- We will show how to use the Decaroter pattern to dynamically add condiments to beverages without modifying Beverages classes

25

Decorator Pattern Example Beverage Classes

```
public abstract class Beverage {
   protected String description = "plain beverage";
   public String getDescription(){
      return description;
   }
   public abstract double getPrice();
}

public class EspressoBeverage extends Beverage {
   public EspressoBeverage(){
      description = "espresso beverage";
   }
   @Override
   public double getPrice() {
      return 3.12;
   }
}
public class AmericanoBeverage extends Beverage {
   public AmericanoBeverage(){
      description = "americano beverage";
   }
   @Override
   public double getPrice() {
      return 2.23;
   }
}
```


26

Decorator Pattern Example
Decorator Class

```
public abstract class BeverageWithCondiments extends Beverage {

   protected Beverage beverage;

   protected BeverageWithCondiments(Beverage beverage){
      this.beverage = beverage;
   }

   public abstract String getDescription();
   public abstract double getPrice();

}
```

Decorator class extends base class

Decorator class aggregates base class

Require concrete decorator classes to override to add new functionality

27

Decorator Pattern Example
Concrete Decorator Classes

```
public class BeverageWithCaramel extends BeverageWithCondiments {

   public BeverageWithCaramel(Beverage beverage){
      super(beverage);
   }

   @Override
   public String getDescription() {
      return "caramel, " + beverage.getDescription();
   }

   @Override
   public double getPrice() {
      return 0.5 + beverage.getPrice();
   }
}
```

Augment base class behavior and add new behavior

28

Decorator Pattern Example
Concrete Decorator Classes

```
public class BeverageWithMilk extends BeverageWithCondiments {

    public BeverageWithMilk(Beverage beverage){
        super(beverage);
    }

    @Override
    public String getDescription() {
        return "milk, " + beverage.getDescription();
    }

    @Override
    public double getPrice() {
        return 0.2 + beverage.getPrice();
    }
}
```

Augment base class behavior
and add new behavior

29

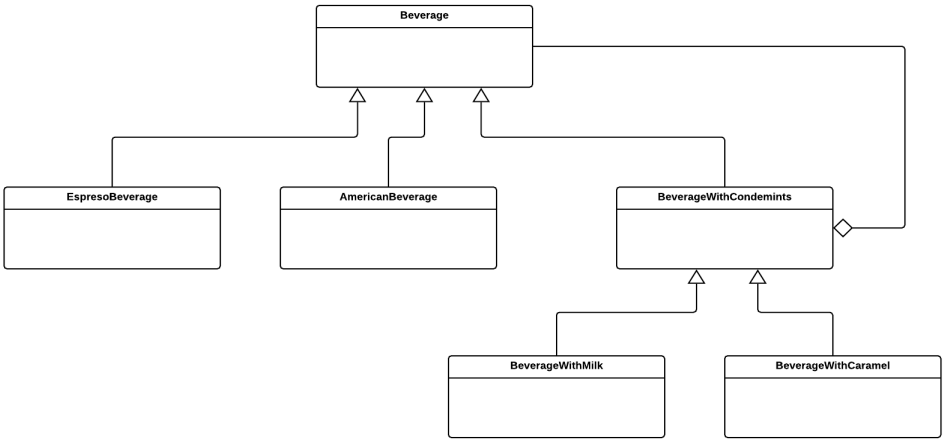
Decorator Pattern Example Test

```
public class BeverageExampleTest {

    public static void main(String[] args){
        Beverage beverage1 = new EspressoBeverage();
        Beverage beverage2 = new BeverageWithCaramel(beverage1);
        System.out.println(beverage1.getDescription());
        System.out.println(beverage2.getDescription());

        Beverage beverage3 = new AmericanoBeverage();
        Beverage beverage4 = new BeverageWithMilk(beverage3);
        beverage4 = new BeverageWithMilk(beverage4);
        beverage4 = new BeverageWithCaramel(beverage4);
        System.out.println(beverage3.getDescription());
        System.out.println(beverage4.getDescription());

    }
}
```

```
// output
espresso beverage
caramel, espresso beverage
americano beverage
caramel, milk, milk, americano beverage
```


30

Decorator Pattern Example
UML Diagram



31

Remarks About The Decorator Pattern

- Aggregation is used to dynamically add new behaviours to already existing classes
- Polymorphism is achieved by extending a single class (or implementing a single interface)
- Aggregation lets us create a stack of objects, each of which is aggregated in a one-to-one relationship with the object below it in the stack

32