Atypon Training
Software Design Patterns
(Behavioral Patterns)

Instructor: Dr. Fahed Jubair

fjubair@atypon.com

ATYPON WILEY

1

Behavioral Patterns

- Deal with how to distribute work between independent objects
- Example of behavioral patterns:
  - Template Method pattern
  - Chain of Responsibility pattern
  - Mediator pattern
  - Observer pattern
  - Visitor pattern

2

Template Method Pattern

- Define the skeleton of an algorithm in an operation, deferring some steps to subclasses
- Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure
- The template method is best used when you can generalize between multiples classes into a new superclass such that the superclass contains a set of general steps that all classes follow

3

Template Method Pattern Example

- This example is taken from https://www.coursera.org/learn/design-patterns
- Consider multiple classes that describe making pasta dishes
- Regardless of the past dish type, all pastas are made following the same steps:
  1. Boil water
  2. Cook pasta
  3. Drain pasta
  4. Add pasta
  5. Add mean
  6. Add sauce
  7. Add garnish
- This example shows how to use the template method pattern by creating superclass PastaDish and two superclasses SpagettiMeatballs and PenneAlfredo

4

Template Method Pattern Example Superclass

```
public abstract class PastaDish {
   public final void makeRecipe(){
      boilWater();
      cookPasta();
      drainPasta();
      addPasta();
      addMeat();
      addSauce();
      addGarnish();
   }
   private void boilWater(){System.out.println("boil water");}
   private void cookPasta(){System.out.println("cook pasta");}
   private void drainPasta(){System.out.println("drain pasta");}
   protected abstract void addPasta();
   protected abstract void addMeat();
   protected abstract void addSauce();
   protected abstract void addGarnish();
}
```

- This is the template method that all subclasses must follow
- The keyword final is used to prevent overriding by superlcasses

Defer implementation of these steps to subclasses

5

Template Method Pattern Example Subclasses

```
public class SpaghettiMeatballs extends PastaDish {
   @Override
   protected void addPasta() {
      System.out.println("add spaghetti");
   }
   @Override
   protected void addMeat() {
      System.out.println("add meatballs");
   }
   @Override
   protected void addSauce() {
      System.out.println("add tomato sauce");
   }
   @Override
   protected void addGarnish() {
      System.out.println("add parmesan cheese");
   }
}
```


6

Template Method Pattern Example Subclasses

```
public class PenneAlfredo extends PastaDish {
   @Override
   protected void addPasta() {
      System.out.println("add penne");
   }
   @Override
   protected void addMeat() {
      System.out.println("add chicken");
   }
   @Override
   protected void addSauce() {
      System.out.println("add alfredo sauce");
   }
   @Override
   protected void addGarnish() {
      System.out.println("add parsley");
   }
}
```


7

Template Method Pattern Example Test Class

```
public class PastaDishTest {
   public static void main(String[] args){
      PastaDish dish1 = new SpaghettiMeatballs();
      PastaDish dish2 = new PenneAlfredo();

      dish1.makeRecipe();
      System.out.println("--------------------");
      dish2.makeRecipe();
   }
}
```

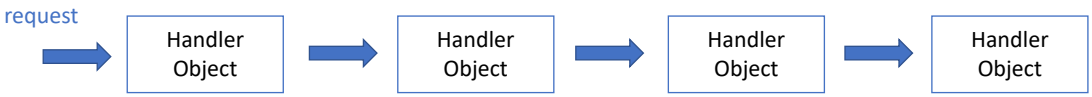
Exercise: draw UML diagrams for the template method pattern example

```
// output
boil water
cook pasta
drain pasta
add spaghetti
add meatballs
add tomato sauce
add parmesan cheese
--------------------
boil water
cook pasta
drain pasta
add penne
add chicken
add alfredo sauce
add parsley
```


8

Chain of Responsibility Pattern

- Chain of objects that is responsible for handling requests
- If one object cannot handle a request, it passes this request to the next object in the chain
- A request cannot be handled only if all objects cannot handle the request



9

Chain of Responsibility Pattern

- Decouple the sender of request from the handler
- The sender of request needs not worry about the details of how the request is handled
- One real-life example on chain of responsibility pattern is fixing a car
  - For example, starts with checking if battery is dead
  - If this fails, check oil levels
  - If this fails, check gas emission
  - If this fails, check starter motor
  - And so on

10

Chain of Responsibility Pattern Example

This example has the following classes

- RequestHandler<T>, which is an abstract class that describes to handle request (in general), where T is the type of the request
- Three concrete subclasses (PrimeHandler, SquareHandler, CubeHandler) that can be used to build a chain of request handlers
- RequestHandlerClient, which is the sender of the request

11

Chain of Responsibility Pattern Example RequestHandler Class

```
public abstract class RequestHandler<T> {

   protected RequestHandler nextHandler = null;

   public void setNext(RequestHandler nextHandler){
      if(nextHandler == null)
         throw new IllegalArgumentException();
      this.nextHandler = nextHandler;
   }

   public abstract void handleRequest(T request);

}
```


12

```
public class PrimeHandler extends RequestHandler<Integer> {
   @Override
   public void handleRequest(Integer request) {
      if(isPrimeNumber(request)){
         System.out.println("Request is handled by PrimeHandler");
         return;
      }
      if(nextHandler!=null)
         nextHandler.handleRequest(request);
      else
         System.out.println("Request cannot be handled");
   }

   private boolean isPrimeNumber(int x){
      if(x%2==0)
         return false;
      int max = (int)Math.sqrt(x);
      for(int k=3; k<=max; k+=2)
         if(x%k==0)
            return false;
      return true;
   }
}
```

Chain of Responsibility Pattern Example Concrete Handler Classes

13

```
public class SquareHandler extends RequestHandler<Integer> {
   @Override
   public void handleRequest(Integer request) {
      if(isSquareNumber(request)){
         System.out.println("Request is handled by SquareHandler");
         return;
      }
      if(nextHandler!=null)
         nextHandler.handleRequest(request);
      else
         System.out.println("Request cannot be handled");
   }

   private boolean isSquareNumber(int x){
      int sqrt = (int)Math.sqrt(x);
      if(sqrt*sqrt == x)
         return true;
      else
         return false;
   }
}
```

Chain of Responsibility Pattern Example Concrete Handler Classes

14

Chain of Responsibility Pattern Example Concrete Handler Classes

```
public class CubeHandler extends RequestHandler<Integer> {
   @Override
   public void handleRequest(Integer request) {
      if(isCubeNumber(request)){
         System.out.println("Request is handled by CubeHandler");
         return;
      }
      if(nextHandler!=null)
         nextHandler.handleRequest(request);
      else
         System.out.println("Request cannot be handled");
   }

   private boolean isCubeNumber(int x){
      int cbrt = (int)Math.cbrt(x);
      if(cbrt*cbrt*cbrt == x)
         return true;
      else
         return false;
   }
}
```


15

Chain of Responsibility Pattern Example Client Class

```
public class RequestHandlerClient {

   private static RequestHandler<Integer> handler1, handler2, handler3;

   public static void main(String[] args){
      handler1 = new PrimeHandler();
      handler2 = new SquareHandler();
      handler3 = new CubeHandler();

      handler1.setNext(handler2);
      handler2.setNext(handler3);

      handler1.handleRequest(9);
      handler1.handleRequest(27);
      handler1.handleRequest(22);
      handler1.handleRequest(13);
      handler1.handleRequest(64);
   }
}
```

```
// output
Request is handled by SquareHandler
Request is handled by CubeHandler
Request cannot be handled
Request is handled by PrimeHandler
Request is handled by SquareHandler
```


16

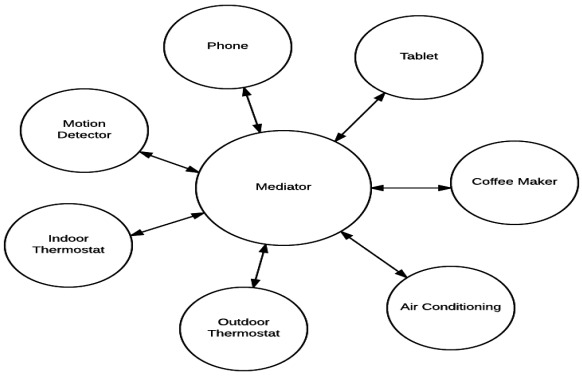
Mediator Pattern

- Introduces a mediator class that is responsible for handling all communication between all classes in the system
- Centralizing the communication in one objects allows for loose coupling and easier maintenance
- However, be careful of making the mediator class too big, which makes it harder to maintain and debug

17

Mediator Pattern

- Imagine building a code that represents a smart home
- Using the mediator pattern, the code is likely to look like this



18

Mediator Pattern Example

- This example uses the mediator pattern to build a simple chat application in which users exchange one-to-one messages
- The example has the following examples:
  - ChatUser class, which represents a user
  - ChatMediator class, which represents the mediator class
  - ChatExampleTest, which is the test class

19

Mediator Pattern Example ChatUser Class

```
public class ChatUser {
    private ChatMediator chatMediator = null;
    private final String userName;

    protected ChatUser(String userName){
        this.userName = userName;
    }

    public String getUserName(){
        return userName;
    }

    public ChatMediator getChatMediator() {
        return chatMediator;
    }

    public void joinChatMediator(ChatMediator chatMediator) {
        if(chatMediator != null)
            chatMediator.leave( chatUser: this);
        this.chatMediator = chatMediator;
        this.chatMediator.join( chatUser: this);
    }

    public void sendMessage(String receiverName, String message){
        System.out.println("[" + userName + " sends the following message to " + receiverName + "]: " + message);
        this.chatMediator.exchangeMessage(this.userName, receiverName, message);
    }

    public void receiveMessage(String senderName, String message){
        System.out.println("[" + userName + " receives the following message from " + senderName + "]: " + message);
    }
}
```


20

Mediator Pattern Example ChatMediator Class

```
import java.util.HashMap;
import java.util.Map;

public class ChatMediator {

    private Map<String,ChatUser> users ;

    public ChatMediator(){
        users = new HashMap<String,ChatUser>();
    }

    public void join(ChatUser chatUser){
        if(users.containsKey(chatUser.getUserName()))
            throw new RuntimeException("Username is already taken");
        users.put(chatUser.getUserName(),chatUser);
    }

    public void leave(ChatUser chatUser){
        if(users.containsKey(chatUser.getUserName()))
            throw new RuntimeException("User does not exist");
        users.remove(chatUser.getUserName());
    }

    public void exchangeMessage(String senderName, String receiverName, String message){
        if(!users.containsKey(senderName) || !users.containsKey(receiverName))
            throw new RuntimeException("User does not exist");
        if(senderName.equals(receiverName))
            throw new RuntimeException("self messages are not allowed");
        users.get(receiverName).receiveMessage(users.get(senderName).getUserName(),message);
    }
}
```


21

Mediator Pattern Example ChatExampleTest Class

```
public class ChatExampleTest {

    public static void main(String[] args){
        ChatMediator chatMediator = new ChatMediator();

        ChatUser user1 = new ChatUser( userName: "Ahmad");
        ChatUser user2 = new ChatUser( userName: "Zaina");
        ChatUser user3 = new ChatUser( userName: "Hashim");
        ChatUser user4 = new ChatUser( userName: "Dalia");

        user1.joinChatMediator(chatMediator);
        user2.joinChatMediator(chatMediator);
        user3.joinChatMediator(chatMediator);
        user4.joinChatMediator(chatMediator);

        user1.sendMessage( receiverName: "Zaina",  message: "Wanna meet tommorrow?");
        user2.sendMessage( receiverName: "Ahmad",  message: "Sure!");
        user3.sendMessage( receiverName: "Dalia",  message: "I do not like you.");
    }
}
```

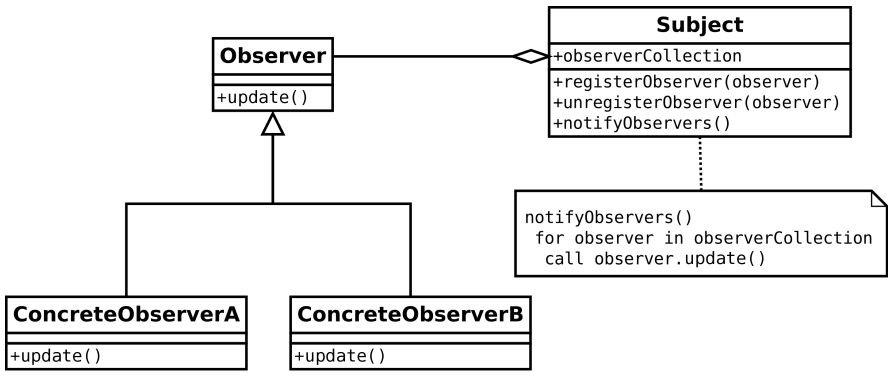

22

Observer Pattern

- Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically
- The observer pattern typically has the following two classes: Observer and Subject
  - Multiple Observer objects are attached to a Subject object
  - The Subject object is responsible for notifying all attached Observer objects when changes to the Subject object's attributes are made

23

Observer Pattern UML Representation



Source: https://en.wikipedia.org/wiki/Observer_pattern

24

Observer Pattern Example

- This example uses the Observer pattern to describe how subscribers to a blog are notified of new updates in the blog
- The example has the following classes:
  - Observer interface, which represents observers (i.e., subscribers in this example)
  - Subject interface, which represents subjects (i.e., blogs in this example)
  - Subscriber class, which represents a concrete Observer object
  - Blog, which represents a concrete Subject object
  - ObserverPatternTest class, which represents the test class
- Observer pattern simplifies distributing and handling notifications of changes across systems in a manageable and controlled way

25

Observer Pattern Example
Observer and Subscriber Classes

```
public interface Observer {

    public void update(String updateMessage);

}

public class Subscriber implements Observer {
    private final String subscriberName;

    public Subscriber(String subscriberName) {
        this.subscriberName = subscriberName;
    }

    public void update(String updateMessage){
        System.out.println(subscriberName + " received the following update: " + updateMessage);
    }
}
```


26

```java
public interface Subject {
    public void register(Observer e);
    public void unregister(Observer e);
    public void notifyUpdate();
}

import java.util.ArrayList;
public class Blog implements Subject {
    private ArrayList<Observer> observers = new ArrayList<Observer>();
    private final String blogName;

    public Blog(String blogName){
        this.blogName = blogName;
    }

    public void register(Observer e){
        observers.add(e);
    }

    public void unregister(Observer e){
        observers.remove(e);
    }

    public int getNumberOfSubscribers(){
        return observers.size();
    }

    public void notifyUpdate(){
        for(Observer o: observers)
            o.update( updateMessage: blogName + " has been updated");
    }
}
```

Observer Pattern Example Subject and Blog Classes

27

```java
public class ObserverPatternTest {

    public static void main(String[] args){

        Subscriber user1 = new Subscriber( subscriberName: "Ahmad");
        Subscriber user2 = new Subscriber( subscriberName: "Hashim");
        Subscriber user3 = new Subscriber( subscriberName: "Zaina");
        Subscriber user4 = new Subscriber( subscriberName: "Dalia");

        Blog blog = new Blog( blogName: "atypon.com");
        blog.register(user1);
        blog.register(user2);
        blog.register(user3);
        blog.register(user4);
        System.out.println(blog.getNumberOfSubscribers());

        blog.notifyUpdate();
        blog.notifyUpdate();

        blog.unregister(user1);
        blog.unregister(user2);
        blog.unregister(user3);
        blog.unregister(user4);
        System.out.println(blog.getNumberOfSubscribers());
    }
}
```

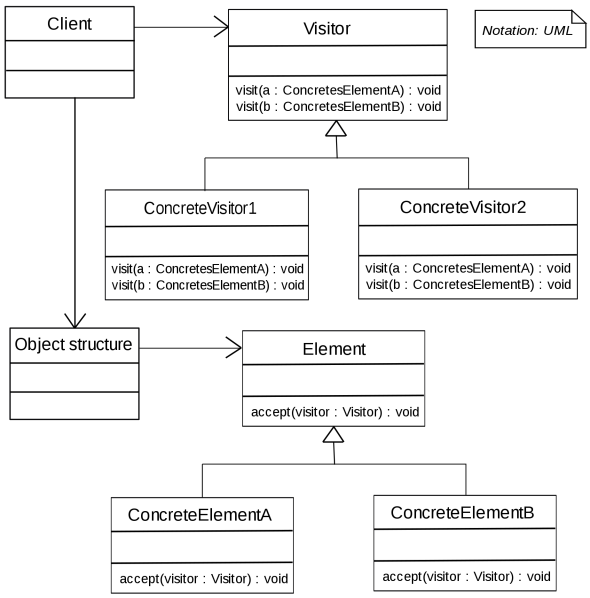
Observer Pattern Example Test Class

28

Visitor Pattern

- Represent an operation to be performed on the elements of an object structure
- Visitor lets you define a new operation without changing the classes of the elements on which it operates
- The Visitor pattern aims to decouple performing an operation on an existing object without modifying its code
- Typically, this pattern has Visitable class and Visitor class

29



Visitor Pattern UML Representation

Source: https://en.wikipedia.org/wiki/Visitor_pattern

30

Visitor Pattern Example

- This example describes how to use the Visitor pattern to calculate the price of fruit items in a shopping cart
- The following classes are used:

1. FruitItem, an abstract class that represents a purchased fruit item (quantity and price)
2. Apple, Kiwi, and Orange, which are concrete subclasses of FruitItem
3. ShoppingCart, a class that represent the shopping cart in our system, which has a list of fruit items
4. ShoppingCartVisitor, an interface that represent the visitor class, which aims to calculate the price of the items in the shopping cart
5. DiscountVisitor and CouponVisitor, which are concrete subclasses of ShoppingCartVisitor

31

```
public abstract class FruitItem {
   private final int quantity;
   protected FruitItem(int quantity) {
      this.quantity = quantity;
   }

   public int getQuantity(){
      return quantity;
   }

   public abstract double unitPrice();

   public double totalPrice(){
      return unitPrice() * getQuantity();
   }
}
```

```
public class Apple extends FruitItem {
   public Apple(int quantity) {
      super(quantity);
   }

   @Override
   public double unitPrice() {
      return 19.21;
   }
}
```

```
public class Orange extends FruitItem {
   public Orange(int quantity) {
      super(quantity);
   }

   @Override
   public double unitPrice() {
      return 28.54;
   }
}
```

```
public class Kiwi extends FruitItem{
   public Kiwi(int quantity) {
      super(quantity);
   }

   @Override
   public double unitPrice() {
      return 49.12;
   }
}
```

Visitor Pattern Example Fruit Items Classes

32

Visitor Pattern Example
Shopping Cart Class

```
import java.util.ArrayList;
import java.util.Iterator;

public class ShoppingCart {
   private ArrayList<FruitItem> fruitItems = new ArrayList<FruitItem>();

   public void addFruitItem(FruitItem e){
      fruitItems.add(e);
   }

   public Iterator<FruitItem> itemsList(){
      return fruitItems.listIterator();
   }

   public double accept(ShoppingCartVisitor shoppingCartVisitor){
      return shoppingCartVisitor.visit(this);
   }
}
```


33

Visitor Pattern Example
Shopping Cart Visitor Interface

```
public interface ShoppingCartVisitor {

   public double visit(ShoppingCart shoppingCart);

}
```


34

Visitor Pattern Example Discount Visitor

```
import java.util.Iterator;

public class DiscountVisitor implements ShoppingCartVisitor {

   private double discountAmount;

   public DiscountVisitor(double discountAmount) {
      if(discountAmount<0 || discountAmount>1.0)
         throw new IllegalArgumentException("discount value must be between 0.0 and 1.0");
      this.discountAmount = discountAmount;
   }

   public double visit(ShoppingCart shoppingCart){
      double totalPrice = 0.0;
      Iterator<FruitItem> fruitItemIterator = shoppingCart.itemsList();
      while(fruitItemIterator.hasNext())
         totalPrice += fruitItemIterator.next().totalPrice();
      return totalPrice * (1.0-discountAmount);
   }
}
```


35

Visitor Pattern Example Coupon Visitor

```
import java.util.Iterator;

public class CouponVisitor implements ShoppingCartVisitor {

   private final double couponValue;

   public CouponVisitor(double couponValue) {
      if(couponValue < 0)
         throw new IllegalArgumentException("coupon value must be non-negative");
      this.couponValue = couponValue;
   }

   public double visit(ShoppingCart shoppingCart){
      double totalPrice = 0.0;
      Iterator<FruitItem> fruitItemIterator = shoppingCart.itemsList();
      while(fruitItemIterator.hasNext())
         totalPrice += fruitItemIterator.next().totalPrice();
      if(totalPrice > couponValue)
         return totalPrice - couponValue;
      else
         return 0.0;
   }
}
```


36

Visitor Pattern Example
Test Class

```
public class VisitorExampleTest {

   public static void main(String[] args){

      ShoppingCart shoppingCart = new ShoppingCart();
      shoppingCart.addFruitItem(new Apple(8));
      shoppingCart.addFruitItem(new Kiwi(4));
      shoppingCart.addFruitItem(new Orange(10));

      ShoppingCartVisitor priceCalculator1 = new DiscountVisitor(0.2);
      ShoppingCartVisitor priceCalculator2 = new CouponVisitor(14.00);

      System.out.println(shoppingCart.accept(priceCalculator1));
      System.out.println(shoppingCart.accept(priceCalculator2));
   }
}
```


37