LBTMxCSCE

Luka Bostick

February 27, 2024

L B T M

Research and Development

0.1 Encapsulation

0.2 Polymorphism

0.3 Association

0.4 Aggregation

0.5 Composition

Inheritance

```java
public class Student {

  // Class fields/attributes:
  String name;
  int totalCredits, numOfClassesTaken;
  float gpa;
  String fieldOfStudy;
  enum Classification{Freshman, Sophomore,
 Junior, Senior};

  Classification classification;
  // Transcription Info

  public void finalGrade(String className,
 int credits, float grade)
  {
    // Well the classname and I guess all
        //of it shouldbe added to the not yet
        //existent transcript,
    // we do that stuff we can do given
        // the constraints

    numOfClassesTaken++;

    gpa=((gpa*totalCredits) + (credits * grade))/(totalCredits+
   numOfClassesTaken);

    totalCredits+=credits;
  }
}
```

0.6 Factory Method Pattern

1. Start by creating the main function

```java
public class Client {
public static void main(String[] args) {

}
}
```

2. Create the interface that defines our Object

```java
interface Animal{
  void displayBehavior();
}
```

3. Define the subclasses

4. The above inherits from the abstract AnimalFactory class

```java
class Dog implements Animal{

public Dog() {
  System.out.println("\nA dog is created.");
}
public void displayBehavior() {
  System.out.println("woof");
  System.out.println("woof!");
}
}

class Tiger implements Animal
{
  public Tiger() {
    System.out.println("\nA tiger is created.");
  }
  public void displayBehavior() {
    System.out.println("rar");
    System.out.println("Chomp");
  }
}
```

5. Created another inheritance hierarchy

```java

abstract class AnimalFactory{
  // This is the "factory method"
  //  instantiation -> subclass

  protected abstract Animal createAnimal();

}


class DogFactory extends AnimalFactory
{

  @Override
  protected Animal createAnimal(/*create+return dog instance*/) {
    return new Dog();
  }

}

class TigerFactory extends AnimalFactory
{

  @Override
  protected Animal createAnimal(/*create+return tiger instance*/) {
    return new Tiger();
  }

}
```

6. Update the client

```java
public class Client {
public static void main(String[] args) {
  System.out.println("Factory Method Pattern demo");
  AnimalFactory factory;
  Animal animal;

  // create a tiger and display its behavior

  factory = new TigerFactory();
  animal = factory.createAnimal();


  // create a dog and display its behavior

  factory = new DogFactory();
  animal = factory.createAnimal();
  animal.displayBehavior();
}
}
```

Complete on your own

1. Modified a factory's constructor method signature to produce a variation of a subclass.

```java
public Dog(String color) {
  System.out.println("\nA dog with "+ color+" color is created.");
}
public void displayBehavior() {
  System.out.println("woof");
  System.out.println("woof!");
}
}

class Tiger implements Animal
{
  public Tiger(String color) {
    System.out.println("\nA tiger with " +color+ "color is created.");
  }
  public void displayBehavior() {
    System.out.println("rar");
    System.out.println("Chomp");
  }
}


```

2. Append the color attribute to the function call of the abstract Animal Factory Class by creating an alternative constructor.

```
abstract class AnimalFactory{
  // This is the "factory method"
  //  instantiation -> subclass

  public void createAndDisplayAnimal(String color /* Factory method defers
    init to subclass*/) {
    Animal animal;
    animal=createAnimal(color);
    animal.displayBehavior();
  }

  protected abstract Animal createAnimal(String color);

}

```

3. Update subclass factory constructors

```
class DogFactory extends AnimalFactory
{

  @Override
  protected Animal createAnimal(String color/*create+return dog instance*/)
     {
    return new Dog(color);
  }

}

class TigerFactory extends AnimalFactory
{

  @Override
  protected Animal createAnimal(String color/*create+return tiger instance
    */) {
    return new Tiger(color);
  }

}

```

4. Update the Client

```java
    public class Client {
public static void main(String[] args) {
  System.out.println("Factory Method Pattern v2 demo");
  AnimalFactory factory;
  Animal animal;

  // create a tiger and display its behavior

  factory = new TigerFactory();
  animal = factory.createAnimal("Green");


  // create a dog and display its behavior

  factory = new DogFactory();
  animal = factory.createAnimal("Blue");
  animal.displayBehavior();
}
}
```

0.7 Bridge Pattern

```java

//PG 311
public class Bridge_Pattern {
public static void main(String[] args) {
  System.out.println("***Bridge Pattern Demo.***");

  System.out.println("Verifying the market price of a television.");
  // Verifying the online price
  ElectronicItem eItem = new Television(new OnlinePrice());
  eItem.showPriceDetail();
  //System.out.println("----------");

  // Verifying the offline/showroom price

  eItem = new Television(new ShowroomPrice());
  eItem.showPriceDetail();
}
}

/* GoF Definition
 * It decouples an abstraction from its implementation so that the two can vary
   independently.
 *
 */

// PriceType.java
interface PriceType{
  void displayProductPrice(String product, double cost);
}

// OnlinePrice.java // This is ConcreteImplementor-1

class OnlinePrice implements PriceType{

  @Override
  public void displayProductPrice(String product, double cost) {
    System.out.println("The " + product + " 's online price is $" + cost*.9);

  }
}

// ShowroomPrice.java // This is ConcreteImplementor-2

class ShowroomPrice implements PriceType{

  @Override
  public void displayProductPrice(String product, double cost) {
    System.out.println("The " + product + "'s showroom price is $" + cost);

  }
}

// ElectronicItem.java
abstract class ElectronicItem{
  // Composition - implementor
  protected PriceType priceType;
  protected ElectronicItem(PriceType priceType) {
    this.priceType=priceType;
  }

  /*
   * This method implementation specific. We'll use an
   * implementor object to invoke this method.
   */
  protected abstract void showPriceDetail();
}

// Television.java
class Television extends ElectronicItem{
  /*
   * Implementation specific:
   * Delegating the task
   * to the Implementor object.
   */

  String productType;
  double cost;

  public Television(PriceType priceType) {
    super(priceType);
    this.productType="television";
    this.cost=2000;
  }

  @Override
  protected void showPriceDetail() {
    priceType.displayProductPrice(productType,cost);
  }

  /**
   * Implementation specific:
   * We are delegating the implementation
   * to the Implementor object.
   */

}
```

Complete on your own

```

```

```java
//PG 311
public class Bridge_Pattern {
public static void main(String[] args) {
  System.out.println("***Bridge Pattern Demo.***");

  System.out.println("Verifying the market price of a television.");
  // Verifying the online price
  ElectronicItem eItem = new Television(new OnlinePrice());
  eItem.showPriceDetail();
  //System.out.println("----------");

  // Verifying the offline/showroom price

  eItem = new Television(new ShowroomPrice());
  eItem.showPriceDetail();
}
}

/* GoF Definition
 * It decouples an abstraction from its implementation so that the two can vary
     independently.
 *
 */

// PriceType.java
interface PriceType{
  void displayProductPrice(String product, double cost);
}

// OnlinePrice.java // This is ConcreteImplementor-1

class OnlinePrice implements PriceType{

  @Override
  public void displayProductPrice(String product, double cost) {
    System.out.println("The " + product + " 's online price is $" + cost*.9);

  }
}

// ShowroomPrice.java // This is ConcreteImplementor-2

class ShowroomPrice implements PriceType{

  @Override
  public void displayProductPrice(String product, double cost) {
    System.out.println("The " + product + "'s showroom price is $" + cost);

  }
}

// ElectronicItem.java
abstract class ElectronicItem{
  // Composition - implementor
  protected PriceType priceType;
  protected ElectronicItem(PriceType priceType) {
    this.priceType=priceType;
  }

  /*
   * This method implementation specific. We'll use an
   * implementor object to invoke this method.
   */
  protected abstract void showPriceDetail();
}

// Television.java
class Television extends ElectronicItem{
  /*
   * Implementation specific:
   * Delegating the task
   * to the Implementor object.
   */

  String productType;
  double cost;

  public Television(PriceType priceType) {
    super(priceType);
    this.productType="television";
    this.cost=2000;
  }

  @Override
  protected void showPriceDetail() {
    priceType.displayProductPrice(productType,cost);
  }

  /**
   * Implementation specific:
   * We are delegating the implementation
   * to the Implementor object.
   */

}
```

0.8 Observer Pattern

```java

import java.util.ArrayList;
import java.util.List;
// Client.java

public class ObserverPattern {

  public static void main(String[] args) {
    System.out.println("***Observer Pattern Desmonstration.***\n");
    // We have 4 different observers.

    Observer roy = new Employee("Roy");
    Observer kevin = new Employee("kevin");
    Observer bose = new Employee("bose");
    Observer jacklin = new Employee("jacklin");


    Company abcLtd = new SpecificCompany("ABC Ltd. ");
    System.out.println("Working with the company: Abc Ltd.");
    // Registering the observer - Roy, Kevin, Bose
    abcLtd.register(roy);
    abcLtd.register(kevin);
    abcLtd.register(bose);
    System.out.println(" ABC Ltd.'s current stock price is $5.");
    abcLtd.setStockPrice(5);
    System.out.println("-----");

    // Kevin doesn'y want to get further notification.
    System.out.println("\nABC Ltd. is removing Kevin from the observer list now
   ");

    abcLtd.unRegister(kevin);
    // No notigication is sent to kevin any more.

    System.out.println("\n ABC Ltd.'s new stock price is $50.");
    abcLtd.setStockPrice(50);
    System.out.println("-----");

    System.out.println("\nKevin registers again to get notification from ABC
   Ltd.");

    abcLtd.register(kevin);

    System.out.println("\nKevin registers again to get noticication from ABC
   Ltd.");


    abcLtd.register(kevin);

    System.out.println("\n ABC Ltd.'s new Stock price is $100.");
    abcLtd.setStockPrice(100);
    System.out.println("-----");

    System.out.println("\n Working with another company: XYZ Co.");

    // Creating another company
    Company xyzCo = new SpecificCompany("XYZ Co.");

    // Registering the observes-Roy and Jacklin
    xyzCo.register(roy);
    xyzCo.register(jacklin);
    System.out.println("\nXYZ Co.'s new stock price is $500.");
    xyzCo.setStockPrice(500);
  }
}
/**GoF Definition
 * It defiances a one-to-many dependency between objects so that when one
   object changes
 * state, all its dependents are notified and updated automatically.
 *
 */

// Observer.java
interface Observer {
  void getNotification(Company company);
  void registerTo(Company company);
  void unregisterFrom(Company company);
  String getObserverName();
}

// Employee.java
// Observer type-1: these are employees

class Employee implements Observer{
  String nameOfObserver;

  public Employee(String name) {
    this.nameOfObserver = name;
  }
  public void getNotification(Company company) {
    System.out.println(nameOfObserver+" has recieved an alert from " + company.getName());
    System.out.println("The current stock price is:$" +
    company.getStockPrice());
  }

  public String getObserverName() {
    return nameOfObserver;
  }
  @Override
  public void registerTo(Company company) {
    company.register(this);
    System.out.println(this.nameOfObserver+"registered himself/herself to " + company.getName());

  }
  @Override
  public void unregisterFrom(Company company) {
    company.unRegister(this);
    System.out.println(this.nameOfObserver+" unregistered himself/herself from "+ company.getName());
  }

}

//Customer.java
//Observer type-2: these are customers
class Customer implements Observer{

  String nameOfObserver;

  public Customer(String name) {
    this.nameOfObserver = name;
  }

  @Override
  public void getNotification(Company company) {
    System.out.println(nameOfObserver + "has received an alert from " + company.getName());
    System.out.println("The current stock price is:$" + company.getStockPrice());

  }


  @Override
  public String getObserverName() {
    return nameOfObserver;
  }

  @Override
  public void registerTo(Company company) {
    company.register(this);
    System.out.println(this.nameOfObserver+"registered himself/herself to " + company.getName());

  }
  @Override
  public void unregisterFrom(Company company) {
    company.unRegister(this);
    System.out.println(this.nameOfObserver+" unregistered himself/herself from "+ company.getName());
  }

}




// Company.java

abstract class Company{
  List<Observer> observerList = new ArrayList<Observer>();

  // Name of the subject
  private String name;

  public Company(String name) {
    this.name = name;
  }
  public String getName() {
    return this.name;
  }

  // For the stock price
  private int stockPrice;

  public int getStockPrice() {
    return this.stockPrice;
  }

  public void setStockPrice(int stockPrice) {
    this.stockPrice=stockPrice;
    // The stock price is changed.
    // So, notify observer(s).
    notifyRegisteredUsers();
  }

  // To register an observer
  abstract void register(Observer o);

  // To unregister an observer
  abstract void unRegister(Observer o);

  // to notify registered users
  abstract void notifyRegisteredUsers();
}
// SpecificCompany.java

class SpecificCompany extends Company{
  public SpecificCompany(String name) {
    super(name);
  }

  @Override
  void register(Observer anObserver) {
    observerList.add(anObserver);
    System.out.println(this.getName()+" register " + anObserver.getObserverName
   ());
  }

  void unRegister(Observer anObserver) {
    observerList.remove(anObserver);
    System.out.println(this.getName()+" unregisters " + anObserver.
   getObserverName());
```

```
    }
    // Notify all registered observers.
    @Override
    void notifyRegisteredUsers() {
      for(Observer observer: observerList) {
        observer.getNotification(this);
      }

    }
}
```