

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

```
1 public class Student {  
2  
3     // Class fields/attributes:  
4     String name;  
5     int totalCredits, numOfClassessTaken;  
6     float gpa;  
7     String fieldOfStudy;  
8     enum Classification{Freshman, Sophomore,  
9     Junior, Senior};  
10  
11     Classification classification;  
12     // Transcription Info  
13 }
```

```
14 public void finalGrade(String className,
15 int credits, float grade)
16 {
17     // Well the classname and I guess all
18     //of it shouldbe added to the not yet
19     //existent transcript,
20     // we do that stuff we can do given
21     // the constraints
22
23     numOfClassessTaken++;
24
25     gpa=((gpa*totalCredits) + (credits * grade))/(totalCredits+
26 numOfClassessTaken);
27
28     totalCredits+=credits;
29 }
```

0.6 Factory Method Pattern

1. Start by creating the main function

```
1 public class Client {
2     public static void main(String[] args) {
3
4     }
5 }
```

2. Create the interface that defines our Object

```
1 interface Animal{
2     void displayBehavior();
3 }
```

3. Define the subclasses

4. The above inherits from the abstract AnimalFactory class

```
1 class Dog implements Animal{
2
3     public Dog() {
4         System.out.println("\nA dog is created.");
5     }
6     public void displayBehavior() {
7         System.out.println("woof");
8         System.out.println("woof!");
9     }
10 }
11
12 class Tiger implements Animal
13 {
14     public Tiger() {
15         System.out.println("\nA tiger is created.");
16     }
17     public void displayBehavior() {
18         System.out.println("rar");
19         System.out.println("Chomp");
20     }
21 }
```

5. Created another inheritance hierarchy

```

1
2 abstract class AnimalFactory{
3     // This is the "factory method"
4     // instantiation -> subclass
5
6     protected abstract Animal createAnimal();
7
8 }
9
10
11 class DogFactory extends AnimalFactory
12 {
13
14     @Override
15     protected Animal createAnimal(/*create+return dog instance*/) {
16         return new Dog();
17     }
18 }
19
20
21 class TigerFactory extends AnimalFactory
22 {
23
24     @Override
25     protected Animal createAnimal(/*create+return tiger instance*/) {
26         return new Tiger();
27     }
28 }
29 }

```

6. Update the client

```

1 public class Client {
2     public static void main(String[] args) {
3         System.out.println("Factory Method Pattern demo");
4         AnimalFactory factory;
5         Animal animal;
6
7         // create a tiger and display its behavior
8
9         factory = new TigerFactory();
10        animal = factory.createAnimal();
11
12
13        // create a dog and display its behavior
14
15        factory = new DogFactory();
16        animal = factory.createAnimal();
17        animal.displayBehavior();
18    }
19 }

```

Complete on your own

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

```

1 public Dog(String color) {
2     System.out.println("\nA dog with "+ color+" color is created.");
3 }
4 public void displayBehavior() {
5     System.out.println("woof");
6 }

```

```
6     System.out.println("woof!");
7 }
8 }
9
10 class Tiger implements Animal
11 {
12     public Tiger(String color) {
13         System.out.println("\nA tiger with " +color+ "color is created.");
14     }
15     public void displayBehavior() {
16         System.out.println("rar");
17         System.out.println("Chomp");
18     }
19 }
20
21
```

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

```
1 abstract class AnimalFactory{
2     // This is the "factory method"
3     // instantiation -> subclass
4
5     public void createAndDisplayAnimal(String color /* Factory method defers
6         init to subclass*/) {
7         Animal animal;
8         animal=createAnimal(color);
9         animal.displayBehavior();
10    }
11
12    protected abstract Animal createAnimal(String color);
13 }
14
```

3. Update subclass factory constructors

```
1 class DogFactory extends AnimalFactory
2 {
3
4     @Override
5     protected Animal createAnimal(String color/*create+return dog instance*/)
6     {
7         return new Dog(color);
8     }
9 }
10
11 class TigerFactory extends AnimalFactory
12 {
13
14     @Override
15     protected Animal createAnimal(String color/*create+return tiger instance
16         */) {
17         return new Tiger(color);
18     }
19 }
20
```

4. Update the Client

```

1     public class Client {
2 public static void main(String[] args) {
3     System.out.println("Factory Method Pattern v2 demo");
4     AnimalFactory factory;
5     Animal animal;
6
7     // create a tiger and display its behavior
8
9     factory = new TigerFactory();
10    animal = factory.createAnimal("Green");
11
12
13    // create a dog and display its behavior
14
15    factory = new DogFactory();
16    animal = factory.createAnimal("Blue");
17    animal.displayBehavior();
18 }
19 }

```

0.7 Bridge Pattern

```

1
2 //PG 311
3 public class Bridge_Pattern {
4 public static void main(String[] args) {
5     System.out.println("***Bridge Pattern Demo***");
6
7     System.out.println("Verifying the market price of a television.");
8     // Verifying the online price
9     ElectronicItem eItem = new Television(new OnlinePrice());
10    eItem.showPriceDetail();
11    //System.out.println("-----");
12
13    // Verifying the offline/showroom price
14
15    eItem = new Television(new ShowroomPrice());
16    eItem.showPriceDetail();
17 }
18 }
19
20 /* GoF Definition
21 * It decouples an abstraction from its implementation so that the two can vary
22   independently.
23 *
24 */
25 // PriceType.java
26 interface PriceType{
27     void displayProductPrice(String product, double cost);
28 }
29
30 // OnlinePrice.java // This is ConcreteImplementor-1
31
32 class OnlinePrice implements PriceType{
33
34     @Override
35     public void displayProductPrice(String product, double cost) {
36         System.out.println("The " + product + "'s online price is $" + cost*.9);
37     }
38 }

```

```
39 }
40
41 // ShowroomPrice.java // This is ConcreteImplementor-2
42
43 class ShowroomPrice implements PriceType{
44
45     @Override
46     public void displayProductPrice(String product, double cost) {
47         System.out.println("The " + product + "'s showroom price is $" + cost);
48     }
49 }
50
51
52 // ElectronicItem.java
53 abstract class ElectronicItem{
54     // Composition - implementor
55     protected PriceType priceType;
56     protected ElectronicItem(PriceType priceType) {
57         this.priceType=priceType;
58     }
59
60     /*
61      * This method implementation specific. We'll use an
62      * implementor object to invoke this method.
63      */
64     protected abstract void showPriceDetail();
65 }
66
67 // Television.java
68 class Television extends ElectronicItem{
69     /*
70      * Implementation specific:
71      * Delegating the task
72      * to the Implementor object.
73      */
74
75     String productType;
76     double cost;
77
78     public Television(PriceType priceType) {
79         super(priceType);
80         this.productType="television";
81         this.cost=2000;
82     }
83
84     @Override
85     protected void showPriceDetail() {
86         priceType.displayProductPrice(productType, cost);
87     }
88
89     /**
90      * Implementation specific:
91      * We are delegating the implementation
92      * to the Implementor object.
93      */
94
95 }
```

Complete on your own

```

2 //PG 311
3 public class Bridge_Pattern {
4 public static void main(String[] args) {
5     System.out.println("***Bridge Pattern Demo***");
6
7     System.out.println("Verifying the market price of a television.");
8     // Verifying the online price
9     ElectronicItem eItem = new Television(new OnlinePrice());
10    eItem.showPriceDetail();
11    //System.out.println("-----");
12
13    // Verifying the offline/showroom price
14
15    eItem = new Television(new ShowroomPrice());
16    eItem.showPriceDetail();
17 }
18 }
19
20 /* GoF Definition
21  * It decouples an abstraction from its implementation so that the two can vary
22   * independently.
23  */
24
25 // PriceType.java
26 interface PriceType{
27     void displayProductPrice(String product, double cost);
28 }
29
30 // OnlinePrice.java // This is ConcreteImplementor-1
31
32 class OnlinePrice implements PriceType{
33
34     @Override
35     public void displayProductPrice(String product, double cost) {
36         System.out.println("The " + product + "'s online price is $" + cost*.9);
37     }
38 }
39
40
41 // ShowroomPrice.java // This is ConcreteImplementor-2
42
43 class ShowroomPrice implements PriceType{
44
45     @Override
46     public void displayProductPrice(String product, double cost) {
47         System.out.println("The " + product + "'s showroom price is $" + cost);
48     }
49 }
50
51
52 // ElectronicItem.java
53 abstract class ElectronicItem{
54     // Composition - implementor
55     protected PriceType priceType;
56     protected ElectronicItem(PriceType priceType) {
57         this.priceType=priceType;
58     }
59
60     /*
61     * This method implementation specific. We'll use an
62     * implementor object to invoke this method.
63     */

```

```
64     protected abstract void showPriceDetail();
65 }
66
67 // Television.java
68 class Television extends ElectronicItem{
69     /*
70      * Implementation specific:
71      * Delegating the task
72      * to the Implementor object.
73      */
74
75     String productType;
76     double cost;
77
78     public Television(PriceType priceType) {
79         super(priceType);
80         this.productType="television";
81         this.cost=2000;
82     }
83
84     @Override
85     protected void showPriceDetail() {
86         priceType.displayProductPrice(productType, cost);
87     }
88
89     /**
90      * Implementation specific:
91      * We are delegating the implementation
92      * to the Implementor object.
93      */
94 }
95 }
```

0.8 Observer Pattern

```
1
2 import java.util.ArrayList;
3 import java.util.List;
4 // Client.java
5
6 public class ObserverPattern {
7
8     public static void main(String[] args) {
9         System.out.println("***Observer Pattern Desmonstration***\n");
10        // We have 4 different observers.
11
12        Observer roy = new Employee("Roy");
13        Observer kevin = new Employee("kevin");
14        Observer bose = new Employee("bose");
15        Observer jacklin = new Employee("jacklin");
16
17
18        Company abcLtd = new SpecificCompany("ABC Ltd. ");
19        System.out.println("Working with the company: Abc Ltd.");
20        // Registering the observer - Roy, Kevin, Bose
21        abcLtd.register(roy);
22        abcLtd.register(kevin);
23        abcLtd.register(bose);
24        System.out.println(" ABC Ltd.'s current stock price is $5.");
25        abcLtd.setStockPrice(5);
26        System.out.println("-----");
27    }
28 }
```

```

27
28 // Kevin doesn't want to get further notification.
29 System.out.println("\nABC Ltd. is removing Kevin from the observer list now
");
30
31 abcLtd.unregister(kevin);
32 // No notification is sent to kevin any more.
33
34 System.out.println("\n ABC Ltd.'s new stock price is $50.");
35 abcLtd.setStockPrice(50);
36 System.out.println("-----");
37
38 System.out.println("\nKevin registers again to get notification from ABC
Ltd.");
39
40 abcLtd.register(kevin);
41
42 System.out.println("\nKevin registers again to get notification from ABC
Ltd.");
43
44
45 abcLtd.register(kevin);
46
47 System.out.println("\n ABC Ltd.'s new Stock price is $100.");
48 abcLtd.setStockPrice(100);
49 System.out.println("-----");
50
51 System.out.println("\n Working with another company: XYZ Co.");
52
53 // Creating another company
54 Company xyzCo = new SpecificCompany("XYZ Co.");
55
56 // Registering the observes-Roy and Jacklin
57 xyzCo.register(roy);
58 xyzCo.register(jacklin);
59 System.out.println("\nXYZ Co.'s new stock price is $500.");
60 xyzCo.setStockPrice(500);
61 }
62 }
63 /**GoF Definition
64  * It defiances a one-to-many dependency between objects so that when one
65  * object changes
66  * state, all its dependents are notified and updated automatically.
67  */
68
69 // Observer.java
70 interface Observer {
71     void getNotification(Company company);
72     void registerTo(Company company);
73     void unregisterFrom(Company company);
74     String getObserverName();
75 }
76
77 // Employee.java
78 // Observer type-1: these are employees
79
80 class Employee implements Observer{
81     String nameOfObserver;
82
83     public Employee(String name) {
84         this.nameOfObserver = name;
85     }

```

```
86 public void getNotification(Company company) {
87     System.out.println(nameOfObserver+" has recieved an alert from " + company.
88     getName());
89     System.out.println("The current stock price is:$" +
89     company.getStockPrice());
90 }
91
92 public String getObserverName() {
93     return nameOfObserver;
94 }
95 @Override
96 public void registerTo(Company company) {
97     company.register(this);
98     System.out.println(this.nameOfObserver+"registered himself/herself to " +
98     company.getName());
99 }
100
101 @Override
102 public void unregisterFrom(Company company) {
103     company.unregister(this);
104     System.out.println(this.nameOfObserver+" unregistered himself/herself from
104     "+ company.getName());
105 }
106
107 }
108
109 //Customer.java
110 //Observer type-2: these are customers
111 class Customer implements Observer{
112
113     String nameOfObserver;
114
115     public Customer(String name) {
116         this.nameOfObserver = name;
117     }
118
119     @Override
120     public void getNotification(Company company) {
121         System.out.println(nameOfObserver + "has received an alert from " + company
121         .getName());
122         System.out.println("The current stock price is:$" + company.getStockPrice()
122         );
123     }
124
125
126
127     @Override
128     public String getObserverName() {
129         return nameOfObserver;
130     }
131
132
133     @Override
134     public void registerTo(Company company) {
135         company.register(this);
136         System.out.println(this.nameOfObserver+"registered himself/herself to " +
136         company.getName());
137     }
138
139     @Override
140     public void unregisterFrom(Company company) {
141         company.unregister(this);
142         System.out.println(this.nameOfObserver+" unregistered himself/herself from
142         "+ company.getName());
```

```

142     }
143
144 }
145
146
147
148
149 // Company.java
150
151 abstract class Company{
152     List<Observer> observerList = new ArrayList<Observer>();
153
154     // Name of the subject
155     private String name;
156
157     public Company(String name) {
158         this.name = name;
159     }
160     public String getName() {
161         return this.name;
162     }
163
164     // For the stock price
165     private int stockPrice;
166
167     public int getStockPrice() {
168         return this.stockPrice;
169     }
170
171     public void setStockPrice(int stockPrice) {
172         this.stockPrice=stockPrice;
173         // The stock price is changed.
174         // So, notify observer(s).
175         notifyRegisteredUsers();
176     }
177
178     // To register an observer
179     abstract void register(Observer o);
180
181     // To unregister an observer
182     abstract void unregister(Observer o);
183
184     // to notify registered users
185     abstract void notifyRegisteredUsers();
186 }
187 // SpecificCompany.java
188
189 class SpecificCompany extends Company{
190     public SpecificCompany(String name) {
191         super(name);
192     }
193
194     @Override
195     void register(Observer anObserver) {
196         observerList.add(anObserver);
197         System.out.println(this.getName()+" register " + anObserver.getObserverName());
198     }
199
200     void unregister(Observer anObserver) {
201         observerList.remove(anObserver);
202         System.out.println(this.getName()+" unregisters " + anObserver.getObserverName());

```

```
203     }
204     // Notify all registered observers.
205     @Override
206     void notifyRegisteredUsers() {
207         for (Observer observer: observerList) {
208             observer.getNotification(this);
209         }
210     }
211 }
212 }
```