Software Engineering - SWEN6304

Engineering and Technology DEPARTMENT

Software Architecture and Design

Design of Shopping Mall System using Design Patterns

Dr. Yousef A. Hassouneh

Made by:
Alaa' Omar, 1185472@birzeit.edu.ps

Contents

1 Introduction **2**

2 Project functional and nonfunctional Requirements **3**

3 Project Source Code Structure and Hierarchy **3**

4 Design Patterns Used **3**
- 4.1 Abstract Factory 3
- 4.2 Singleton Design Pattern 5
- 4.3 Iterator Design Pattern 6
- 4.4 Observer Design Pattern 7
- 4.5 Composite Design Pattern 9

Listings

1 Factory Design Pattern Sample Code 5
2 Singleton Design Pattern Sample Code 5
3 Iterator Design Pattern CreateIerator Sample Code 6
4 Iterator Design Pattern Concrete implementation sample Code 7
5 Observer Design Pattern Sample Code 7
6 Composite Design Pattern Sample Code 9

List of Figures

1 Mall Store Project Modified Class diagram 2
2 Project Source Code Structure and Hierarchy 4
3 Abstract Factory Design Pattern Class Diagram 4
4 Iterator Design Pattern 6
5 Observer Design Pattern Class Diagram 8
6 Composite Design Pattern Class Diagram 9

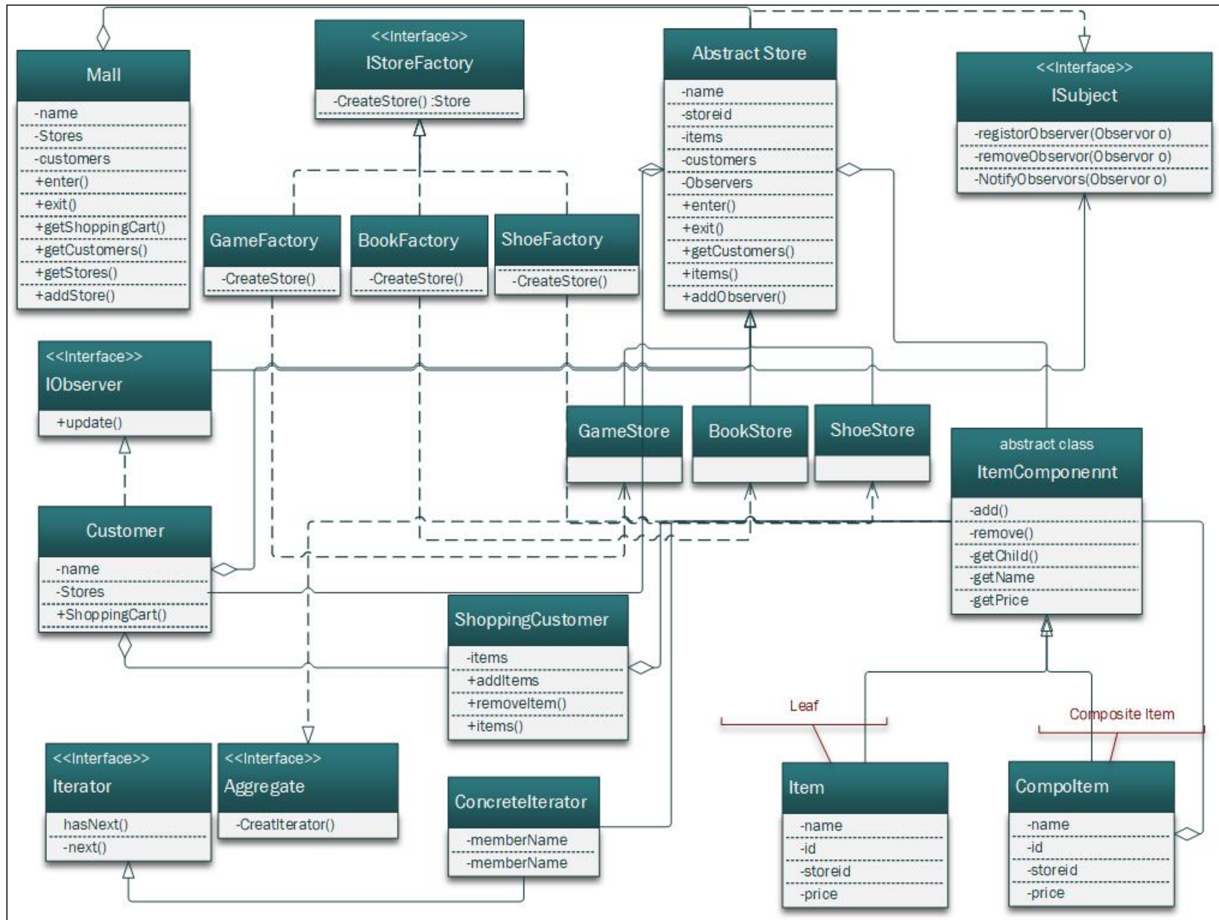

Figure 1: Mall Store Project Modified Class diagram

1 Introduction

This Project represents a redesign to the original Project class diagram, the new design is implemented using design patterns, that provides the quality attribute "design for change", depending on object oriented programming four principles (Abstraction, Encapsulation, Inheritance, Polymorphism), side by side with the SOLID Design Principles. The Design Patterns used in this project are (Abstract Factory Design Pattern,Singleton Design Pattern, Observer Design Pattern, Iterator Design Pattern, Composite Design Pattern), the article contains detailed sections about each design pattern usage in the project. In the section 2 lists the functional and non functional requirements for the project, section 3 shows the source code structure and hierarchy, section 4 talks about each design pattern used in the project in a separate subsection.

2 Project functional and nonfunctional Requirements

Nonfunctional Requirements:

- Design For Change

Functional Requirements:

- The Mall contains one or more stores.
- Many customers can enter the mall.
- Allow only one instance of any Store type created.
- Any Customer can enter any store.
- The customer must register for Items sale update for any store.
- The customer can unsubscribe for updates in any store.
- The customers observing any store should be notified automatically for updates.

3 Project Source Code Structure and Hierarchy

The Figure 2 illustrates The Source Code hierarchy, every design pattern source files are combined together in a folder with the design pattern name, the first folder which is the default package has the client code **myMall.java**, the **CompositeDP** holds the **ComboItem.java** wich is the concrete class that repersents the composite item, the **ItemComponent** represents the abstract class for the **comboItem.java**, and **ComboItem.java**, while the leaf item is in the **SharedClasses** package, refer to figure 6. The package **FactoryDP** contains all items of the abstract factory design pattern, strating from the interface **IStoreFactory**, ending with concrete classes **(ShoeStore, GameStore, BookStore)**, figure 2 illustrates the relations between the classes of abstract factory. **ObserverDP** package contains the interfaces needed for the observer design pattern, while the concrete subject is the concrete stores, and the concrete Observer is the customer class which is in the **SharedClasses**, refer figure 5 for the complete structur. **SharedClasses** package contains class that are used in more than one design pattern (item, customer), and other classes (mall, shopping cart).

4 Design Patterns Used

4.1 Abstract Factory

Abstract Factory Design Pattern which is of kind structural design pattern, has been used to create new Stores, the abstract factory is used to create a family of products, in this project, I introduced one line of products, which is Store, with three Concrete stores (concrete products) that inherits the abstract stores, so this design pattern can be Considered as two in one design pattern,the figure illustrates the use of this design in the project:

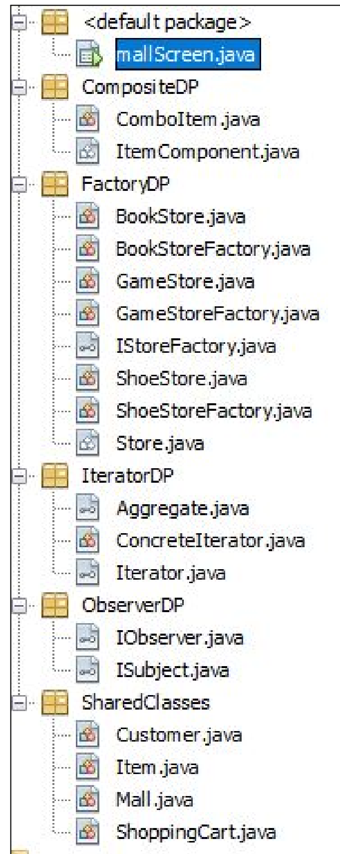

Figure 2: Project Source Code Structure and Hierarchy

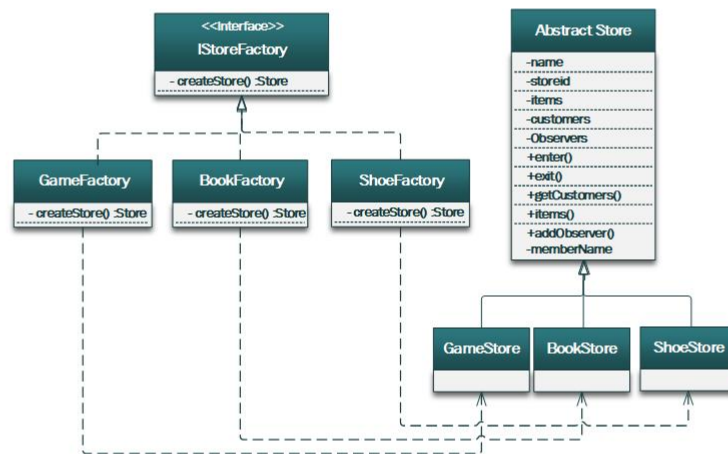

Figure 3: Abstract Factory Design Pattern Class Diagram

- **Abstract Store** represents an abstract line of products, it has three concrete classes **(shoeStore, GameStore, BookStore)**, A new product of line can be added easily by adding a new abstract line of product that has as many concrete classes as needed.
- **GameStor, ShoeStore, BookStore** represents concrete classes that inherits from Abstract Store.
- **IStoreFactory** represents the interface for the Store factory in which each product of line must be represented by a factory method.
- **GameFactory, BookFactory, ShoeFactory** represent a concrete factories that implements the IStoreFactory **IStoreFactory** and overrides the **CreateFactory()** method that returns object of type Store, each of these concrete classes has a direct association with the **concrete Stores (GameStore, BookStore, ShoeStore)**. This dependency means that each concrete factory is responsible for creating each corespondent concrete Stores.

The Idea of using Abstract factory is to decouple the client from knowing about the concrete classes by providing an interface for creating family of related products, which is **IStoreFactory** in this project. Using abstract factory provides the ease of adding new concrete classes, related to line of product without bothering clients, which intern leads to maintainable systems that are designed for change, Furthermore it is important to mention that some of the solid design principles are used in this design pattern, **Open-Closed Principle OCP**, and **Inversion Dependency Principle**.

```
1 /*Create A new Game Store*/
2         GameStoreFactory gFactory = GameStoreFactory.getFactoryInstance();
3         Store gStore = gFactory.createStore();
```

Listing 1: Factory Design Pattern Sample Code

4.2 Singleton Design Pattern

Singleton Design Pattern which is categorized as creational design patterns, it is used to allow only Creation of one instance of each Factory, and provides a global access to it. It Allows only one instance of any factory type created.

```
1  private volatile static GameStoreFactory instance = null;
2      private GameStoreFactory() {}
3        public static GameStoreFactory getFactoryInstance() {
4          GameStoreFactory localInstance = GameStoreFactory.instance;
5          if (localInstance == null) {
6              synchronized (GameStoreFactory.class) {
7                  localInstance = GameStoreFactory.instance;
8                  if (localInstance == null) {
9                      return new GameStoreFactory(); }
10                 }
11             }
12         return localInstance;
13     }
```

Listing 2: Singleton Design Pattern Sample Code

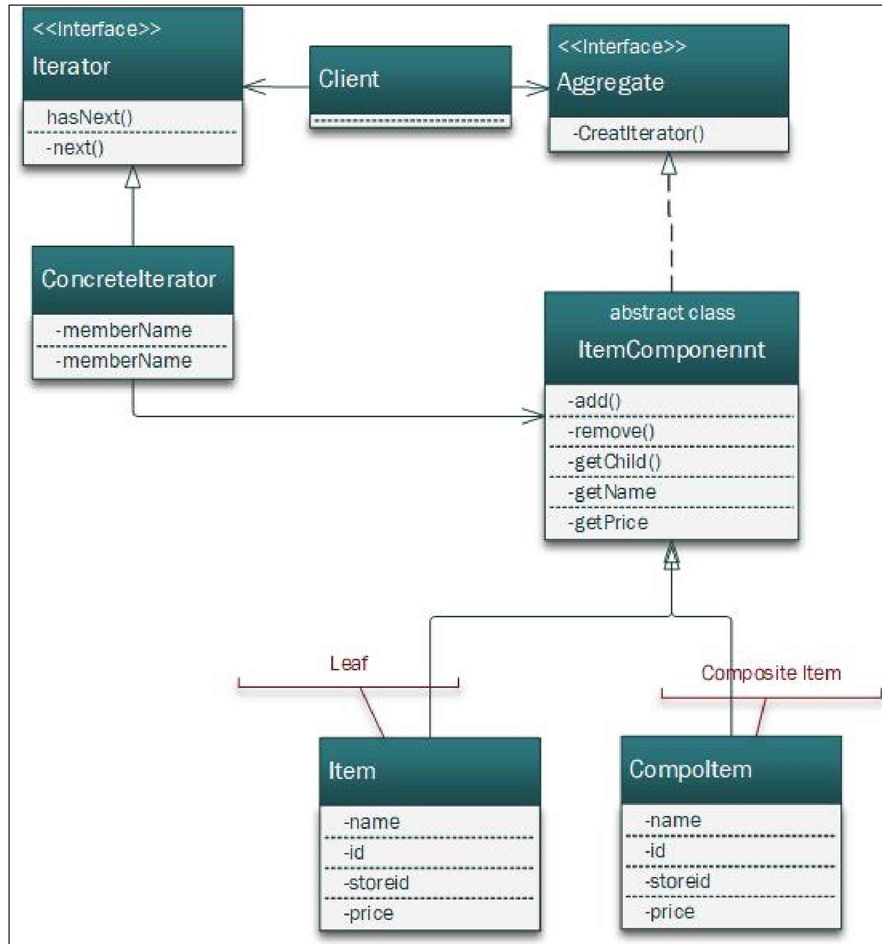

Figure 4: Iterator Design Pattern

4.3 Iterator Design Pattern

The Iterator Pattern provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation. The Iterator interface is ready to use in Java JDK, under the **java.util** library. All The enumeration Methods in this project are implemented using Iterator design pattern. The next code will be directly imported from a file:

```
@Override
    public IteratorDP.Iterator CreateIterator() {

    return new ConcreteIterator(itemComponents);
    }
```

Listing 3: Iterator Design Pattern CreateIerator Sample Code

the hasNext(), Next() implementation on the concrete Iterator

```
    @Override
    public Object next() {
        ItemComponent menuItem = items.get(position);
        position = position + 1;
        return menuItem;
    }

    @Override
    public boolean hasNext() {
        if (position >= items.size() || items.get(position) == null) {
            return false;
        } else {
            return true;
        }
    }
```

Listing 4: Iterator Design Pattern Concrete implementation sample Code

4.4 Observer Design Pattern

Observer Design Pattern categorized as behavioral design pattern defines a one to many relationship between objects, that's when an object (Subject) changes its state, all dependent objects (Observers) are notified and updated automatically.
in the figure below, the observer design patterns in our project consist of tow main interfaces as listed:

- IObserver interface that contains a prototype for update() method, that gets called when the Subject (Observee) state changed.
- Concrete Observer class that implements the IObserver interface, which is in our project the customer, it override the update() method. Each observer registers with a concrete subject (concrete Store Classes in our project) to receives updates includes the new item added Name and Price.
- ISubject interface, which is used by other objects (observers) to register as observers, or to be removed from observers list, when an object is resisted as an observers, it is then notified automatically.
- Concert Subjects are our Concrete Stores (GameStore, BookStore, ShoeStore). The concrete Store implements the ISubject interface, and override the methods.

All Concrete Stores are observable objects. The Customer (Concert Subject) in this projects needs to be notified whenever a new item is added for sale in the store the customer is registered in as an observer. Each Store Can has many observers (Customers).

```
@Override
    public void notifyObservers() {
        int index = items.size() - 1;
        ItemComponent itm = (ItemComponent) items.get(index);
        Iterator<IObserver> ObserverIterator = observers.iterator();
```

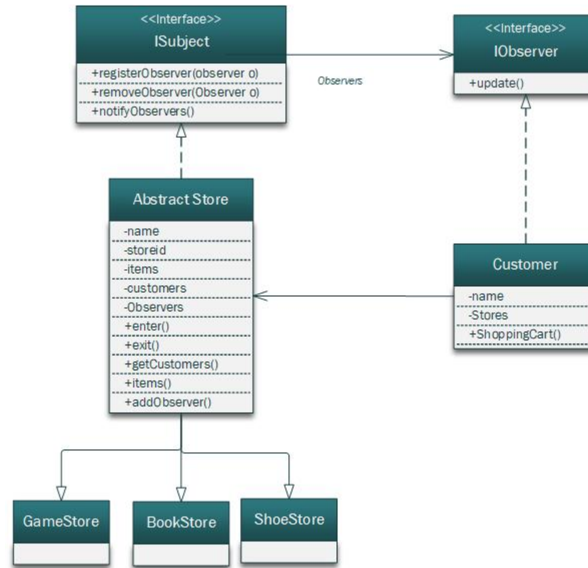

Figure 5: Observer Design Pattern Class Diagram

```
6          while (ObserverIterator.hasNext()) {
7              ObserverIterator.next().update(itm);
8          }
9      }
10     public void ItemsForSaleAdded() {
11         notifyObservers();
12     }
13     @Override
14     public void addItem(ItemComponent itm) {
15         items.add(itm);
16         System.out.println("New Item added: " + itm.getName() + " To Store: " + this.getStoreName());
17         ItemsForSaleAdded();
18     }
```

Listing 5: Observer Design Pattern Sample Code

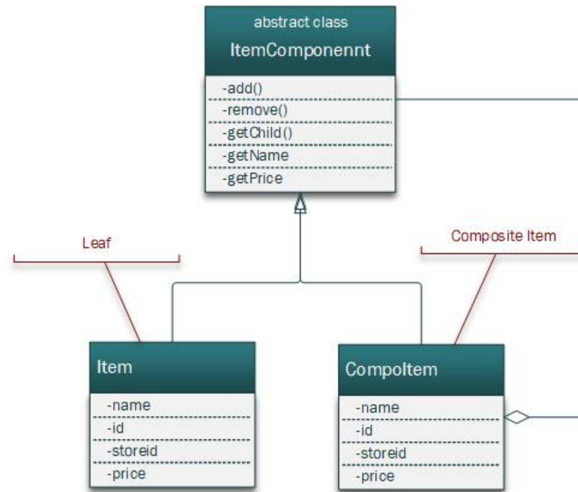

Figure 6: Composite Design Pattern Class Diagram

4.5 Composite Design Pattern

The Composite design Pattern is used to represent the item (composite item), the item ,and **CompoItem** (concrete classes), both inherits from the interface **ItemComponent**, the item is composed into tree structure in which each node could be of type **(item)** leaf, or composite of type **CompoItem**. the figure below shows the Class diagram.

- **ItemComponent** represents the Composite item abstract calss.
- **CompoItem** a concrete Item of type composite, it inherits the **ItemComponent** and override it's methods.
- **(item)** represents the concrete Leaf item in the tree structure (Tree Stub), it has no children, and can not add any other items of any type.

```
1  public double getPrice() {
2      double total = 0;
3      Iterator iterator = itemComponents.iterator();
4      while (iterator.hasNext()) {
5          ItemComponent itmComponent
6                  = (ItemComponent) iterator.next();
7          total += itmComponent.getPrice();
8      }
9      return total;
10 }
```

Listing 6: Composite Design Pattern Sample Code