Applied Design Pattern - Ecommerce Design

Nitisha Bharathi (16PT25)

November 2020

1 Problem Statement

To design an e-commerce appplication using GOF design patterns and domain specific patterns.

2 Design Patterns Used

List of patterns used in this e-commerce application design has been listed below

Pattern Name	Usage
Singleton	User's browsing history list. The user can see what are the products they have searched and viewed earlier
Prototype	When a user asks for information for a certain product the application could gather that information and keep them in a cache, when an item is requested, it is retrieved from cache and cloned.
Decorator	Certain products like gift items can be customized. Here the product can be customized by adding personalized names or photos.
Observer	When the item is not in stock observer pattern can be useful to notify user about the availability of the item. Second use is notification to the user in case of shipping the item. The item undergoes various stages like shipped delivered such phases can be notified to user through Observer Pattern.
Strategy	Various payment methods are implemented using Strategy Pattern.
Catalog	The Pattern classifies and describes a variety of products so that customers can find easily what they want.
Invoice	The pattern describes events such as the creation and validation of an invoice, followed by the payment process.

Table 1: List of design patterns used

3 UML Class Diagram

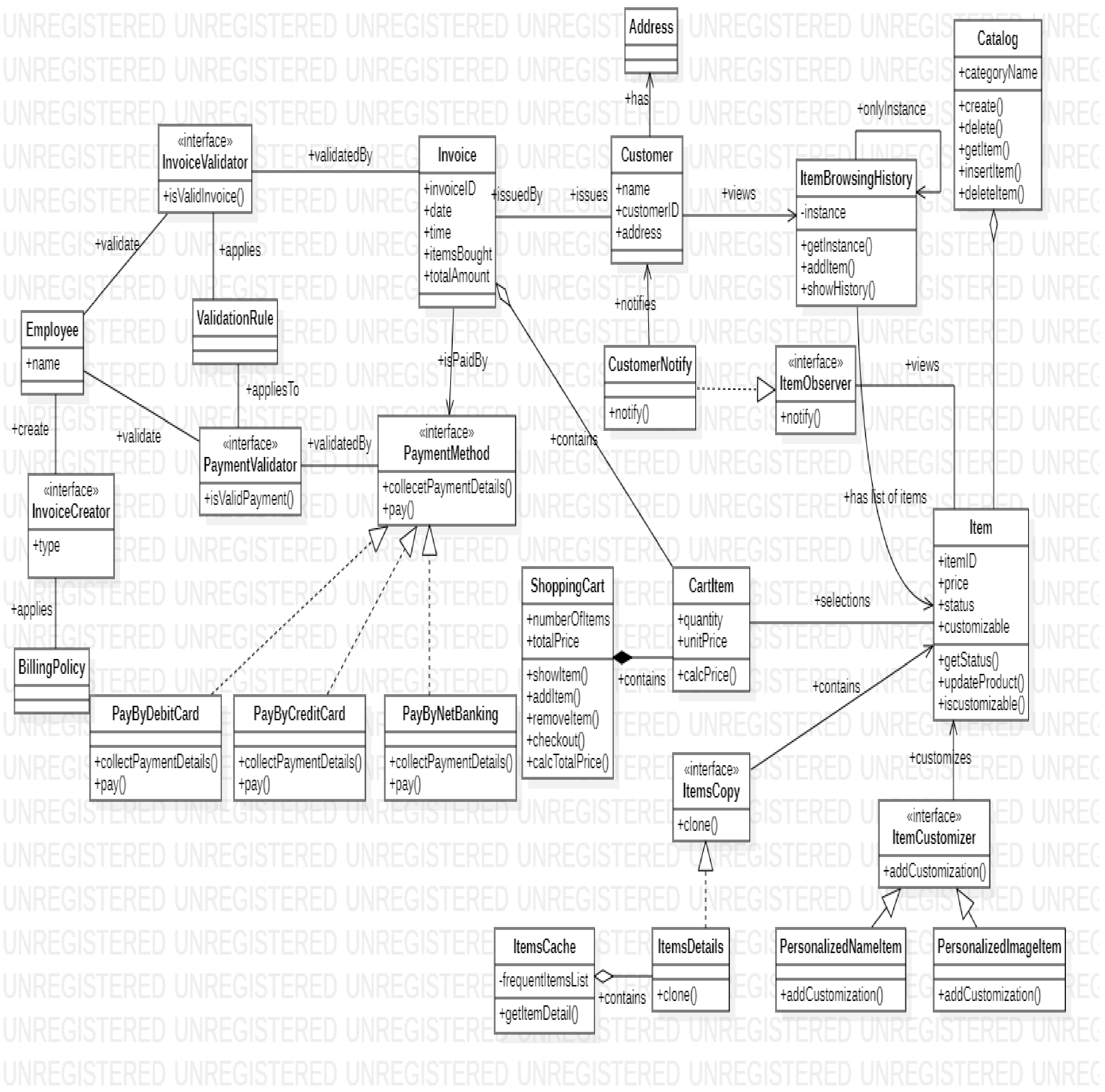

Figure 1: UML Class Diagram for E-Commerce Application[1]

[1]The Star UML software used for this UML diagram creation is unlicensed hence, unregistered watermark is present

4 Implementation

1. **Singleton Pattern**

```java

public class ItemBrowsingHistory {
  private static ItemBrowsingHistory onlyInstance;
  private Item item;

  private ItemBrowsingHistory() {

  }

  public static synchronized ItemBrowsingHistory getInstance() {
    if(onlyInstance == null)
    {
      onlyInstance = new ItemBrowsingHistory();

    }
    return onlyInstance;
  }


  public void addItem(Item item)
  {

  }

  public Item showHistory()
  {
    return item;
  }
}
```

2. **Decorator Pattern**

```java

abstract class ItemsCustomizer{
  Item item;
  public abstract void addCustomization();
}
```

```java

public class NameCustomizer extends ItemsCustomizer {
  Item item;
  public NameCustomizer(Item i)
  {
    this.item = i;
  }
  public void addCustomization() {
    if(item.isCustomizable())
    {
      // logic to add personalized name
    }

  }
}
```

```java

public class ImageCustomizer extends ItemsCustomizer{
  Item item;
  public ImageCustomizer(Item i)
  {
    this.item = i;
  }
  public void addCustomization() {
    if(item.isCustomizable())
    {
      // logic to add personalized image
    }

  }
}
```

3. Strategy Pattern

```java

public interface PaymentMethod {
  void pay(float paymentAmount);
    void collectPaymentDetails();
}
```

```java

public class PaybyCreditCard implements PaymentMethod {
  public void pay(float paymentAmount)
  {
    //logic to carry on the transaction
  }

  public void collectPaymentDetails()
  {

  }
}
```

```java

public class PayByDebitCard implements PaymentMethod{
  public void pay(float paymentAmount)
  {
    //logic to carry on the transaction
  }

  public void collectPaymentDetails()
  {

  }
}
```

```java

public class PayByNetBanking implements PaymentMethod {
  public void pay(float paymentAmount)
  {
    //logic to carry on the transaction
  }

  public void collectPaymentDetails()
  {

  }
}
```