Assignment – 4 – Group 9

Find two applications of behavioral patterns to your system and implement in java. Create a class diagram and sequence diagram for it.

Solution:

1. **Memento – applying pattern on pizza's Cart**

   Java code:

   Cart class// Orginator class

```
public class Cart
{
  private long id;
  private String status;
  private int orderValue;

  public Cart(long id, String status) {
    super();
    this.id = id;
    this.status = status;
  }

  //Setters and getters

  public CartMemento createMemento()
  {
    CartMemento m = new CartMemento(id, status, orderValue);
    return m;
  }

  public void setStatus(String status) {
        this.status = status;
}

  public void setOrderValue(int orderValue) {
        this.orderValue = orderValue;
}
```

```
  public void restore(CartMemento m) {
    this.id = m.getId();
    this.status = m.getStatus();
    this.orderValue = m.getOrderValue();
  }

  @Override
  public String toString() {
    return "Cart [id=" + id + ", status=" + status + ", order Value="+
orderValue+"]";
  }

}
```

CartMemento Class // Memento

```
public class CartMemento {
  private final long id;
  private final String status;
  private final int orderValue;

  public CartMemento(long id, String status, int orderValue) {
    super();
    this.id = id;
    this.status = status;
    this.orderValue = orderValue;
  }

  public long getId() {
    return id;
  }

  public String getStatus() {
    return status;
  }

  public int getOrderValue() {
    return orderValue;
  }
```

```
}
```

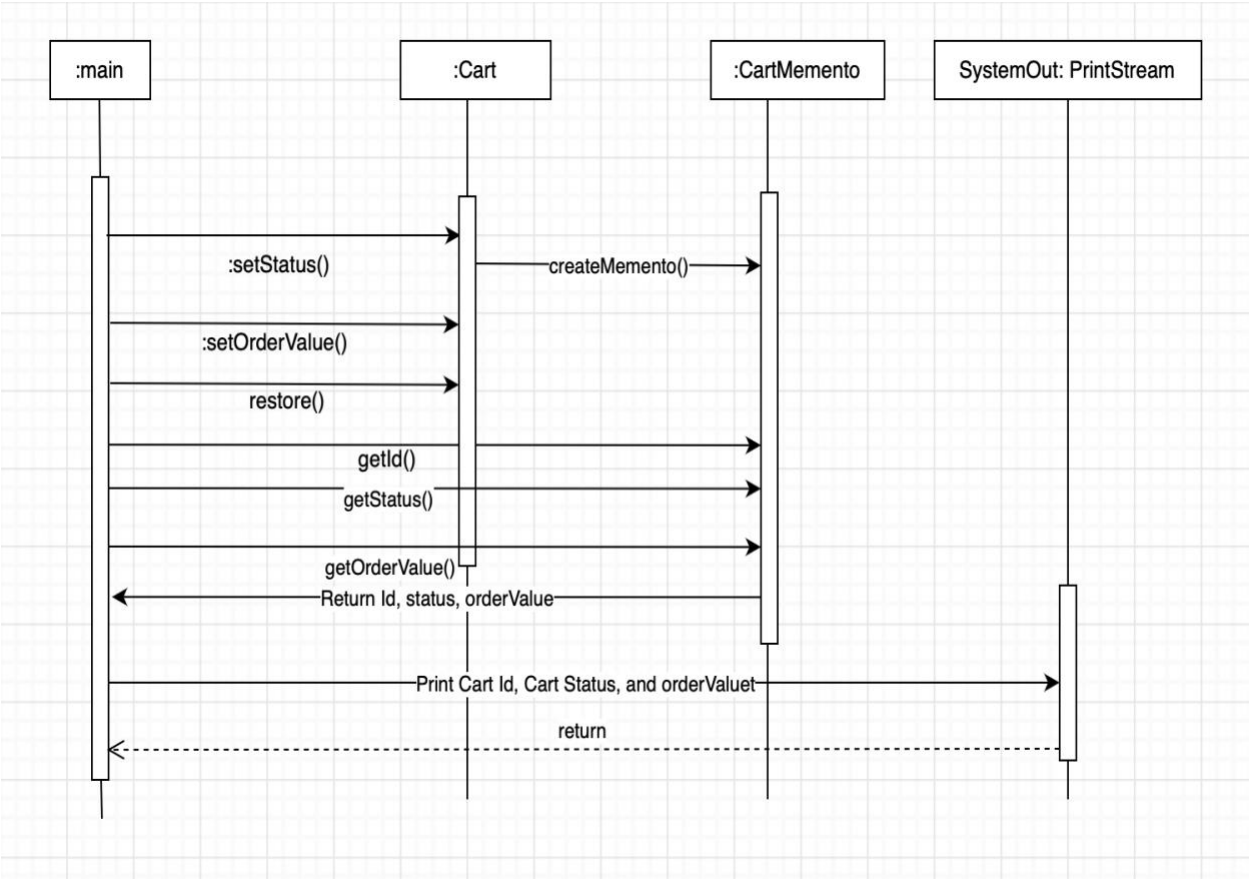
Cart class // Caretaker class

```
public class CartMain {
  public static void main(String[] args)
  {
    Cart cart1 = new Cart(1, "Empty");
    cart1.setOrderValue(0);
    System.out.println(cart1);

    CartMemento memento = cart1.createMemento(); //created immutable
memento

    cart1.setStatus("Order is present");
    cart1.setOrderValue(1);    //changed content
    System.out.println(cart1);

    cart1.restore(memento);   //UNDO operation on cart
    System.out.println(cart1);  //original content
  }
}
```

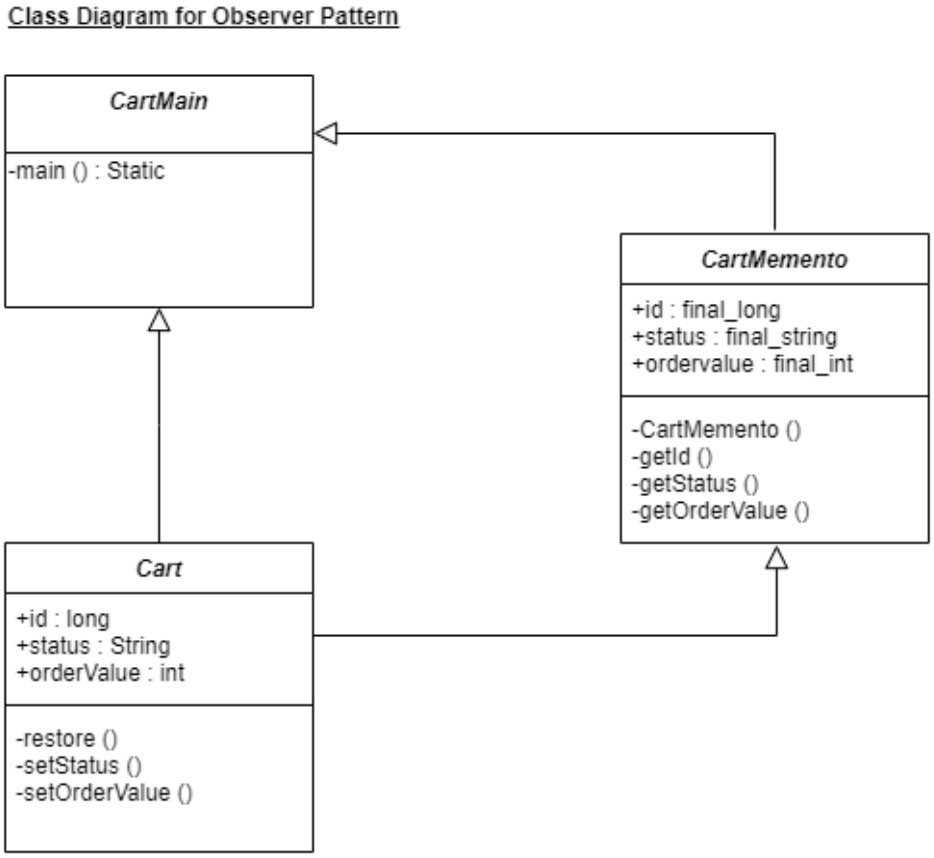
Output:

```
[Running] cd "/Users/juythikkalathiya/Desktop/Coding/" && javac CartMain.java && java CartMain
Cart [id=1, status=Empty, order Value=0]
Cart [id=1, status=Order is present, order Value=1]
Cart [id=1, status=Empty, order Value=0]
```

Sequence Diagram for Memento Pattern –



Class Diagram for Memento Pattern :



2. **Observer Pattern –**

Observer Pattern is used when you have a requirement where multiple entities are interested in any possible update.
In observer pattern the application creates a subject object, then all the observers will register themselves to be notified for any further update in the state of the object. As the state notifies all registered observers, the observers can access the states and act accordingly.
Here, in our pizza delivery system, the observers can be chefs, customers, delivery person, who will see the states of the order.
So the subject class will notify the observers about the states of the current order.

The observer pattern will have below participants –

1. Subject : here, interface or abstract class will be defined for operations for attaching and deattaching the observers.
2. Concrete Subject: It will maintain the state of the object and when a change will occur in the state, it will notify all observers.
3. Observer: it will be an interface or abstract class to define the operations to be used to notify this object.
4. Concrete Observer: these will be the observers who all needs to be notified about the state of the pizza order.

Java Source Code :

//Concrete Observer Classes

```
public class DeliveryPerson implements Observer
{
   @Override
   public void update(PizzaStatus status) {
      System.out.println("Delivery Person : Pizza Status :: " +
status.getOrderStatus());
   }
}

public class CustomerClass implements Observer
{
   @Override
```

```
    public void update(PizzaStatus status) {
        System.out.println("Customer : Pizza Status :: " +
status.getOrderStatus());
    }
}

public class ChefClass implements Observer
{
    @Override
    public void update(PizzaStatus status) {
        System.out.println("Chef : Pizza Status :: " + status.getOrderStatus());
    }
}
```

//State object class – so no class can change the status

```
public class PizzaStatus
{
    final String orderStatus;

    public PizzaStatus (String orderStatus) {
        this.orderStatus = orderStatus;
    }

    public String getOrderStatus() {
        return orderStatus;
    }
}
```

//Observer Class

```
public interface Observer
{
    public void update(PizzaStatus status);
}


public interface Observable
{
    public void attach(Observer o);
    public void detach(Observer o);
    public void notifyUpdate(PizzaStatus status);
```

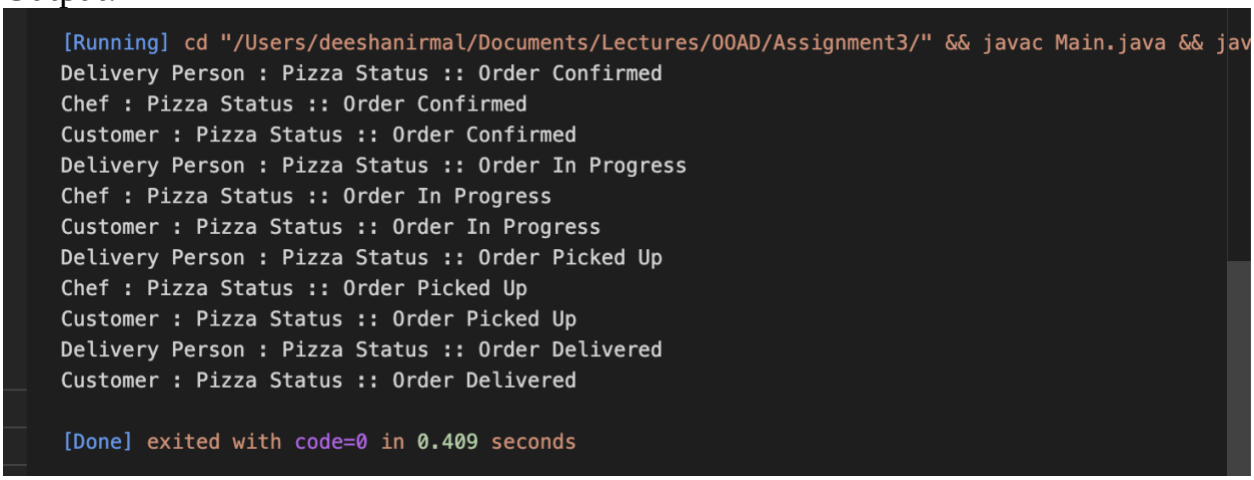
```
}
```

//Concrete Subject Class

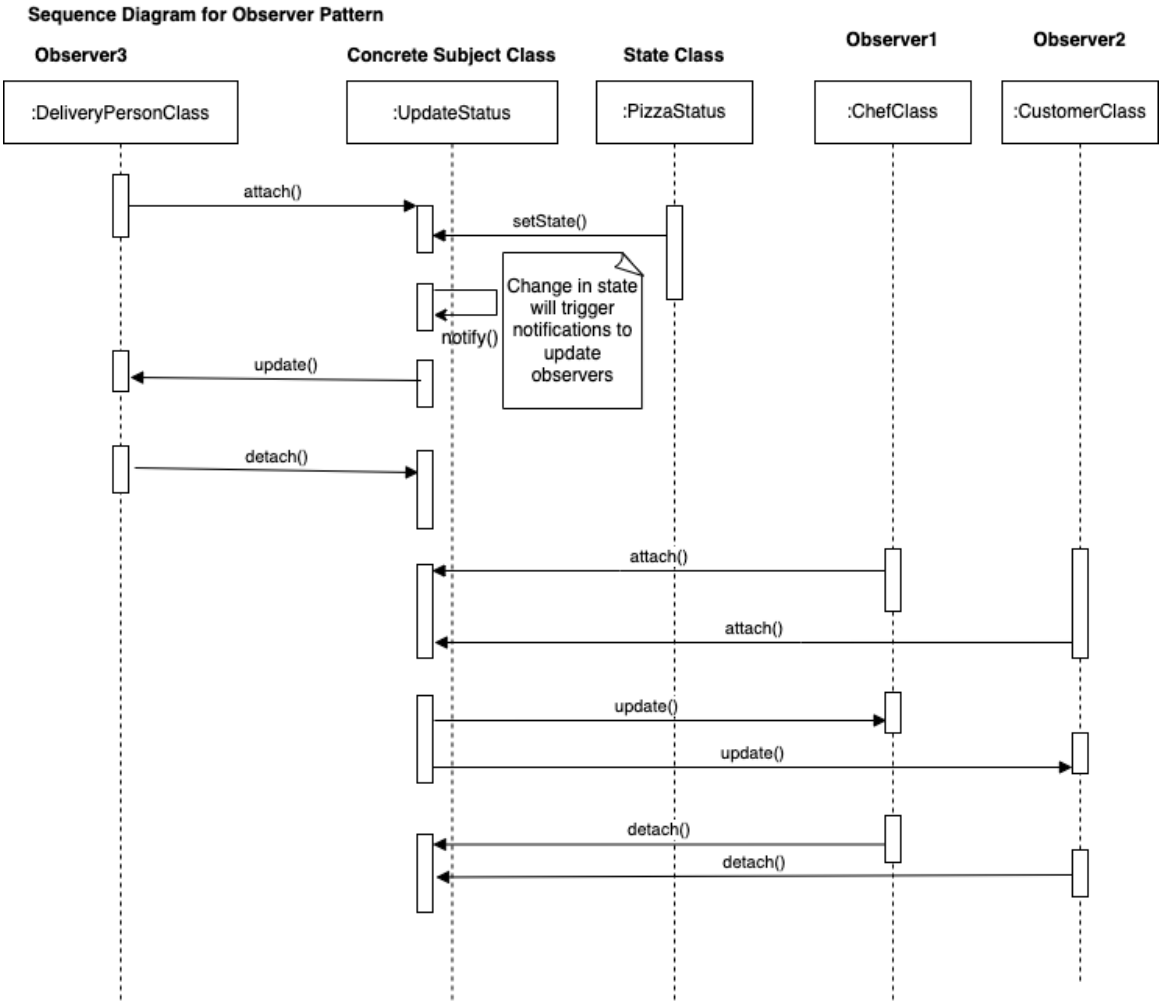
```
import java.util.ArrayList;
import java.util.List;

public class UpdateStatus implements Observable {

   private List<Observer> observers = new ArrayList<>();

   @Override
   public void attach(Observer obj) {
      observers.add(obj);
   }

   @Override
   public void detach(Observer obj) {
      observers.remove(obj);
   }

   @Override
   public void notifyUpdate(PizzaStatus status) {
      for(Observer obj: observers) {
         obj.update(status);
      }
   }
}


public class Main
{
   public static void main(String[] args)
   {
      DeliveryPerson dPerson = new DeliveryPerson();
      ChefClass chef = new ChefClass();
      CustomerClass customer = new CustomerClass();

      UpdateStatus o = new UpdateStatus();
```

```
        o.attach(dPerson);
        o.attach(chef);
        o.attach(customer);

        o.notifyUpdate(new PizzaStatus("Order Confirmed"));  //the chef,
customer and delivery people will recieve the status update
        o.notifyUpdate(new PizzaStatus("Order In Progress"));
        o.notifyUpdate(new PizzaStatus("Order Picked Up"));

        o.detach(chef);
        o.notifyUpdate(new PizzaStatus("Order Delivered"));

    }
}
```

Output:



Sequence Diagram –



Class Diagram :