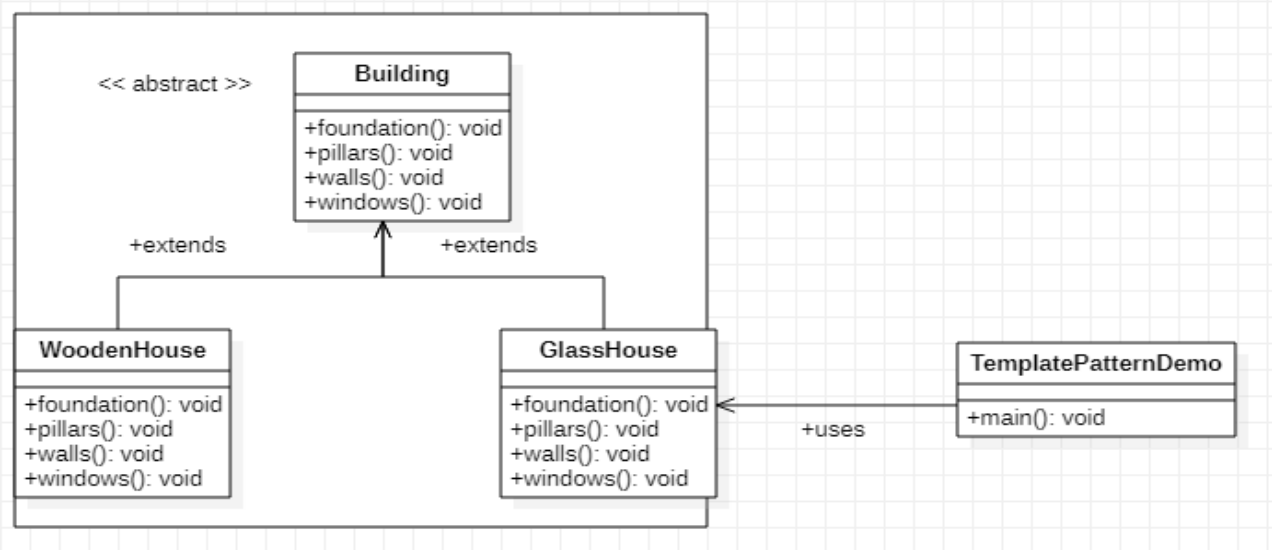
Lab Session 05

Practice different types of Behavioral Design Patterns

Exercises

1. Suppose we want to provide an algorithm to build a house. The steps needed to be performed to build a house are – building foundation, building pillars, building walls and windows. The important point is that we can't change the order of execution because we can't build windows before building the foundation. So in this case we can create a template method that will use different methods to build the house. Now building the foundation for a house is the same for all types of houses, whether it is a wooden house or a glass house. So we can provide base implementation for this, if subclasses want to override this method, they can but mostly it's common for all the types of houses. To make sure that subclasses don't override the template method, we should make it final. Draw class diagram and implement them using template design pattern. Attach printouts.



Step1: Create an abstract class with a template method being final.

*Building.java

```
1  package lab5_q1;
2
3  public abstract class Building {
4      abstract void foundation();
5      abstract void pillars();
6      abstract void walls();
7      abstract void windows();
8
9      //template method
10     public final void build(){
11         foundation();
12         pillars();
13         walls();
14         windows();
15     }
16 }
```

Step 2: Create concrete classes extending the above class.

*WoodenHouse.java

```java
package lab5_q1;

public class WoodenHouse extends Building {
    void walls() {
        System.out.println("Wooden house walls build");
    }
    void foundation() {
        System.out.println("Wooden house foundation layed! Start building.");
    }
    void pillars() {
        System.out.println("Wooden house pillars made");
    }
    void windows() {
        System.out.println("Wooden house windows installed");
    }
}
```

*GlassHouse.java

```java
package lab5_q1;

public class GlassHouse extends Building {
    void walls() {
        System.out.println("Glass walls installed!");
        }
    void foundation() {
        System.out.println("Glass house foundation layed! Start building.");
        }
    void pillars() {
        System.out.println("Glass house Pillar made");
        }
    void windows() {
        System.out.println("Glass house windows installed");
        }
}
```

Step 3: Use the Building's template method build() to demonstrate a defined way to build a building.

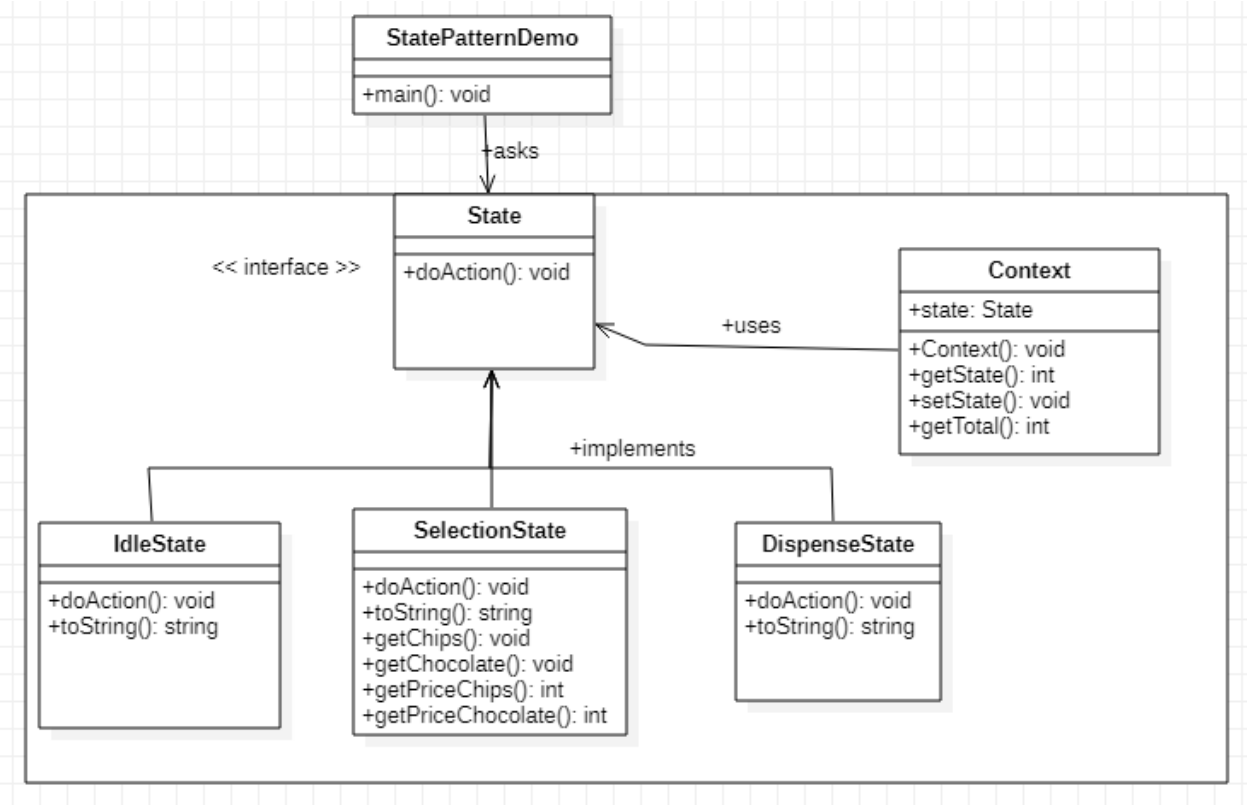
*TemplatePatternDemo.java

```java
package lab5_q1;

public class TemplatePatternDemo {
    public static void main(String[] args) {
        Building house = new WoodenHouse();
        house.build();
        System.out.println();
        house = new GlassHouse();
        house.build();
    }
}
```

Output:

```
<terminated> TemplatePatternDemo [Java Application] C:\Us
Wooden house foundation layed! Start building.
Wooden house pillars made
Wooden house walls build
Wooden house windows installed

Glass house foundation layed! Start building.
Glass house Pillar made
Glass walls installed!
Glass house windows installed
```

2. Implement the operation of a Vending Machine using an appropriate design pattern in Java. Also draw the class diagram and attach printouts. Initially the machine is idle. Once the customer arrives he can purchase either a chocolate or a packet of chips or both. Chocolate worth 10 cents and a packet of chips worth 20 cents. A customer can enter one coin at a time. Finally the requested product is dispensed and the total cost appears at the display panel. Machine again goes to the idle state.



Step 1: Create an interface.

State.java

```
1  package lab5_q2;
2
3  public interface State {
4      public void doAction(Context context);
5  }
```

Step 2: Create concrete classes implementing the same interface.

*IdleState.java

```
1  package lab5_q2;
2
3  public class IdleState implements State {
4      public void doAction(Context context){
5          System.out.println("Machine is in idle state");
6          context.setState(this);
7
8      }
9      public String toString() {
10         return "Idle State";
11     }
12 }
```

*SelectionState.java

```java
package lab5_q2;

public class SelectionState implements State {
    public void doAction(Context context){
        System.out.println("Machine is in selection state");
        context.setState(this);
    }
    public void getChips(Context context){
        System.out.println("you have selected chips");
        context.setState(this);
    }
    public void getChocolate(Context context){
        System.out.println("you have selected chocolate");
        context.setState(this);
    }
    public String toString() {
        return "Selection State";
    }
    public int getPriceChocolate() {
        return 10;
    }
    public int getPriceChips() {
        return 20;
    }
}
```

*DispensedState.java

```java
package lab5_q2;

public class DispensedState implements State  {
    public void doAction(Context context){
        System.out.println("Machine is in dispensed state");
        context.setState(this);
    }
    public String toString() {
        return "Dispensed State";
    }
}
```

Step 3: Create Context Class.

Context.java

```java
package lab5_q2;

public class Context {
    private State state;
    public Context() {
        state=null;
    }
    public void setState(State state) {
        this.state=state;
    }
    public State getState() {
        return state;
    }
    public State getTotal() {
        return state;
    }
}
```

Step 4: Use the Context to see change in behaviour when State changes.

*StatePatternDemo.java

```
1  package lab5_q2;
2
3  public class StatePatternDemo {
4      public static void main(String[] args) {
5          Context context=new Context();
6
7          IdleState idleState=new IdleState();
8          idleState.doAction(context);
9          System.out.println(context.getState().toString());
10         System.out.println("");
11         SelectionState selectionState=new SelectionState();
12         selectionState.doAction(context);
13         System.out.println(context.getState().toString());
14         selectionState.getChips(context);
15         System.out.println("Chips price:"+((SelectionState) context.getTotal()).getPriceChips()+"cents");
16         System.out.println("");
17         selectionState.getChocolate(context);
18         System.out.println("Chocolate price:"+((SelectionState) context.getTotal()).getPriceChocolate()+"cents");
19         System.out.println("");
20         System.out.println("Both Chips & Chocolate price:"+
21         (((SelectionState) context.getTotal()).getPriceChocolate()+((SelectionState) context.getTotal()).getPriceChips())+
22         "cents");
23         System.out.println("");
24         DispensedState dispensedState=new DispensedState();
25         dispensedState.doAction(context);
26         System.out.println(context.getState().toString());
27     }
28 }
```

Output:

Console

<terminated> StatePatternDemo [Java Application] C:

```
Machine is in idle state
Idle State

Machine is in selection state
Selection State
you have selected chips
Chips price:20cents

you have selected chocolate
Chocolate price:10cents

Both Chips & Chocolate price:30cents

Machine is in dispensed state
Dispensed State
```

3. Create an interface Order which is acting as a command. Also create a Stock class which acts as a request. Then create concrete command classes BuyStock and SellStock implementing Order interface which will do actual command processing. A class Broker must be created which acts as an invoker object. It can take and place orders. Broker object uses command pattern to identify which object will execute which command based on the type of command. CommandPatternDemo, a demo class, will use Broker class to demonstrate command pattern. Attach printouts.

Step 1: Create a command interface.

*Order.java

```
1  package lab5_q3;
2
3  public interface Order {
4      void execute();
5  }
```

Step 2: Create a request class.

Stock.java

```
1  package lab5_q3;
2
3  public class Stock {
4      private String name = "Atlas Honda Ltd";
5      private int quantity = 5;
6
7      public void buy(){
8          System.out.println("Stock Name: "+ name +", Quantity bought: " + quantity);
9          }
10     public void sell(){
11         System.out.println("Stock Name: "+ name +", Quantity sold: " + quantity);
12         }
13 }
```

Step 3: Create concrete classes implementing the *Order* interface.

*BuyStock.java

```
1  package lab5_q3;
2
3  public class BuyStock implements Order {
4      private Stock abcStock;
5
6      public BuyStock(Stock abcStock){
7          this.abcStock = abcStock;
8          }
9
10     public void execute() {
11         abcStock.buy();
12         }
13 }
```

*SellStock.java

```
1  package lab5_q3;
2
3  public class SellStock implements Order {
4      private Stock abcStock;
5
6      public SellStock(Stock abcStock){
7          this.abcStock = abcStock;
8          }
9
10     public void execute() {
11         abcStock.sell();
12         }
13     }
```

Step 4: Create command invoker class.

*Broker.java

```
1  package lab5_q3;
2
3  import java.util.ArrayList;
5
6  public class Broker {
7      private List<Order> orderList = new ArrayList<Order>();
8
9      public void takeOrder(Order order){
10           orderList.add(order);
11       }
12
13     public void placeOrders(){
14         for (Order order : orderList) {
15             order.execute();
16             }
17         orderList.clear();
18         }
19 }
```

Step 5: Use the Broker class to take and execute commands.

*CommandPatternDemo.java

```
1  package lab5_q3;
2
3  public class CommandPatternDemo {
4      public static void main(String[] args) {
5          Stock abcStock = new Stock();
6
7          BuyStock buyStockOrder = new BuyStock(abcStock);
8          SellStock sellStockOrder = new SellStock(abcStock);
9
10         Broker broker = new Broker();
11         broker.takeOrder(buyStockOrder);
12         broker.takeOrder(sellStockOrder);
13
14         broker.placeOrders();
15        }
16 }
```

Output: