DESIGN PATTERNS HW1

Betülnaz Hayran-28354853660 Github:Naz-bh

Batuhan Kesikbaş-40573251614 Github:BatuhanKesikbas

1-Statement of Work

We are calculating monthly interest on different types of bank accounts. We are only dealing with two account types. Interest will not be applicable to any other types of accounts. Current accounts paying 2% interest per annum, and savings accounts paying 4% per annum. Interest will not be applicable to any other types of accounts. Our account types are defined by an enum.

We need to add support for two different money market accounts. A standard money market account paying 6% per annum, and a special "high-roller" money market account that pays 7.5%, but only if the customer maintains a minimum balance of $100.000,00.

Our code gets messier with every set of new requirements that we implement. If we keep on adding more and more business rules into our calculator, we are going to end up with something that could become very difficult to maintain.

The problem that we have is this:

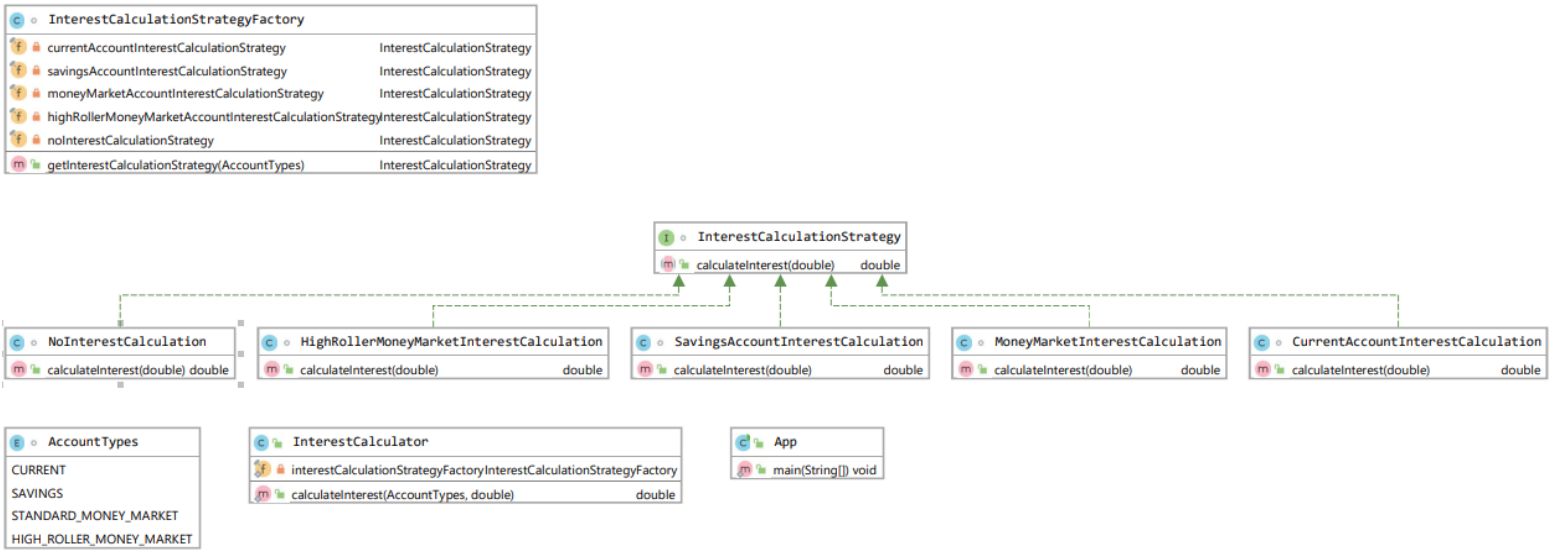
- We have a single, convoluted, hard-to-maintain method that is trying to deal with a number of different scenarios.

2- Explanation on Utilized Design Patterns

The Strategy pattern allows us to dynamically swop out algorithms (i.e. application logic) at runtime. In our scenario, we want to change the logic used to calculate interest, based on the type of account that we are working with.

The Factory pattern allows us to create objects without necessarily knowing or caring about the type of objects that we are creating. This is exactly what our calculator needs.

3- UML Class Diagram



4- Research

4.1-The Strategy Pattern

```
public interface Strategy {
    int execute(int a,int b);
}


public class Context {
    private Strategy strategy;
    public Context(Strategy strategy){
        this.strategy = strategy;
    }
    public int executeStrategy(int a, int b) {
        return strategy.execute(a, b);
    }
}


public class ConcreteStrategyAdd implements Strategy{
```

```
    @Override
    public int execute(int a, int b) {
        return a+b;
    }
}

public class ConcreteStrategySubtract implements Strategy{
    @Override
    public int execute(int a, int b) {
        return a-b;
    }
}
public class ConcreteStrategyMultiply implements Strategy{
    @Override
    public int execute(int a, int b) {
        return a*b;
    }
}

public class ExampleApplication {
        public static void main(String[] args) {
            String action="subtract";
            int firstNumber=15;
            int secondNumber=10;

            Context context;
            if (action=="add") {
                context = new Context(new ConcreteStrategyAdd());
                System.out.println(firstNumber+"+"+secondNumber +"="+ context.executeStrategy(firstNumber,
secondNumber));
```

```
        }

        if (action=="subtract") {

            context = new Context(new ConcreteStrategySubtract());

            System.out.println(firstNumber+"-"+secondNumber +"="+ context.executeStrategy(firstNumber,
secondNumber));

        }

        if (action=="multiply") {

            context = new Context(new ConcreteStrategyMultiply());

            System.out.println(firstNumber+"*"+secondNumber +"="+context.executeStrategy(firstNumber,
secondNumber));

        }

    }

  }
```

How to Implement

We create a Strategy interface defining an action and concrete strategy classes implementing the Strategy interface. Context is a class which uses a Strategy.

ExampleApplication, will use Context and strategy objects to demonstrate change in Context behavior based on strategy it deploys or uses.

1. In the context class, identify an algorithm that's prone to frequent changes. It may also be a massive conditional that selects and executes a variant of the same algorithm at runtime.
2. Declare the strategy interface common to all variants of the algorithm.
3. One by one, extract all algorithms into their own classes. They should all implement the strategy interface.
4. In the context class, add a field for storing a reference to a strategy object. Provide a setter for replacing values of that field. The context should work with the strategy object only via the strategy interface. The context may define an interface which lets the strategy access its data.
5. Clients of the context must associate it with a suitable strategy that matches the way they expect the context to perform its primary job.

Source: https://refactoring.guru/design-patterns/strategy

https://www.tutorialspoint.com/design_pattern/strategy_pattern.htm

4.2-The Factory Pattern

```
abstract class Plan{
   protected double rate;
   abstract void getRate();

   public void calculateBill(int units){
      System.out.println(units*rate);
   }
}

class  DomesticPlan extends Plan{
   @Override
   void getRate() {
      rate=3.50;
   }
}

class  CommercialPlan extends Plan {
   @Override
   void getRate() {
      rate = 7.50;
   }
}

class  InstitutionalPlan extends Plan {
   @Override
   void getRate() {
      rate = 5.50;
```

```
    }
}

class GetPlanFactory{
    public Plan getPlan(String planType){
        if(planType == null){
            return null;
        }
        if(planType.equalsIgnoreCase("DOMESTICPLAN")) {
            return new DomesticPlan();
        }
        else if(planType.equalsIgnoreCase("COMMERCIALPLAN")){
            return new CommercialPlan();
        }
        else if(planType.equalsIgnoreCase("INSTITUTIONALPLAN")) {
            return new InstitutionalPlan();
        }
        return null;
    }
}

import java.io.*;
class GenerateBill{
    public static void main(String args[])throws IOException {
        GetPlanFactory planFactory = new GetPlanFactory();

        System.out.print("Enter the name of plan for which the bill will be generated: ");
        BufferedReader br=new BufferedReader(new InputStreamReader(System.in));

        String planName=br.readLine();
```

```
      System.out.print("Enter the number of units for bill will be calculated: ");
      int units=Integer.parseInt(br.readLine());

      Plan p = planFactory.getPlan(planName);

      System.out.print("Bill amount for "+planName+" of  "+units+" units is: ");
      p.getRate();
      p.calculateBill(units);
   }
}
```

How to Implement

1. Create a Plan abstract class.
2. Create the concrete classes that extends Plan abstract class.
3. Create a GetPlanFactory to generate object of concrete classes based on given information..
4. Generate Bill by using the GetPlanFactory to get the object of concrete classes by passing an information such as type of plan DOMESTICPLAN or COMMERCIALPLAN or INSTITUTIONALPLAN.

Example output:

Enter the name of plan for which the bill will be generated: DomesticPlan

Enter the number of units for bill will be calculated: 10

Bill amount for DomesticPlan of 10 units is: 35.0

Source: https://www.javatpoint.com/factory-method-design-pattern

https://www.tutorialspoint.com/design_pattern/factory_pattern.htm

5-Implementation codes