**Software Development Tools laboratory work 9**

**EXERCISE 01 - CHOOSE THE BEST PATTERN**

**IMPORTANT: PERFORM ALL TASKS ACCORDING TO YOUR OWN TOPIC.**

PURPOSE

In this exercise, you will experience using the pattern repository structure and pattern descriptions to identify a suitable pattern for a programming task.

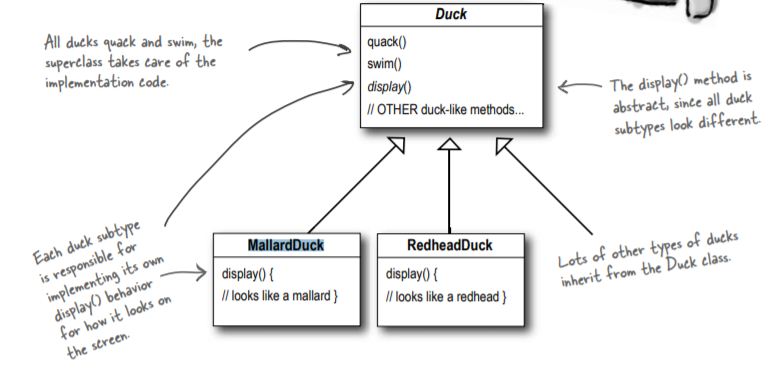
ACTIVITIES

To successfully complete this exercise, you must:

1. Individually read through the following coding problem.

“You need to create an application to sort lists of names that vary greatly in number of names. You know that sort algorithms are more or less efficient depending on the number of elements to be sorted so you want to supply a variety of sort algorithms and have the choice of algorithm employed to be made at runtime.”

1. Discuss the problem in your group and identify the key issues involved with solving the problem. Write down the category most likely to contain the needed design pattern.
2. On the Internet find web sites that provide pattern repositories.
3. Search the repository for a pattern that would be useful in solving the coding problem.
4. Write down the name of the pattern and defend your choice of patterns.
5. **After that, define a problem in your project and try to choose suitable pattern to solve this problem.**
6. Discuss the problem in your group and identify the key issues involved with solving the problem. Write down the category most likely to contain the needed design pattern.
7. Implement chosen pattern using any programming language you know.
8. Upload your work to the DL.
9. Follow an example below.



1. There is a superclass Duck that represents generalized implementation code, contains methods quack() and swim() and abstract class display(), because child classes are displayed differently.

2. There happens a task, that requires to create a method fly(), which is added in superclass Duck, and all subclasses inherit a method fly(). However, here is a confusion, because not all ducks can fly (for example, rubber ducks).

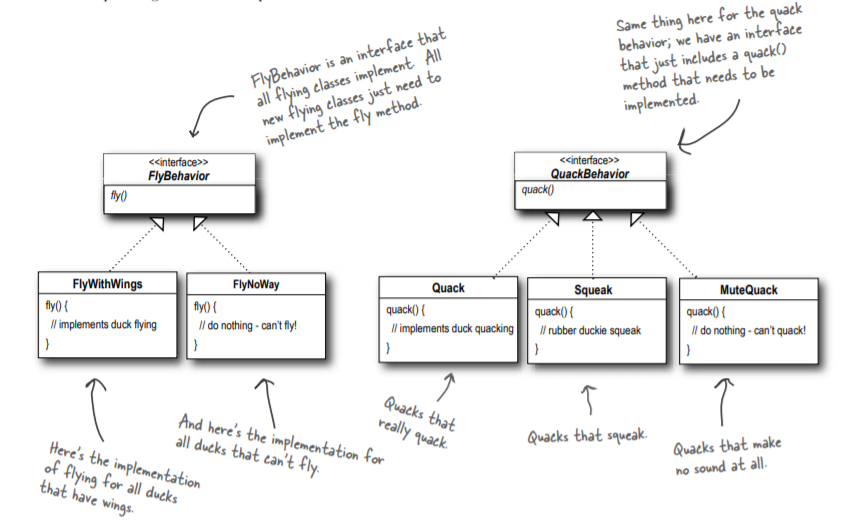
3. Obviously, this method can be overridden, but also we have another several subclasses where there is no need of a method fly(). There are also classes where there is no need of a method quack(), by the way. In general, inheritance can’t solve this task.

4. We need a way to make fly and quack only certain ducks. May we try interfaces? You can create interfaces Flyable() with a method fly() and Quackable() with a method quack(). As same as overriding it takes time to make change in behavior of a method fly() in all "flyable" classes.

5. As a result, we need to use Strategy pattern, which says “highlight what changes and encapsulate these aspects so that they do not affect the rest of the code”. So it requires to separate variable from constant.

6. Class Duck is almost constant, we know that fly() and quack() are parts of a class Duck that vary by subclasses. To separate these aspects of behavior from the class Duck, we take both methods outside the class Duck and create a new set of classes to represent each aspect.

7. Then it is desirable to be able to create a new instance of MallardDuck (for example) and initialize it with a specific type of behavior of the method fly(). Now, behavioral aspects will be in separate classes that implement the interface of a particular aspect.



8. Testing Duck code

* Type and compile the Duck class and the MallardDuck class.

public abstract class Duck{

FlyBehavior flyBehavior;

QuackBehavior quackBehavior;

public Duck(){}

public abstract void display();

public void performFly(){

flyBehavior.fly(); }

public void performQuack(){

quackBehavior.quack(); }

public void swim(){

System.out.println("All ducks float, even decoys!"); }

}

public class MallardDuck extends Duck{

public MallardDuck(){

quackBehavior = new Quack();

flyBehavior = new FlyWithWings();

}

public void display(){

System.out.println("I am a real Mallard duck!");

}

}

* Type and compile the FlyBehavior interface (FlyBehavior.java) and the two behavior implementation classes (FlyWithWings.java and FlyNoWay.java).

public interface FlyBehavior{

public void fly();

}

public class FlyWithWings implements FlyBehavior{

public void fly(){

System.out.println("I am flying!");

}

}

public class FlyNoWay implements FlyBehavior{

public void fly(){

System.out.println("I cannot fly!");

}}

* Type and compile the QuackBehavior interface (QuackBehavior.java) and the three behavior implementation classes (Quack.java, MuteQuack.java, and Sqeak.java).

public interface QuackBehavior{

public void quack();

}

public class Quack implements QuackBehavior{

public void quack(){

System.out.println("Quack!");

}

}

public class MuteQuack implements QuackBehavior{

public void quack(){

System.out.println("<<Silence>>");

}

}

public class Squeak implements QuackBehavior{

public void quack(){

System.out.println("Squeak!");

}}

* Type and compile the test class (MiniDuckSimulator.java).

public class MiniDuckSimulator{

public static void main(String[] args) {

Duck mallard = new MallardDuck();

mallard.performQuack();

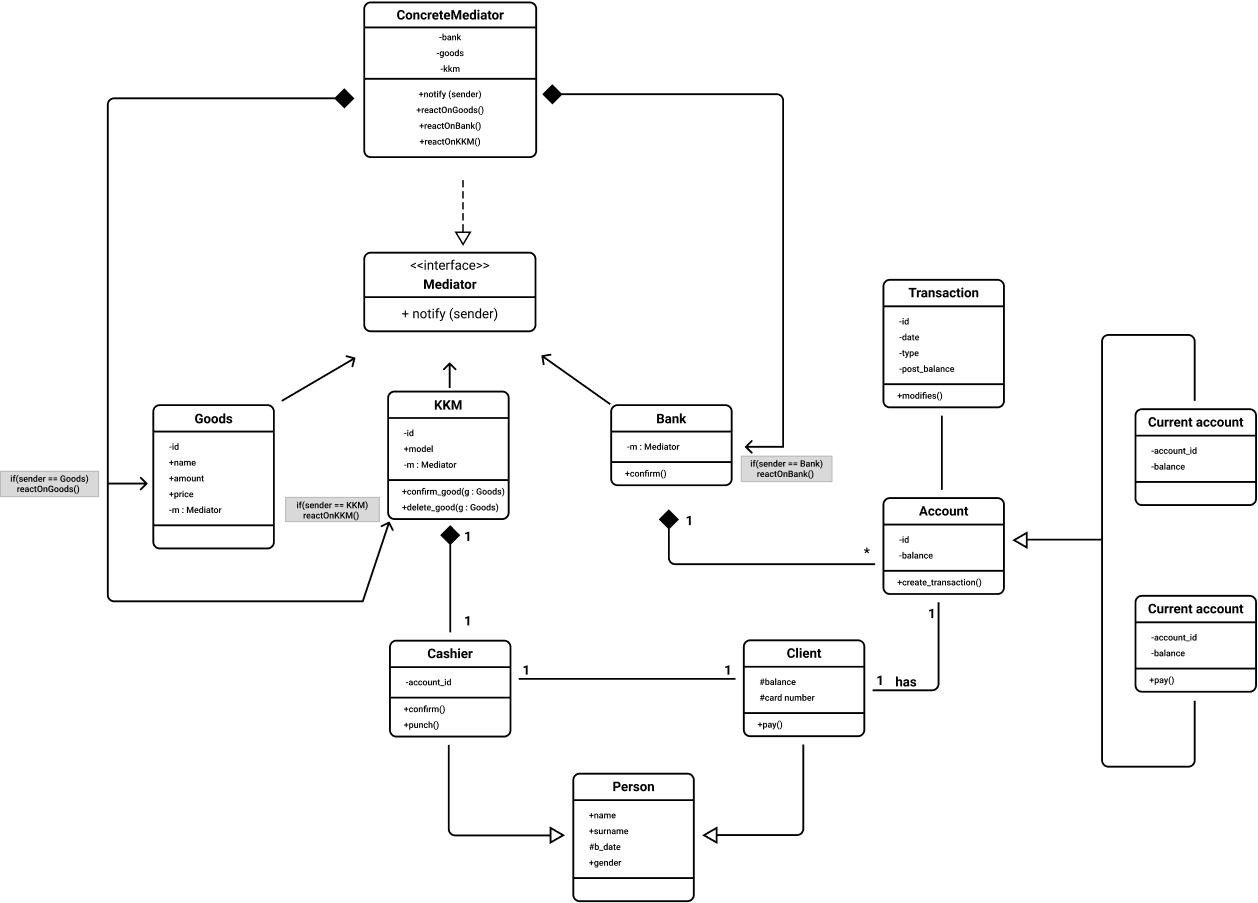
mallard.performFly();

}

}

* Run the code.

Screenchot:



Code:

*package* com.company;  
  
*import* java.util.Scanner;  
  
*public class* Main {  
  
 *public static void* main(String[] args) {  
  
 System.*out*.println("Hello, choose method:");  
 System.*out*.println("1.Bank Notify To KKM");  
 System.*out*.println("2.GoodsDB Notify To KKM");  
 System.*out*.println("3.KKM Notify To Bank");  
 System.*out*.println("4.KKM Notify To GoodsDB");  
 Scanner sc = *new* Scanner(System.*in*);  
 *boolean* flag = *true*;  
  
 *while* (flag){  
 *int* n = sc.nextInt();  
 *switch* (n){  
 *case* 1: Bank bank = *new* Bank();  
 bank.Notify("KKM");  
 *break*;  
 *case* 2: Goods goods = *new* Goods();  
 goods.Notify("KKM");  
 *break*;  
 *case* 3: KKM kkm = *new* KKM();   
 kkm.Notify("Bank");  
 *break*;  
 *case* 4: KKM kkm1 = *new* KKM();  
 kkm1.Notify("GoodsDB");  
 *break*;  
 *case* 0: flag = *false*;  
 *break*;  
 *default*:  
 *break*;  
 }  
 }  
 System.*out*.println("Good Bye!");  
 }  
}