CSC 631 Object Oriented Software Engineering

Homework 3

Shuowen Wei

1.

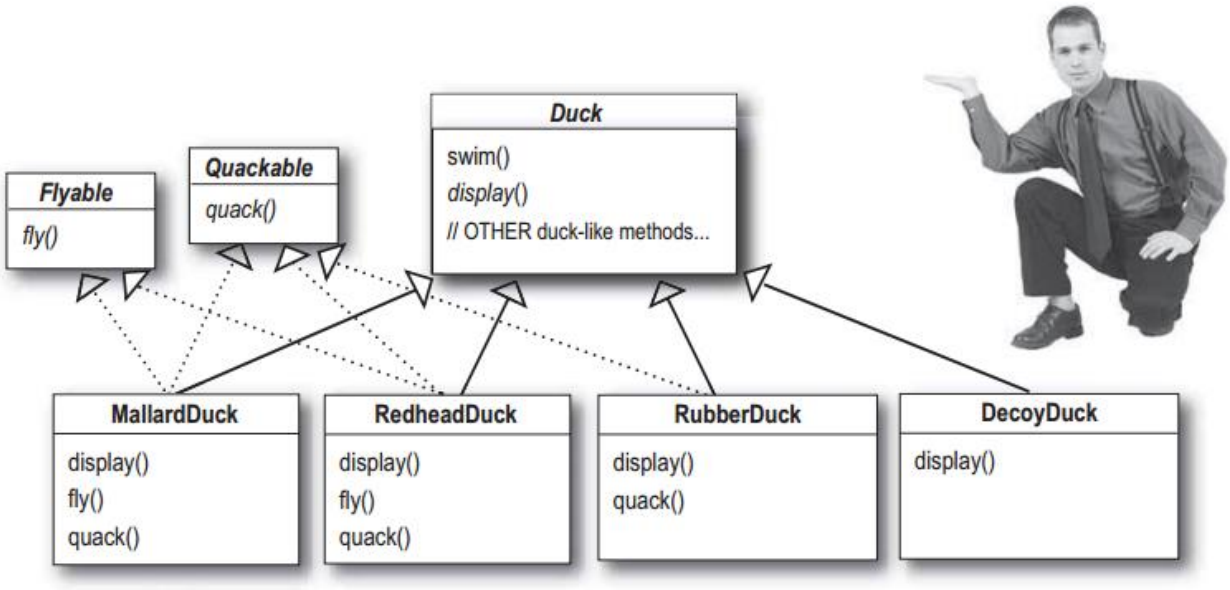
The three design principles are:

(1). Identify the aspects of your application that vary and separate them from what stays the same.

(2). Program to an interface, not an implementation.

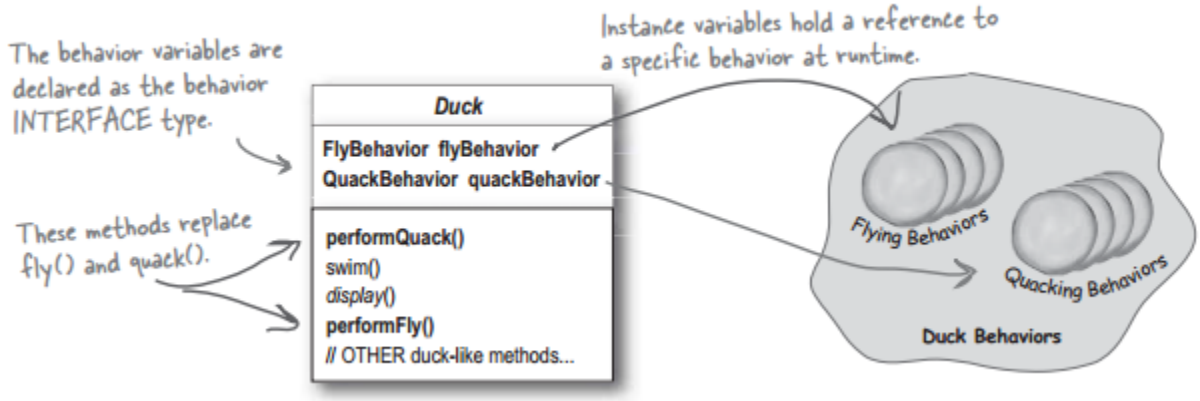
(3). Favor composition over inheritance.

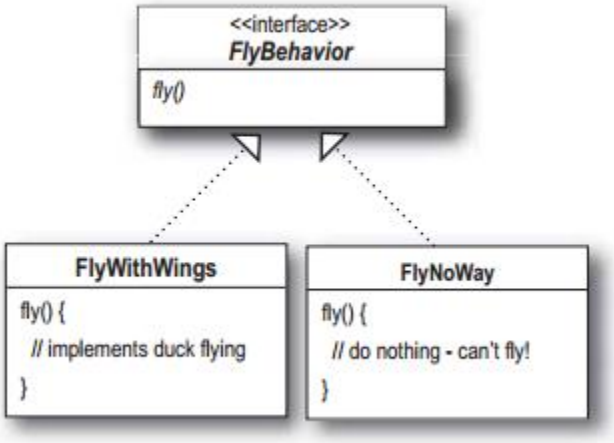
In chapter 1, the strategy pattern brought up an issue that not all subclasses should have all the behaviors (functions) defined in the superclass, so inheritance directly from the superclass is not always working, thus we have to first of all identify that which part of methods are changing and what stays the same, hence we can take the parts that vary and encapsulate them, so that we can alter or extend the parts that vary without affecting those that don't. In the textbook, the ideas of applying interfaces sounds promising at the beginning, see the UNL diagram blow,



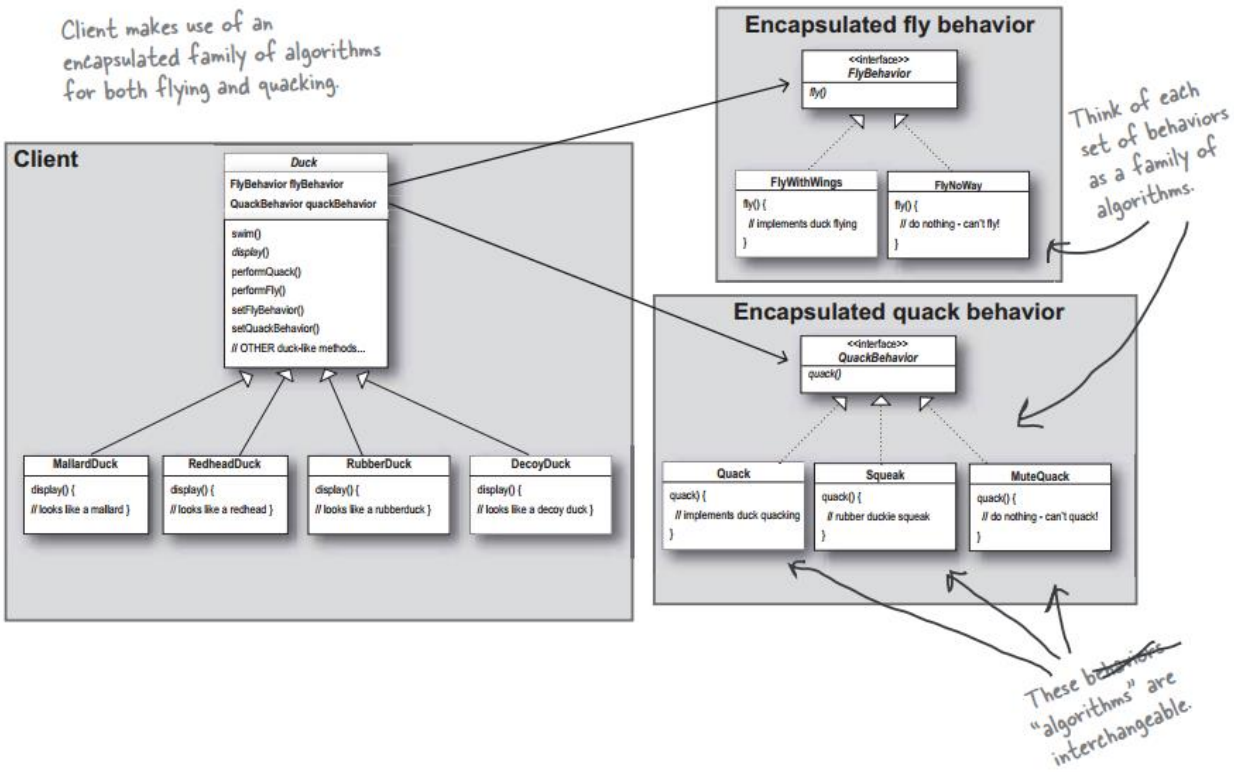
But an new issue was brought in, that it completely destroys code reuse for those methods (since Java interfaces have no implementation code). Luckily, the principles 2 and 3 saved it. We

already know that we should separate fly() and quack() behaviors from Duck class, now we can use an interface to represent each behavior, which is to program to an interface, not an implementation, see the UML diagrams below:





Hence, the Duck behaviors will live in a separate class-----a class that implements a particular behavior interface. That way, the Duck classes won't need to know any of the implementation details for their own behaviors.



Here, when putting two classes together like above UML diagram, this is composition. Instead of inheriting the behaviors, the ducks get their behavior by being composed with the right behavior object. This brings a lot more flexibility. Not only does it let you encapsulate a family of algorithms into their own set of classes, but it also lets you change behavior at runtime as long as the object you're composing with implements the correct behavior interface.

2.

The Strategy pattern is to be used where you want to choose the algorithm to use at runtime. A good use of the Strategy pattern would be saving files in different formats, running various sorting algorithms, or file compression.

Here we use the example of a file compression tool - where we create either *.zip* or *.rar* files.

```
1.//Strategy Interface
2.public interface CompressionStrategy{
3.
4.    public void compressFiles(ArrayList<File> files);
5.}
```

And we'll need to provide our two implementations, one for *.zip* and one for *.rar*

```
01.public class ZipCompressionStrategy implements CompressionStrategy{
02.
03.        public void compressFiles(ArrayList<File> files){
04.              //using ZIP approach
05.              }
06.}
```

```
01.public class RarCompressionStrategy implements CompressionStrategy{
02.
03.   public void compressFiles(ArrayList<File> files){
04.              //using RAR approach
05.        }
06.}
```

Our context will provide a way for the client to compress the files. Let's say that there is a preferences setting in our application that sets which compression algorithm to use. We can change our strategy using the setCompressionStrategy method in the Context, as below

```
01.public class CompressionContext{
02.
03.   private CompressionStrategy strategy;
04.
05.   //this can be set at runtime by the application preferences
06.   public void setCompressionStrategy(CompressionStrategy strategy){
07.        this.strategy = strategy;
08.        }
09.
10.   //use the strategy
11.   public void createArchive(ArrayList<File> files){
12.        strategy.compressFiles(files);
13.        }
14.}
```

Now, all the client has to do now is to pass through the files to the CompressionContext

```
01.public class Client{
02.   public static void main(String[] args){
03.
04.   CompressionContext ctx = new CompressionContext();
05.   //we could assume context is already set by preferences
06.   ctx.setCompressionStrategy(new ZipCompressionStrategy());
07.   //get a list of files
08.   ...
09.   ctx.createArchive(fileList);
10.   }
11.}
```

3.

Duckbehavior.cpp is attached here:

```
// Homework 3
// by Shuowen Wei

#include <iostream>
using namespace std;

//  Quack and fly interfaces
class QuackBehavior
{
	public:
		virtual void quack() = 0;
};

class FlyBehavior
{
	public:
		virtual void fly() = 0;
};

// concrete Quack and fly behaviors
class Quack : public QuackBehavior
```

```
{
	public:
		void quack()
		{
			cout << "Quack" << endl;
		}
};

class MuteQuack : public QuackBehavior
{
	public:
		void quack()
		{
			cout << "<<Silence>>" << endl;
		}
};

class Squack : public QuackBehavior
{
	public:
		void quack()
		{
			cout << "Squack" << endl;
		}
```

```
};

class FlyWithWings : public FlyBehavior
{
	public:
		void fly()
		{
			cout << "I'm flying!" << endl;
		}
};

class FlyNoWay : public FlyBehavior
{
	public:
		void fly()
		{
			cout << "I can't fly!" << endl;
		}
};

class FlyRocketPowerwd : public FlyBehavior
{
	public:
		void fly()
```

```
		{
			cout << "I'm flying with a rocket!" << endl;
		}
};

// the Duck superclass
class Duck
{
	public:
		QuackBehavior *quackBehavior;
		FlyBehavior   *flyBehavior;

		virtual void display() = 0;

		void performFly()
		{
			flyBehavior->fly();
		}

		void performQuack()
			{
				quackBehavior->quack();
			}
```

```
        void setQuackBehavior(QuackBehavior *qb)
        {
                cout << "Changing quack behavior..." << endl;
                quackBehavior = qb;
        }

        void setFlyBehavior(FlyBehavior *fb)
        {
                cout << "Changing fly behavior..." << endl;
                flyBehavior = fb;
        }

        void swim()
        {
                cout << "All ducks float, even decoys!" << endl;
        }

};


class MallardDuck : public Duck
{
        public:
                MallardDuck()
```

```
		{
			quackBehavior = new Quack();
			flyBehavior   = new FlyWithWings();
		}

		void display()
		{
			cout << "I'm a real Mallard duck!" << endl;
		}
};

class ModelDuck : public Duck
{
	public:
		ModelDuck()
		{
			flyBehavior   = new FlyNoWay();
			quackBehavior = new Quack();
		}

		void display()
		{
			cout << "I'm a model duck" << endl;
		}
```

```
};


// main function is here
int main()
{
	Duck *mallard = new MallardDuck();
	mallard->performQuack();
	mallard->performFly();


	Duck *model = new ModelDuck();
	model->performFly();
	model->setFlyBehavior(new FlyRocketPowerwd);
	model->performFly();


	return 0;
}
```