HACETTEPE UNIVERSITY
DEPARTMENT OF COMPUTER ENGINEERING

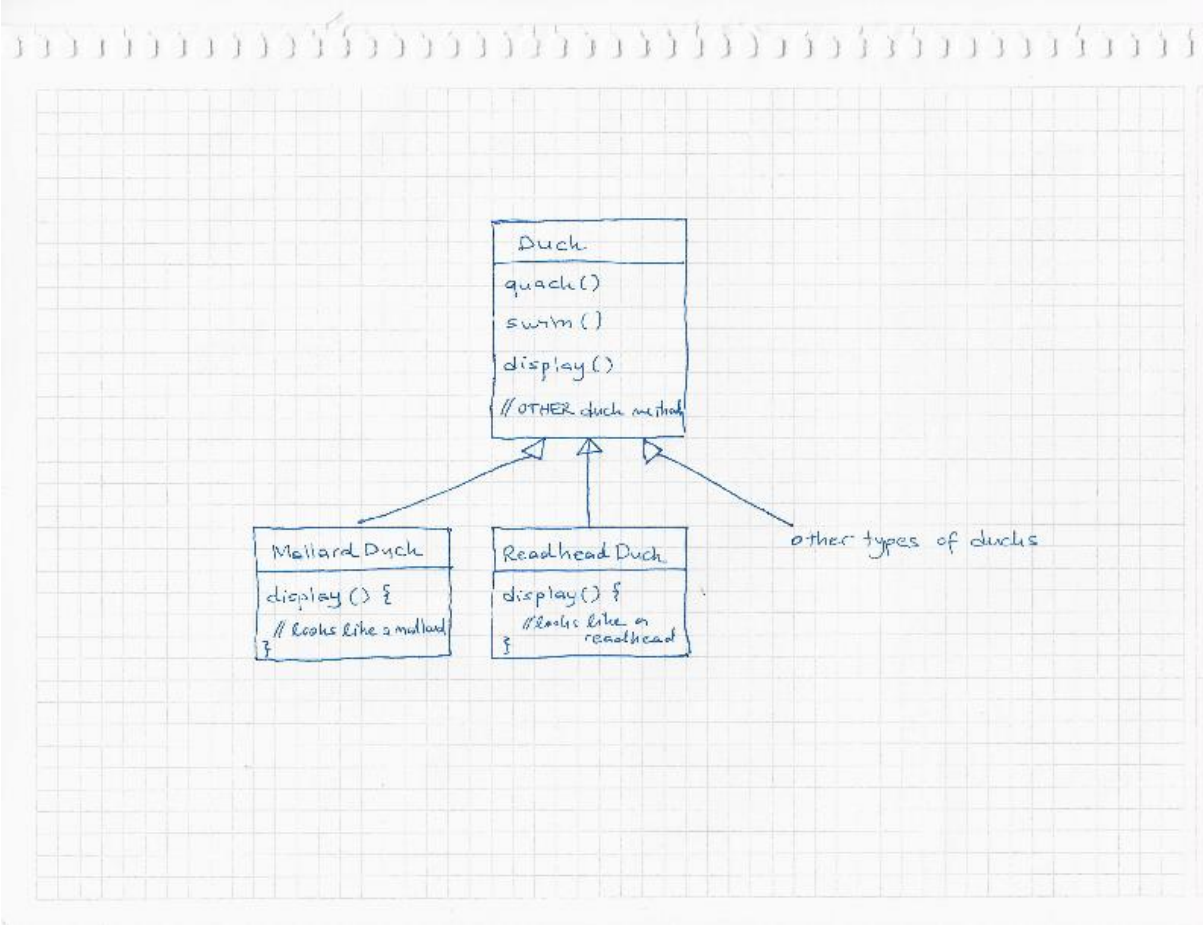
BBM 486 – DESIGN PATTERNS

2. THE STRATEGY PATTERN

The strategy pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

We will illustrate the strategy pattern with an example. You are building a duck pond simulation game, *SimUDuck*. The game can show a large variety of duck species swimming and making quacking sounds.

1. The initial design of the system used standard OO techniques and created one Duck superclass from which all other duck types inherit: All ducks quack and swim, the superclass takes care of the implementation code. The display() method is abstract, since all duck subtypes look different. Each duck subtype is responsible for implementing its own display() behavior for how it looks on the screen.



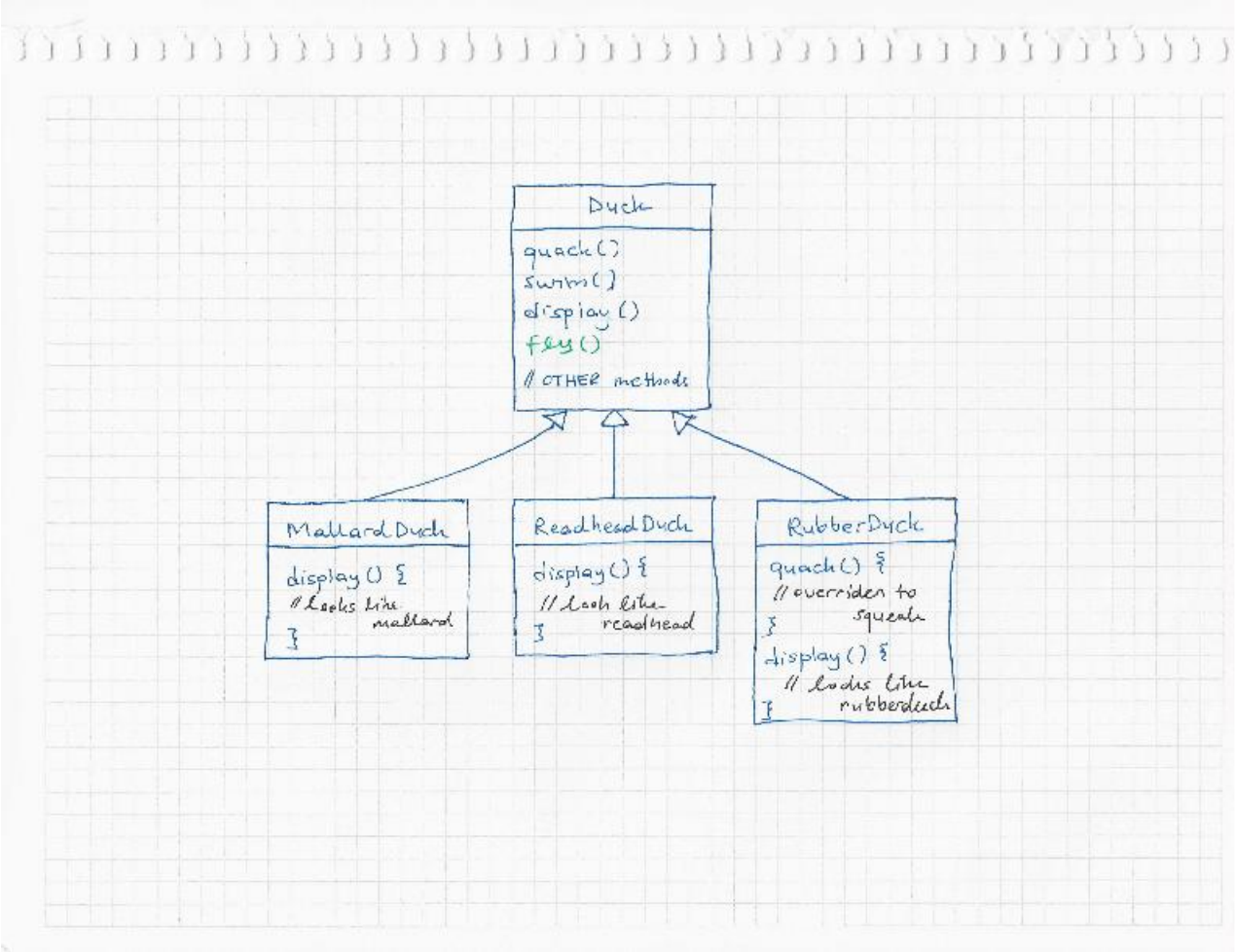
2. We received a request to add a new feature where ducks can fly. We consider adding the fly() method in the Duck class and then all the ducks will inherit it. This is a good use of inheritance for the purpose of reuse.

3. We however noticed that not all subclasses of duck can fly. Rubber duckies for example cannot fly. By putting fly() in the superclass, we gave flying ability to all ducks, including those that should not. Rubber ducks pose another problem that they don't quack. So we overrode them to squeak.

4. Which of the following are disadvantages of using inheritance to provide Duck behavior?

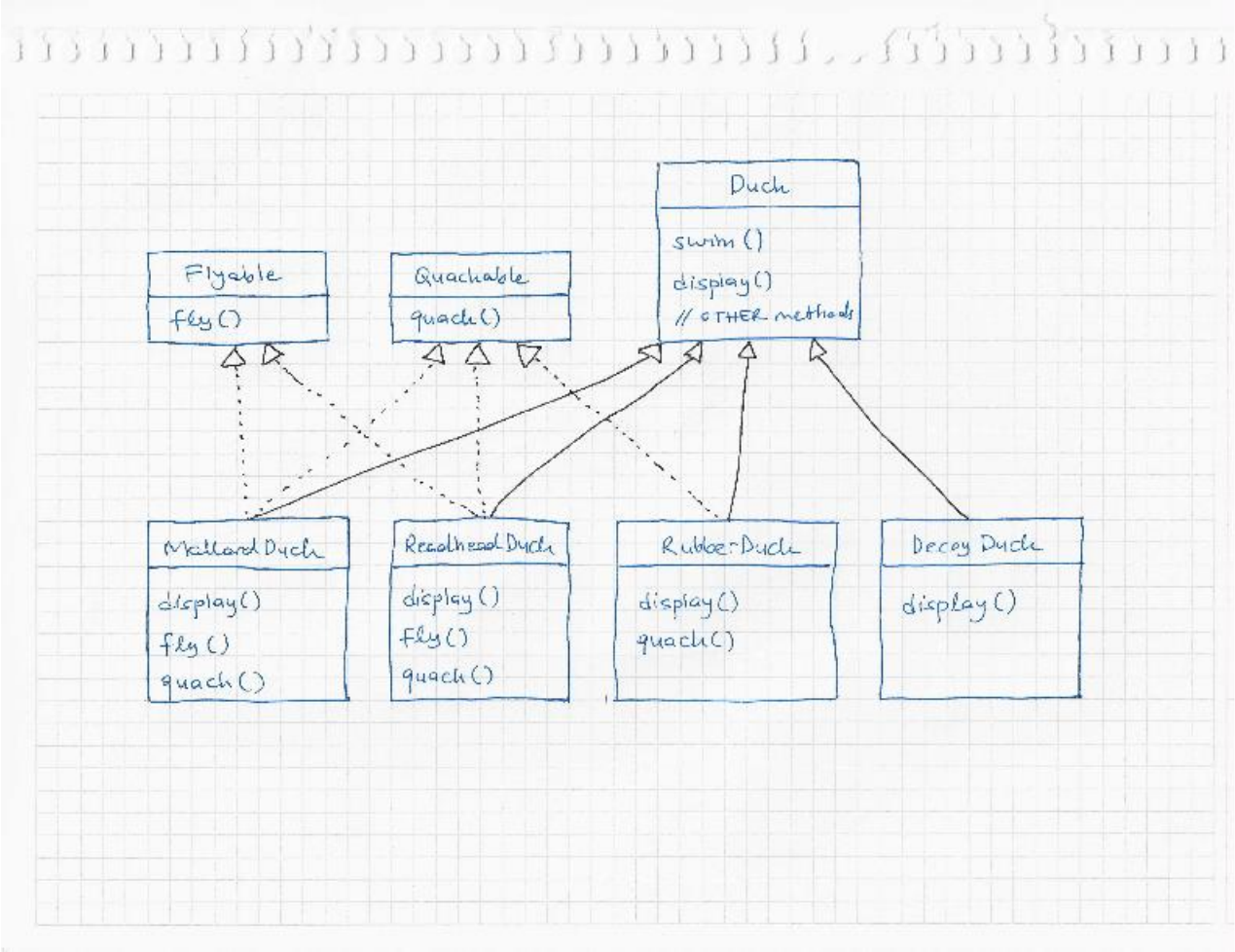
   A. Code is duplicated across subclasses. X
   B. Runtime behavior changes are difficult. X
   C. We cannot make ducks dance.
   D. Hard to gain knowledge of all duck behaviors. X
   E. Ducks cannot fly and quack at the same time.
   F. Changes can unintentionally affect other ducks. X

5. We thought about overriding the fly() method (to do nothing) in rubber duck, the way we did with the quack() method. But then what happens when we add wooden decoy ducks to the program? They are not supposed to fly or quack. Can that be another class in the hierarchy where we override both quack() and fly() to do nothing?



6. We realize that inheritance probably is not the answer, because we received a requirement that the product features will be updated every six moths. Since the spec will keep changing, we will be forced to look and possibly override fly() and quack() for every new Duck subclass that is ever added to the program.

7. How about an interface? We can take the fly() out of the Duck superclass, and make a Flyable() interface with a fly() method. That way, only the ducks that are supposed to fly will implement that interface and have a fly() method, and we might as well make a Quackable() too, since not all ducks can quack.



8. However, while having the subclasses implement Flyable and/or Quackable solves part of the problem (no inappropriately flying rubber ducks), it completely destroys code reuse for those behaviors, so it just creates a different maintenance nightmare. Moreover, there might be more than one kind of flying behavior even among the ducks that do fly.

9. The one constant in software development. No matter where you work, what you are building, or what language you are programming in, what's the one true constant that will be with you always? CHANGE.

10. Lots of things can drive change:
    Customers or users may decide they want something else or they want new functionality.
    Company may decide to go with another database vendor and purchase its data from another supplier that uses a different data format.
    Technology may change and we have got to update our code to make use of protocols.
    We may have learned enough building our system that we'd like to go back and do things a little better.

11. We know using inheritance has not worked out well, since the duck behavior keeps changing across subclasses, and it's not appropriate for all subclasses to have those behaviors. The Flyable and Quackable interface sounded promising at first, except **Java interfaces have no implementation code, so no code reuse**. That means whenever you need to modify a behavior, you are forced to track down and change it in all the different subclasses where that behavior is defined, probably introducing new bugs along the way.

12. **Design Principle**: Identify the aspects of your application that vary and separate them from what stays the same.

13. Take the parts that vary and "encapsulate" them, so that later you can alter or extend the parts that vary without affecting those that don't. We know that fly() and quack() are the parts of the Duck class that vary across ducks. To separate these behaviors from the Duck class, we will pull both methods out of the Duck class and create a new set of classes to represent each behavior.

14. **Design Principle**: Program to an interface, not an implementation.

15. We'll use an interface to represent each behavior (FlyBehavior and QuackBehavior), and each implementation of a behavior will implement one of those interfaces. So, this time it won't be the Duck classes that will implement the flying and quacking interfaces. Instead, we will make a set of classes whose entire reason for living is to represent a behavior, and it is the behavior class, rather than the Duck class, that will implement the behavior interface.

16. This is in contrast to the way we were doing before, where a behavior either came from a concrete implementation in the superclass Duck, or by providing a specialized implementation in the subclass itself. In both cases, we were locked into using that specific implementation and there was no room to for changing out the behavior other than writing more code.

17. The word interface is overloaded here. There is the concept of interface, but there is also the Java construct `interface`. You can program to an interface, without having to actually use a Java `interface`. The point is to exploit polymorphism by programming to a supertype so that the actual runtime object isn't locked into the code.

18. Here is a simple example of using a polymorphic type – imagine an abstract class Animal, with two concrete implementations, Dog and Cat. **Programming to an implementation** would be:

```
Dog d = new Dog();
d.bark();
```

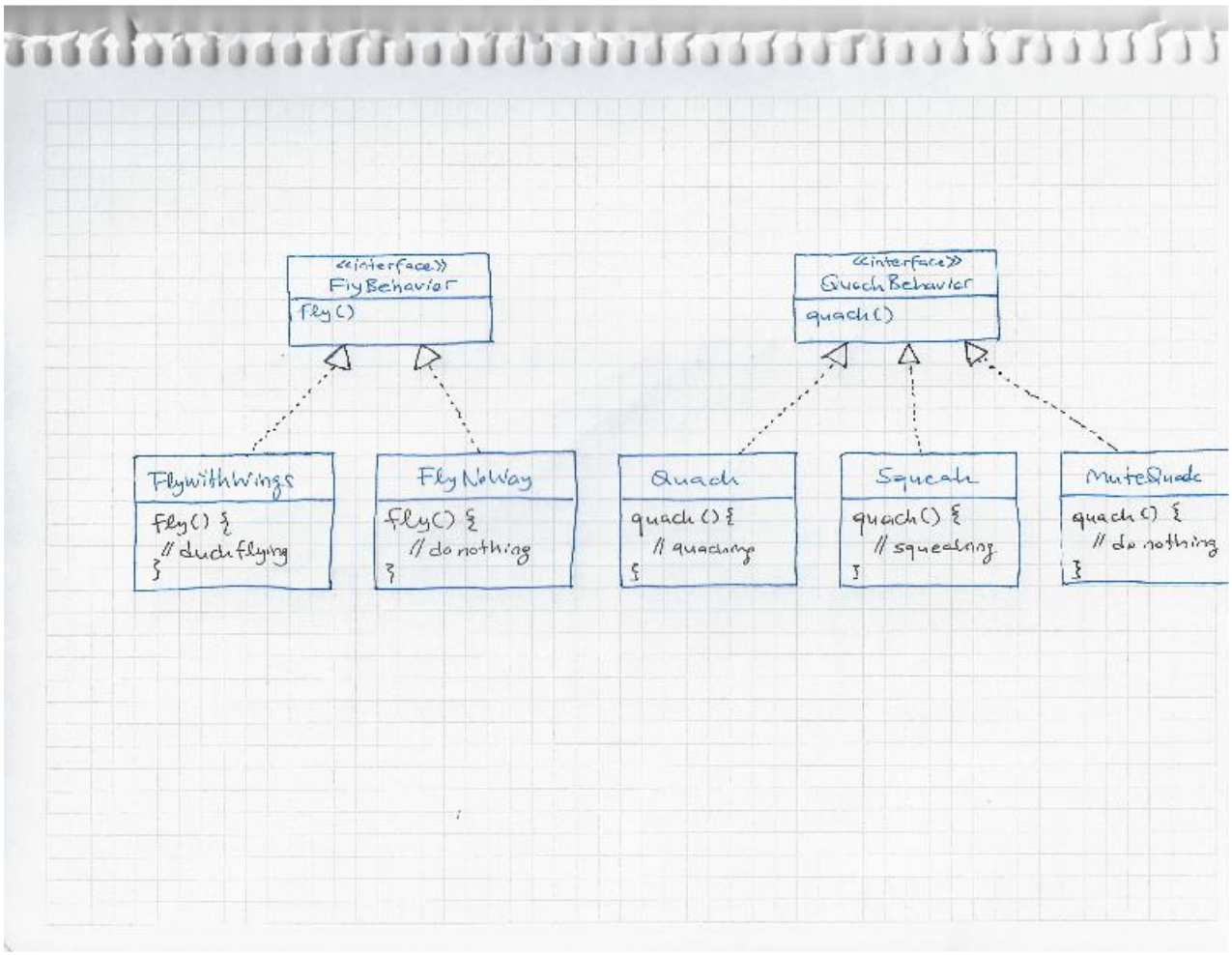
    But **programming to an interface/supertype** would be:

```
Animal animal = new Dog();
Animal.makeSound();
```

    Even better, rather than hard-coding the instantiation of the subtype (like new Dog()) into the code, **assign the concrete implementation object at runtime**:

```
a = getAnimal();
a.makeSound();
```

19. With this design, other types of objects can reuse our fly and quack behaviors because these behaviors are no longer hidden away in our Duck classes. And we can add new behaviors without modifying any of our existing behavior classes or touching any of the Duck classes that use flying behaviors.



20. Integrating the Duck Behavior: The key is that a Duck will now delegate its flying and quacking behavior, instead of using quacking and flying methods defined in the Duck class (or subclass).
    - **First, we will add two instance variables** to the Duck class called flyBehavior and QuackBehavior, that are declared as the interface type. Each duck object will set these variables polymorphically to reference the specific behavior type it would like at runtime (FlyWithWings, Squeak, etc.).
      We also remove the fly() and quack() methods from the Duck class, because we have moved this behavior out into the FlyBehavior and QuackBehavior classes.
      We will replace fly() and quack() in the Duck class with two similar methods, called performFly() and performQuack().

- **Now we implement performQuack()**:

```
public class Duck {
    QuackBehavior quackBehavior;
    // more
    Public void performQuack() {
            quackBehavior.quack();
    }
}
```

- Time to worry about **how the flyBehavior and quackBehavior instance variables are set**:

```
public class MallardDuck extends Duck {

    public MallardDuck() {
            quackBehavior = new Quack();
            flyBehavior = new FlyWithWings();
    }

    public void display() {
            System.out.println("I'm a real Mallard duck");
    }
}
```

When a MallardDuck is instantiated, its constructor initializes the MallardDuck's inherited quackBehavior instance variable to a new instance of type Quack. And the same is true for the duck's flying behavior.

21. Testing the Duck code
    - **Type and compile the Duck class below (Duck.java), and the MalardDuck class (MallardDuck.java).**

```
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!");
    }
}
```

- **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'm flying!!");
    }
}

public class FlyNoWay implements FlyBehavior {
    public void fly() {
        System.out.println("I can't fly");
    }
}
```

- **Type and compile the QuackBehavior interface (QuackBehavior.java), and the three behavior implementation classes (Quack.java, MuteQuack.java and Squeak.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 MiniDuckSimulator1 {
      public static void main(String[] args) {
            Duck mallard = new MallardDuck();
            mallard.performQuack();
            mallard.performFly();
      }
}
```

- **Run the code!**

22. Setting Behavior Dynamically: Imagine you want to set the Duck's behavior type through a setter method on the duck subclass, rather than by instantiating it in the duck's constructor:

- **Add two new methods to the Duck class:**

```
public void setFlyBehavior (FlyBehavior fb) {
       flyBehavior = fb;
}

public void setQuackBehavior(QuackBehavior qb) {
      quackBehavior = qb;
}
```

- **Make a new Duck type ModelDuck:**

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

      public void display() {
            System.out.println("I'm a model duck");
      }
```

```
}
```

- **Make a new FlyBehavior type FlyRocketPowered:**

```
public class FlyRocketPowered implements FlyBehavior {
    public void fly() {
        System.out.println("I'm flying with a rocket");
    }
}
```

- **Change the test class, add the ModelDuck and make the ModelDuck rocket-enabled:**

```
public class MiniDuckSimulator1 {
    public static void main(String[] args) {
        Duck mallard = new MallardDuck();
        mallard.performQuack();
        mallard.performFly();

        Duck model = new ModelDuck();
        model.performFly();
        model.setFlyBehavior(new FlyRocketPowered());
        model.performFly();
    }
}
```

- **Run it!**

23. The Big Picture on encapsulated behaviors

    Diagram in the class

24. HAS-A can be better than IS-A. Each duck has a FlyBehavior and a QuackBehavior to which it delegates flying and quacking. When you put two classes together like this, you are using **composition**. Instead of inheriting their behavior; the ducks get their behavior by being composed with the right behavior object.

25. **Design Principle**: Favor composition over inheritance.

26. Composition gives you 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 are composing with implements the correct behavior interface.

Composition is used in many design patterns and you will see a lot more about its advantages and disadvantages throughout the class.