DESIGN PATTERNS I

Design Pattern, Defined

- “A solution to a problem in a context.”
- “A language for communication solutions with others.”

- Pattern languages exist for many problems, but we focus on design
- Best known: “Gang of Four” (Gamma, Helm, Johnson, Vlissides)
  - *Design Patterns: Elements of Reusable Object-Oriented Software*

Caveats

- Design patterns are not a substitute for thought
- Class names and directory structures do not equal good design
- Design patterns have tradeoffs
  - It does not completely remove complexity in interactions but just provides a structure for centralizing it.
- Design patterns depend on the programming language
  - Certain language restrictions may necessitate certain patterns (e.g., patterns related to object creation and destruction)

Motivation for Design Patterns

- They provide an abstraction of the design experience
  - Can often serve as a reusable base of experience
- They provide a common vocabulary for discussing complete system designs
- They reduce system complexity by naming abstractions
  - Thereby increasing program comprehension and reducing learning time with a new piece of code
- They provide a target for the reorganization or refactoring of class hierarchies

THAT FIRST PATTERN

SimUDuck

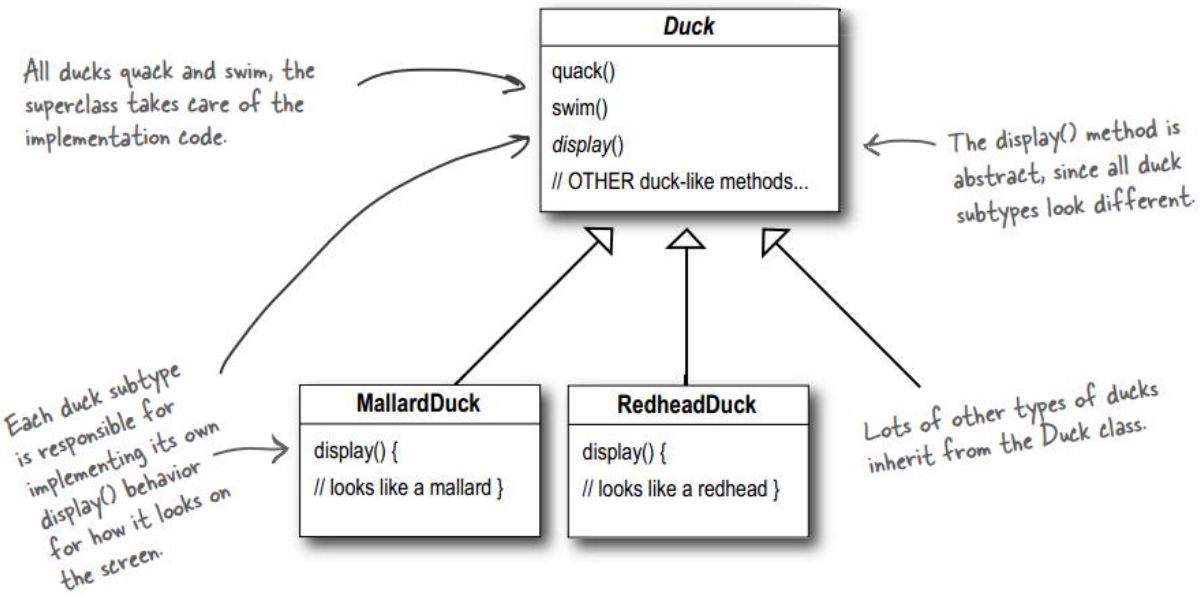
The ducks should be able to swim!

We should have Mallard ducks!

The ducks should quack!

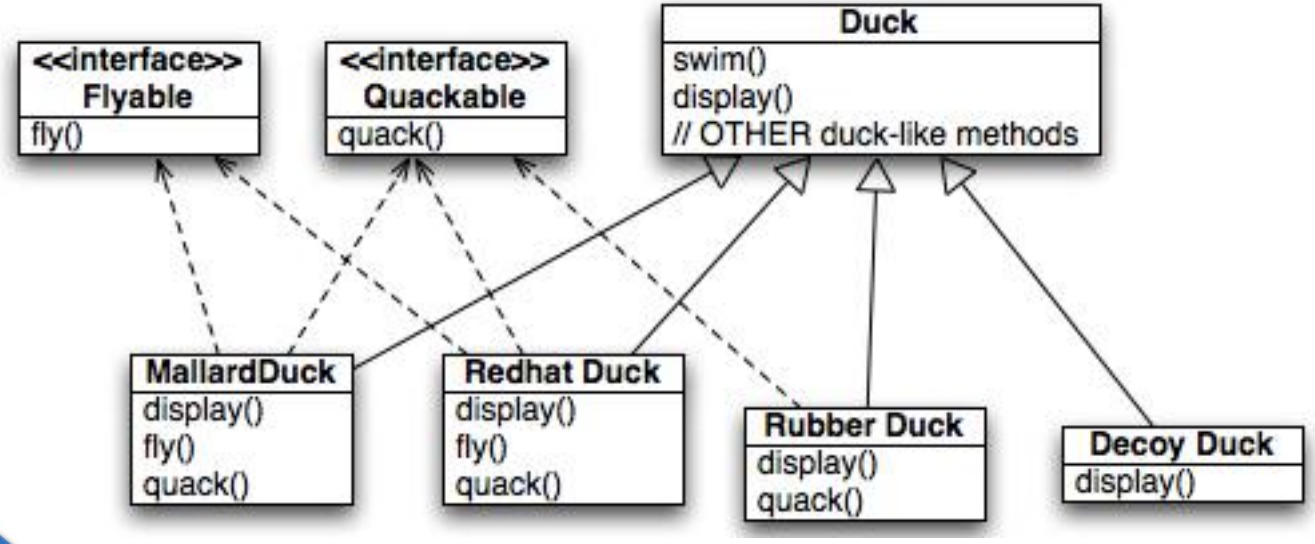
What about a redheaded duck?

Don't ducks fly, too?



SimUDuck, Take 2

Let's use an interface!



I need to control the quacking volume. Make the quack method take a volume.

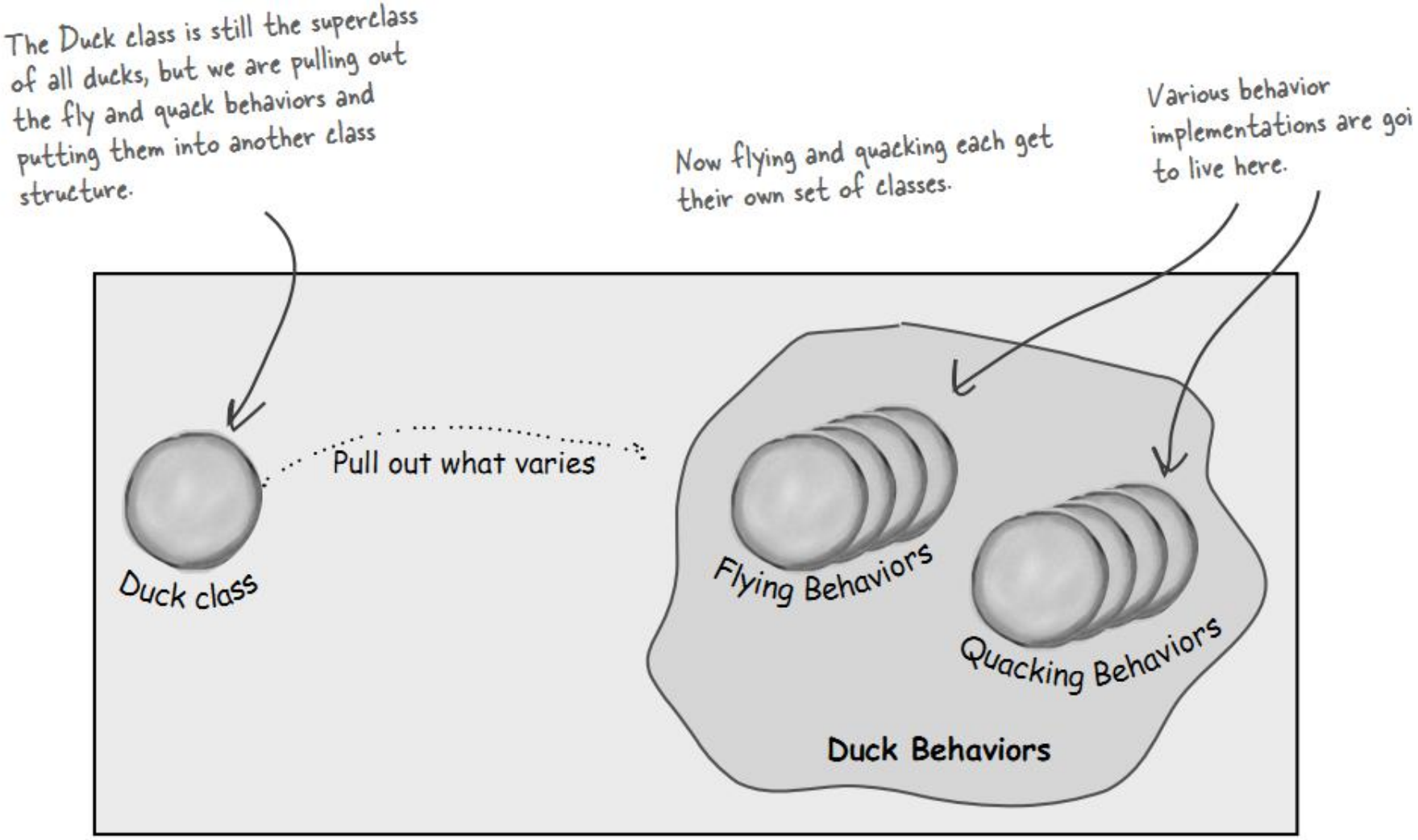
Design Principles

- **Identify the aspects of your application that vary and separate them from what stays the same.**
  - Encapsulate what varies
  - Program to an interface, not to an implementation
  - Favor composition over inheritance
- For our example:
  - Pull the duck *behavior* out of the duck *class*

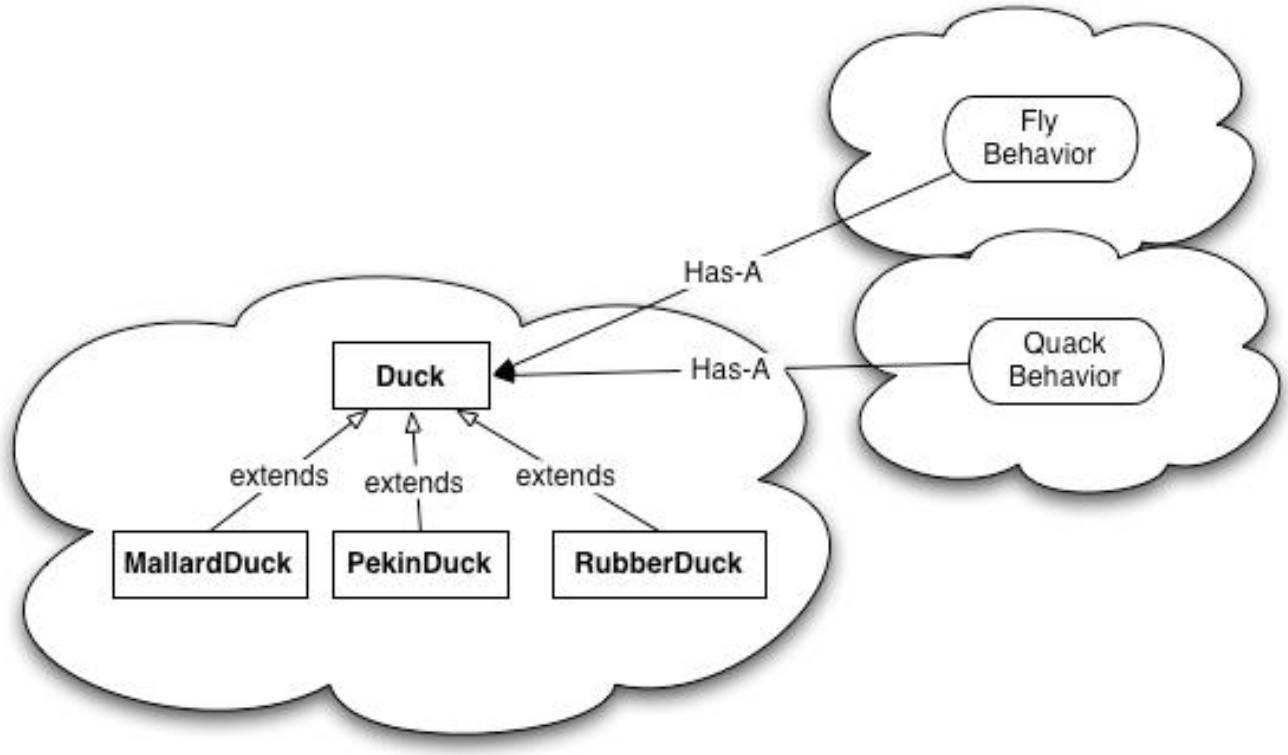
Liskov Substitution Principle

Let q(x) be a property provable about objects x of type T. Then q(y) should be provable for objects y of type S where S is a subtype of T.

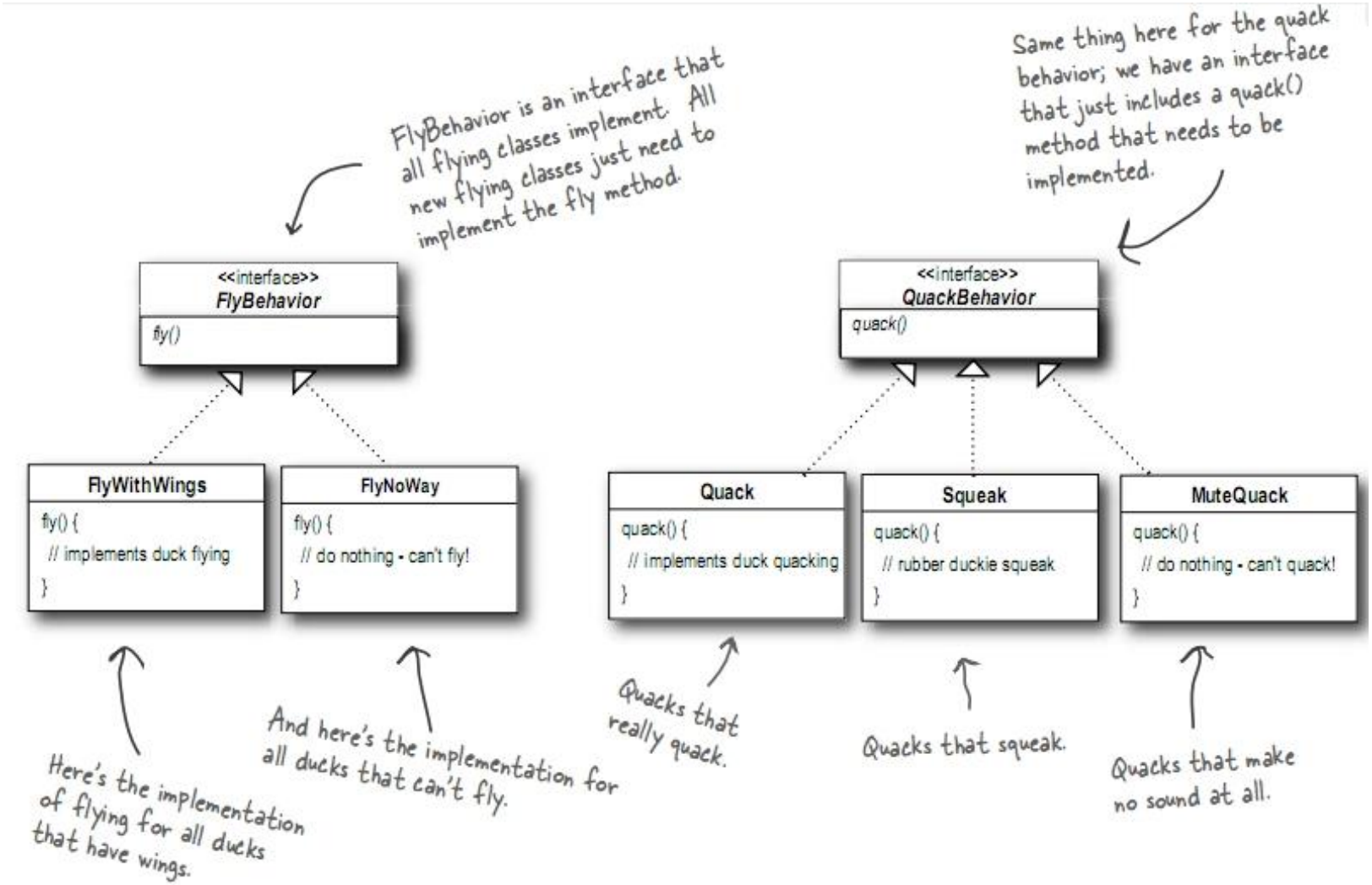
Encapsulate what varies



… or in UML


Fly
Behavior
Has-A
Quack
Behavior
Has-A
Duck
extends
extends
extends
MallardDuck
PekinDuck
RubberDuck

Program to an interface

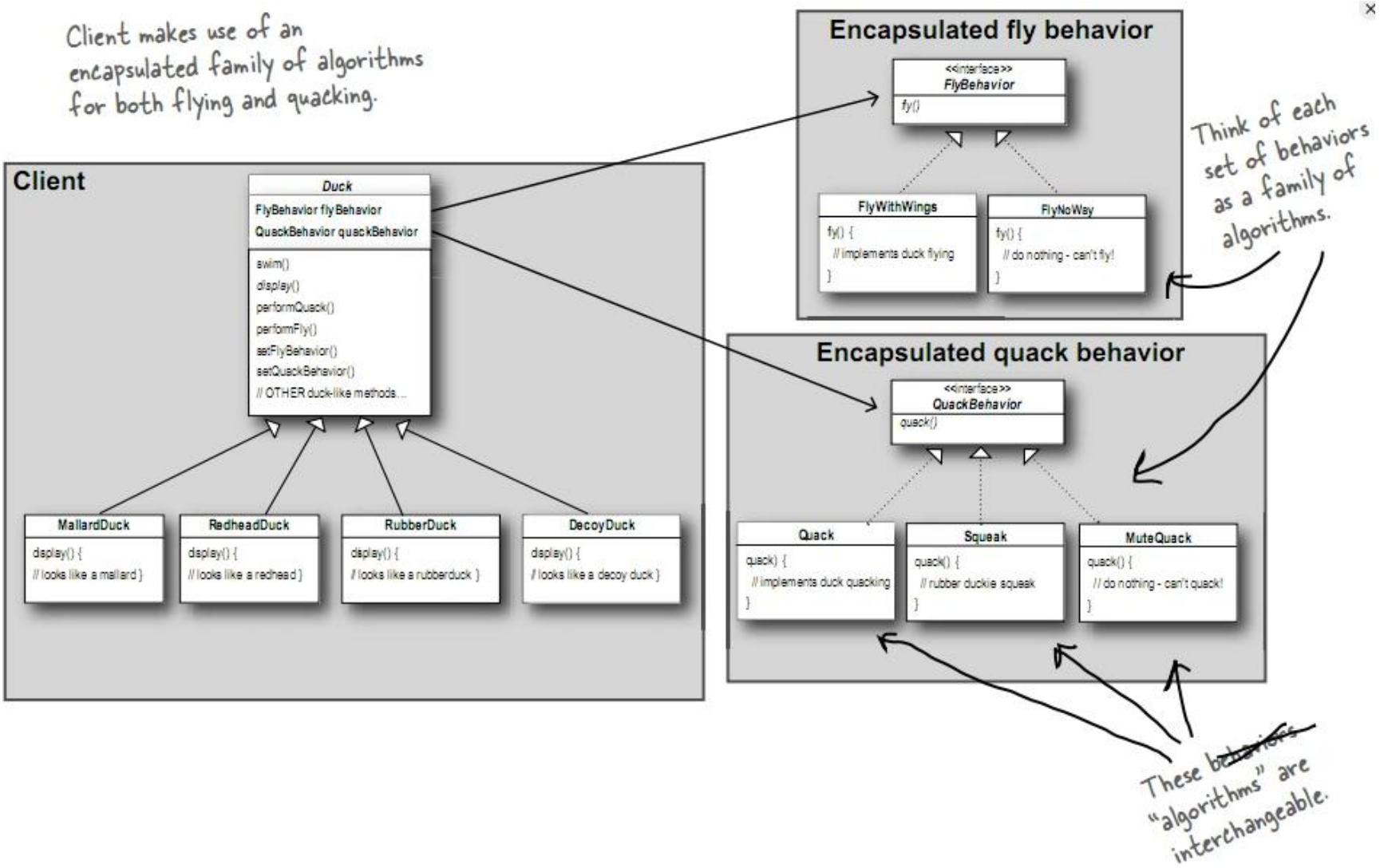


Extension and Reuse

- Ducks ***delegate*** the flying and quacking behaviors

- Now, other classes can use our quacking and flying behaviors since they're not specific to ducks
  - Who would want to do that?
- We can easily add new quacking and flying styles without impacting our ducks!

The Big Picture



And what does that really look like?

- Now it’s time for some code…

What's a Duck?

```
public class Duck {
   QuackBehavior quackBehavior;
   // more

   public void performQuack() {
      quackBehavior.quack();
   }
}
```

A duck has a reference to something that implements the QuackBehavior interface

Instead of quacking all on its own, a Duck delegates that behavior to the quackBehavior object

It doesn't matter what **kind** of Duck it is; all that matters is a Duck knows how to quack

How do we make Ducks quack and fly?

```
public interface QuackBehavior {
  public void quack();
}

public class Quack implements QuackBehavior{
  public void quack(){
     System.out.println(“Quack”);
  }
}
```

How do we make Ducks quack and fly?

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

How do we make Ducks quack and fly?

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

But how do Ducks *learn* to quack and fly?

- How could you teach a Duck a new way to quack or a new way to fly?

But how do Ducks *learn* to quack and fly?

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

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

But how do Ducks *learn* to quack and fly?

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

    public void display() {
        system.out.println("I am a model duck.");
    }
}
```

But how do Ducks *learn* to quack and fly?

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

But how do Ducks *learn* to quack and fly?

```
public class MiniDuckSimulator {
  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();
  }
}
```

Favor Composition over Inheritance

- Stated another way… “has-a is better than is-a”

- Duck’s **have** quacking behaviors and flying behaviors
  - Instead of **being** Quackable and Flyable

- Composition is good because:
  - It allows you to encapsulate a family of algorithms into a set of classes (the **Strategy** pattern)
    - The what? Yup, that was your “first” pattern…
  - It allows you to easily change the behavior at **runtime**

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 the clients that use it.

Give it a shot

```
public interface Strategy {
    public int doOperation(int num1, int num2);
}

public class OperationAdd implements Strategy{
    @Override
    public int doOperation(int num1, int num2) {
        return num1 + num2;
    }
}
```

Give it a shot

```
public class OperationSubstract implements Strategy{
   @Override
   public int doOperation(int num1, int num2) {
      return num1 - num2;
   }
}

public class OperationMultiply implements Strategy{
   @Override
   public int doOperation(int num1, int num2) {
      return num1 * num2;
   }
}
```

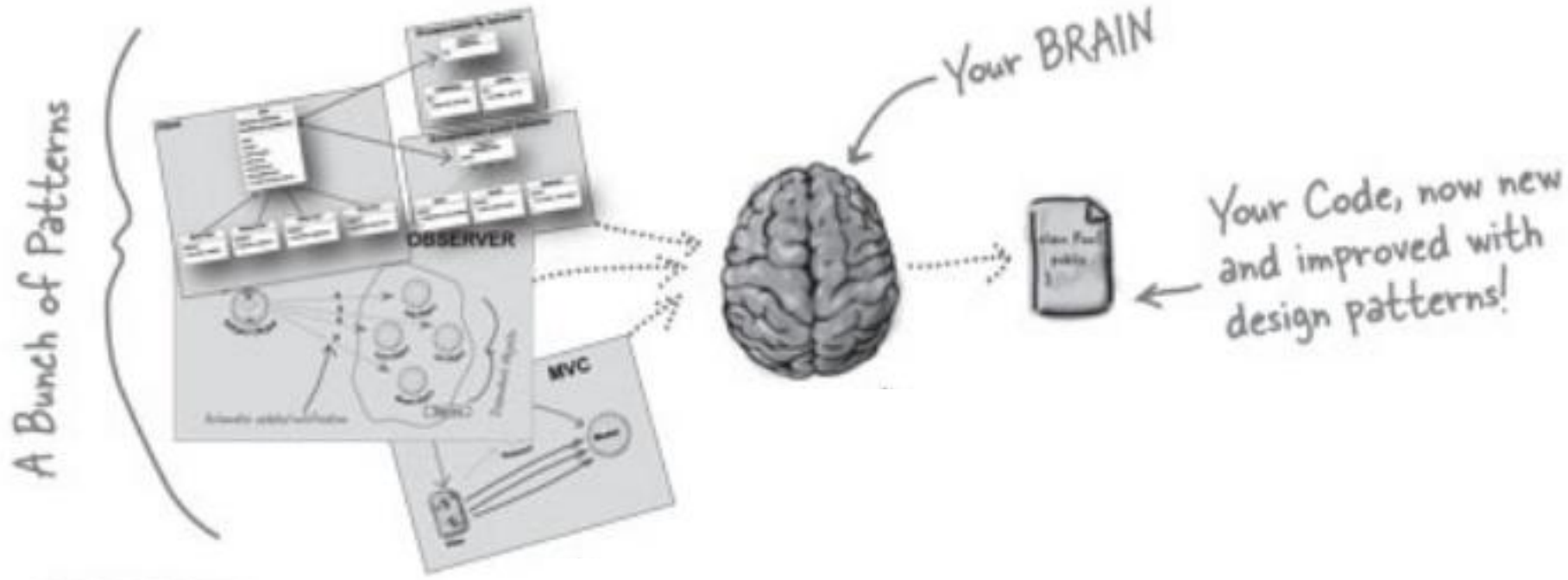
Give it a shot (Your client)

```
public class StrategyPatternDemo {
   public static void main(String[] args) {
      Context context = new Context(new OperationAdd());
      System.out.println("10 + 5 = " + context.executeStrategy(10, 5));

      context = new Context(new OperationSubstract());
      System.out.println("10 - 5 = " + context.executeStrategy(10, 5));

      context = new Context(new OperationMultiply());
      System.out.println("10 * 5 = " + context.executeStrategy(10, 5));
   }
}
```

What is your context.java

- public class Context {
- ???
- }

How to Use Design Patterns



QUESTIONS?