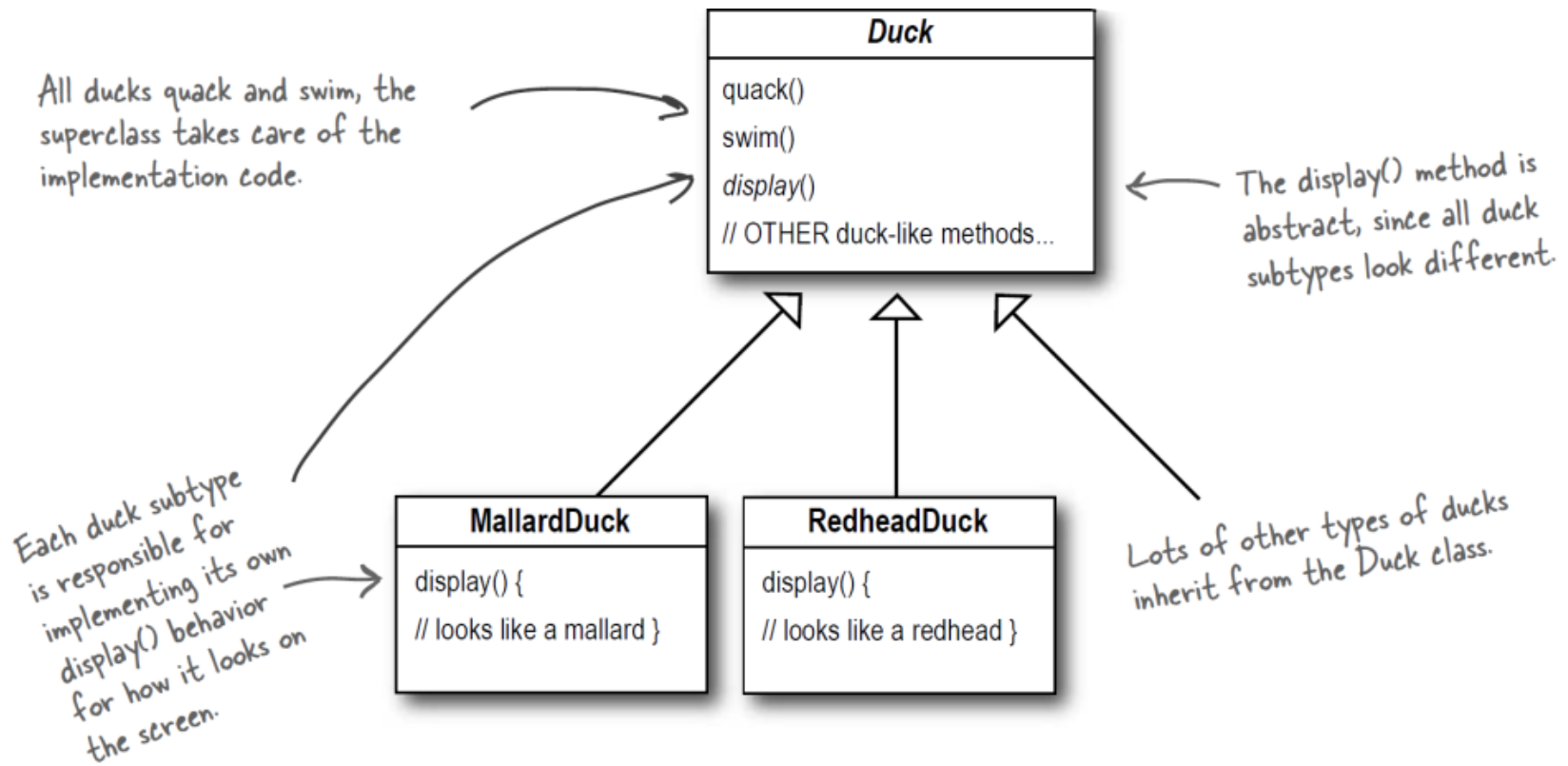


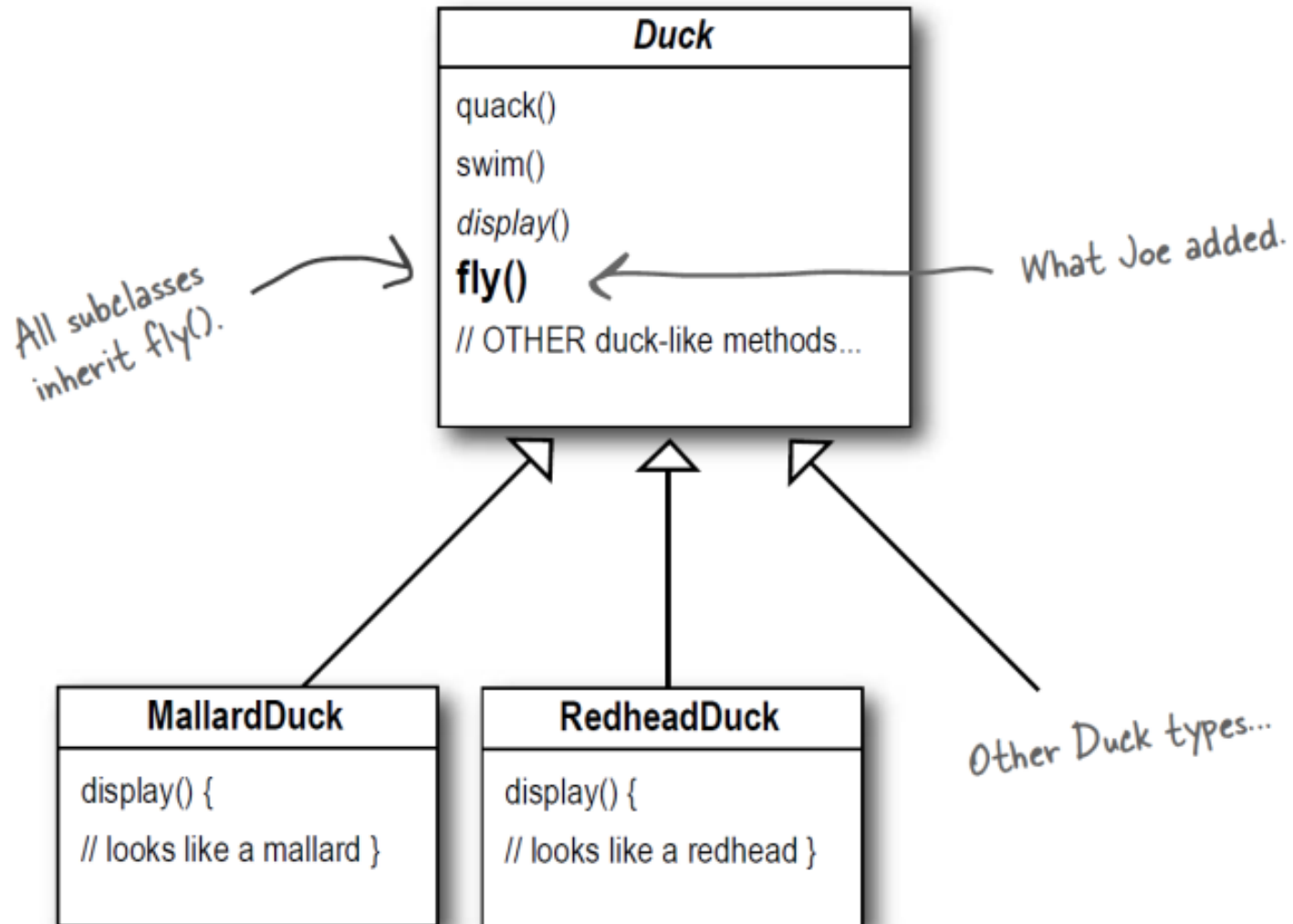


# 设计模式

# It is started with a simple SimUDuck application



# Now we need the ducks to Fly



# But something went horribly wrong

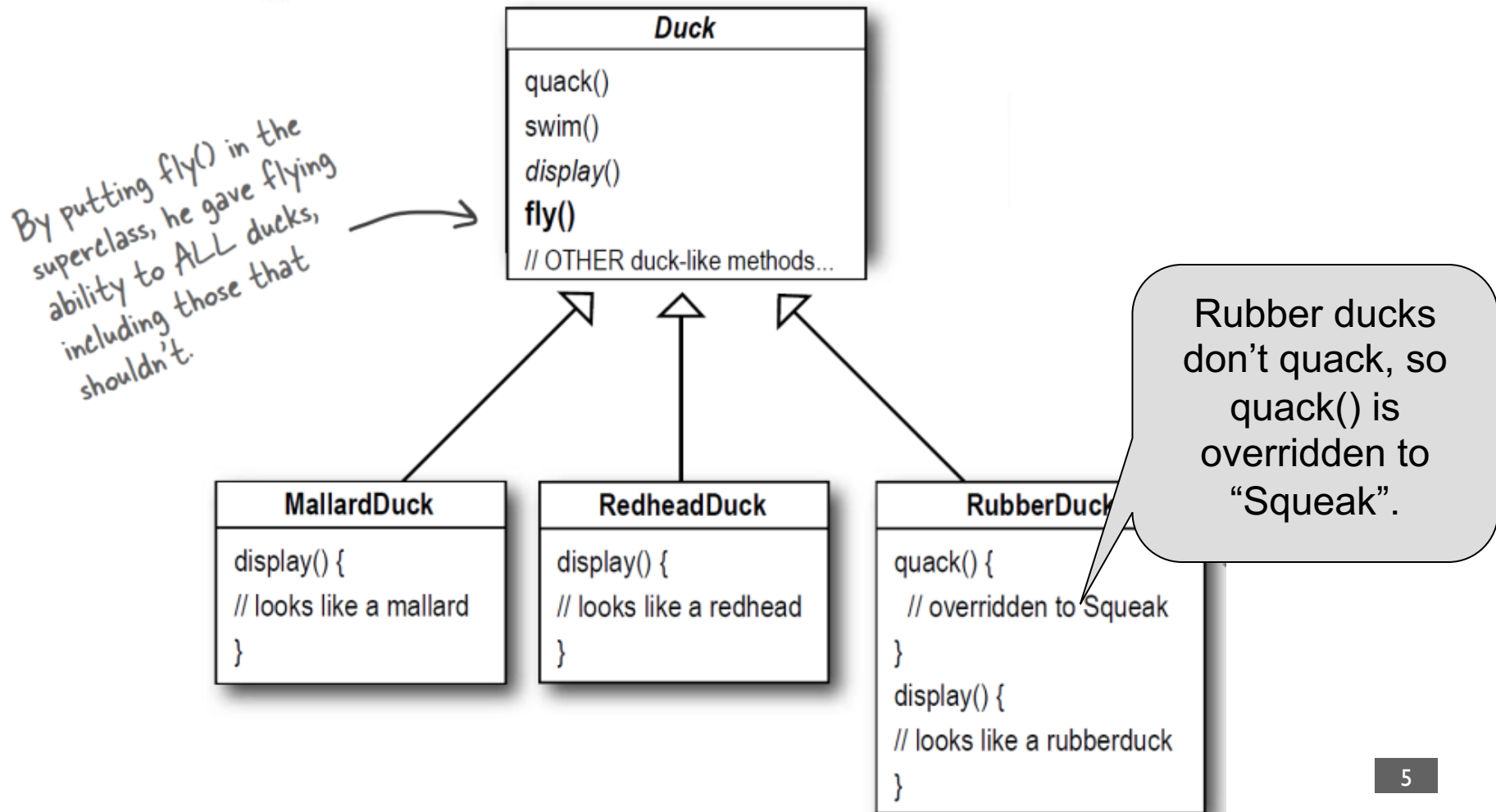
What we  
want



What we get.  
Rubber duckies  
flying



# What happened?



# Think about inheritance

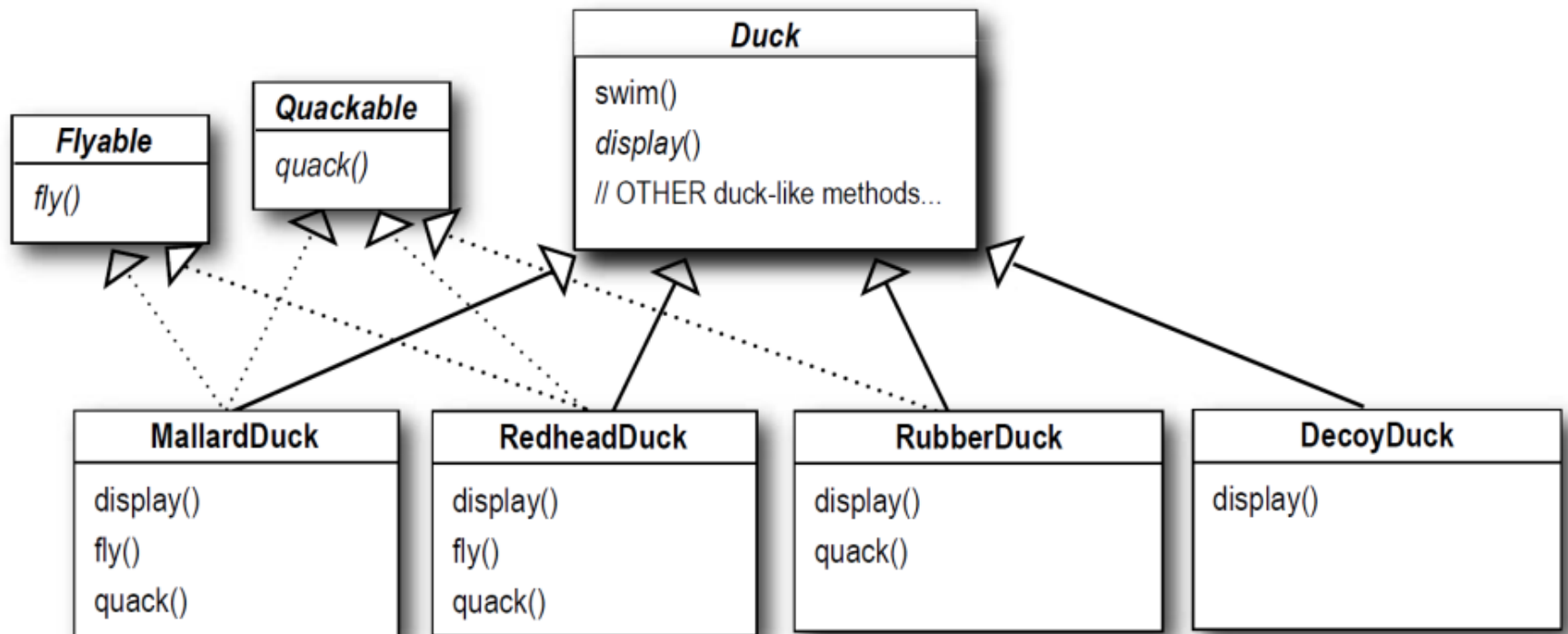
We could always just override the fly() method in rubber duck, the way we are with the quack() method...

```
RubberDuck  
quack() { // squeak}  
display() { .// rubber duck }  
fly() {  
    // override to do nothing  
}
```

But then what happens when we add wooden decoy ducks to the program? They are not supposed to fly or quack...

The executives want to update the product every six months. There will be new Duck subclasses in every update, what a nightmare!

# How about an interface?



## No. Duplicate code

- If you thought having to override a few methods was bad, how are you gonna feel when you need to make change to the flying behavior ... in all 48 of the flying Duck subclasses.
- Change the not-flying subclass (inheritance, override)  
VS.
- Chang the flying subclass (interface)



# Change

- The one constant in software development.
- Lots of things can drive change. List some reasons you've had to change code in your applications.
- Write some change in software development.

# Change

- My customers or users decide they want something else, or they want new functionality.
- My company decided it is going with another database vendor and it is also purchasing its data from another supplier that uses a different data format. Argh!
- Well, technology changes and we've got to update our code to make use of protocols.
- We've learned enough building our system that we'd like to go back and do things a little better.

# Design Principle

- **Encapsulate what varies.**
- *Identify the aspects of your application that vary and separate them from what stays the same.*
- *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.*

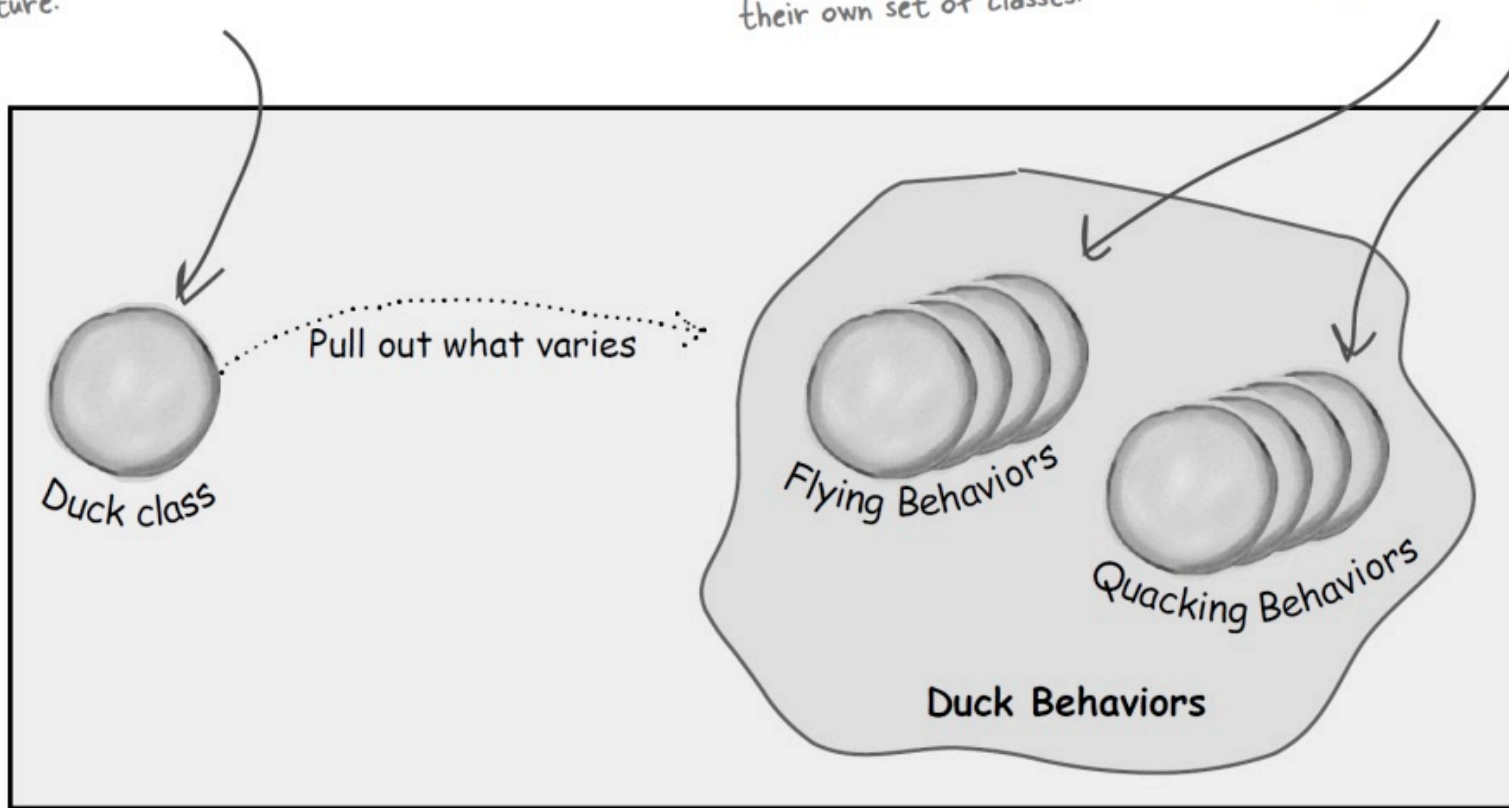
## Separating what changes from what stays the same

- 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'll pull both methods out of the Duck class and create a new set of classes to represent each behavior.

The Duck class is still the superclass of all ducks, but we are pulling out the fly and quack behaviors and putting them into another class structure.

Now flying and quacking each get their own set of classes.

Various behavior implementations are going to live here.



# Designing the Duck Behaviors

- We want:
  - Keep things flexible;
  - Assign behaviors to the instances of Duck;
  - Change the behavior of a duck dynamically;
    - Change the duck's behavior at runtime.

# Design Principle

*Program to an interface, not an implementation.*

# Classes to represent behavior

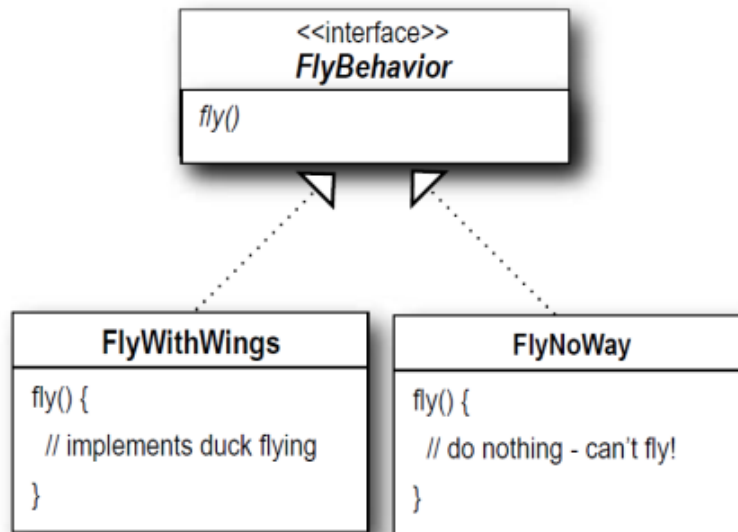
- It won't be the Duck classes that will implement the flying and quacking interfaces.
- We'll make a set of classes whose entire reason for living is to represent a behavior. (Strange?)
- Our first solution, concrete implementation in superclass. Second solution, providing a specialized implementation in the subclass itself. **They are all relying on an implementation.**



## Recall for the polymorphism

- “*Program to an interface*” really means “*program to a supertype*”.
- There is the **concept** of interface, but there’s also the Java construct **interface**. You can *program to an interface*, without having to actually use a Java interface.
- The declared type of the variables should be a supertype, usually an abstract class or interface, so that the objects assigned to those variables can be of any concrete implementation of the supertype, which means the class declaring them doesn’t have to know about the actual object types!

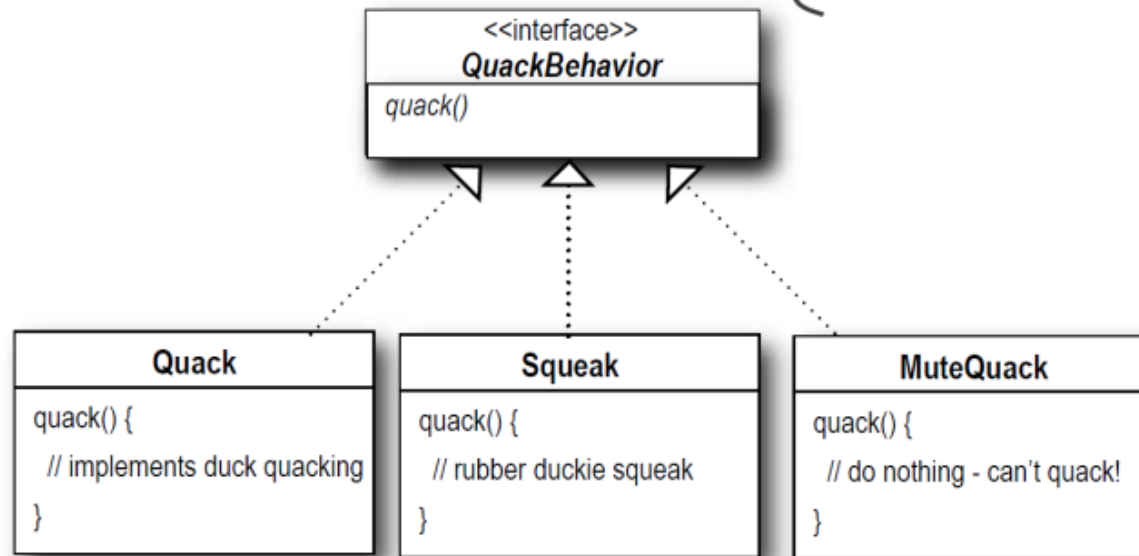
FlyBehavior is an interface that all flying classes implement. All new flying classes just need to implement the fly method.



Here's the implementation of flying for all ducks that have wings.

And here's the implementation for all ducks that can't fly.

Same thing here for the quack behavior; we have an interface that just includes a quack() method that needs to be implemented.



Quacks that really quack.

Quacks that squeak.

Quacks that make no sound at all.

## With the design

- Other type of objects can reuse our fly and quack behavior because these behaviors are no longer hidden away in our Duck classes.
- We can add new behaviors without modifying any of our existing behavior classes or touching any of the Duck classes that use flying behaviors.
- So we get the benefit of REUSE without all the baggage that comes along with inheritance.

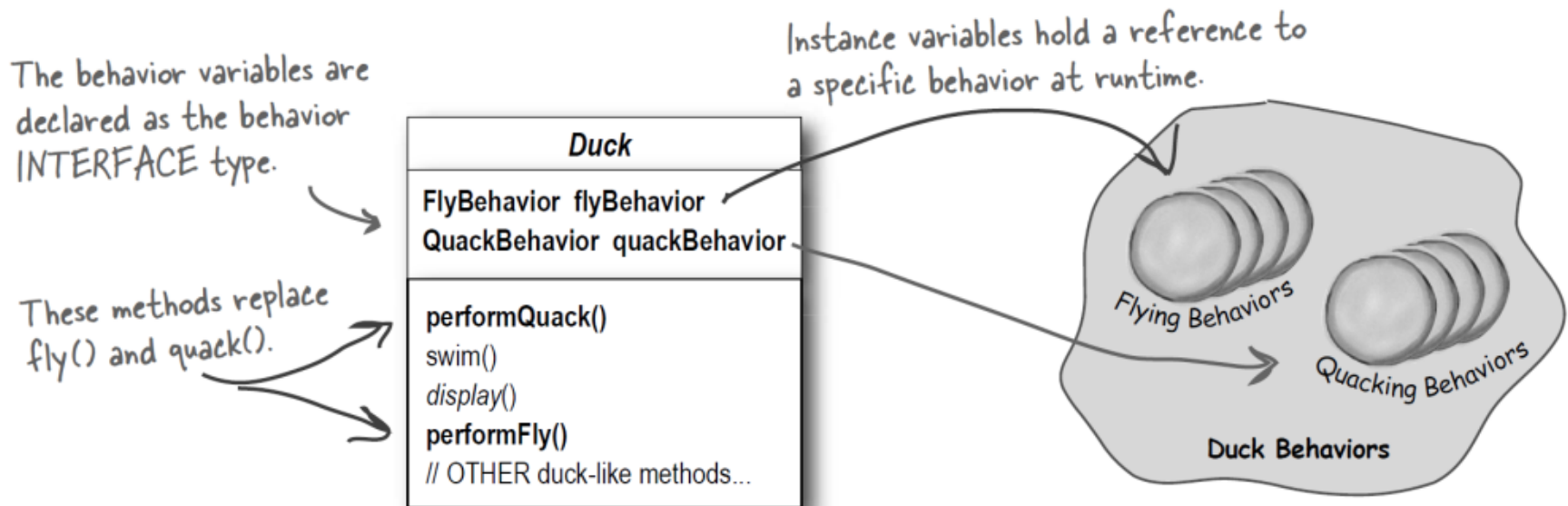
## Q and A

- Q: It feels a little weird to have a class that's just a behavior. Aren't classes supposed to represent things? Aren't classes supposed to have both state AND behavior?
- A: In an OO system, yes, classes represent things that generally have both state (instance variables) and methods. And in this case, the *thing* happens to be a behavior. But even a behavior can still have state and methods; a flying behavior might have instance variables representing the attributes for the flying (wing beats per minute, max altitude and speed, etc.) behavior.

# 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).

# 1. First we'll add two instance variables to the Duck class



## 2. Now we implement performQuack()

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

Each Duck has a reference to something that implements the QuackBehavior interface.

Rather than handling the quack behavior itself, the Duck object delegates that behavior to the object referenced by quackBehavior.



### 3. How the flyBehavior and quackBehavior instance variables are set

```
public class MallardDuck extends Duck {  
  
    public MallardDuck() {  
        quackBehavior = new Quack();  
        flyBehavior = new FlyWithWings();  
    }  
  
}
```

Remember, MallardDuck inherits the quackBehavior and flyBehavior instance variables from class Duck.

A MallardDuck uses the Quack class to handle its quack, so when performQuack is called, the responsibility for the quack is delegated to the Quack object and we get a real quack.

And it uses FlyWithWings as its FlyBehavior type.

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

# Testing the code

```
public abstract class Duck {
```

```
    FlyBehavior flyBehavior;  
    QuackBehavior quackBehavior;  
    public Duck() {  
    }
```

← Declare two reference variables  
for the behavior interface types.  
All duck subclasses (in the same  
package) inherit these.

```
    public abstract void display();
```

```
    public void performFly() {  
        flyBehavior.fly();  
    }
```

← Delegate to the behavior class.

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

```
    public void swim() {  
        System.out.println("All ducks float, even decoys!");  
    }
```

```
}
```

# Testing the code

```
public interface FlyBehavior {  
    public void fly();  
}
```

The interface that all flying behavior classes implement.

---

```
public class FlyWithWings implements FlyBehavior {  
    public void fly() {  
        System.out.println("I'm flying!!");  
    }  
}
```

Flying behavior implementation for ducks that DO fly...

---

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

Flying behavior implementation for ducks that do NOT fly (like rubber ducks and decoy ducks).

# Testing the code

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

---

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

---

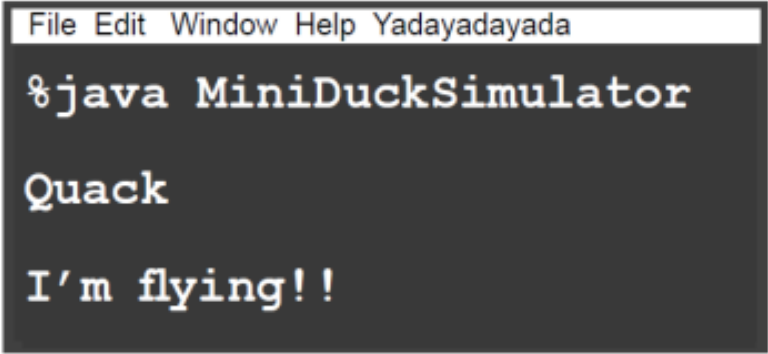
```
public class MuteQuack implements QuackBehavior {  
    public void quack() {  
        System.out.println("<< Silence >>");  
    }  
}
```

---

```
public class Squeak implements QuackBehavior {  
    public void quack() {  
        System.out.println("Squeak");  
    }  
}
```

# Testing the code

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



A terminal window with a menu bar containing 'File', 'Edit', 'Window', 'Help', and 'Yadayadayada'. The terminal displays the command to run the Java program, followed by the output of the program.

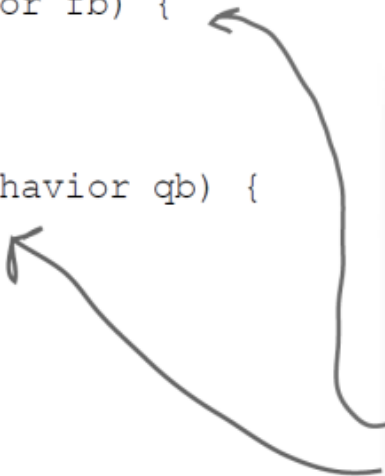
```
File Edit Window Help Yadayadayada  
%java MiniDuckSimulator  
Quack  
I'm flying!!
```

# Setting behavior dynamically

## Add two new methods to the Duck class:

```
public void setFlyBehavior(FlyBehavior fb) {  
    flyBehavior = fb;  
}  
  
public void setQuackBehavior(QuackBehavior qb) {  
    quackBehavior = qb;  
}
```

<i>Duck</i>
FlyBehavior flyBehavior; QuackBehavior quackBehavior;
swim() <i>display()</i> performQuack() performFly() setFlyBehavior() setQuackBehavior() // OTHER duck-like methods...



# Make a new Duck type

```
public class ModelDuck extends Duck {  
    public ModelDuck() {  
        flyBehavior = new FlyNoWay();  
        quackBehavior = new Quack();  
    }  
  
    public void display() {  
        System.out.println("I'm a model duck");  
    }  
}
```

← Our model duck begins life grounded...  
without a way to fly.

# Make a new FlyBehavior type

That's okay, we're creating a rocket powered flying behavior.



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



# Make the ModelDuck rocket-enabled

```
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();  
    }  
}
```

If it worked, the model duck dynamically changed its flying behavior! You can't do THAT if the implementation lives inside the duck class.

**Run it!**

```
File Edit Window Help Yabadabadoo  
%java MiniDuckSimulator  
Quack  
I'm flying!!
```

before



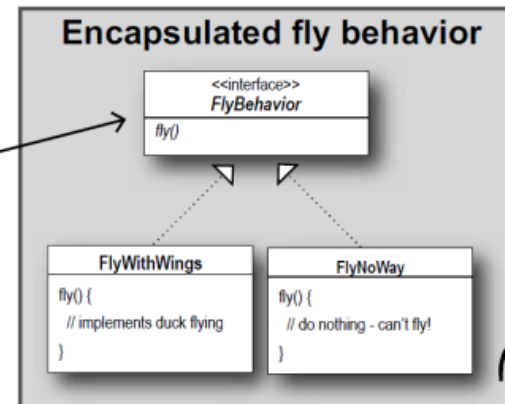
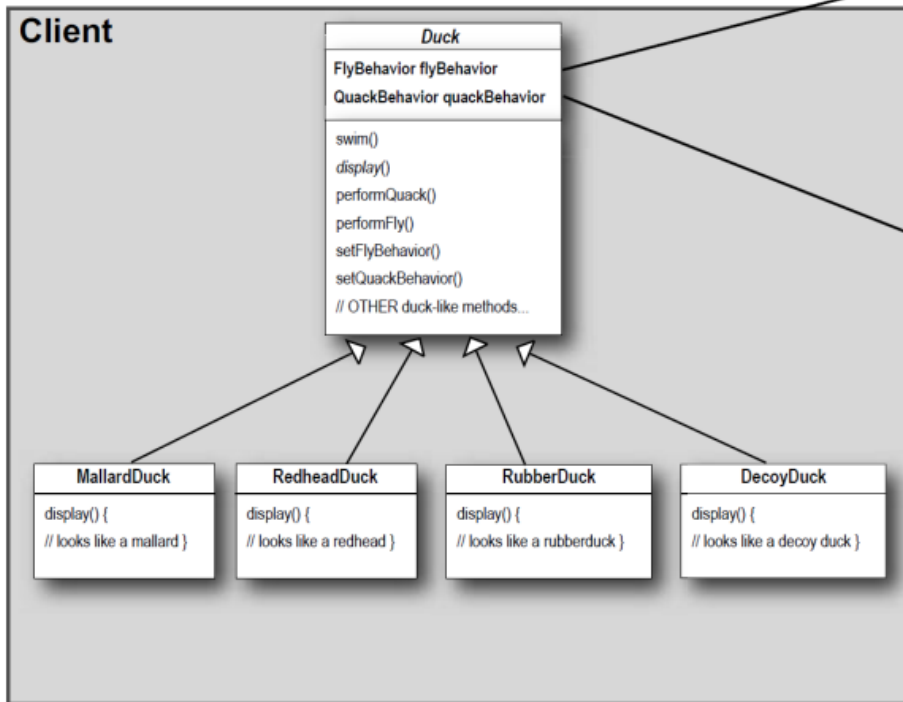
The first call to `performFly()` delegates to the `flyBehavior` object set in the `ModelDuck`'s constructor, which is a `FlyNoWay` instance.

This invokes the model's inherited behavior setter method, and...voila! The model suddenly has rocket-powered flying capability!

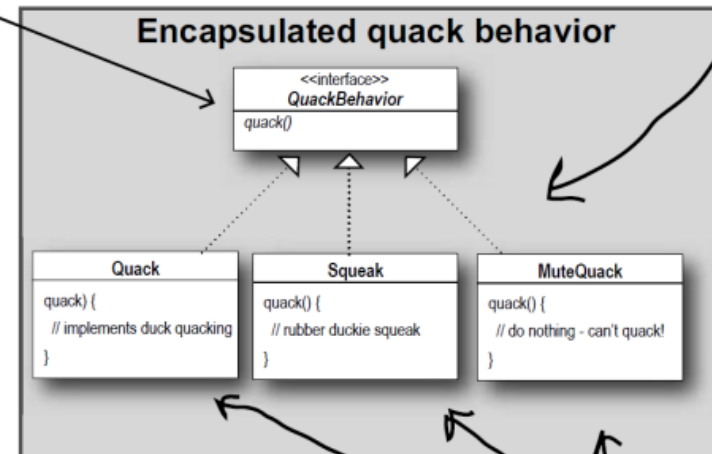


after

Client makes use of an encapsulated family of algorithms for both flying and quacking.



Think of each set of behaviors as a family of algorithms.



These behaviors "algorithms" are interchangeable.

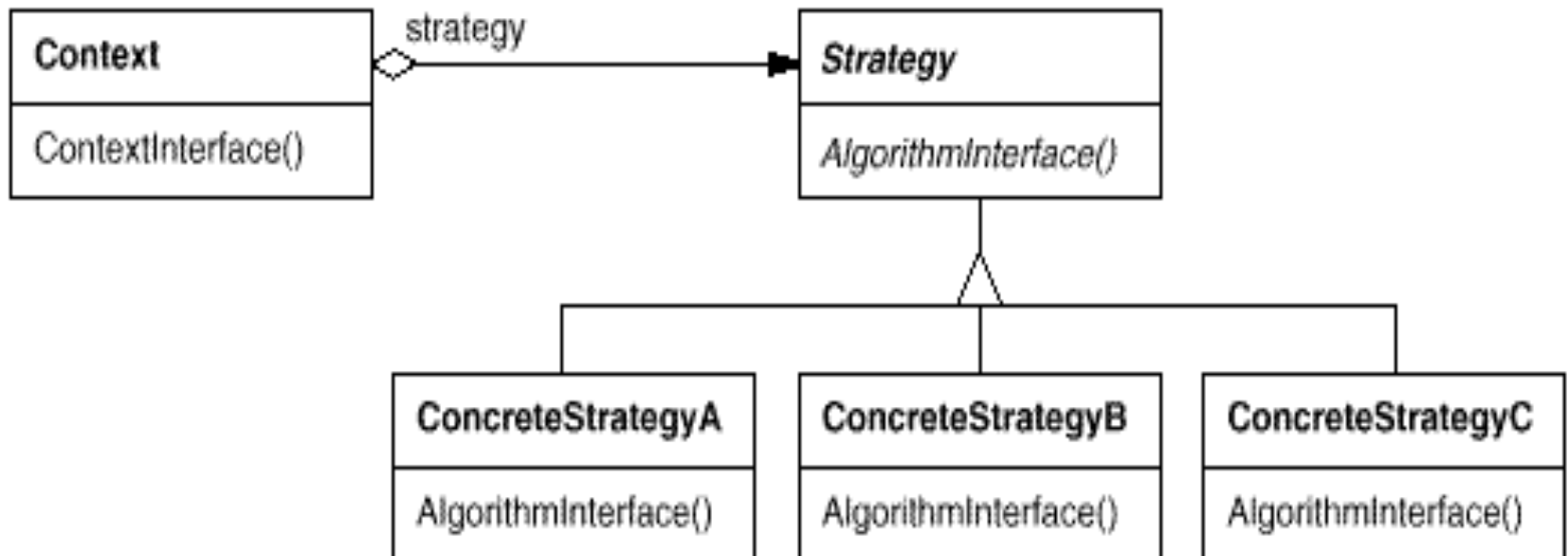
# HAS-A can be better than IS-A

*Favor composition over inheritance.*

1. Composition gives you a lot more flexibility.
2. 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.

# The first design pattern-- STRATEGY

- 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.



# Pattern

- Name:
  - Strategy
- Intent:
  - Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.
- Also known as:
  - Policy

# Motivation – breaking a stream of text into lines

- Many algorithms exist for breaking a stream of text into lines. Hard-wiring all such algorithms into the classes isn't desirable .

```
public class Context
{
    .....
    public void algorithm(String type)
    {
        .....
        if(type == "strategyA")
        {...}
        else if(type == "strategyB")
        {...}
        else if(type == "strategyC")
        {...}
        .....
    }
    .....
}
```

# Applicability

- Use the Strategy pattern when
  - many related classes differ only in their behavior. Strategies provide a way to configure a class with one of many behaviors.
  - you need different variants of an algorithm. For example, you might define algorithms reflecting different space/time trade-offs. Strategies can be used when these variants are implemented as a class hierarchy of algorithms.

# Applicability

- Use the Strategy pattern when
  - an algorithm uses data that clients shouldn't know about. Use the Strategy pattern to avoid exposing complex, algorithm-specific data structures.
  - a class defines many behaviors, and these appear as multiple conditional statements in its operations. Instead of many conditionals, move related conditional branches into their own Strategy class.



# Consequences

- *Families of related algorithms.* Hierarchies of Strategy classes define a family of algorithms or behaviors for contexts to reuse. Inheritance can help factor out common functionality of the algorithms.
- *An alternative to subclassing.*
- *Strategies eliminate conditional statements.*

# Consequences

- *A choice of implementations.* Strategies can provide different implementations of the *same* behavior. The client can choose among strategies with different time and space trade-offs.
- *Clients must be aware of different Strategies.* The pattern has a potential drawback in that a client must understand how Strategies differ before it can select the appropriate one. Clients might be exposed to implementation issues.
- *Communication overhead between Strategy and Context.*
- *Increased number of objects.*

# Shared vocabulary

- ALICE :
- I need a Cream cheese with jelly on white bread, a chocolate soda with vanilla ice cream, a grilled cheese sandwich with bacon, a tuna fish salad on toast, a banana split with ice cream & sliced bananas and a coffee with a cream and two sugars, ... oh, and put a hamburger on the grill!

The same order

- FLO :
- Give me a C.J. White, a black & white, a Jack Benny, a radio, a house boat, a coffee regular and burn one!

## Shared vocabulary cont' d

- Design Patterns give you a shared vocabulary with other developers.
- It also elevates your thinking about architectures by letting you ***think at the pattern level***, not the nitty gritty *object* level.

# The power of a shared pattern vocabulary

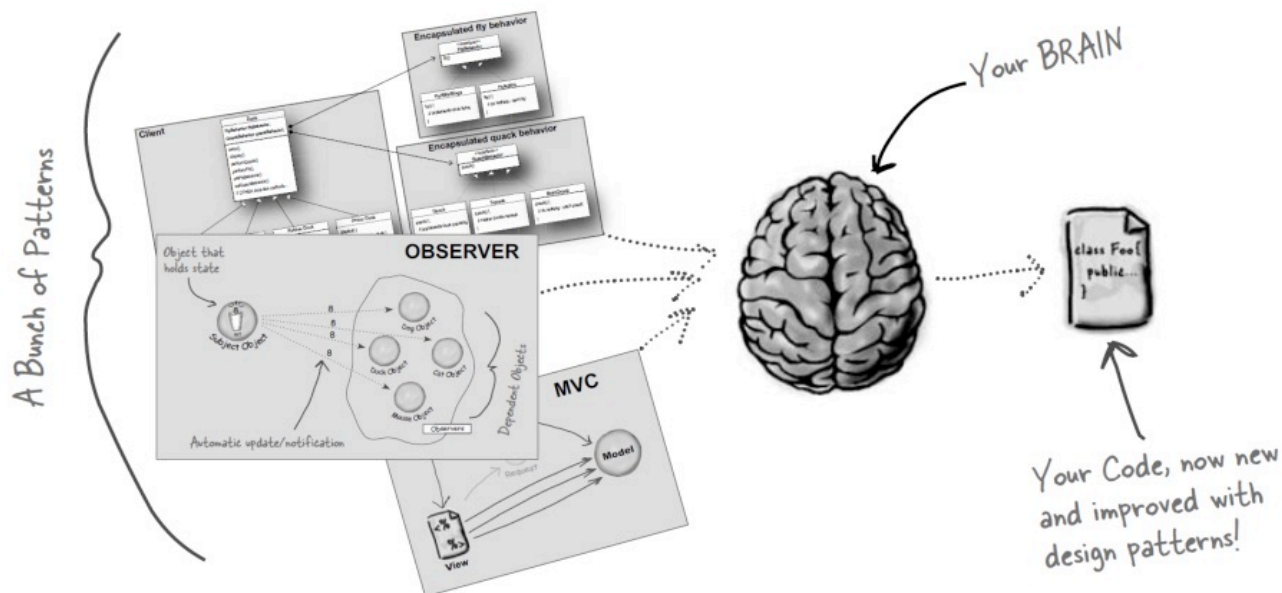
- Shared pattern vocabularies are POWERFUL.
- When you communicate with another developer or your team using patterns, you are communicating not just a pattern name but a whole set of qualities, characteristics and constraints that the pattern represents.
- Patterns allow you to say more with less.
- When you use a pattern in a description, other developers quickly know precisely the design you have in mind.

## The power of a shared pattern vocabulary – cont' d

- Talking at the pattern level allows you to stay “in the design” longer.
  - Do not lost in the details.
- Shared vocabularies can turbo charge your development team.
- Shared vocabularies encourage more junior developers to get up to speed.

How do we use design patterns?

- Libraries and frameworks.
- DP help us structure our own applications to be more maintainable and flexible.
- DP first go into *your* BRAIN.



## Q&A

**Q:** If design patterns are so great, why can't someone build a library of them so I don't have to?

**A:** Design patterns are higher level than libraries. Design patterns tell us how to structure classes and objects to solve certain problems and it is our job to adapt those designs to fit our particular application.

**Q:** Aren't libraries and frameworks also design patterns?

**A:** Frameworks and libraries are not design patterns; they provide specific implementations that we link into our code. Sometimes, however, libraries and frameworks make use of design patterns in their implementations. That's great, because once you understand design patterns, you'll more quickly understand APIs that are structured around design patterns.



# Patterns are nothing more than using OO design principles?

- Knowing concepts like abstraction, inheritance, and polymorphism do not make you a good object oriented designer. A design guru thinks about how to create flexible designs that are maintainable and that can cope with change.

# Tools for your design toolbox

- OO Basics

- Abstraction
- Encapsulation
- Polymorphism
- Inheritance

- OO Principles

- Encapsulate what varies
- Favor composition over inheritance
- Program to interfaces, not implementation

- OO Patterns

- Strategy

# Reviews 1

- Knowing the OO basics does not make you a good OO designer.
- Good OO designs are reusable, extensible and maintainable.
- Patterns show you how to build systems with good OO design qualities.
- Patterns are proven object oriented experience.

## Reviews 2

- Patterns don't give you code, they give you general solutions to design problems. You apply them to your specific application.
- Patterns aren't invented, they are discovered.
- Most patterns and principles address issues of change in software.
- Most patterns allow some part of a system to vary independently of all other parts.
- We often try to take what varies in a system and encapsulate it.
- Patterns provide a shared language that can maximize the value of your communication with other developer.