Design Patterns

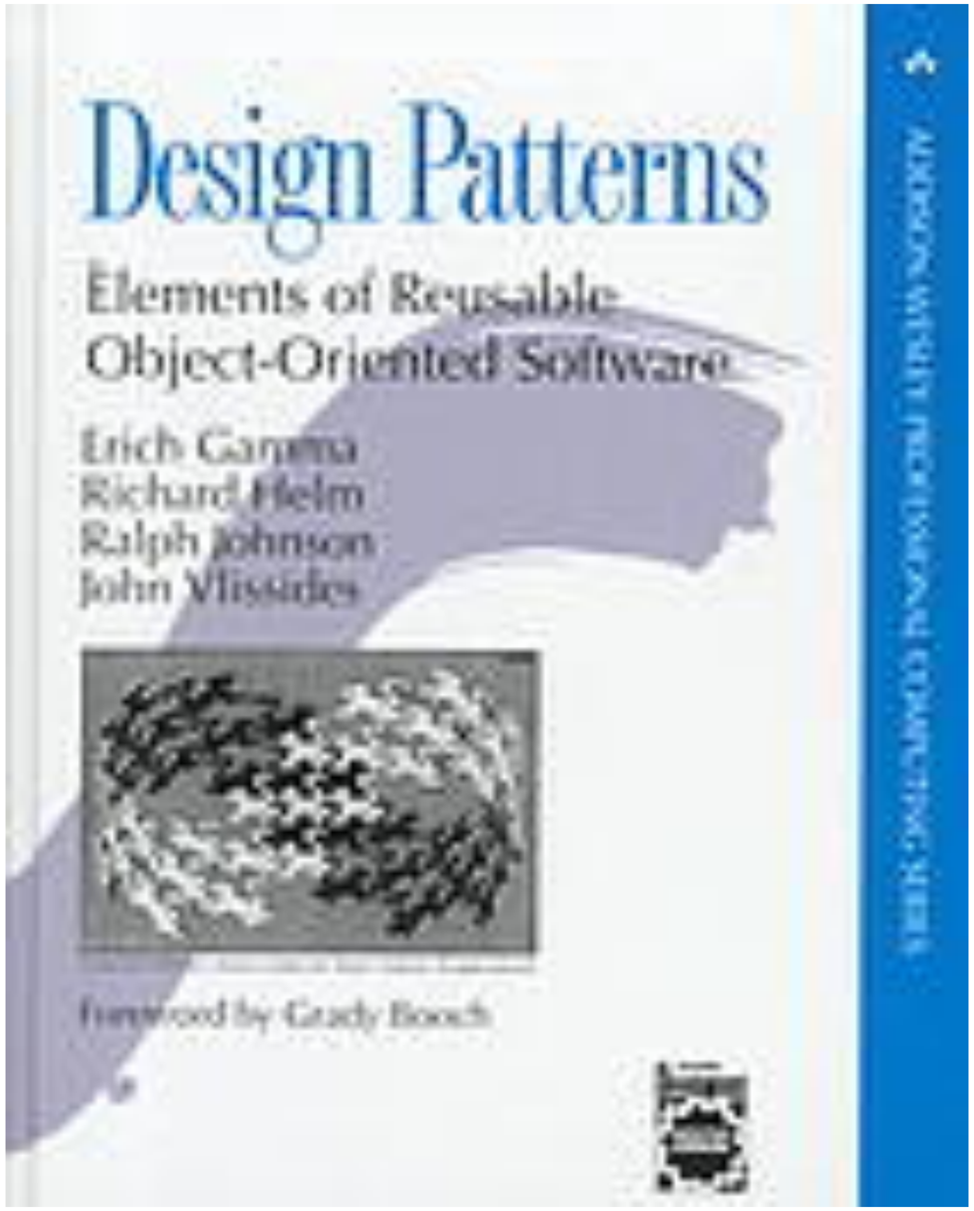
“OO Programming++”

Patterns

- What is a pattern
    - A named, reusable template for solving a common issue in software design.
    - A higher order abstractions for program organization —Peter Norvig
    - Language independent
- Why patterns
    - “Someone has already solved your problem. Instead of *code* reuse, with patterns you get *experience* reuse.” — Head First Design Patterns
    - Allow developers to communicate about design

Gang of Four (GoF) Book

- 23 Patterns
- More focused on C++
- Hard to read
- Not updated since 1994


Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

Head First…

O'REILLY
10th Anniversary
Updated for Java 8
Head First
Design Patterns
A Brain-Friendly Guide
Avoid those embarrassing coupling mistakes
Learn why everything your friends know about Factory pattern is probably wrong
Discover the secrets of the Patterns Guru
Load the patterns that matter straight into your brain
Find out how Starbuzz Coffee doubled their stock price with the Decorator pattern
See why Jim's love life improved when he cut down his inheritance
Eric Freeman & Elisabeth Robson
with Kathy Sierra & Bert Bates

Image source: Amazon

Categories of Patterns

- Creational
    - Create objects on your behalf
    - Factory
- Structural
    - How objects are composed and work together
    - Adapter
- Behavioral
    - Object-to-object communication
    - Observer
- Concurrency
    - Support concurrent and distributed programming via message passing
    - Join

Criticisms of Patterns

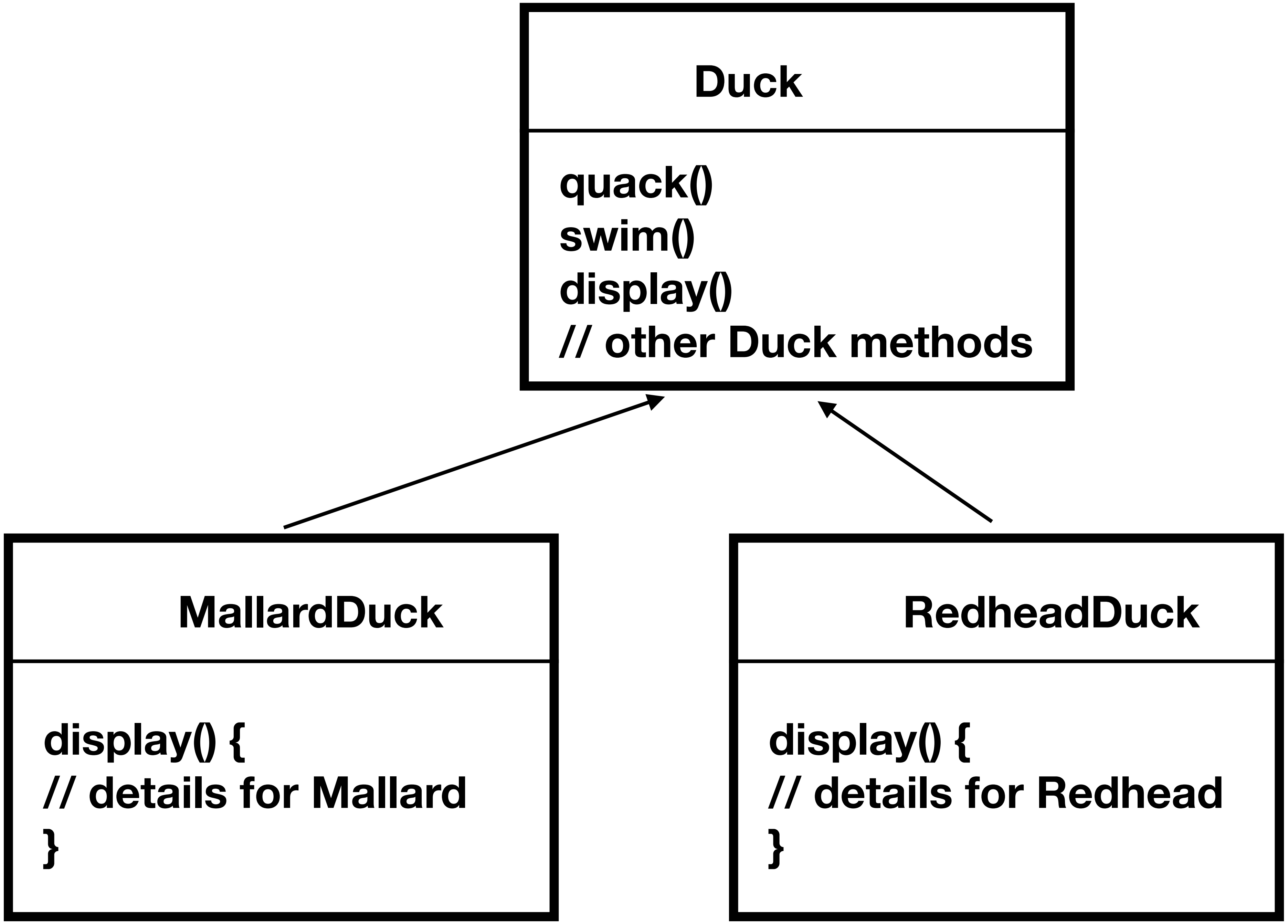
- Separate from the GoF book
  - Patterns make up for features missing in programming language X.
  - Patterns may signal weak program abstractions.

Case Study: SimUDuck

- A ficticious, highly successful duck-pond simulation game
- All ducks inherit from Duck superclass

Adapted from Head First Design Patterns

Duck UML



Duck

quack()
swim()
display()
// other Duck methods

MallardDuck

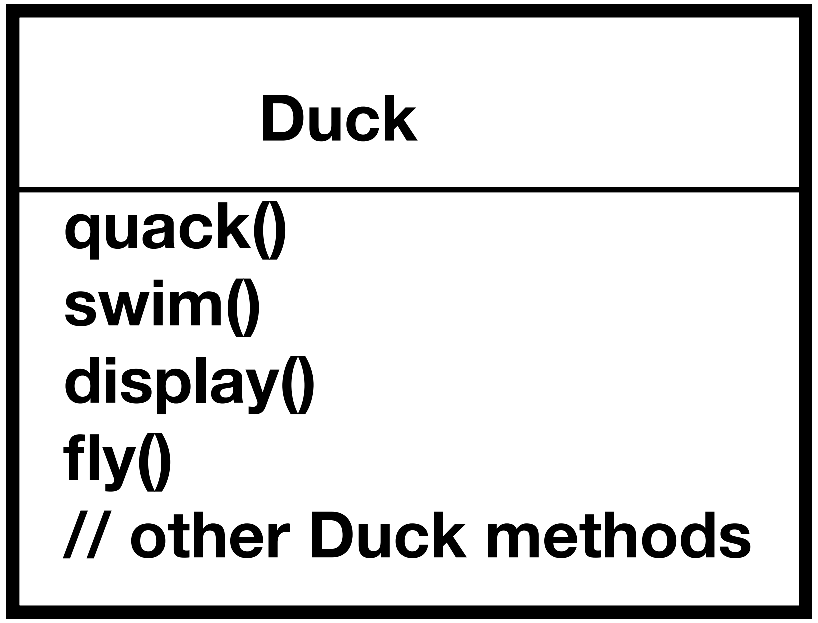
display() {
// details for Mallard
}

RedheadDuck

display() {
// details for Redhead
}

Innovation?

- Ducks can fly
- All subclasses inherit fly()

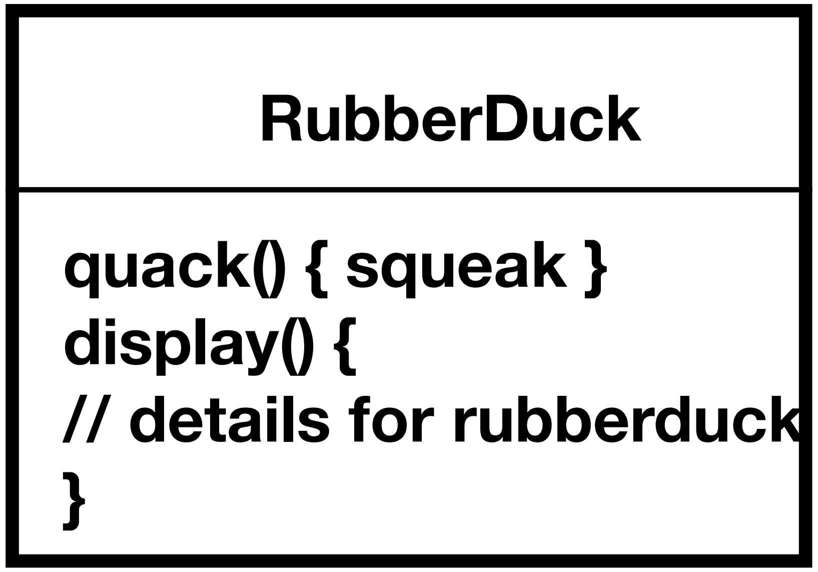


Duck

quack()
swim()
display()
fly()
// other Duck methods

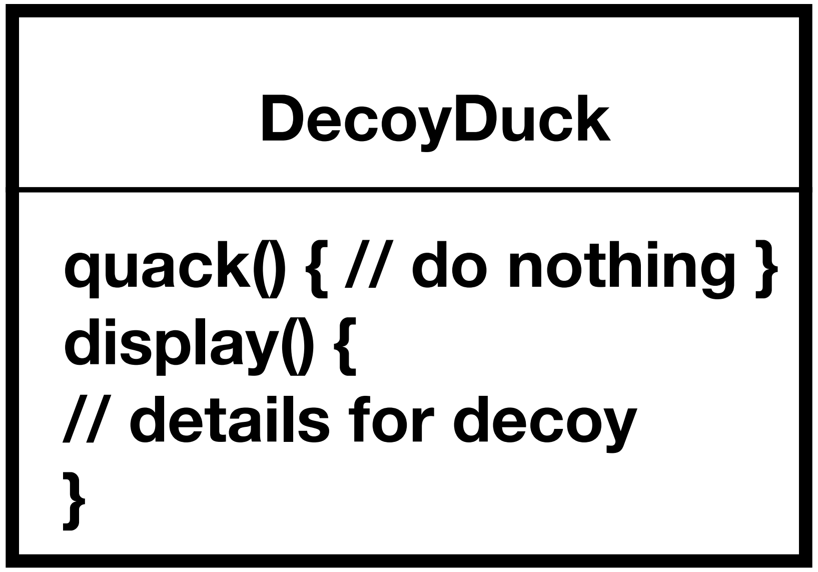
Oops

- Not all ducks fly
- Not all ducks quack the same way



RubberDuck

```
quack() { squeak }
display() {
// details for rubberduck
}
```



DecoyDuck

```
quack() { // do nothing }
display() {
// details for decoy
}
```

Inheritance: Disadvantages

- Code in the super class is duplicated across subclasses
- Changes can unintentionally affect other ducks
- fly() and quack() may need to be edited for every new Duck subclass

Flyable Interface?

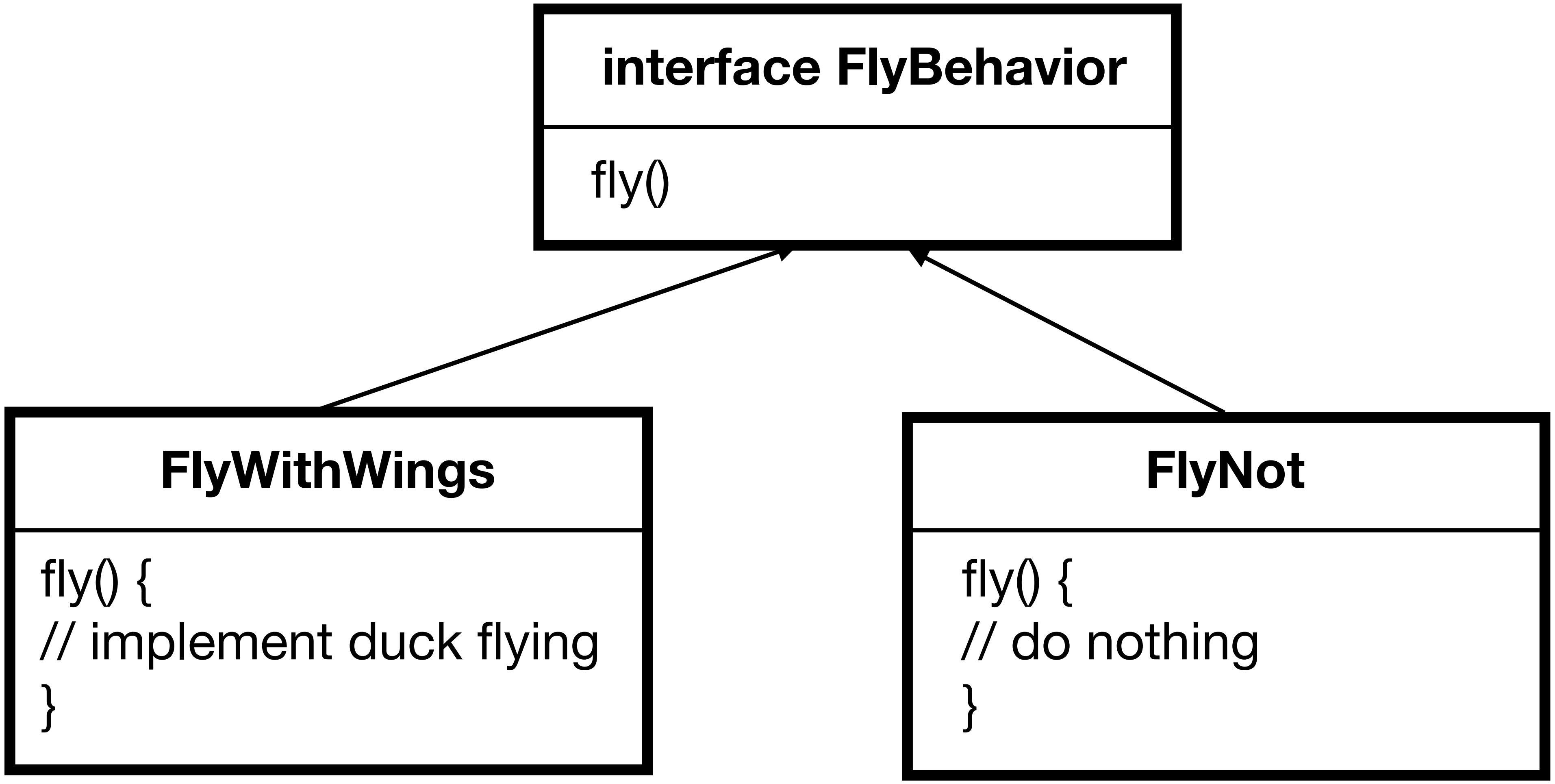
- Flyable interface with fly() method
- Only flying ducks would implement this interface
    - Non-flying ducks not impacted

- Leads to duplicate code in all subclasses that fly

Design Principle

- Separate aspects of an application that change from what stays the same.
  - Remove fly() and quack() from Duck
  - Create sets of classes for each behavior
- Reduce unintended consequences

FlyBehavior



interface FlyBehavior

fly()

FlyWithWings

```
fly() {
// implement duck flying
}
```

FlyNot

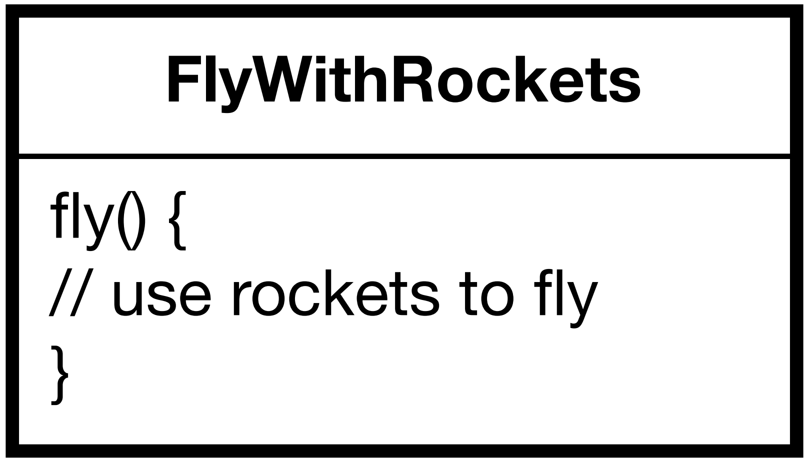
```
fly() {
// do nothing
}
```

A set of classes for the fly behavior.

Quiz

- How would we create a behavior for flying with rockets?
- Draw a UML-like diagram.
- Implement the class.

Answer

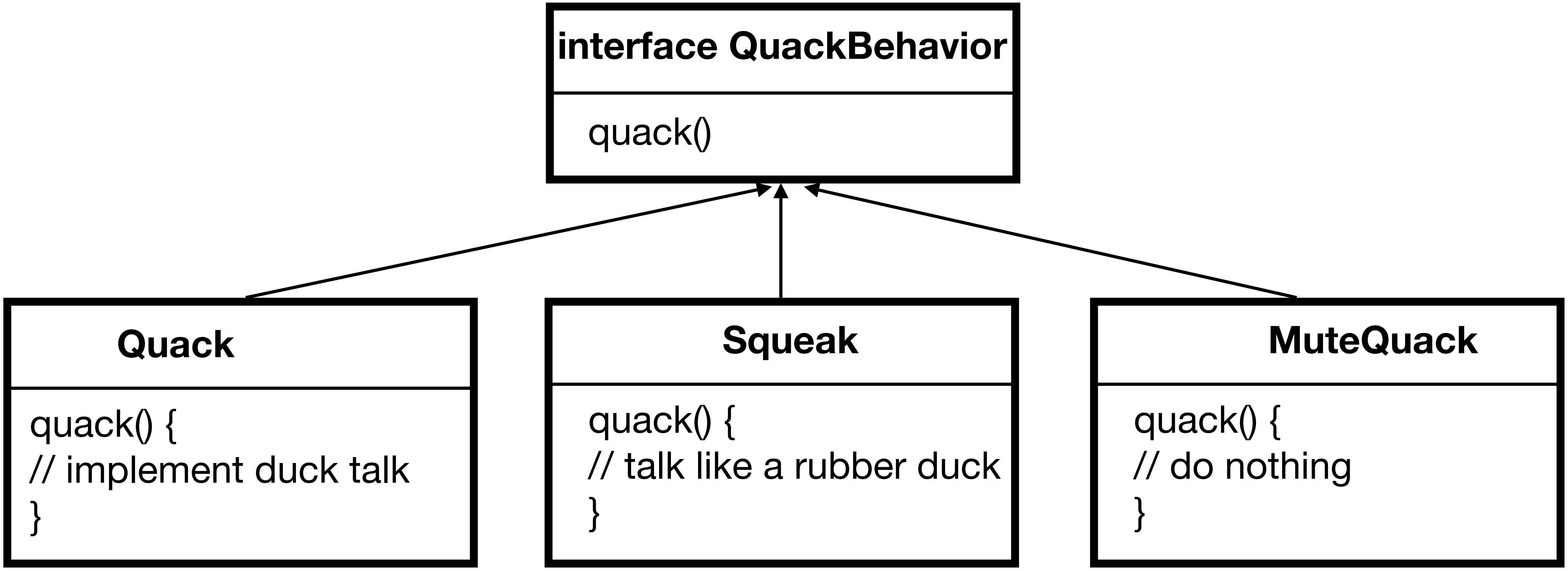


FlyWithRockets

fly() {
// use rockets to fly
}

```
public class FlyWithRockets implements FlyBehavior {
     public void fly() {
          System.out.println(“Rocket Power!”);
     }
}
```

QuackBehavior



interface QuackBehavior

quack()

Quack

```
quack() {
// implement duck talk
}
```

Squeak

```
quack() {
// talk like a rubber duck
}
```

MuteQuack

```
quack() {
// do nothing
}
```

A set of classes for the quack behavior.

Benefits

- Other objects can re-use these behaviors
- Can add new behaviors without changing any existing behaviors or any Duck subclasses

Design Principle

- Program to an interface, not a concrete implementation.

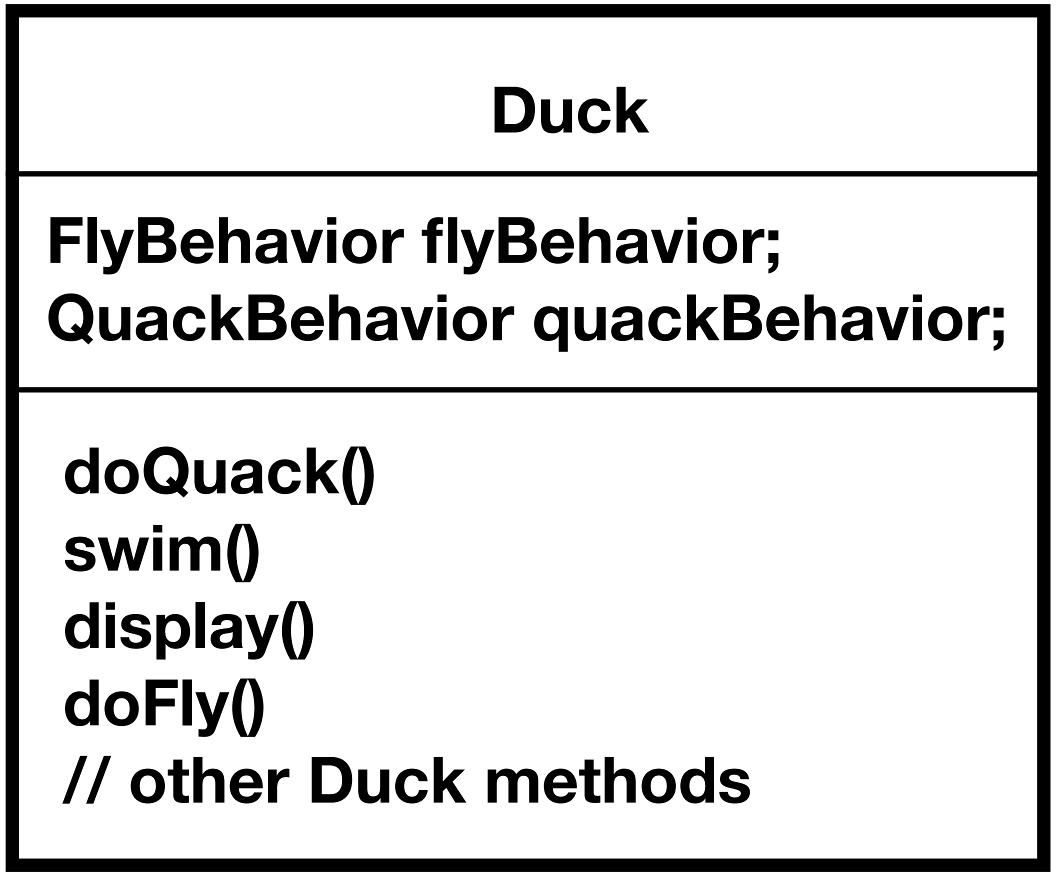
Do this…

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

Not this…

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

Interface Integration



Duck

FlyBehavior flyBehavior;
QuackBehavior quackBehavior;

doQuack()
swim()
display()
doFly()
// other Duck methods

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

    public void doFly() {
        flyBehavior.fly();
    }

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

The Duck object **delegates** flying and quacking behaviors.

Concrete Integration

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

    public void display() {
        System.out.println(“I’m a mallard”);
    }
}
```

Dynamic Behaviors

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

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

    public void display() {
        System.out.println(“I’m a mallard”);
    }
}
```

Pull it together

```
public class DuckTester {
    public static void main(String[] args) {
        Duck duck = new MallardDuck();
        duck.doFly();           // fly with wings
        duck.setFlyBehavior(new FlyWithRockets());
        duck.doFly();           // fly with rockets
    }
}
```

Design Principle

- Favor composition over inheritance
- Inheritance is based on an “is-a” relationship between objects.
- Composition is a “has-a” relationship between objects.
- ex. Duck “has-a” FlyBehavior.

Strategy Pattern

- Define a family of algorithms, encapsulate each algorithm, and make them interchangeable.