Strategy Pattern

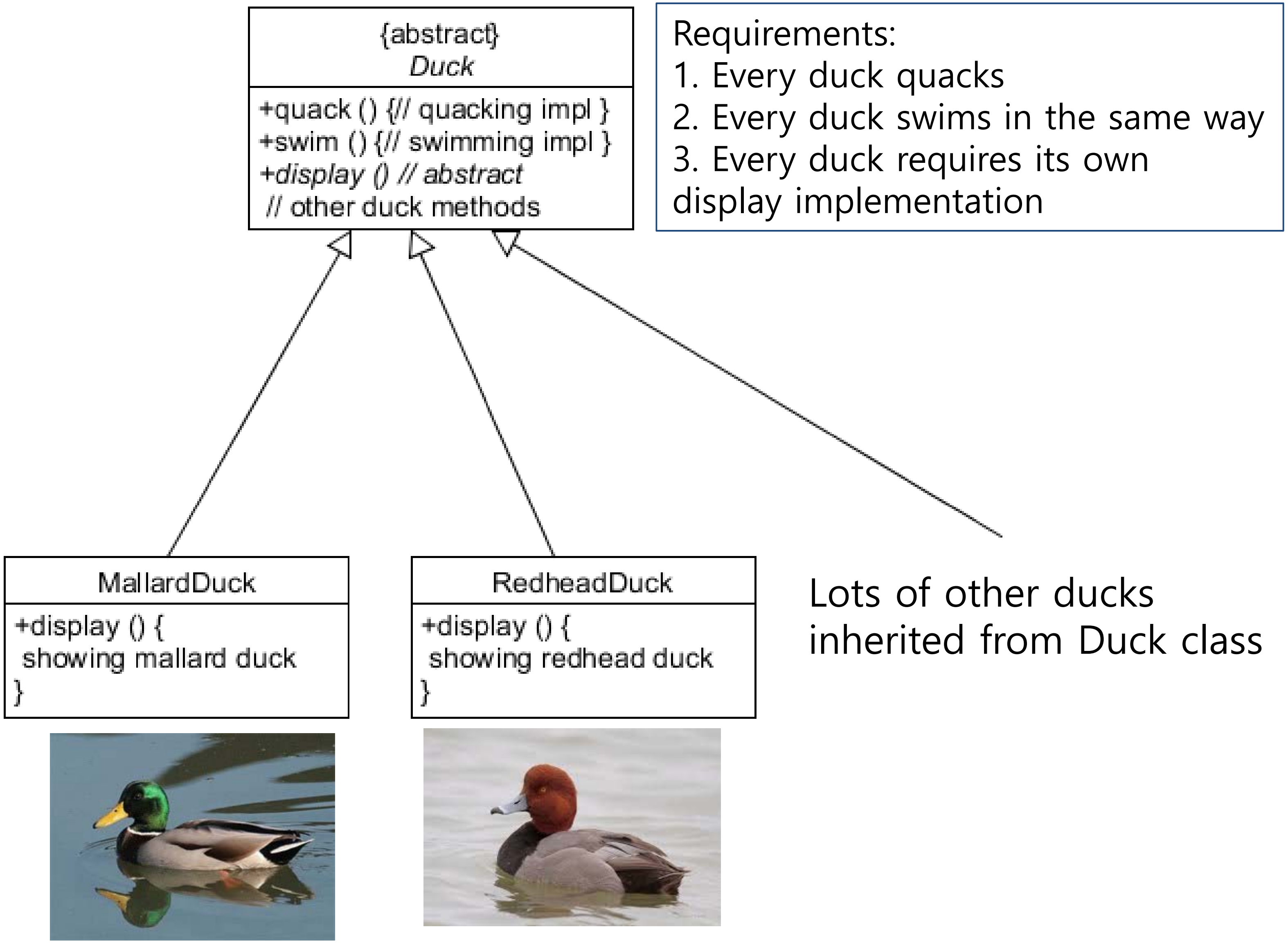
Contents

- Intro. to the problem
- How the strategy pattern is discovered
- Structure of strategy pattern
- Examples using the strategy pattern

Strategy Pattern

- Purpose
    - Defines a set of encapsulated algorithms that can be swapped to carry out a specific behavior.
- Use When
    - The only difference between many related classes is their behavior.
    - Multiple versions or variations of an algorithm are required.
    - The behavior of a class should be defined at runtime.
    - Conditional statements are complex and hard to maintain.

Our Original Design



{abstract}
Duck

+quack () {// quacking impl }
+swim () {// swimming impl }
+*display () // abstract*
// other duck methods

Requirements:
1. Every duck quacks
2. Every duck swims in the same way
3. Every duck requires its own display implementation

MallardDuck

+display () {
showing mallard duck
}

RedheadDuck

+display () {
showing redhead duck
}

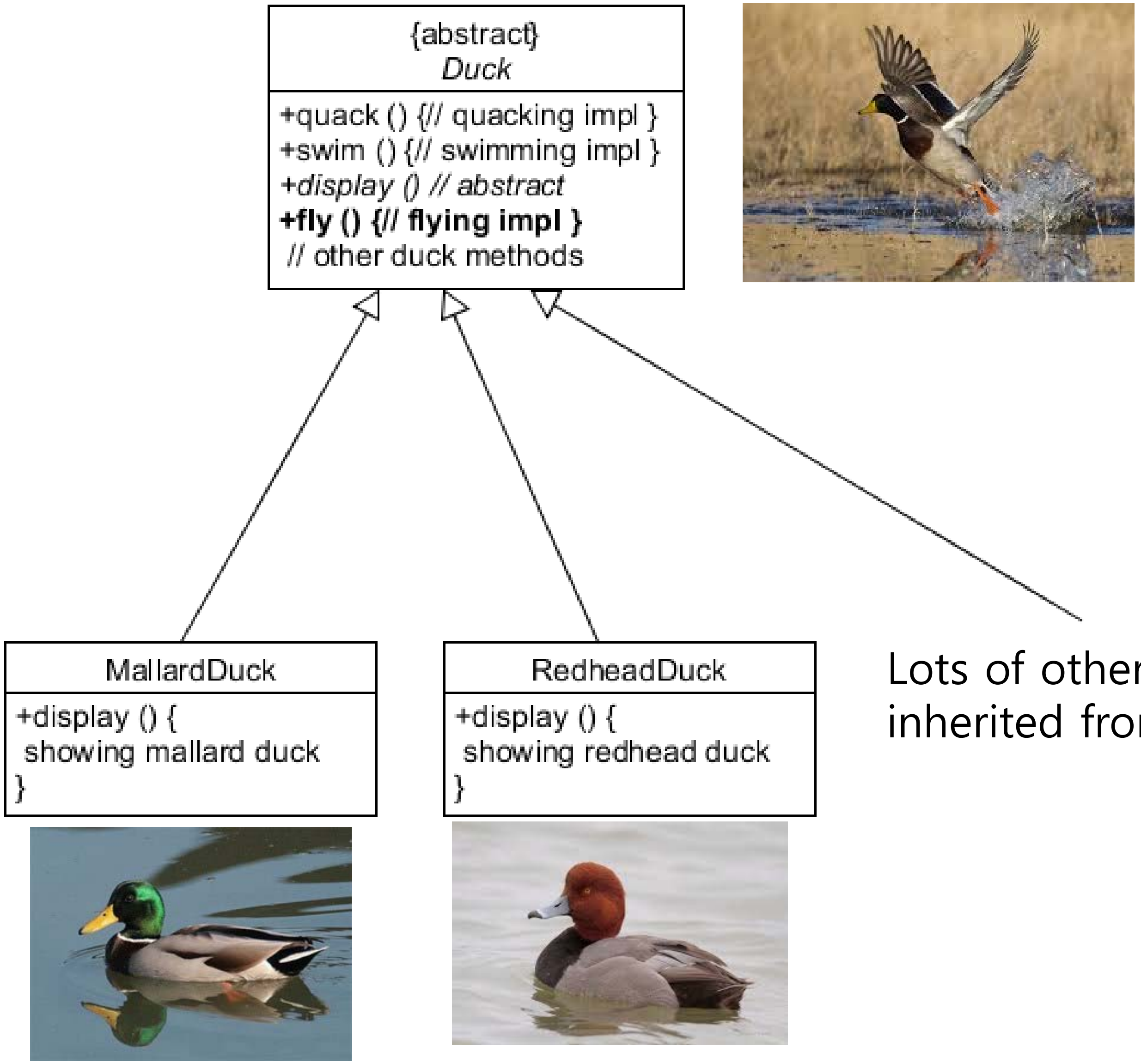
Lots of other ducks inherited from Duck class

Requirement Changes …

- The original design has been okay
- We just got additional requirement
  - We need to add fly behavior to ducks

- Okay, let’s add it to the base class!

Our First Attempt



{abstract}
Duck

+quack () {// quacking impl }
+swim () {// swimming impl }
+display () // abstract
+fly () {// flying impl }
// other duck methods

MallardDuck

+display () {
showing mallard duck
}

RedheadDuck

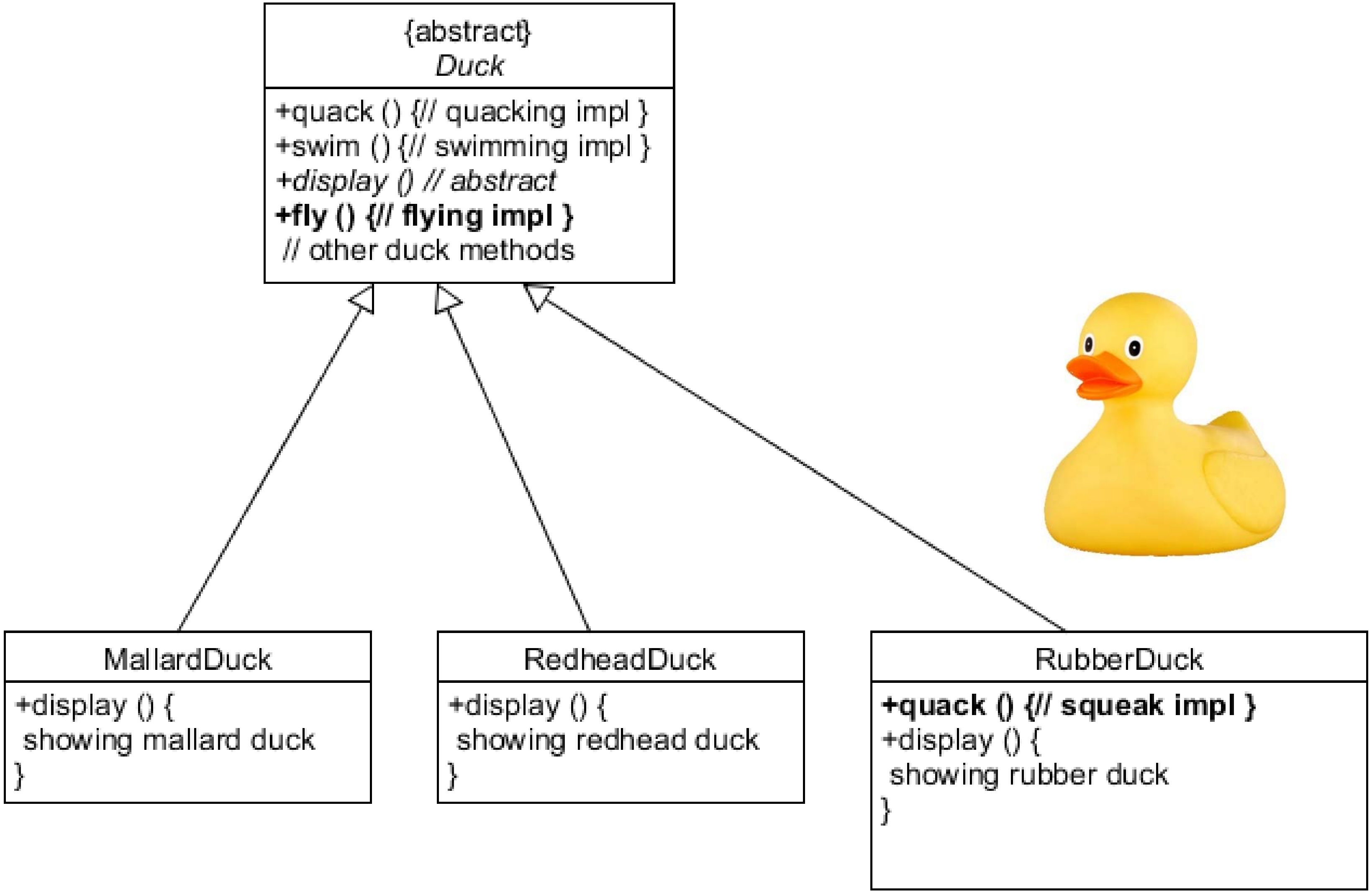
+display () {
showing redhead duck
}

Lots of other ducks
inherited from Duck class

What???

A rubber duck can’t fly, at least in our application!

Looking into Our Design



{abstract}
Duck

+quack () {// quacking impl }
+swim () {// swimming impl }
+display () // abstract
+fly () {// flying impl }
// other duck methods

MallardDuck

+display () {
showing mallard duck
}

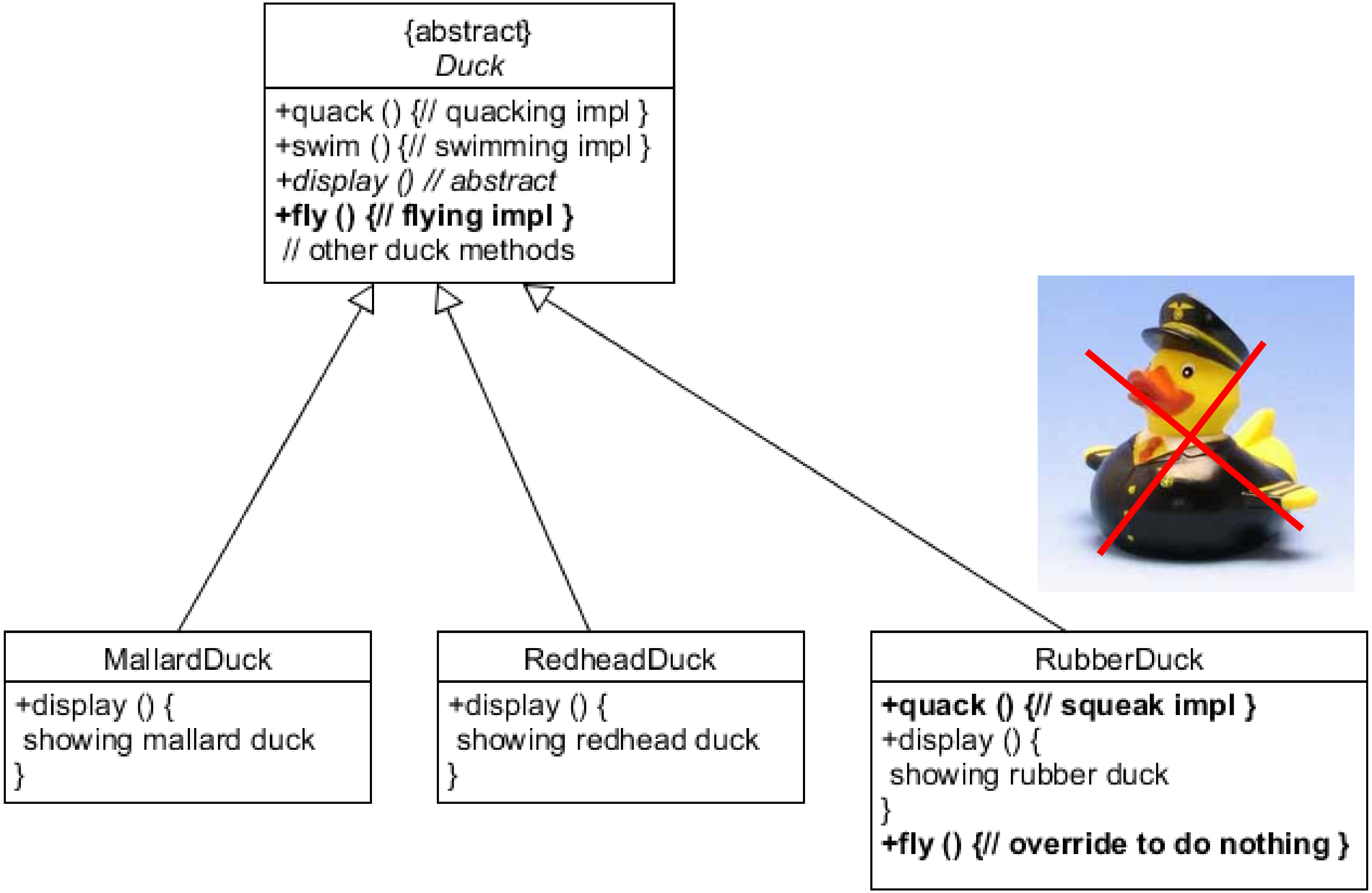
RedheadDuck

+display () {
showing redhead duck
}

RubberDuck

+quack () {// squeak impl }
+display () {
showing rubber duck
}

Overriding RubberDuck Would Solve the Problem



{abstract}
Duck

+quack () {// quacking impl }
+swim () {// swimming impl }
+display () // abstract
+fly () {// flying impl }
// other duck methods

MallardDuck

+display () {
showing mallard duck
}

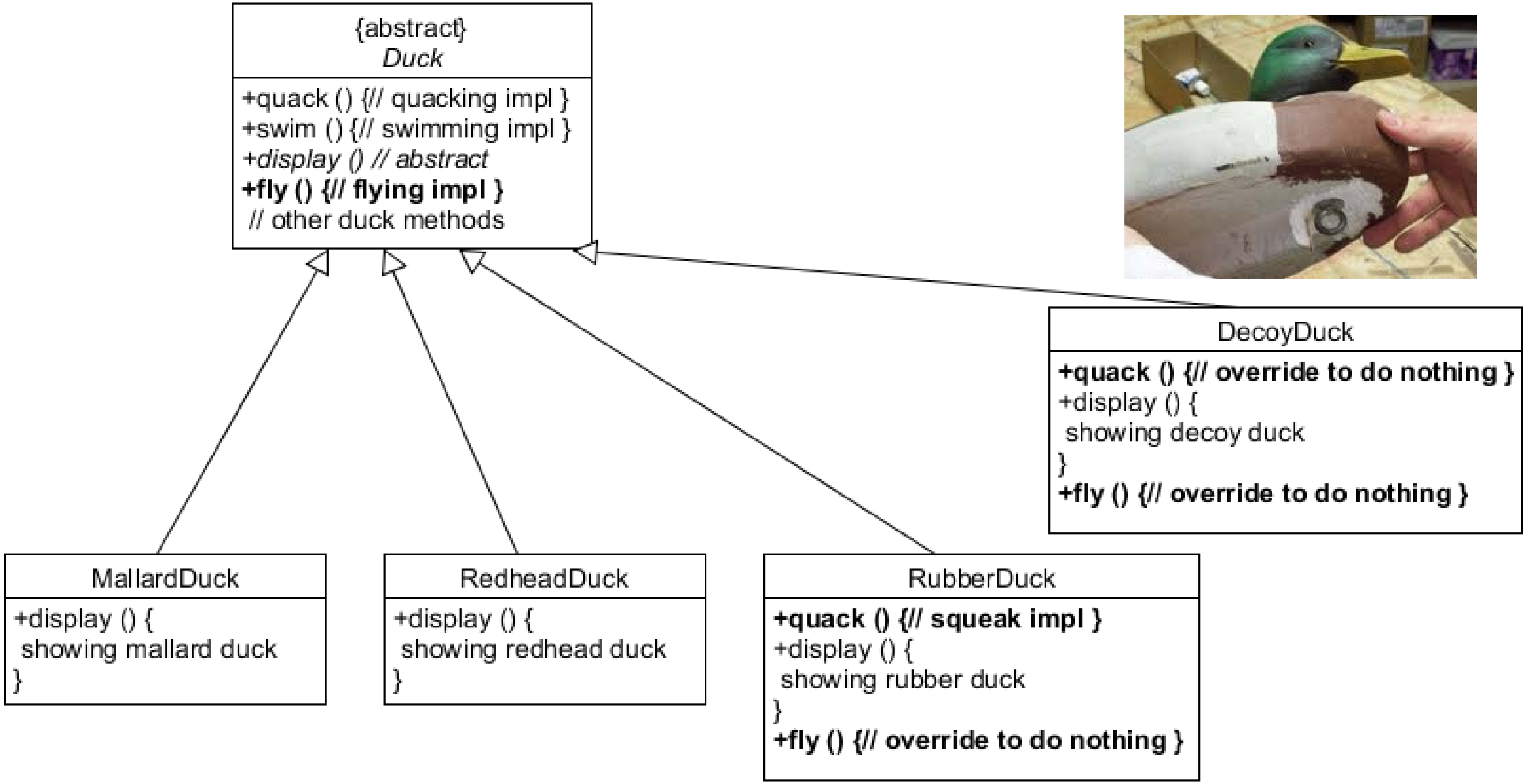
RedheadDuck

+display () {
showing redhead duck
}

RubberDuck

+quack () {// squeak impl }
+display () {
showing rubber duck
}
+fly () {// override to do nothing }

More and More Ducks to Override



{abstract}
Duck

+quack () {// quacking impl }
+swim () {// swimming impl }
+display () // abstract
+fly () {// flying impl }
// other duck methods

MallardDuck

+display () {
showing mallard duck
}

RedheadDuck

+display () {
showing redhead duck
}

RubberDuck

+quack () {// squeak impl }
+display () {
showing rubber duck
}
+fly () {// override to do nothing }

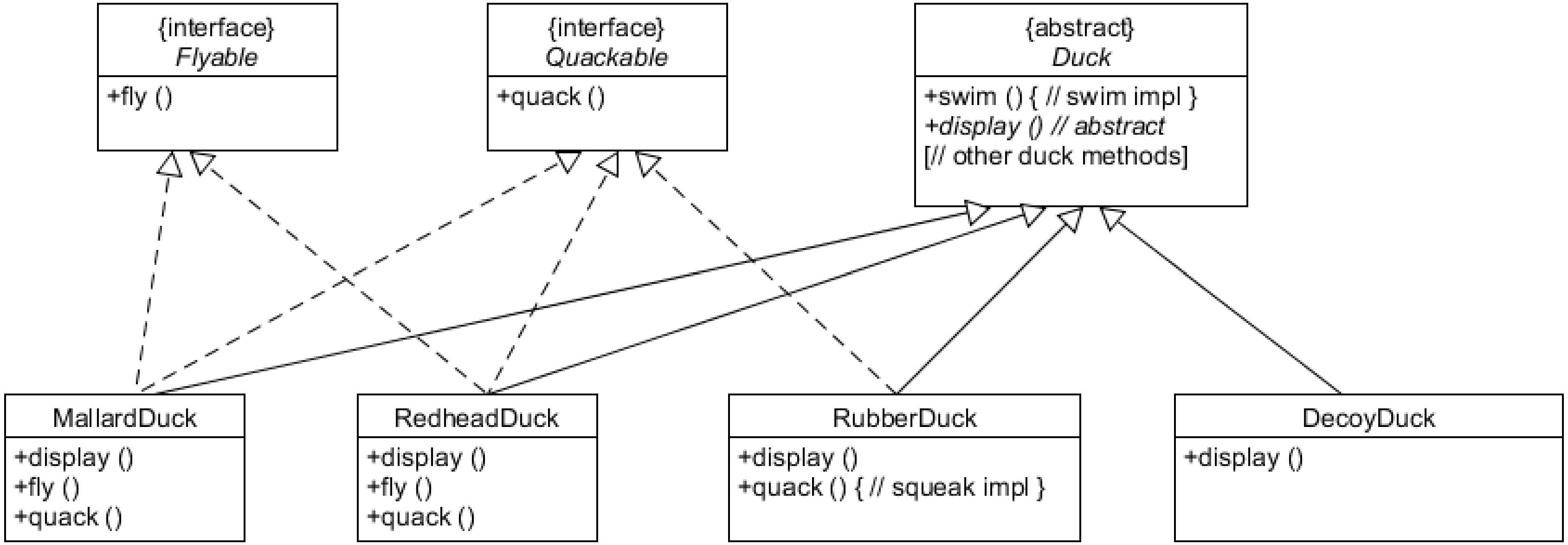
DecoyDuck

+quack () {// override to do nothing }
+display () {
showing decoy duck
}
+fly () {// override to do nothing }

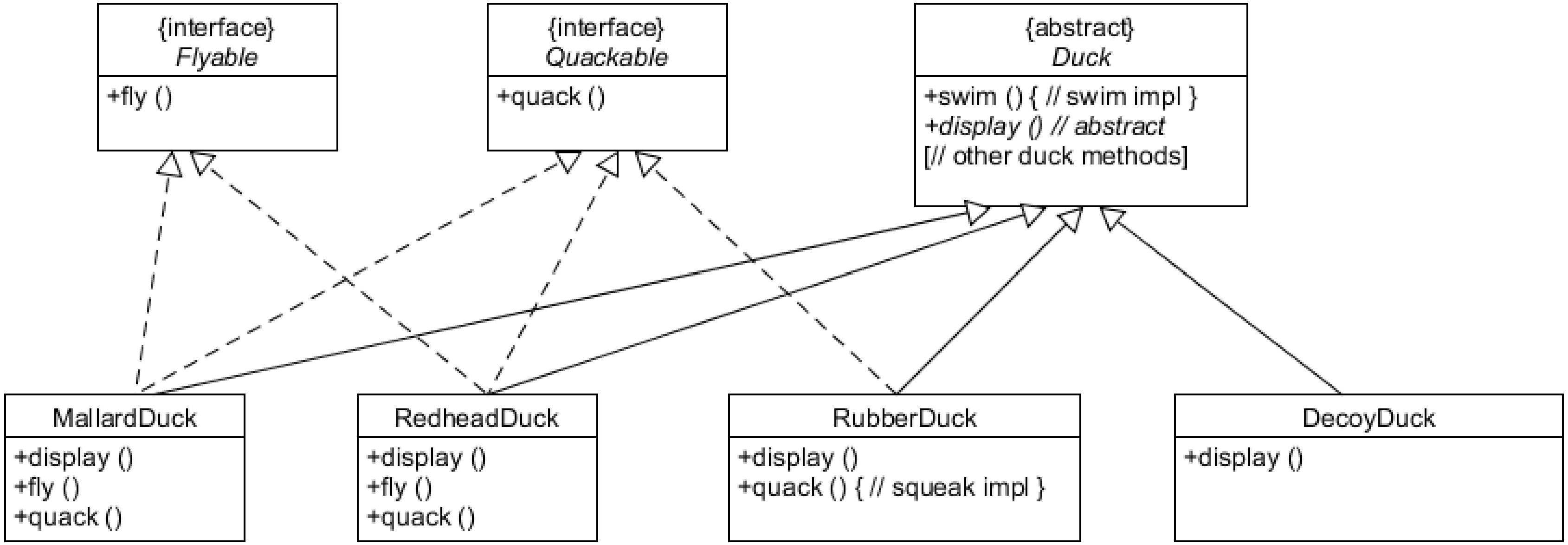
What we have here

- A localized update to the code caused a non-local side effect!
- A great use of inheritance for the purpose of reuse hasn’t turned out so well when time comes to maintenance

Our Second Attempt: How about an Interface?



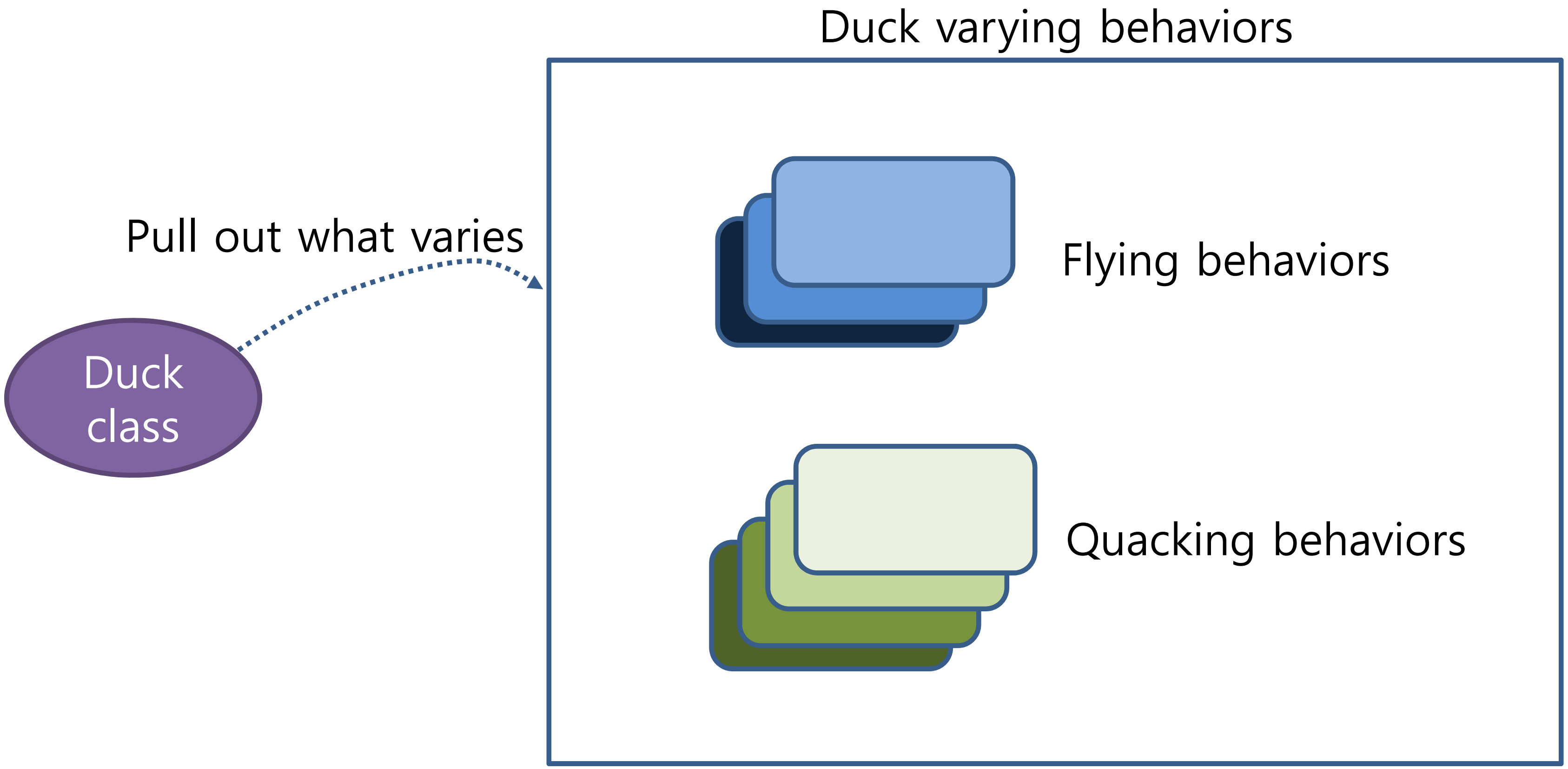
Our Second Attempt: How about an Interface?



- Completely **destroys code reuse**
- Creates a *different* **maintenance nightmare**
- There might be more than one kind of flying behavior even among the ducks that do fly
  - Turkey, Chicken, …

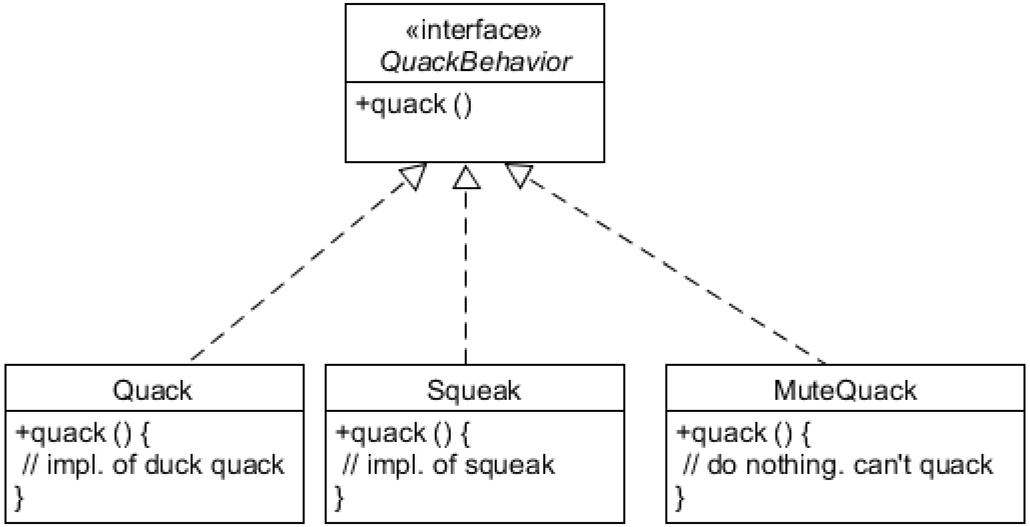
Design Principle: Encapsulate what varies

- Identify the aspects of your application that vary and separate them from what stays the same!



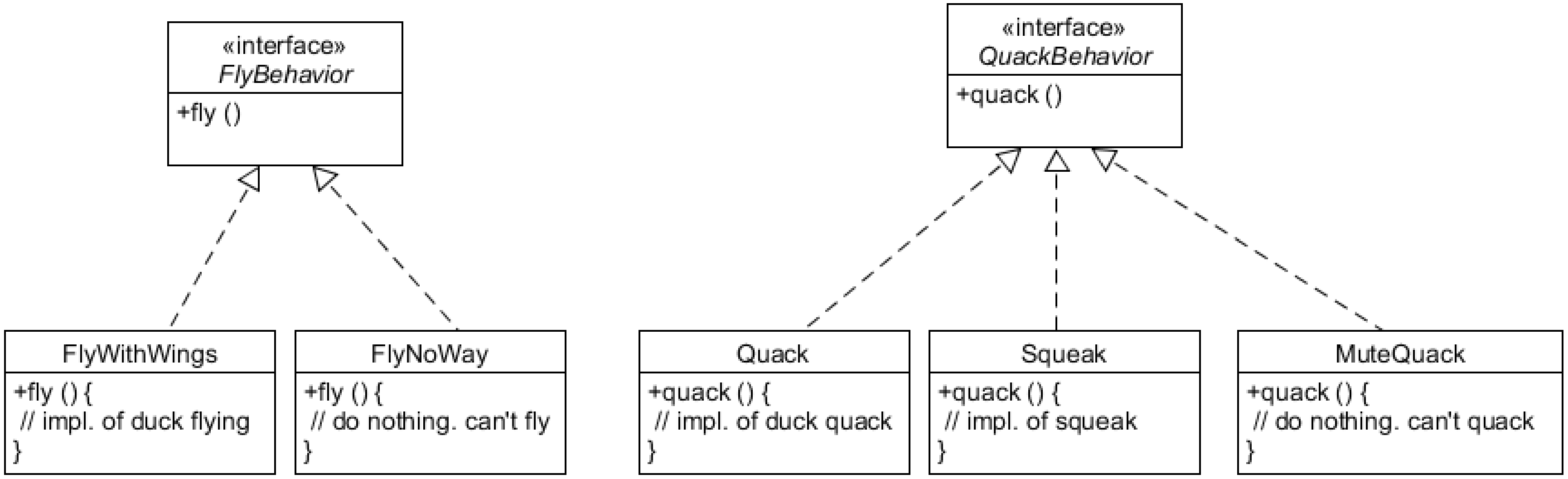
Separating what changes from what stays the same

- What is changing in SimDuck?
  - Flying behavior and quack behavior
  - Each type of behavior has a family
- To separate these behaviors from the Duck class, we'll
  - pull both methods out of the Duck class
  - Create a new set of classes to represent each behavior

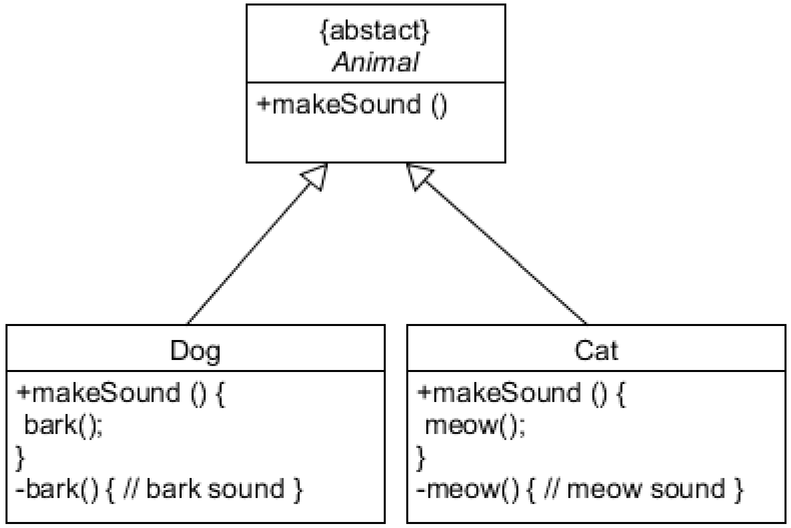


Design Principle

- Program to an interface, not an implementation
    - Programming to a super type
    - Exploit Polymorphism
    - The declared type of the variables should be a super type
        - The class declaring them doesn't have to know the actual object types



Program to interface, not an implementation



- **Programming to an implementation** would be:

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

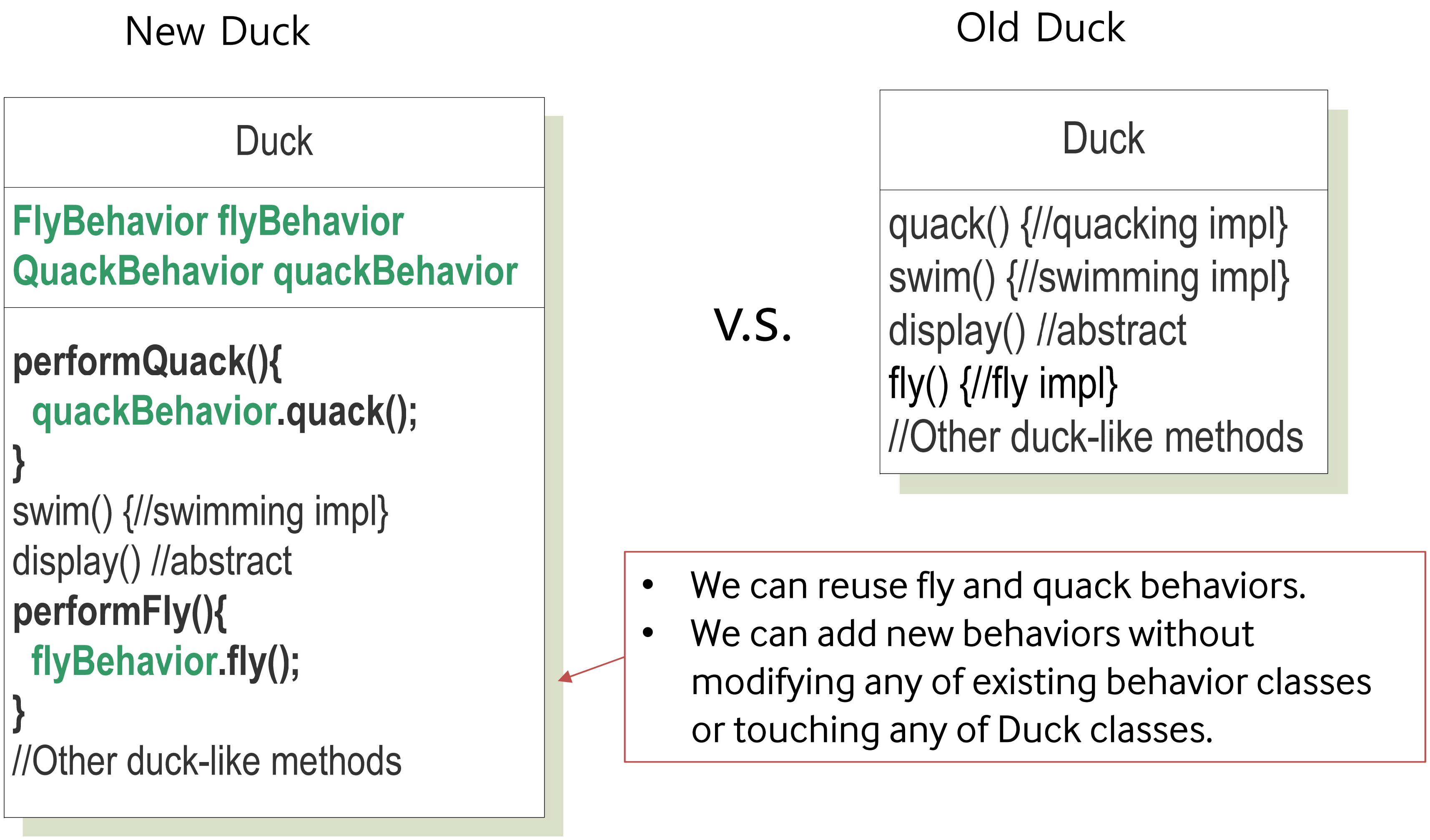
- **Programming to an interface/supertype** would be:

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

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

```
Animal a = getAnimal("Dog");
a.makeSound();
```

Improved Design



New Duck

```
Duck
FlyBehavior flyBehavior
QuackBehavior quackBehavior
performQuack(){
  quackBehavior.quack();
}
swim() {//swimming impl}
display() //abstract
performFly(){
  flyBehavior.fly();
}
//Other duck-like methods
```

V.S.

Old Duck

```
Duck
quack() {//quacking impl}
swim() {//swimming impl}
display() //abstract
fly() {//fly impl}
//Other duck-like methods
```

- We can reuse fly and quack behaviors.
- We can add new behaviors without modifying any of existing behavior classes or touching any of Duck classes.

Class Duck

```
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!”);
   }
}
```

Class Duck

```
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!”);
   }
}
```

object composition

object composition

delegation

delegation

Interface FlyBehavior

```
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”);
   }
}
```

Interface QuackBehavior

```
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”);
   }
}
```

Concrete Ducks

```
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”);
   }

}
```

Test Drive!

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

```
public abstract class Duck {
   ...
   QuackBehavior quackBehavior;
   FlyBehavior flyBehavior;
   public void performQuack() {
      quackBehavior.quack();
   }
   public void performFly() {
      flyBehavior.fly();
   }
...
}
```

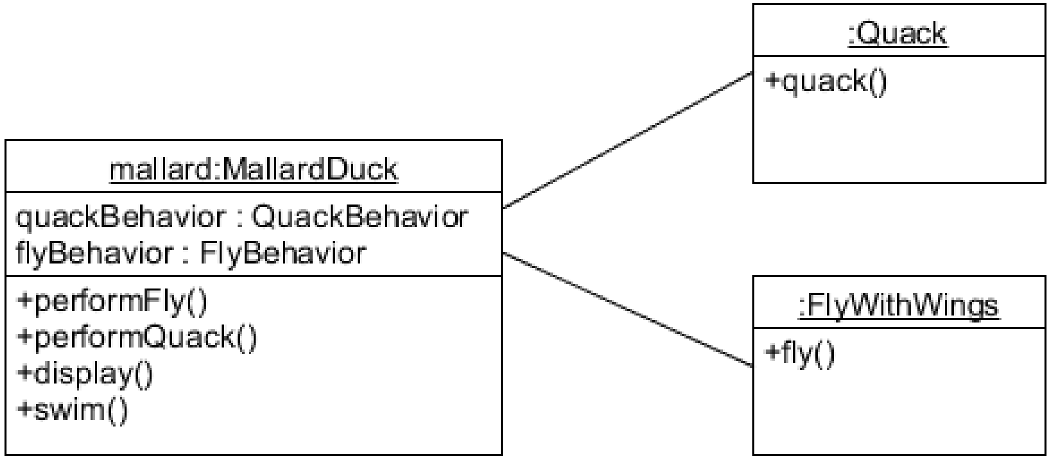
Output

```
Quack
I'm flying!!
```

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

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

Object Diagram for a Mallard Duck

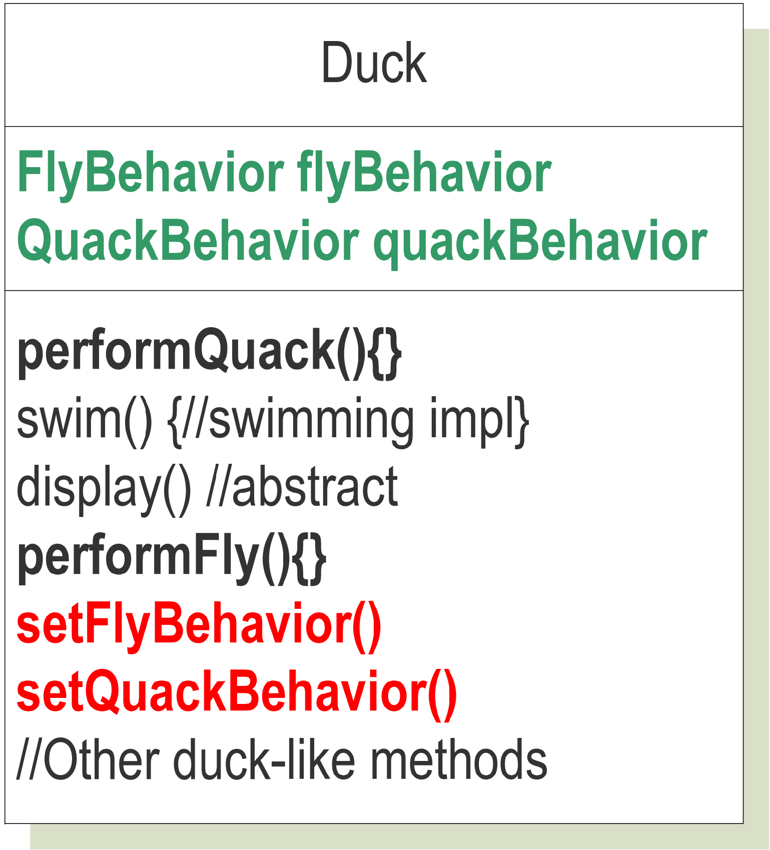


```
public abstract class Duck {
   ...
   QuackBehavior quackBehavior;
   FlyBehavior flyBehavior;
   public void performQuack() {
      quackBehavior.quack();
   }
   public void performFly() {
      flyBehavior.fly();
   }
...
}
```

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

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

Setting Behavior Dynamically



Duck

FlyBehavior flyBehavior
QuackBehavior quackBehavior

performQuack(){}
swim() {//swimming impl}
display() //abstract
performFly(){}
setFlyBehavior()
setQuackBehavior()
//Other duck-like methods

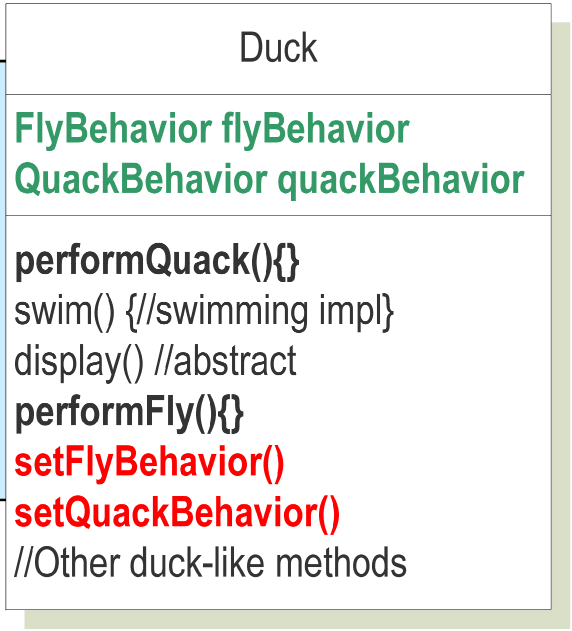
Setting behavior dynamically

(1) Add two new methods to the Duck class

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

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

We can call these methods anytime we want to change the behavior of a duck on the fly.



Duck

FlyBehavior flyBehavior
QuackBehavior quackBehavior

performQuack(){}
swim() {//swimming impl}
display() //abstract
performFly(){}
setFlyBehavior()
setQuackBehavior()
//Other duck-like methods

Setting behavior dynamically (cont’d)

(2) Make a new Duck type (ModelDuck.java)

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

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

(3) Make a new FlyBehavior type (FlyRocketPowered.java)

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

Setting behavior dynamically (cont'd)

(4) Change the test driver, and make ModelDuck rocket-enabled!

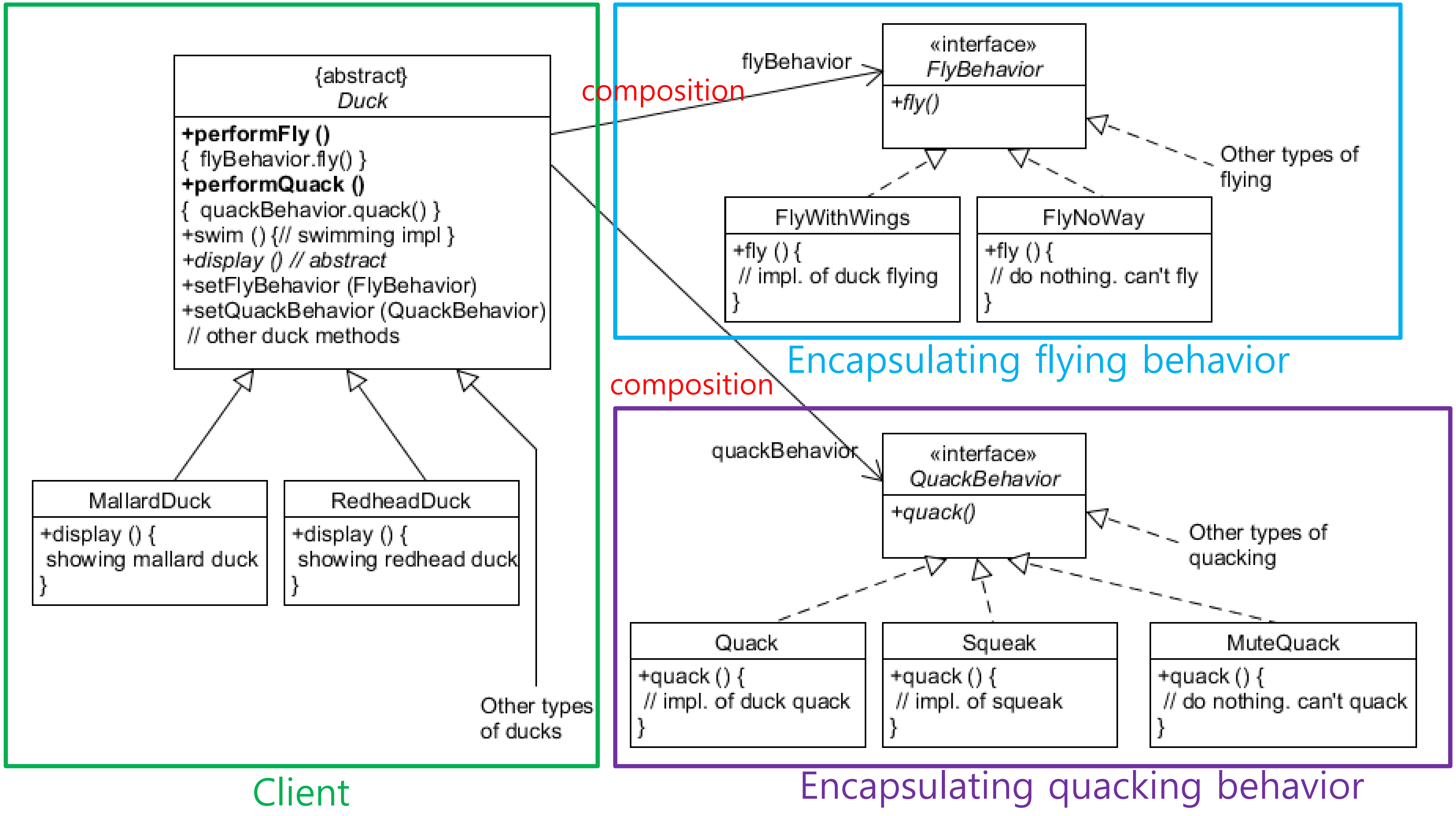
```java
public class MiniDuckSimulator {
   public static void main(String [] args) {
      Duck mallard = new MallardDuck();
      mallard.performQuack();
      mallard.performFly();

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

Output

```
Quack
I'm flying!!
I can't fly
I'm flying with a rocket
```

The Big Picture



Design Principle: Favor composition over inheritance

- Reusing functionality in object-oriented systems
  - Class Inheritance: white-box reuse
  - Object Composition: black-box reuse
- Class Inheritance (or subclassing)
  - allows a subclass' implementation to be defined in terms of the parent class' implementation.
  - the parent class implementation is often visible to the subclasses.
  - (+) done statically at compile-time and is easy to use
  - (-) the subclass becomes dependent on the parent class implementation
  - (-) the implementation inherited from a parent class cannot be changed at run-time

Object Composition & Delegation

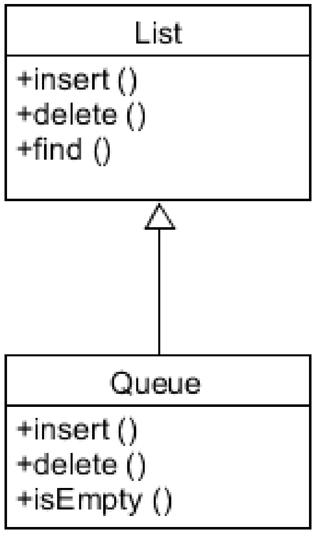
- Object Composition
  - objects are composed to achieve more complex functionality
  - requires that the objects have well-defined interfaces since the internals of the objects are unknown
  - functionality is acquired dynamically at run-time by objects collecting references to other objects

    (+) implementations can be replaced at run-time

    (+) there are less implementation dependencies

    (-) behavior of the system may be harder to understand just by looking at the source code
- Delegation
  - you pass method calls to a composed object
- Favor (Object) Composition over (Class) Inheritance!
  - Most design patterns emphasize object composition over inheritance whenever it is possible

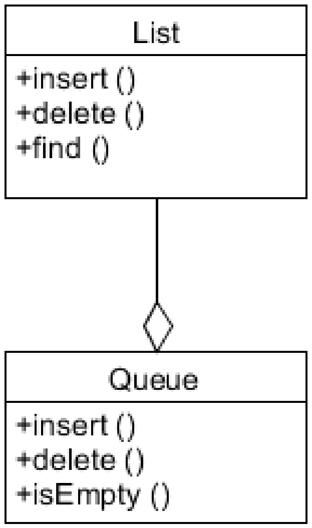
Favor Composition over Inheritance



List
+insert ()
+delete ()
+find ()

Queue
+insert ()
+delete ()
+isEmpty ()

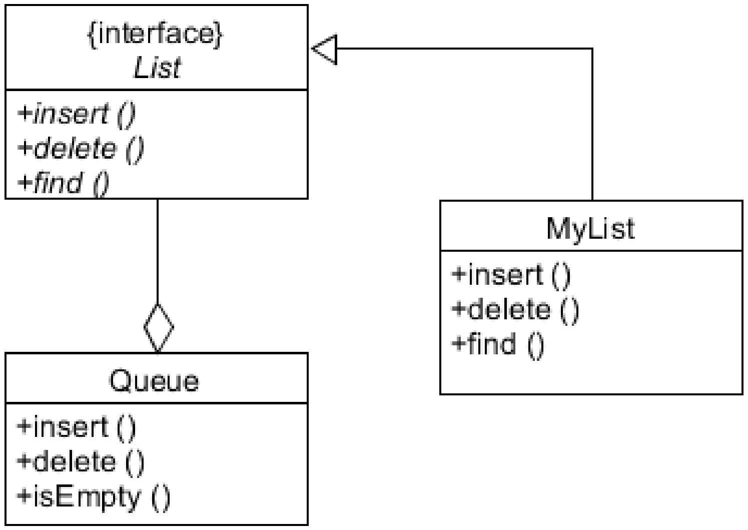
Violation of LSP!



List
+insert ()
+delete ()
+find ()

Queue
+insert ()
+delete ()
+isEmpty ()

Good



{interface}
List
+insert ()
+delete ()
+find ()

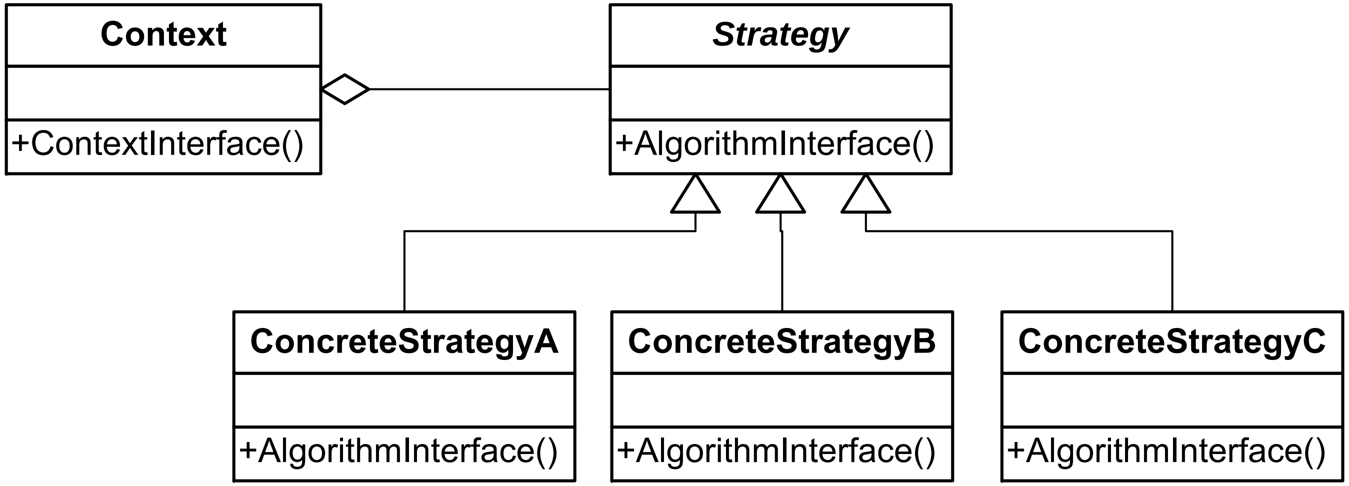
MyList
+insert ()
+delete ()
+find ()

Queue
+insert ()
+delete ()
+isEmpty ()

Better!

The Strategy Pattern

- Strategy Pattern
  - Defines a family of algorithms,
  - Encapsulates each one,
  - And makes them interchangeable
- Strategy Pattern lets the algorithm vary independently from clients that use it



Sorting Example

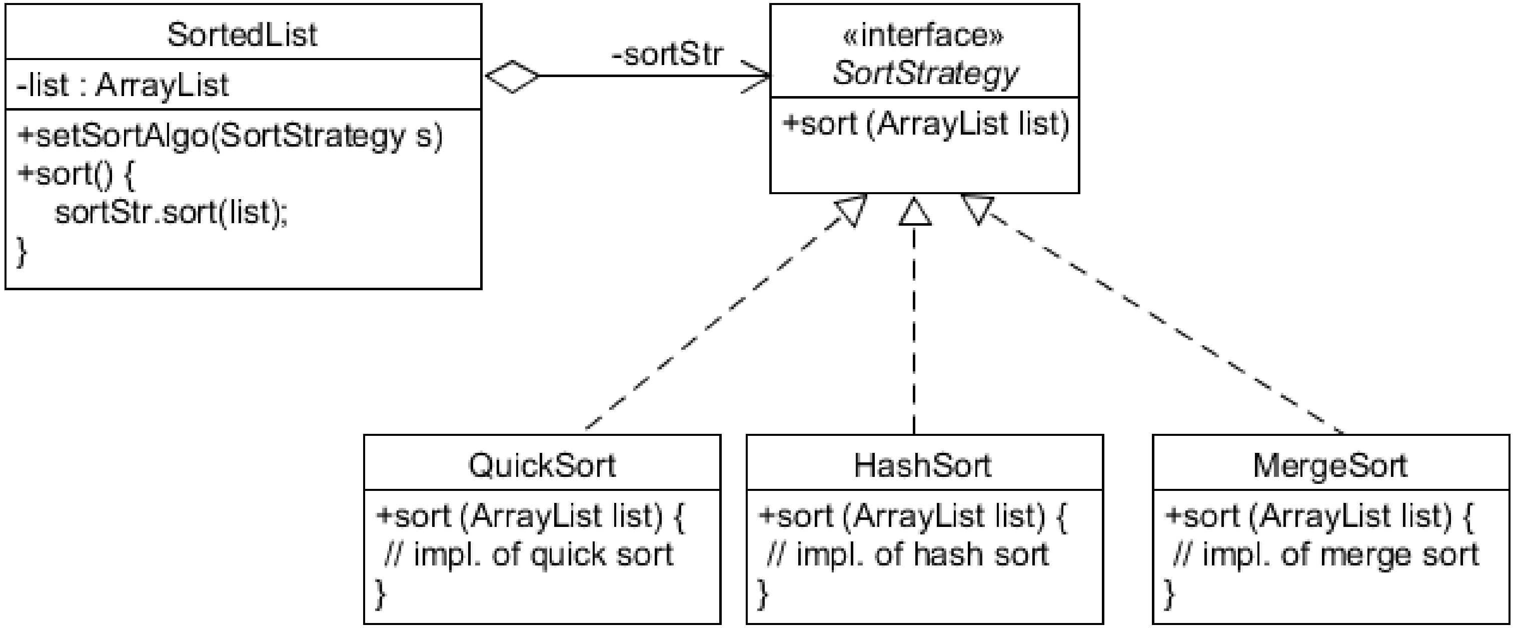
- **Requirement:** we want to sort a list of integers using different sorting algorithms, e.g. quick sort, selection sort, insertion sort, etc.

- One way to solve this problem is to write a function for each sorting algorithm, e.g.
  - quicksort(int[] in, int[] res)
  - insertionsort(int[] in, int[] res)
  - mergesort(int[] in, int[] res)

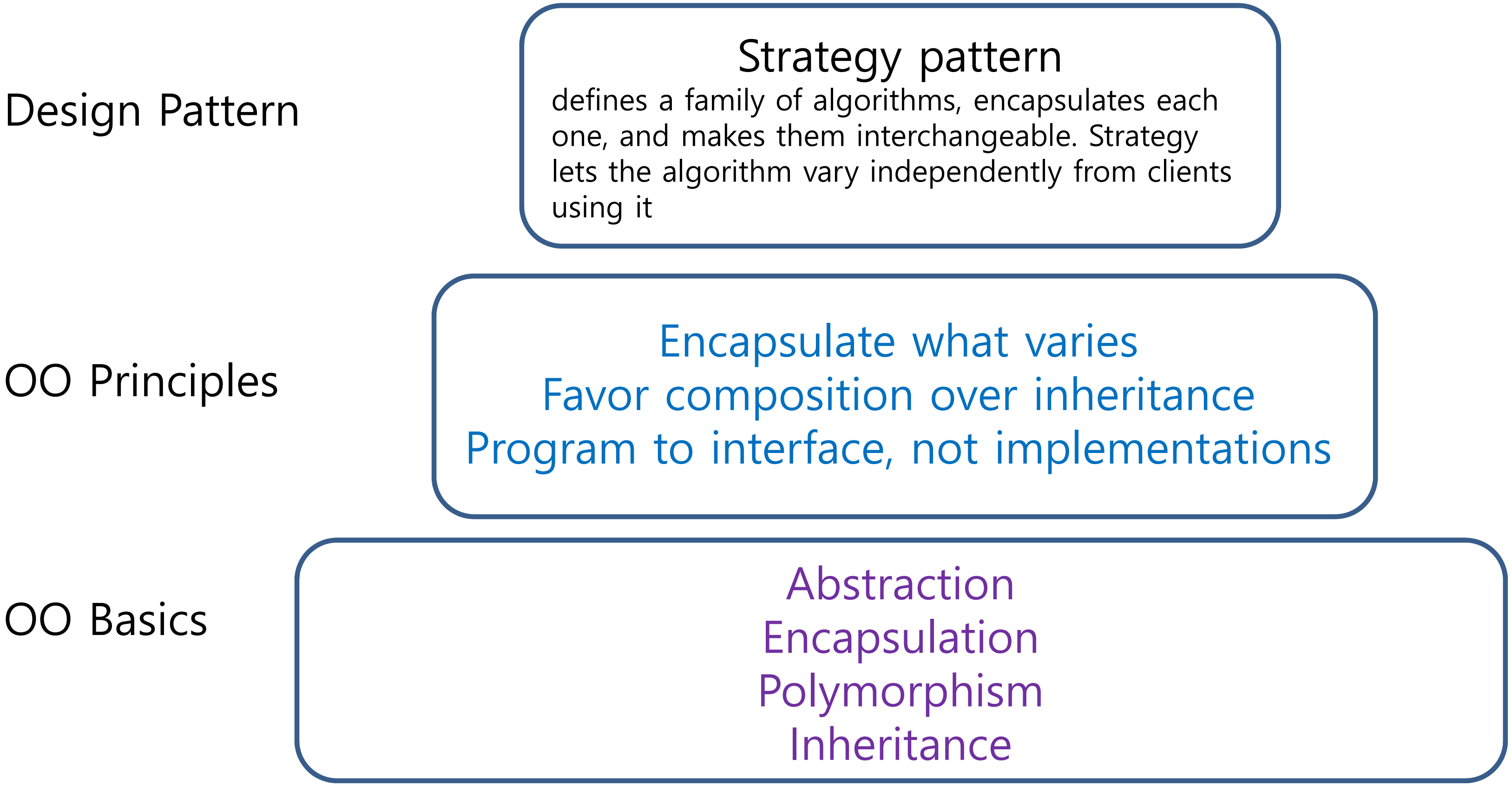
- A better way is to use the Strategy pattern

Sorting Example

```
void MethodOfClient(ArrayList al) {
    SortedList sl = new SortedList(al);
    sl.setSortAlgo( new QuickSort() );
    sl.sort();
}
```



Hierarchy of Pattern Knowledge



Design Pattern

Strategy pattern

defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients using it

OO Principles

Encapsulate what varies
Favor composition over inheritance
Program to interface, not implementations

OO Basics

Abstraction
Encapsulation
Polymorphism
Inheritance

Design Patterns

- Knowing the OO basics does not make you a good OO designer
- Good OO design are reusable, extensible and maintainable
- Patterns don’t give you code, they give you general solutions to design problems
- Most patterns and principles address issues of change in software.

Way to Learn Design Patterns

- Learning how such a pattern was discovered
  - When people begin to look at design patterns, they often focus on the solutions the patterns offer. This seems reasonable because they are advertised as providing good solutions to the problems at hand.
  - However, this is starting at the wrong end. When you learn patterns by focusing on the solutions they present, it makes it hard to determine the situations in which a pattern applies. This only tells us *what to do* but **not** ***when to use it*** or ***why to do it***.

Design Puzzle

① Arrange the classes.

② Identify one abstract class, one interface and eight classes.

③ Draw arrows between classes.

  a. Draw this kind of arrow for inheritance(“extends”). ◁————————

  b. Draw this kind of arrow for interface(“implements”). ◁- - - - - - - - - - - -

  c. Draw this kind of arrow for “HAS-A”. ◇————————>

④ Put the method setWeapon() into the right class.

```
void setWeapon (WeaponBehavior w) {
    this.weapon = w;
}
```

⑤ Fill the method fight() in Character class.

Design Puzzle

```
Prince
+display() { ... }
```

```
Queen
+display() { ... }
```

```
AxeBehavior
+useWeapon() {
  // impl. attack with Axe
}
```

```
«interface»
WeaponBehavior
+useWeapon()
```

```
King
+display() { ... }
```

```
KnifeBehavior
+useWeapon() {
  // impl. attack with Knife
}
```

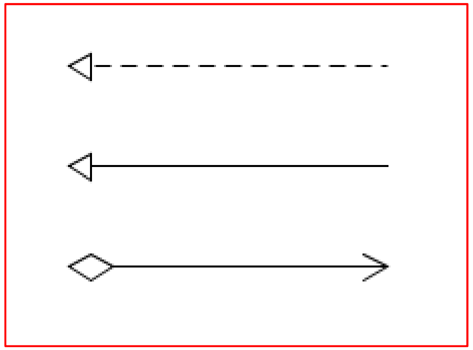
```
SwordBehavior
+useWeapon() {
  // impl. attck with Sword
}
```

```
{abstract}
Character
WeaponBehavior weapon
+display()
+fight() {
 // TO DO delegation
}
```

```
BowAndArrowBehavior
+useWeapon() {
  // impl. attck with Bow and Arrow
}
```

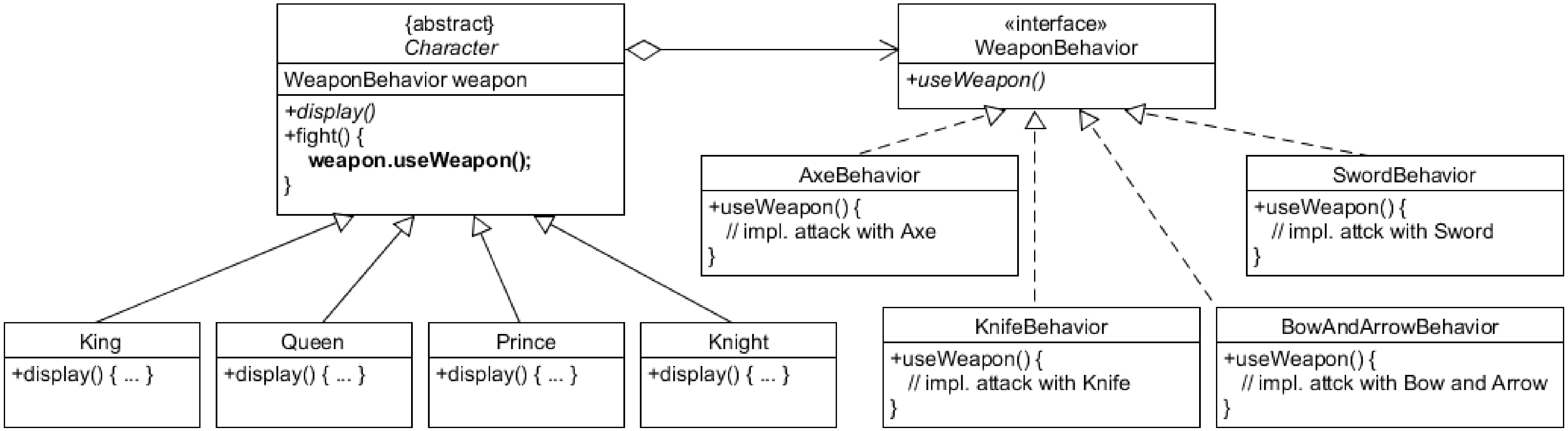
```
Knight
+display() { ... }
```

```
void setWeapon (WeaponBehavior w) {
    this.weapon = w;
}
```



- Put the method setWeapon() into the right class
- Fill the method fight() in Character class.

Solution



{abstract}
Character

WeaponBehavior weapon

```
+display()
+fight() {
    weapon.useWeapon();
}
```

«interface»
WeaponBehavior

+useWeapon()

King

+display() { ... }

Queen

+display() { ... }

Prince

+display() { ... }

Knight

+display() { ... }

AxeBehavior

```
+useWeapon() {
   // impl. attack with Axe
}
```

SwordBehavior

```
+useWeapon() {
   // impl. attck with Sword
}
```

KnifeBehavior

```
+useWeapon() {
   // impl. attack with Knife
}
```

BowAndArrowBehavior

```
+useWeapon() {
   // impl. attck with Bow and Arrow
}
```

- Design Principles
  - Encapsulate what varies
  - Program to interface, not implementations
  - Favor composition over inheritance

- Strategy pattern
  - lets the algorithm vary independently from clients that use it
  - uses composition & delegation mechanism