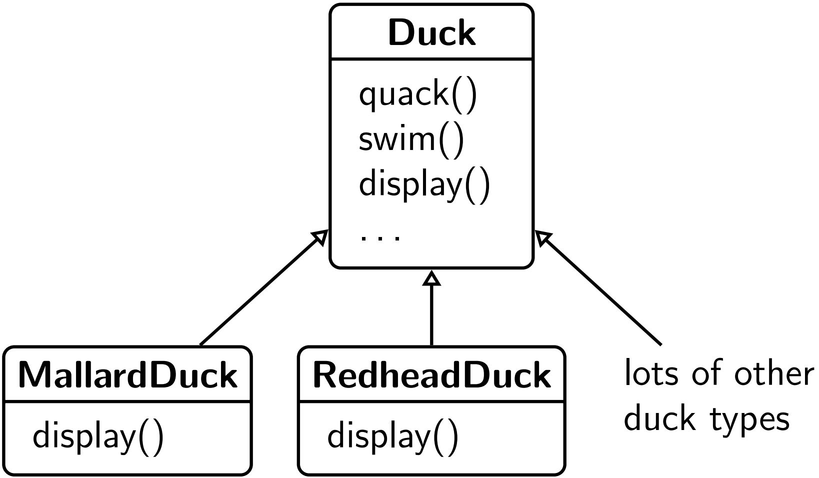
CS 351
Design of Large Programs
Strategy Pattern

September 20, 2021

Example: Duck Simulator

- Game has many duck species swimming and quacking
- Initial design has Duck superclass extended by other types
- Parent class has abstract display method implemented by child classes

Duck Class Hierarchy



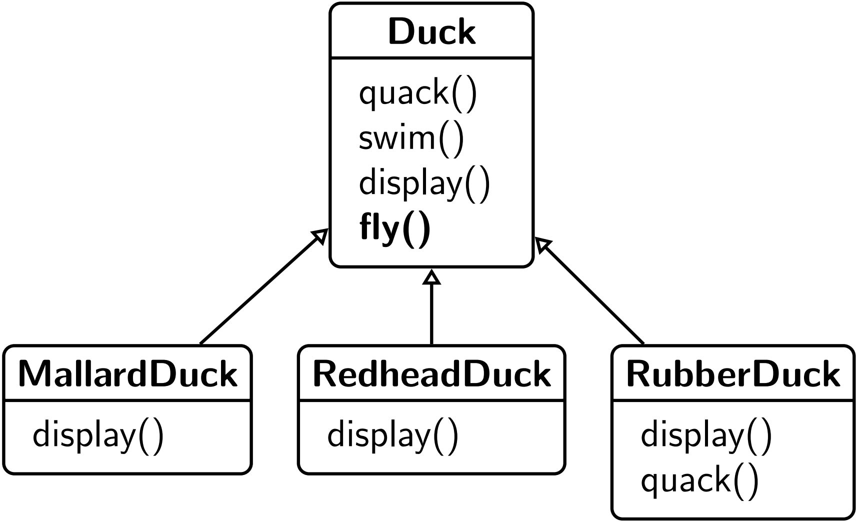
Requirements Change

- Let's make ducks fly!
- How hard can it be?

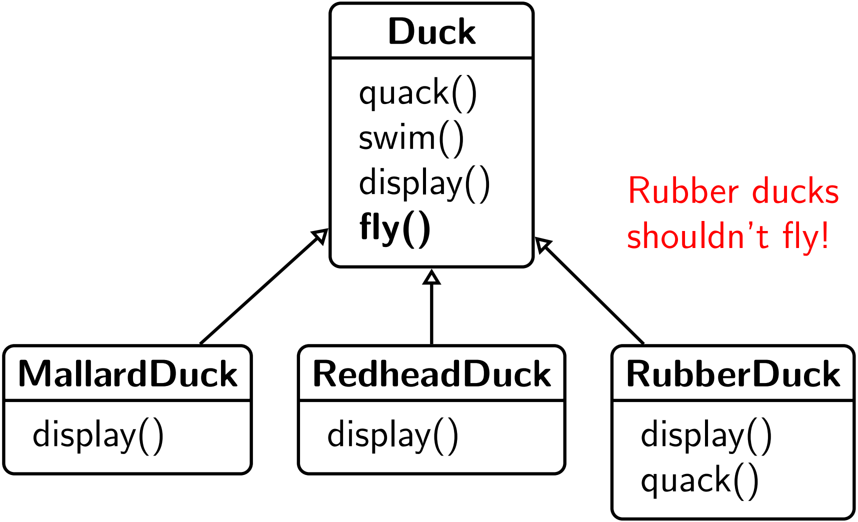
Requirements Change

- Let's make ducks fly!
- How hard can it be?
- Let's add a fly method to our Duck class and all the children will inherit it!

Duck Classes with Flying

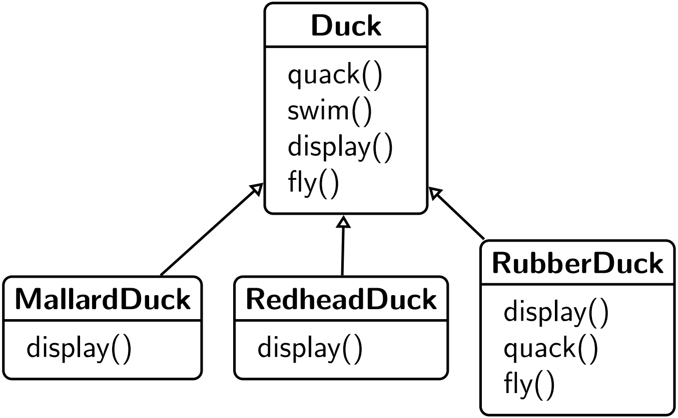


Duck Classes with Flying

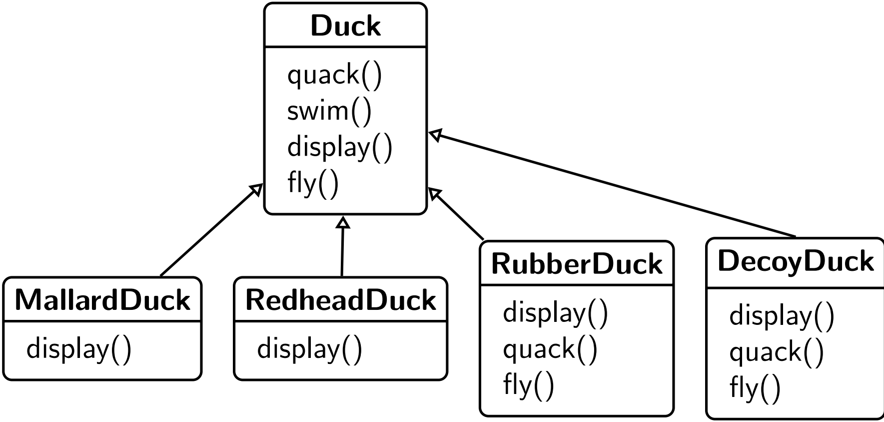


Rubber ducks shouldn't fly!

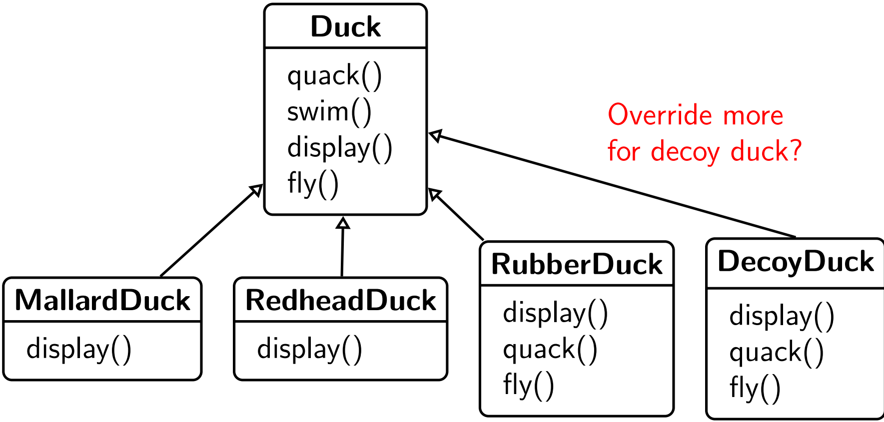
Just override fly for RubberDuck?



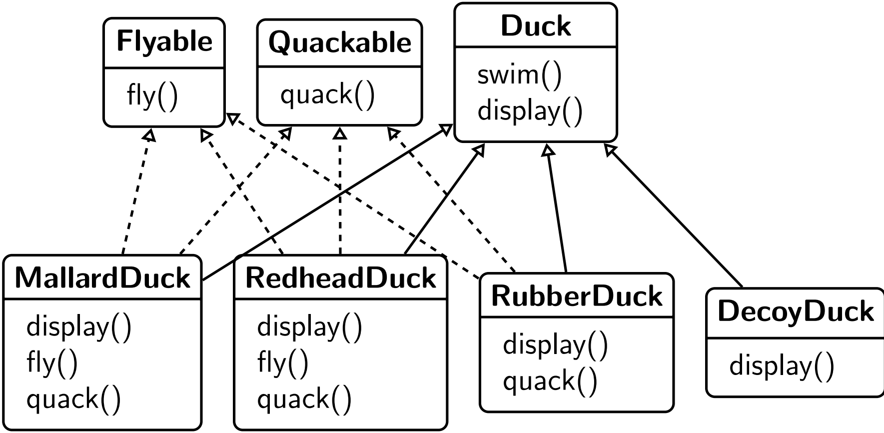
Just override fly for RubberDuck?



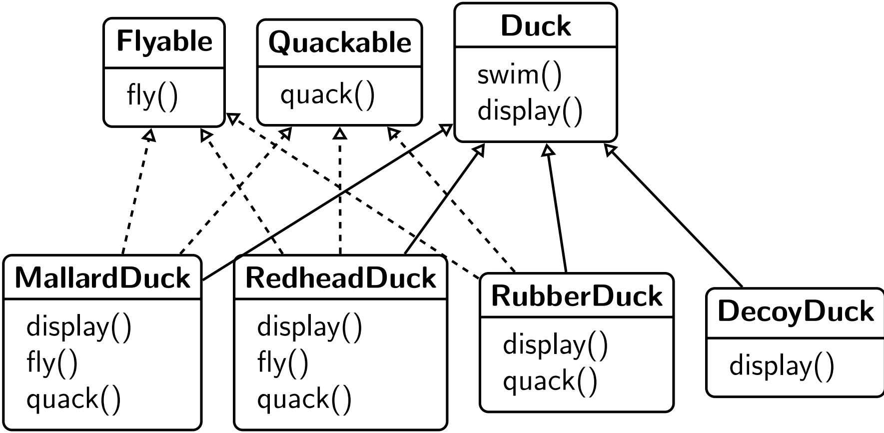
Just override fly for RubberDuck?



Use an interface?



Use an interface?



This has a lot of duplicate coding!

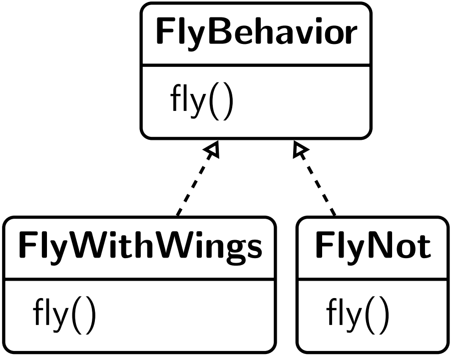
Design Principles

- 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*

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*
    - A Duck *has a* flying behaviour
    - A Duck *has a* quacking behaviour

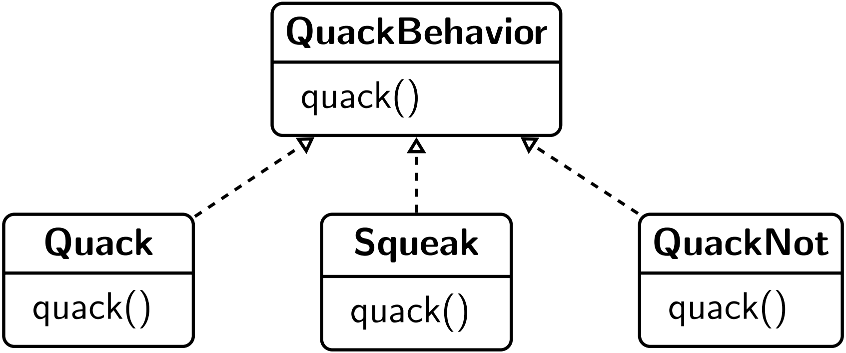
Program to an Interface: Flying



Flying implementation for ducks with wings

Implementation for ducks that can't fly

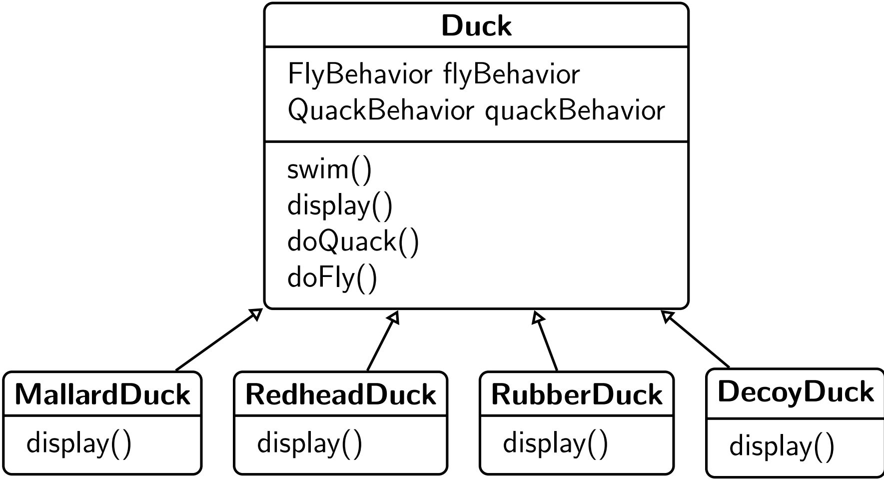
Program to an Interface: Quacking



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
- We can easily add new quacking and flying styles without impacting our ducks!

Duck Classes again



Example code

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

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

- Instead of quacking on its own, a Duck delegates that behavior to the quackBehavior object
- It doesn't matter what kind of Duck it is; all it matters is a Duck knows how to quack.

How to 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!");
  }
}
```

This is not quite right yet because we're still programming to the implementation (i.e., we have to know about the specific Quack behavior and FlyWithWings behavior).
We can fix this with another pattern. . . later. . .

Can Ducks learn to Quack and Fly?

- How could you teach a Duck a new way to quack or a new way to fly?
- Add new methods to the Duck class:

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

Favor Composition over Inheritance

- Stated another way... "has-a is better than is-a"
- Ducks *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)
    - 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.