

A group of ducks of various breeds, including mallards and brown ducks, are swimming in a pond. The water is calm, and the ducks are scattered across the frame. The text "Strategy Design Pattern" is overlaid in the center in white. The background is a light, overcast sky.

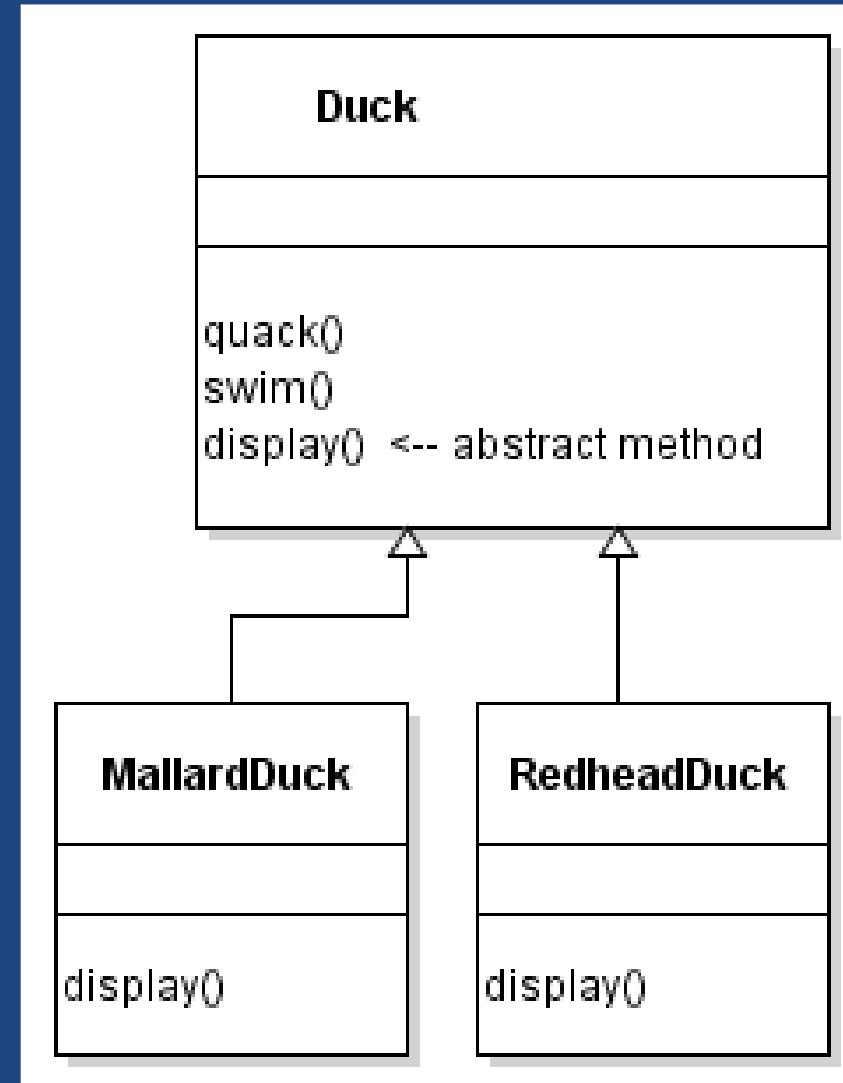
# Strategy Design Pattern

# Topics

- 1) What are the **limits of using inheritance**?
- 2) What **principles** can we use to evaluate an OOD?
- 3) How can we **configure an object with a new behaviour at runtime**? (flexibility)

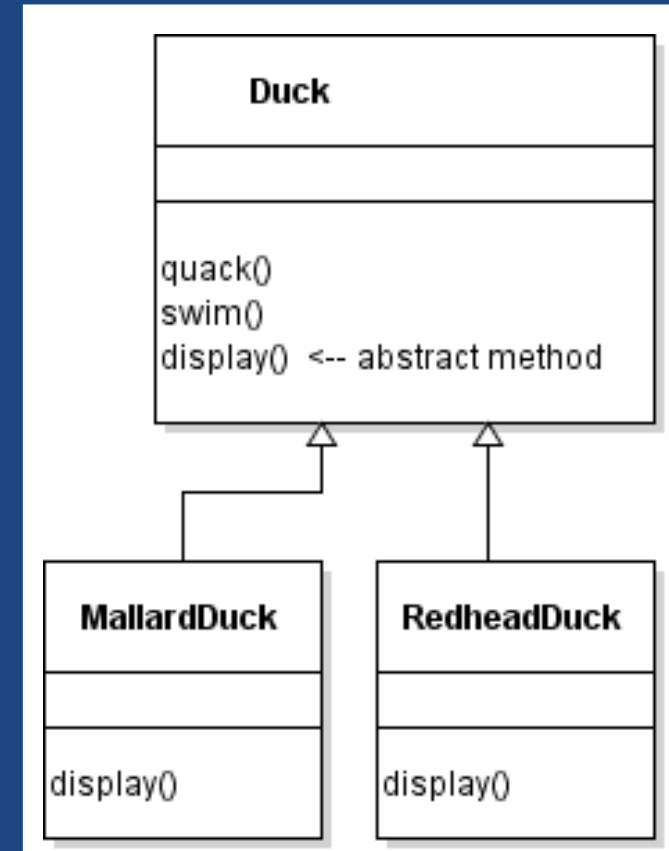
# Case Study

- We want to:
  - Create a **duck simulator** which shows ducks **swimming** and **quacking**
  - Make it **flexible** to add new features



# Case Study

- Inheritance good because:
  - : implement `quack()` and `swim()` just once
  - gives runtime polymorphism



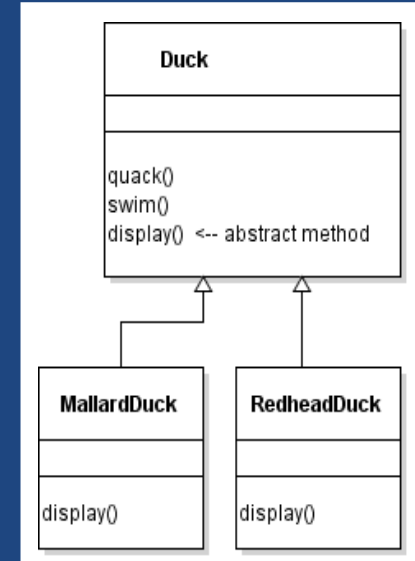
# Polymorphism

- ..
  - The specific method called is decided at runtime based on ..
  - **myDuck.display()** could run one of two (or more) implementations.

- **Static (compile time) polymorphism:**
  - method overloading, C++ template classes

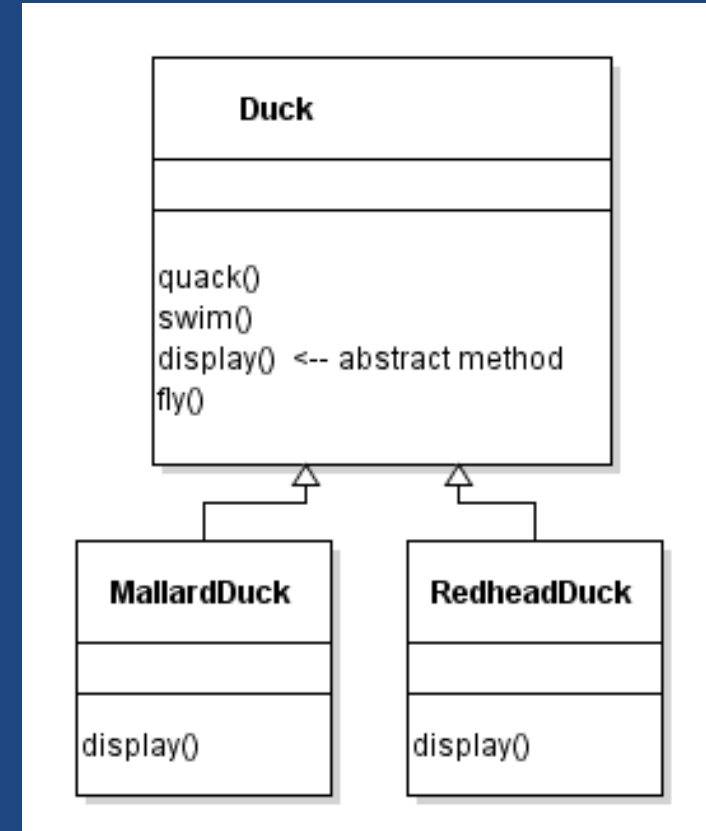
An  
aside

```
Duck myDuck;  
if (wantsMallard()) {  
    myDuck = new MallardDuck();  
} else {  
    myDuck = new RedHeadDuck();  
}  
myDuck.display();
```

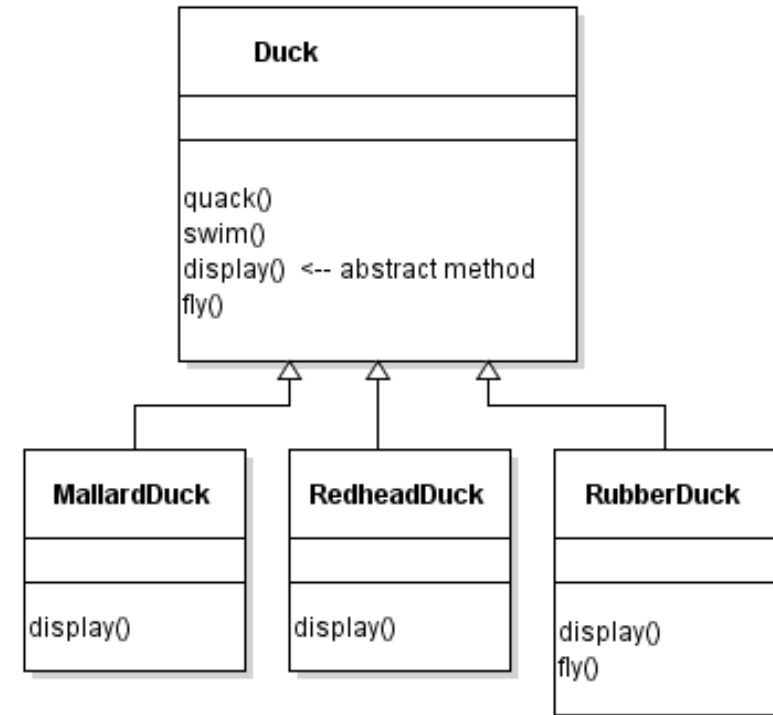


# Make Ducks Fly

- **Make ducks fly!**
  - Add **fly()** to base class
  - Derived classes get behaviour for free
- But, adding **fly()**  
..



# Make Ducks Fly



# Make Ducks Fly – Bad?

- **Inheritance is bad here because:**

– ..

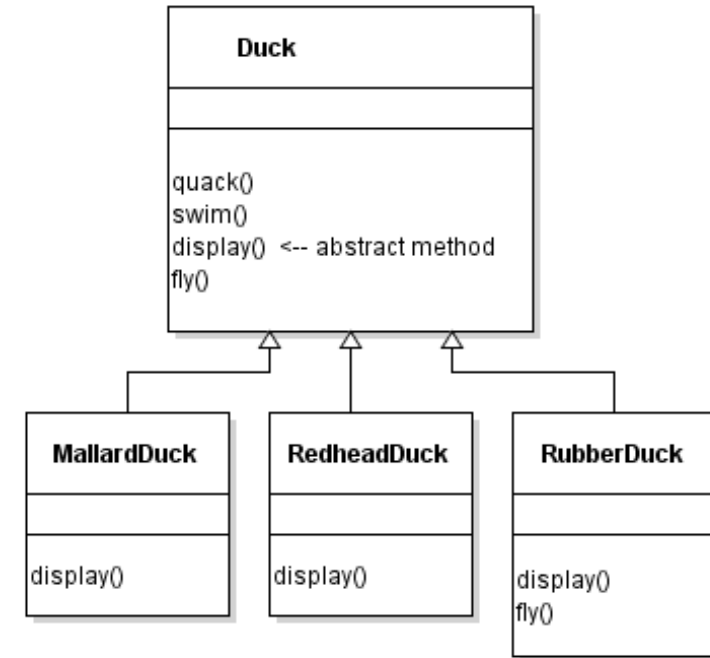
(non-local effects);

– ..

– ..

(fixed when instantiated)

- **Inheritance requires:**  
all base-class functionality to be shared by all derived classes



Polymorphism allows you to point to different objects, but once instantiated, a concrete object has but one behaviour/method

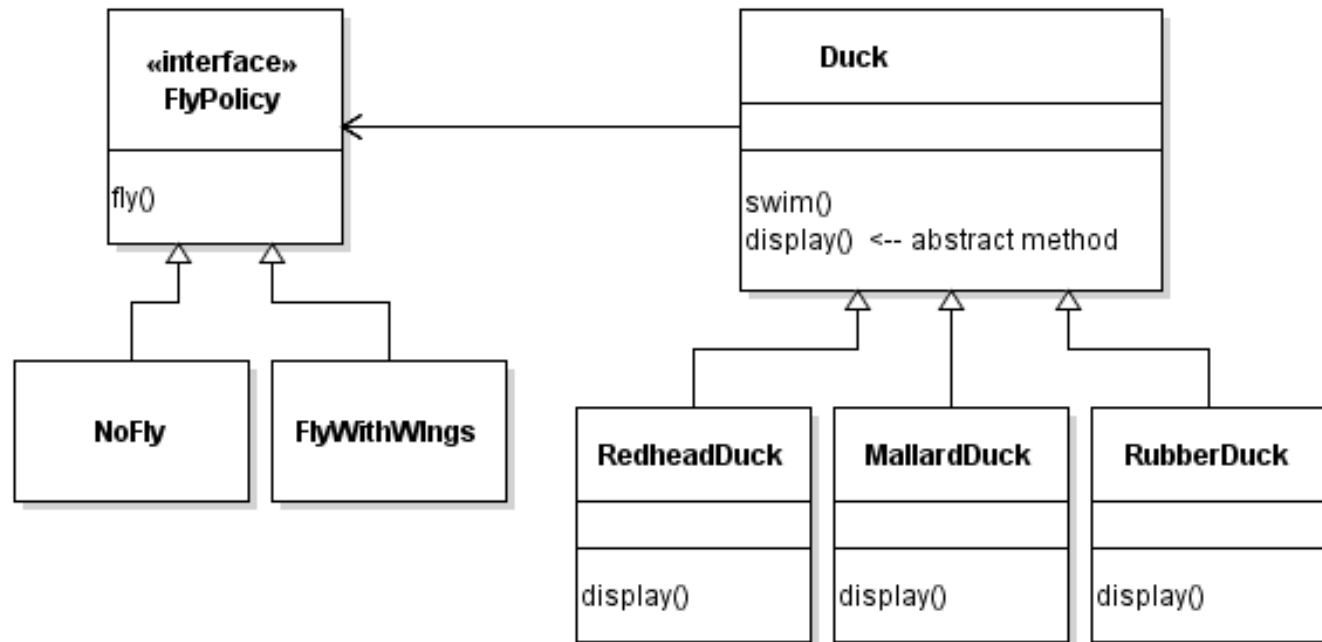


# Separate out Behaviours

- Put `fly()` into a family of..
  - Instantiate the desired fly policy;  
each **Duck** has-a **FlyPolicy**
- ..  
vs having its own hard coded policy

Create an supertype  
(interface or ABC);

All concrete policies  
(behaviours)  
implement this  
supertype.

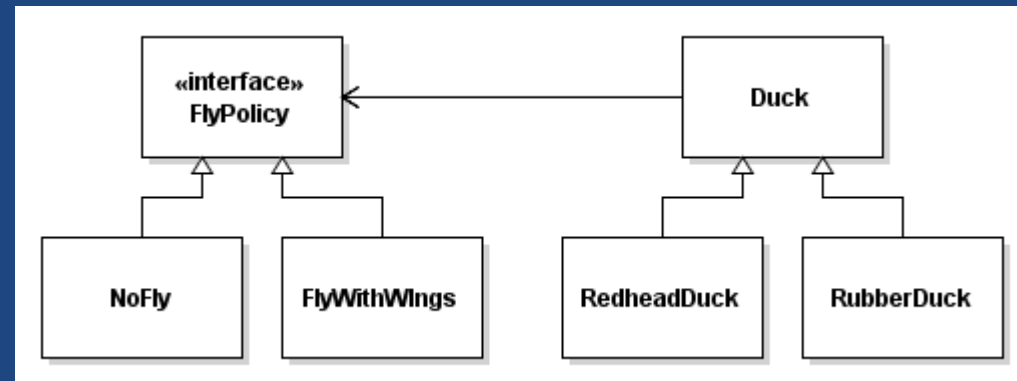


# Design Principle (Separate Change)

- Design Principle

..

- Limits the extent of a likely change by encapsulating that feature inside a class.



# Design Principle (OCP)

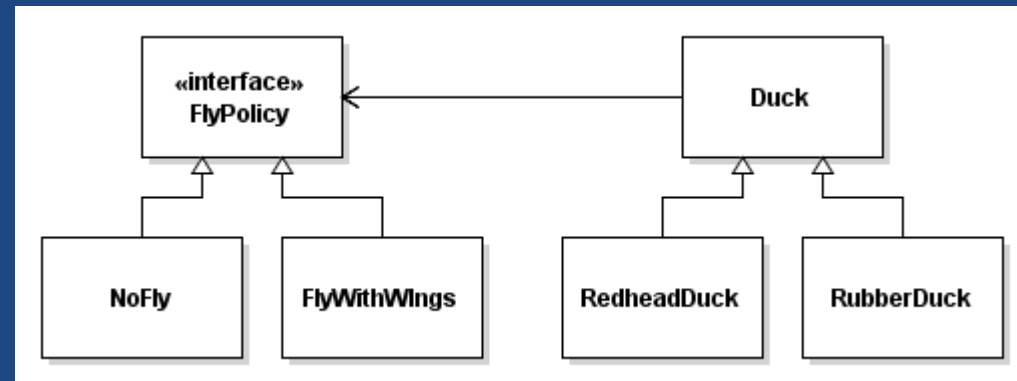
- Design Principle: Open-Closed Principle

- ..

- Ex: adding a new policy/behaviours should not require re-coding parts of existing system.

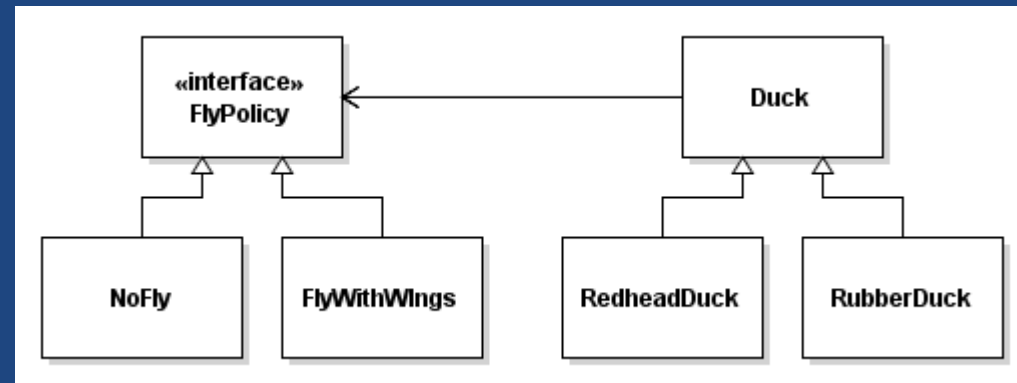
- ..

- not changing existing (tested/debugged) code, such as at the root of the inheritance tree.



# Design Principle (OCP cont)

- **Ex:** Observable classes can be extended with **new observers without modification**
- **Don't apply open/closed principle everywhere:**
  - adds extra **level of abstraction** (*complexity*)
  - only use in areas **expected to change**
- Predict likely changes with OOD experience and domain knowledge



# Design Principle (Interface over Impl.)

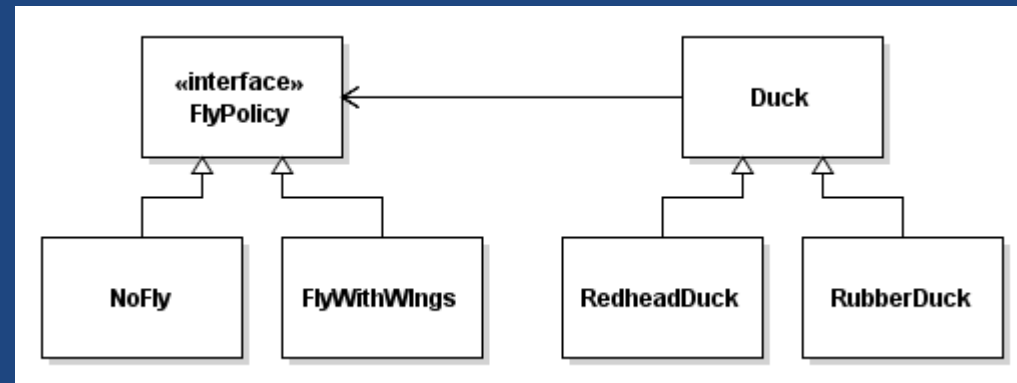
- Design Principle

- ..

- “interface” = super-type for polymorphism (interface/ABC)

- Code depend on just the supertype,  
not a concrete implementation

- the behaviour we get completely depends  
on the object we are given



# Code Example

```
abstract class Duck {
    FlyPolicy flyP;

    Duck(FlyPolicy flyP) {
        this.flyP = flyP;
    }

    void performFly() {
        flyP.fly();
    }

    abstract void display();
}
```

```
class MallardDuck extends Duck {

    MallardDuck() {
        super(new FlywithWings());
    }

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

- Enhancements

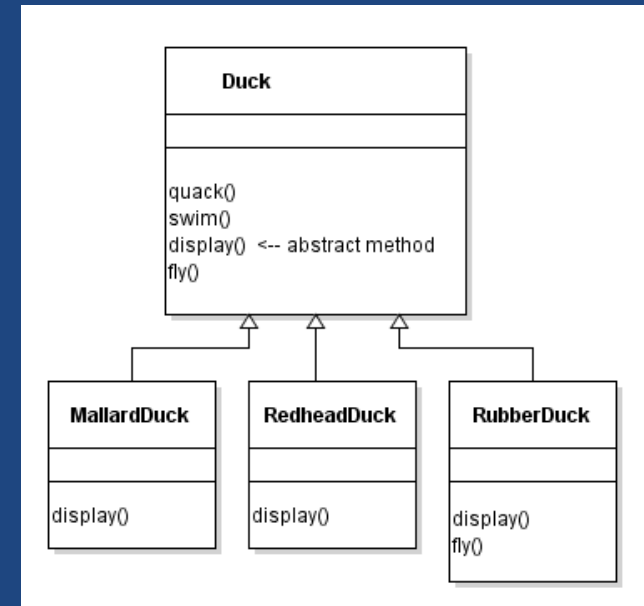
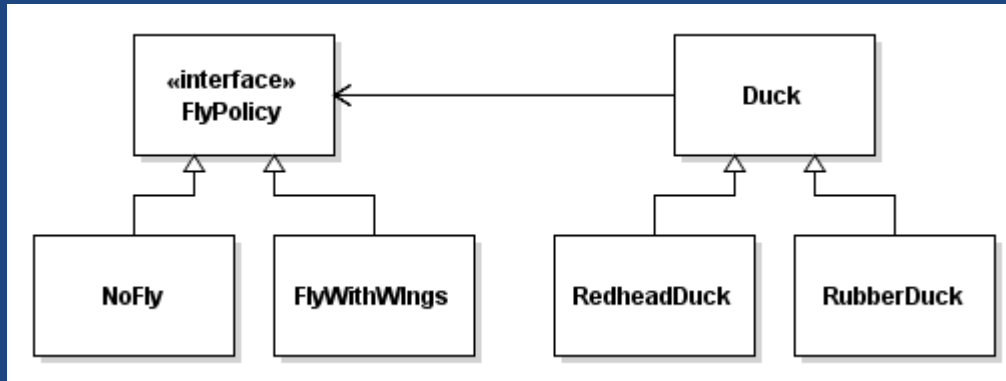
- Add a `setFlyPolicy()` to Duck so it can change at runtime.
- Imagine a game where the duck is configured/updated.

# Design Principle (Favour Composition)

- Design Principle

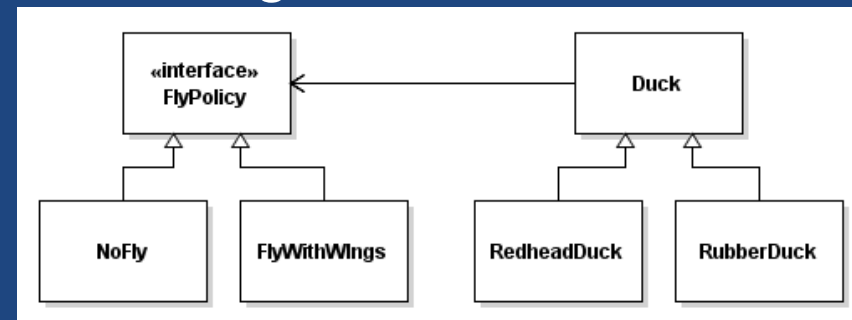
- ..

- runtime selectable behaviour, not dictated by rigid compile-time inheritance hierarchy



# Strategy Pattern

- Strategy Pattern..
  - Strategy lets the algorithm vary independently from clients that use it.
- Other applications
  - Tax codes for different provinces
  - Defining what to accept when searching file names





# Code

- See example
  - HeadFirstDesign --> strategy --> Duck
- Improvements
  - Duck's constructor accepts Quack/Fly policy
  - Make Duck fields private
  - NullQuack, NullFly?
- Discuss:
  - Do we even need different types of ducks?  
Just use policies for each variant?

# Inheritance vs Composition

- **Inheritance is still good!**
  - Reduces duplication, supports polymorphism.
  - Pull classes that change out of primary inheritance hierarchy (rigid) and into composition (flexible).  
Composition of these uses inheritance for flexibility
- **Use inheritance as long as it serves your needs;** you should not be locked in by it.
  - ..
  - Ex Classes: **Student, TA, Employee**  
vs **People** w/ role class **StudentRole, TARole, ...**

# Summary

- **Inheritance is limited because:**
  - local changes have non-local effects
  - inflexible for code maintenance
  - no run-time changes possible
- **Design Principles**
  - **Separate aspects that change** from those that stay the same.
  - Classes should be **open for extension**, but **closed for modification**.
  - **Program to an interface**, not an implementation
  - **Favour composition** over inheritance
- **Strategy Design Pattern**
  - **Encapsulate possible behaviours** into a family of interchangeable objects.