COMP 2511
Object Oriented Design & Programming

Story so far,

Basic OO principles

Abstraction, Encapsulation, Inheritance, Polymorphism

Basic refactoring techniques

Extract method, Rename variable, Move Method, Replace Temp With Query

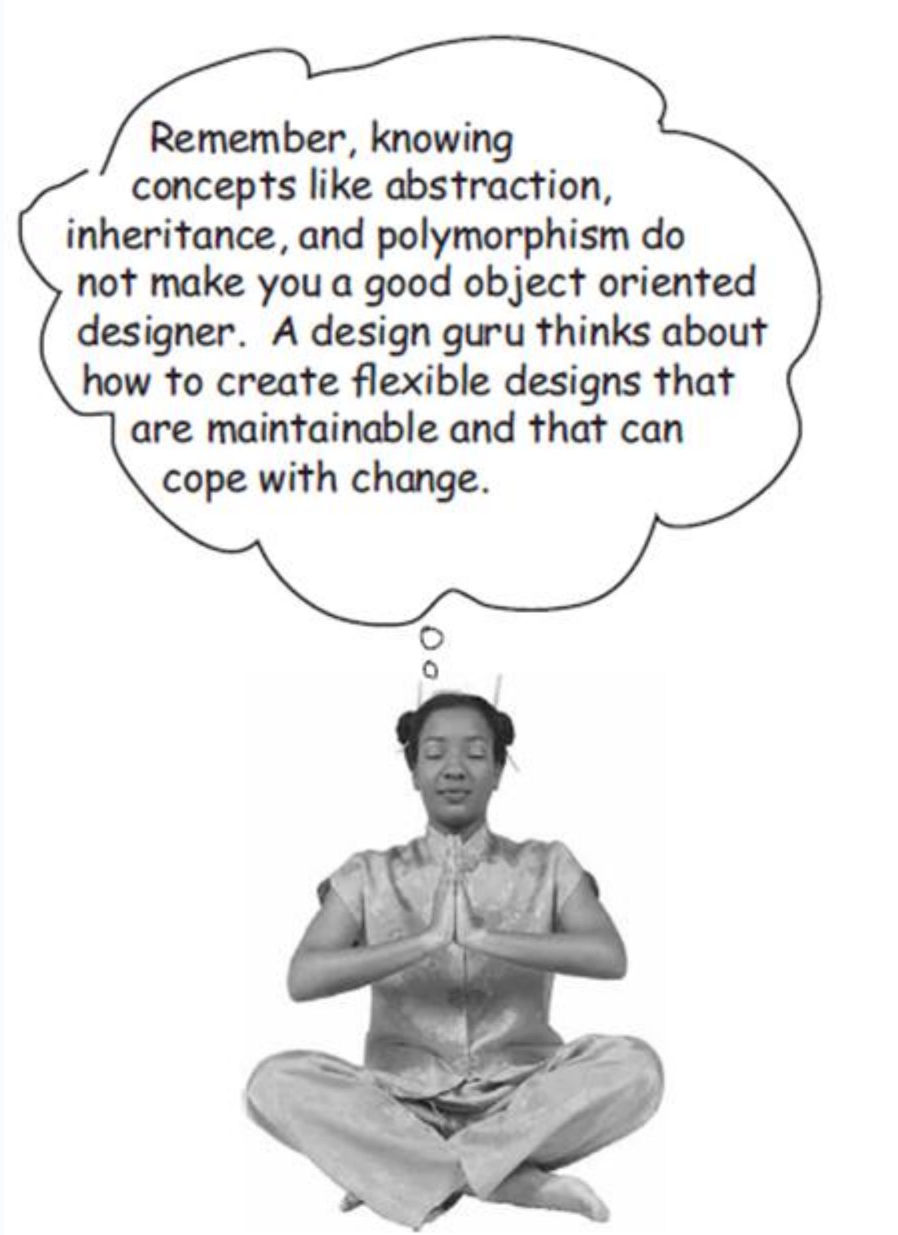
This week

OO design principles

- Encapsulate what varies
- Program to an interface, not an implementation
- Favour composition over inheritance

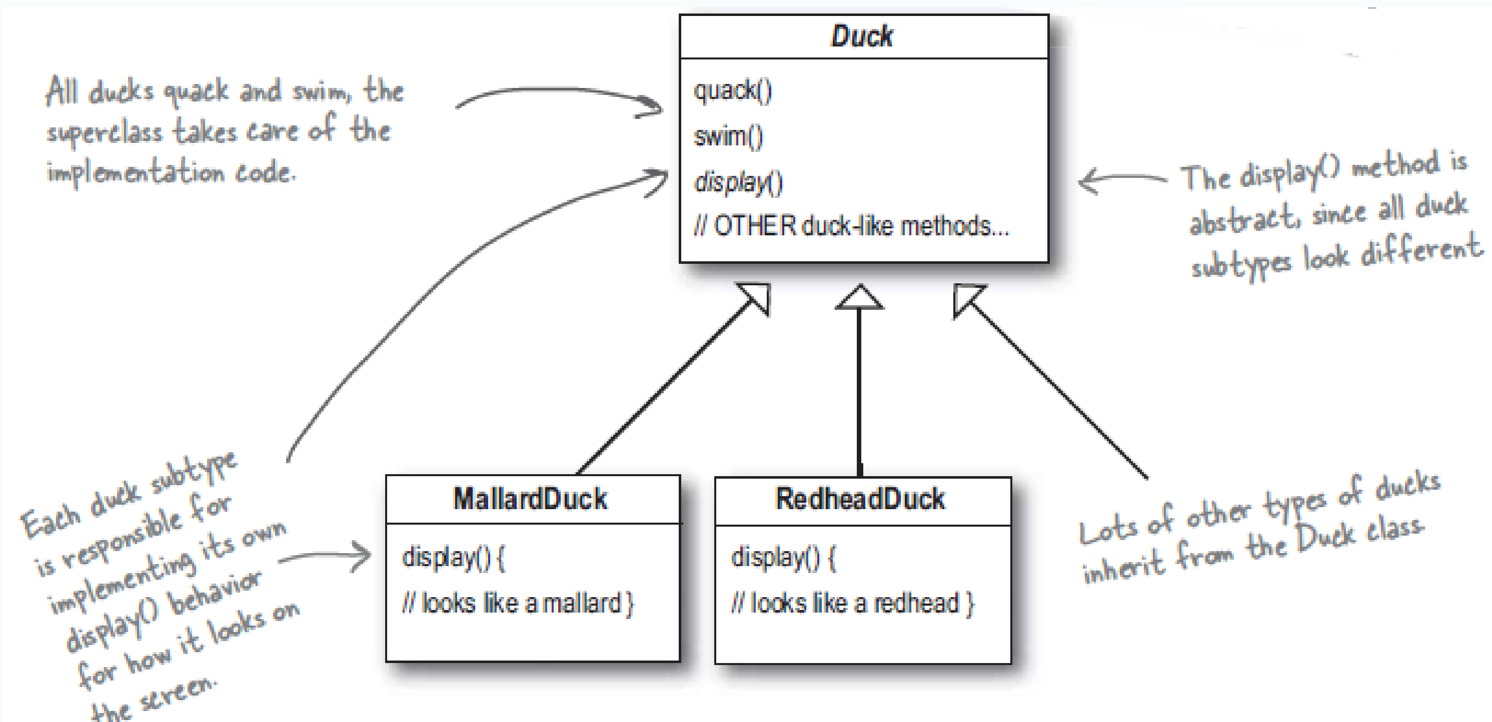
Design Patterns

- Strategy and State Patterns


Remember, 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.

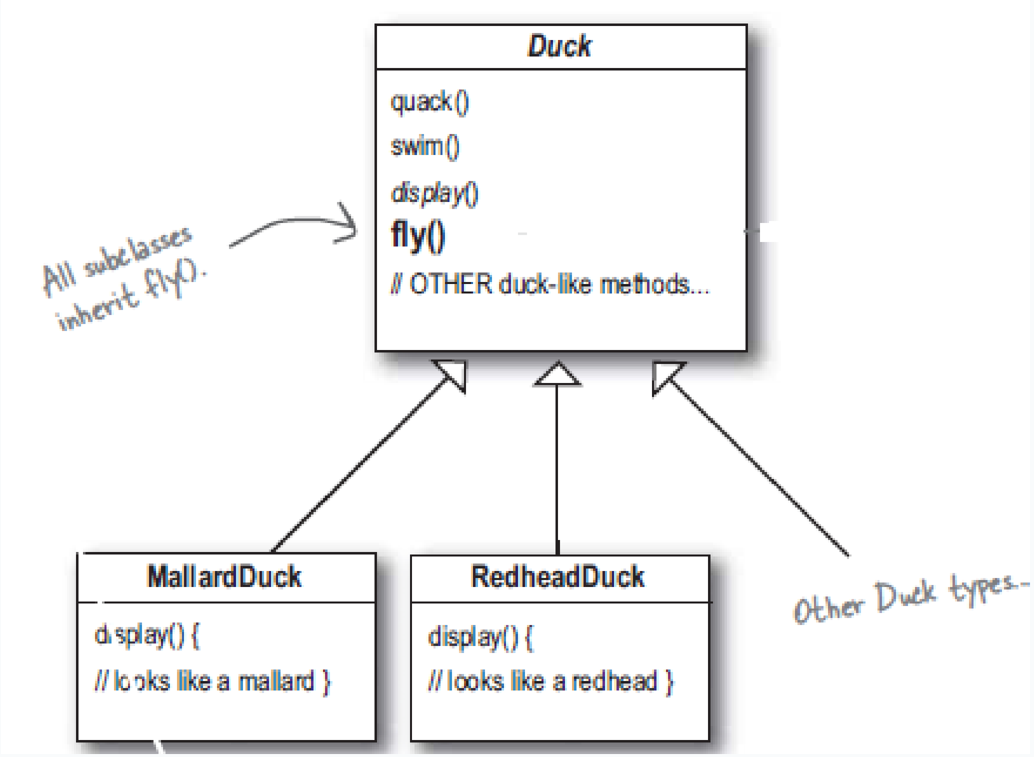
A simple Duck Simulator App

- A game that shows a large variety of duck species swimming and making quacking sounds
- What we need is a base class `Duck` and sub-classes `MallardDuck`, `RedheadDuck` for the different duck species



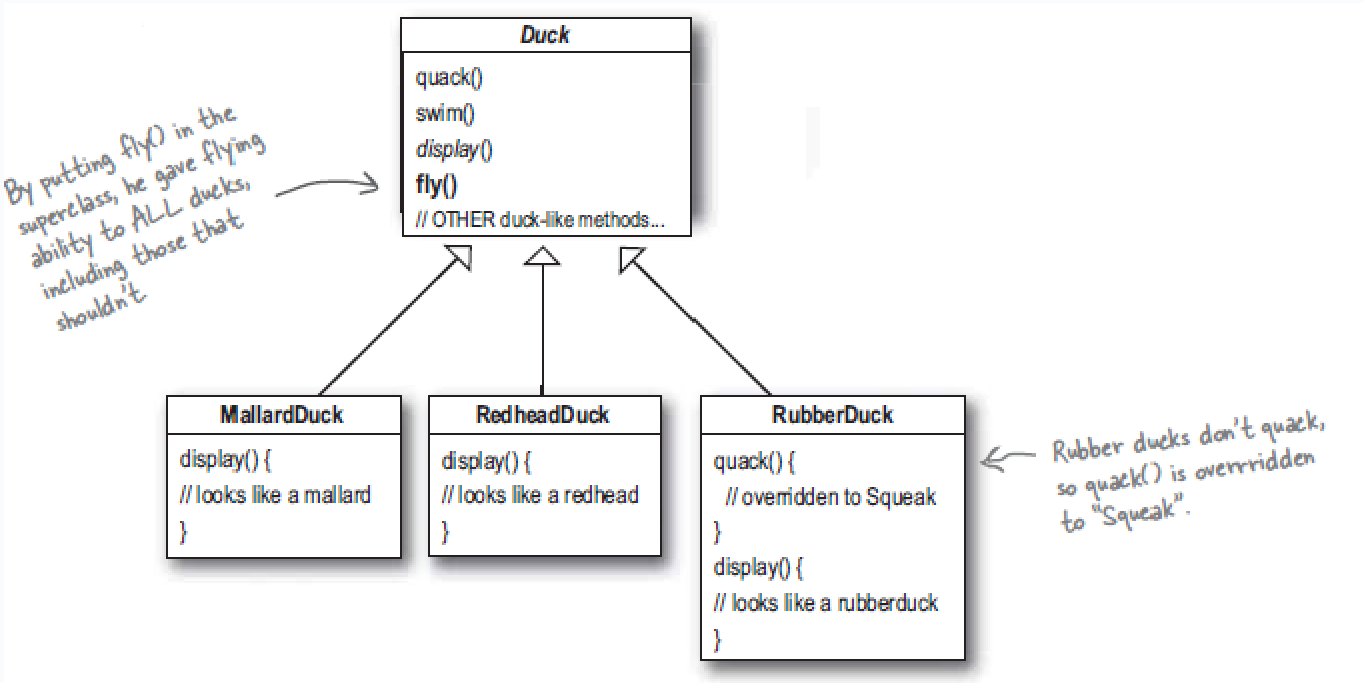
But, now we need ducks that fly

- How hard can that be?
- Just add a `fly()` method to the class Duck and all sub-types will inherit it



Design Flaw…

- A localised update to the code caused a non-local side effect (*flying rubber ducks*)
- What normally is thought of as a great use of inheritance for the purpose of "reuse" actually didn't turn out so well, when it comes to maintenance



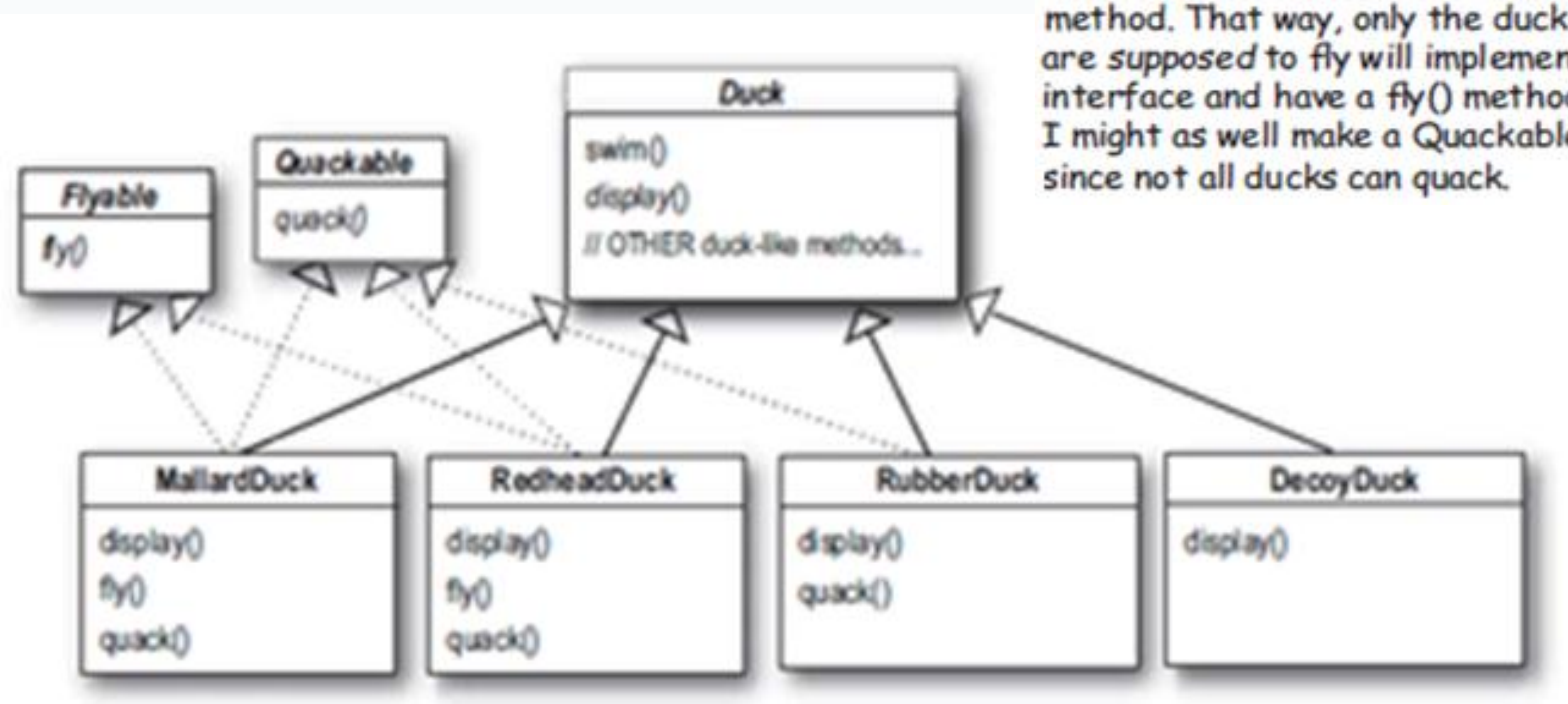
Solution 1…

- Solution 1:
  - We could simply override fly() method to do nothing, just as the quack() method was overridden
  - But, what happens, when more different types of ducks were added that didn't quack or fly

Solution 2…

- Solution 2:
  - Need a cleaner way, so that only some ducks fly and some quack. How about an interface?

could take the fly() out of the Duck superclass, and make a ***Flyable() interface*** with a fly() method. That way, only the ducks that are *supposed* to fly will implement that interface and have a fly() method… and I might as well make a Quackable, too, since not all ducks can quack.



Solution 2…

- Using interfaces, all sub-classes that fly implement *flyable* interface and that quack implement *Quackable* interface
- But, completely destroys code reuse – every class must implement *fly()* method (perhaps not an issue in Java 8), but what if there are more than two kinds of flying behaviour among ducks that fly.
- Change every class where behaviour has changed? – **maintenance night-mare**
- Need a design pattern to come riding on a white horse and save the day.

- Is there a way to build software so that when we need to change it, we could do so with the least possible impact on the existing code?

Solution

Design Principle #3:

Identify aspects of your code that varies and "encapsulate" and separate it from code that stays the same, so that it won't affect your real code.

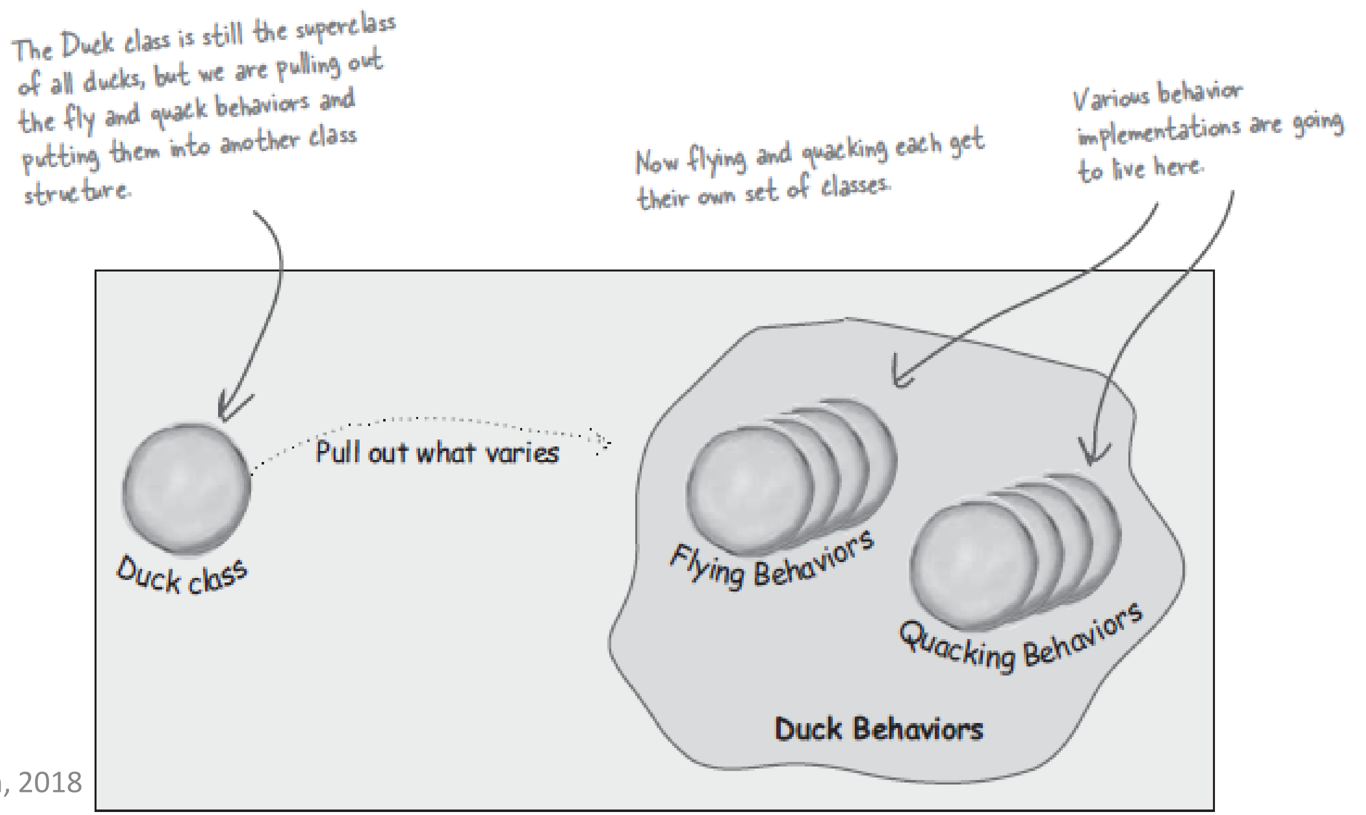
- By separating what changes from what stays the same, the result is fewer unintended consequences from code changes and more flexibility in your software

- Another way to think about this principle: *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.*

So, let us pull out the duck behaviour from the duck class

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.



- How are we going to design the set of classes that implement the fly and $quack$ behaviour?

Design Principle #4:

Program to a an interface, not to an implementation

- Program to an interface, really means “program to a super-type” i.e., the declared type of the variable should be a super-type (abstract class or interface)
  - e.g., Dog d = new Dog(); d.bark(); // programming to an implementation
  - Animal a = new Dog(); a. makeSound(); // programming to an interface
- What we want is to exploit polymorphism by programming to a super-type so that actual run-time object isn’t locked into the code

Programming to a super-type

Previously,

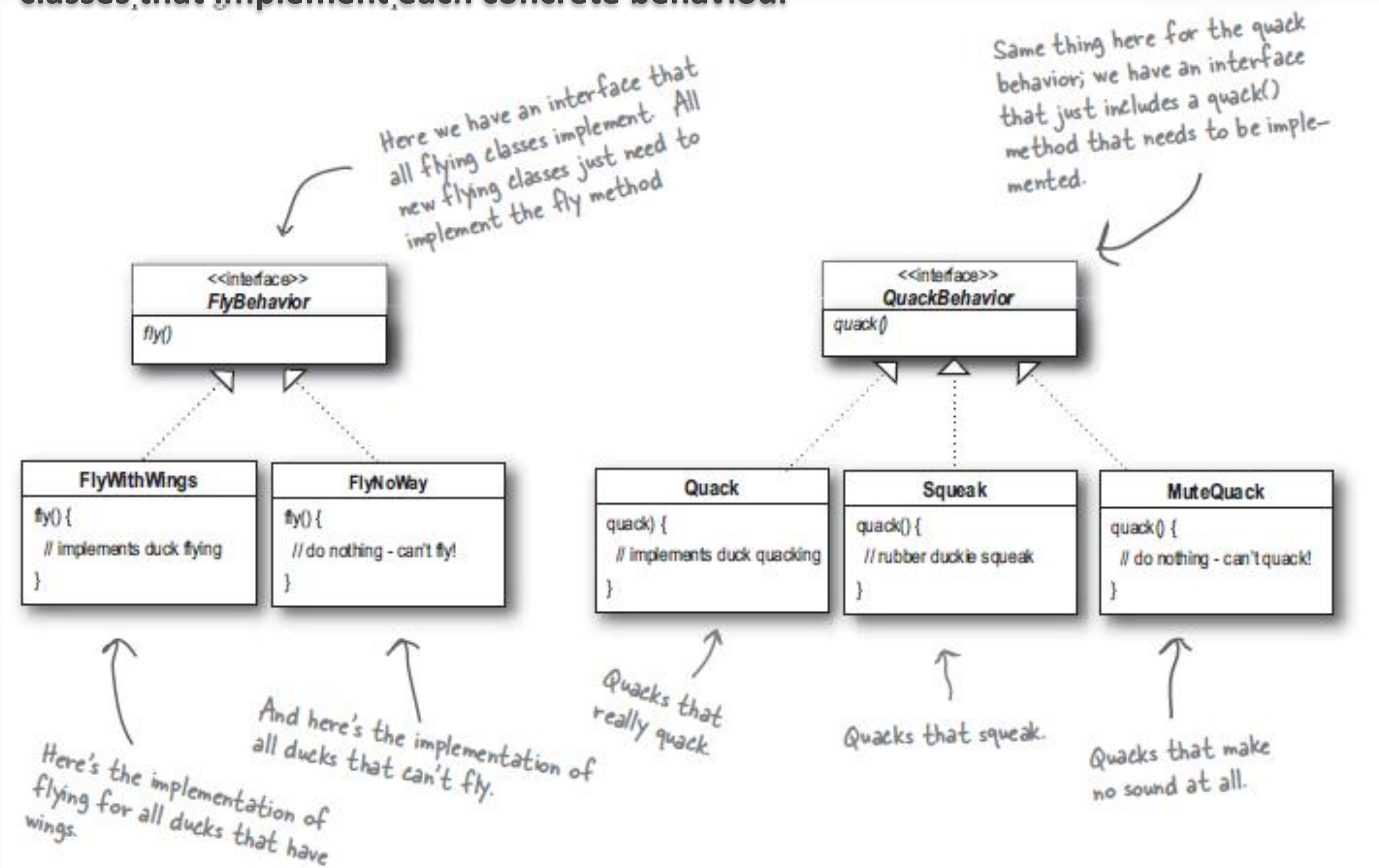
- A behaviour was locked into a concrete implementation in the Duck class or a specialised implementation in the sub-class
- Either way, we were locked into using a specific implementation
- There was no room for changing that behaviour

Now,

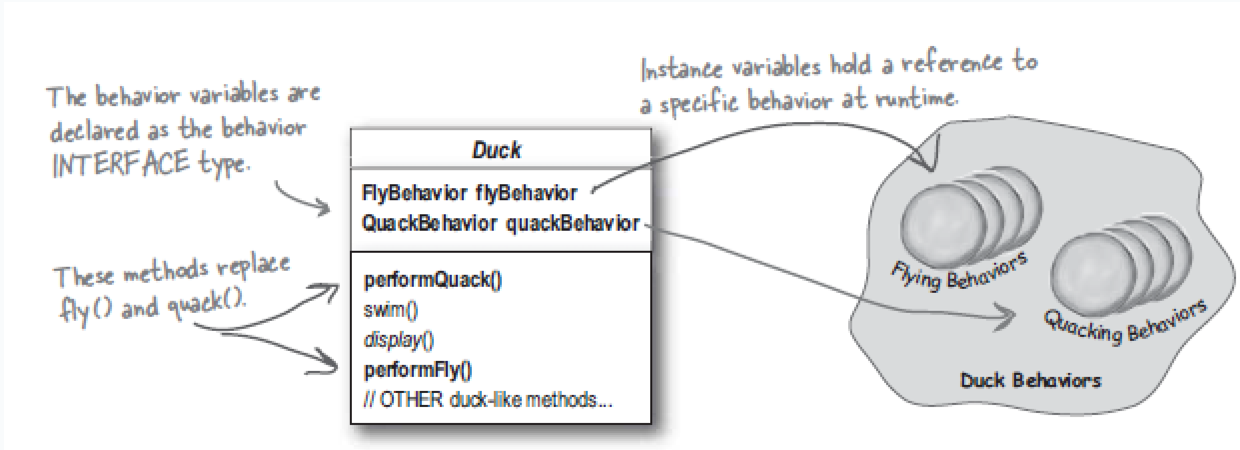
- Use an interface to represent the behaviour
- Implement a set of separate “behaviour” classes that implement this interface
- Associate a duck instance with a specific “behaviour” class
- The Duck classes won’t need to know any of the implementation details for their own behaviour

Implementing the Behaviours

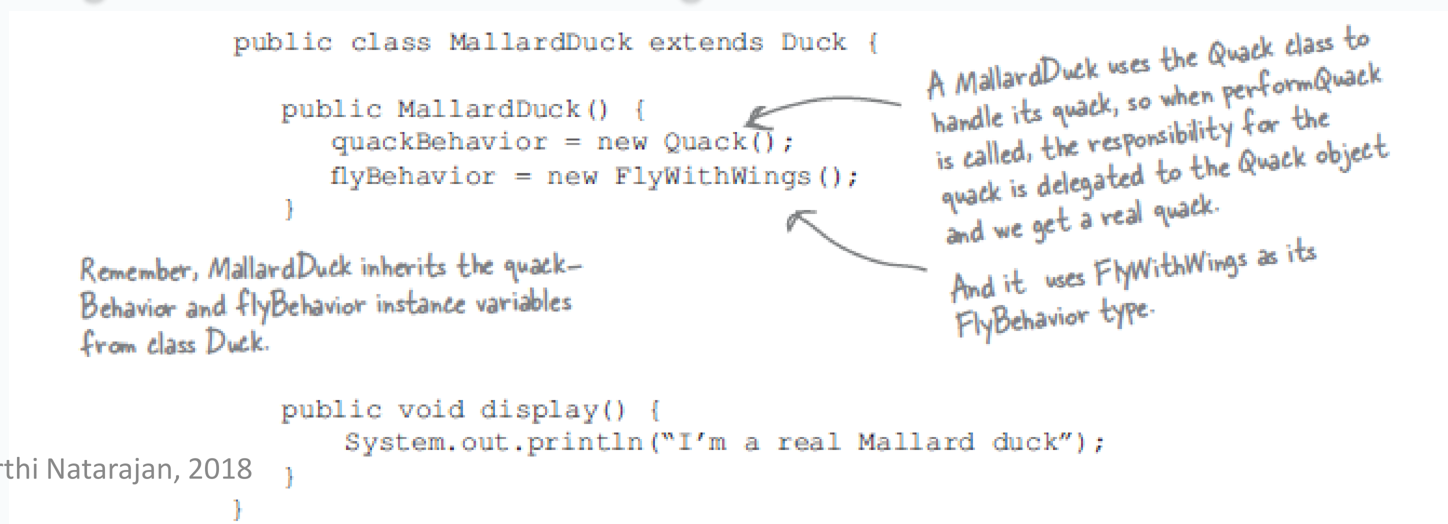
Here, there are two interfaces, FlyBehavior and QuackBehavior along with set of classes that implement each concrete behaviour



Integrating Duck behaviour with the Duck instance



1. Define two instance variables in the Duck class
2. Implement performQuack() and performFly() that delegate the quacking and flying behavior to other objects
3. Assign the instance variables the right behavior



Setting behaviour dynamically

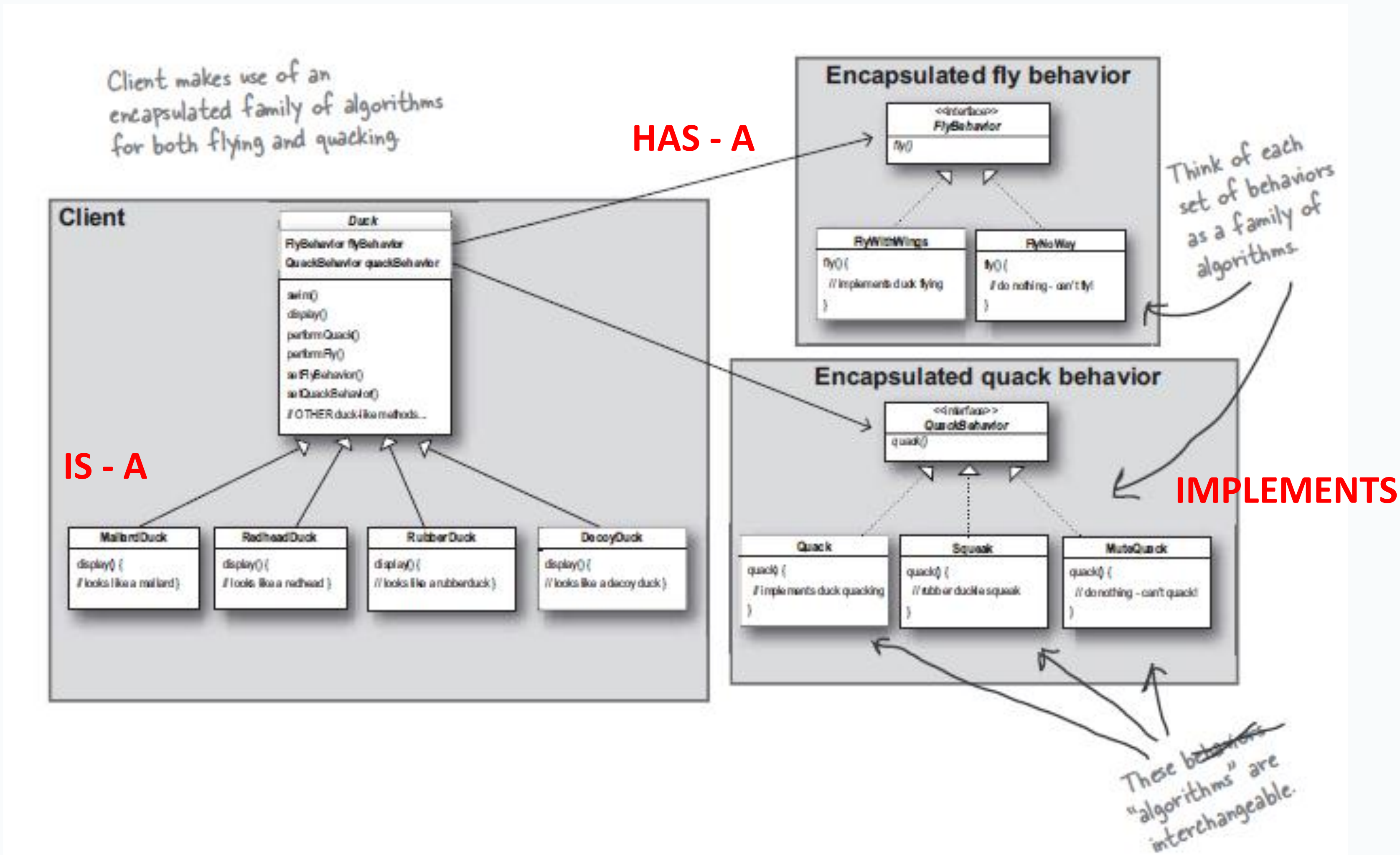
1. Create two setter methods `setFlyBehavior()` and `setQuackBehavior()` inside class Duck
2. To change a duck's behavior at run-time, call the duck's setter method for that behavior

```
public abstract class Duck {
...

 // Add setter methods to change behavior at run-time
  public void setFlyBehavior(FlyBehavior f) {
        this.flyBehavior =f ;
  }
  public void setQuackBehavior(QuackBehavior q) {
        this.quackBehavior = q;
  }
}
```

Our complete design

Think about the different relationships - IS-A, HAS-A, IMPLEMENTS



HAS-A can be better than IS-A

- Each duck has a fly behaviour and has a quack behaviour. Haven't we heard of this relationship?

COMPOSITION

- Instead of inheriting their behaviour, the ducks get their behaviour by being **composed** with the right behaviour objects and **delegate** to the behaviour objects
- This allows you to encapsulate a family of algorithms
- Enables you to "change behaviour" at run-time

Design Principle #5:

Favour composition over inheritance

Our first design pattern

- We have just applied our first design pattern to design our Duck app

STRATEGY PATTERN

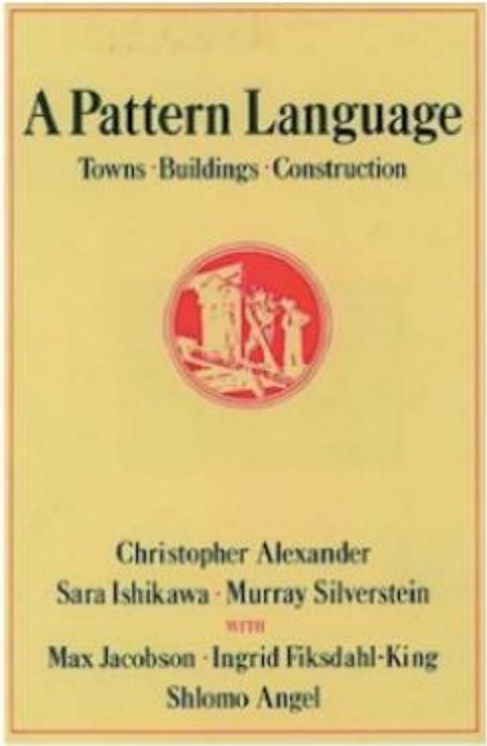
- This allows you to encapsulate a family of algorithms
- Enables you to "change behaviour" at run-time

Design Pattern #1: Strategy Pattern

This pattern defines a family of algorithms, encapsulates each one

Design pattern

- A design pattern is a tried solution to a commonly recurring problem
- Original use comes from a set of 250 patterns formulated by Christopher Alexander et al for architectural (building) design
- Every pattern has
  - A short name
  - A description of the context
  - A description of the problem
  - A prescription for a solution



Design pattern

- In software engineering, a design pattern is a general repeatable solution to a commonly occurring problem in software design
- A design pattern is
  - Represents a template for how to solve a problem
  - Captures design expertise and enables this knowledge to be transferred and reused
  - Provide shared vocabularies, improve communications and eases implementation
  - Is not a finished solution, they give you general solutions to design problems

How to use Design Patterns?

Using Design Patterns is essentially an "art & craft"

- Have a good working knowledge of patterns
- Understand the problems they can solve
- Recognise when a problem is solvable by a pattern

Design Patterns Categories

- Behavioural Patterns
- Structural Patterns
- Creational Patterns

Pattern #1: Strategy Pattern

- Motivation
  - Need a way to adapt the behaviour of an algorithm at runtime
- Intent
  - Define a family of algorithms, encapsulate each one, and make them interchangeable
  - Strategy pattern is a behavioural design pattern that lets the algorithm vary independently from the context class using it

Strategy Pattern: Implementation

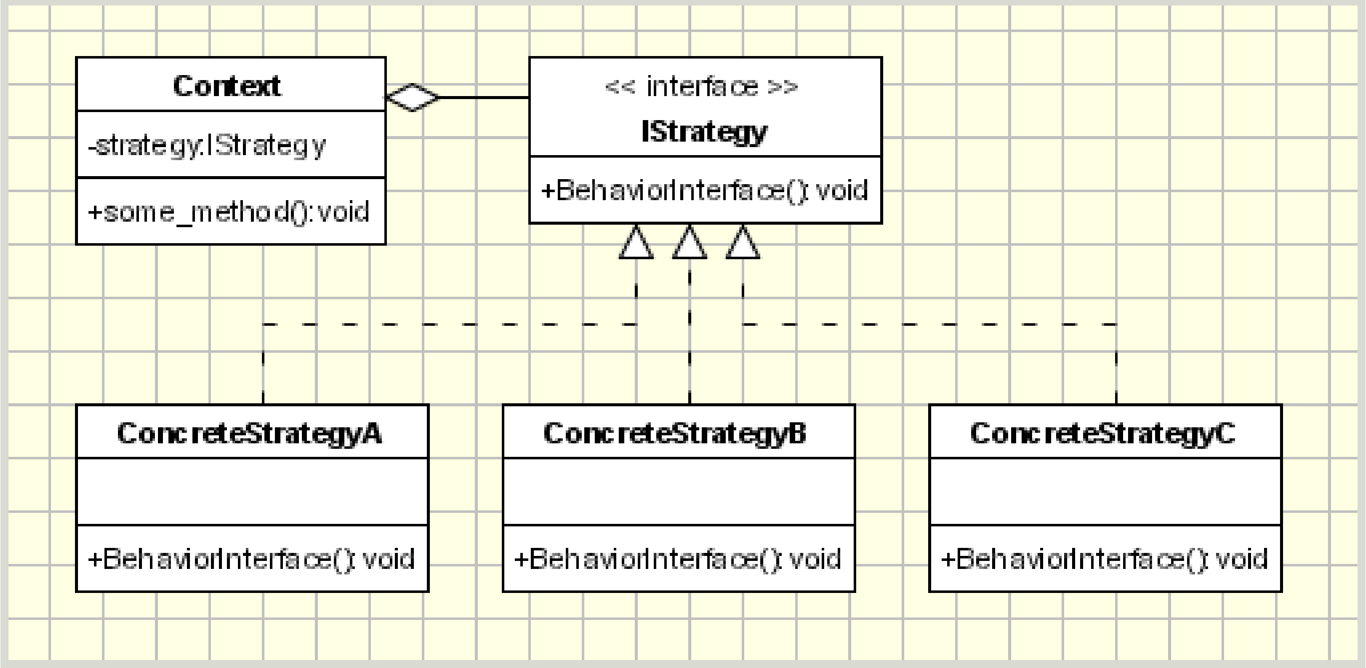


Image http://www.oodesign.com/

Strategy Pattern: Uses, Benefits, Liabilities

- Applicability
  - Many related classes differ in their behaviour
  - A context class can benefit from different variants of an algorithm
  - A class defines many behaviours, and these appears as multiple conditional statements (e.g., if or switch). Instead, move each conditional branch into their own concrete strategy class
- Benefits
  - Uses composition over inheritance which allows better decoupling between the behaviour and context class that uses the behaviour
- Drawbacks
  - Increases the number of objects
  - Client must be aware of different strategies

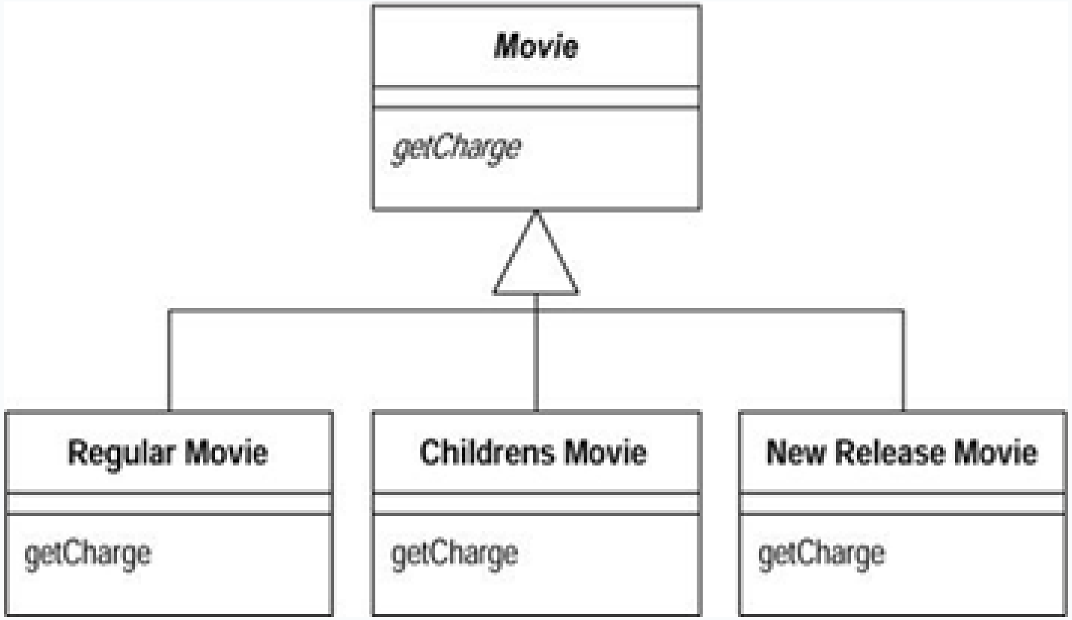
Strategy Pattern: Examples

- Sorting a list (quicksort, bubble sort, merge-sort)
  - Encapsulate each sort algorithm into a concrete strategy class
  - Context class decides at run-time, which sorting behaviour is needed
- Search (binary search, DFS, BFS, A*)

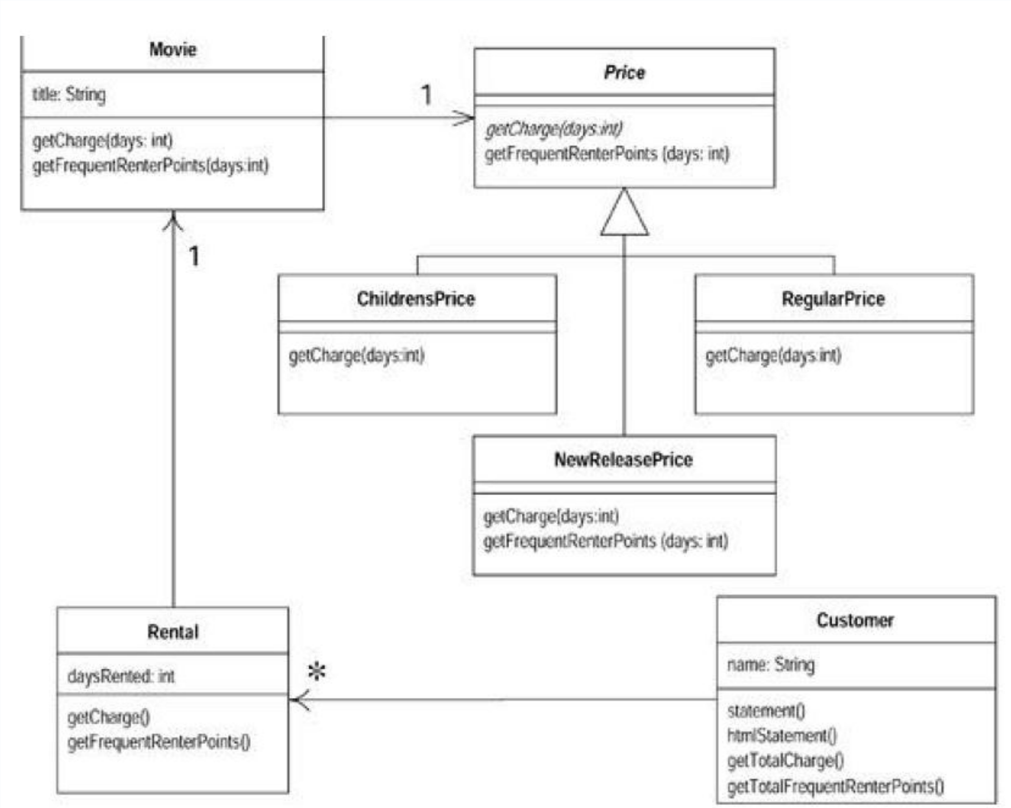
Next,

- State Pattern
- Revisit our video rental example

Recall our Video Rental Example from Week 03



- A movie can change its classification during its life-time, hence the price of the movie would vary
- The design above is not right, for the same reason we cannot have `fly()` inside the Duck class



- Remember our design principles
  - encapsulate what varies
  - compose and delegate
- Refactoring Techniques that support these principles
  - Replace Type Code with Strategy/State Pattern
  - Replace conditional logic with polymorphism

Summary

- Knowing OO basics does not make you a good OO designer
- Good OO designs are reusable, extensible and maintainable

OO Basics

- *Abstraction*
- *Encapsulation*
- *Inheritance*
- *Polymorphism*

OO Principles

- *Principle of least knowledge – talk only to your friends*
- *Encapsulate what varies*
- *Favour composition over inheritance*
- *Program to an interface, not an implementation*

OO Patterns

- *Strategy*
- *State*