Learning Objectives for Today

- Design Pattern Introdcution
- Behavioural Patterns 1

Design Patterns: What are they?

A Design Pattern:

- Describes software solutions and component structures that are common
- Defines software classes, roles and relationships
- Does not depend on any particular programming language
- Must be applied, tailored to create particular solutions

The benefits are many:

- The patterns **work**
- They facilitate **documentation**
- They enhance **communication**
- They promote **re-use**
- They limit **errors**

Classified in four main groups:

Behavioural, Creational, Structural, Concurrency

UNIVERSITY OF TORONTO MISSISSAUGA

Design Patterns and UML

A Design Pattern:

- Describes software solutions and component structures that are common
- Defines software classes, roles and relationships
- Does not depend on any particular programming language
- Must be applied, tailored to create particular solutions

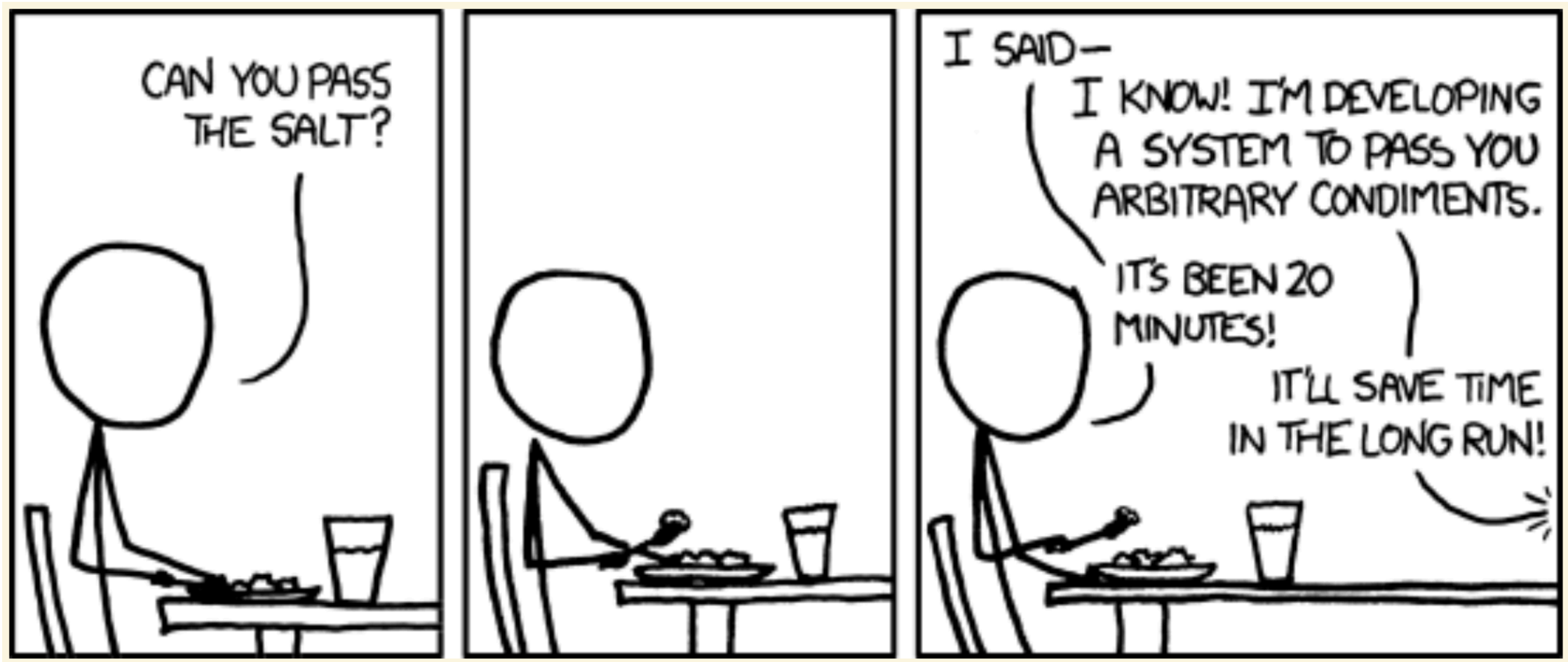
A Design Pattern is typically expressed with UML.

Design Patterns and UML

A Design Patterns can include:

- guidelines for when your code needs a given pattern
- typical application scenarios
- descriptions of the basic classes involved (in UML and text)
- descriptions of class relationships and collaborations (in UML and text)
- sample code
- consequences, known uses, related patterns, anti-patterns

Design Patterns: Why?



Design Patterns: Why?

The benefits are many:

- The patterns **work**
- They facilitate **documentation**
- They enhance **communication** and **teamwork**
- They promote **re-use**
- They limit **errors**

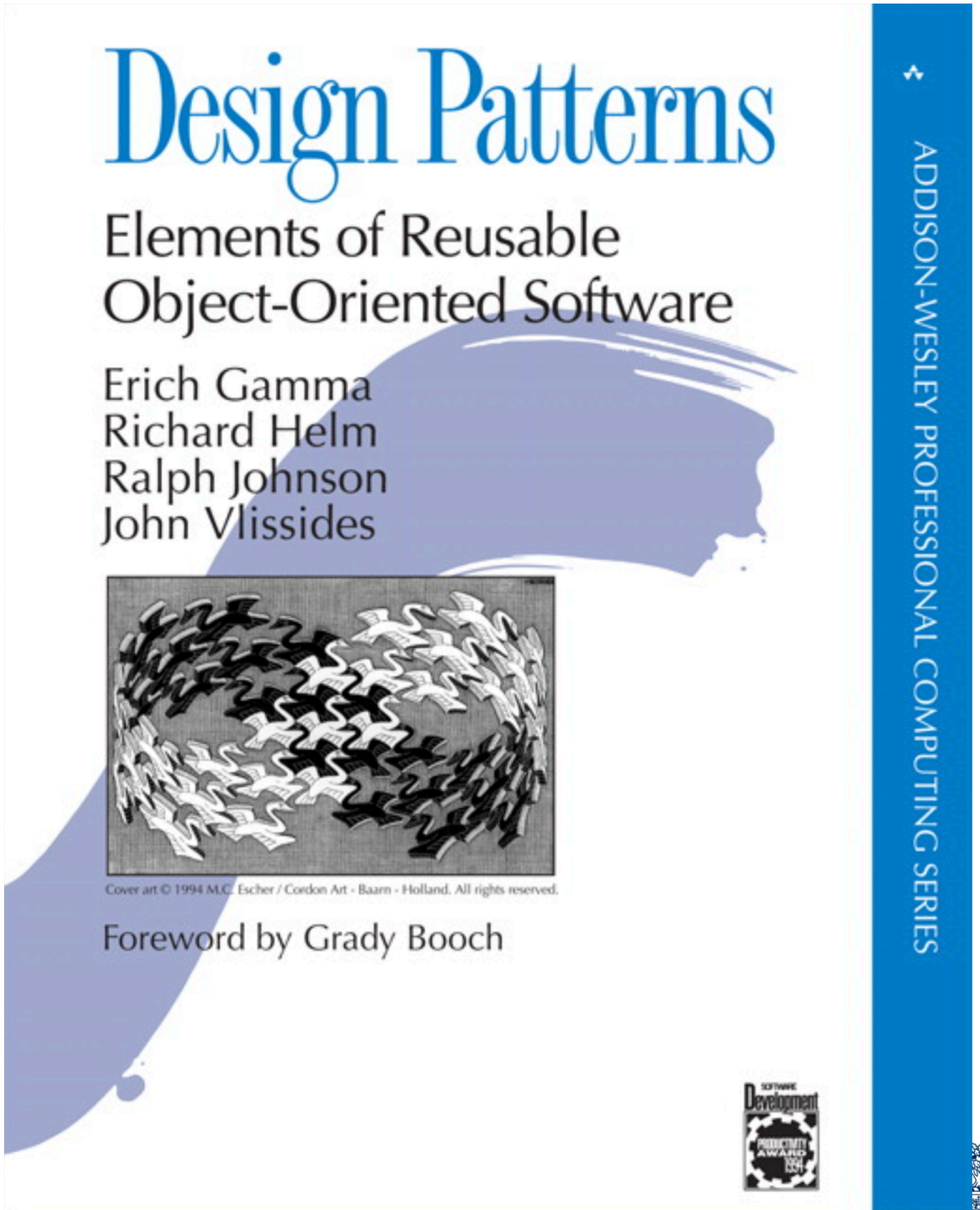
Design Patterns: Where?

First codified by the Gang of Four in 1995

- Book describes 23 patterns

There are of course many more.

We've assembled some that might be of interest to you for your group project on Quercus.



Design Patterns Types

Classified in four main groups:

Behavioural, Creational, Structural, Concurrency

We'll give some examples from each group as the class goes by.

Creational patterns provide structures and techniques for class and object creation.

Structural patterns explain how to assemble objects and classes into larger structures, while keeping these structures flexible and efficient.

Behavioural patterns detail how one class interacts with others.

Concurrency Patterns provide strategies for managing processes that may be happening simultaneously within a program, and that may depend on shared resources.

An Example of a Behavioural Pattern

The Iterator Pattern is a pattern that you've encountered to some degree in the past! The problem it solves:

We often have collections of objects we want to iterate over.

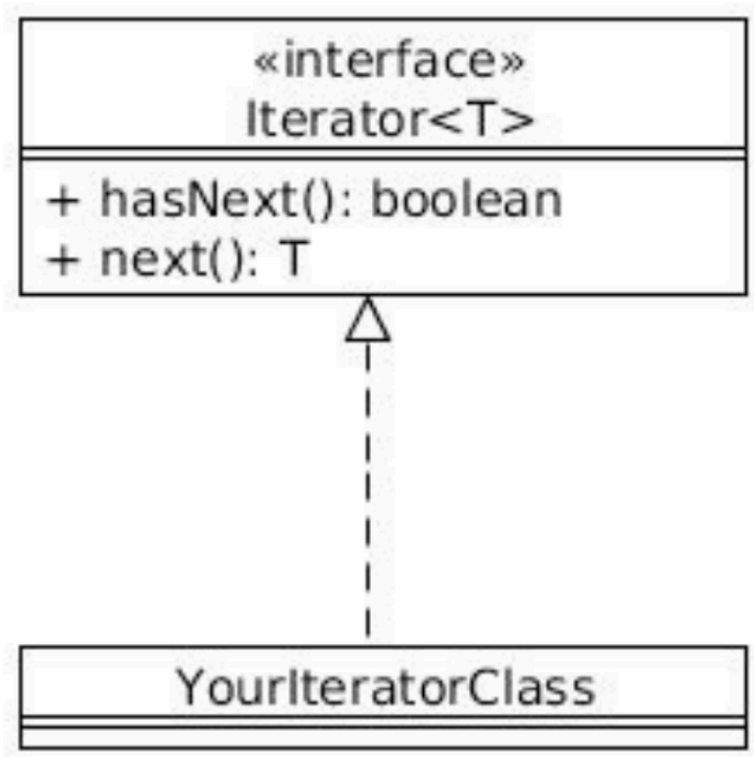
We may even want multiple iterators to operate over the same collection.

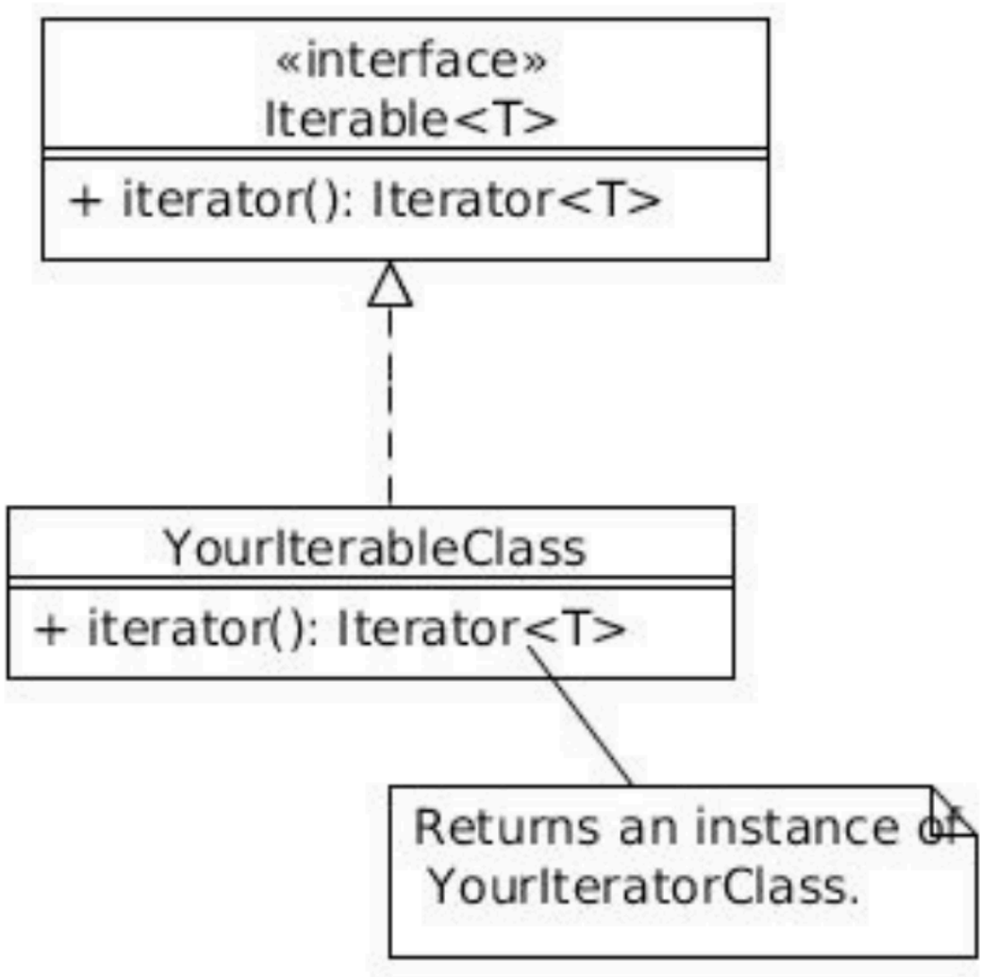
But as we iterate, we don't want to be bothered by details of the objects in the collection; we don't need to know how they are stored.

The iterator pattern is a **Behavioural** pattern, as it structures a **traversal behaviours** related to collections.

UNIVERSITY OF TORONTO MISSISSAUGA

The Iterator Pattern, in UML





Design Patterns: Observer Example

The Observer Pattern is also a **behavioural** pattern!

The observer pattern allows an object or a set of objects of **observe** another.

A real world example is any social media platform.

When a person says something, followers (or **observers**) of that person get a notification.

A follower can follow or unfollow another person at any point of time.

Design Patterns: Observer Example

Another application example: **Magazine Subscription.**

You subscribe to a magazine issued by a publisher. Every time there is a new issue, it gets delivered to you, for as long as you remain a subscriber.

You unsubscribe when you do not want the magazine anymore.

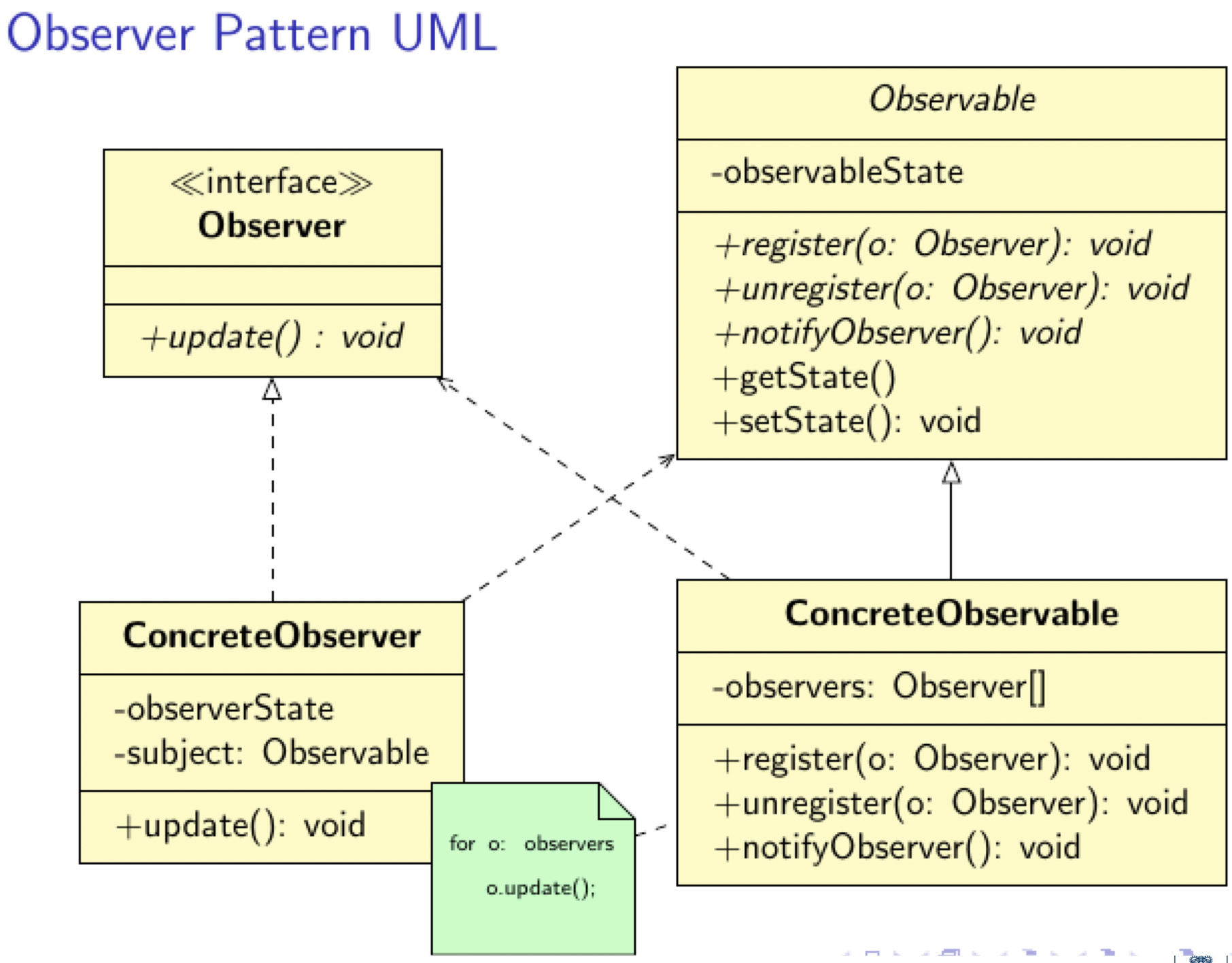
Other people also subscribe and unsubscribe to the same magazine.

Design Patterns: Observer Example

Some implementation considerations:

- Two categories of object are implied: the **observers** and the **observed**.
- This is kind of like what we saw with the iterator pattern, where we had **iterators** and **iterables**.
- With the observer pattern, we need to maintain consistency between related objects.
- An object should be able to **notify** other objects without making assumptions about what kind of objects they are.

Design Patterns: Observer Example



Design Patterns: Observer Example

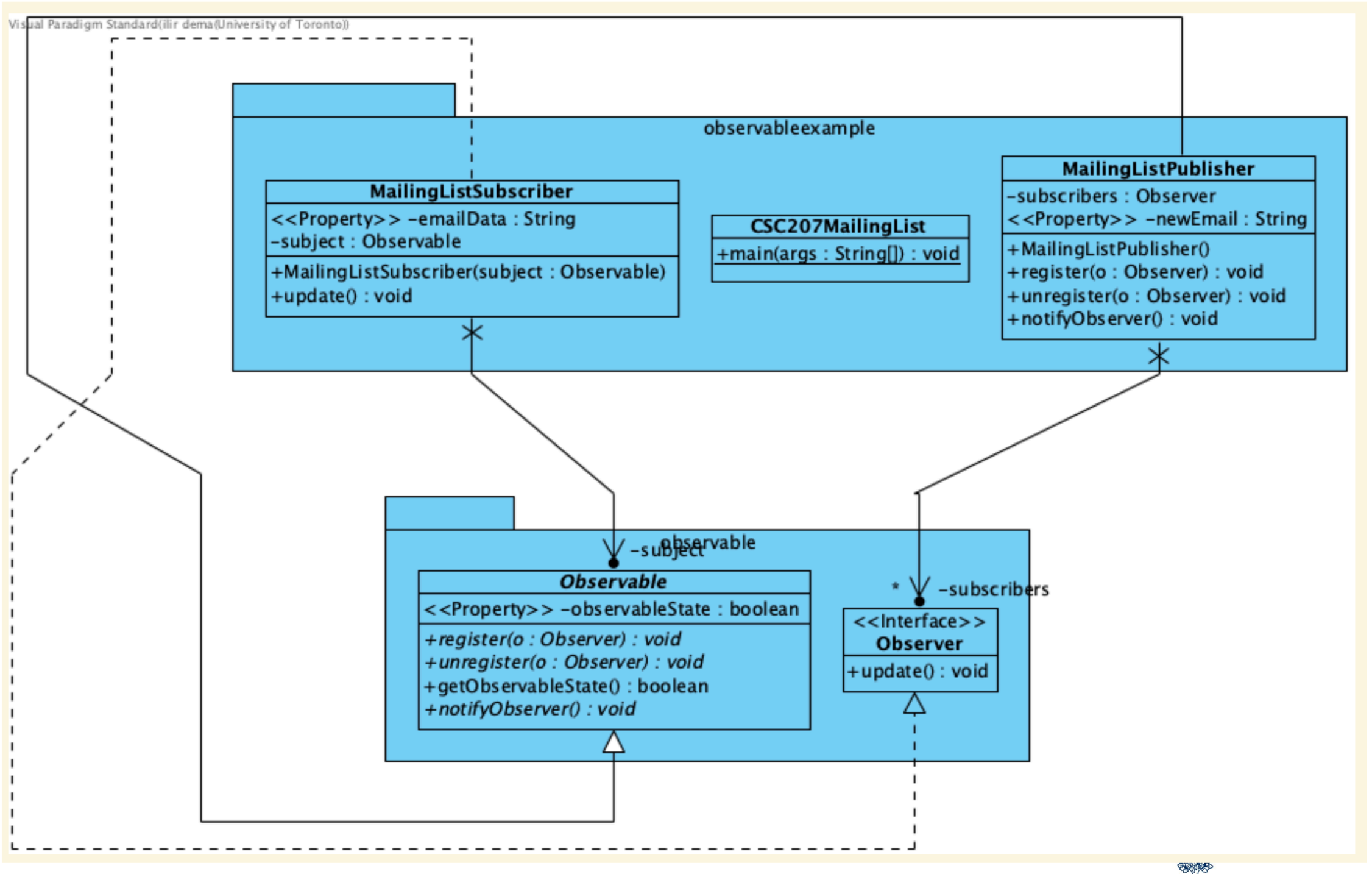
Each concrete observable contains an **aggregate of concrete observers**

- **Aggregate**: a structure of components
- Each **component** can exist on its own

The observable notifies observers **polymorphically**.

What makes the observer pattern well designed?

Design Patterns: Observer Example



Learning Objectives for Today

- More **Behavioural** Patterns
  - **Strategy**
  - **Command**

The Strategy Pattern in MVC

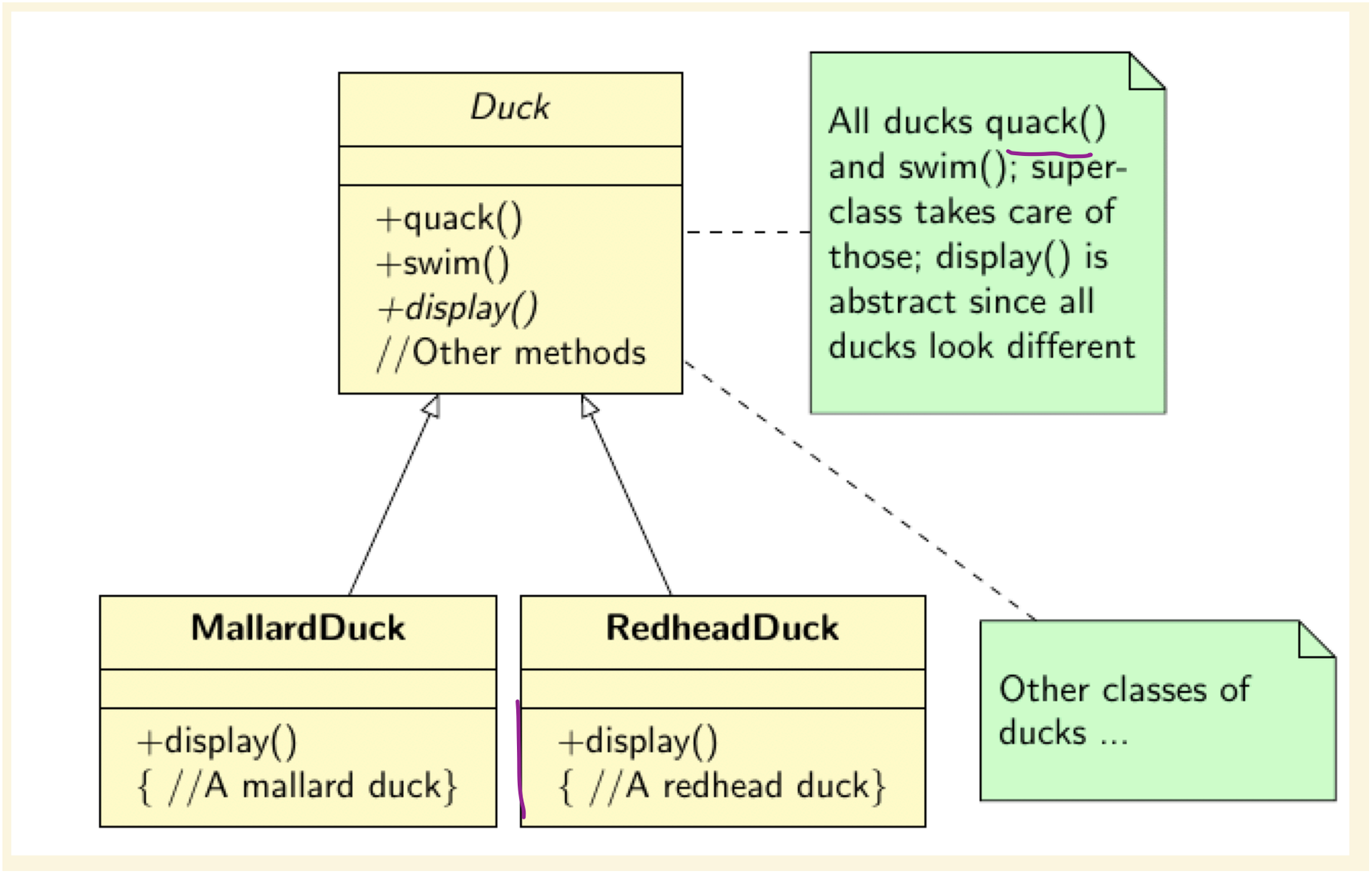
The Strategy Pattern can be used to change the way an object behaves while an application is running.

For example, in a game you might have a farmer character and a warrior character that you represent as Objects. Each may have a different movement or fighting strategy.

In general, the **Strategy Pattern** is useful when you want to **replace behaviours of an object** either statically or dynamically or when you have a lot of **variants in object behaviour**.

The Strategy Pattern: Example

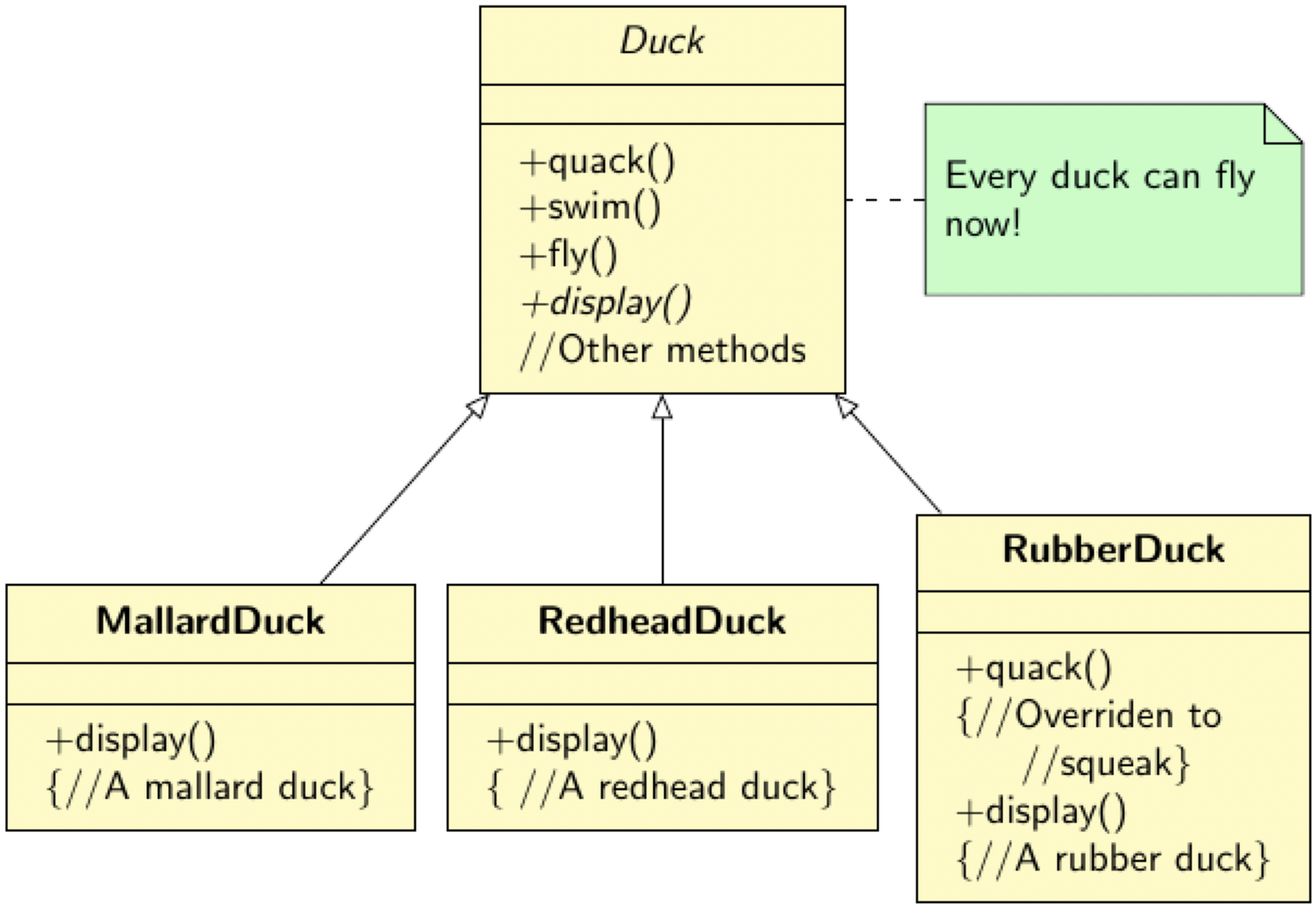
Assume we are creating an application to model ducks.



The Strategy Pattern: Example

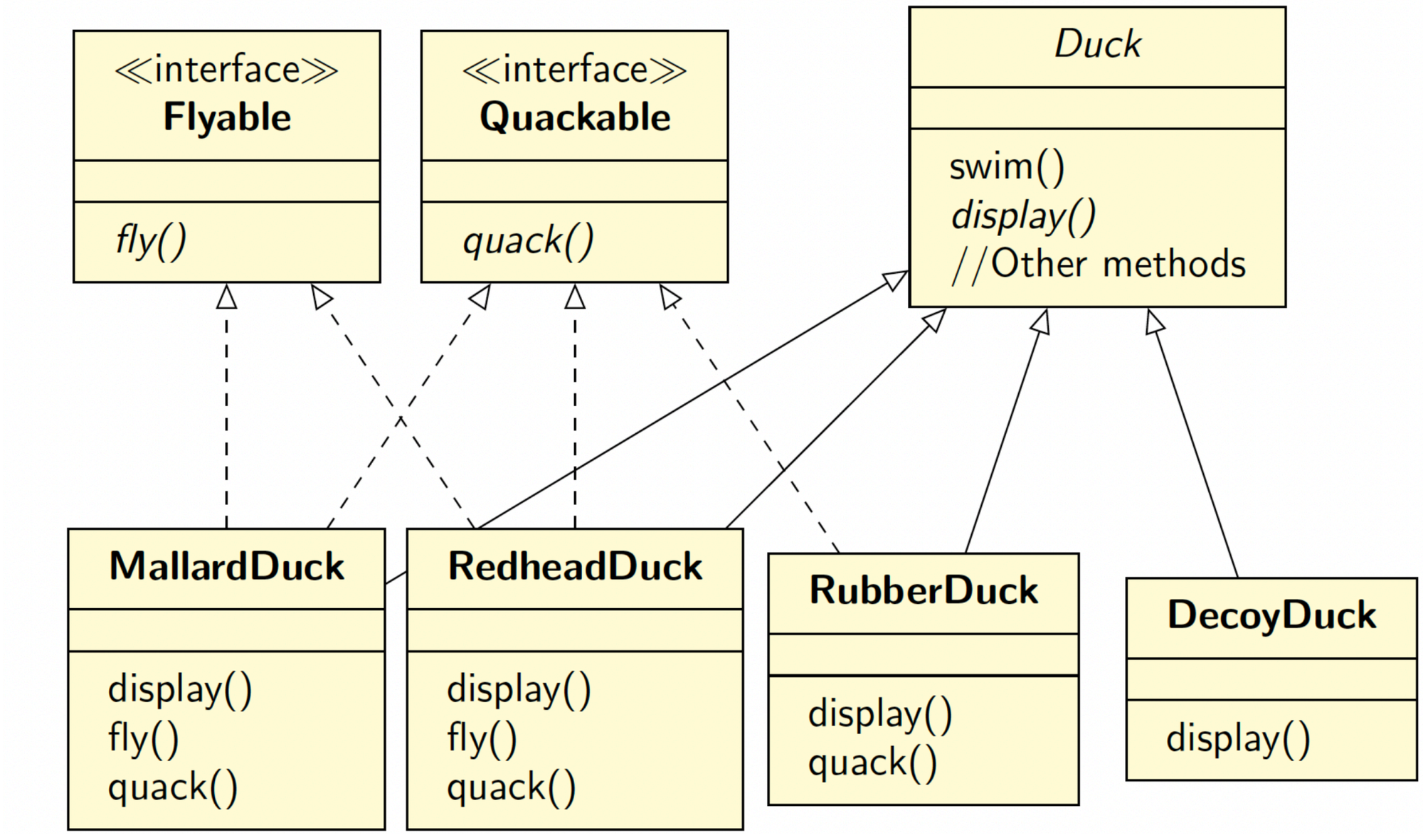
We want to modify our application to allow ducks to fly.

What happens if we net add a DecoyDuck, which neither flies nor quacks? Can you identify problems with design, and/or suggest a fix?



The Strategy Pattern: Example

We could try using **interfaces**. Is this a better design?



The Strategy Pattern: Example

The use of interfaces solves part of the problem (no flying rubber ducks), but reuse of code is limited, which creates a maintenance problem.

This can be addressed by using a design principle:

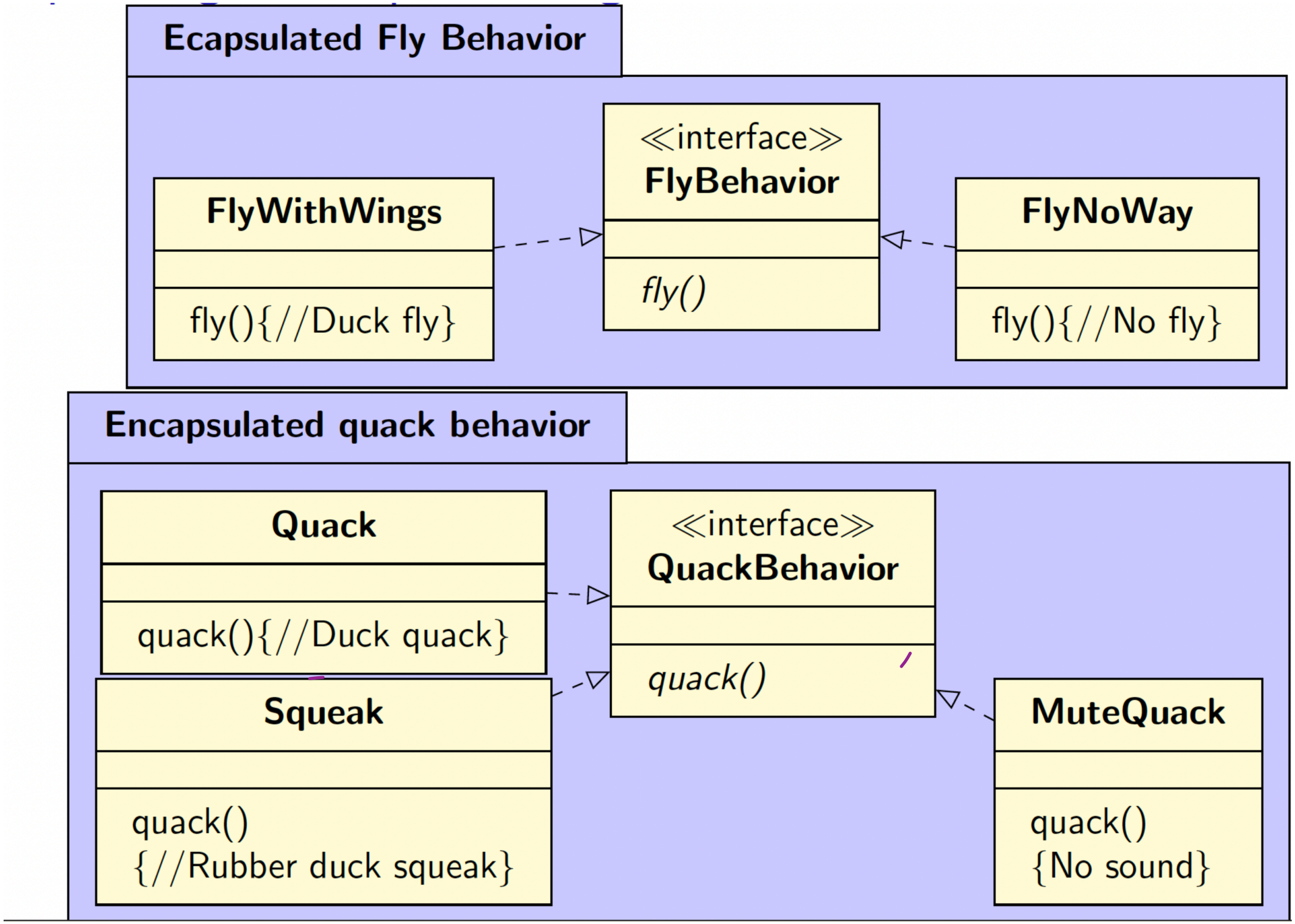
- Identify the aspects of your application that **vary** and separate them from **what stays the same**.
- You can **encapsulate** the parts that **vary**, so you can modify or extend them without affecting those parts that **stays the same.**

The Strategy Pattern: Example

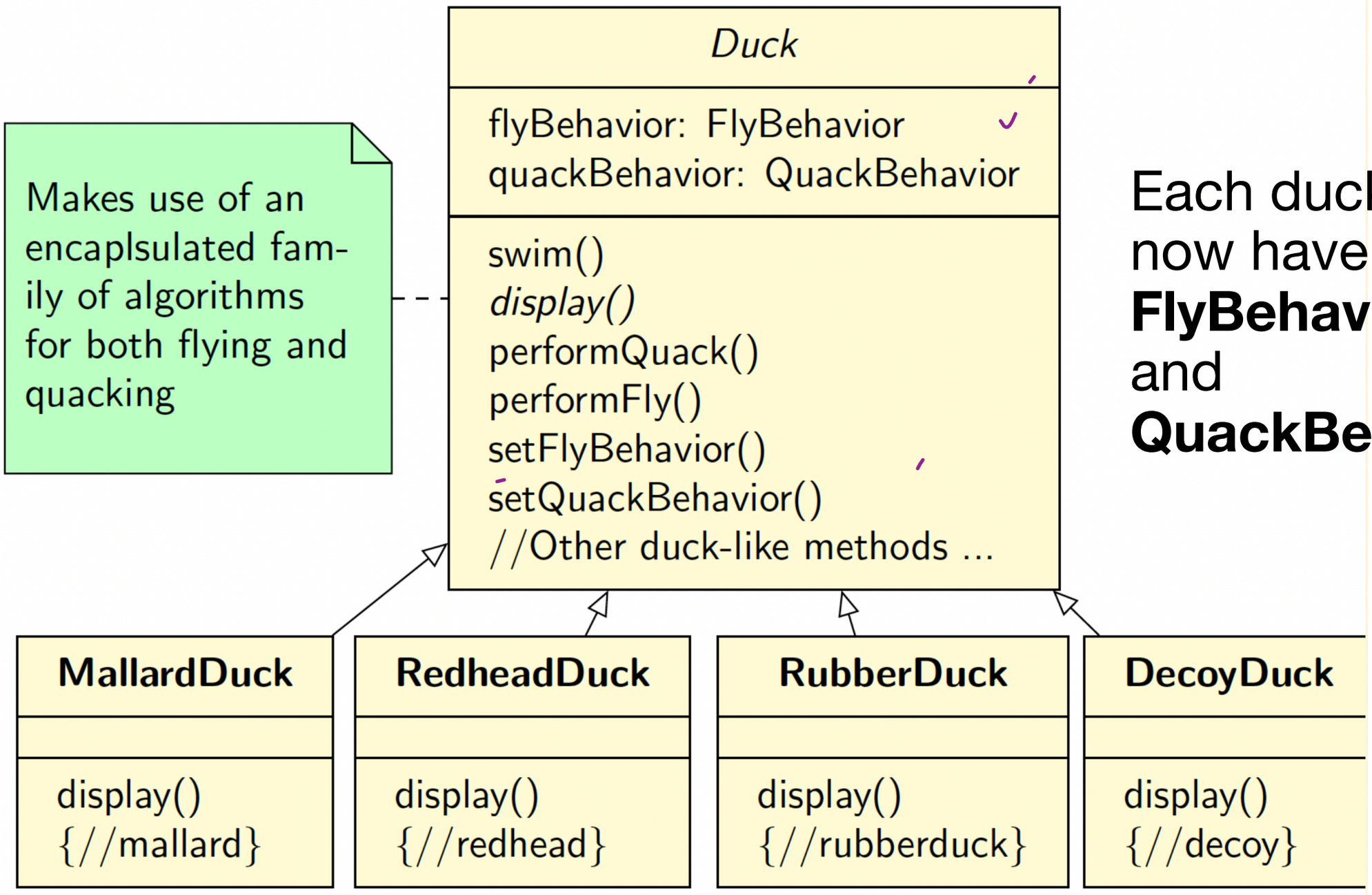
How this relates to our ducks:

- The parts of Duck that **vary** are **fly()** and **quack(),** so we will pull them out of Duck class.
- Create a new set of classes to represent the variable behaviours.

The Strategy Pattern: Example



The Strategy Pattern: Example



Each duck can now have a **FlyBehaviour** and **QuackBehaviour**

The Strategy Pattern: Summary

The **Strategy Pattern** defines families of functionality, encapsulates each one, and makes them interchangeable.

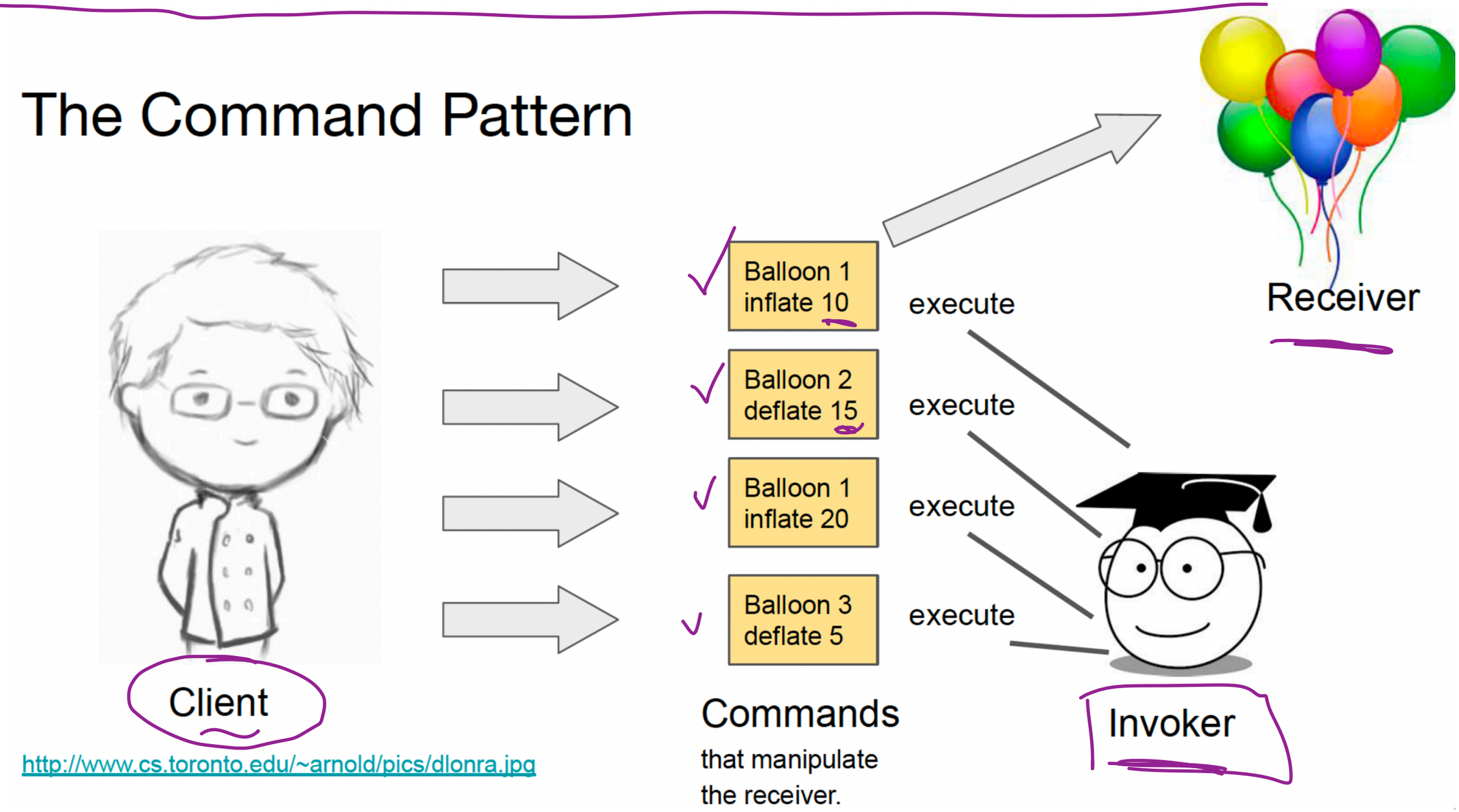
Benefits:

- Provides a substitute to subclassing.
- Defines each behaviour within its own class.
- Makes it easier to extend and incorporate new behaviours without changing the application.

Usage:

- When many classes differ only in their behaviour.
- When you need variations of an algorithm.

The Command Pattern: Example



The **client** issues commands, which are stored. The **invoker** invokes commands, in order.

The Command Pattern: Example

Goal is to send requests/commands to a receiver object, to make the receiver object perform various actions.

- e.g., send “inflate”, “deflate” commands to a Balloon object (receiver)

Plan involves encapsulating each request as an object, so we can store them in queue of requests, or undo past requests.

The Command Pattern in MVC

In an **MVC** design, this pattern might be used to facilitate **undo** and **redo** operations.

To make this work, we might wrap each command in an object and add them to a **history list** when the command is executed.

To **undo** or **redo**, we would move up or down the **history list.**

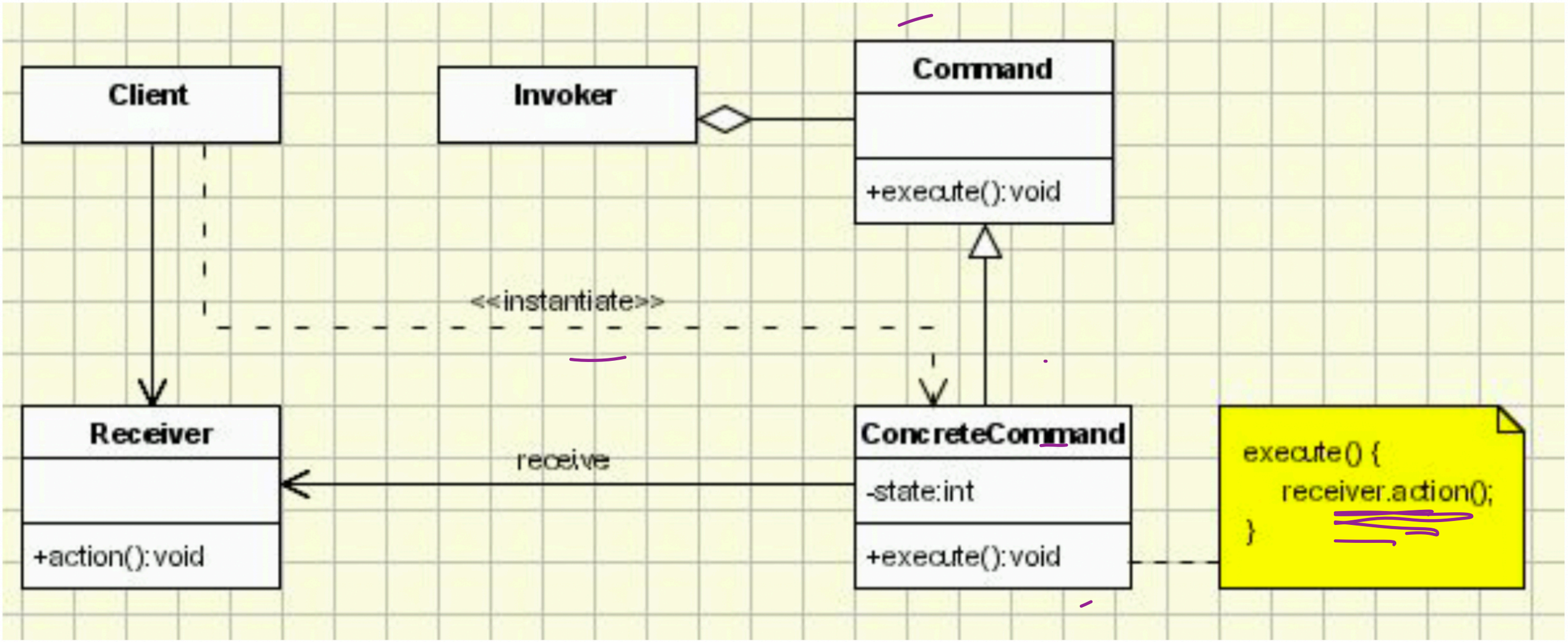
The Command Pattern in MVC

All **requests/commands** are unified by a **common interface** for execution, and can be stored in the same **collection**.

The **client** (who sends the commands) and the **receiver** (who knows how to perform the command) are decoupled.

The client programmer doesn’t even need to read the JavaDoc of the receiver object. The client just needs to know how to construct a command object, and place it in the collection of commands.

The Command Pattern: UML



The Command Pattern: Example

```
Light light = new Light();

RemoteControl control = new RemoteControl();

Command lightsOn = new TurnOnCommand(light);
Command lightsOff = new TurnOffCommand(light);

// switch on
control.setCommand(lightsOn);
control.pressButton();

// switch off
control.setCommand(lightsOff);
control.pressButton();
```

The Command Pattern: Implementation

Step 1: Create a common **interface** for all **commands** for a given receiver object.

This interface should include an **execute()** method.

Each **command** object will need a reference for the **receiver** object in order to make execute() do its thing.

The Command Pattern: Implementation

Step 2: Implement the interface within each class of **concrete command,** i.e., implement the *execute()* method.

This will invoke the action methods of the **receiver** class.

Step 3: Instantiate a **receiver** object and **concrete commands.**

Utilize an **invoker** (some collection) to store the commands.

Execute each command with: *command.execute()*