Strategy Design Pattern
Part A

Fani Deligianni

fani.Deligianni@glasgow.ac.uk

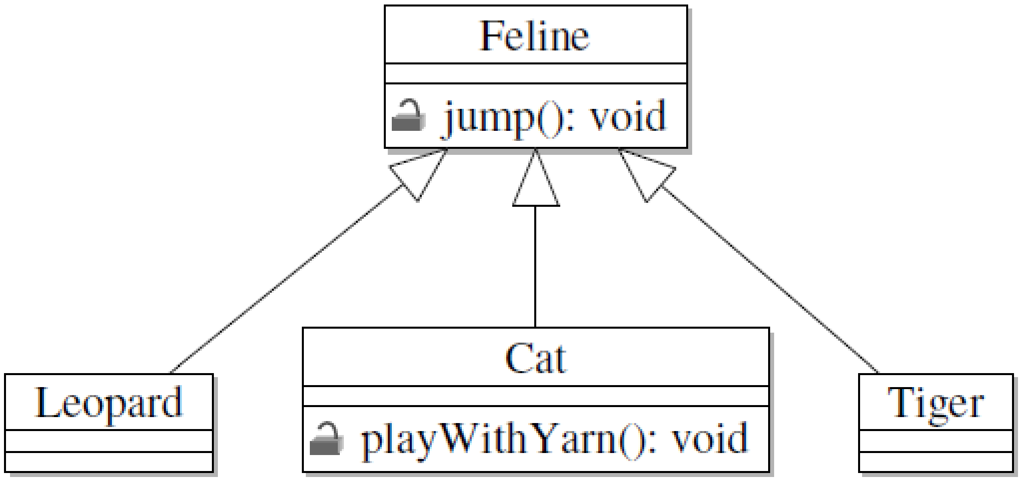
https://www.gla.ac.uk/schools/computing/staff/fanideligianni/

Learning Outcomes

- The problem with using inheritance
- Know when to favour composition over inheritance
- Understand how to program to interfaces and not implementation
- Understand different design principles

The problem with inheritance and interfaces

- Inheritance is one of the core principles of object oriented programming and design.
- You think of one class inheriting from the other if they share an **is-a** relationship (if a cat is a feline then a cat should inherit from the feline class)



The problem with inheritance and interfaces

- There are many designs where inheritance is exactly the right choice. But it's also easy to overuse inheritance, and make it the basis for all object-orient design.
  - Pay attention when most class relationships are becoming is-a relationships.
  - When inheritance is overused, you end up with a design and code that is inflexible and not amendable to change.

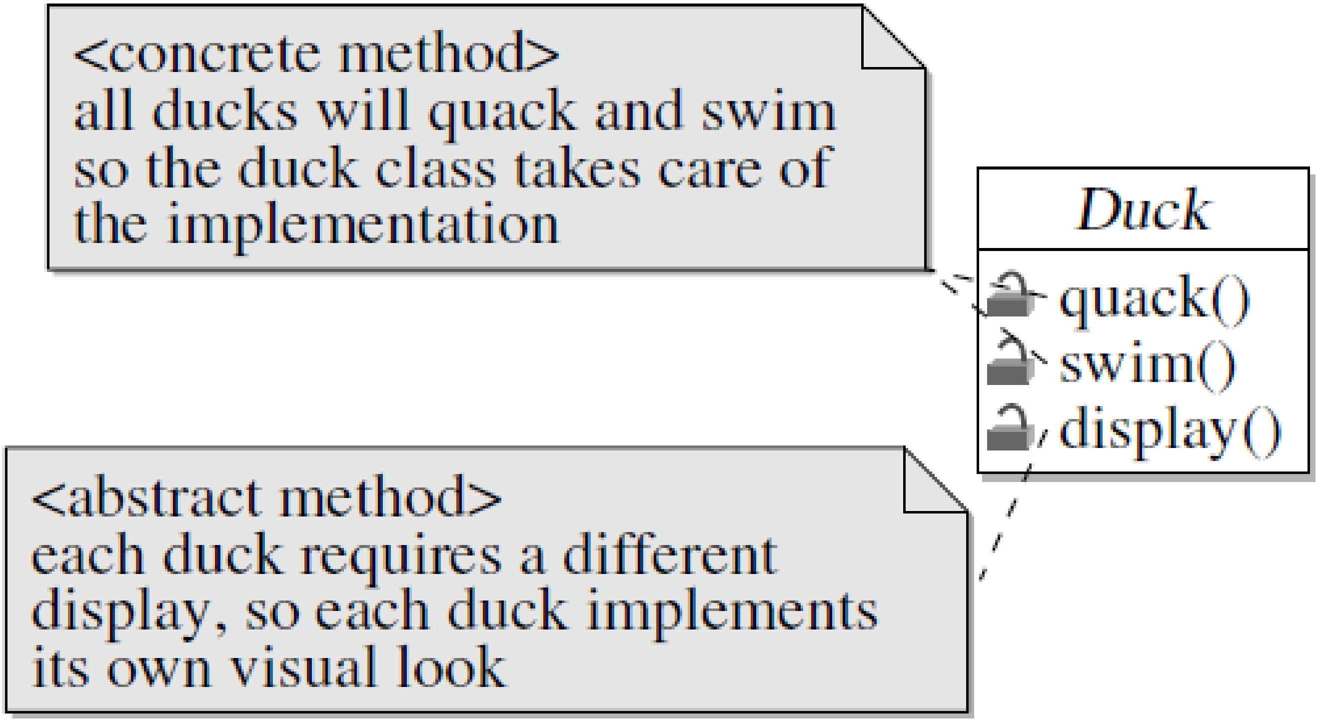
Example: Simple SimUDuck App

Joe works or a company that makes duck pond simulation game, SimUDuck

- The game can show large variety of duck species swimming and making quacking sounds.
- Initial designers of the system created one Duck superclass from which all other duck types inherit.

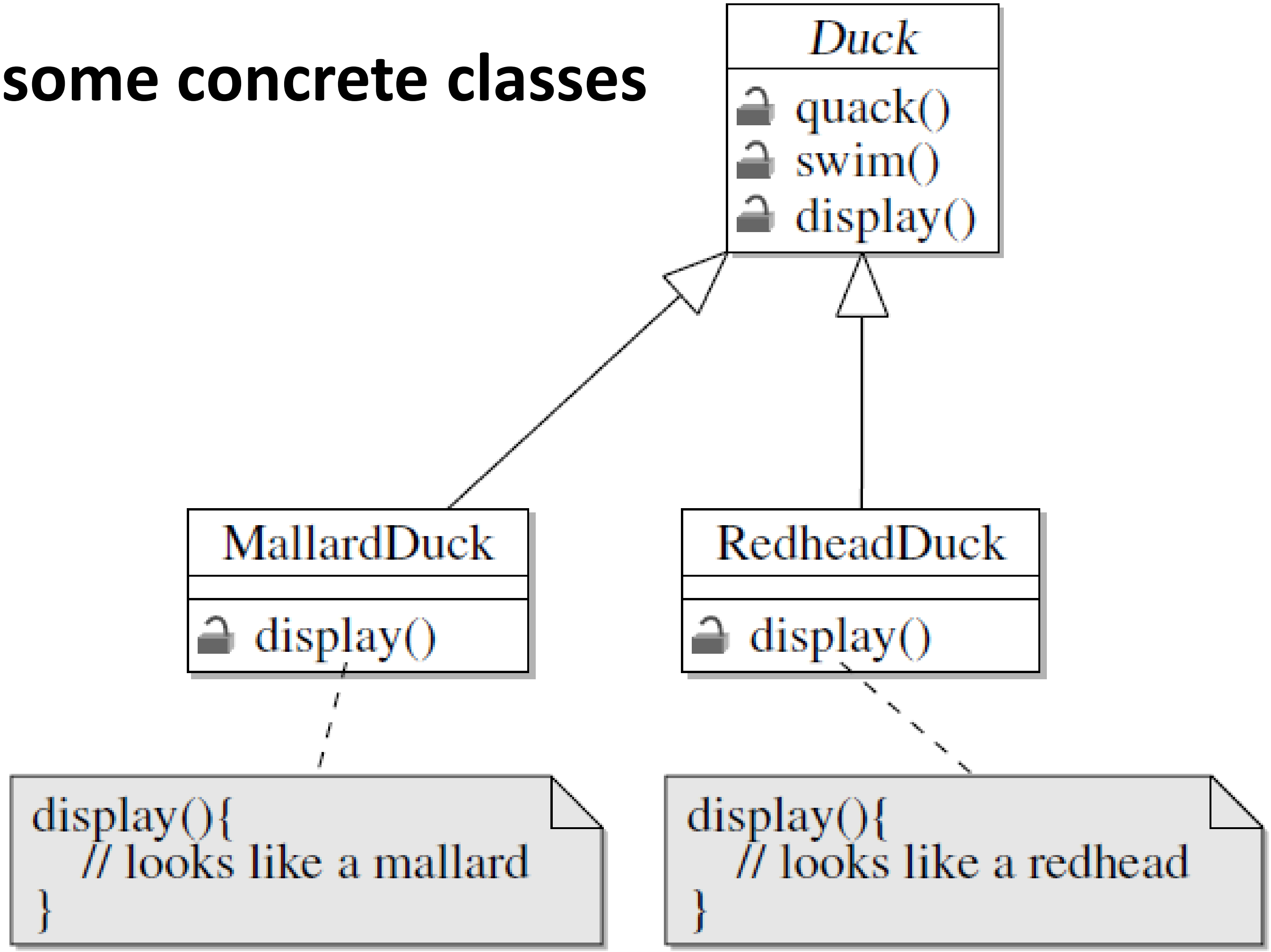
Duck simulator design - Using inheritance

Step 1: Start with a duck super class



Duck simulator design - Using inheritance

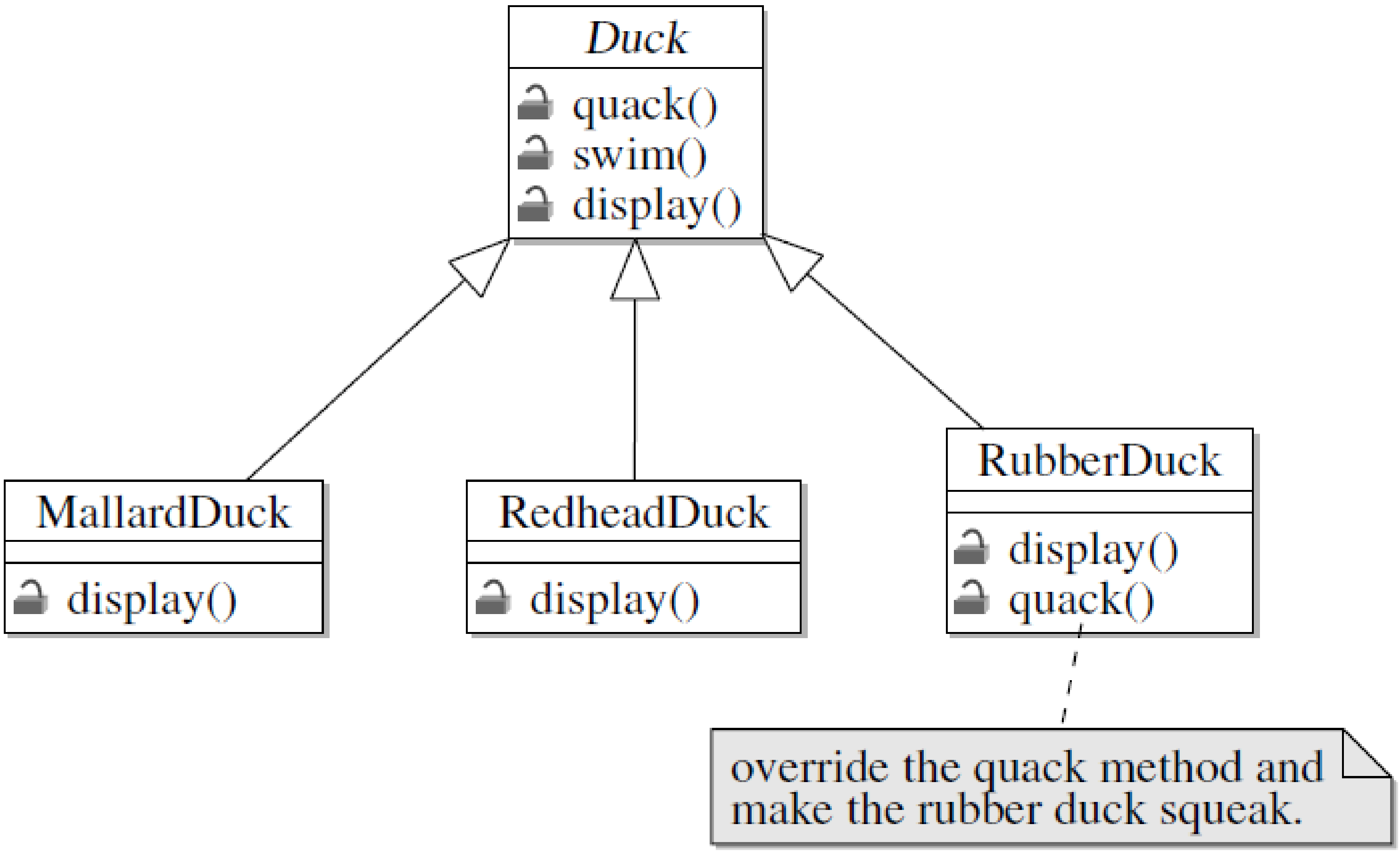
Step 2: Create some concrete classes



Duck simulator design - Using inheritance

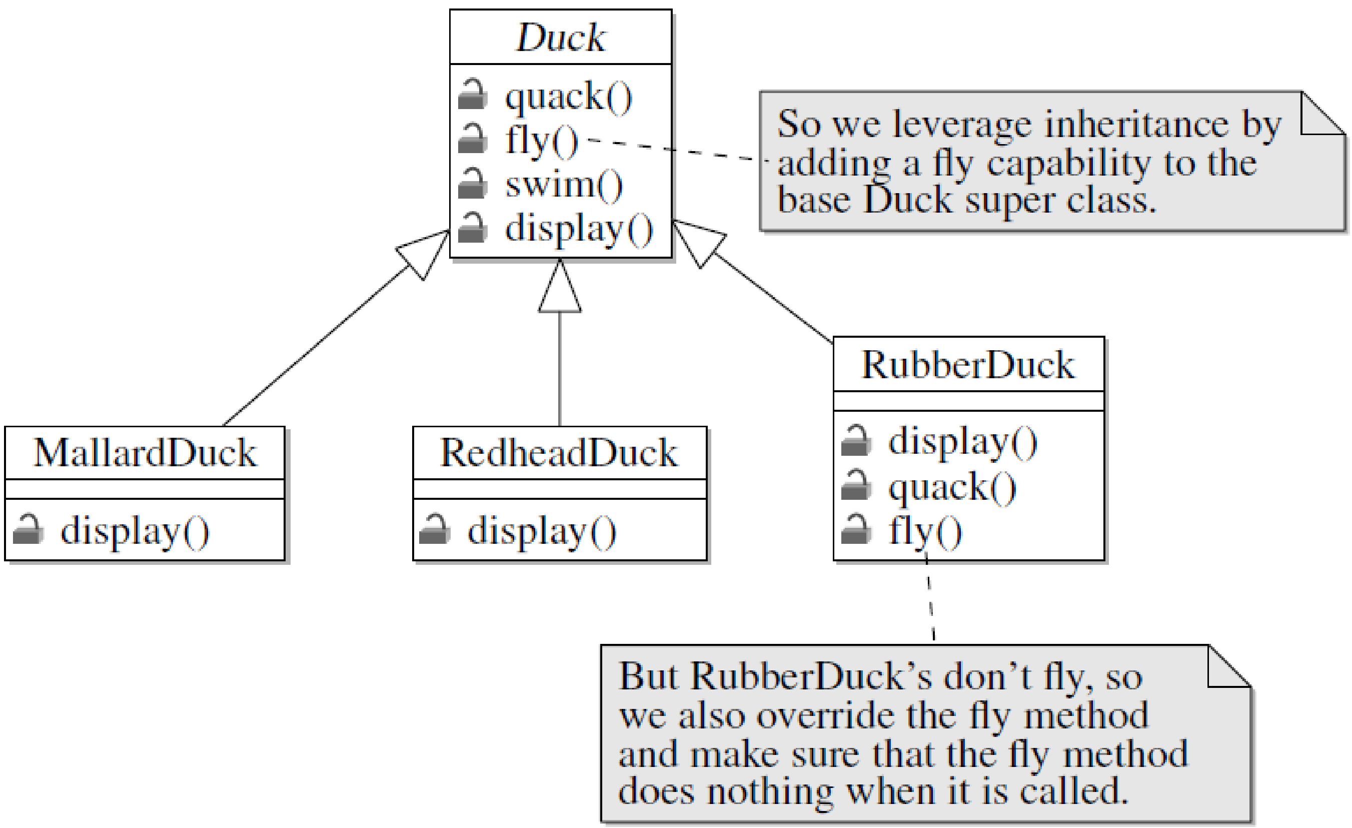
Step 3: We get a request to add another new duck type, a RubberDuck.

- But RubberDuck don’t quack, the squeak



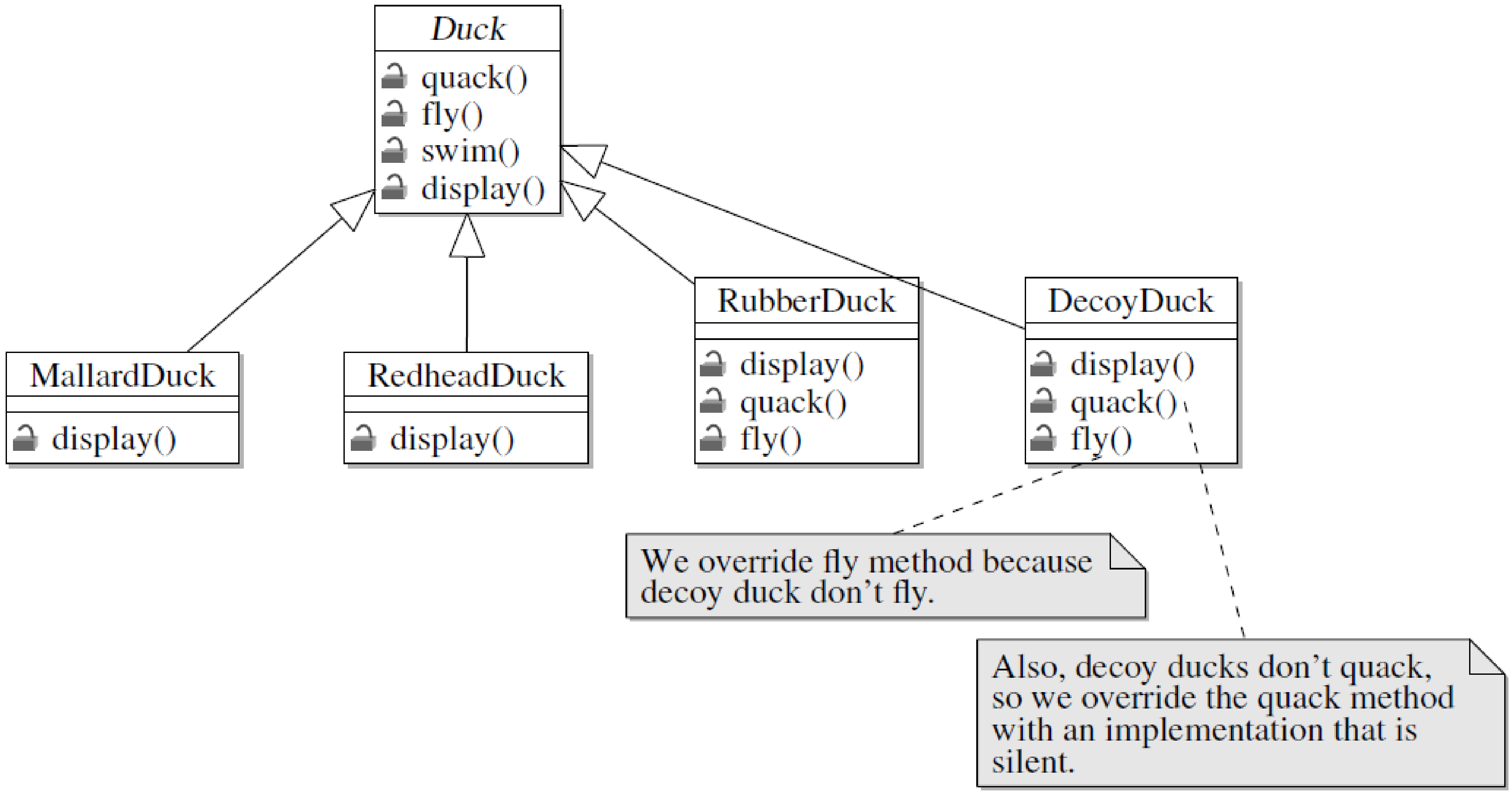
Duck simulator design - Using inheritance

Step 4: We get a feature request to make ducks fly



Duck simulator design - Using inheritance

Step 5: Ha! again we get a request to add another Duck, this time around a DecoyDuck.



Duck simulator design - Using inheritance

Problem: Because we are overriding most of the methods in the super class, we aren't getting lot of benefit from inheritance

1. We are loosing on reuse benefit of inheritance.
2. We start to get code duplication across classes
3. Hard to gain direct knowledge from the super class. Have to navigate each subclass to learn what the code does.
4. Simple changes to super class (e.g adding new feature) leads to unintended side effects on subclasses.
5. All behaviour is assigned at compile time.

Duck simulator design - Using inheritance

Thus far, the design does not give us much flexibility as the application becomes more sophisticated.

Strategy Design Pattern Part B

Fani Deligianni

fani.Deligianni@glasgow.ac.uk

https://www.gla.ac.uk/schools/computing/staff/fanideligianni/

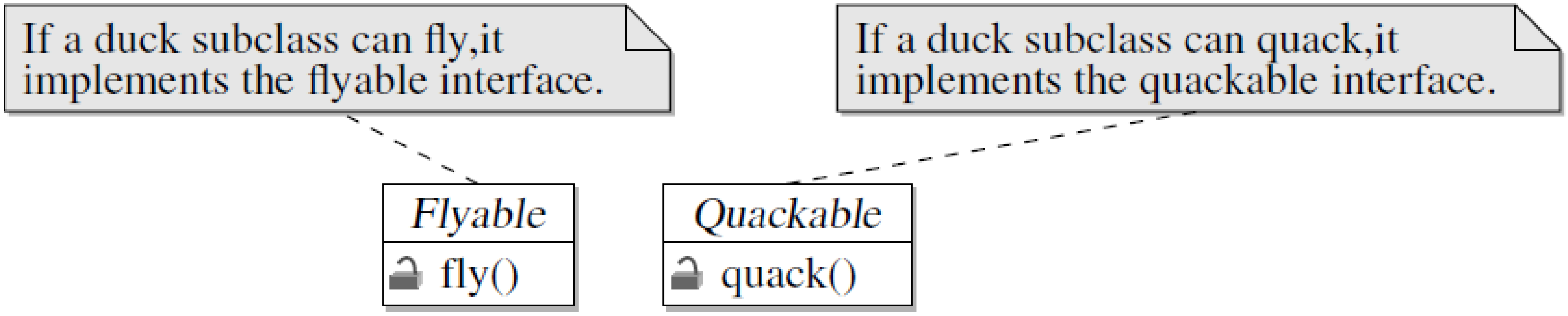
Duck simulator design - Using Interfaces

- Interfaces allow different classes to share similarities.
  - Example, ducks sharing a fly behavior.
- Also, interfaces allow for having two classes that are alike, but don’t have to have the same behavior
  - Example, like some ducks have fly behavior, and some not.

Duck simulator design - Using Interfaces

Step 1: Start by implementing two interfaces: Flyable and Quackable

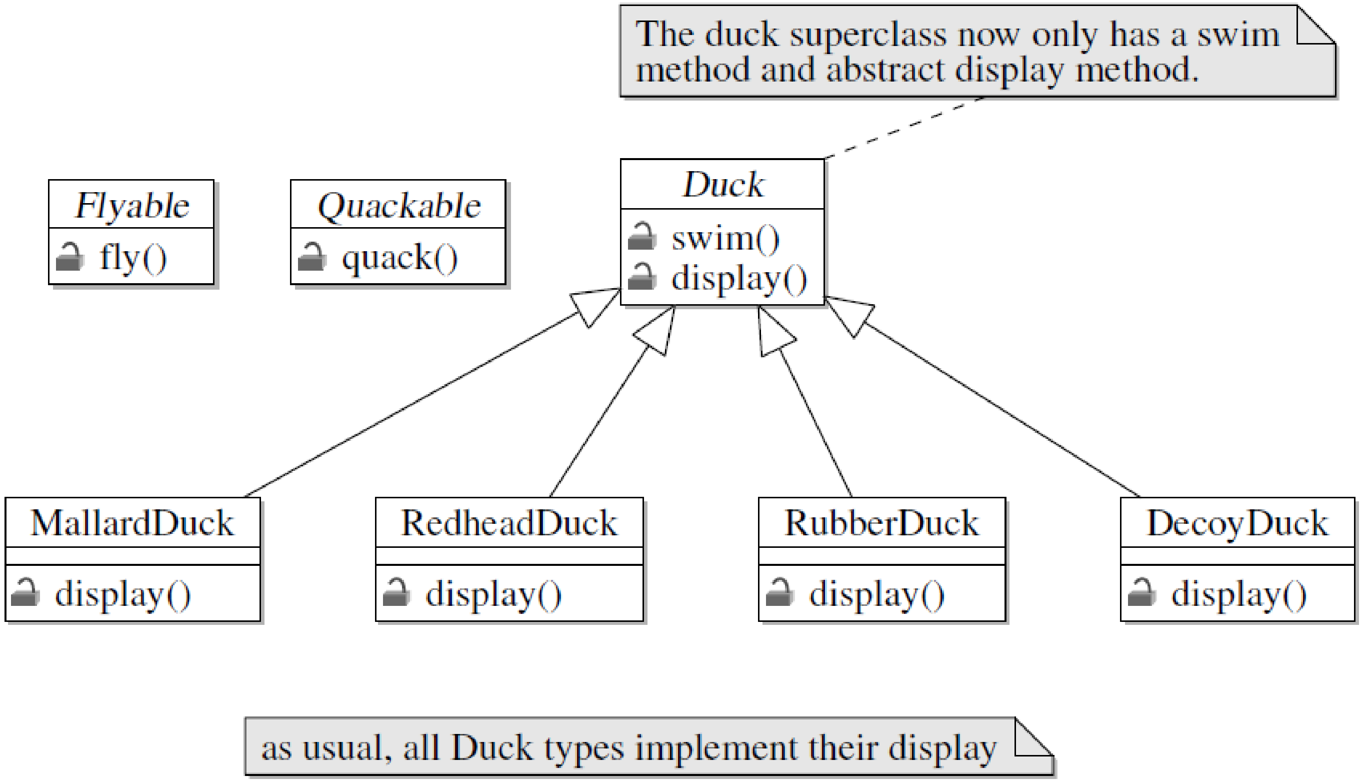
- Guide: identify methods in the super class that change frequently and move to an interface.



Duck simulator design - Using Interfaces

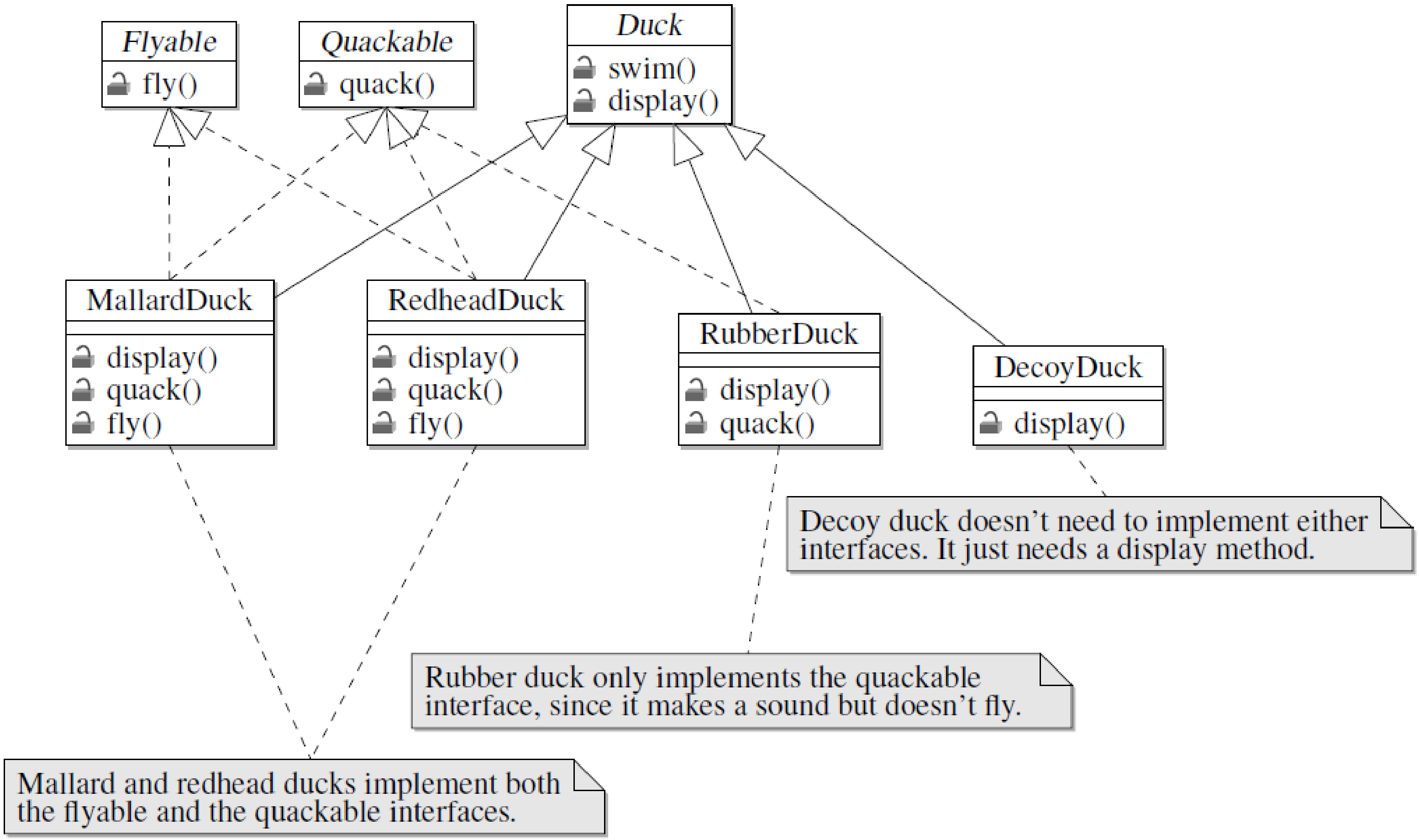
Step 2: Create Duck superclass and subclasses

- Guide: The duck superclass now just needs a swim method and abstract display method.



Duck simulator design - Using Interfaces

Step 3: Implement necessary interfaces



Duck simulator design - Using Interfaces

Problem: This design solves part of our issue but also introduces other problems

1. It destroys any possibility of code reuse.
   - Imagine having 40 ducks, every single duck will have to implement it's own fly and quack methods.
2. Any change needed to flying or quacking would cause a maintenance nightmare.
   - as we'll probably have to look at every concrete duck implementation and make changes.
3. Does not still allow for runtime changes in behaviours other than flying or quacking.

Duck simulator design - Rethink

- Inheritance has not worked well
  - Duck behavior keeps changing
  - Not suitable for all subclasses to have those properties
- Interface was at first promising, but
  - No code re-use
  - Tedious. Every time a behavior is changed, you must track down and change it in all the subclasses where it is defined.
  - error prone

OO Design Principles

Design Principle #1:

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

OO Design Principles

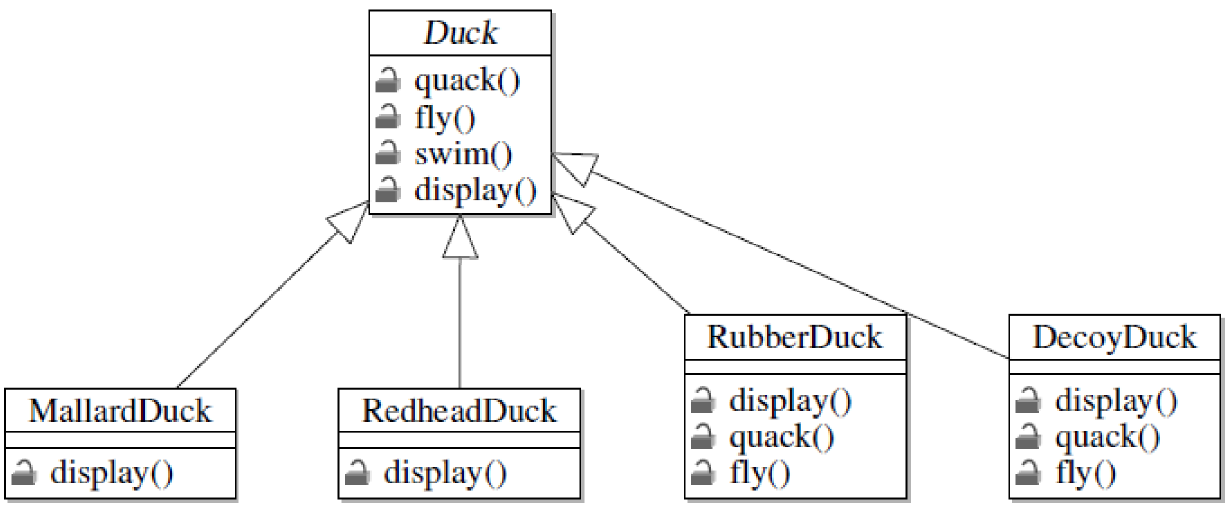
Encapsulate what varies

- If some aspect of code is changing, then it is a sign that the aspect should be pulled out and separated.
- By separating the aspects that vary, you can extend or alter then without affecting the rest of the code

This principle is fundamental to almost every design pattern.

Duck simulator design - Applying OO Design Principles

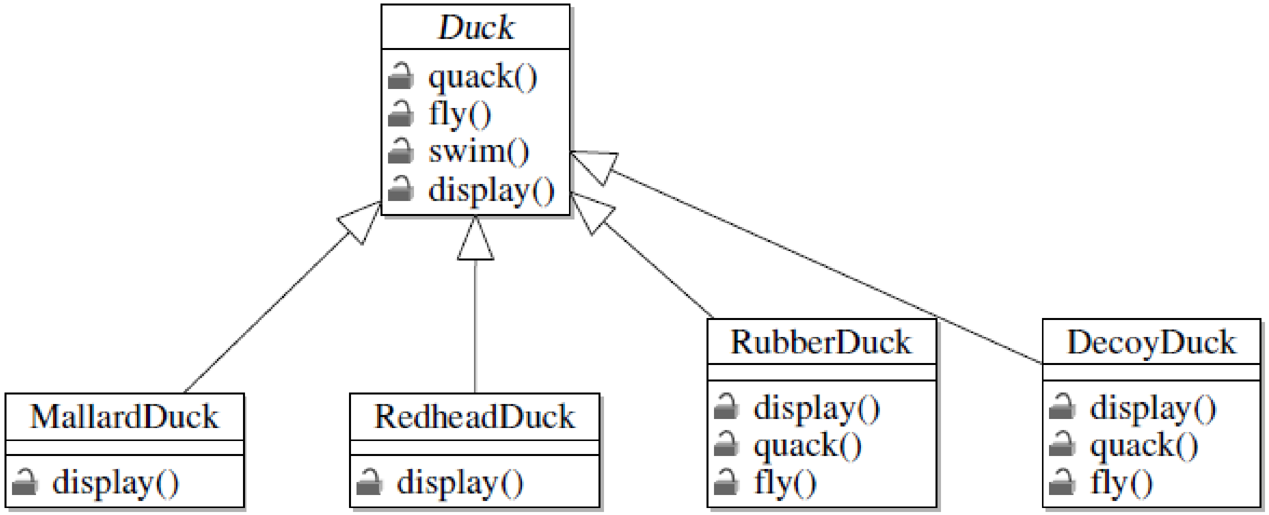
Applying design principle #1: Identifying part of the design that either varies or changes



- quack varies, because some ducks quack, some squeak, and some make no sounds at all.
  - In future there may be other variants of quacking as well

Duck simulator design - Applying OO Design Principles

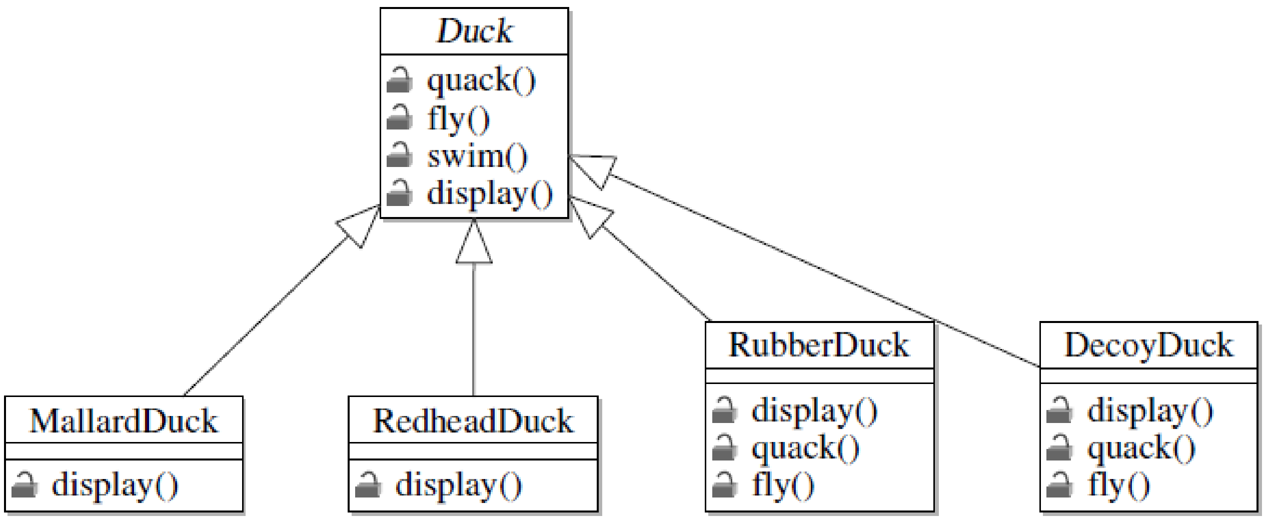
Applying design principle #1: Identifying part of the design that either varies or changes



- fly also varies across subclasses. Some ducks fly, some don't, and some ducks fly in different ways.

Duck simulator design - Applying OO Design Principles

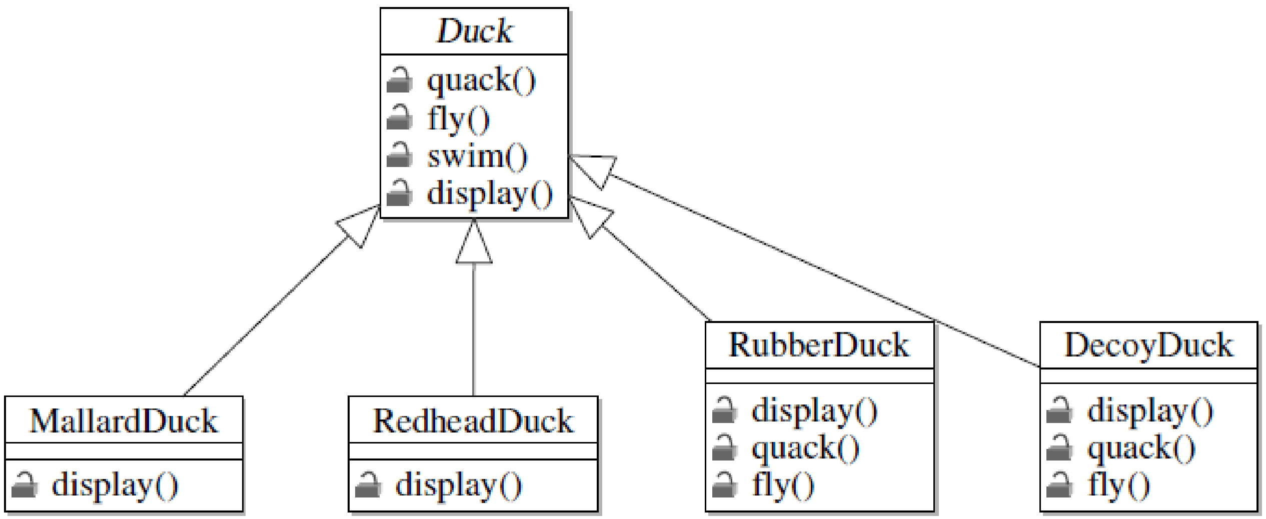
Applying design principle #1: Identifying part of the design that either varies or changes



- swim so far seems to be a constant and doesn't change.

Duck simulator design - Applying OO Design Principles

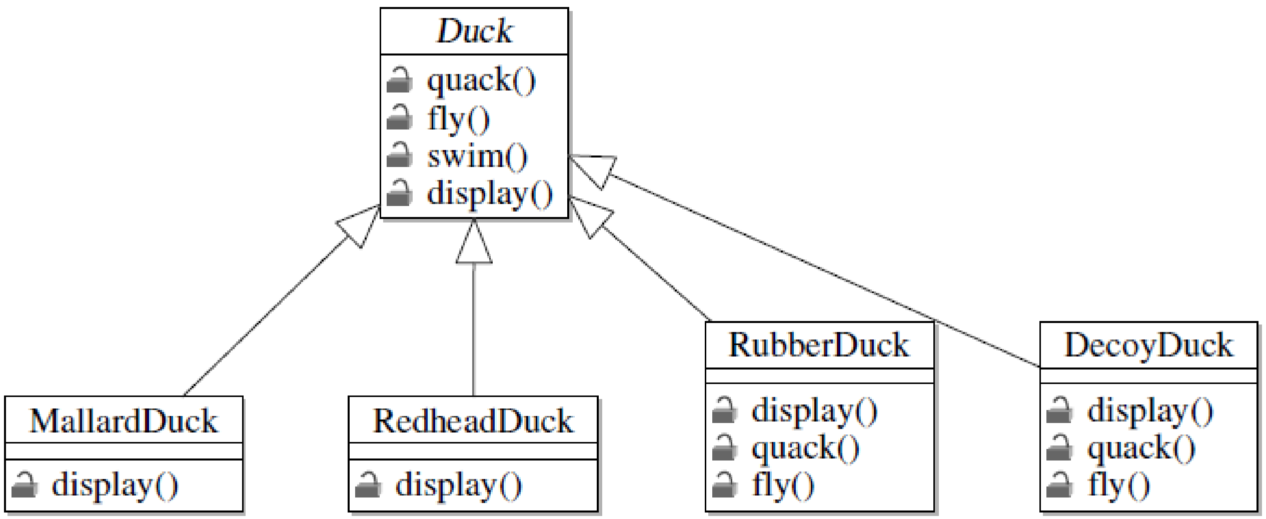
Applying design principle #1: Identifying part of the design that either varies or changes



- display is implemented already by each individual duck by design.

Duck simulator design - Applying OO Design Principles

Applying design principle #1: Identifying part of the design that either varies or changes



Action: Pull these identified duck behaviors that change or vary out of the Duck class and create new classes for these behaviors.

Strategy Design Pattern
Part C

Fani Deligianni

fani.Deligianni@glasgow.ac.uk

https://www.gla.ac.uk/schools/computing/staff/fanideligianni/

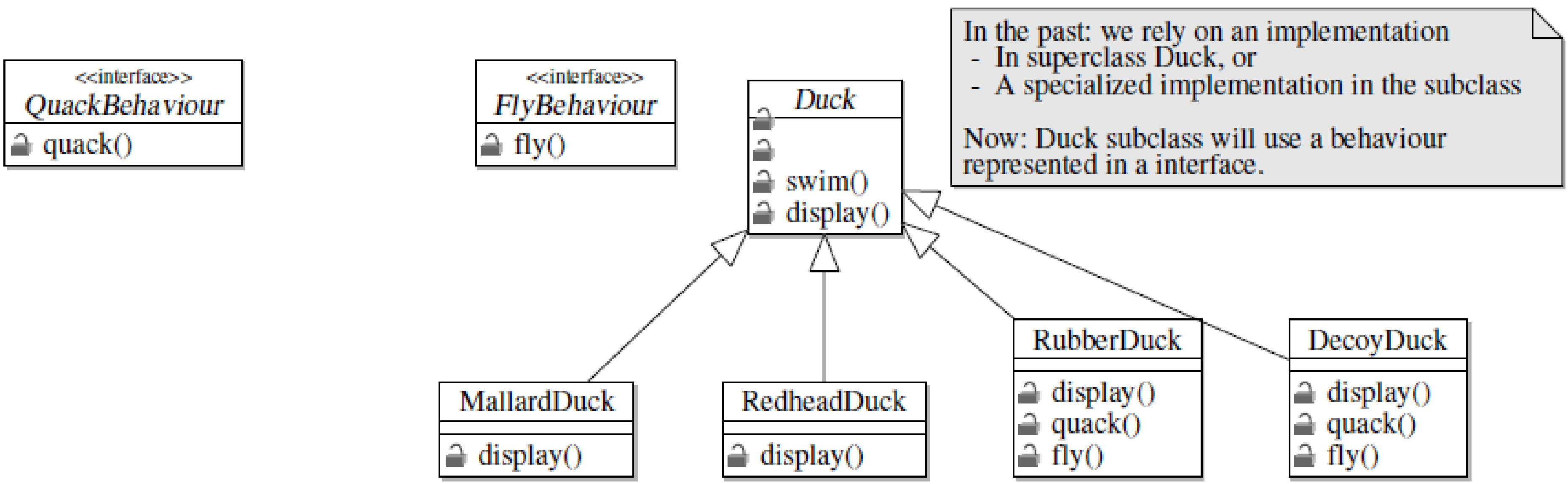
OO Design Principles

Design Principle #2:
Program to an interface, not an implementation

Duck simulator design - Applying OO Design Principles

Applying design principle #2: Step 1: Use interface (supertype) to represent each Δ behaviour

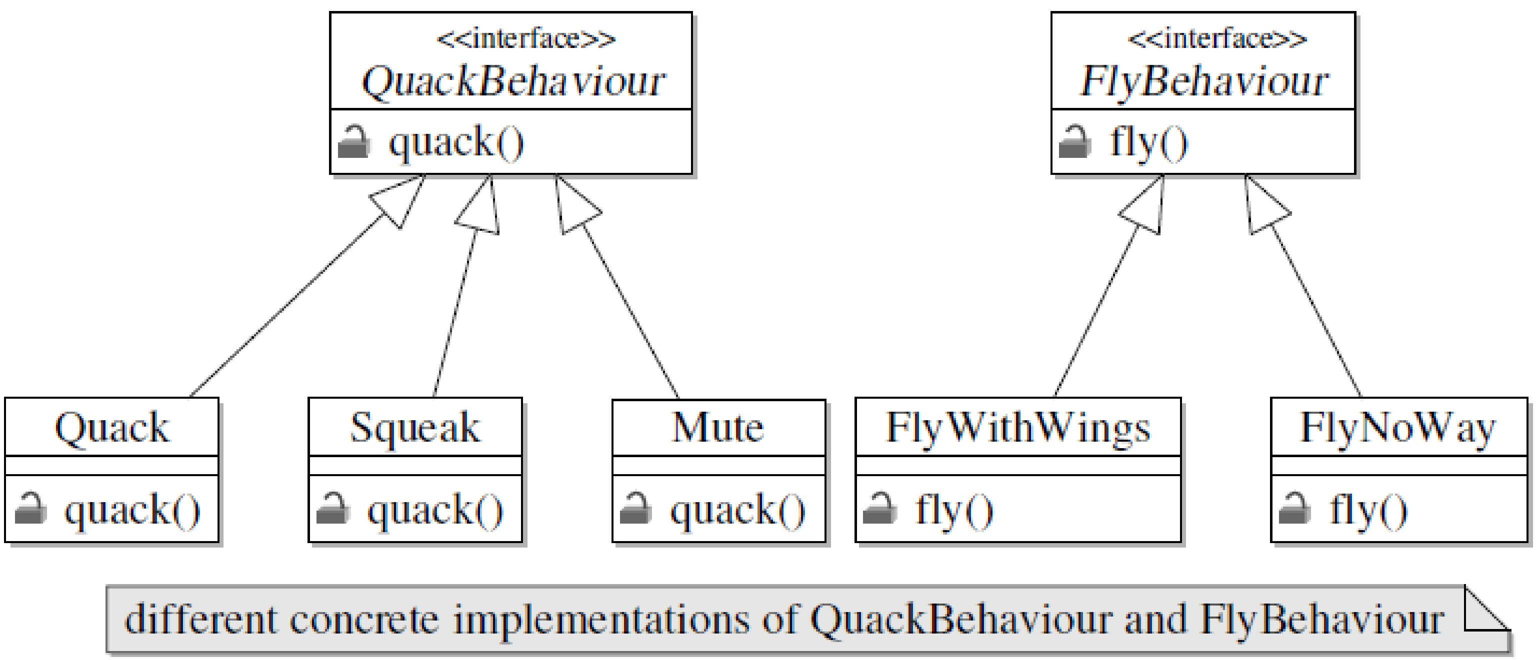
- Each implementation of a behavior will implement one of these interfaces.



Duck simulator design - Applying OO Design Principles

- **Applying design principle #2:**

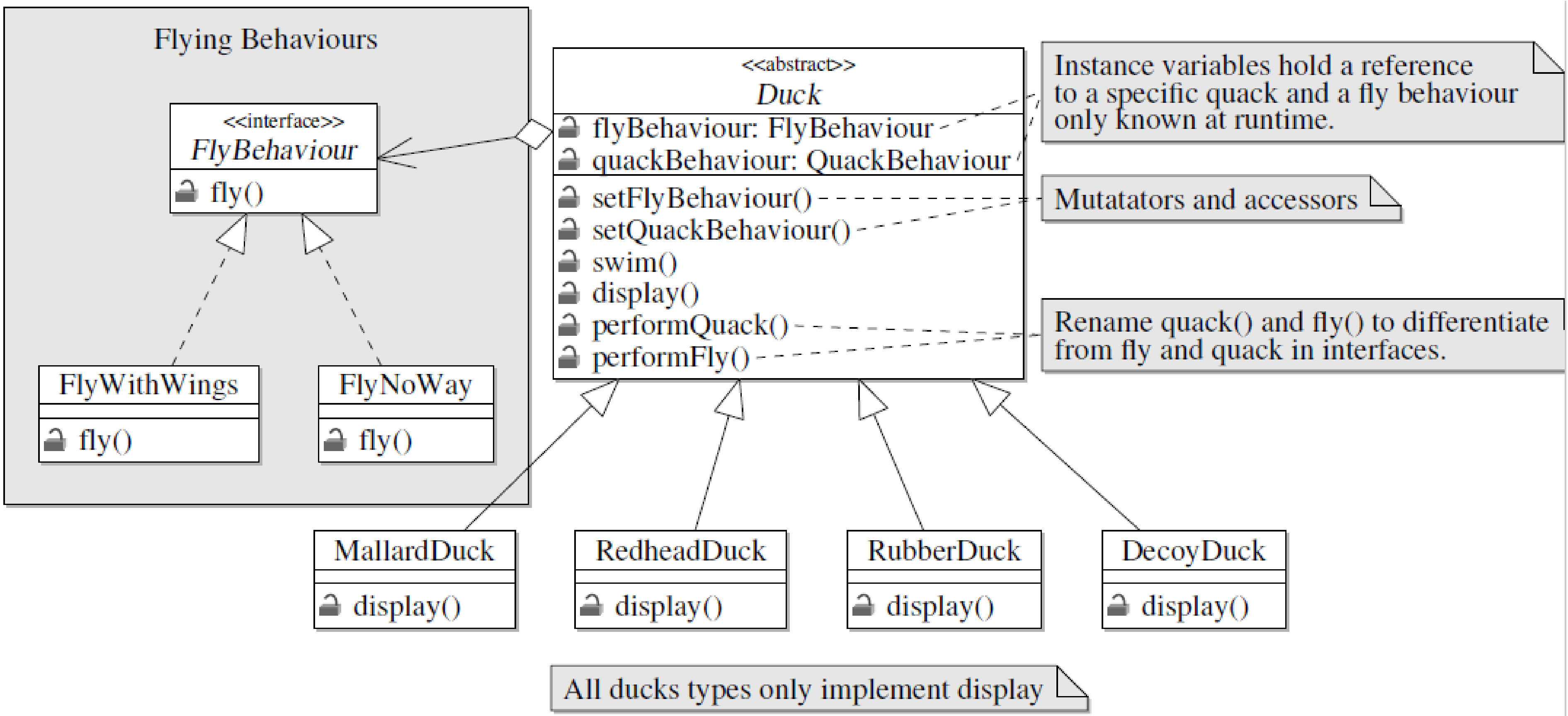
Step 2: Use these interfaces to implement some concrete quacking and flying behaviors.



Duck simulator design - Applying OO Design Principles

- **Applying design principle #2:**

Step 3: Rework the Duck class



Duck simulator design - Applying OO Design Principles

- **Applying design principle #2:**
- **Result:**
  - Rather than relying on an implementation of behaviour in our ducks, we are relying on an interface.

Duck simulator design - Applying OO Design Principles

Applying design principle #2:

Result:

- This means ducks are no longer locked into specific implementations
  - i.e Ducks do not need to know how details of how they implement the behaviours, but we'll know at run time, when it set.

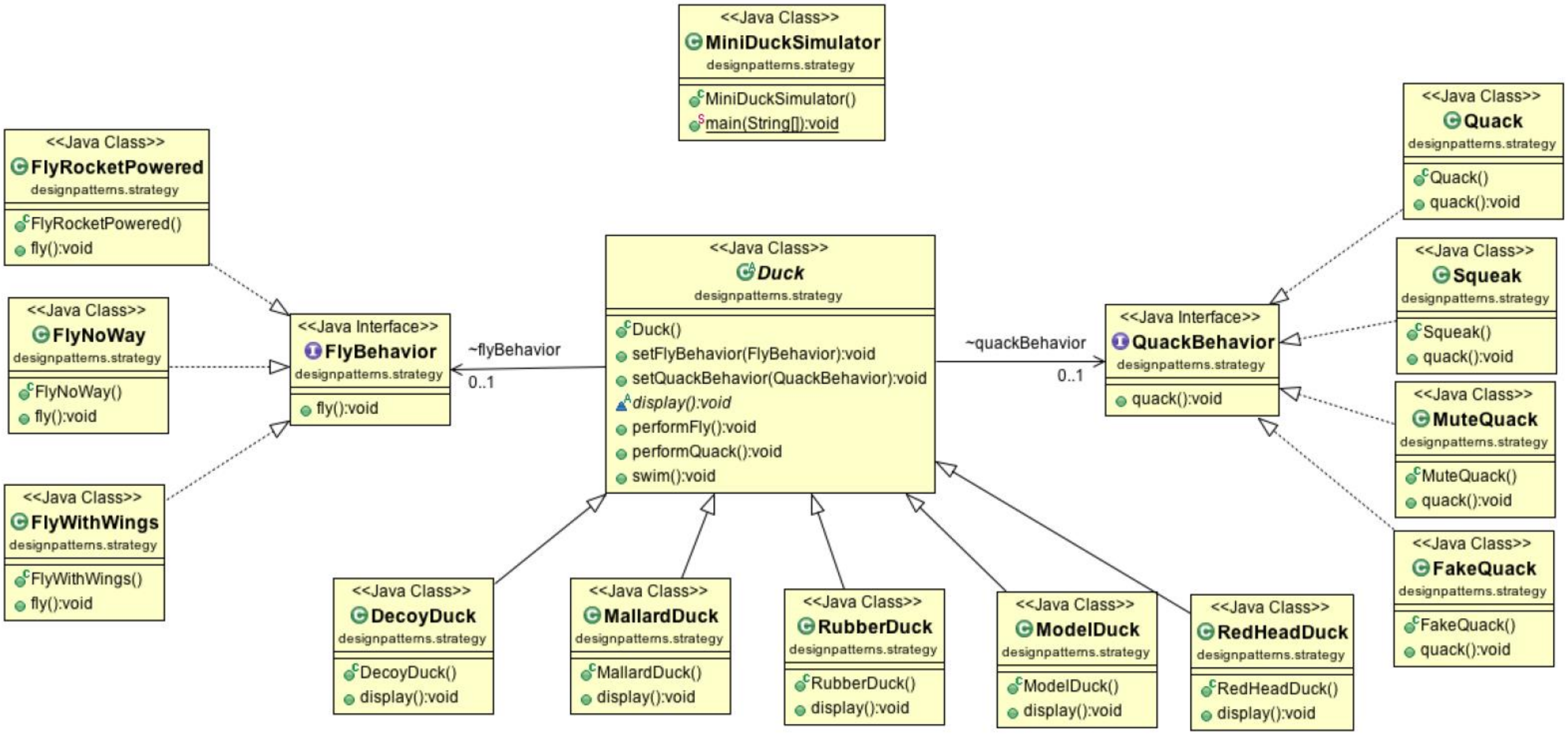
Duck simulator design - Applying OO Design Principles

Applying design principle #2:

Result:

- This means ducks are no longer locked into specific implementations
  - i.e Ducks do not need to know how details of how they implement the behaviours, but we'll know at run time, when it set.

Duck simulator design - Applying OO Design Principles



Duck simulator design - Applying OO Design Principles

We have now used composition for the behaviors that need more flexibility, and inheritance for behaviors that you know won’t need to change.

- This is an example of the Strategy Design Pattern

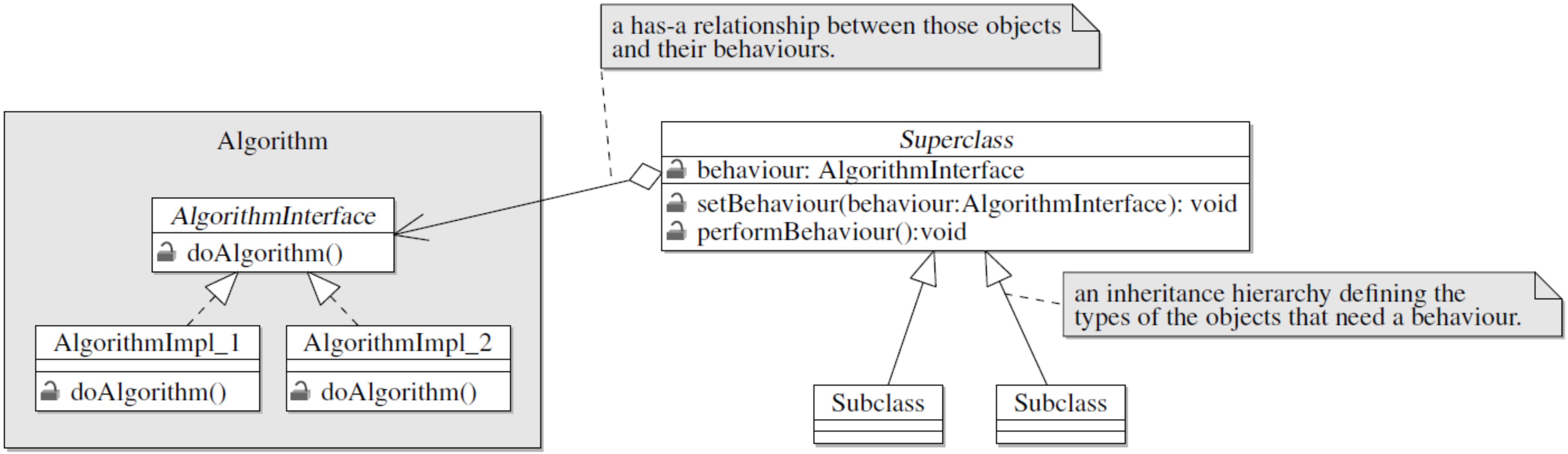
Strategy Design Pattern
Part D

Fani Deligianni

fani.Deligianni@glasgow.ac.uk

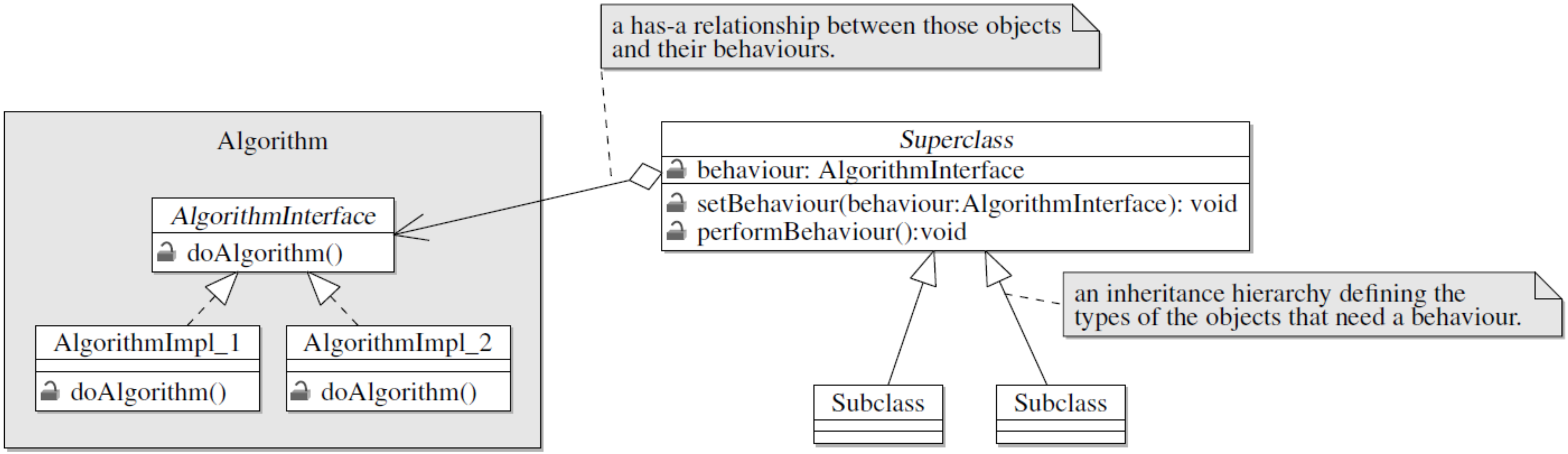
https://www.gla.ac.uk/schools/computing/staff/fanideligianni/

Strategy Design Pattern



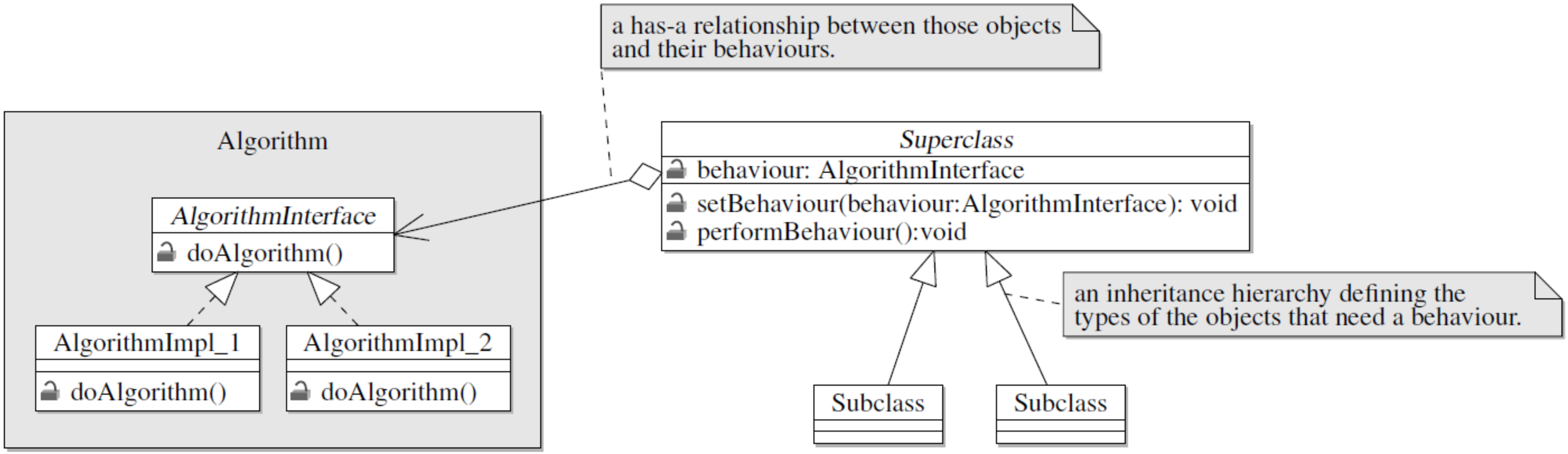
- By moving the algorithms out from the main inheritance hierarchy, we get the benefit of being able to choose which algorithm each object gets.

Strategy Design Pattern



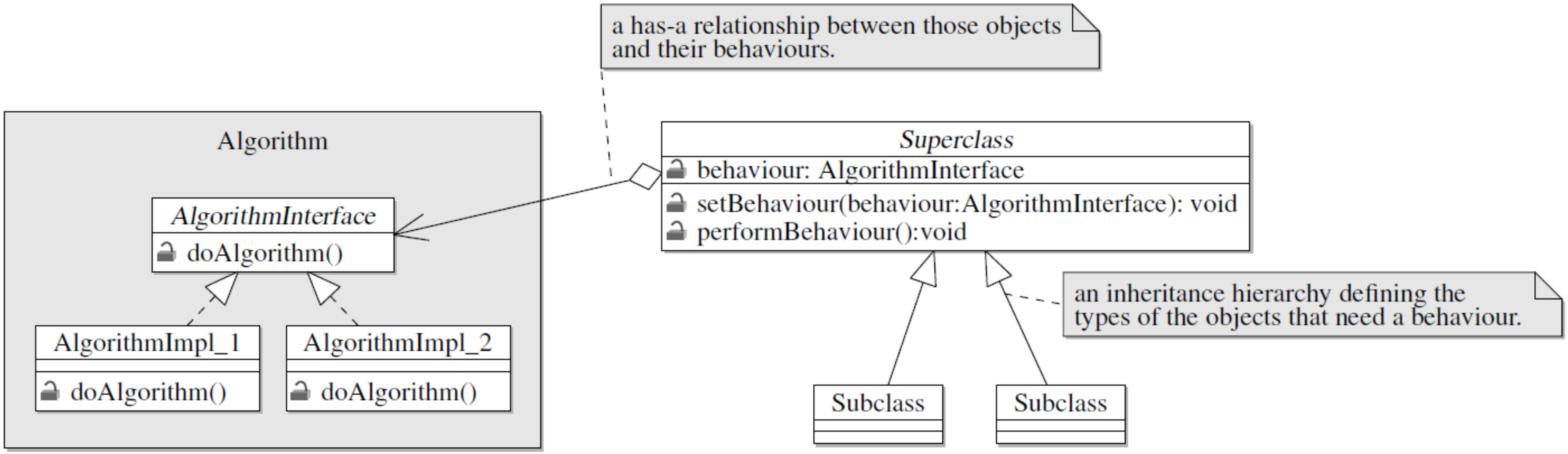
- We can change these algorithms at run time

Strategy Design Pattern



- If multiple objects need to use the same algorithm, we get the benefit of code reuse too.

Strategy Design Pattern



Definition

- The strategy pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable.
- Strategy lets the algorithm vary independently from clients that use it.

HAS-A vs IS-A relationships

In our previous design, we used **is-a** relationship. Thus:

- For example, we said a decoy duck **is a** duck.
- We also had ducks inherit a flying implementation, or, worse, inheriting one they don't want and have to override anyway.

HAS-A vs IS-A relationships

In our new design, we used **has-a** relationship. Thus:

- Each duck has a FlyBehavior and a QuackBehavior to which it delegates flying and quacking.
- Instead of inheriting behavior, ducks get their behavior by being composed with the right behavior object

OO Design Principles

Design Principle #3:

Favor composition over inheritance

1. More flexibility
2. Encapsulate a family of algorithms into their own set of classes
3. Able to change behavior at runtime

Summary

- The strategy Pattern
  - Defines a family of algorithms;
  - Encapsulates each one;
  - Makes them interchangeable.
- Strategy lets the algorithm vary independently from clients that use it