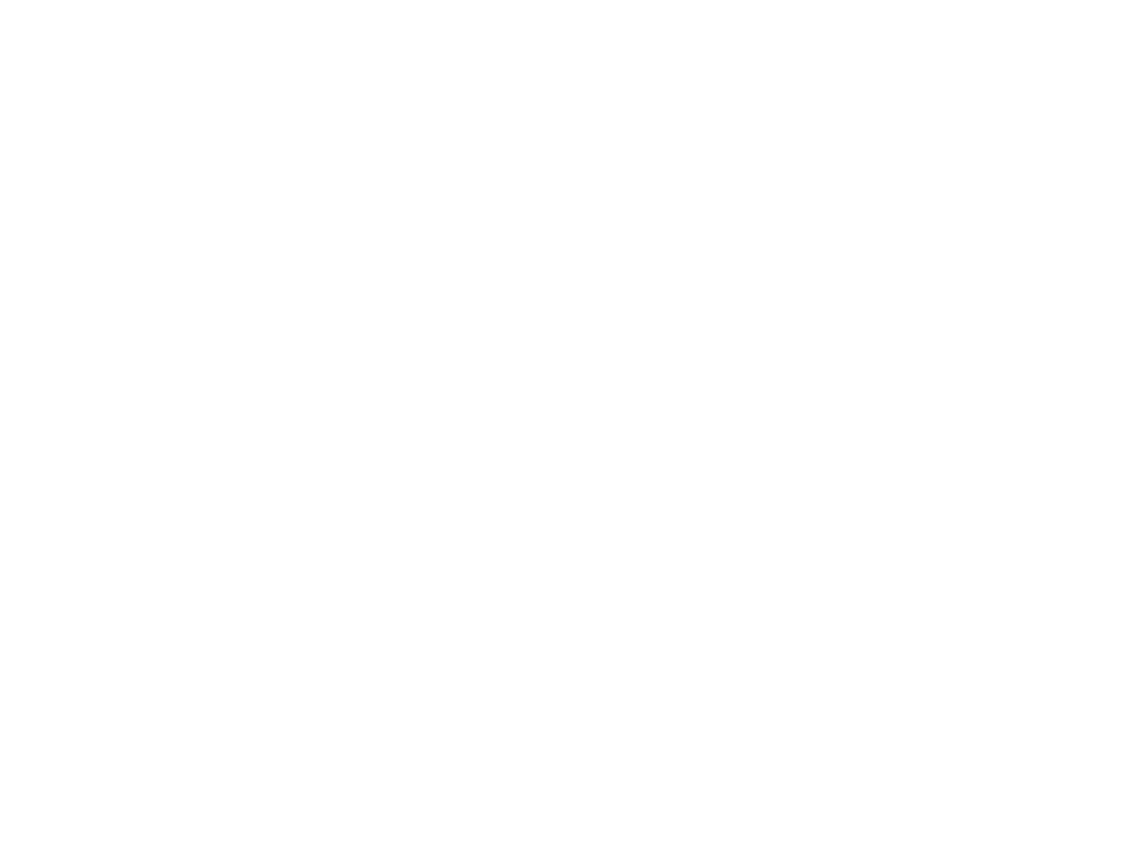
Introduction to Design Patterns

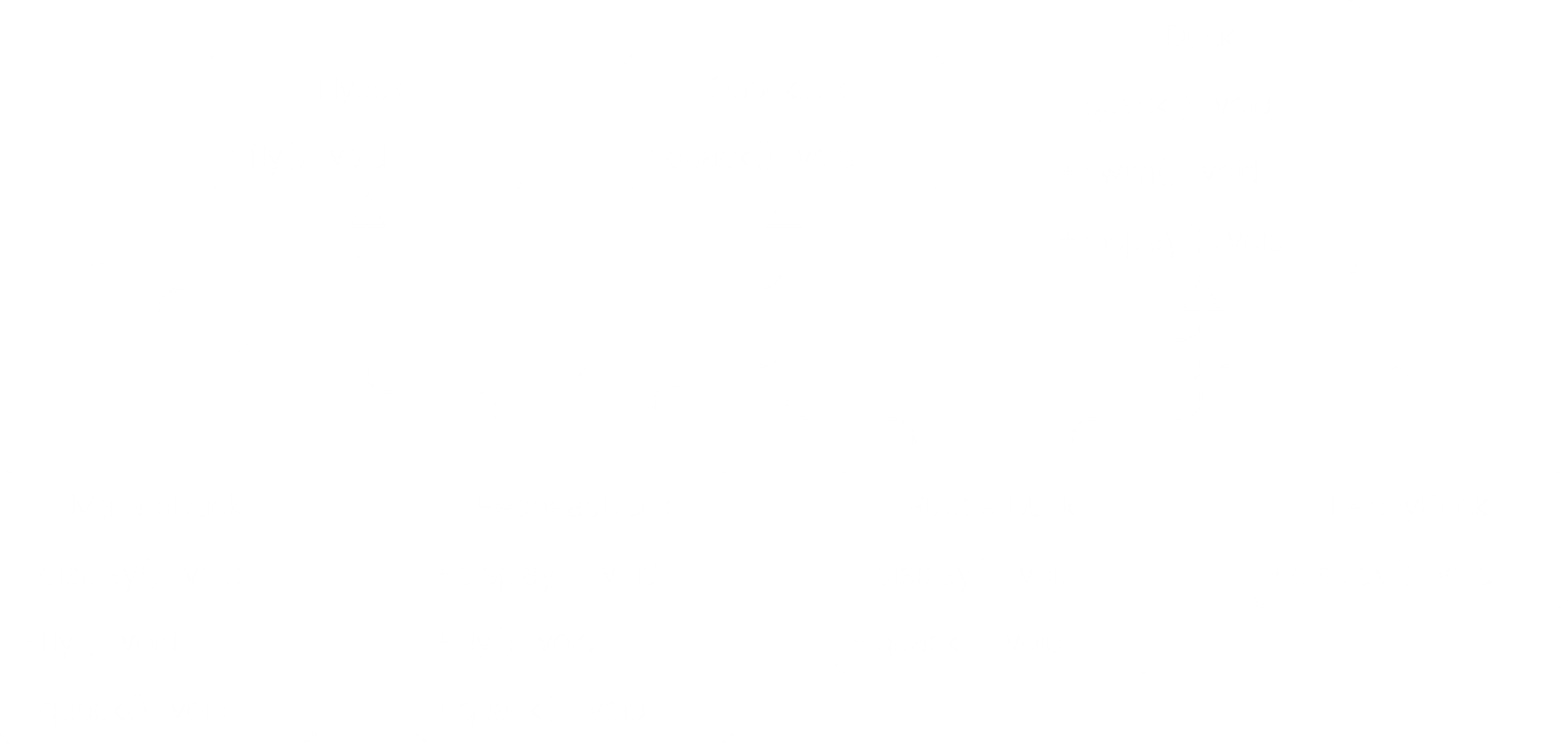
Consider that we want to make a game that has different types of ducks. This suggests that there should be an **abstract parent class**, Duck. In our implementation, let’s assume that there are three methods, quack, swim, and display, the last of which is different for every type of duck. As such, the display method is made abstract in the Duck class (you can tell because it’s in italics).



Say we need to add a new method, fly. However, not every duck can fly. If we add this method as an abstract method to the parent class, then every duck has to implement it, even the ducks that cannot fly. This is an inappropriate solution. We should not force classes that do not require some behaviour to inherit that behaviour.

Instead, we should use an **interface**. This is particularly useful if we have a number of variable behaviours.

💡**Design Principle 01**: Identify the aspects of your application that vary. ‘Encapsulate’ them so they do not affect the rest of your code and separate them from what stays the same.



By creating an interface, we ensure that any new type of flying behaviour means that the objects that need that behaviour just implement the interface. There is no change to our existing code. This is achievable because we are programming to the interface, not the implementation.

💡**Design Principle 02**: Program to the interface, not to the implementation.

What this means is that our code will always call the method of the interface, not the method implemented by the object.

// Coding to implementation -> bad  
MallardDuck duck = new MallardDuck();  
duck.fly();

// Coding to interface -> good  
Duck d = new MallardDuck();  
d.fly();

JAVA

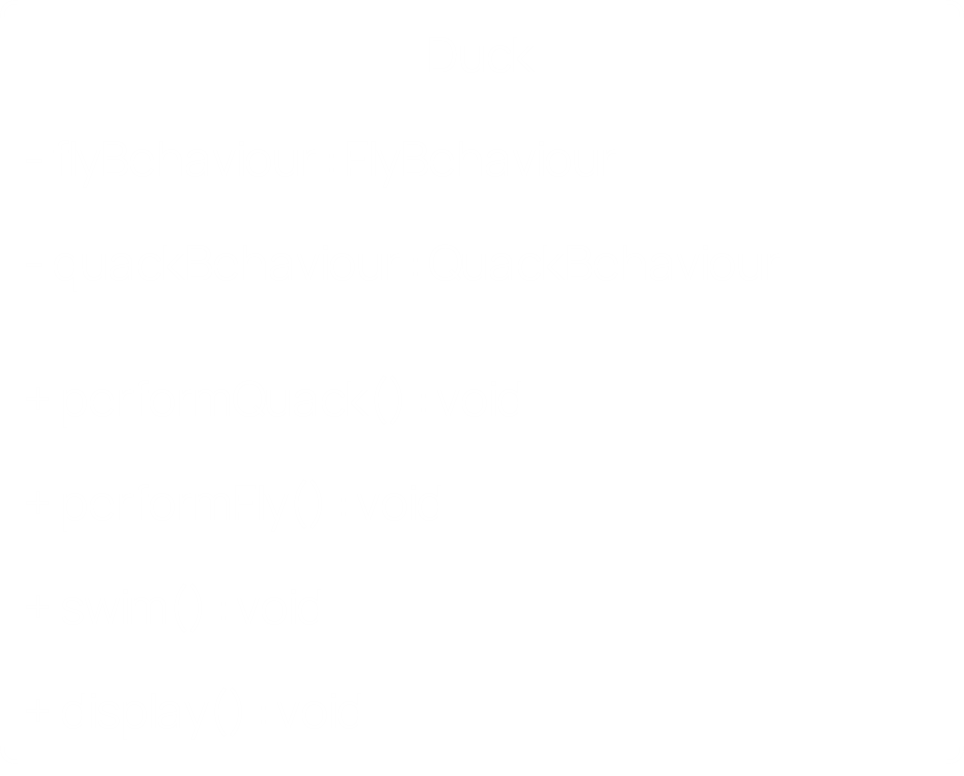
This is better since it provides us with more flexibility. In fact, we will usually not even create a specific type of duck here but rather have another method that retrieves the correct type of duck. This completely separates our code from the implementation.

d = getDuck();  
d.fly();

JAVA

For the keen-eyed, there are two issues that appear with this approach. Firstly, the current setup requires every duck having to implement the new *FlyBehavior* interface. This is a lot of code repetition. Secondly, we are using the Duck abstract class to call the fly method, but how do we ensure that the specific object really did implement some flying behaviour?

To solve both issues at once, we add the *FlyBehavior* interface as a reference to the Duck class.



By doing this, we ensure that any duck must have a flying behaviour. In addition, to avoid the issue of code repetition, we have pre-built implementations of the FlyBehavior interface that new classes can use.



public class MallardDuck extends Duck {  
  
 public MallardDuck() {  
 flyBehaviour = new FlyWithWings();  
 }  
}

JAVA

The above solution brings us to yet another important design principle.

💡**Design Principle 03**: Favour composition over inheritance.

What we did above is that we used **composition** to add behaviour dynamically to the Duck class instead of using **inheritance** to inherit behaviour. This ensures that our code is much cleaner. The purpose of inheritance is not to reduce code reuse, but rather to provide structure. Using inheritance instead of composition to reduce code reuse is one of the most common pitfalls of writing code.

## Strategy Pattern

Everything we have done so far brings us to our first design pattern, 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.

That very complicated sentence is actually very simple. We created a **family of algorithms** (*FlyBehavior*), encapsulated each one (the different implementations of *FlyBehavior*) and made them **interchangeable** (we can replace one *FlyBehavior* implementation with another). This allowed the algorithm (i.e., the implementation of *FlyBehavior*) to vary independently from the clients that used it (i.e., the calls to the fly method were unaffected by the implementation of *FlyBehavior* changing).

The strategy pattern highlights the importance of understanding the difference between is-a and has-a relationships. We should only ever use inheritance for is-a relationships, not has-a ones. Every type of duck **is-a duck**, so we use inheritance. But flying behaviour is a has-a relationship. Every type of duck **has-a flying behaviour**. Thus, we use composition.

The strategy pattern enforces the **Open Closed Principle**, in that every class is made open for extension, but closed for modification.

The overall picture looks like this:

