**Design Patterns**

Table of Contents

[Strategy Pattern 4](#_Toc83393372)

[Observer Pattern 9](#_Toc83393373)

[Decorator Pattern 11](#_Toc83393374)

[General Architecture 14](#_Toc83393375)

[Advantages 15](#_Toc83393376)

[Difference Between Decorator and Strategy Patterns 16](#_Toc83393377)

The main take away from this section is going to be that someone has already solved our problems. Essentially, whatever problem it is that we are trying to solve, we do not have to figure out the method in which to solve it. The method already exists for whatever problem we are having. We simply need to implement the solution according to our needs.

A **design pattern** is a general solution to a common problem in a context. For each pattern we will be seeing, we will first describe a **problem** which occurs **repeatedly** and commonly and then describe the **core** of the problem.

In software development, design patterns have two main uses:

1. Using design patterns creates a **common platform** for developers. If a design pattern is used, other developers who know the design pattern will easily understand the code.
2. It is **best practice** to use design patterns since they have evolved over a long time and thus provide the best possible solutions to certain problems. Knowing these patterns can allow an inexperienced developer to learn good software design quickly and easily.

A design pattern will have four components:

1. The **name** of the pattern.
2. The **purpose** of the pattern, i.e. what problem it solves.
3. **How** to solve the problem.
4. The **constraints** that have been considered in the solution.

There are about **26 patterns** that have been discovered so far. These can be divided into **three categories**:

1. **Creational**: These patterns deal with class instantiation, either creating classes or objects.
2. **Structural**: These patterns deal with a class’s structure and composition. The main goal for these patterns is to increase the functionality of the classes without changing much of the composition.
3. **Behavioural**: These patterns deal with how a class communicates with other classes.

|  |  |  |
| --- | --- | --- |
| **Creational** | **Structural** | **Behavioural** |
| Factory Method | Adapter | Template |
| Abstract Factory | Bridge | Strategy |
| Builder | Composite | Command |
| Singleton | Decorator | State |
| Multiton | Façade | Visitor |
| Object Pool | Flyweight | Chain of Responsibility |
| Prototype | Front Controller | Interpreter |
|  | Proxy | Observer |
|  |  | Iterator |
|  |  | Mediator |
|  |  | Memento |

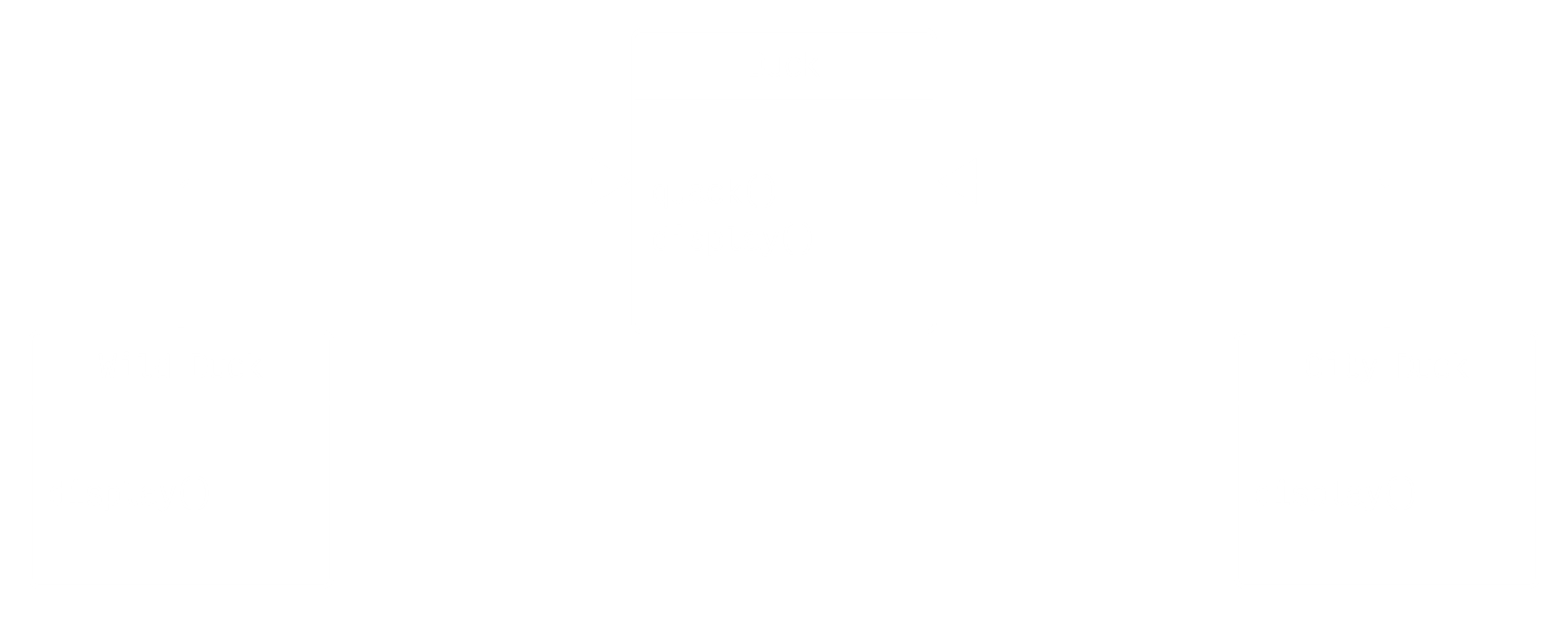
We should be careful when using design patterns to ensure we are not using them too often. The simplest solution is always the best, so design patterns should only be used when the need arises.

## Strategy Pattern

[YouTube link](https://www.youtube.com/watch?v=v9ejT8FO-7I)

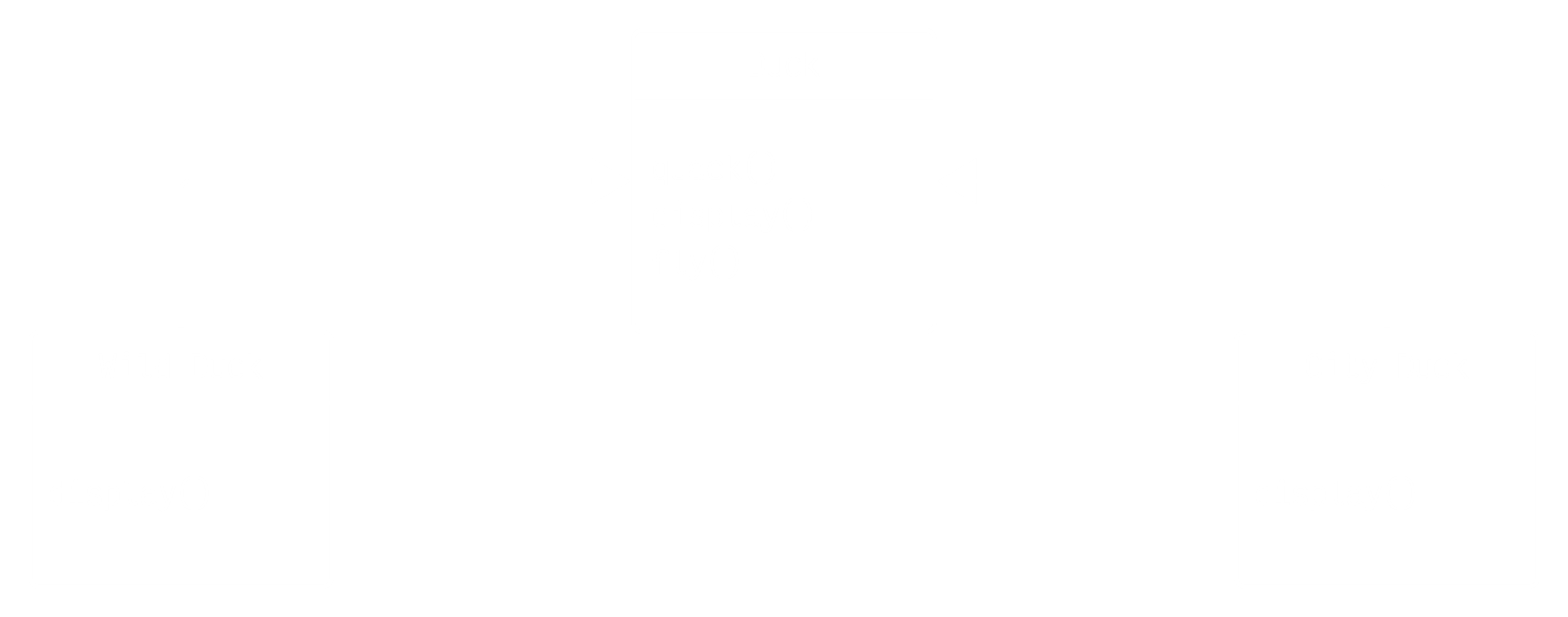
The **Strategy Pattern** is the simplest pattern. It is about using **composition** rather than **inheritance**. We need to understand that inheritance is not intended to be used for **code reuse**.

The strategy pattern defines a **family of algorithms**, **encapsulates** each one and makes them **interchangeable**. This allows the algorithms to vary independently from the clients that use it.

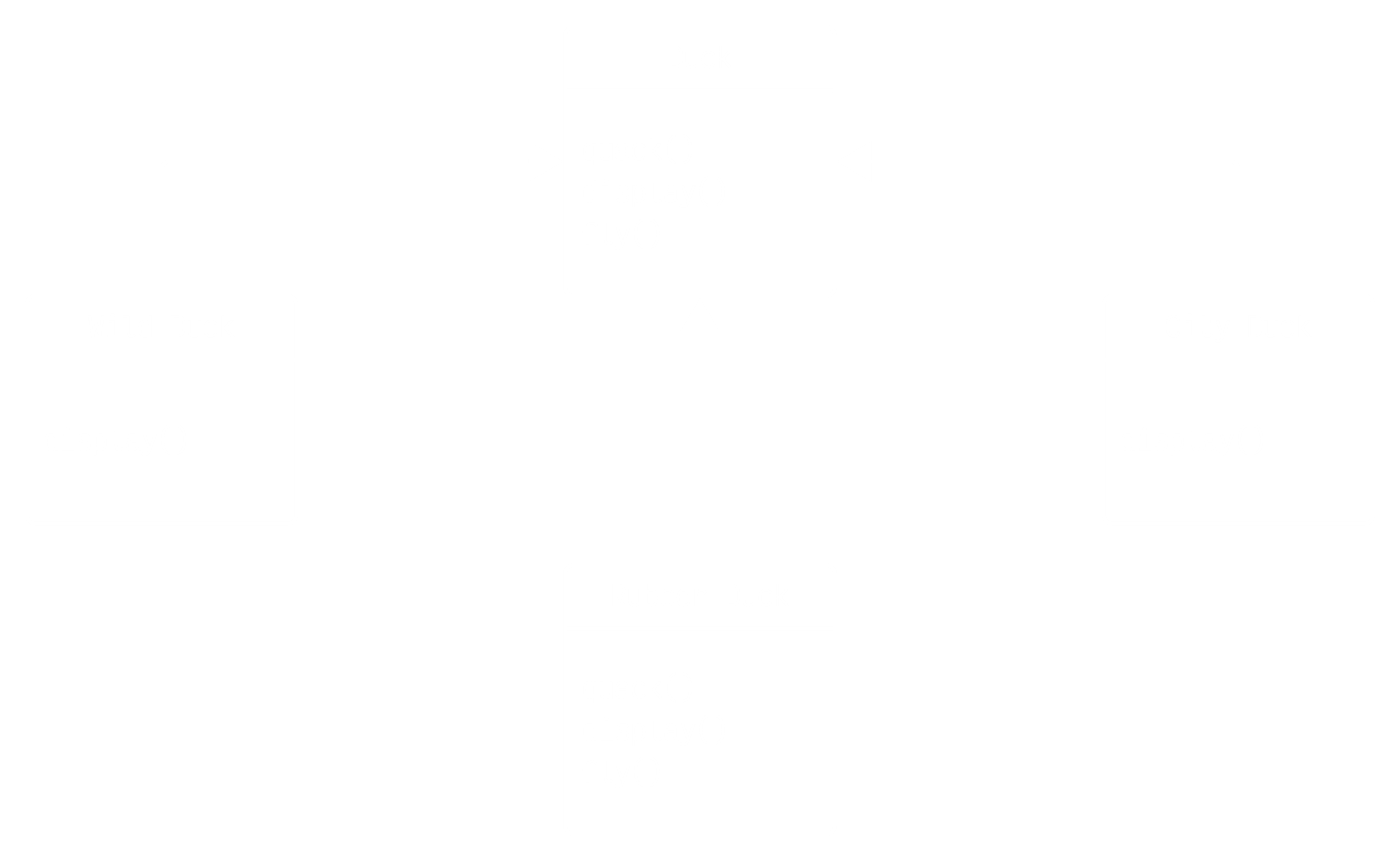


Consider we have the situation shown above. The quack() method is the same for both subcategories of ducks, but the display() method is different, which is why it has been overridden in each of the child classes.

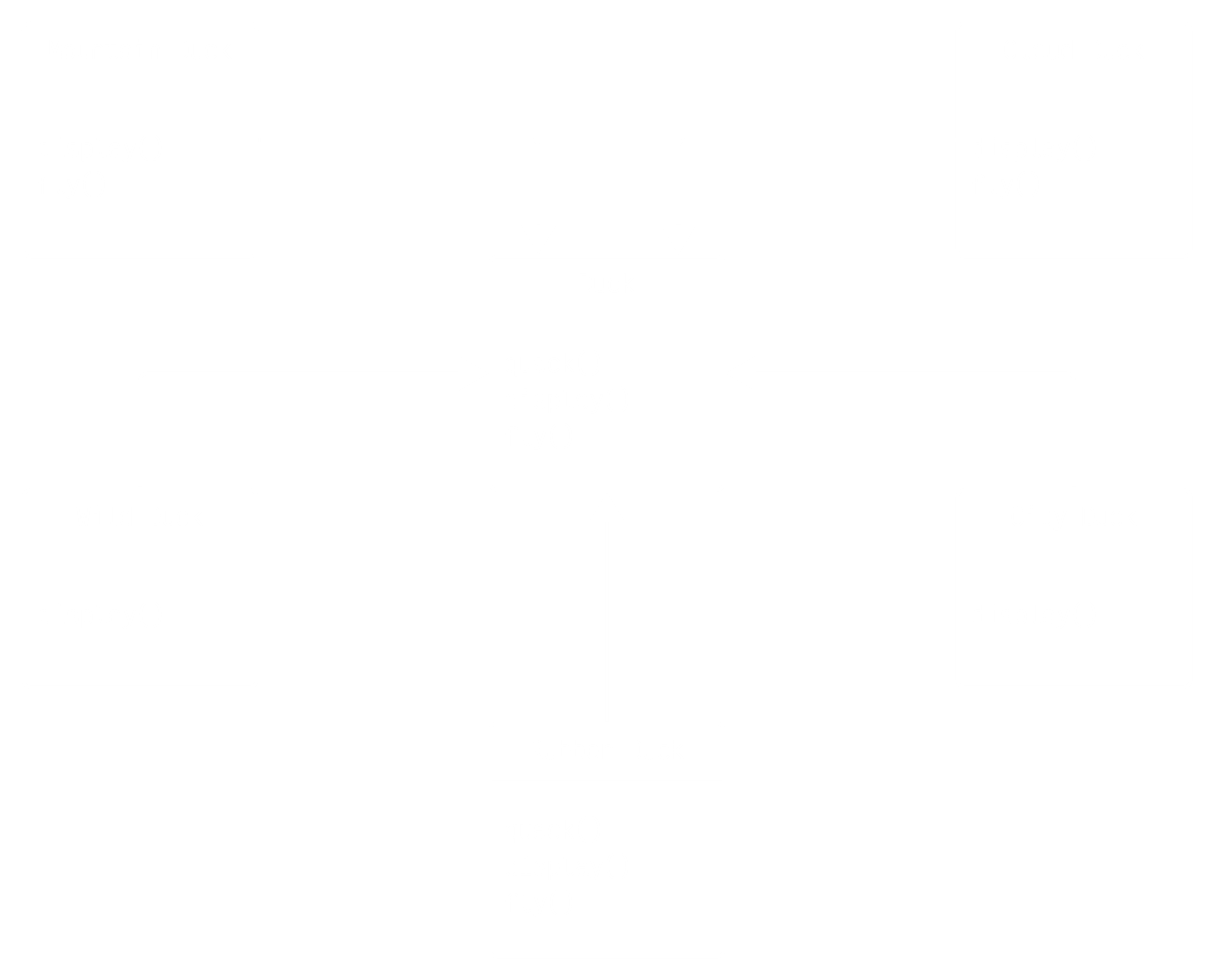
Now we need to add another method for all of the classes, the fly() method. To do this, we can simply add it to the super class and not touch the sub classes.



Now say we need to add another class, rubberDuck, which is a fake duck and cannot fly or quack. One way we could handle this situation is if we **override** the fly() method in the rubberDuck class and make it do nothing.



Now say we are having to add two more classes of ducks both of which have the same fly() method, but that method is still different from the base class’s one.

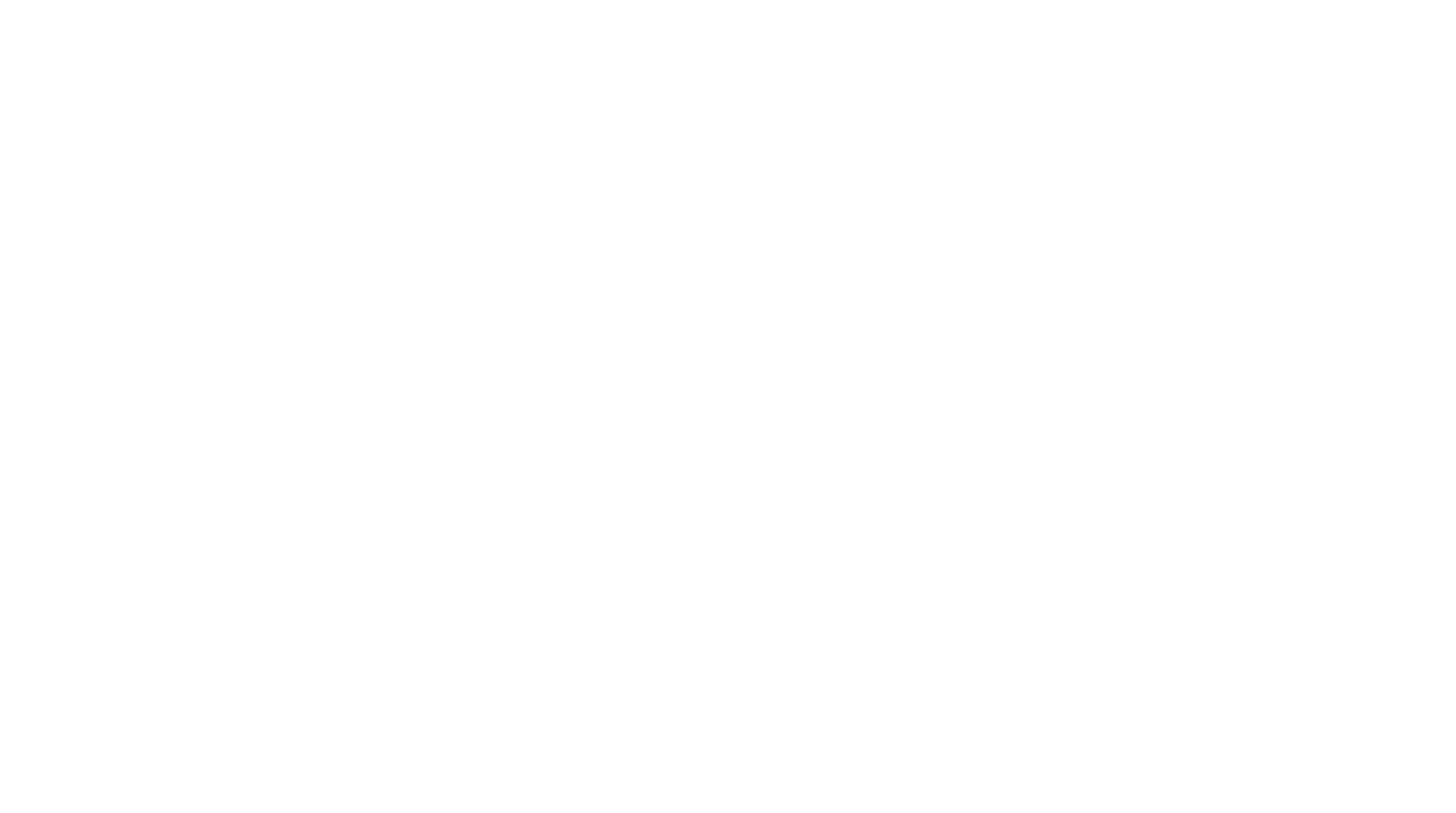


The use of the fly() method in the original base class is becoming pointless. We are having to implement the method for each new class anyways.

Notice that the first sentence from this section is applicable here. We are trying to use **inheritance** to perform **code reuse**, but that is not the purpose of inheritance, which is why things are getting messy. We just considered one method here, but if we add more methods that are varying from class to class, things will become impossible to keep track of. For example, the display() method could also have multiple different implementations.

Instead of doing things this way, we will essentially be using a **plug-and-play** approach. We will have all the different variations of methods available to us and for each new class, we will simply use those methods. In this way, if a particular method needs to be changed, the different classes that use the method do not need to worry about it.

In order to do this, we add mandatory interfaces to the base class. We then create classes that implement those interfaces and have specific implementations of the methods.



If we do this, we no longer need all those different classes for different types of ducks. Instead, we can have **objects** of the IFly and IDisplay interfaces in the implementation of the Duck class. Using the **constructor** of the Duck class, we can then assign a particular child of each of those interfaces when creating the object.

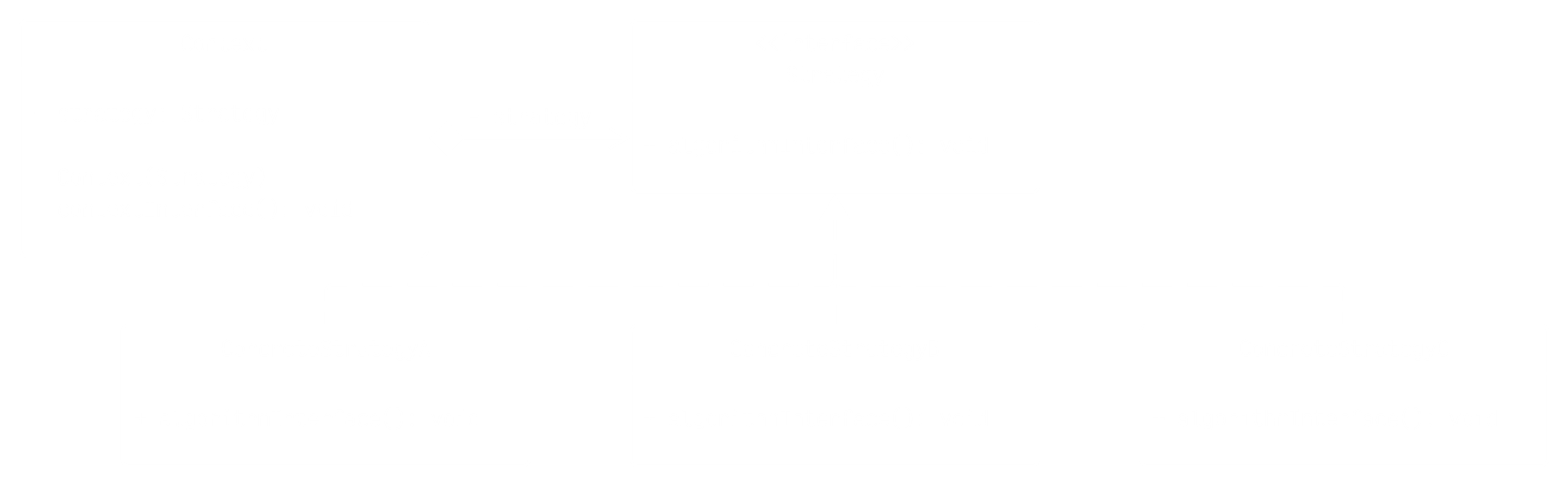
Duck  
{  
 IFly flyBehaviour;  
 IDisplay displayBehaviour;  
 public Duck (IFly flyBehaviour, IDisplay displayBehaviour)  
 {  
 this.flyBehaviour = flyBehaviour;  
 this.displayBehaviour = displayBehaviour;  
 }  
 public void fly() { flyBehaviour.fly(); }  
 public void display() { displayBehaviour.display(); }  
}

Duck(flyOne, displayTwo);

JAVA

We have made the Duck **independent** from the **algorithms** it is using. We could use any algorithm we want without having to modify the Duck class.

A general diagram for the strategy pattern is shown below:



## Observer Pattern

Consider that we have a complex object that can change its data over time. Whenever this data changes, we need to update other parts of our program.

The solution is to make the object observable by observers. Instead of making the observers repeatedly query the object to check for changes, the object will notify the observers when a change takes place.

An **observer** is an object that watches the state of another object and takes action when the state changes.

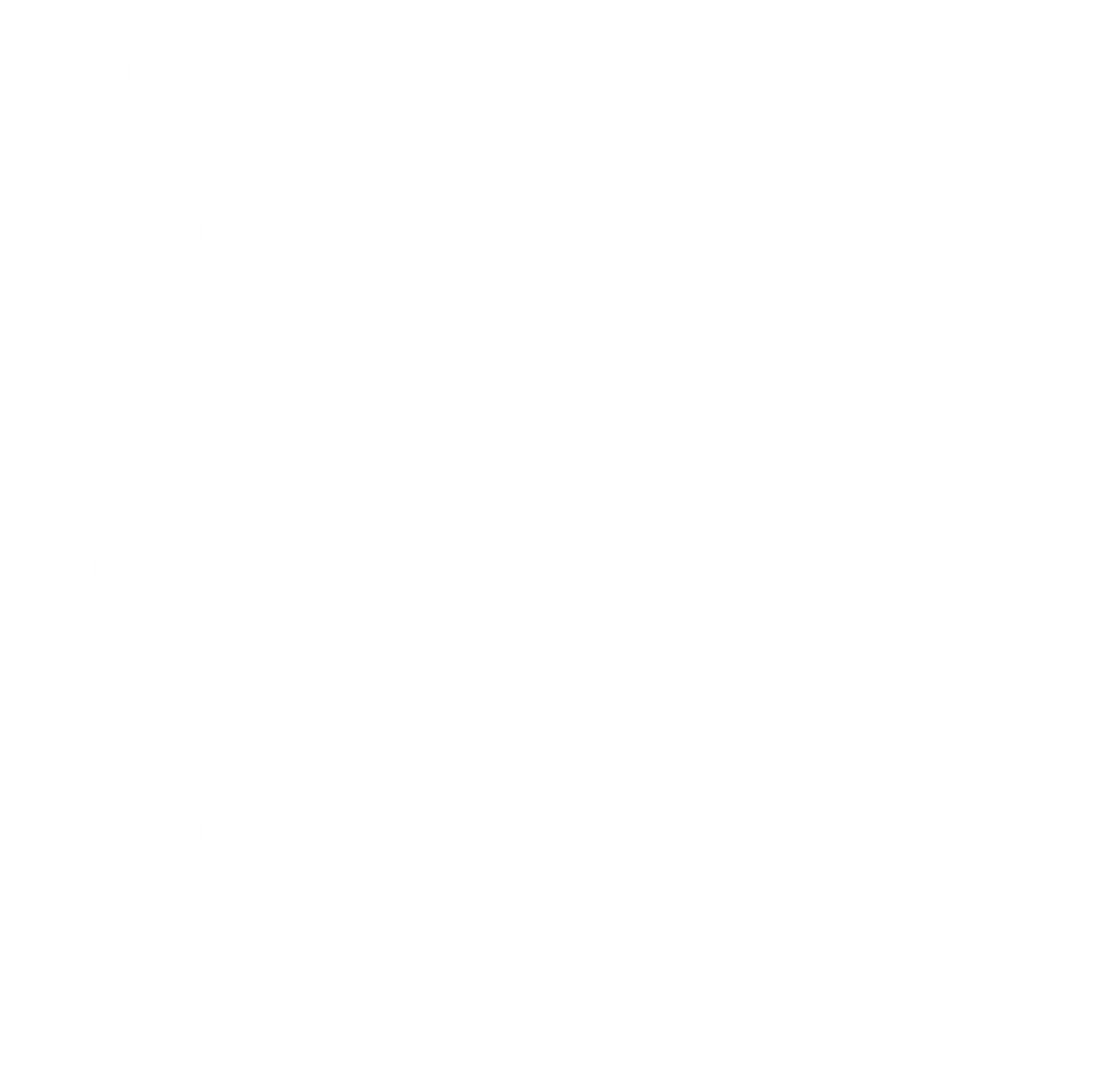
An **observable object** is one which notifies its observers whenever it has changed.

The observer pattern allows us to use extensive, event-based behaviour for data modelling and graphics.

The observer pattern has several parts:

* **Subject** – This is the observable object. It has a list of observers and interfaces for attaching and detaching observers.
* **Observer** – This is an updating interface for objects that gets notified about changes in a subject.
* **Concrete Subject** – This stores the states of interest that are to be observed. It sends a notification when a state change occurs.
* **Concrete Observer** – This is something that implements the observer.

Formally, the observer pattern defines a **one-to-many** dependency between objects so that when one object changes, all its dependencies are notified and updated automatically.



The diagram above should make the setup clear.

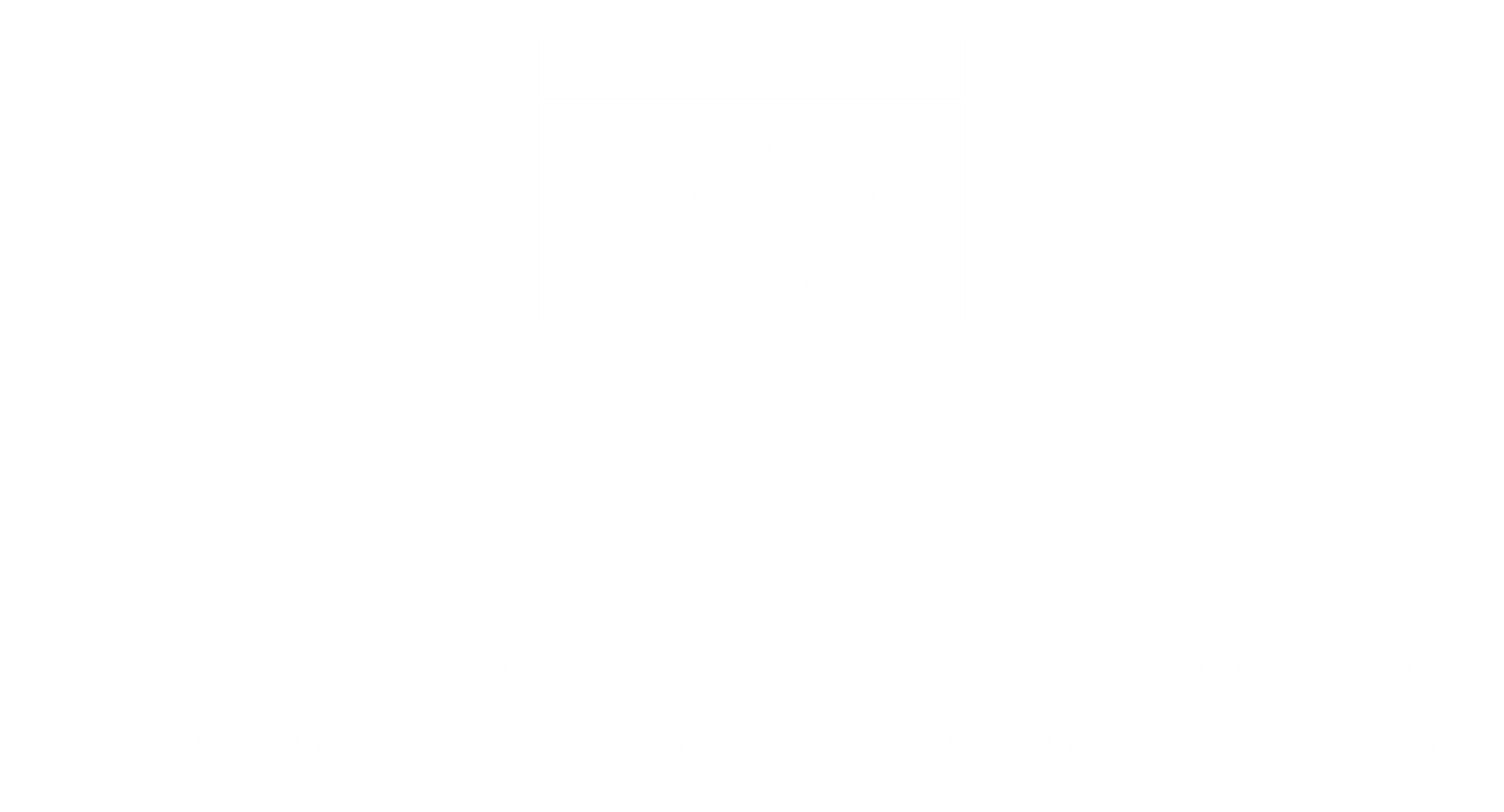
The IObservable interface is the **subject**. It has methods to add and remove **observers** to a list. Because observers can be **multiple types** of objects, we use the IObserver **interface** instead, which the observers must implement.

The **actual object** that has the states, concreteObservable, implements the IObservable interface for similar reasons, because each observer might connect to **multiple objects**.

Whenever some **change** takes place in concreteObservable, it calls the notify() method. This method in turn calls the update() method for each of the **attached observers**. That method now retrieves the state from the concreteObservable object using the getState() method.

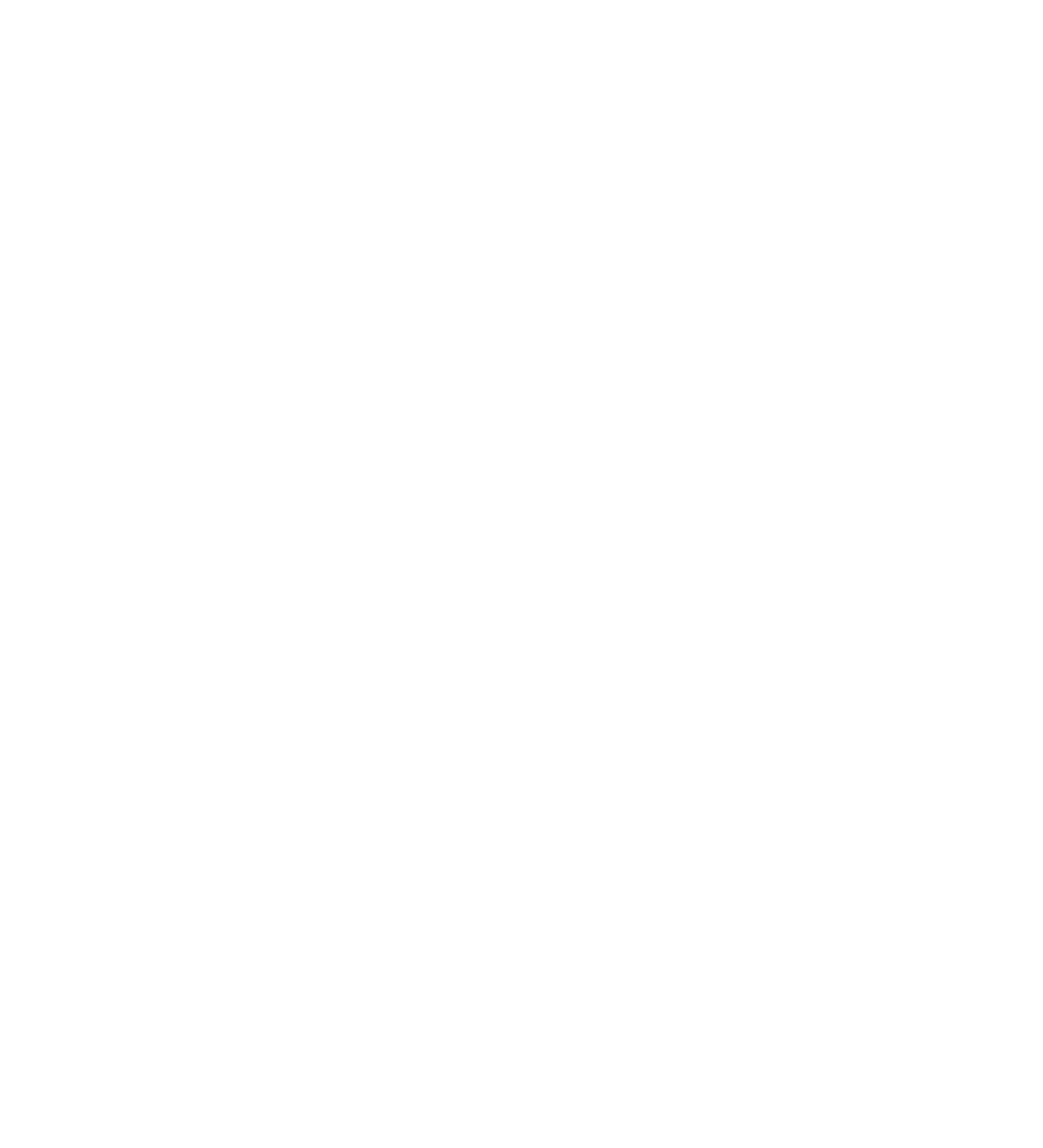
## Decorator Pattern

**Decorators** allow us to use **wrapper code** that extends the functionality of the core code. This is just wrapping objects around other objects so that we are defining new objects that have the capabilities of existing objects, thereby extended the functionality. This means the wrapper object and the main object have the same superclass.



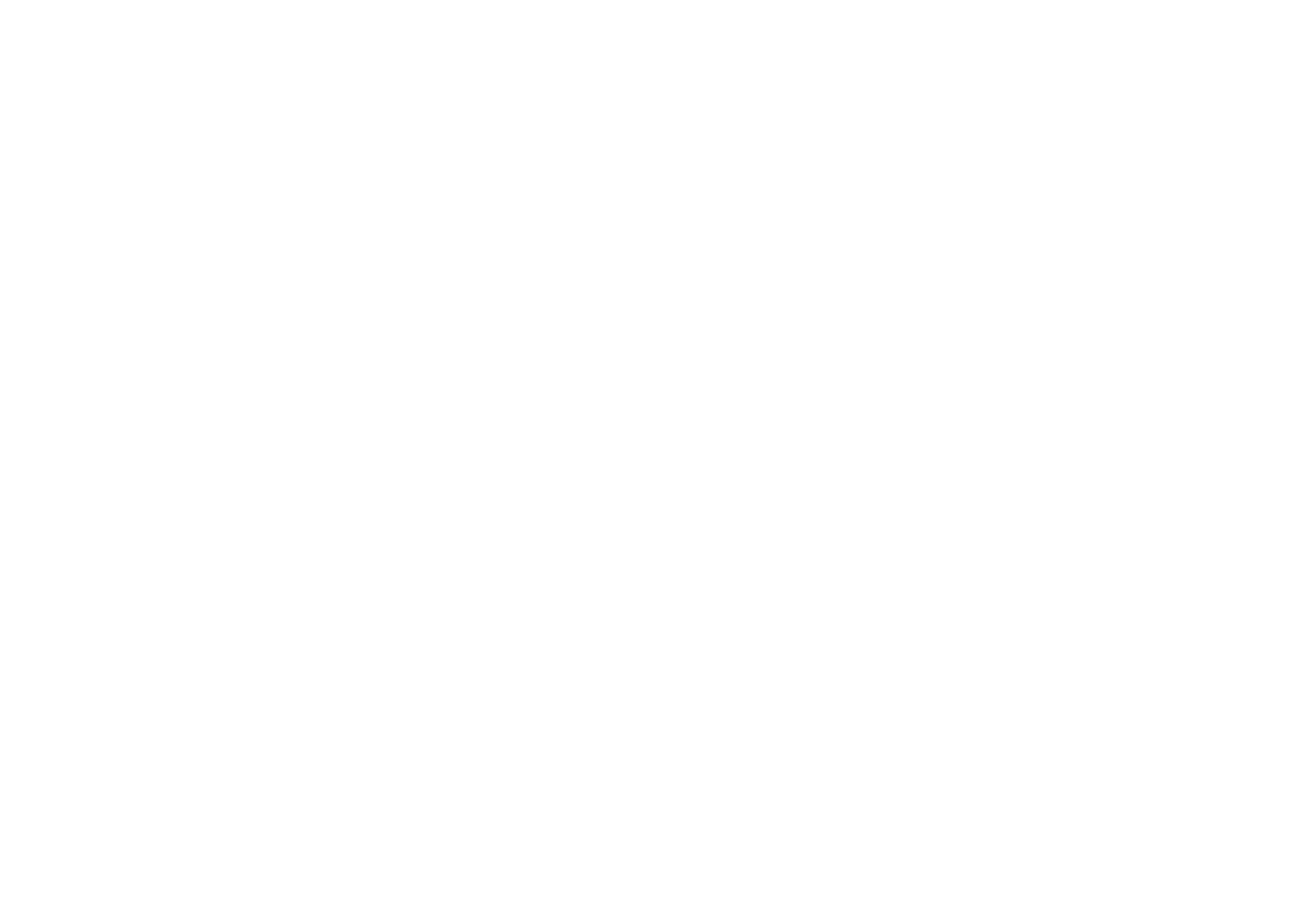
Consider the example above. We have a base class, Pizza, which in turn has some subclasses, each with its own cost. Thus, the getCost() method is overridden.

Now say we want to have the ability to choose any toppings on any pizza. One option would be to create subclasses for all the toppings.



This is clearly an absurd solution.

Another option is to add Boolean values for the toppings directly to the Pizza class.



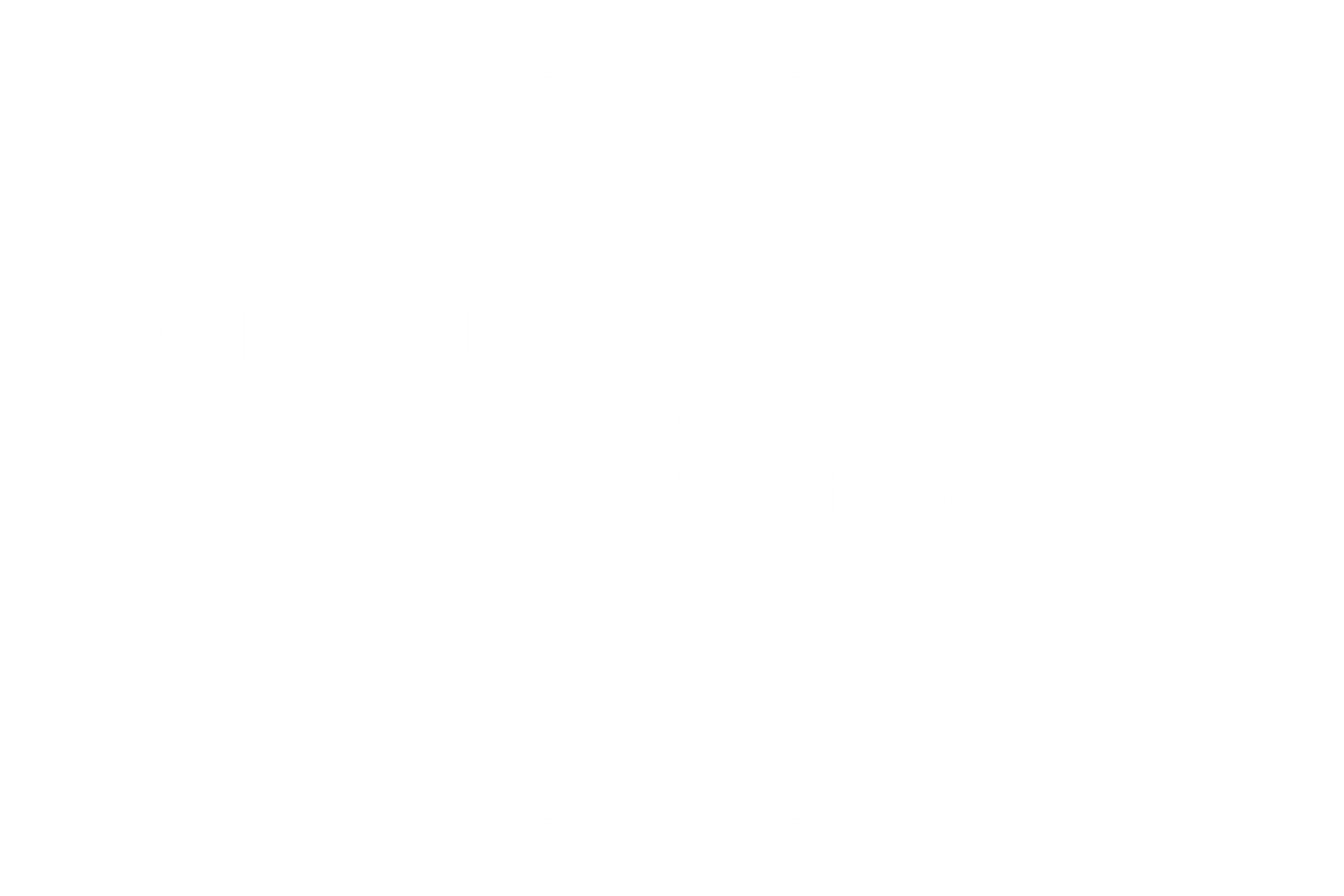
This seems to work for now, but has multiple problems:

* If we add more variables, such as crust types or sizes, even this will become too difficult to manage.
* Any new toppings or changes to topping prices will force us to change code within the superclass, which violates OCP.
* The subclasses inherit all the toppings, even though some toppings may not be appropriate, which violates ISP.

The **decorator pattern** attaches additional responsibilities to an object **dynamically**. Decorators provide flexible alternatives to subclassing for extending functionality.

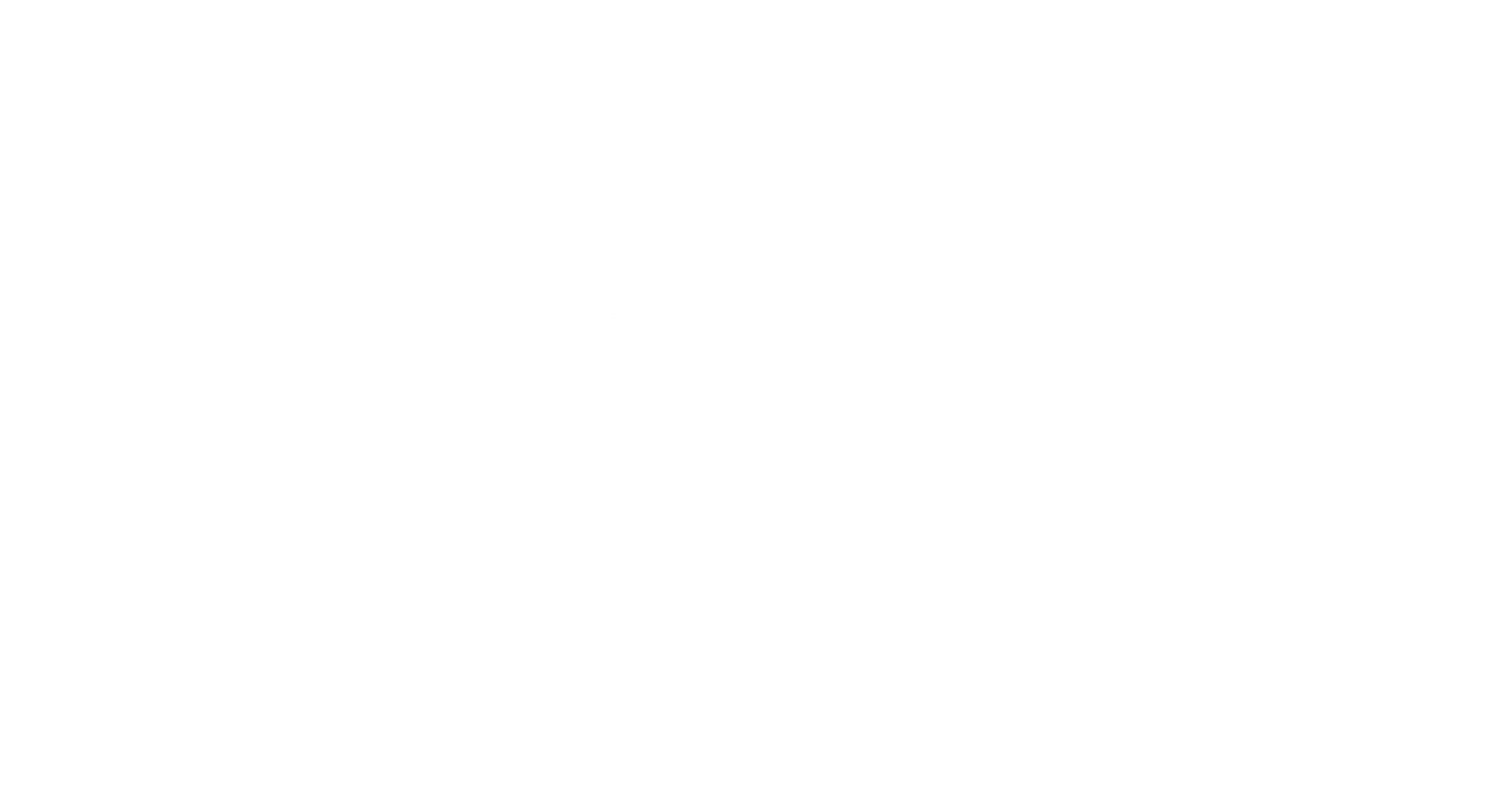
The important thing to remember is that a decorator **is a** base object and it also **has a** base object. Basically, say we have a Pizza object to start with. This object is an abstract object, so the concrete implementation could be one of the basic pizza types.

We also have a **decorator**, which will have a Pizza object inside of it. Say the decorator is Capsicum.



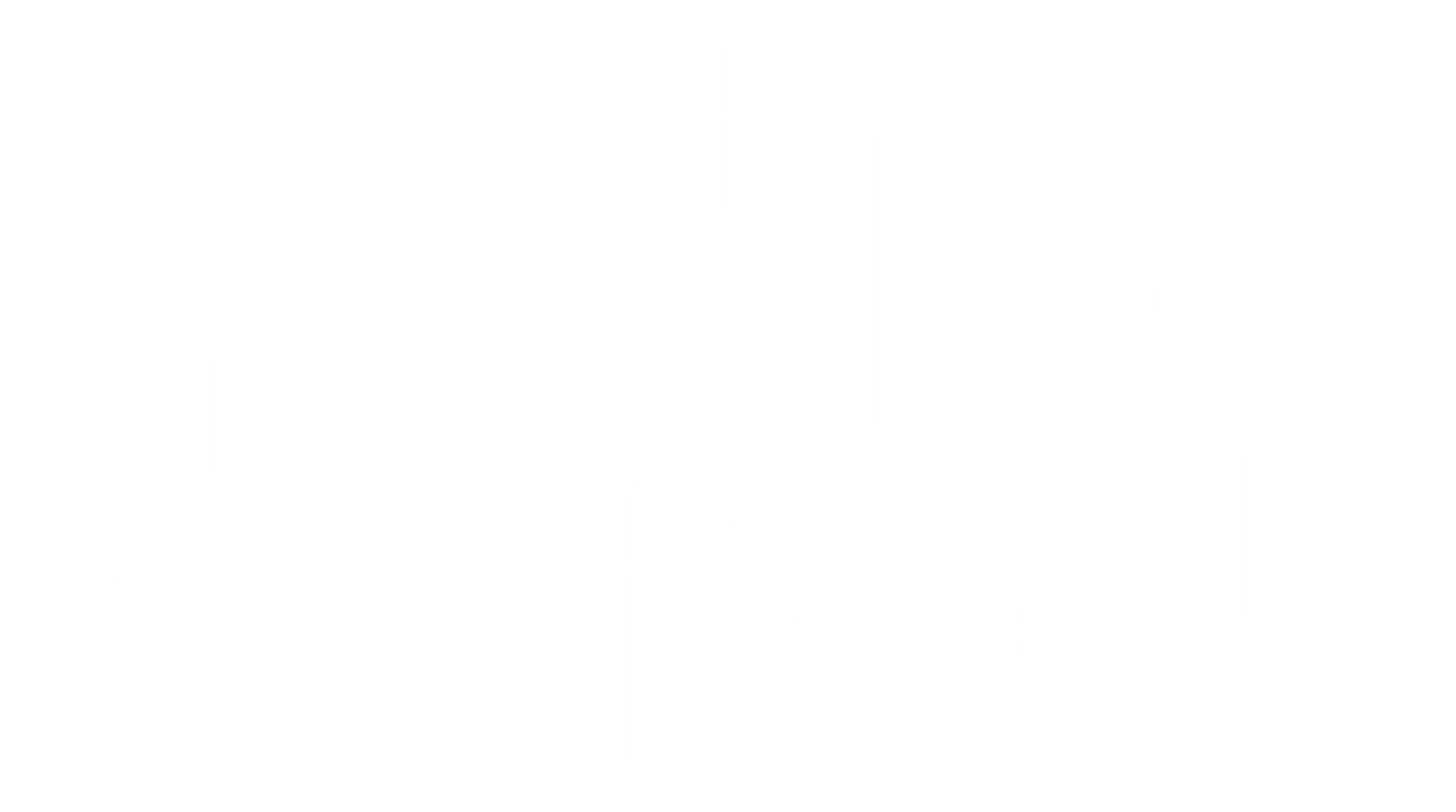
Thus, when we call the getCost() method of the Capsicum object, it actually asks the Pizza object for its cost and then adds its own cost to it. This is possible because Capsicum **has a** Pizza.

Now say we also want another decorator, Cheese. This decorator will actually wrap the **previous decorator** instead of the base class directly. This is possible because again, Capsicum **is a** Pizza. The Cheese object actually has no clue that the thing it is wrapping is not actually a Pizza object, which allows us to use **multiple decorators** to wrap an object.



Now we will initially call the getCost() method of Cheese, which will call the getCost() method of Capsicum which will call the getCost() method of Pizza. Notice that **recursion** becomes involved here.

### General Architecture



From the diagram above, we can see that we have an **abstract** Component class, from which both Concrete Component classes and Decorator classes inherit. However, the Decorator classes themselves are abstract and Concrete Decorator classes inherit from them.

The Component thus serves as the **common interface** for both wrappers and wrapped objects.

The Concrete Components are the objects being wrapped. They define the basic behaviour that will be altered by decorators. They are the objects we will dynamically decorate.

The Decorator class has a field which references the wrapped object, the object from the Concrete Component class. Thus, we can add decorators at **runtime**, we can literally just set the value of this field to a different object, be it one of the original components or a decorator pretending to be a component.

### Advantages

* The decorator pattern makes it possible to **extend the functionality** of a certain object **at runtime**.
* The decorator pattern is an **alternative** **to subclassing**. Subclassing adds behaviour at compile time and forces **all objects** to inherit that behaviour, regardless of whether they use it. Decorating allows new behaviour to be added at runtime for **individual objects**.

### Difference Between Decorator and Strategy Patterns

In the **strategy pattern**, we essentially create an object using different parts. For example, if we have different sorting algorithms available, we can make a decision at runtime based on our data to pick one of them.

In the **decorator pattern**, we take an existing object and add extra features to it. We do not change the existing object, as we do in the strategy pattern.