## Strategy Pattern

### Definition

A behavioural software design pattern that enables selecting an algorithm at runtime. Instead of implementing a single algorithm directly, code receives run-time instructions as to which in a family of algorithms to use.

In a Strategy pattern, a class behaviour can be changed at run time. This type of design pattern comes under behaviour pattern.

It relies in dependency injection, but is a more specific case than that define under SOLID.

There are some good videos showing how to build examples, but this link is to a nice simple clean explanation.

**https://www.tutorialspoint.com/design\_pattern/strategy\_pattern.htm**

### Meaning

Inheritance doesn’t always work well. If you have a base method you can override this method in subclasses, but each class that overrides it will have to have the code re-written with the code the subclass needs. This could easily mean copying and pasting method code to different subclasses. The problem is that if this code needs changing, the code for all subclasses will also need updating or if some of the code gets altered, it becomes very difficult to manage.

The strategy patten allows you to get around this by replacing the code executed within the base class with other external ‘action’ classes. That way, a number of different action classes can be written and managed for the various subclasses and inserted at run time.

When a method is called on the base class, it actually runs the code of the method and so the code with ‘action’ class. As far as I can tell its almost identical to Injection, but its done in the method rather than the constructor.

### Example

If your super class AirborneMachinery has a method called StartEngine(), we might write this in the base class as something that starts a normally aspirated spitfire engine. If the subclass happens to be another normally aspirated plane, we can then we may be able to use that base class method again.

If our subclass is a JetPlane, the StartEngine () code in the base class won’t be applicable, but inheritance allows us to override the StartEngine() method and provide specific code in the subclass to start the turbines instead.

However a TurboProp is also type of plane that also uses turbines, but we might not know if it works in exactly the same way as a JetPlane, so we have choices. We could create a class called TurboProp, inherit from AirborneMachinery and paste in the StartEngine() method from the JetPlane, we could inherit from JetPlane and use the JetPlanes code.

You could than start adding in helicopters, some of which also use turbine engines TurbineHelicopter and again we could inherit from AirborneMachinery copy the code manually, inherit from jetEngine keep its code, we could override the inherited code etc. Bear in mind that all the naming as to be correct down the class hierarchy e.g. If we use JatPlane to inherit from is a helicopter a type of JetPlane?

So very quickly we are in is a mess, some code is repeated, some is not and some of the inheritance naming doesn’t work. If the way a Turbine works changes, or there was an issue with our original code in JetPlane, then every class that uses the copy paste code must be manually changed. Inheritance soon becomes un manageable, especially when other subclasses of JetPlane, are not actually JetPlanes but Helicopters..

So, instead if the base class executing its own StartEngine() code we make the StartEngine () method execute the start engine code from any external class that implements the code to do what we actually need it to do. To do this we write an external class that signs up to IEngine and provides the necessary functionality by providing its own version of StartEngine ().

We then use IEngine to create a variable within the base class, \_engine and we can then pass in any class from above that implements IEngine into it. Maybe using a base method called SetEngine() .

We can create new classes JetPlaneEngineEngine, TurboPropEngine, HelicopterTurbineEngine, infact any that Implement to IEngine and agree to provide a StartEngine() method.

This time, when we create an instance of a Turbo Prop plane we can pass the correct engine into in to the base class the new class by using SetEngine(), if we have a helcipter, we can pass in the required helicopter engine.

In every case, when we call the base class mthod, it exectures the StartEngine method from the allocated engine.

This sideways approach is called composition and provides an alternative to inheritance. We can now write individual chunks of code and test them. We can pick and choose from the library of tested code and as we have different requirements pass in the version we want to use in to the object derived from the base class that suits our requirement.