## Command Pattern

### Definition

Where an object is used to encapsulate all information needed to perform an action or trigger an event at a later time. This information includes the method name, the object that owns the method and values for the method parameters.

### Meaning

If we have a bunch of different classes, all with their own methods and we need to execute those methods in a particular order at some future point in time, we can create an individual instruction for each method we want to call and add the instructions to a list. When the time comes we can execute the list and the correct methods will execute in the correct order, we can define that order to be whatever we need it to be. In the example, the order each method is executed in, is the order it was added into the list.

There are other ways of doing this, for instance creating a class that instantiates each class inside it and then calling the method needed in the correct order. The main problem with this is that is it very inflexible. Using the Command Patten means we can use whatever mechanism we like for defining our classes on the fly, we can add them in to the list, we can change the order they are executed, we can remove them and importantly we can add in completely non related tasks. In the example, most of the code relates to different instances of a car object, but we can add a kettle into the process as well without causing any issues.

### Example

I have broken the example into 3 main elements

**Object classes** : these are the classes with methods we need to trigger to do something

**Command Classes** : these are small classes that when constructed get a reference to an object from the object classes. The Command classes implement ICommand which stipulates each class should have a Execute() method. Its here were we call the corresponding Object class method.

**Broker Class** : The broker class accepts a list of comment classes and allows us to manage the list e.g. add to, remove, change order etc. It will accept any Object that implements ICommand, which means we can reference whichever type object class we like.

Finally we have the section that creates all the instances of the object, broke and command classes. In this simple example if passes the created command classes into the Broker.

So again..

We have object classes that do something using methods, and the objects could be different types.

We need to use different methods from the different object to complete a process , in our example it’s start car 1, turn off lights in car 2, boil a kettle etc… these are completely different fuctions in different class types but they are still part of the over ll process.

For each method of each class we want to use, we create a command class that implement ICommand and so has an Execute method and accepts an object of a particular type. Each In each control class the Execute() method calls the corresponding method in the class it accepted.

Each of these classes are added to the broker class, the broker class will accept any onject that implements ICommand.

When we want to execute the commands, the broker classes calls the execute() method on each of the Command classes in its list, which in turn calls the method of the class we need to execute.