***Exploring and Using Creational, Structural and Behavioural Design Patterns***

**By George Haughton**

This report explores the three different types of design patterns that software engineers will look at and choose from while working in software engineering. There are three main design patterns within software development and they are Creational, Structural and Behavioural. Within my report I am going to explore the different examples of each pattern and explain what they do, how they do it, and if they are a good option to choose when developing.

**Creational**

Within software engineering, Creational design patterns tend to deal with object creation mechanism where they will create objects that suit the situation at hand. The easiest from of object creating to understand will normally result in an added complexity to the design or problems with the design. Creational design patterns try to solve these problems by trying to control the object creation.

Factory design pattern. This method is to create objects of several related classes without having to specifically state the exact object which is being created. This method makes the design a little more complicated but with this it also makes the design more customizable, other methods will require you to make new classes whereas with this method you only need to create a new operation. An intent of the Factory method is to define an interface for creating objects, but when it comes to the instantiate of the class, it lets the subclasses decide. Additionally, it will refer to the new created object using a common interface.

Prototype design pattern. This method allows an object to create several customizable objects without knowing any details about to create them or even which class it belongs to, which is similar to the Factory method, but with this method the palette of prototypical objects can contain more than one object whereas the Factory method never contains more than one. The intent of this method is to specify the kind of objects to create, using a prototypical instance, additionally, it is used to create new objects by copying the prototype.

Builder design pattern. This method allows a client object to construct a complex object just by stating its content and type, while being shielded from the details about the objects representation. By doing it this way it allows the construction process to be used to create different representations. The logic of this process is hidden from the actual steps used when creating the complex objects, because of this the process can be used again and again to create a different object from the same set of objects as the first.

I am going to choose the factory design pattern to go into more detail about. This method is one of the most used and I will need to explore it in full to make sure I am prepared when it comes to developing in the future. Adding on from my paragraph before Factory is a design method in which the goal is to create objects of several related classes without having to state the exact object that is being created. Additionally, one of the main reason it is an important pattern is the fact that it introduces a separation between the application and the family of classes, it introduces a weak coupling instead of the normal tight coupling which hides the concrete classes from the application; additionally, when the developer/user needs to extend the family of products they have all they need to do is make minor changes to the application code. Finally, a few disadvantages of the factory method are the fact it can make your code harder to read because it is behind an abstraction which may hide abstractions. Also, a client may have to sub-class the creator class just to be able to create a concrete product object. Lastly, if any changes are made to any underlining detail of one factory, then the interface may need to be altered within all factories.

**Structural**

Within software engineering, structural design patterns are used as a way easing the design by identifying a simple way to identify relationships between entities. Additionally, they grant solutions and efficient standards to do with class compositions and object structures; they also rely on the process of inheritance and interfaces to allow multiple objects or classes to work together and form a single working whole.

Adapt design pattern. This pattern is used to adapt one interface to another. This pattern will allow programmers to bridge together two unrelated and occasionally two completely incompatible interfaces. A real-world example of this would be, consider a USB to ethernet adapter, we need to use this when we have an ethernet interface on one end and a USB on the other, since they are incompatible we need to use an adapter to covert one to be able to connect to the other. It can also wrap an existing class with a new interface; and can impedance match an old component to a new system. Additionally, this pattern helps to achieve reusability and flexibility.

Filter design pattern. This pattern is used when we would need to filter through sets of objects with different custom criteria to each other. Additionally, it builds a criteria to filter items or objects dramatically, which is why it is referred to as the filter pattern. We can chain criteria when dealing with an even narrower filter, which is done by using a decoupled approach. A good thing about using the filter design pattern is the fact it increases flexibility as the generic common components can be applied and removed which in turn improves flexibility.

Composite design pattern. This pattern describes a group of objects that are treated the same way or a similar way to a single instance of the same type of object. The intent of this design pattern is to compose objects into a tree structure to represent part-whole hierarchies. This allows you to ask each node within the tree to perform a certain task. This is normally done by the class that owns the group of objects and provides a set of protocols/methods to treat them all equally just as if they were one object.

I am now going to speak about the adapt design pattern in more detail. The main concept of adapt is combining two interfaces with each other that wouldn’t normally fit with each other. It does this by introducing an additional adapter class between an interface and an existing class. It implements the expected interface and will keep a reference to an object of the class you would like to reuse. Additionally, this method is also defined by an interface which calls one or more methods on the referred to object and returns a value of the same type. Some disadvantages of this design pattern are the fact that all requests are forwarded and because of this there is a slight increase in the overhead. Additionally, many adaptions are required along with an adapter chain to reach the type of which is required. Lastly, a few advantages of the adapt design method are a client class is not complicated by having to use a different interface and it can use polymorphism to switch between different implementation adapters.

**Behavioural**

Within software engineering a behavioural design pattern is a pattern that identifies common communication patterns between objects and identify these patterns, after doing this these patterns increase flexibility in communication. They are concerned with algorithms and the assignment of responsibilities between several objects.

Observer design pattern. This patter is used to define one-to-many dependency between objects, so when one of these objects changes its state all the other objects are notified and updated with the new state automatically. Additionally, it is used to encapsulate the core, engine and common components within a subject abstraction and the variable, optional and user interface components with an observer hierarchy. A good point about this design pattern is the fact that there is no need to modify the subject, add or remover observers; making it easier to use and a less time-consuming pattern.

State design pattern. This pattern is used to allow an object to alter one’s behaviour when its state changes, the object will then change within its class. Additionally, it can be used as an object-orientated state machine which is a fundamental object and is used for example in an automate factory and control over automotive. When we must change the behaviour of an object because of its state, we can use a state variable within the object and use an if-else condition block to perform different actions based on one’s state.

Chain of responsibility design pattern. This pattern is used to accomplish loose coupling within software design where a request from a client is passed along to a change of objects to process them. The objects will then decide which one will process the request and whether the request should be sent to the next object in the chain or can the first object deal with the request alone. A scenario where this pattern should be used is when you would like to give a request to one of the several objects within the chain but do not want to specify the receiver explicitly.

I will now go into more detail about the observer design pattern. The main intent of the observer design pattern is to define a one-to-many dependency to make sure when one objects state changes the other objects will be told and they will update automatically. This pattern provides a loosely coupled design between objects that interact with each other. Objects that are loosely coupled have changing requirements and are very flexible. Loosely coupling within this scenario means the interacting object won’t have that much information about each other within side them. Some advantages of this pattern are there is no need to modify the subject, add or removed observers, you can reuse the subject and observer classes independently. A disadvantage of this pattern is the fact it can cause memory leaks because of explicit register and unregistering observers.