21. Creating a family of related algorithms

Imagine that you need to create a family of related algorithms. Each of them will have the same parameter types, and the same output object type, but will be composed of completely different internal logic.

You can just create many similar methods, but design patterns allow you to achieve your goal in a better way.

Suitable design patterns

Template Method

Template Method is, perhaps, the simplest way of achieving the goal of defining a family of related algorithms.

In a nutshell, Template Method is just a fairly standard usage of a standard object-oriented feature of inheritance. You define either a skeleton method without implementation or a method with a default algorithm implementation in your base class. And then, you just inherit from this class and modify the algorithm steps as needed.

There is a caveat though. This design pattern is only suitable for very similar algorithms. Otherwise, if you alter an algorithm behavior significantly, you may be violating the Liskov substitution principle.

Why would use Template Method

- The easiest way of creating several related algorithms.
- You can get the clients to only override specific parts of the default algorithm.
- You can store the bulk of your code in the base class.

Visitor

Visitor is a design pattern that allows you to separate algorithms from the objects they operate on. So, you can add new behavior to an object without having to change the object itself.

Visitor is a class that operates on an object but is separate from it. Therefore, because your Visitor is separate from your object, you can define a number of different Visitors if you want to apply different algorithms to your object.

In order for a Visitor to work, a visitor class needs to be able to "visit" the object. Then it can just call and modify the values of any accessible members of the objects. The only components that the Visitor will not have access to are private variables and methods.

Why would you want to use Visitor

- Separating algorithms from the objects they operate on, which enforces the single responsibility principle.
- Can add any new behavior to an object without having to change the object.
- Can have different sets of behaviors that can be applied to objects.
- Visitor class can accumulate a lot of useful information about an object by visiting it.

State

State is a design pattern that makes an object change its behavior when its internal state changes.

For example, think of your mobile phone. If your phone is in a locked state, a particular button unlocks it. However, when the phone is in an unlocked state, the behavior of the same button would be different. Perhaps, it would take you to the homepage.

This design pattern can also include state transition logic, which would be able to undo and redo actions. This is how the State can be used in conjunction with Memento.

Why would use State

- Separating object behaviors from the object, which would enforce the single responsibility principle.
- Allows you to implement a state transition logic.
- Allows you to introduce new behaviors to an object easily without violating the open-closed principle.

Strategy

Strategy, as described in **chapter 6** was intended to be used inside a specific set of conditional logic, so, by definition, it is also a way of defining a series of related algorithms.

After all, all strategy objects within the same set will implement exactly the same interface, making this design pattern a suitable solution for this specific problem.