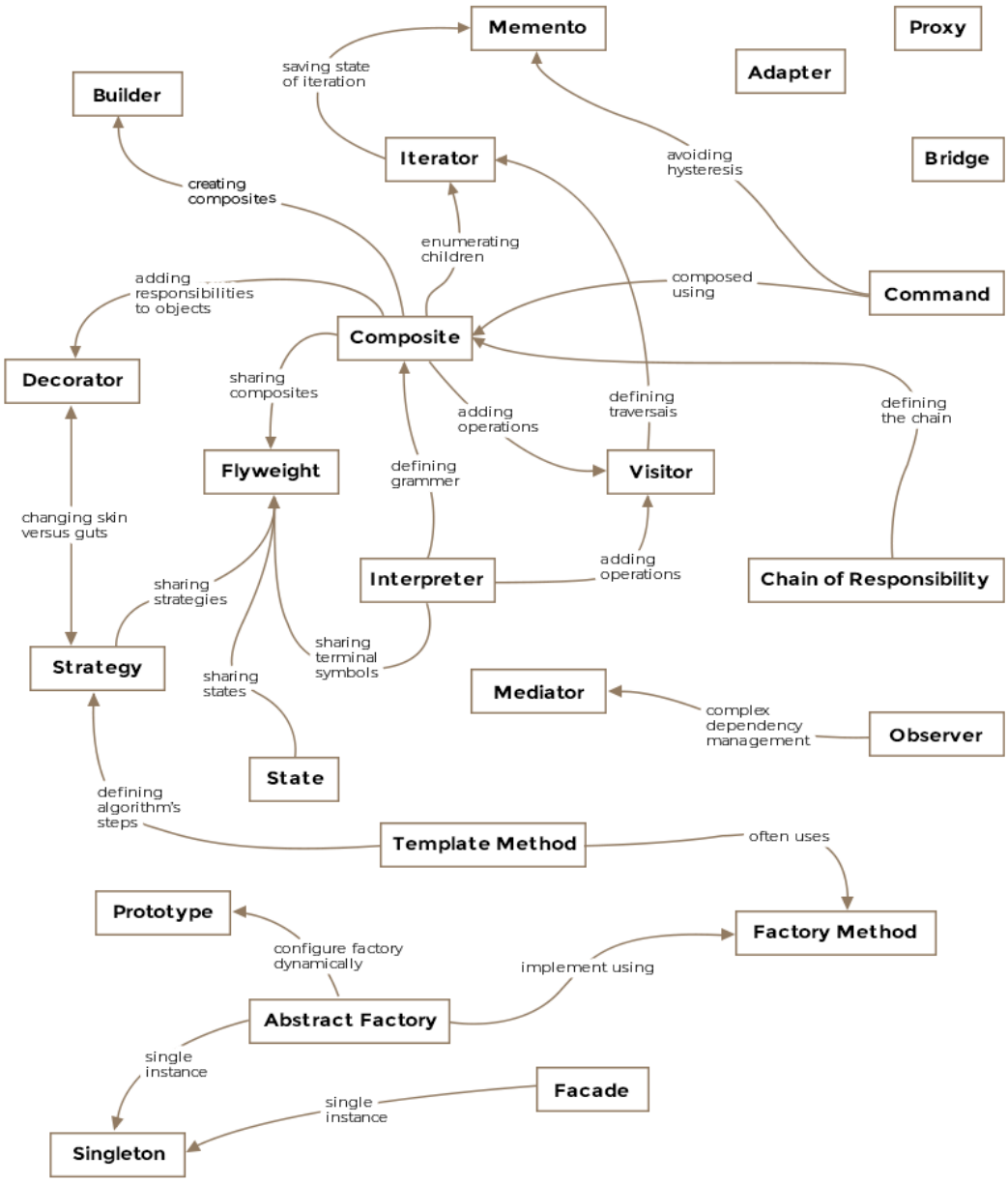
Introduction

This lesson lays down the groundwork for understanding design patterns

Why Patterns ?

Why do we need patterns? The blunt answer is ***we don't want to reinvent the wheel!*** Problems that occur frequently enough in tech life usually have well-defined solutions, which are flexible, modular and more understandable. These solutions when abstracted away from the tactical details become design patterns. If you experienced a déjà vu feeling when designing a solution for a problem that felt eerily similar to the solution of a previous problem, albeit in a different domain, then you were probably using a pattern unknowingly.

Below is an image showing the relationship among the various design patterns as explained by the seminal design patterns work done by the gang of four.



Design Pattern Relationships

Example

Let's consider an example to understand what design patterns are and how do they get applied. The class constructor is one of the basic concepts in an object orientated language. The constructors help create objects of the class and can take in parameters. Let us take the following class as an example.

```
public class Aircraft {


    private String type;

    public Aircraft(String type) {
        this.type = type;
    }
}
```

In the above example, we have the default constructor for the class that takes in a single parameter the `type` of the aircraft. Now say after a few days, you realize you want to add additional properties to your `Aircraft` class. Say you want to add the color of the aircraft as a property, but you have already released a version of your library and can't modify the original constructor. The solution is to add another constructor with two parameters like so

```
public class Aircraft {

    private String type;
    private String color;

    public Aircraft(String type) {
        this.type = type;
    }

    public Aircraft(String type, String color) {
        this.type = type;
        this.color = color;
    }
}
```

If you continue this way you'll end up having a bunch of constructors with increasing number of arguments looking like a telescope:

```


Aircraft(String type)
Aircraft(String type, String color)
Aircraft(String type, String color, String prop3)
Aircraft(String type, String color, String prop3, String prop4)


```

The telescoping pattern is called an ***anti-pattern: how NOT to do things!*** The way to approach a class with an increasing number of variables is to use the **Builder Pattern** that we'll discuss in depth in the following chapters.

Seasoned developers are expected to be well-versed in design patterns and applying them makes the code reusable and maintainable for future. Design patterns aren't limited to object orientated languages but also exist for other domains of Computer Science such as distributed systems, big data system or user interface.

Suggestions for Object Orientated Design

Whenever writing code in an object orientated language, sticking to the following list of suggestions will make your code amenable to changes with the least effort.

- Separate out parts of code that vary or change from those that remain the same.
- Always code to an interface and not against a concrete implementation.
- Encapsulate behaviors as much as possible.
- Favor composition over inheritance. Inheritance can result in explosion of classes and also sometimes the base class is fitted with new functionality that isn't applicable to some of its derived classes.
- Interacting components within a system should be as loosely coupled as possible.
- Ideally, class design should inhibit modification and encourage extension.
- Using patterns in your day to day work, allows exchanging entire implementation concepts with other developers via shared pattern vocabulary.

Some of the above suggestions are embodied in the patterns we'll be discussing in the upcoming lessons. However, remember that making one's design flexible and extensible correspondingly increases the complexity and understandability of the code base. One must walk a fine line between the two competing objectives when designing and writing software.