**OO Principles**

* Encapsulate what varies.
* Favor composition over inheritance
* Program to interfaces, not implementations
* Strive for loosely coupled designs between objects that interact.

Be careful when choosing the areas of the code that need to be extended. Applying the Open-Closed principle everywhere is wasteful, unnecessary, and can lead to complex, hard to understand code.

Dependency inversion

* No variable should hold a reference to a concrete class
* No class should derive from a concrete class
* No method should override an implemented method of any of its base classes.

**Design patterns**

Goal – take the parts that vary and encapsulate them, so that later you can alter or extend the parts that vary without affecting those that don’t.

All patterns provide a way to let some part of the system vary independently of all other parts.

Program to an interface, not an implementation.

Favor composition over inheritance.

**OO Patterns**

**Observer**

The observer pattern defines a one-to-many dependency between object so that when one object changes state, all of its dependents are notified and updated automatically.

There are a few ways to implement the Observer Pattern but most revolve around a class design that includes Subject and Observer interfaces.

* The observer pattern defines a one-to-many relationship between objects
* Subject, or as we also know them, Observables, update Observers using a common interface.
* Observers are loosely coupled in that the Observable knows nothing about them, other than that they implement Observer interface.
* You can push or pull data, pull is considered more “correct”
* Don’t depend on specific order of notification for your observers.

**Decorator**

The Decorator Pattern attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

* The Decorator pattern provides an alternative to subclassing for extending behavior.
* This pattern involves a set of decorator classes that are used to wrap concrete components.
* Decorator class mirror the type of the components they decorate. In fact, they are the same type as the components they decorate, either through inheritance or interface implementation
* Decorators change the behavior of their components by adding new functionality before and/or after method calls to the component.
* You can wrap a component with any number of decorators.
* Decorators can result in many small objects in our design, and overuse can be complex.

**The Factory Method Pattern**

Defines an interface for creating an object, but lets subclasses decide which class to instantiate. Factory method lets a class defer instantiation to subclasses.

Simple factory vs Factory method

Think of Simple Factory as a one shot deal, while with Factory Method you are creating a framework that let’s the subclasses decide which implementation will be used.

With simple factory, it’s just a class that encapsulates the creation of object. With factory method, we have Factory interface with “create Obj” method, concrete factories decide what objects will be made.

Abstract factory allows to substitute factories to get different behaviors.

**The Abstract Factory Pattern**

Provides an interface for creating families of related or dependent objects without specifying their concrete classes.

Factory methods are a natural way to implement your product methods in your abstract factories.

The main difference between a factory method and abstract factory is that the factory methods is a single method, and an abstract factory is an object.

The factory method is just a method, it can be overridden in a subclass, whereas the abstract factory is an object that has multiple factory methods on it.

About both creational patterns:

* All factories encapsulate object creation
* Simple factory is not a design pattern, just a simple way to decouple your clients fr