**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.

Principle of least knowledge

* Talk only to your immediate friends.

**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 creational patterns:

* All factories encapsulate object creation
* Simple factory is not a design pattern, just a simple way to decouple your clients from concrete classes.
* Factory method relies on inheritance: object creation is delegated to subclasses which implement the factory method to create objects.
* Abstract factory relies on object composition. Object creation is implemented in methods exposed in the factory interface. Has multiple object creations in interface.
* All factory patterns promote loose coupling by reducing the dependency of your application on concrete classes.
* The intent of Factory Method is to allow a class to defer instantiation to its subclasses.
* The intent of Abstract factory is to create families of related object without having to depend on concrete classes.
* Dependency inversion guides us to avoid dependencies on concrete types and to strive for abstractions.

**Singelton**

Ensures a clas has only one instance, and provides a global point of access to it.

**Command**

Encapsulates a request as an object, thereby letting you parametrize other objects with different requests, queue or log requests, and support undoable operations.

* Decouples an object, making a request from the one that knows how to perform it.
* Invoker makes a request of command object by calling “Execute()”, which invokes those actions on the receiver.

**Adapter**

Converts the interface of a class into another interface the clients expect. Adapter lets classes work together that couldn’t otherwise because of incompatible interfaces.

Adapter vs Decorator. Adapter intent is to make use of new libraries for old positions without changing code. Decorator intent is to add new functionality. But both of them are just wrappers. Facade although, is for simplifying interface. Wrapper that simplifies more tasks.

**Façade**

Provides a unified interface to a set of interfaces in a subsystem. Façade defines a higher-level interface that makes the subsystem easier to use.

**Template**

Defines the skeleton of an algorithm in a method, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm’s structure.

**State**

Allows an object to alter its behavior when its internal state changes. The object will appear to change it’s class. Pattern encapsulates state into separate classes and delegates to the object representing the current state.