**Homework Week 3 Bryan Andres Almeida Flores**

**## App Architecture Assignment**

Write a small excerpt on each of the above architectural patterns and discuss how they are different.  
Give a list of reasons why to use one over the other.

**Model-View-Controller (MVC)**

The Model-View-Controller is a pattern of software architecture that allows to separate the business logic from the logic of the view. This pattern seeks to enhance the ease of maintenance and reuse of code and the separation of responsibilities in the application. It is fundamentally based on the separation of these responsibilities into three layers:

1. Model: represents the business logic.
2. View: is the representation of data in interfaces.
3. Controller: is the intermediary between the model and the view. Responds to user actions, modifying the model when necessary. It communicates with the view so that it is updated with the latest changes of the model.

This model is used in a very large number of projects the problem is usually that all its code can be written in the same view and usually has many lines of code that can be more difficult to maintain

**Model-View-ViewModel (MVVM)**

Is a model architecture pattern that, like many others, allows to decouple the logic of the application of the logic from the view of the application. This division of responsibilities aims to simplify the development tasks to make it possible to have several developers working on each part of the code, as well as a greater ease of reuse of the code and maintenance of the code.

• Model: represents the business logic. It must be a simple code structure (class or struct) that defines the properties of an object and has no functionality.

1. View Controller (Vista): it must not have logic it is only responsible for modifying the elements of the view according to what the View Model orders. Includes the interfaces.
2. View Model: is the intermediary between the Model and the View, contains the presentation logic.

**Model-View-Presenter (MVP)** is an architecture pattern that separates objects into three distinct types mainly made up of Models, Views and Presenter. MVP was invented to facilitate unit testing and to improve separation of concerns in presentation logic.

Each of the components in MVP play an important role:

* Model holds application data.
* View delegate user interaction to presenter.
* Presenter retrieves data and format it for display in View.

The presenter has no dependences to UIKit and there is a reference between View and Presenter where they both know about each other existence.

**View-Interactor-Presenter-Entity-Routing** **(VIPER)** A typical VIPER Module consists of five components: Here is a short description of what each component does:

* View is passive and doesn't do much on its own. Its sole responsibility is to message events to the presenter and display UI elements.
* Interactor **is** a UIKit-independent component that performs all business logic. For example, in our app, it uses Services to communicate with the REST API in order to obtain image data from an URL.
* Presenter is also UIKit-independent. It receives messages from the view and decides whether to send messages to the interactor or the router. It also receives the data from the interactor and prepares it for the view to display in the suitable format.
* Entity is a plain model that’s used by the interactor
* Router,in our app, is responsible for creating a particular module and navigating from one module to another.

**## Design Patterns Assignment**

Write a small paragraph about different design patterns. (1 paragraph per) Particularly Singleton, Factory, Facade, Decorator, and two others of your choice. Explain what they are and why/when they should be used.

Exist 3 types of patron’s Creational patterns, Structural patterns, Behavioural patterns

**Singleton**: is a creational design pattern that allows us to ensure that a class has a single instance, while providing a global access point to that instance. We'll use the Singleton pattern when a class in your program needs to have only one instance available to all customers, for example, a single database object shared by different parts of the program.

The Singleton pattern disables all other ways to create objects in a class, except for the special creation method. This method creates a new object or returns an existing object if it has already been created. We use the Singleton pattern when we need tighter control of global variables.

**Factory Method** is a creational design pattern that provides an interface for creating objects in a superclass, while allowing subclasses to alter the type of objects to be created by using this method when dependencies are not known in advance,

We use Factory Method when you want to offer users of your library or framework a way to extend its internal components.

Use the Factory Method when you want to save system resources by reusing existing objects instead of rebuilding them every time.

**Facade** is a structural design pattern that provides a simple interface to a complex class library, framework, or system. It is used when the code must deal with multiple objects from a complex library or framework. You must initialize all these objects, track the correct order of dependencies, and so on. As a result, the business logic of your classes is intertwined with the implementation details of other classes. This code is difficult to read and maintain. The **Facade** pattern provides a simple interface for working with complex subsystems that contain many classes. This simplified interface provides only the functions that a client needs and hides all the others.

We will use this pattern when,

## You have to provide a simple or unified interface to a complex subsystem.

## Need to decompose a subsystem into separate layers.

## Decorator is a structural design pattern that allows you to dynamically attach new functionalities to an object by wrapping them **in** useful wrappers. No wonder this design pattern is also called the Wrapper design pattern. This name more accurately describes the central idea behind this pattern: it places a target object inside another container object that triggers the basic behavior of the target object and adds its own behavior to the result.

Both objects share the same interface, so the user doesn't care which of the objects they interact with: clean or wrapped. You can use multiple containers simultaneously and get the combined behavior of all these containers.

## The Decorator pattern is used when,

1. When you want to dynamically add responsibilities to objects and hide those objects from the code that uses them.
2. When it is impossible to extend the responsibilities of an object through inheritance.

**Builde****r** is a creation design pattern that allows you to create complex objects from simple objects step by step. This design pattern helps you use the same code to create different object views. The Builder design pattern requires separating the construction of an object from its own class. Instead, the construction of this object is assigned to special objects called constructors and divided into several steps. To create an object, successively call the constructor methods.

You must apply the Builder design pattern when...

1. When you want to avoid using a telescopic constructor (when a constructor has too many parameters, it becomes difficult to read and manage).
2. When your code needs to create different views of a specific object.

* When you need to compose complex objects.

## Adapter is a structural design pattern that allows objects with incompatible interfaces to work together. In other words, it transforms the interface of an object to fit it to a different object. An adapter wraps around an object, completely hiding it from another object. For example, you can wrap an object that handles meters with an adapter that converts data into inches or centimeters.

## You should use the adapter design pattern when...

1. When you want to use a third-party class, but its interface does not match the rest of your application's code.
2. When you need to use several existing subclasses, but they lack a particular functionality and, in addition, you cannot extend the superclass.