Apple's MVC Pattern

Apple's MVC architectural pattern can be broken down into 3 different types of objects:

- Model objects
- View objects
- Controller objects

This is relevant because many Cocoa technologies and architectures are based on MVC and require that the application's custom objects play one of the MVC roles.

The model objects can have to-one and to-many relationships with other model objects, and so sometimes the model layer on an application effectively is on or more object graphs. Much of the data that is part of the persistent state of the application should reside in the model objects after the data is loaded into the application. Ideally, a model object should have no explicit connection to the view objects. This might be one of the most significant differences between traditional MVC and Apple's MVC pattern, since traditional MVC allows, and actually encourages, direct communication between the model and the view objects.

A view object is an object in an application that users can see. A. view object knows how to draw itself and can respond to user actions. Both the UIKit and AppKit frameworks provide collections of view classes, and interface Builder offers dozens of view objects in its Library.

A controller object acts as an intermediary between one or more of an application's view objects and one or more of its model objects.

In summary, the way Apple's MVC works is that the view objects learn about changes in model data through the application's controller objects and communicate user-initiated changes, for example, text entered in a text field, through controller objects to an application's model objects.

The pros of this pattern, according to Apple's documentation, is that many objects in their corresponding applications tend to be more reusable, and their interfaces tend to be better defined. Applications having an MVC design are also more easily extensible than other applications.

The cons of MVC are that the distinction between view and controller objects, or controller and model objects can sometimes be hard to define or ambiguous. Also, this pattern can lead to overloading in some classes, especially the Controller classes, with all the functionality of an application, which is bad practice because that can prevent us from applying good OOP practices such as encapsulation and modularity. Furthermore, the MVC pattern can make applications exceedingly difficult to maintain when they become bigger and more complex.