Model-View-Controller

The Model-View-Controller design pattern (MVC) is quite old. Variations of it have been around at least since the early days of Smalltalk. It is a high-level pattern in that it concerns itself with the global architecture of an application and classifies objects according to the general roles they play in an application. It is also a compound pattern in that it comprises several, more elemental patterns.

Object-oriented programs benefit in several ways by adapting the MVC design pattern for their designs. Many objects in these programs tend to be more reusable and their interfaces tend to be better defined. The programs overall are more adaptable to changing requirements—in other words, they are more easily extensible than programs that are not based on MVC. Moreover, many technologies and architectures in Cocoa—such as bindings, the document architecture, and scriptability—are based on MVC and require that your custom objects play one of the roles defined by MVC.

Roles and Relationships of MVC Objects

The MVC design pattern considers there to be three types of objects: model objects, view objects, and controller objects. The MVC pattern defines the roles that these types of objects play in the application and their lines of communication. When designing an application, a major step is choosing—or creating custom classes for—objects that fall into one of these three groups. Each of the three types of objects is separated from the others by abstract boundaries and communicates with objects of the other types across those boundaries.

Model Objects Encapsulate Data and Basic Behaviors

Model objects represent special knowledge and expertise. They hold an application’s data and define the logic that manipulates that data. A well-designed MVC application has all its important data encapsulated in model objects. Any data that is part of the persistent state of the application (whether that persistent state is stored in files or databases) should reside in the model objects once the data is loaded into the application. Because they represent knowledge and expertise related to a specific problem domain, they tend to be reusable.

Ideally, a model object has no explicit connection to the user interface used to present and edit it. For example, if you have a model object that represents a person (say you are writing an address book), you might want to store a birthdate. That’s a good thing to store in your Person model object. However, storing a date format string or other information on how that date is to be presented is probably better off somewhere else.

In practice, this separation is not always the best thing, and there is some room for flexibility here, but in general a model object should not be concerned with interface and presentation issues. One example where a bit of an exception is reasonable is a drawing application that has model objects that represent the graphics displayed. It makes sense for the graphic objects to know how to draw themselves because the main reason for their existence is to define a visual thing. But even in this case, the graphic objects should not rely on living in a particular view or any view at all, and they should not be in charge of knowing when to draw themselves. They should be asked to draw themselves by the view object that wants to present them.

View Objects Present Information to the User

A view object knows how to display, and might allow users to edit, the data from the application’s model. The view should not be responsible for storing the data it is displaying. (This does not mean the view never actually stores data it’s displaying, of course. A view can cache data or do similar tricks for performance reasons). A view object can be in charge of displaying just one part of a model object, or a whole model object, or even many different model objects. Views come in many different varieties.

View objects tend to be reusable and configurable, and they provide consistency between applications. In Cocoa, the AppKit framework defines a large number of view objects and provides many of them in the Interface Builder library. By reusing the AppKit’s view objects, such as NSButton objects, you guarantee that buttons in your application behave just like buttons in any other Cocoa application, assuring a high level of consistency in appearance and behavior across applications.

A view should ensure it is displaying the model correctly. Consequently, it usually needs to know about changes to the model. Because model objects should not be tied to specific view objects, they need a generic way of indicating that they have changed.

Controller Objects Tie the Model to the View

A controller object acts as the intermediary between the application's view objects and its model objects. Controllers are often in charge of making sure the views have access to the model objects they need to display and act as the conduit through which views learn about changes to the model. Controller objects can also perform set-up and coordinating tasks for an application and manage the life cycles of other objects.

In a typical Cocoa MVC design, when users enter a value or indicate a choice through a view object, that value or choice is communicated to a controller object. The controller object might interpret the user input in some application-specific way and then either may tell a model object what to do with this input—for example, "add a new value" or "delete the current record"—or it may have the model object reflect a changed value in one of its properties. Based on this same user input, some controller objects might also tell a view object to change an aspect of its appearance or behavior, such as telling a button to disable itself. Conversely, when a model object changes—say, a new data source is accessed—the model object usually communicates that change to a controller object, which then requests one or more view objects to update themselves accordingly.

Controller objects can be either reusable or nonreusable, depending on their general type. [Types of Cocoa Controller Objects](https://developer.apple.com/library/ios/documentation/General/Conceptual/CocoaEncyclopedia/Model-View-Controller/Model-View-Controller.html#//apple_ref/doc/uid/TP40010810-CH14-SW7) describes the different types of controller objects in Cocoa.

Combining Roles

One can merge the MVC roles played by an object, making an object, for example, fulfill both the controller and view roles—in which case, it would be called a *view controller*. In the same way, you can also have model-controller objects. For some applications, combining roles like this is an acceptable design.

A *model controller* is a controller that concerns itself mostly with the model layer. It “owns” the model; its primary responsibilities are to manage the model and communicate with view objects. Action methods that apply to the model as a whole are typically implemented in a model controller. The document architecture provides a number of these methods for you; for example, an NSDocument object (which is a central part of the document architecture) automatically handles action methods related to saving files.

A view controller is a controller that concerns itself mostly with the view layer. It “owns” the interface (the views); its primary responsibilities are to manage the interface and communicate with the model. Action methods concerned with data displayed in a view are typically implemented in a view controller. An NSWindowControllerobject (also part of the document architecture) is an example of a view controller.

[Design Guidelines for MVC Applications](https://developer.apple.com/library/ios/documentation/General/Conceptual/CocoaEncyclopedia/Model-View-Controller/Model-View-Controller.html#//apple_ref/doc/uid/TP40010810-CH14-SW14) offers some design advice concerning objects with merged MVC roles.

**Further Reading:** *Document-Based Applications Overview* discusses the distinction between a model controller and a view controller from another perspective.

Types of Cocoa Controller Objects

[Controller Objects Tie the Model to the View](https://developer.apple.com/library/ios/documentation/General/Conceptual/CocoaEncyclopedia/Model-View-Controller/Model-View-Controller.html#//apple_ref/doc/uid/TP40010810-CH14-SW21) sketches the abstract outline of a controller object, but in practice the picture is far more complex. In Cocoa there are two general kinds of controller objects: mediating controllers and coordinating controllers. Each kind of controller object is associated with a different set of classes and each provides a different range of behaviors.

A *mediating controller* is typically an object that inherits from the NSControllerclass. Mediating controller objects are used in the Cocoa bindings technology. They facilitate—or mediate—the flow of data between view objects and model objects.

**iOS Note:** AppKit implements the NSController class and its subclasses. These classes and the bindings technology are not available in iOS.

Mediating controllers are typically ready-made objects that you drag from the Interface Builder library. You can configure these objects to establish the bindings between properties of view objects and properties of the controller object, and then between those controller properties and specific properties of a model object. As a result, when users change a value displayed in a view object, the new value is automatically communicated to a model object for storage—via the mediating controller; and when a property of a model changes its value, that change is communicated to a view for display. The abstract NSController class and its concrete subclasses—NSObjectController, NSArrayController, NSUserDefaultsController, and NSTreeController—provide supporting features such as the ability to commit and discard changes and the management of selections and placeholder values.

A *coordinating controller* is typically an NSWindowController or NSDocumentControllerobject (available only in AppKit), or an instance of a custom subclass of[NSObject](https://developer.apple.com/library/ios/documentation/Cocoa/Reference/Foundation/Classes/NSObject_Class/index.html#//apple_ref/occ/cl/NSObject). Its role in an application is to oversee—or coordinate—the functioning of the entire application or of part of the application, such as the objects unarchived from a nib file. A coordinating controller provides services such as:

* Responding to delegation messages and observing notifications
* Responding to action messages
* Managing the life cycle of owned objects (for example, releasing them at the proper time)
* Establishing connections between objects and performing other set-up tasks

NSWindowController and NSDocumentController are classes that are part of the Cocoa architecture for document-based applications. Instances of these classes provide default implementations for several of the services listed above, and you can create subclasses of them to implement more application-specific behavior. You can even use NSWindowController objects to manage windows in an application that is not based on the document architecture.

A coordinating controller frequently owns the objects archived in a nib file. As File’s Owner, the coordinating controller is external to the objects in the nib file and manages those objects. These owned objects include mediating controllers as well as window objects and view objects. See [MVC as a Compound Design Pattern](https://developer.apple.com/library/ios/documentation/General/Conceptual/CocoaEncyclopedia/Model-View-Controller/Model-View-Controller.html#//apple_ref/doc/uid/TP40010810-CH14-SW9) for more on coordinating controllers as File's Owner.

Instances of custom NSObject subclasses can be entirely suitable as coordinating controllers. These kinds of controller objects combine both mediating and coordinating functions. For their mediating behavior, they make use of mechanisms such as target-action, outlets, delegation, and notifications to facilitate the movement of data between view objects and model objects. They tend to contain a lot of glue code and, because that code is exclusively application-specific, they are the least reusable kind of object in an application.

**Further Reading:** For more on the Cocoa bindings technology, see *Cocoa Bindings Programming Topics*.

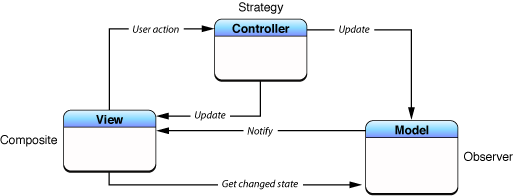
MVC as a Compound Design Pattern

Model-View-Controller is a design pattern that is composed of several more basic design patterns. These basic patterns work together to define the functional separation and paths of communication that are characteristic of an MVC application. However, the traditional notion of MVC assigns a set of basic patterns different from those that Cocoa assigns. The difference primarily lies in the roles given to the controller and view objects of an application.

In the original (Smalltalk) conception, MVC is made up of the Composite, Strategy, and Observer patterns.

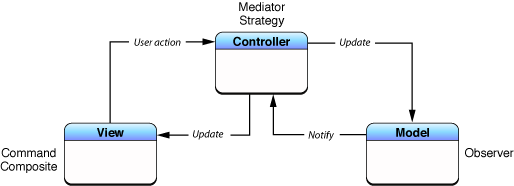
* Composite—The view objects in an application are actually a composite of nested views that work together in a coordinated fashion (that is, the view hierarchy). These display components range from a window to compound views, such as a table view, to individual views, such as buttons. User input and display can take place at any level of the composite structure.
* Strategy—A controller object implements the strategy for one or more view objects. The view object confines itself to maintaining its visual aspects, and it delegates to the controller all decisions about the application-specific meaning of the interface behavior.
* Observer—A model object keeps interested objects in an application—usually view objects—advised of changes in its state.

The traditional way the Composite, Strategy, and Observer patterns work together is depicted by Figure 7-1: The user manipulates a view at some level of the composite structure and, as a result, an event is generated. A controller object receives the event and interprets it in an application-specific way—that is, it applies a strategy. This strategy can be to request (via message) a model object to change its state or to request a view object (at some level of the composite structure) to change its behavior or appearance. The model object, in turn, notifies all objects who have registered as observers when its state changes; if the observer is a view object, it may update its appearance accordingly.

**Figure 7-1**  Traditional version of MVC as a compound pattern

The Cocoa version of MVC as a compound pattern has some similarities to the traditional version, and in fact it is quite possible to construct a working application based on the diagram in Figure 7-1. By using the bindings technology, you can easily create a Cocoa MVC application whose views directly observe model objects to receive notifications of state changes. However, there is a theoretical problem with this design. View objects and model objects should be the most reusable objects in an application. View objects represent the "look and feel" of an operating system and the applications that system supports; consistency in appearance and behavior is essential, and that requires highly reusable objects. Model objects by definition encapsulate the data associated with a problem domain and perform operations on that data. Design-wise, it's best to keep model and view objects separate from each other, because that enhances their reusability.

In most Cocoa applications, notifications of state changes in model objects are communicated to view objects *through* controller objects. Figure 7-2 shows this different configuration, which appears much cleaner despite the involvement of two more basic design patterns.

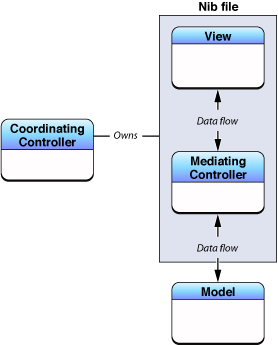
**Figure 7-2**  Cocoa version of MVC as a compound design pattern

The controller object in this compound design pattern incorporates the Mediator pattern as well as the Strategy pattern; it mediates the flow of data between model and view objects in both directions. Changes in model state are communicated to view objects through the controller objects of an application. In addition, view objects incorporate the Command pattern through their implementation of the target-action mechanism.

**Note:** The target-action mechanism, which enables view objects to communicate user input and choices, can be implemented in both coordinating and mediating controller objects. However, the design of the mechanism differs in each controller type. For coordinating controllers, you connect the view object to its target (the controller object) in Interface Builder and specify an action selector that must conform to a certain signature. Coordinating controllers, by virtue of being delegates of windows and the global application object, can also be in the responder chain. The bindings mechanism used by mediating controllers also connects view objects to targets and allows action signatures with a variable number of parameters of arbitrary types. Mediating controllers, however, aren’t in the responder chain.

There are practical reasons as well as theoretical ones for the revised compound design pattern depicted in Figure 7-2, especially when it comes to the Mediator design pattern. Mediating controllers derive from concrete subclasses of NSController, and these classes, besides implementing the Mediator pattern, offer many features that applications should take advantage of, such as the management of selections and placeholder values. And if you opt not to use the bindings technology, your view object could use a mechanism such as the Cocoa notification center to receive notifications from a model object. But this would require you to create a custom view subclass to add the knowledge of the notifications posted by the model object.

In a well-designed Cocoa MVC application, coordinating controller objects often own mediating controllers, which are archived in nib files. Figure 7-3 shows the relationships between the two types of controller objects.

**Figure 7-3**  Coordinating controller as the owner of a nib file

Design Guidelines for MVC Applications

The following guidelines apply to Model-View-Controller considerations in the design of applications:

* Although you can use an instance of a custom subclass of [NSObject](https://developer.apple.com/library/ios/documentation/Cocoa/Reference/Foundation/Classes/NSObject_Class/index.html#//apple_ref/occ/cl/NSObject) as a mediating controller, there's no reason to go through all the work required to make it one. Use instead one of the ready-made NSController objects designed for the Cocoa bindings technology; that is, use an instance of NSObjectController,NSArrayController, NSUserDefaultsController, or NSTreeController—or a custom subclass of one of these concrete NSController subclasses.

However, if the application is very simple and you feel more comfortable writing the glue code needed to implement mediating behavior using outlets and target-action, feel free to use an instance of a custom NSObject subclass as a mediating controller. In a custom NSObject subclass, you can also implement a mediating controller in the NSController sense, using key-value coding, key-value observing, and the editor protocols.

* Although you can combine MVC roles in an object, the best overall strategy is to keep the separation between roles. This separation enhances the reusability of objects and the extensibility of the program they're used in. If you are going to merge MVC roles in a class, pick a predominant role for that class and then (for maintenance purposes) use categories in the same implementation file to extend the class to play other roles.
* A goal of a well-designed MVC application should be to use as many objects as possible that are (theoretically, at least) reusable. In particular, view objects and model objects should be highly reusable. (The ready-made mediating controller objects, of course, are reusable.) Application-specific behavior is frequently concentrated as much as possible in controller objects.
* Although it is possible to have views directly observe models to detect changes in state, it is best not to do so. A view object should always go through a mediating controller object to learn about changes in an model object. The reason is two-fold:
  + If you use the bindings mechanism to have view objects directly observe the properties of model objects, you bypass all the advantages that NSController and its subclasses give your application: selection and placeholder management as well as the ability to commit and discard changes.
  + If you don't use the bindings mechanism, you have to subclass an existing view class to add the ability to observe change notifications posted by a model object.
* Strive to limit code dependency in the classes of your application. The greater the dependency a class has on another class, the less reusable it is. Specific recommendations vary by the MVC roles of the two classes involved:
  + A view class shouldn't depend on a model class (although this may be unavoidable with some custom views).
  + A view class shouldn't have to depend on a mediating controller class.
  + A model class shouldn't depend on anything other than other model classes.
  + A mediating controller class shouldn’t depend on a model class (although, like views, this may be necessary if it's a custom controller class).
  + A mediating controller class shouldn't depend on view classes or on coordinating controller classes.
  + A coordinating controller class depends on classes of all MVC role types.
* If Cocoa offers an architecture that solves a programming problem, and this architecture assigns MVC roles to objects of specific types, use that architecture. It will be much easier to put your project together if you do. The document architecture, for example, includes an Xcode project template that configures an NSDocumentobject (per-nib model controller) as File's Owner.

Model-View-Controller in Cocoa (OS X)

The Model-View-Controller design pattern is fundamental to many Cocoa mechanisms and technologies. As a consequence, the importance of using MVC in object-oriented design goes beyond attaining greater reusability and extensibility for your own applications. If your application is to incorporate a Cocoa technology that is MVC-based, your application will work best if its design also follows the MVC pattern. It should be relatively painless to use these technologies if your application has a good MVC separation, but it will take more effort to use such a technology if you don’t have a good separation.

Cocoa in OS X includes the following architectures, mechanisms, and technologies that are based on Model-View-Controller:

* **Document architecture**. In this architecture, a document-based application consists of a controller object for the entire application (NSDocumentController), a controller object for each document window (NSWindowController), and an object that combines controller and model roles for each document (NSDocument).
* **Bindings**. MVC is central to the bindings technology of Cocoa. The concrete subclasses of the abstract NSController provide ready-made controller objects that you can configure to establish bindings between view objects and properly designed model objects.
* **Application scriptability**. When designing an application to make it scriptable, it is essential not only that it follow the MVC design pattern but that your application’s model objects are properly designed. Scripting commands that access application state and request application behavior should usually be sent to model objects or controller objects.
* **Core Data**. The Core Data framework manages graphs of model objects and ensures the persistence of those objects by saving them to (and retrieving them from) a persistent store. Core Data is tightly integrated with the Cocoa bindings technology. The MVC and object modeling design patterns are essential determinants of the Core Data architecture.
* **Undo**. In the undo architecture, model objects once again play a central role. The primitive methods of model objects (which are usually its accessor methods) are often where you implement undo and redo operations. The view and controller objects of an action may also be involved in these operations; for example, you might have such objects give specific titles to the undo and redo menu items, or you might have them undo selections in a text view.