Element	U.S. Patent No. 7,010,536	iOS-Compatible Devices

The delegate that implements these methods can block the impending event (by returning `NO` in the first two methods) or alter a suggested value (the index set and the frame rectangle in the last two methods). It can even defer an impending event; for example, the delegate implementing the `applicationShouldTerminate:` method can delay application termination by returning `NSTerminateLater`.

Other delegation methods are invoked by messages that don't expect a return value and so are typed to return `void`. These messages are purely informational, and the method names often contain "Did", "Will", or some other indication of a transpired or impending event. Listing 5-2 shows a few examples of these kinds of delegation method.

Listing 5-2 Sample delegation methods returning `void`

```
- (void) tableView:(NSTableView*)tableView
    mouseDownInHeaderOfTableColumn:(NSTableColumn *)tableColumn;     // NSTableView
- (void)windowDidMove:(NSNotification *)notification;                // NSWindow
- (void)application:(UIApplication *)application
    willChangeStatusBarFrame:(CGRect)newStatusBarFrame;              //
UIApplication
- (void)applicationWillBecomeActive:(NSNotification *)notification;   //
NSApplication
```

There are a couple of things to note about this last group of methods. The first is that an auxiliary verb of "Will" (as in the third method) does not necessarily mean that a return value is expected. In this case, the event is imminent and cannot be blocked, but the message gives the delegate an opportunity to prepare the program for the event.

The other point of interest concerns the second and last method declarations in Listing 5-2 . The sole parameter of each of these methods is an `NSNotification` object, which means that these methods are invoked as the result of the posting of a particular notification. For example, the `windowDidMove:` method is associated with the `NSWindow` notification `NSWindowDidMoveNotification`. The section "Notifications" (page 227) discusses notifications in detail, but here it's important to understand the relationship of notifications to delegation messages in AppKit. The delegating object automatically makes its delegate an observer of all notifications it posts. All the delegate needs to do is implement the associated method to get the notification.

To make an instance of your custom class the delegate of an AppKit object, simply connect the instance to the `delegate` outlet or property in Interface Builder. Or you can set it programmatically through the delegating object's `setDelegate:` method or `delegate` property, preferably early on, such as in the `awakeFromNib` or `applicationDidFinishLaunching:` method.

Element	U.S. Patent No. 7,010,536	iOS-Compatible Devices
		Delegation and the Cocoa Application Frameworks The delegating object in a Cocoa application is often a responder object such as a `UIApplication`, `NSWindow`, or `NSTableView` object. The delegate object itself is typically, but not necessarily, an object, often a custom object, that controls some part of the application (that is, a coordinating controller object). The following AppKit classes define a delegate: • `NSApplication` • `NSBrowser` • `NSControl` • `NSDrawer` • `NSFontManager` • `NSFontPanel` • `NSMatrix` • `NSOutlineView` • `NSSplitView` • `NSTableView` • `NSTabView` • `NSText` • `NSTextField` • `NSTextView` • `NSWindow` The UIKit framework also uses delegation extensively and always implements it using formal protocols. The application delegate is extremely important in an application running in iOS because it must respond to application-launch, application-quit, low-memory, and other messages from the application object. The application delegate must adopt the `UIApplicationDelegate` protocol. Delegating objects do not (and should not) retain their delegates. However, clients of delegating objects (applications, usually) are responsible for ensuring that their delegates are around to receive delegation messages. To do this, they may have to retain the delegate in memory-managed code. This precaution applies equally to data sources, notification observers, and targets of action messages. Note that in a garbage-collection environment, the reference to the delegate is strong because the retain-cycle problem does not apply.

Element	U.S. Patent No. 7,010,536	iOS-Compatible Devices

		Some AppKit classes have a more restricted type of delegate called a *modal delegate*. Objects of these classes (`NSOpenPanel`, for example) run modal dialogs that invoke a handler method in the designated delegate when the user clicks the dialog's OK button. Modal delegates are limited in scope to the operation of the modal dialog.
		
		The existence of delegates has other programmatic uses. For example, with delegates it is easy for two coordinating controllers in the same program to find and communicate with each other. For example, the object controlling the application overall can find the controller of the application's inspector window (assuming it's the current key window) using code similar to the following:
		
		`id winController = [[NSApp keyWindow] delegate];`
		
		And your code can find the application-controller object—by definition, the delegate of the global application instance—by doing something similar to the following:
		
		`id appController = [NSApp delegate];`
		
		(EV0002204-08.)
		
		Publicly available information indicates that gateways are used throughout the iOS operating system of the iOS-Compatible Devices. The gateways corresponding to the above evidence are only examples. Other examples are described in the Apple iOS Developer Library documentation.
		
		It is believed that the structure and operation of the gateways are more fully set forth in the source code of the iOS operating system, which is not publicly available. Evolutionary Intelligence reserves its right to supplement and/or modify this claim chart after obtaining discovery of this source code.
		
		Evolutionary Intelligence presently contends that this element is literally present in the accused instrumentality. Evolutionary Intelligence reserves its right to contend that this element is satisfied under the doctrine of equivalents because any differences between this claim element and any accused instrumentality are insubstantial and the accused instrumentality preforms substantially the same function, in substantially the same way, to reach substantially the same result.

12	The apparatus of claim 1 or 2, wherein the gateway includes means	Each iOS-Compatible Device infringes claim 1 and claim 2 (*see* above.) In addition, with respect to both claims 1 and 2, the gateway includes means for reporting information, the means for reporting information by providing register information to other containers, systems or processes that interact with

Element	U.S. Patent No. 7,010,536	iOS-Compatible Devices
	for reporting information, the means for reporting information providing register information to other containers, systems or processes that interact with the container.	the container. For example, gateways of the iOS Delegates include means for reporting information by providing register information to other containers, systems or processes that interact with the Delegates: **Delegates and Data Sources** A delegate is an object that acts on behalf of, or in coordination with, another object when that object encounters an event in a program. The delegating object is often a responder object—that is, an object inheriting from `NSResponder` in AppKit or `UIResponder` in UIKit—that is responding to a user event. The delegate is an object that is delegated control of the user interface for that event, or is at least asked to interpret the event in an application-specific manner. To better appreciate the value of delegation, it helps to consider an off-the-shelf Cocoa object such as a text field (an instance of `NSTextField` or `UITextField`) or a table view (an instance of `NSTableView` or `UITableView`). These objects are designed to fulfill a specific role in a generic fashion; a window object in the AppKit framework, for example, responds to mouse manipulations of its controls and handles such things as closing, resizing, and moving the physical window. This restricted and generic behavior necessarily limits what the object can know about how an event affects (or will affect) something elsewhere in the application, especially when the affected behavior is specific to your application. Delegation provides a way for your custom object to communicate application-specific behavior to the off-the-shelf object.

Element	U.S. Patent No. 7,010,536	iOS-Compatible Devices

The programming mechanism of delegation gives objects a chance to coordinate their appearance and state with changes occurring elsewhere in a program, changes usually brought about by user actions. More importantly, delegation makes it possible for one object to alter the behavior of another object without the need to inherit from it. The delegate is almost always one of your custom objects, and by definition it incorporates application-specific logic that the generic and delegating object cannot possibly know itself.

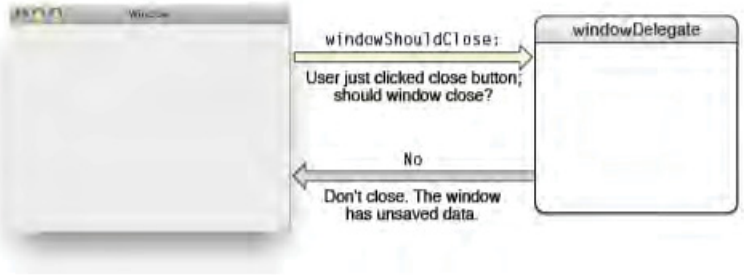
How Delegation Works

The design of the delegation mechanism is simple (Figure 5-1). The delegating class has an outlet or property, usually one that is named `delegate`; if it is an outlet, it includes methods for setting and accessing the value of the outlet. It also declares, without implementing, one or more methods that constitute a formal protocol or an informal protocol. A formal protocol that uses optional methods—a feature of Objective-C 2.0—is the preferred approach, but both kinds of protocols are used by the Cocoa frameworks for delegation.

In the informal protocol approach, the delegating class declares methods on a category of `NSObject`, and the delegate implements only those methods in which it has an interest in coordinating itself with the delegating object or affecting that object's default behavior. If the delegating class declares a formal protocol, the delegate may choose to implement those methods marked optional, but it must implement the required ones.

Delegation follows a common design, illustrated by Figure 5-1.

Figure 5-1 The mechanism of delegation



DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS
Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS
Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS
Sync your system to PACER to automate legal marketing.