Text Input Management

Cocoa > Events & Other Input

2007-02-08

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Simultaneously published in the United States and Canada.

Contents

Introduction to Text Input Management

The text input management system provides Cocoa applications with a wide variety of text input capabilities. This document describes the Cocoa classes related to text input management and explains how to use them.

Who Should Read This Document

Read this document if you want to create a custom input server for your Cocoa application or if you want to implement text input capabilities in a view.

Organization of This Document

The text input management system is described in the following articles:

- "Text Input Management Architecture" (page 7) provides an overview of the text input management system.
- "About Key Bindings" (page 9) explains how the text input management system binds keystrokes to character data.

Detailed information about creating and delivering a custom input server is covered in the following articles:

- "Creating Input Servers" (page 11) explains how to implement an input server.
- "Deploying Input Servers" (page 13) explains how to install an input server.

If you want to create a custom view to handle text input capabilities, read this article:

- "Creating Custom Views" (page 15) explains how to add keyboard input capabilities to a custom view.

See Also

For more information, refer to the following documents:

- *Text Editing Programming Guide for Cocoa* describes the text-editing process, including text-input key event processing through the input management system.
- NSInputServer, NSInputManager, and NSTextView are the primary classes that interact to provide text input management for Cocoa applications.

Text Input Management Architecture

The Cocoa text input management system centers around three classes that interact with each other to transmit input from the user's keyboard or mouse to a text view. An **input server** (NSInputServer or other NSInputServiceProvider object) receives keyboard characters and transmits characters to be inserted into a text view object. An **input manager** (NSInputManager object) serves as a proxy for a particular input server and passes messages to the active input server. Finally, a **text view** (NSTextView or other NSTextInput object) displays, stores, and manages onscreen text.

How Input Management Works

In the simplest case, the text view passes keyboard events to the input manager through the standard message-passing hierarchy. The input manager checks the events against the **key-bindings dictionary**, which maps certain Control-key events to selectors. If a key-bindings entry is found in the dictionary, the selector is sent to the input server. Most events, however, are passed directly to the active input server as strings of text. The input server processes the received text and sends a message asking the originating text view to insert the text, possibly in a modified form.

In more complex scenarios, the input server can do much more than simply send back the text it receives. Every text view tracks a region of **marked text**, a region of text (distinct from the selection, and usually indicated with a yellow-orange highlight color) that is incomplete—the input server can retrieve and modify the marked text depending on subsequent characters. Additionally, input servers can manage palette windows, including one that appears immediately below the insertion point, for complex character sets. These facilities allow the input server to output more characters than it inputs, or vice versa.

Each localization of Mac OS X has a **platform input server**, which serves as the default input server on that system. For example, the U.S. English platform input server transforms Option-E followed by E (two keystrokes) to a single character, an e with an acute accent. (The Option-E appears as marked text until the second keystroke completes the input process.) The Japanese platform input server demonstrates more complex processing, allowing the user to enter hiragana (phonetic) characters, to pick from a list of possible kanji (pictogram) characters that could match the hiragana input in a palette window, and to insert the kanji character into the text view.

Note that all Application Kit event handling, including text input, is not thread-safe. If you are writing a multithreaded application, make sure all event handling takes place in the main thread.

Input Servers

Input servers are objects that implement the NSInputServiceProvider protocol. Input server objects are typically instances of NSInputServer or a subclass, and instances of NSInputServer are typically assigned a delegate to provide custom input management. For information on creating an input server, see "Creating Input Servers" (page 11).

Input Managers

Input managers are instances of the NSInputManager class. You never have to instantiate or subclass NSInputManager, and in general you never have to directly access its methods either. The only place that its methods need to be accessed are within the implementation of a text view that does not inherit from NSTextView.

Text Views

Text views are instances of classes that implement the NSTextInput protocol. The most common example of a text view is an NSTextView object or instance of a subclass of NSTextView.

If you need more sophisticated text handling than NSTextView provides, for example in a word processing application, you may need to create a text view that does not inherit from NSTextView. In this case, you will need to implement the NSTextInput protocol and access NSInputManager methods from within the implementation. For information on creating custom text views, see "Creating Custom Views" (page 15).

About Key Bindings

Input managers use a dictionary property list, called a **key-bindings dictionary**, to interpret keyboard events before passing them to an input server.

During the processing of a keyboard event, the event passes through the NSMenu object, then to the first responder via the `keyDown:` method. The default implementation of the method provided by the NSResponder class propagates the message up the responder chain until an overridden `keyDown:` implementation stops the propagation. Typically, an NSResponder subclass can choose to process certain keys and ignore others (for example, in a game) or to call the `interpretKeyEvents:` method, which passes the event to the current input manager.

The input manager checks the event to see if it matches any of the keystrokes in the user's key-bindings dictionary. A key-bindings dictionary maps a keystroke (including its modifier keys) to a method name. For example, the default key-bindings dictionary maps `^d` (Control-D) to the method name `deleteForward:`. If the keyboard event is in the dictionary, then the input manager calls the input server's `doCommandBySelector:client:` method with the selector associated with the dictionary entry. If the input server's `doCommandBySelector:client:` method doesn't find a matching method, then it passes the command selector onto the text view's `doCommandBySelector:` method, which may or may not find a matching method to call.

If the input manager cannot match the keyboard event to an entry in the key-bindings dictionary, it extracts the string from the event by using its `characters` method and passes the returned characters to the input server's `insertText:client:` method.

The standard key-bindings dictionary is in the file `/System/Library/Frameworks/AppKit.framework/Resources/StandardKeyBinding.dict`. You can override the standard dictionary entirely by providing a dictionary file at the path `~/Library/KeyBindings/DefaultKeyBinding.dict`. However, defining custom key bindings dynamically (that is, while the application is running) is not supported.

Creating Input Servers

This article explains how to implement an input server for the Cocoa text input system. If you want to create an input method that can be accessed from both Cocoa and Carbon applications, use the Text Services Manager in the Carbon framework.

For an example of a custom Cocoa input server, see HexInputServer, located at `/Developer/Examples/AppKit/HexInputServer`.

Setting Up Input Server Classes

To create an input server, you can either subclass NSInputServer or instantiate an NSInputServer object with a delegate. The second choice normally suffices. If you choose to subclass NSInputServer, you usually override most or all of the methods required by the NSInputServiceProvider protocol. If you assign a delegate object, the delegate must implement the NSInputServiceProvider protocol.

Tracking Input Clients

The input management classes identify each text-modification message with a **client** and a **conversation identifier**. Clients are instances of the NSInputManager class associated with applications, and conversation identifiers specify the text views within applications. The input server talks to multiple clients (applications), and each client controls multiple conversations (text views).

Unless your input server processes only single characters, it needs a way to keep track of multiple clients and conversations. A straightforward way to do this is to define a context object class to handle individual input clients, and map clients to context objects with a dictionary. You can convert the addresses of client objects (pointers) to NSNumber objects, and use the resulting NSNumber as a key for the context object. The following C macro converts client addresses to NSNumber keys:

```
#define KEY_FROM_CLIENTID(theClient) ([NSNumber numberWithUnsignedInt:
                                            (unsigned int)theClient])
```

Additionally, within each context object, you need to map conversation numbers (type `long`) to check that input server messages correspond to the current conversation. You can again use a dictionary or other data structure to perform the mapping.

Managing Marked Text

Input servers can manage an active portion of a text view, called the **marked text**, which is visually marked so the user knows it is not final. To set the marked text range, use the input client's `setMarkedText:selectedRange:` method. To retrieve the current marked range, use the `markedRange` method. This code fragment replaces the current marked range with the string `outputBuffer` in the text input client `client` and sets the selected range to the new string:

```
id client;              // Assume this exists and implements NSTextInput.
NSString *outputBuffer; // Assume this exists.

[client setMarkedText:outputBuffer
        selectedRange:NSMakeRange(0,[outputBuffer length]);
```

To unmark the text in a client view, use the NSTextInput `unmarkText` method.

To retrieve the text formatting attributes supported by a client view, use the `validAttributesForMarkedText` method. See the additions to NSAttributedString in the Application Kit for the set of string constants supported by the view.

Tracking Mouse Events

To make an input server receive mouse events—for example, to handle selection—as well as keyboard events, you should implement its `wantsToHandleMouseEvents` to return `YES`. Additionally, to handle mouse events the input server must implement the NSInputServerMouseTracker protocol, which consists of three methods: `mouseDownOnCharacterIndex:atCoordinate:withModifier:client:`, `mouseDraggedOnCharacterIndex:atCoordinate:withModifier:client:`, and `mouseUpOnCharacterIndex:atCoordinate:withModifier:client:`, for mouse-down, mouse-dragged, and mouse-up events, respectively.

Deploying Input Servers

This section explains how to deploy an input server for use with the Cocoa text input management system.

Initializing Input Servers

An input server is implemented as an NSInputServer object wrapped in a server application, which communicates via interprocess communication (IPC) with the current input manager. The input server must be initialized with its delegate (if a delegate is used) and an IPC connection name at runtime. The IPC connection name must match the one in the input server's `Info` file (see "Installing Input Servers" (page 13)).

The easiest way to do this is to perform the initialization in the application's `main()` function—for example in the following code:

```
int main(int argc, const char *argv[])
{
    NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
    [[NSInputServer alloc] initWithDelegate:[HexInputServer sharedInstance]
                          name:@"HexInputServer2ConnectionName"];
    [[NSApplication sharedApplication] run];
    [pool release];
    return 0;
}
```

Note that the HexInputServer class uses a `sharedInstance` method to create a HexInputServer object because only one such object will ever be created.

Installing Input Servers

An input server is installed as a directory hierarchy, which must be installed in `~/Library/InputManagers` or `/Library/InputManagers` to be useful to users. At the top level of the hierarchy there is a text file called `Info`, an application bundle, and optionally a key-bindings dictionary file. The `Info` file contains a dictionary property list. When an application starts up, NSInputManager scans the `InputManagers` directories for input servers and uses the data from each folder's `Info` file to create an input manager for its input server. If there are any custom input servers installed, an Input submenu appears in the Edit menu which lets the user switch between the different installed input servers. When the user chooses an input server, the input manager launches the input server if it's not already running and connects to it via interprocess communication.

The `Info` file tells NSInputManager how to find the application executable, tells what the IPC connection name to use is, provides an optional key-bindings dictionary, and contains localized text strings to represent the input server in the Edit > Input submenu. The file can be an XML property list or an old-style ASCII property list, such as this example:

```
{
    // relative path to the executable to be launched
    ExecutableName = "HexInputServer.app/Contents/MacOS/HexInputServer";

    // The name registered with the IPC mechanism, the same as the one
    // used to initialize the NSInputServer object
    ConnectionName = "HexInputServer2ConnectionName";
    DisplayName = "Hexadecimal Input";
    DefaultKeyBindings = "HexInputServer.dict";

    // The name that will appear in the Edit > Input submenu
    LocalizedNames = { Deutsch = "Hexadezimaler Input"; }

    // Default name used for languages not found in LocalizedNames
    DisplayName = "Hexadecimal Input"

    // Optional. Returned by the current input manager's language: method.
    LanguageName = "English";

    // Optional file to map keyboard characters to selector names
    DefaultKeyBindings = "HexInputServer.dict"
}
```

The `DefaultKeyBindings` file overrides the system's default key-bindings file. (See "About Key Bindings" (page 9) for more information.)

If you want to override the key-bindings mechanism entirely, make your input server's `wantsToInterpretAllKeystrokes` method return `YES`.

Creating Custom Views

This section explains how to use the text input management system when implementing a custom view. This information is relevant to anyone adding keyboard input capabilities to their view that NSTextView does not provide.

Implementing Text Management Support

Custom Cocoa views can provide varying levels of support for the text input management system. There are essentially three levels of support to choose from:

1. Override the `keyDown:` method.
2. Override `keyDown:` and use `interpretKeyEvents:` to support key bindings.
3. Also implement the full NSTextInput protocol.

In the first level of support, the `keyDown:` method recognizes a limited set of events and ignores others. This level of support is typical of games. (When overriding `keyDown:`, you must also override `acceptsFirstResponder` to make your custom view respond to key events, as described in Event Objects and Types.)

In the second level of support, you can override `keyDown:` and use the `interpretKeyEvents:` method to receive key-binding support without implementing the NSTextInput protocol. You then implement the standard key-binding methods that your view wants to support, such as `moveForward:` or `deleteForward:`. (The full list of key-binding methods can be found in `NSResponder.h`.)

If you are writing your own text view from scratch, you should use the third level of support and implement the NSTextInput protocol in addition to overriding `keyDown:` and using `interpretKeyEvents:`. NSTextView and its subclasses are the only classes provided in Cocoa that implement NSTextInput, and if your application needs more complex behavior than NSTextView can provide, as a word processor might, you may need to implement a text view from the ground up. To do this, you must subclass NSView and implement the NSTextInput protocol. A class implementing this protocol—by inheriting from NSTextView or another class, or by implementing the protocol directly—is called a **text view**.

If you are implementing NSTextInput, your view needs to manage marked text and communicate with the input server to support the text input management system. These tasks are described in the next two sections.

Managing Marked Text

One of the primary things that a text view must do to cooperate with an input server is to maintain a (possibly empty) range of **marked text** within its text storage. The text view should highlight text in this range in a distinctive way, and it should allow selection within the marked text. A text view must also maintain an **insertion point**, which is usually at the end of the marked text, but the user can place it within the marked text. The text view also maintains a (possibly empty) **selection** range within its text storage, and if there is any marked text, the selection must be entirely within the marked text.

A common example of marked text appears when a user enters a character with multiple keystrokes, such as "é," in an NSTextView object. To enter this character, the user needs to type Option-E followed by the E key. After pressing Option-E, the accent mark appears in a highlighted box, indicating that the text is marked (not final). After the final E is pressed, the "é" character appears and the highlight disappears.

Communicating With the Input Server

An input server and a text view must cooperate so that the input server can implement its user interface. The bulk of the NSTextInput protocol comprises methods called by an input server to manipulate text within the text view for the input server's user interface purposes.

A text view must also inform the current input manager when its marked text range changes or a mouse event happens. The text view accomplishes this by calling the `markedTextSelectionChanged:client:`, `markedTextAbandoned:`, and `handleMouseEvent:` methods with the current input manager.

If the text view receives a mouse event within its text area, and a marked text range exists, it must call the current input manager's `wantsToHandleMouseEvents`. If `wantsToHandleMouseEvents` returns `YES`, it must forward those mouse events to the input manager's `handleMouseEvent:` method. If it returns `NO`, it is up to the text view to determine what to do with the event.

If the input server returns `NO` to both `wantsToHandleMouseEvents` and `handleMouseEvent:`, and the text view decides to change selection within the marked range, it must notify the current input manager of the change with the `markedTextSelectionChanged:client:` method. In general, however, the input server is responsible for managing the selection within the marked text.

Also, the text view must call the current input manager's `markedTextAbandoned:` method when the insertion point leaves the marked text or whenever it decides to clear the marked range.

The current input server generally uses all of the methods in the NSTextInput protocol except for the `hasMarkedText` method. The text view itself may call this method to determine whether there is currently marked text. NSTextView, for example, disables Copy in the Edit menu when `hasMarkedText` returns `YES`.

Document Revision History

This table describes the changes to *Text Input Management*.

Date	Notes
2007-02-08	In "About Key Bindings," added that defining custom key bindings dynamically is not supported. In "Creating Custom Views," added that one must also override acceptsFirstResponder when overriding keyDown:.
2005-01-11	Made editorial revisions throughout.
2004-02-09	Revised introduction and added an index.
2002-11-12	Revision history was added to existing document.

Index

N

P

R

S

T

U

V

W