Xcode 2.0 User Guide

(Legacy)

Tools > Xcode

2006-11-07

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

Simultaneously published in the United States and Canada.

Contents

Figures, Tables, and Listings

Introduction to Xcode 2.0 User Guide

Important: The information in this document is obsolete and should not be used for new development.

Software development can be thought of as a complex problem space in which you manage files to produce products. The types of files can include source files, resource files, and supporting files (documentation, timelines, notes, or any other files that help you build the software but aren't part of the product). You use various tools to process the files into a variety of outputs. To automate the process and keep track of all the details and interactions, you use an IDE.

The Xcode IDE is designed to help you work in this type of problem space. It allows you to perform most tasks quite simply, using its basic user interface. Many features should be familiar to most developers. Xcode is Apple's tool suite and integrated development environment (IDE) for creating Mac OS X software. The Xcode application includes a full-featured code editor, a debugger, compilers, and a linker. The Xcode application provides a user interface to many industry-standard and open-source tools, including GCC, javac, jikes, and GDB. It provides all of the facilities you need to build a program for Mac OS X, whether it's an application, kernel extension, or command-line tool.

This document describes the Xcode application and how you can use it to develop software for Mac OS X. It provides a comprehensive guide to Xcode's features and user interface. This document is intended for developers using Xcode to build software for Mac OS X. This document is written for Xcode 2.0.

Organization of This Document

This document contains several parts, each of which contains chapters devoted to a major functional area of the Xcode application. These parts are:

- "Developing a Software Product With Xcode" (page 27) describes the development process and how Xcode helps you with each step along the way.
- "Projects" (page 41) introduces the Xcode project and its primary components, and covers important project management concepts. The chapters in this part show you how to create an Xcode project, add and manage project files, organize project items, and modify project attributes. They describe the project window and other important Xcode user interface conventions; as well as mechanisms for finding information in your Xcode project, including documentation lookup, project-wide searches, and the class browser.
- "Design Tools" (page 121) describes the class modeling and data modeling design tools included in Xcode. The chapters in this part describe common user interface features of these two tools, demonstrate how to model classes in your application, and describe how to create a schema for use with the Core Data framework.
- "Editing Source Files" (page 159) describes Xcode's source code editor. The chapters in this part describe the user interface for Xcode's built-in editor, and show you how to use features such as code completion, text macros, and the navigation bar to quickly author source code and navigate source code files. They also discuss how to use an external application to edit project files.

- "Version Control" (page 201) discusses the version control systems supported by the Xcode application. The chapters in this part show you how to configure a version control system in Xcode and how to perform common version control tasks, such as updating files, committing changes, and comparing file revisions.
- "The Build System" (page 229) describes Xcode's build system and how to use Xcode to build a product. The chapters in this section describe targets, build styles, and the other information that Xcode uses to build a product. They also show you how you can customize the build process by adding custom tasks to the build process or change the way a product is built by modifying build settings. This part also includes information on features that you can use to reduce the amount of time it takes to build, such as distributed builds, precompiled prefix headers, and predictive compilation.
- "Debugging" (page 335) describes Xcode's graphical debugger and shows you how to run and debug your program in Xcode. The chapters in this part demonstrate how to use features such as Fix and Continue, which lets you make changes to your program while it is running and continue your debugging session, and remote debugging, which allows you to debug an application running on a remote host.
- "Customizing Xcode" (page 383) describes how you can customize your work environment using scripts, preferences, and custom key bindings sets.

See Also

For an introduction to the developer tools available for Mac OS X, see Getting Started With Tools.

For an introduction to Mac OS X system architecture and system technologies, see *Mac OS X Technology Overview*.

To see a full list of the tools available with Xcode Tools, see Mac OS X Developer Tools in *Mac OS X Technology Overview*.

To learn more about the types of software you can create for Mac OS X, see Software Development Overview in *Mac OS X Technology Overview*.

To learn more about the Mac OS X standard user interface, see *Apple Human Interface Guidelines*.

For a tutorial introduction to Xcode, see *Xcode Quick Tour for Mac OS X*.

For tips on converting Code Warrior projects and other existing code to build in Xcode, see *Porting CodeWarrior Projects to Xcode*.

To learn more about the GNU compiler collection, see *GNU C/C++/Objective-C 3.3 Compiler*.

For more information on debugging with GDB, see *Debugging with GDB*.

For information on using cross-development to develop for multiple versions of Mac OS X, see *Cross-Development Programming Guide*.

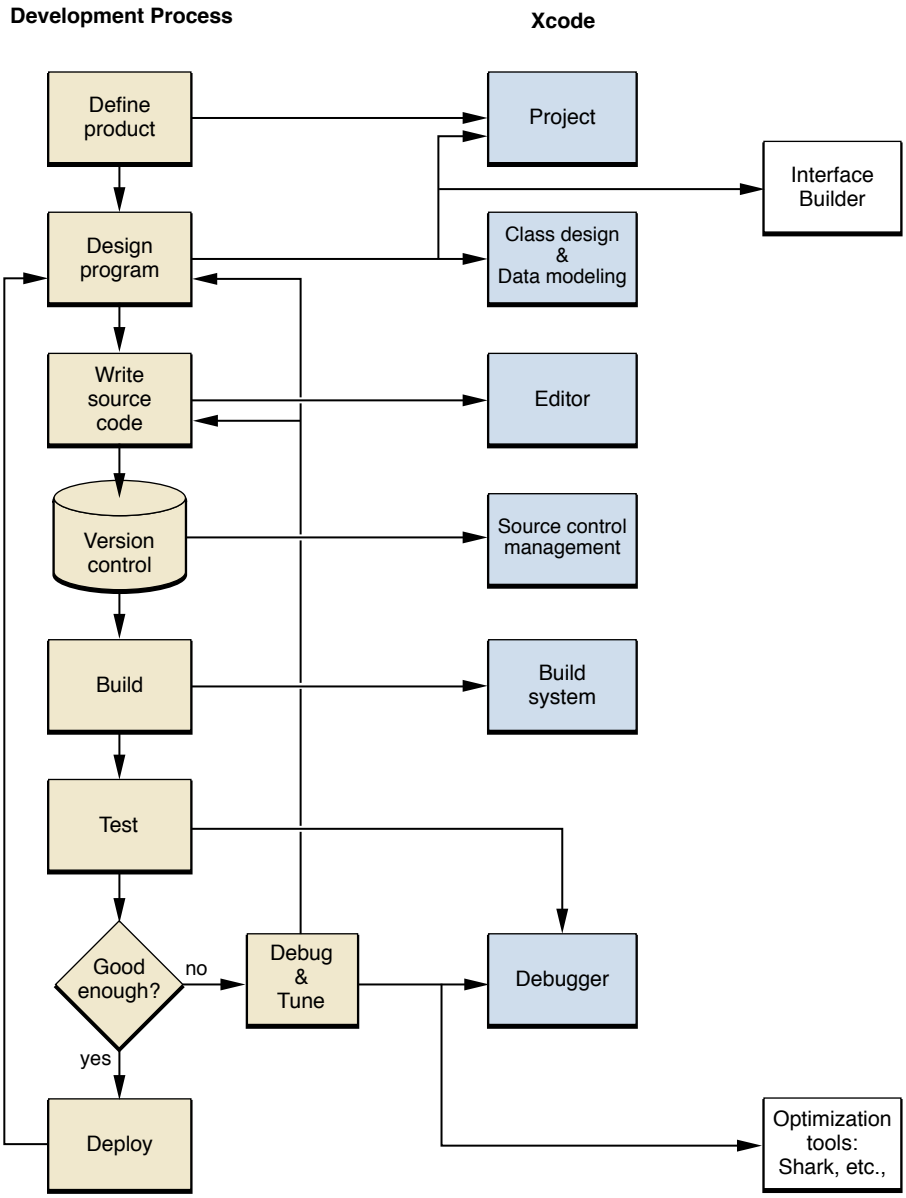
In addition, many other documents are referred to or recommended throughout this document.

Developing a Software Product With Xcode

Xcode can help you at each step in the process of developing a software product. That includes steps such as researching Apple technologies; writing and compiling code; and building, linking, testing, debugging, and optimizing the software for your product. By taking a closer look at these steps, you'll see how Xcode fits into the development process.

This section gives a brief overview of the software development process and how Xcode helps you at each stage in that process. Figure 1-1 shows the typical development process and how it relates to Xcode.

Figure 1-1 Xcode and the software development process



Briefly, the stages of the development process are as follows:

- Define a product. The earliest stages in the creation of an application are conceptual; you decide what problem you are trying to solve and think about the best programmatic approach and the best interface design. You make fundamental decisions about the programming language you will use, the architecture of your application, and the Mac OS X technologies you will use. No matter what your decisions, chances are Xcode has a project template to support them.
- Design your program. The design process doesn't stop with your decisions about programming language, Mac OS X technology, and product type. If your product has a graphical user interface, you can use the Interface Builder application to design the user interface for your application. Xcode knows how to work

with Interface Builder nib files; a nib file contains the resources that describe the various user interface elements in our program's interface. When you double-click a nib file in the Xcode application, the Interface Builder application opens. For more information on Interface Builder, see *Interface Builder Help*.

If you are programming in an object-oriented language, Xcode's design tools let you model classes in your application and entities that represent your data. The class modeling tool lets you understand the classes in your project, whether they're written in Objective-C, C++, Java, or a mixture of those languages. The data modeling tool lets you diagram entities and the relationships between them, and define a schema for use with the Core Data framework. For more on Xcode's design tools, see "Design Tools" (page 121).

- Write source code. Once you have designed your program and user interface, you need to implement your design. Xcode's editor has many features to facilitate your job, including code completion, direct linking to function, class, and method descriptions, support for syntax coloring, and many shortcuts for moving between files. For more on Xcode's editor, see "Editing Source Files" (page 159).
- Choose a version control system. Version control systems let you track changes to your source files. Using version control, several developers can work on the same project at the same time. Xcode works with three version control systems: Concurrent Versions System (CVS), Subversion, and Perforce. To learn more about using version control in Xcode, see "Version Control" (page 201).
- Build your product. To create a program that you can run and test, you must first build the product. Building a product involves many steps, including compiling source files, linking object files, copying resource files, and more. Xcode includes a powerful build system that can build any Mac OS X software product. Xcode's build system provides a user interface to industry-standard tools such as the GNU compiler collection. In addition, Xcode provides numerous opportunities to tailor the build process. To learn more about Xcode's build system, see "The Build System" (page 229).
- Debug and test your program. In most large application projects, debugging and tuning the code proceeds in parallel with implementation. Xcode includes a source-level debugger that provides a graphical user interface to the GNU debugger, GDB. GDB is a command-line debugger for C, Objective-C, C++, and Objective-C++ code. For more information on using GDB, see *Debugging with GDB* and *GDB Quick Reference*.

  The Xcode application lets you step through your code line by line, set and modify breakpoints, view variables, stack frames, and threads, and access GDB directly through a command line.

  In order to enhance your users' perception of your application, you should minimize the application's launch time, execution time, and memory footprint. Xcode Tools includes a number of tools, such as Shark, to help you achieve those goals. You can launch your program with many of these performance tools directly from Xcode. For more information on the performance tools included with Xcode Tools and on performance tuning in general, see *Performance Overview*.
- Deploy your product. The last step in application development is bundling the various object files, frameworks, and data files into a package that can be installed by the user. Xcode automatically packages all of the files that it knows are part of the application. Whenever possible, you should package your application for drag-and-drop installation. When necessary, however, as when you want to let the user install a new version of an application over an old one without replacing all the files, you can use the PackageMaker application to create an installation package. Installer, located in `/Applications/Utilities`, is the native installer for Mac OS X. PackageMaker and Installer are documented in *Software Delivery Guide*.

Defining a Product

As you work to define a software product, you typically draw from a number of sources, such as requirements specifications, existing software products, technology documentation, and your own knowledge of what you need to accomplish. In doing your analysis, you don't want your choices to be restricted by your development environment—rather, you want to have confidence that the IDE supports the product decisions you make.

The following are some of the questions you might ask as you focus on defining a product. In most cases, you'll find that Xcode can accommodate the requirements identified by your answers.

- What kind of product do you want to create: a new application, a plug-in for an existing application, a library to be used by several other products?

  Xcode provides project templates for creating applications, plug-ins, dynamic libraries, kernel extensions, and more. "Creating a Project" (page 47) describes the project templates provided by Xcode. For a description of the various types of software you can develop on Mac OS X, see Software Development Overview in *Mac OS X Technology Overview*.

- Does the product have multiple parts? For example, is it an application with an embedded framework?

  An Xcode project can contain multiple targets, which each build a different product. For particularly large or complex products, you might choose separate projects—and Xcode can manage dependencies between them.

- What Mac OS X technologies are appropriate for your product?

  Xcode documentation contains both conceptual and reference material for a wide range of available technologies. For a good introduction, see *Mac OS X Technology Overview*.

- Do you need to deploy your product on multiple versions of Mac OS X?

  Xcode supports cross-development for different versions of Mac OS X, so that you can develop on one version and deploy to many, taking advantage of features in each version. For example, you can build on Mac OS X version 10.3 (Panther) and target version 10.1 or 10.2, as well as 10.3. To be able to use this feature, you must install SDKs as part of installing Xcode. For detailed information, see *Cross-Development Programming Guide*, in Tools Xcode Documentation.

- Do you need to do some rapid user-interface prototyping, or provide a complex interface for an administrative tool?

  AppleScript Studio lets you use AppleScript (or system languages such as C, Objective-C, or Java) to drive applications with complex user interfaces. For more information, see the learning path "Creating an AppleScript Studio Application" in Getting Started With AppleScript.

- Do you need to work on an existing CodeWarrior or Project Builder project?

  Xcode can import these projects. For details, see "Creating a Project" (page 31).

- Do you have existing code that uses Cocoa or Carbon?

  Cocoa is an object-oriented application environment designed specifically for developing native applications for Mac OS X. Cocoa applications are written in Objective-C or Java, but can also make use of C.

  Carbon is a set of procedural C APIs for developing full-featured, high-performance, applications for Mac OS X.

  Xcode supports development with C, C++, Objective-C, Objective-C++, and Java, and provides project and target templates for many types of products.

- Do you need to provide a cross-platform solution or have existing code that uses Java?

  The Java application environment in Mac OS X provides a development environment, a runtime environment, and an application framework that includes AWT and Swing. For more information, see the documents in Java Documentation, as well as *Mac OS X Technology Overview*.

- Are you working as part of a team? With an existing code base?

  Xcode's source control management (SCM) can manage your code, using CVS, Subversion, or Perforce.

  Team members can use source trees to define commonly named paths that point to different directories on different machines. For example, Xcode uses source trees when you import a CodeWarrior project. (For more on importing, see "Creating a Project" (page 31).)

  Xcode scales well, from small to large projects, so you can divide the work into as many projects and targets as its scope requires. For more information on dividing up large projects, see "Dividing Your Work Into Projects and Targets" (page 83).

Of course these are not the only questions that you may need to resolve in the course of defining your product.

Creating a Project

Once you have made your decisions about the type of product (application, library, command-line tool, and so on) and language or languages (C, C++, Objective-C, Java, and others) you plan to use, you're ready to create a project.

Xcode provides the project as the primary workplace for your software development. When your design has reached the point where it's time to start working on the code, you can do one of the following:

- Create a new project based on one of the templates described in Table 3-1 (page 47).

  The new project contains a default target that is preconfigured to build a product of the type you specified when you chose the project template. Most projects also contain default source files, resource files, framework references, and other items.

- Import an existing Project Builder project by simply opening the project with Xcode.

- Import an existing CodeWarrior project.

  To import a CodeWarrior project, you must have CodeWarrior available on the same computer, and must take a specific import step. After importing, the new project is likely to require some modification before you can successfully build it. You can find detailed information on this process in *Porting CodeWarrior Projects to Xcode*.

- Import an existing PBWO project (a project built with the older Project Builder for WebObjects application). Projects of this type should build with a minimum of additional steps.

Once you've created an Xcode project, you can add files, add targets, modify target settings, and make any required modifications to develop your software.

Project Organization and Navigation

Before you plunge headlong into working with the project that you have just created, it's a good idea to familiarize yourself the many ways in which you can organize and quickly access project information and items. Xcode provides many different ways for you to view, organize, and find information in your project, so you can work efficiently.

Organizing a Project

The Xcode project is the primary mechanism for grouping the files and information you need to build one or a set of related products. Within a project, a target specifies the files and other information needed to build a specific product. In addition, Xcode provides groups, as a way to organize information within a project, and to navigate to project files, symbols, and so on.

Xcode defines groups for source code, targets, errors and warnings, bookmarks and other items. You can also create your own groups to help you organize information in ways that make sense to you. To learn more about groups in Xcode, see "Groups in Xcode" (page 56).

"Organizing Xcode Projects" (page 83) provides a detailed look at the high-level issues involved in organizing complex projects and targets. It includes "Organizing Files" (page 86), which provides a more detailed look at working with groups.

Project Navigation

Xcode provides convenient navigation at several levels, whether you're editing source code in multiple files, looking up technical documentation, building, debugging, or performing other tasks.

The project window, described in "The Project Window" (page 55), offers several options for navigation:

- You can find any project file by first selecting the project group (which reveals a list of all the files in the project), then use filtered searching to find the file you're looking for.
- You can define bookmarks to access specific file locations, then use the Bookmarks group to locate specific bookmarks.
- You can use the Project Symbols group to find a method or function and go directly to its source code.
- You can use the Errors and Warnings group to go directly to the specific line in your code where an error occurred.

To examine the hierarchy of classes defined in object-oriented languages, Xcode provides a class browser. Using the browser, you can navigate to code (both for Apple frameworks and for classes you have defined) and documentation.

Xcode also provides options for jumping between header and implementation files, jumping to methods or functions within a file, or instantly opening a selected or typed filename. And you can locate text by performing single-file or batch find operations, described in "Searching in a Project" (page 33).

Finding Information

An IDE should help you find the information you need while you're working on a project. Here are some ways to search for information in Xcode.

Filtered Searching

Filtered searching is available in many places across the Xcode user interface. It refers to the ability to type letters in a search field so that as you type, Xcode filters an associated list, removing any items that don't match the text you type.

For example, filtering is available in the project window for items such as files, symbols, and errors and warnings; in Info windows (described below) for build settings; and in the Developer Documentation window for symbol names defined by various Mac OS X technologies.

Searching in a Project

Searching for text in your project is a common task that must be fast and convenient. In Xcode, you can perform search and replace operations in a single file, or perform batch searches on multiple files and frameworks. You can search for text, regular expressions, or symbol definitions. You can also define complex search criteria to reuse, and you can store your search results for later reference. To learn more about searching in Xcode projects, see "Searching in a Project" (page 97).

Getting Information About Items in a Project

The project window is the main starting point for getting information about items in your project. For most kinds of information, you won't need more than a few steps:

1. Select an item in the project window.
2. If you need more information about a selected item, open an Info window (by pressing Command-I, choosing Get Info from the File menu, or clicking the Info button in the project window toolbar).

An Info window allows you to view, and in some cases modify, information on items in your project. For example, you can view and change file attributes for one or more selected files. To learn more about Info windows, see "Inspector and Info Windows" (page 73).

Using the Documentation

The importance of documentation in software development can't be overemphasized. The technical documentation distributed with Xcode provides critical conceptual and reference documentation for creating high-quality, high-performance software for Mac OS X.

At different times in the product cycle, you're likely to use the documentation to:

- Learn about the operating system and the technologies it supports
- Find and compare solutions for technical requirements
- Read about supported languages and frameworks

- Look up individual API definitions
- Learn how to use a required tool

When you install Xcode, technical documentation is installed on your hard drive. You can view it in the Developer Documentation window, accessed through the Help menu.

The Mac OS X documentation distributed with Xcode includes both Apple and open source documentation. Xcode also includes a variety of sample code, installed at `/Developer/Examples`. Documentation and sample code are also available, free of charge, at the Apple Developer Connection website at http://developer.apple.com. To learn more about viewing documentation in Xcode, see "Viewing Documentation" (page 110).

HeaderDoc

Apple provides the open source HeaderDoc system for creating HTML reference documentation from embedded comments in C, C++, and Objective-C header files. Similar to JavaDoc, the system allows you to document your interfaces and export that information into HTML. For more information on HeaderDoc, see http://developer.apple.com/darwin/projects/headerdoc/.

Editing Files

As you develop your software, you spend a lot of time editing files. To be efficient, you want to be able to work with familiar keystrokes and have access to features such as code completion, automatic indenting, syntax coloring, and so on. You also want to open files quickly, find API documentation, enter API declarations, move between header and implementation files, and work with as many or as few windows as you need.

Xcode handles these requirements through an advanced editor with many customizable features:

- You can, in Xcode's Preferences window:
  - View and modify settings for syntax coloring, indenting, and source code formatting.
  - Turn on code completion, so the editor suggests context-sensitive function names, method names, and arguments as you type.
  - Customize keystroke equivalents for menu items and editing tasks to use the keystrokes you are most familiar with; Xcode provides predefined sets that are compatible with BBEdit, CodeWarrior, and even MPW (Macintosh Programmer's Workshop, the Apple development environment for Mac OS 9).
  - Choose an external editor for any file type.
- You can edit in one or more standalone editor windows, or in an editor pane in the project window.
- You can use many Xcode shortcuts, such as Command–double-click to go from a selected function name to its definition.
- You can use features described in previous sections to quickly find a desired file, symbol, or text string.

For more information on Xcode's editor, see "The Xcode Editor" (page 169). To learn more about code completion, see "Code Completion" (page 189). To learn how to use an external editor with Xcode, see "Using an External Editor" (page 193).

Resources and Localization

In addition to source code, most projects include resources such as images, sounds, and nib resource files. Many project templates provide default resource files when you create a new project; these resource files are typically organized in the Resources source group. In addition, when you add resource files to a target, Xcode automatically adds them to the correct step of the build process, so that they will be added to your software.

Most Mac OS X software, including applications, plug-ins, and frameworks, is packaged in the form of a bundle. Xcode provides mechanisms both to help you localize resource files that need it, and, when you build your product, to copy localized resources into localized directories in your software bundle. At runtime, your source code can use various APIs provided by Mac OS X to obtain localized information from the bundle.

The following sections provide an overview of how to work with resources, support localization, and provide needed information to the Mac OS X system. To learn more about working with localized files in Xcode, see "Customizing for Different Regions" (page 197).

Information Property List Files

Any Mac OS X software that is packaged in the form of a bundle requires an information property list file named `Info.plist`. This file, which is critical to configuring your software, contains key-value pairs that specify various information used at runtime, such as the version number. Information in the property list is used by Mac OS X (for example, when launching applications) and is also available to the product that contains the property list.

When you create a new project for a bundled product, Xcode automatically creates an `Info.plist` file for the project. When you build the product, Xcode copies the property list file into the product's bundle. The information property list is associated with a target, and you can open an Info window on the target to modify property list values. You can also double-click a property list file in the project window to edit it as an XML text file.

Strings Files

Any text strings in your project that may be displayed to users should be localized. To do this, you place them in strings files, providing one localized variant for each language you support. A strings file, which has the extension `strings`, stores a series of keys and values, where the values are the strings and the keys uniquely identify the strings. Xcode supports localization with strings files by providing options to make a file localizable and to add files for local variants.

When you build your product, Xcode copies each localized strings file into the appropriate localized directory within the `Resources` directory of the product bundle. For example, if you localize for French, the French version of a strings file is copied to the `French.lproj` directory in the bundle. Mac OS X provides various APIs you can use in your source code to obtain localized information from a bundle, such as the correct text string to display for the user's current locale. For more information, see *Internationalization Programming Topics*.

The `InfoPlist.strings` file is an example of a strings file. It is used to provide localized values for any properties in the `Info.plist` file that may be displayed to users. When you create a new project for a bundled product, Xcode automatically creates this file in the Resources group. By default, it provides just an English variant, but you can add localized strings files for other languages you support.

Nib Files

A nib file, which has the extension `nib`, is a resource file that stores user interface information for a product. You create nib files with Interface Builder, which provides a powerful mechanism for graphically laying out the user interface for your software. When you add a nib file to a target, Xcode adds it to the correct stage of the build process, which causes it to be copied into the product's bundle when you build the product. Your code then has access, at runtime, to user interface items in the nib file.

Important: Because nib files are bundles, which means they are really a directory structure, not a single file, you must use special mechanisms to check them in and out with some SCM systems.

Resource Manager Files

In previous versions of the Mac OS, applications traditionally used Resource Manager `.r` (text) and `.rsrc` (compiled) resource files. While nib files are now the preferred mechanism for defining user interface items, Xcode has built-in support for Resource Manager resources as well.

When you add a Resource Manager resource file to a target, Xcode recognizes it by its extension. When it builds the target, Xcode automatically compiles `.r` files with the Rez tool. Xcode then copies the resulting `.rsrc` file into the `Resources` folder of the product bundle. If you localize any `.r` files, a `.rsrc` file is copied into the appropriate localized directory as well.

For related information about these types of resources, see "Working With Resources" in *Porting CodeWarrior Projects to Xcode*.

The Edit/Build/Debug Cycle

Once a product has been designed, you spend time in the edit/build/debug cycle: adding or modifying code, building the product, testing it, and repeating these steps, as you find and correct bugs or add additional features. You've seen Xcode's editing features in "Editing Files" (page 34). The following sections describe tools, features, and performance enhancements Xcode provides so that you can take control of the coding cycle.

Tools

The Xcode Tools include the Xcode application, Interface Builder, and a set of integrated compilers, debuggers, and build tools. Along with many tools created by Apple, Xcode incorporates several tools from the UNIX open source community. Together, these tools build on years of software development experience and take advantage of the UNIX-compatible underpinnings of Mac OS X.

Among the open source tools available in the Xcode IDE are the GCC compiler (which supports development in C, C++, Objective-C and Objective-C++), the GDB source code debugger, and the `javac` and Jikes Java compilers. Apple contributes to improvements in many of these open source tools. Standard UNIX tools are available in subdirectories of `/usr`.

Tools that originate with Apple include:

- `ld`, a linker that supports dynamic shared libraries.

  To work efficiently with dynamic shared libraries at runtime, the Mac OS X runtime architecture provides the `dyld` (dynamic loader) library manager.
- `xcodebuild`, a command-line tool for building Xcode projects.
- Terminal, an application for doing command-line work in shell windows.

In addition, Xcode includes tools to help you locate hard-to-find bugs, such as memory leaks and bugs in threaded code, as well as tools to help you analyze and optimize the performance of your software.

For more information, see the documents in Tools Xcode Documentation and Performance Documentation.

Building

The Xcode build system provides flexibility and customizability to your workflow. You can control the build process from the toolbar or with keyboard shortcuts, can view errors in the project window or in a separate Build Results window, and can go quickly from errors to the offending line of source code. In the Build Results window, you can control the level of detail—for example, you can choose whether to show warning messages and whether to display build steps.

What takes place at build time depends on several factors. When you create a new project in Xcode, it contains a great deal of build information, including default build settings, build rules that specify tools for processing source files, build styles for development and deployment builds, and build phases for performing the steps of the actual build. To learn more about the information that goes into building a product, see "The Build System" (page 229).

For a simple project, default values are sufficient for Xcode to build your product, performing such steps as compiling, linking, and copying files to the appropriate locations in an application bundle. For projects with special requirements, Xcode provides numerous options for controlling the process. For example, you can set per-file compiler flags or add a step to the build process that executes a shell script to perform special processing.

Debugging

The open source GNU Debugger, GDB, sits behind Xcode's debugger user interface. It also makes available powerful command-line debugging features. As a result, you can debug at whatever level is most comfortable for you. You can work in the user interface for most of your debugging tasks, but drop down to the command line to take advantage of advanced features that are less commonly needed. For debugging Java products, Xcode communicates directly with the Java Virtual Machine. To learn more about Xcode's debugger, see "Running in Xcode's Debugger" (page 345).

Optimizing the Edit/Build/Debug Cycle

Beside the standard features you expect in an IDE, Xcode sports a number of innovative features that can speed up your edit/build/debug cycle and make a big contribution to an efficient workflow:

- Code completion allows the editor to suggest context-sensitive function names, method names, and arguments as you type.
- Predictive compilation reduces the time required to compile single file changes by beginning to compile a file while you are still editing it.
- Distributed builds can dramatically reduce build time for large projects by distributing compiles to available computers on the network. And if you have a dual-CPU machine, Xcode automatically takes advantage of the second CPU for compiling and other operations.
- Fix and Continue improves your debugging efficiency by allowing you to change the source in a file, recompile just that file, and run the changed code without stopping the current debugging session.
- ZeroLink shortens link time for development builds and lets you quickly relaunch your application after making changes. (Don't forget to turn ZeroLink off for deployment builds!)

To learn more about these features, see "Optimizing the Edit-Build-Debug Cycle" (page 325), "Using ZeroLink" (page 320), and "Using Fix and Continue" (page 373).

Analyzing and Optimizing Your Software

Performance optimization should be an integral part of the development cycle, and should include steps such as providing a plan to continually measure and improve the performance of your code. Xcode Tools provide a number of tools to help you fine-tune your software, including tools to:

- Examine memory use and find leaks
- Analyze where your software spends its time, in both application code and system code
- Determine what an unresponsive application is doing
- Graphically track the activity of an application's threads

You can launch your program in many of these tools directly from the Xcode application. For more information on analyzing and optimizing your software in Mac OS X, see the learning paths in Getting Started With Performance.

Customizing Your Work Environment

You can work most efficiently when your development environment complements the way you like to work. Xcode provides many options for customizing its interface, from setting the keystrokes for menu and text-editing equivalents, to configuring the contents and layout of the project window, to setting conventions for editing code.

Preferences

The Xcode Preferences window is the key to customization, and you should spend some time investigating it. It provides access to settings for features such as text editing, syntax coloring, indentation, navigation, building, debugging, source code management, and key bindings for menus and text editing. To learn more about Xcode's Preferences, see "Xcode Preferences" (page 393).

Customizing the Xcode User Interface

You can customize Xcode's project window and many other windows too. For example, Xcode provides a number of different project window configurations, or layouts, for you to choose from. You can embed an editor and specify which columns and groups should be shown in the project window. You can also customize toolbars and menus, and control the amount of information shown in the Build Results window.

Working in a Shell

Mac OS X incorporates the FreeBSD variant of UNIX, which includes a command-shell environment. The Terminal application provides an interface for invoking command-line utilities and executing shell scripts. You can build Xcode projects from a shell with the `xcodebuild` command, in order to, for example, run nightly builds. For more information, see "Building From the Command Line" (page 312).

Xcode also makes it easy to execute shell scripts as part of your development work. You can execute selected text as a shell script or run scripts from the User Scripts script menu. That menu contains default scripts that you can execute as is, or use as examples for scripts you write. You can also write shell scripts that Xcode executes during the build process.

For more information on modifying your work environment and working with shell scripts in Xcode, see "Customizing Xcode" (page 383).

Projects

A **project** is a repository for all the information required to build one or more software products. It contains all the elements used to build your products and maintains the relationships between those elements. You can think of it as a kit that contains all the parts to build one or more products, plus the instructions on how to build them. A project gives you a convenient place to find every file and piece of information associated with your work.

The following chapters introduce the various parts of a project, show you how to create new Xcode projects or convert existing projects, and describe how you can organize the contents of a project. They also describe the project window, Xcode's interface for performing project management tasks; and show you how to use that interface to find and discover information in Xcode.

PART I

Projects

Projects in Xcode

To carry out the development process, Xcode relies on certain key components. It uses projects to organize this information. The project is a repository for all of the information needed to build one or more software products. It is also the primary workspace for your software development. This chapter describes the contents of an Xcode project and gives an overview of the information required to develop software with Xcode.

Components of an Xcode Project

A project contains and organizes everything you need to create one or more software products. Your Xcode project serves two important purposes; it:

1. Organizes build system inputs for building a product.
2. Maintains information on the items in your project and their relationships, to assist you in the development process.

To develop a product using Xcode, you must understand the key components of your project. Figure 2-1 shows a simplified representation of a project and its essential pieces.

Figure 2-1 The key components of a project

The project's contents include the following essential items; in the course of developing a product, you will work with each of these:

1. Files. **Source files** are the files used to build a product. These include source code files, resource files, image files and others.

   A project keeps all of the source files you use for a particular product or suite of related products. A project can also contain files that are not directly used by Xcode to build a product, but contain information that you use during the development process, such as notes, test plans, and more.

   In the course of developing a product you edit project files—using Xcode's built-in editor or an external editor—to author source code for a product, and organize files into a target (described below) to define the build system inputs for creating the product.

A project keeps a reference to each file you add to the project. A project can also contain folder references, if you want to manipulate a group of files as a whole; framework references to access the contents of a framework, or references to other projects. "Files in a Project" (page 75) describes how Xcode stores these references and discusses the files in a project in more detail.

2. Targets. When it comes time to actually create, or **build**, a product, you use a target. A **target** defines a single product; it organizes the inputs into the build system—the source files and instructions for processing those source files—required to build that product. To create a finished product, build its target. Projects can contain one or more targets, each of which produces one product.

   Targets, and the products they create, may be related. If a target requires the output of another target in order to build, the first target is said to depend upon the second. Xcode lets you add target dependencies to express this relationship. "Targets" (page 231) describes targets and the instructions they contain in more detail.

   For each target in your project, Xcode adds a **product reference**. This is a file reference to the output generated by the target, such as an application. You can use this product reference to refer to the products in your project the same way you use a file reference to refer to a file; however, the product reference does not actually refer to anything in the file system until you build the product.

3. Executables. After you've successfully built a product, you need to test it to make sure that it works. When it comes time to run or debug your product, you use an executable environment to tell Xcode how to do so. An **executable environment** tells Xcode

   a. What program to launch when you run or debug from within Xcode.

   b. How to launch the program. The executable environment lets you tell Xcode what command-line arguments to pass, environment variables to set, debugger to use, and so forth.

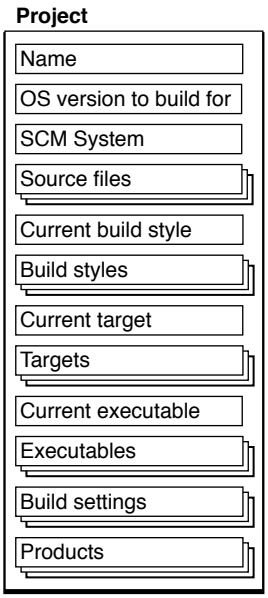
   If you are building a product that can be run on its own—an application, command-line tool, and so forth—Xcode automatically sets the default executable to the target's product. However, if you have a product such as a plug-in or framework, you must create an executable environment to specify a program to run and test your product with.

   Even if your product generates an executable that can run on its own, you may want to customize the executable environment to specify command-line arguments for Xcode to pass to the program on launch, environment variables to set, and so forth. "Executable Environments" (page 337) describes executable environments in Xcode in more detail, and explains how to modify executable settings.

   A project can contain any number of executables. There is not a one-to-one correspondence between targets and executables, although Xcode automatically creates an executable environment for each target that creates a product that can be run on its own. You can, however, define multiple executable environments to use to test the product of a single target under different circumstances.

In addition to these fundamental building blocks of the development process, your Xcode project also maintains a great deal of information about the items in your project and their current state. Figure 2-2 shows a representation of a project with this additional information.

Figure 2-2 An Xcode project



Your Xcode project tracks:

- Organizational information that Xcode uses to help you do your work. For example, projects can contain groups to help you organize and find files, or bookmarks to your favorite locations. Xcode also maintains a symbolic index for your project; it uses this information to provide assistance such as code completion, project-wide symbol searching, and more. Groups and other features for organizing project contents are described in "Organizing Xcode Projects" (page 83); project-wide searches and other features for finding information in your project are described in "Finding Information in a Project" (page 97).
- Project-wide settings that affect the build process and other software-development operations for all targets and project items. For example, the project tracks the "active" target; this is the target that Xcode builds when you hit the Build button.

The settings that an Xcode project tracks include:

- Source control management system (SCM). Xcode supports the CVS, Perforce, and Subversion version control systems for managing changes to source code. "Version Control" (page 201) describes how to choose a version control system and how to work with version control features in Xcode.
- Mac OS X SDK version. Using Xcode, you can develop software that can be deployed on versions of Mac OS X different from the one you are developing on. You can choose which version (or SDK) of Mac OS X headers and libraries to build with. "Using Cross-Development in Xcode" (page 333) briefly describes how to choose a Mac OS X system version in Xcode. For detailed information, see *Cross-Development Programming Guide*.
- Build styles. A **build style** lets you modify how a target is built. It contains build settings that let you override some of the information in a target, without creating a whole new target. In this way, you can rapidly try variations on a target without maintaining multiple copies. The project keeps track of the available build styles and their definitions. See "Build Styles" (page 297) for more information on build styles in Xcode.

- Active target and build style. The **active target** is the target that gets built when you initiate a build in Xcode. The **active build style** is the build style that Xcode applies to the active target, and any targets it depends upon, when you build. See "Setting the Active Target and Build Style" (page 304) for more information.
- Active executable. The **active executable** is the executable environment that specifies which program is launched, and how, when you run or debug from within Xcode.

Because active build style, choice of version control system, and SDK version are all attributes of the project, they must be the same for all targets in a project. For example, if you have a target that uses the Perforce source control system, you can't have another target in the same project that uses the CVS system. You can, however, use cross-project references and dependencies to tie together targets in separate projects that use different version control systems.

A project can have multiple targets and multiple executables. However, there can only be one active target, one active build style, and one active executable. So, for example, if a project builds more than one application, only one executable—corresponding to one application—can be active and that's the only executable you can debug in Xcode's graphical debugger. If you want to debug both applications at once with the graphical debugger, you'll have to build them in separate projects.

The Project Directory

When you create a new project, Xcode creates a project directory to hold your project's contents. The project directory contains the project package, which holds project metadata—as described in the previous section—and user information. The project package has the same name as the project and carries the extension `.xcode`. For more on the project package, see "Project Packages" (page 205).

In addition to the project package, the project directory can also contain:

- Project files and folders. Source files can live anywhere on your system, but keeping them in your project directory makes it easy to move the project and its contents around. By default, Xcode interprets most paths relative to the project directory.

  You can organize files into any number of subfolders within the project directory. This includes folders for localized resources, as described in "Customizing for Different Regions" (page 197).

  If you create a project from one of Xcode's project templates, the project directory already contains a number of example source files. For more on the files in a project, see "Files in a Project" (page 75).
- Build folder. When you build a target, Xcode generates a number of files, including the target's finished product. By default, Xcode creates a `build` folder in the project directory to hold the files that it creates. The build folder can, however, reside at any location in the filesystem. For more information on the build folder, see "Build Locations" (page 301).

Creating a Project

Once you know what product you are working on, you need an Xcode project. If you are working on a new product, and do not already have an Xcode project, you can create one from scratch. Xcode provides project templates to help you create a wide variety of products.

If you are working on an existing product, you probably already have a project. If you have an existing Xcode project, you can simply open the project in Xcode, as described in "Opening and Closing Projects" (page 54). If you have an existing CodeWarrior project, you can import your project into a new Xcode project. Or, if you have an existing Project Builder project, you can simply open it in Xcode.

This chapter shows you how to create a new project and describes the available project templates. It also shows you how to import CodeWarrior, Project Builder, and ProjectBuilderWO projects.

Choosing a Project Template

Fairly early in your design process, you make decisions related to the type of product (application, library, command-line tool, and so on) and language or languages (C, C++, Objective-C, Java, and others) you plan to use. For a Mac OS X product, you also decide which Apple technologies to use, and whether to use an application framework, such as Cocoa or Java.

Once you've resolved these issues, you'll find that Xcode provides a wide variety of project templates to support your goals. Table 3-1 provides brief descriptions for the project templates currently supplied by Xcode. You can find similar descriptions when you select a project template in the New Project Assistant in Xcode. When you add a target to a project, you'll get a selection of targets to choose from that is similar to the list of available project templates. For more information on targets and target templates, see "Creating Targets" (page 233).

The project template you choose specifies a default target and it also determines the default source files, resources, framework references, and other information that Xcode includes automatically in the project. A project generally contains all the information it needs to build a product for its default target. This includes a minimal set of source files that you can compile into a running product, as well as default build settings.

Table 3-1 Xcode project templates

Project template	Use to create
Empty Project	A project with no files or targets.
Action	
Automator actions are loadable bundles that perform discrete tasks for users to link together in a workflow, using the Automator application. For more information, see *Automator Programming Guide* .	
AppleScript Automator Action	An Automator action written using AppleScript.

Project template	Use to create
Cocoa Automator Action	An Automator action written in Objective-C.
Application	
AppleScript Application	An AppleScript Studio application: a simple Cocoa application which can be written in AppleScript, Objective-C, and other languages.
AppleScript Document-based Application	An AppleScript Studio application that uses the Cocoa document architecture.
AppleScript Droplet	An AppleScript Studio application that processes files dropped on it.
Carbon Application	An application, based on the Carbon framework, that uses nib files for resources (a **nib file** typically defines and lays out objects for a product's graphical interface).
Cocoa Application	An application, based on the Cocoa framework, that is written in Objective-C and relies on nib files to define its graphical interface.
Cocoa Document-based Application	A Cocoa application that uses the Cocoa document architecture.
Cocoa-Java Application	A Cocoa application that is written in Java.
Cocoa-Java Document-based Application	A Cocoa application that is written in Java and uses the Cocoa document architecture.
Core Data Application	An application, based on the Cocoa framework, that is written in Objective-C and uses the Core Data framework to save and restore objects. For more information on the Core Data framework, see *Data Modeling Guide* and *Core Data Programming Guide*.
Core Data Document-based Application	A Core Data application that uses the Cocoa document architecture.
Bundle	
A bundle is a file system directory that stores executable code and the software resources related to that code. Bundle templates are for loadable bundles—code (such as application plug-ins) that can be loaded when it is needed. For more information, see *Bundle Programming Guide*.	
Carbon Bundle	A bundle that links against the Carbon framework.
CFPlugin Bundle	A bundle that links against the Core Foundation framework.
Cocoa Bundle	A bundle that links against the Cocoa framework.
Command Line Utility	
A command-line tool is a utility without a graphical user interface. Command-line utilities are typically used in the command-line environment.	

Project template	Use to create
C++ Tool	A tool that links against the stdc++ library.
CoreFoundation Tool	A tool that links against the Core Foundation library.
CoreServices Tool	A tool that links against the Core Services library.
Foundation Tool	A tool that links against the Foundation framework. (Foundation is one of the frameworks in the Cocoa framework.)
Standard Tool	A tool written in C.
Dynamic Library	
A dynamic library is a library for which binding of undefined symbols is delayed until execution; code in dynamic shared libraries can be shared by multiple, concurrently running programs.	
BSD Dynamic Library	A dynamic library, written in C, that makes use of BSD (a part of the Mac OS X kernel environment; BSD stands for Berkeley Software Distribution)
Carbon Dynamic Library	A dynamic library that links against the Carbon framework.
Cocoa Dynamic Library	A dynamic library that links against the Cocoa framework.
External Build System	A project that contains no files but has a single target which you can configure to use any command-line build tool.
Framework	
In Mac OS X, a **framework** is a hierarchical directory that encapsulates a dynamic library and shared resources in a single package. You access many Mac OS X technologies in Xcode through frameworks. For more information, see *Framework Programming Guide*. Note that Cocoa is an *application* framework—one which supplies the basic building blocks of an application, to which you add your own code and features—that is implemented as a framework (as defined above).	
Carbon Framework	A framework that links against the Carbon framework.
Cocoa Framework	A framework that links against the Cocoa framework.
Java	
Ant-based Application Jar	An application, written in Java and built as a JAR file (JAR is the Java Archive file format). The application is built using the Ant build tool. The project contains a default `build.xml` file.
Ant-based Empty Project	An empty project with no files, that contains a single external target configured to use the Ant build tool. You must supply the `build.xml` file.
Ant-based Java Library	A library, written in Java and packaged as a JAR file, that is built using the Ant build tool. The project contains a default `build.xml` file.

Project template	Use to create
Java AWT Applet	An AWT-based Java applet, built as a JAR file (AWT is the Advanced Windowing Toolkit).
Java AWT Application	An AWT-based application, built as an application bundle.
Java JNI Application	A JAR file-based JNI Application (JNI is the Java Native Interface).
Java Swing Applet	A Swing-based Java applet, built as a JAR file.
Java Swing Application	A Swing-based application, built as an application bundle.
Java Tool	A library or application, built as a JAR file.
Kernel Extension	
A **kernel extension** (or KEXT) is a piece of code that can be dynamically loaded into the Mac OS X kernel. A driver is a kernel extension that supports one or more devices.	
Generic Kernel Extension	A kernel extension.
IOKit Driver	A device driver that uses the I/O Kit (an object-oriented framework for developing device drivers for Mac OS X).
Standard Apple Plug-ins	
Plug-ins for standard Apple applications, including Interface Builder, preference panes, the Screen Effects pane in System Preferences, and Sherlock.	
Address Book Action Plug-in for C	A C-based plug-in that implements an Address Book action. See *Address Book Programming Guide for Mac OS X* for more information.
Address Book Action Plug-in for Objective-C	An Objective-C-based plug-in that implements an Address Book action.
AppleScript Xcode Plug-in	An AppleScript-based plug-in for Xcode.
IBPalette	A plug-in for Interface Builder that adds a palette of user-interface items.
Image Unit Plug-in for Objective-C	An Objective-C-based plug-in that implements one or more image filters for Core Image. For more information on creating image units, see *Core Image Programming Guide*.
Installer Plug-in	An Objective-C-based plug-in that implements a custom interface for the Installer.
Metadata Importer	A metadata importer that allows Mac OS X to extract metadata from custom document formats for use with Spotlight. See *Spotlight Importer Programming Guide*.

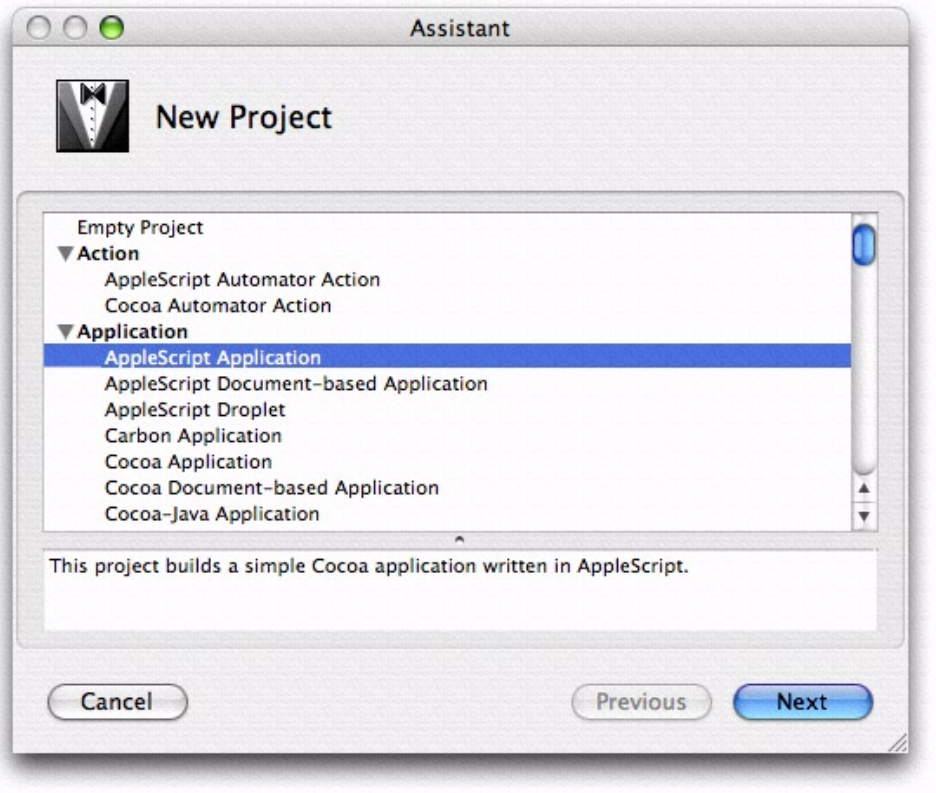
Project template	Use to create
PreferencePane	A plug-in for a preference pane bundle that can be used with the System Preferences application or an application's user preferences.
Screen Saver	A plug-in for a screen saver bundle that can be used with the Screen Effects panel in the System Preferences application.
Sherlock Channel	A plug-in for a Sherlock Channel that can be used with the Sherlock Internet search application. See *Sherlock Channels*.
Sync Schema	A `.syncschema` bundle that allows your application to sync user data with other applications and devices on the same computer using Sync Services. See *Sync Services Programming Guide*.
Static Library	
A library for which all referenced symbols are bound at link time.	
BSD Static Library	A static library, written in C, that makes use of BSD (see BSD Dynamic Library above)
Carbon Static Library	A static library that links against the Carbon framework.
Cocoa Static Library	A static library that links against the Cocoa framework.

The project template names and descriptions should give you a good idea of which project template is right for your product. One way to learn more about a project template is to create a project with that template, examine its contents, and see what happens when you build it. Project templates may change, and new templates are added from time to time with releases of Xcode, but by trying out a template, you can easily examine its default contents in that version of Xcode.

Creating a New Project

To create a new project, choose File > New Project. Xcode displays the New Project Assistant, shown here.

Figure 3-1 The New Project Assistant



The first screen of the assistant shows the available project templates; these are described in "Choosing a Project Template" (page 47). You can choose a project template from this list, or you can select Empty Project to create a new project with nothing in it.

When you select a template from this list, a brief description appears in the text field directly below the template list. If there is no text field visible, grab the resize control below the template list and drag it upward.

After you select a project template, click Next; Xcode displays the second screen of the assistant. In this screen, assign a name to the new project and select a location for the project folder. If you type in a path containing directories that do not exist on disk, Xcode will create those directories for you before creating the new project. By default, the Project Directory path is set to '~/' which specifies that the project directory be placed at the top-level of your home directory. When you click Finish, Xcode creates a new project from the specified template.

Importing a Project

If you already have existing code in another IDE, you can import your project contents into Xcode. Xcode provides an assistant for importing CodeWarrior and ProjectBuilderWO projects. Xcode also opens Project Builder projects. This section describes how to use Xcode's project importer and open Project Builder projects in Xcode.

Importing CodeWarrior Projects

If you have existing code in a CodeWarrior project, you can use the Xcode importer to import your CodeWarrior project and create a corresponding Xcode project. This section gives a brief description of how to import a CodeWarrior project into Xcode and lists a few steps that you can take to make the conversion process easier. While this section assumes that you are converting a project that builds an application, many of the steps apply to other types of software as well.

Before You Import

To make it easier to get your project building in Xcode, you may need to make some changes to your project before you import it. Each of these steps is described in further detail in *Moving Projects from CodeWarrior to Xcode*.

1. Convert your code to use Carbon.
2. Set the Project Type setting of your CodeWarrior project to Application Package to build your application as a package.
3. Build the application in the Mach-O format. Mach-O is the native executable format in Mac OS X and is the only format supported by Xcode.
4. Remove unnecessary targets from the project.

Importing Your CodeWarrior Project into Xcode

To import your project:

- Choose File > Import Project, which opens the Import Project Assistant.
- Select Import CodeWarrior Project and click the Next button.
- Click the Choose button and navigate to the CodeWarrior project file to import (or type in the path and filename).

  You can specify a name for the new project in the New Project Name field. Otherwise, Xcode automatically uses the same name as that of the project that you are importing. The Xcode importer creates the new project in the same folder as the CodeWarrior project file.
- When you import a CodeWarrior project, Xcode determines the location of the CodeWarrior root folder (referred to as `{Compiler}` in CodeWarrior's search path) and adds it to the Source Trees list in the Preferences window.

  Select Import "Global Source Trees" from CodeWarrior if you want the importer to add any global source trees from CodeWarrior's preferences to the Source Trees list in Xcode.
- Select 'Import referenced projects' to import any additional CodeWarrior projects referenced by the project you are currently importing.
- Click the Finish button to dismiss the assistant and start the import.

  When the import is complete, the new project window opens.

Most projects will not build immediately; see *Moving Projects from CodeWarrior to Xcode* for more information on how to get your new Xcode project to build.

Converting a Project Builder Project

If you already have an existing Project Builder project, there is very little you have to do to get it converted to, and building in, Xcode. Xcode works seamlessly with Project Builder projects. You can simply open the project in Xcode, by double-clicking on the project (`.pbproj`) bundle, dragging the project bundle to the Xcode application, or using the Open command in Xcode and selecting the project bundle.

Note that if you also have Project Builder installed on your computer, double-clicking the `.pbproj` bundle will open the project in Project Builder. To open the project in Xcode, you will have to use one of the other methods mentioned here.

Once you have opened your Project Builder project in Xcode, it should build with little or no additional work on your part. If you wish to take advantage of some of Xcode's more significant features, such as ZeroLink, Fix and Continue, and the like, you will have to convert the targets in your project to use the native build system. Xcode does not automatically upgrade the existing targets, which use Project Builder's Jam-based build system, when you open the project. To learn more about converting native targets, see "Converting a Project Builder Target" (page 247).

Importing Projects From ProjectBuilderWO

Xcode also supports importing projects from ProjectBuilderWO. To import a ProjectBuilderWO project, do the following:

1. Choose File > Import Project.
2. From the Import Project Assistant, choose Import ProjectBuilderWO Project and click Next.
3. Type the path to the ProjectBuilderWO project in the ProjectBuilderWO Project field or click Choose to select the project from an Open dialog.
4. Type the name you want to use for the new Xcode project in the New Project Name field and click Finish. If you do not specify a different name, Xcode uses the name of the existing project.

Xcode imports the ProjectBuilderWO project and creates a new Xcode project file. Where possible, Xcode imports the targets in the ProjectBuilderWO project as native targets. If the ProjectBuilderWO project contains targets not supported by Xcode's native build system, the importer imports those targets as Jam-based targets.

Opening and Closing Projects

To open any project, choose File > Open. To open a project you've recently used, select the project from the File > Recent Projects menu. To close a project, you can choose File > Close Project, or close the project window.

You can have Xcode automatically remember the state of a project's windows and restore them when you next open the project. To do so, choose Xcode > Preferences, click General, and select "Save window state" in the Environment options. If this option is disabled, opening a project displays only the project window. If this option is enabled, opening the project restores the project window and any other open windows to the state they were in when you last closed the project.

The Project Window

The project window is where you do most of your work in Xcode. The project window displays and organizes your source files, targets, and executables. It allows you to access and edit all of the pieces of your project. To work effectively in Xcode, you need to recognize the parts of the project window and understand how to use them to navigate your project's contents.

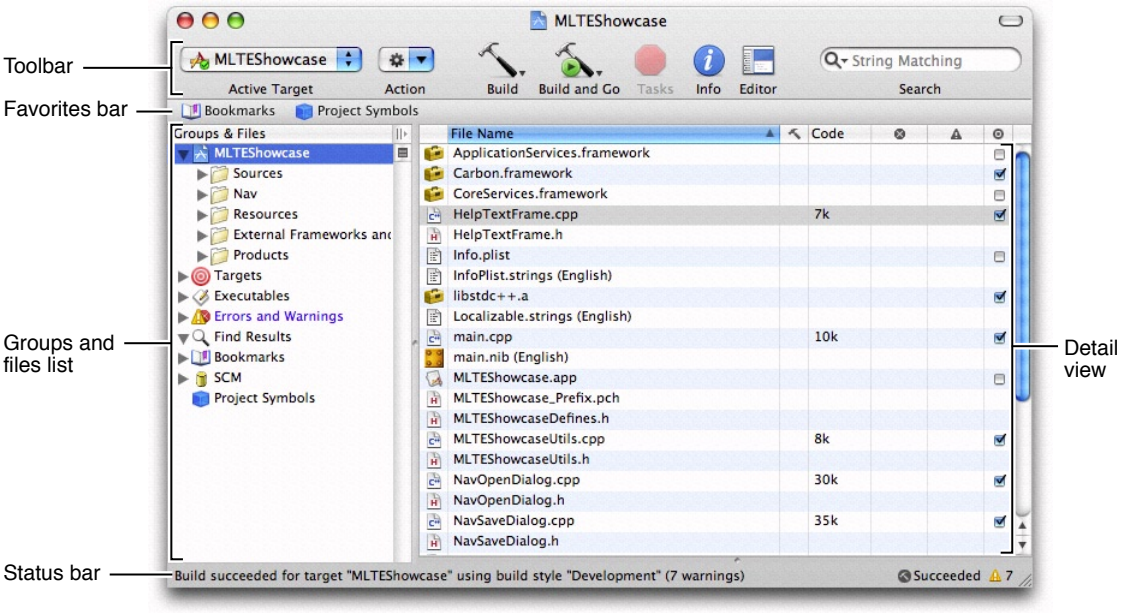
Of course, everyone has their own way of organizing their workspace. To help you be as efficient and productive as possible, Xcode provides several different project window configurations or **layouts**. A project window layout specifies a particular project window configuration, as well as ancillary task-specific windows.

This chapter introduces the basic project window components and describes the available project window layouts. It also introduces other important Xcode windows and provides tips on effectively using the Xcode interface to locate information on project items.

The Project Window and its Components

When you first launch Xcode as a new user, Xcode displays the default project window configuration, shown here.

Figure 4-1 Xcode's default project window



The project window contains the following key tools for navigating your project:

- The Groups & Files list provides an outline view of your project contents. You can move files and folders around and organize your project contents in this list. The current selection in the Groups & Files list controls the contents displayed in the detail view.
- The detail view shows the item or items selected in the Groups & Files list. You can browse your project's contents in the detail view, search them using the Search field, or sort them according to column. The detail view helps you rapidly find and access your project's contents.
- The toolbar provides quick access to the most common Xcode commands.
- The status bar displays status messages for the project. During an operation—such as building or indexing—Xcode displays a progress indicator in the status bar to show the progress of the current task.
- The Favorites bar optionally lets you store and quickly return to commonly accessed locations in your project. The Favorites bar is described in "The Favorites Bar" (page 90).

The project window can also contain an attached editor that lets you edit files directly in the project window. You can navigate through the views in a window, including the attached editor, by pressing the Tab key. To take the focus away from the editor, press Control-Tab.

The exact configuration of the project window depends on the current layout; nonetheless, each layout uses these components help you view and access project items. For a description of each of the available project window layouts, see "Project Window Layouts" (page 64).

The Groups & Files List

The Groups & Files list provides an outline view of your project's contents. The contents of your project—files, folders, targets, executables, and other project information—are organized into groups. A **group** lets you collect related files or information together. The Groups & Files list gives you a hierarchical view of these groups.

Using the Groups & Files list, you can:

- View your project's contents, organized hierarchically. You can have as much or as little of your project's contents as you want visible at once.
- Create additional Groups & Files list views to focus on multiple groups at once.
- Drag files, folders, groups, and other project items to rearrange and organize them.
- Rename files, folders, and other project items. To do so, Option-click the item and type the new name; or Control-click and choose Rename from the menu.

Groups in Xcode

Xcode supports two types of groups; you can view and edit groups of either type in the Groups & Files list:

- **Source groups** organize your project's source files, including implementation files, resources, frameworks, headers, and other files. A source group, indicated by a yellow folder icon, can contain any number of files and other source groups. Source groups help you organize the files in your project into more manageable chunks. The project group, represented by the project icon at the top of the Groups & Files list, is a source group that contains all of the files, frameworks, and libraries included in your project.
- **Smart groups** collect files or information that match a certain rule or pattern. Xcode provides several built-in smart groups:

- The Targets group contains all of the targets in your project. As mentioned earlier, a target contains the instructions for creating a single software product. Targets are described in more detail in "Targets" (page 231).
- The Executables group contains all of the executables defined in your project.
- The Errors and Warnings smart group lists the errors and warnings generated when you build. This group is described further in "Viewing Errors and Warnings" (page 309).
- The Find Results smart group contains the results of any searches you perform in your project. Each search creates a new entry in this group. For more information on the Find Results group, see "Viewing Search Results" (page 99).
- The Bookmarks smart group lists those commonly accessed locations—files or specific locations within a file—that you have saved in order to return to them easily. For more information on the Bookmarks smart group, see "Saving Commonly Accessed Locations as Bookmarks" (page 91).
- The SCM smart group lists all of the files that have source control information. This group is described further in "Viewing File Status" (page 211).
- The Project Symbols smart group lists all of the symbols defined in your project. This group is described further in "Viewing the Symbols in Your Project" (page 105).

Xcode also lets you create your own smart groups and define your own rules for what they contain; these smart groups are indicated with a purple folder icon. For more information on using smart groups and source groups to organize your project contents, see "Organizing Files" (page 86).

Hiding and Showing Groups

Xcode provides a number of smart groups to help you organize and find information; however, you may not need all of those smart groups all of the time. You can customize the display of the current Groups & Files list to show only those smart groups that you currently need. You can rearrange or delete any of the smart groups that appear in a Groups & Files list view, including those that Xcode supplies.

To rearrange a smart group, drag it to its new position in the Groups & Files list. To hide a smart group in the current Groups & Files list, select it and press Delete or choose Edit > Delete. To view the smart group again, choose it from the View > Show menu, as described in the next section.

Xcode also provides contextual menu items to show and hide smart groups in the Groups & Files list. Control-click in the Groups & Files list to show the contextual menu and choose the smart group from the Preferences menu. If the smart group is currently visible, choosing it from this menu hides the smart group. Otherwise, it shows the smart group in the Groups & Files list. A checkmark next to the smart group's name in the Preferences menu indicates that the smart group is currently visible. You can show and hide Xcode's built-in smart groups, as well as any smart groups that you have defined for your projects.

Viewing the Contents of a Group

You have a couple of options for viewing the contents of a group in the Groups & Files list. If you prefer the outline view, you can disclose the group's contents directly in the Groups & Files list. You can also select one or more groups to view their contents in a simple, searchable list in the detail view, described in "The Detail View" (page 60).

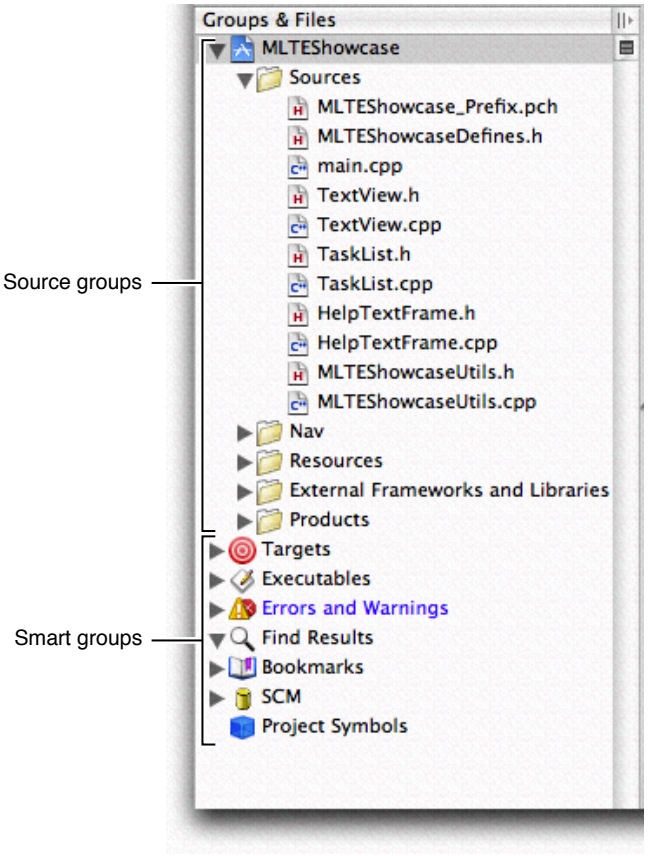
To view the contents of a group in the Groups & Files list you can:

1. Double-click the group to expand its contents in the outline view, shown below. Double-clicking the group a second time closes the group.
2. Click the disclosure triangle next to the group to show its contents in the outline view.
3. Choose any of Xcode's built-in smart groups from the View > Show menu to disclose that group's contents in the outline view. Xcode also selects the group and displays its contents in the detail view. Choosing View > Show > All Files discloses the contents of the project group. You can use the Vie w > Show menu to display a smart group, even if you had previously deleted from the current Groups & Files list.
4. Select the group to show the contents of the group, and of any other groups it contains, in the detail view.

You can sort the contents of a source group in the Groups & Files list by choosing an item from the View > Sort menu. You can sort by name or by file types. Xcode sorts the contents of the currently selected group.

The Groups & Files list can display additional information for files in the list. You can view status for files under version control or see whether a file is included in the active target. To see this information in the Groups & Files list, choose View > Groups & Files Columns > SCM or View > Groups & Files Columns > Target Membership, respectively.

Figure 4-2 Outline view of the project

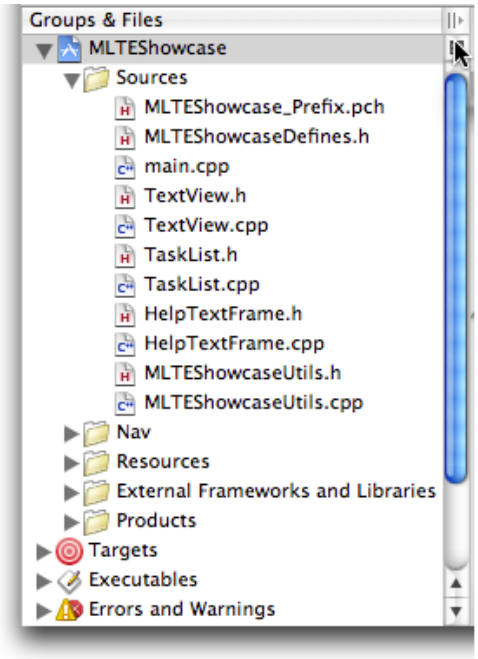


To view more information on any item in the Groups & Files list, select that item. Xcode displays additional information about that item in the associated detail view, if one is open.

Splitting the Groups & Files View

In large projects, the Groups & Files list can get quite long, making it difficult to move items around. You can split the Groups & Files view by clicking the icon in the scrollbar next to the Groups & Files list, as shown here. You can also split a view by choosing View > Split 'Files' Vertically or typing Command–double-quote. Xcode splits the view that currently has focus.

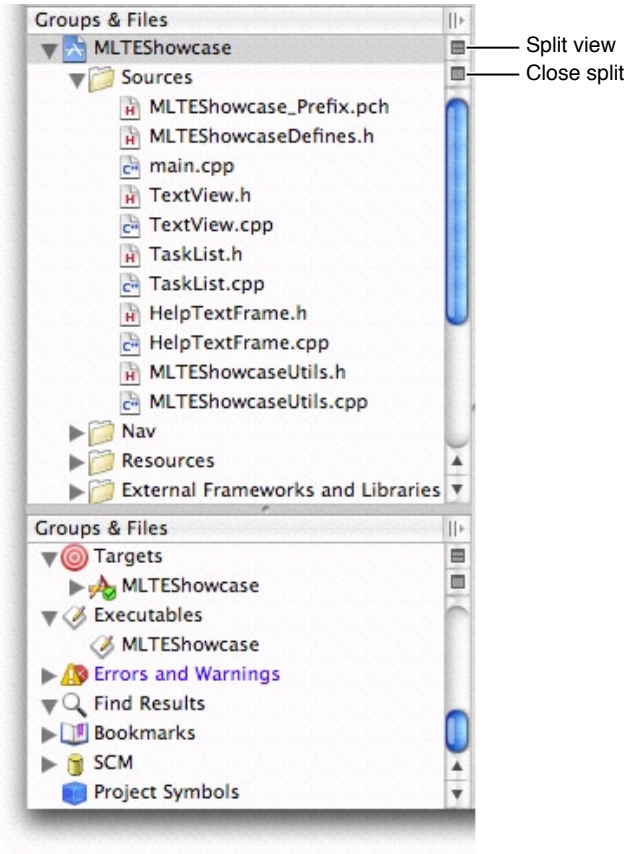
Figure 4-3 Splitting the Groups & Files view



Each view can display a different location, making it easy to keep commonly accessed groups handy or to move items between groups. You can drag the resize control between the Groups & Files views to repartition the space between them as you see fit. Each view has its own split-view button; you can split each of these views, creating as many split-views as you need. After you create a split, a new button appears below the split-view button; to close a split Groups & Files view, click this second button. You can also close a split view by choosing View > Close Split 'Files'.

By default, Xcode splits the view vertically. However, you can also split a view horizontally. To split a view horizontally, hold down the Option key while clicking the split-view button or choosing View > Split 'Files' Horizontally. You can also type Option–Command–double-quote.

Figure 4-4 The split Groups & Files view



The Detail View

As you learned, the Groups & Files view lets you see the contents of your project in an organized outline. In contrast, the detail view shows you items in a flat list. You can quickly search and sort the items in this list, gaining rapid access to important information in your project.

You control the scope of the information shown in the detail view with your selection in the Groups & Files list. If the selected item is a group, the detail view displays information for all of the members of that group and of any sub-groups that it contains. You can select multiple items in the Groups & Files list; the detail view displays all of the selected items and their members. This applies to both contiguous selections and to selections of items that are not adjacent.

Note: Note that the content of groups such as frameworks and bundles are only shown in the detail view when that framework or bundle is directly selected in the Groups & Files view, to avoid mixing items such as external framework headers and project headers.

The Information Displayed in the Detail View

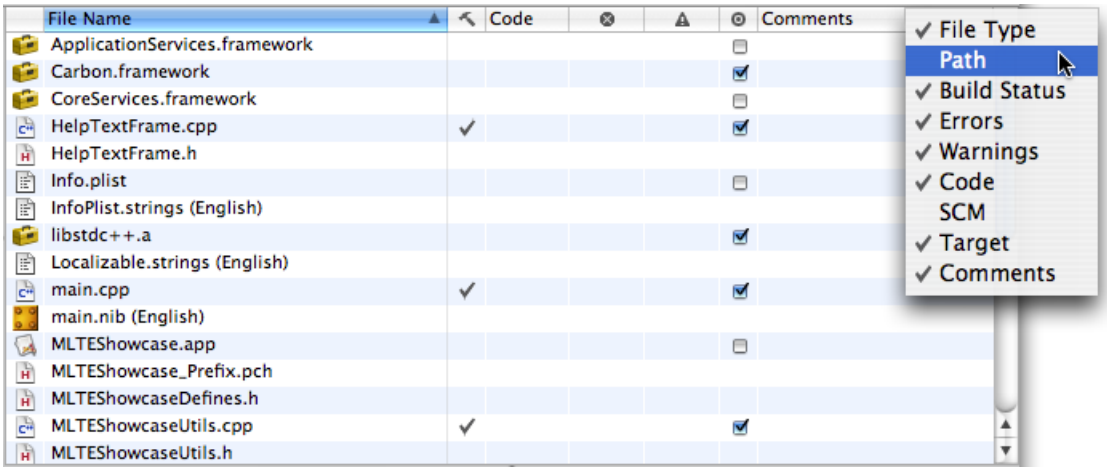
The type of information displayed in the detail view varies, depending on the item selected in the Groups & Files list. For example, if you select a group of source files in the Groups & Files list, the detail view displays each of the files in that group, along with information about those files, such as the file's build status or code

size. However, build status and code size make no sense for errors and warnings, so when you select the Errors and Warnings group, you see a list of error and warning messages and the locations at which they occurred.

You can choose what information Xcode shows in the detail view by choosing which columns are visible. To make a column visible, choose it from the View > Detail View Columns menu. You can also Control-click any of the column headings; Xcode brings up a menu like the one shown below, which allows you to choose which columns are visible.

Note: Some columns in the detail view are required, depending on the currently selected group. These columns do not appear in the View > Detail View Columns menu or in the contextual menu.

Figure 4-5 Choosing the type of information to display in the Detail view



You can display the columns of the detail view in any order. To reorder the columns, drag the heading of any column to its new position.

You can use the menu items View > Previous Detail and View > Next Detail, or their keyboard shortcuts, to move the selection up or down in the detail view. To disclose the currently selected detail in the Groups & Files list, choose View > Reveal in Group Tree. For example, if the current selection in the detail view is an individual source file, Xcode selects that file in the Groups & Files list, disclosing the contents of any source groups that the file belongs to, as necessary.

To rename an item in the detail view, Option-click the item and type the new name; or Control-click the item and choose Rename from the menu.

Searching and Sorting in the Detail View

With the detail view you have a couple of ways to find and view information. You can sort the contents of the detail view according to the information in any of the visible categories simply by clicking on the column heading for that category. For example, to sort by file name, click the File Name heading.

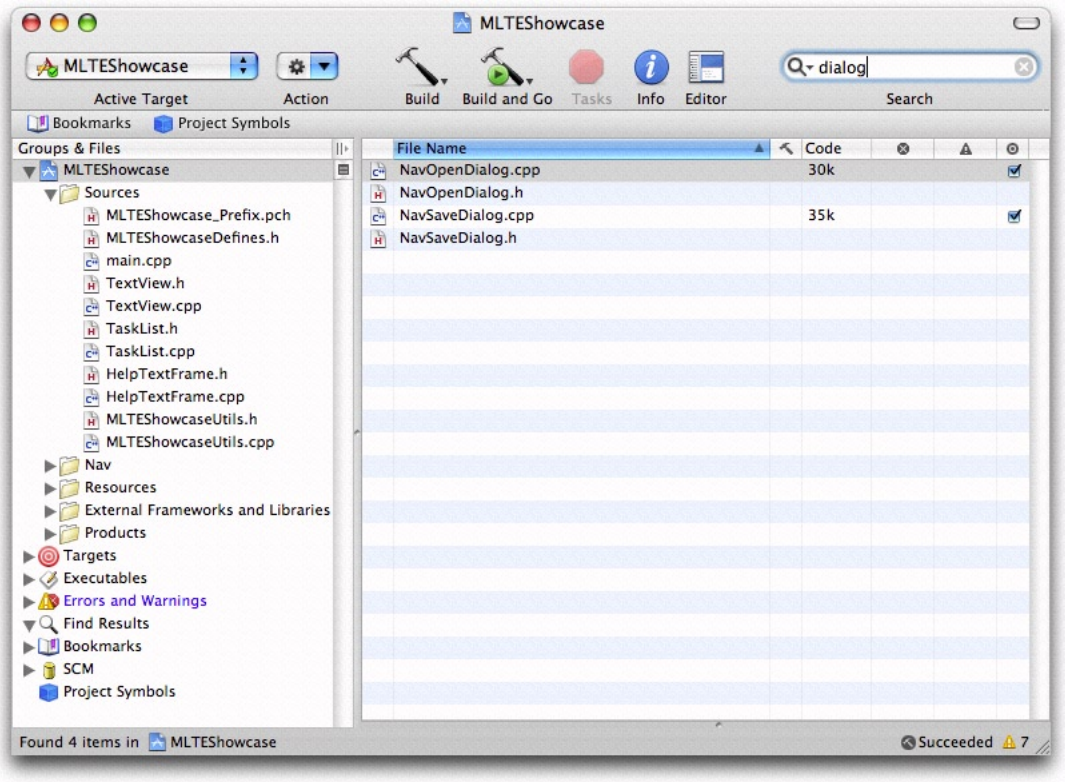
Using the search field in the toolbar, you can quickly search the contents of the detail view. As you type, Xcode filters the contents of the detail view, displaying only those items that have matching text in at least one of the columns.

The Search field supports several different types of search; you can choose the search type from the pop-up menu in the Search field. Xcode supports the following searches:

1. String Matching. Xcode determines a match using simple string comparison, filtering out items that do not match the string in the Search field. This is the default type of search for the Search field. It is also the fastest.
2. Regular Expression. Xcode uses the regular expression in the Search field to find matching items. For example, to find all C implementation and header files in a source group, enter `\.(c|h)$`
3. Wildcard Pattern. Xcode uses the wildcard pattern in the Search field to find items that contain the specified characters anywhere in any of the visible columns. For example, enter *View*.h to find all header files with "View" in their name.

For example, in a large list of source files, you can find all of the files with the word "dialog" in their title by choosing a String Matching search and typing "dialog" in the Search field, as shown here.

Figure 4-6 Searching for files with "dialog" in their name



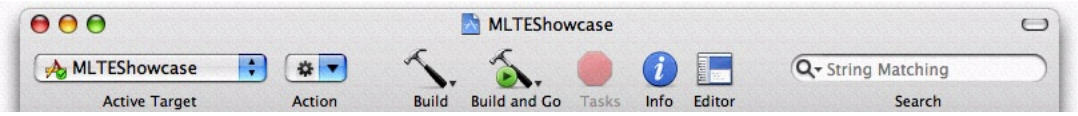
As you type, the status bar displays the scope of the search—the current selection in the Groups & Files list—and the number of items found. Pressing the "Home" key or choosing the project item (indicated by the project icon) from the search field's pop-up menu changes the focus of the search field to the whole project.

By default, Xcode clears the Search field when you select a different smart group in the Groups & Files list. To tell Xcode to preserve the contents of the Search field, open the General pane of Xcode Preferences and disable the "Automatically clear smart group filter" option in the Environment options.

The Project Window Toolbar

The project window toolbar gives you quick access to the most common Xcode commands. The project window toolbar for the Default layout contains the following items:

Figure 4-7 The project window toolbar



- Active Target pop-up menu. The target menu lists the **active target**, which is the target that is used whenever you build the project. You can change the active target using this menu.
- The Action button. The Action button lets you perform common operations on the currently selected item in the project window. The actions available from this button are those appropriate for the selected item; they are the same actions available in the contextual menu that appears when you Control-click the selected item. For example, when the current selection is a file, available operations include opening the file in a separate editor, performing version control operations, and grouping files.
- The Build buttons. The Build buttons initiate common build actions, such as building, cleaning, and running. The triangle at the bottom of the Build buttons indicates that there are multiple actions associated with them. A single click on one of these buttons performs the action represented by the button's icon. If you press and hold the mouse button over one of these buttons, you get a pull down menu of all of the actions associated with the button. You can initiate any of these actions by selecting it.
- Tasks button. The Tasks button allows you to stop any operation that is currently in progress in your project. The badge in the lower right corner of the Tasks button indicates the operation that is stopped when you click the button. If more than one operation is in progress, the Tasks button lets you select the one you want to stop. For example, if you have both a build and a search running, you can stop either operation by pressing and holding the Tasks button. Xcode displays a pop-up list of the tasks currently in operation; choose a task to stop it.
- Editor button. The Editor button shows and hides the code editor in the project window. Using this editor, you can view and edit files directly in the project window. You can also view the source associated with an error or warning or with a search result. The editing features of Xcode are described further in "Editing Source Files" (page 159).
- Info button. The Info button brings up an Info window, allowing you to examine and edit groups, files, targets, and other items in your project. See "Inspector and Info Windows" (page 73) for more information on inspecting items in your project.
- The Search field. The Search field allows you to search the items currently displayed in the detail view. As you type, Xcode filters the list of items in the detail view to include only those items with matching content in one of the visible columns. See "The Detail View" (page 60) for more information on using the Search field to find items in the detail view.

You can customize the project window's toolbar by choosing View > Customize Toolbar. Drag one or more toolbar items to or from the toolbar to get the set that is most useful to you.

The Project Window Status Bar

The project window status bar lets you view the progress of the current operation in Xcode. It gives you feedback during potentially lengthy tasks, such as building, as well as displaying the results of those tasks. In particular, the status bar lets you quickly access important information about project operations. From the status bar, you can:

- Click the progress indicator during an operation to open a more detailed account of the currently running operations in the Activity Viewer window, described in "Viewing the Progress of Operations in Xcode" (page 74).
- Click the build result message, or error or warning icon, to open the Build Results window and view build system commands and output. For more information on the ways in which Xcode displays the status of build operations, see "Viewing Build Status" (page 305).

Project Window Layouts

There are many factors affecting the optimal workspace arrangement for you. How much screen real estate do you have? What do you spend most of your time working on? How many projects do you normally have open at once?

Configuring your development environment to allow you to be as productive as possible is critical. Whatever your preferred workflow, Xcode provides several different project window layouts for you to choose from. Xcode defines the following layouts:

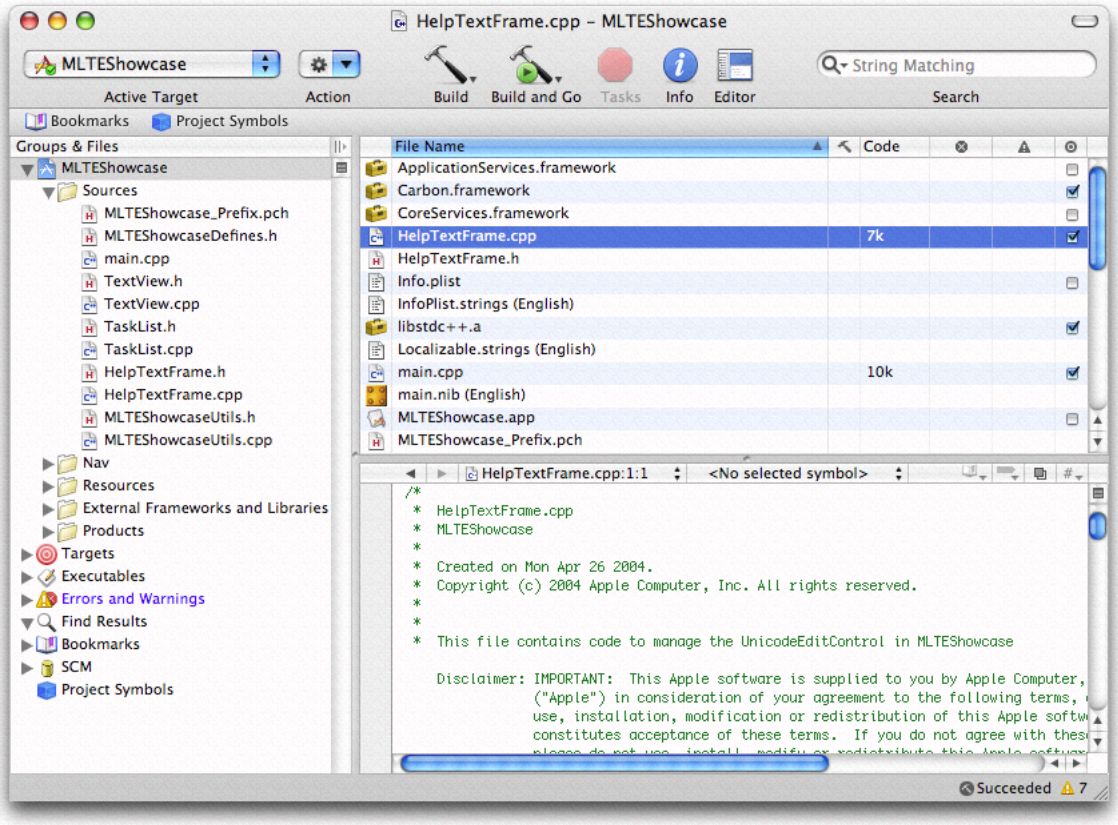
- Default. This configuration provides the traditional Xcode project window experience, described in the previous section. This is Xcode's default layout, combining outline and detail views to let you quickly navigate your project.
- Condensed. This layout provides a smaller, simpler project window with an outline view of your project contents and separate windows for common development tasks, such as debugging and building.
- All-in-One. This layout provides a single project window that lets you perform all of the tasks typical of software development—such as debugging, viewing build results, searching, and so forth—within a single window.

This section describes each project window layout and the differences between them.

The Default Layout

Xcode's default layout provides the project window described in "The Project Window and its Components" (page 55). As mentioned there, the default project window contains:

Figure 4-8 The Default Workspace project window



1. The Groups & Files list gives you an outline view of your project contents, as described in "The Groups & Files List" (page 56).

2. The detail view shows a flat list of the items selected in the Groups & Files list, as described in "The Detail View" (page 60). In the Default layout, you can hide the detail view by collapsing the project window. To collapse the detail view, double-click the button above the split-view control in the Groups & Files list. Double-clicking the button a second time shows the detail view again.

   In addition, you can customize the toolbar shown for the project window in each of these states, both collapsed and uncollapsed. To do so, collapse or expand the project window to the appropriate state and customize the toolbar in the usual way, described in "The Project Window Toolbar" (page 63). Xcode stores the contents of the toolbar for each state separately.

3. An optional attached editor. In the Default layout, you can choose to view and edit all files within the project window, using the attached editor. Or, you can choose to have Xcode use a separate editor window, and use only the Groups & Files list and the detail view in the project window.

Although you can accomplish most of your daily development tasks in the project window, Xcode also provides a number of other task-specific windows that let you focus on a particular part of the development process. Table 4-1 shows the separate windows available in the Default layout.

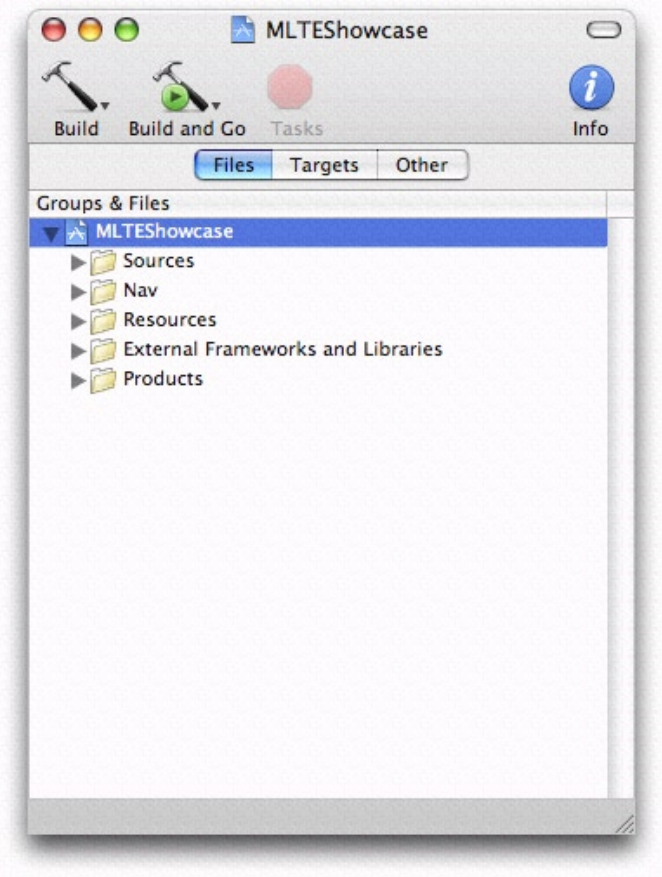
Table 4-1 Additional windows available with the Default project window layout

Window	Use to
Build Results	View the build system output generated when you build a target. To open, choose Build > Build Results or Window > Tools > Build Results. See "Viewing Build Status" (page 305).
Debugger	Debug your program; you can control execution of your code, view threads, stack frames and variables, and so forth. To open, choose Debug > Debugger or Window > Tools > Debugger. See "The Debugger Window" (page 347).
SCM	View the status of only those files under version control. To open, choose SCM > SCM or Window > Tools > SCM. See "Viewing File Status" (page 211).
Project Find	Search for text, symbol definitions, and regular expressions in your project. To open, choose Find > Find in Project or Window > Tools > Project Find. See "Searching in a Project" (page 97).
Debugger Console	Interact with the debugger on the command line and see debugger commands and output. To open, choose Debug > Console Log or Window > Tools > Debugger Console.
Run Log	View information or messages logged by your program when running in Xcode. To open, choose Debug > Run Log or Window > Tools > Run Log.
Class Browser	View the class hierarchy of your project and browse classes and class members. To open, choose Project > Class Browser or Window > Tools > Class Browser. See "Viewing Your Class Hierarchy" (page 107).
Breakpoints	View and edit all breakpoints set in your project. To open, choose Debug > Breakpoints or Window > Tools > Breakpoints.
Bookmarks	View your project's bookmarked locations in a dedicated window. To open, choose Window > Tools > Bookmarks or double-click the Bookmarks smart group.

The Condensed Layout

The Condensed layout provides a smaller, more compact version of the project window. In this configuration, the project window contains several views, each showing a different subset of the items in your project in the Groups & Files list. You can switch between these outline views using the tabs at the top of the project window.

Figure 4-9 The Condensed Project Workspace project window



The condensed project window contains:

1. The Files pane shows your project and all of the source groups and files in your project.
2. The Targets pane shows the targets and executables defined in your project.
3. The Other pane shows all of the remaining smart groups. This includes the standard smart groups defined by Xcode, as well as any smart groups you have added to the project.

You can show any of the built-in smart groups, opening the appropriate pane if necessary, by choosing an item from the View > Show menu.

The toolbar of the Condensed project window layout is also simpler than that of the other available layouts. By default, the condensed project window toolbar contains only the Build buttons, Tasks button, and Info button. These buttons are described in "The Project Window Toolbar" (page 63).

The Condensed layout provides the same additional windows as the Default layout, listed in Table 4-1 (page 66). The Condensed layout also includes the following additional windows:

Table 4-2 Additional windows available with the Condensed project layout

Window	Use to
Editor	Edit project files. Although each of the available layouts let you open files in a separate editor window, the condensed project window is the only one that does not include an attached editor; when you open a file from the project window, Xcode opens a new editor window.
Detail	View and search your project's contents in a simple list. The Condensed layout does not include a detail view in the project window; you can open a separate Detail window that includes a Groups & Files list on the left side of the window and a detail view on the right side. The Groups & Files list is tabbed to allow you to see different parts of your project. To open choose View > Detail or Window > Tools > Detail.

The All-In-One Layout

The All-In-One project window layout provides a single project window in which you can perform all of the tasks necessary for software development. In this project window, you can edit files, view project items in an outline view or detail view, view build system output, run and debug your executable, search and more. The project window of the all-in-one configuration provides three different views, or **pages**. To switch between pages, use the Page control in the project window toolbar, shown in Figure 4-10. The available pages are:

1. Project. This page lets you perform general project management tasks, such as searching, sorting and viewing SCM status. To open the project page, click the project icon.
2. Build. This page lets you view build results and the run log. To open the build page, click the hammer icon.
3. Debug. This page includes an integrated debugger view, similar to the standalone debugger window available with the other layouts. To open the debug page, click the spray can icon. For a description of the debugger interface, see "The Debugger Window" (page 347).

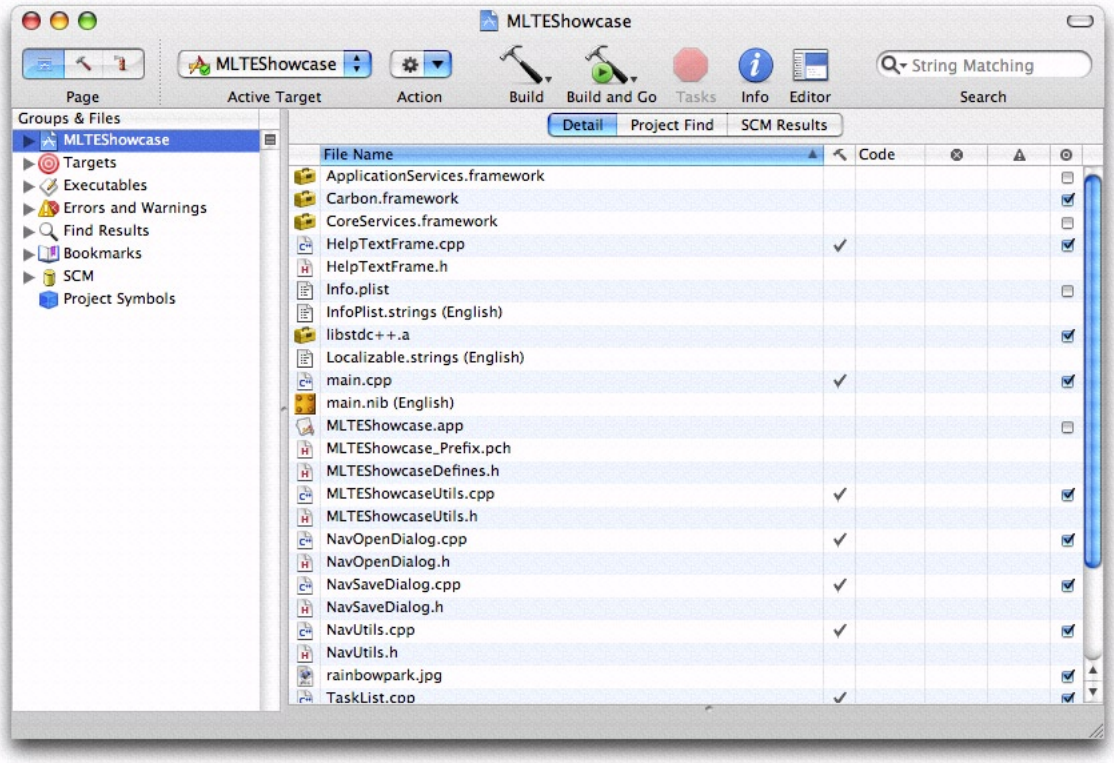
In addition to using the Page control, choosing any of the menu items for opening other Xcode windows will, in the All-In-One layout, open the appropriate page to the correct pane. For example, choosing Build > Build Results opens the Build page and selects the Build pane.

Each page of the All-In-One layout has a different project window toolbar, that contains items specific to the development tasks performed in that page.

The Project Page

The project page, shown here, lets you perform typical project management tasks. You can view the contents of your project in outline view, search for project items in the detail view, perform a project-wide find, and view status for the files under version control in your project.

Figure 4-10 The project page of the All-In-One project window



The project page contains a Groups & Files list, which shows all of the contents of your project in outline view; an attached editor, which lets you edit source files right in the project window; and a tabbed view, which lets you switch between several panes, each of which provides an interface for a common project management task. These panes are:

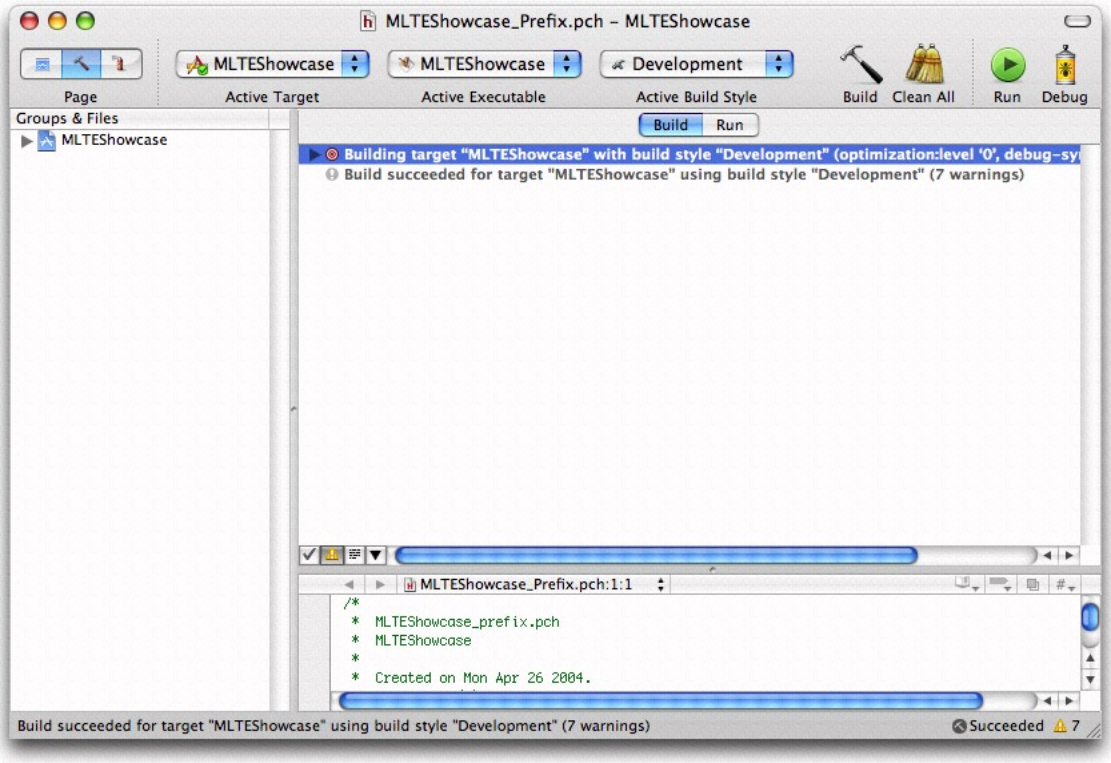
- The Detail pane includes a detail view, which lets you view additional information about project items selected in the Group & Files list or quickly filter project items. The detail view is described in "The Detail View" (page 60).
- The Project Find pane lets you perform project-wide searches and view search results. The interface is the same as that provided by the Project Find window in other layouts. See "Searching in a Project" (page 97).
- The SCM Results pane opens a dedicated view displaying only those project items under version control and their status. It is similar to what you see in the SCM window with other layouts. See "Viewing File Status" (page 211).

The default toolbar for the project page of the All-In-One layout is the same as that of the Default project window, described in "The Project Window Toolbar" (page 63), with the addition of the Page control.

The Build Page

The build page, shown here, provides an interface for common build tasks. It lets you see the progress of your build, view errors and warnings, and see the run log.

Figure 4-11 The build page of the All-In-One project window



The build page contains an attached editor, that lets you view source code and jump to the location of any build errors; a Groups & Files list that shows your project contents in an outline; and tabs that let you switch between:

1. The Build pane displays the commands used to build your project and the output of the build system. This is the same information as that in the Build Results window available with the other layouts. See "Viewing Build Status" (page 305).
2. The Run pane displays any information logged by your program while running in Xcode.

If the outline view is not visible in the build page, you can open it by dragging or double-clicking the splitter at the far left side of the window, or by choosing an item from the View > Show menu.

The default toolbar of the build page contains items that give you easy access to commands you commonly use when building and running.

Other Windows in the All-In-One Layout

The All-In-One layout is designed to let you perform all development tasks in the project window; it does, however, include a few additional windows, listed in Table 4-3. These windows let you view content already available from the project window in a separate window, should you choose to do so.

Table 4-3 Additional windows available with the All-In-One layout

Window	Use to
Class Browser	View the class hierarchy of your project and browse classes and class members. To open, choose Project > Class Browser or Window > Tools > Class Browser. See "Viewing Your Class Hierarchy" (page 107).
Breakpoints	View and edit all breakpoints set in your project. To open, choose Debug > Breakpoints or Window > Tools > Breakpoints.
Bookmarks	View your project's bookmarked locations in a dedicated window. To open, choose Window > Tools > Bookmarks or double-click the Bookmarks smart group.
Run Log	View the contents of the run log, available in the Run pane of the build page, in a separate window. To open, choose Window > Tools > Run Log.
Project Find	Perform a project-wide search and view search results in a separate window. This window shows the same information as the Project Find pane of the project page. To open, choose Window > Tools > Project Find.
SCM	View the status of files under version control in your project. This window contains the same information as the SCM pane of the project page. To open, choose Window > Tools > SCM.

Changing the Project Window Layout

You can change the current project window layout in the General pane of Xcode Preferences. From the Layout menu, choose the Default, Condensed, or All-In-One layouts. Selecting a layout from this menu shows a brief description of the layout below the menu. Note that you cannot change the project window layout when any projects are open; you must first close all open projects. The project window layout is a user-specific setting; it applies to all projects that you open.

Saving Changes to the Current Layout

Xcode's available project window layouts give you the flexibility to choose the configuration that best suits your preferred workflow. You can further customize your work environment by saving the changes that you make to the windows in an open project and applying them to all projects using that layout.

For example, the project window of the condensed layout shows three different outline views, each of them focused on a different subset of a project's groups. By default, the Files view shows only the project group, which contains all files and folders in the project. The Other view shows all smart groups. If you want access to both your project files and your bookmarked locations in the same view, you could add the Bookmarks smart group to the Files view by choosing View > Show > Bookmarks.

Note: If the Bookm arks smart group exists in another pane, choosing View > Show > Bookmarks opens that pane in the project window. However, if you have previously deleted the smart group from the project window, choosing View > Show > Bookmarks adds that smart group to the current pane.

To save this change, and have the Bookmarks smart group appear in the Files pane for all projects using the Condensed layout, choose Window > Defaults.

In the dialog that Xcode displays, click Make Layout Default to save your changes to the current layout. Clicking Restore to Factory Default discards all of your changes—both current changes and those you've previously saved—to the current layout. Xcode restores the original configuration settings for the layout.

You can save changes to:

- Window size and position.
- Visibility of views—whether they are hidden or revealed—in a window.
- Contents and visibility of outline views.
- The default set of toolbar items.

If the "Save window state" option in the General pane of Xcode Preferences is enabled, Xcode saves the state of the open windows for each project when you close that project. However, when you choose Window > Defaults, you save configuration changes that apply to all projects when you open them using the modified layout.

Viewing Additional Information on Project Items and Operations

Knowing how to use Xcode's user interface to find and view project items and information is essential to working efficiently in Xcode. Xcode gives you a number of different ways to find and access project contents. In previous sections, you've learned how to use the Groups & Files list to see an organized outline view of your project and how to use the detail view to quickly filter project contents. Using these tools, you can view as wide or as narrow a cross-section of your project as you choose. However, these tools aren't as useful when you wish to view or modify individual items in your project. For this, Xcode provides inspector and Info windows.

In previous chapters you also learned how the project window status bar lets you view the progress and results of operations in Xcode. However, the amount of information visible in the status bar is limited and it reflects only operations in the current project. To let you view a more detailed account of all operations in Xcode, Xcode provides the Activity Viewer window.

This section describes the Info and inspector windows, as well as the Activity Viewer and shows you how to use them to obtain additional information on project items and Xcode operations.

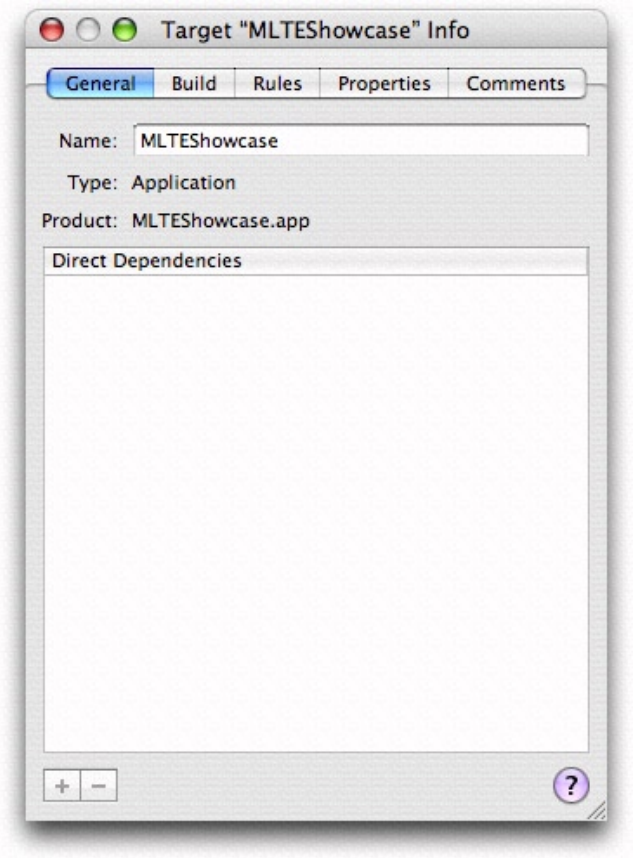
Inspector and Info Windows

Xcode includes two special types of windows that you will use frequently to view and modify items that appear in your project window. These are the Info window and the inspector window. The Info and inspector windows allow you to examine the components of your project in detail and make changes to them. You can inspect any item that is selected in the project window or is open in the active editor window. Xcode provides inspectors for the following project items:

- File and folder references
- User-defined smart groups
- Localized file variants
- Targets
- Executables
- Build phases
- The project itself

The type of information displayed in the inspector and Info windows changes, depending on the type of item you are inspecting. For example, here is the Info window for a target:

Figure 4-12 An Info window



For any particular item in your project, the Info window and inspector window both display the same information. However, the two windows behave differently. The Info window continues to display information about the item that was selected in the project window when you opened it, regardless of the current selection. You can have multiple Info windows open at any time, each describing a different component of your project, and that information will continue to be visible until you close those windows. To open an Info window, choose File > Get Info, click the Info button in the project window toolbar, or double-click the item to inspect.

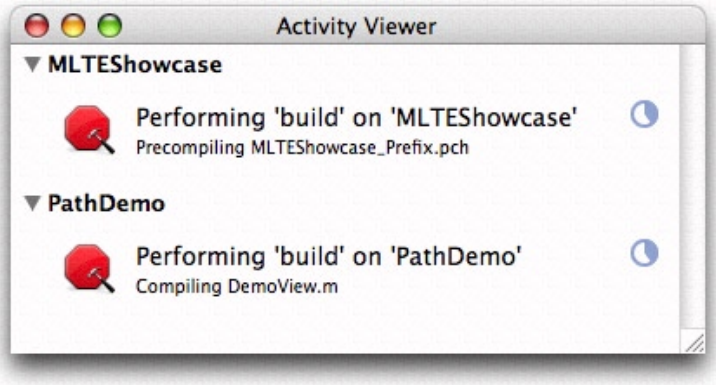
The inspector window tracks the current selection in the project window. As you select different files, targets, and groups in the project window, the inspector window changes to show information about that item. To open an inspector window, hold down the Option key and choose File > Show Inspector or click the Inspector button. The Inspector button is not in the project window toolbar by default; to add it to the toolbar, choose View > Customize Toolbar.

Specific inspector and Info windows are described in more detail later in this document, when the objects that they modify—files, targets, projects, groups, and so forth—are discussed. In general, however, if you are at a loss as to how to make a change to a basic project component, try inspecting it. Throughout this document, wherever an inspector window is used, remember that you can also use an Info window, and vice versa.

Viewing the Progress of Operations in Xcode

The Activity Viewer window lets you watch the operations currently in progress in Xcode. While the Tasks button in the project window toolbar lets you control the progress of tasks in the current project, the Activity Viewer lets you see the progress of all Xcode operations across all open projects. The Activity Viewer provides a single, persistent window which you can leave open to monitor the progress of all running tasks. To open the Activity Window, shown here, choose Window > Activity Viewer or click the progress indicator in the project window status bar.

Figure 4-13 The Activity Viewer Window



The operations in the Activity Viewer are grouped by project; you can show or hide the operations specific to any project by clicking the disclosure triangle next to the project name. To stop any of the tasks shown in the Activity Viewer, simply click the Stop sign icon next to that task. You can cancel the most recently initiated operation in the active project by choosing Project > Stop <*task name*> or typing Command—period.

Files in a Project

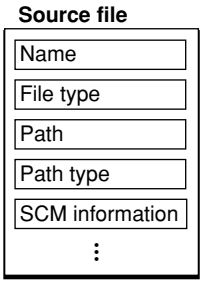
The files in a project are the fundamental building blocks from which you create your end product. Files contain the source code that you write and serve as the inputs to the build system for creating a product. They can also hold notes, performance metrics, and the like to aid you in the development process.

This chapter discusses the files in a project, describes how Xcode references project files, and shows you how to add files, frameworks, and folders to your project. It also describes how to use source trees to set up alternative access paths for project files and how to use a cross-project reference to access the contents of another project.

Files in Xcode

For each source file included in a project, Xcode tracks file attributes, such as the name and type, as well as other information. Figure 5-1 shows the information that Xcode tracks for source files.

Figure 5-1 A source file



The name is the file system name for the file. The file type identifies the file as being one of several classifications (source file, image file, text file, and so on.) Depending on the file type, Xcode stores additional information about the file, such as the file encoding, type of line endings and so forth.

The path to the file specifies the file system location for the file; the path type—which you can modify—indicates how Xcode stores the path; that is, whether it is absolute or relative to the project directory or another location. "How Files Are Referenced" (page 77) describes the various ways in which Xcode stores paths.

You can view and edit these file attributes in the file inspector.

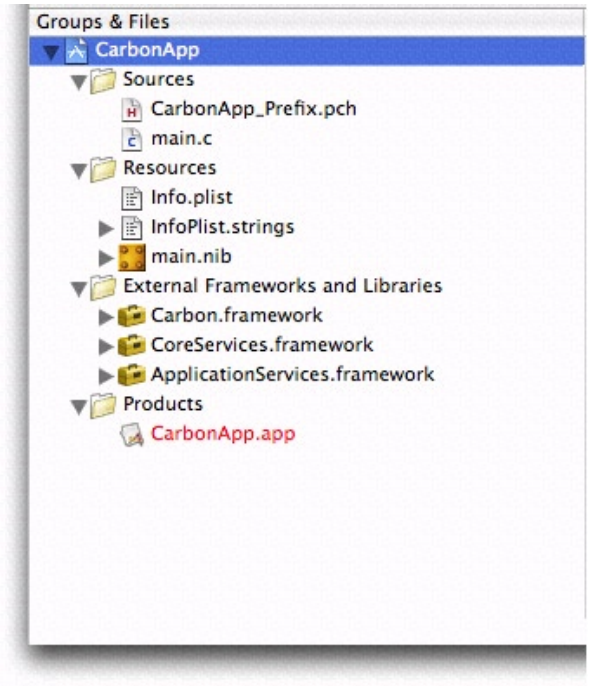
The Files in a Project

The project lets you pull together all of the files and other information required to build a set of related software products. Within a project, you use a target to specify the files needed for an individual product. The files can reside at any location in your file system; they do not need to be placed in your project folder. A project can contain:

- Files. A source file is any file that Xcode uses in building a target, including source code files, resource files, image files, and others. For files you need direct access to in Xcode—for example, files you wish to edit using Xcode's editor—you should explicitly add a reference to each file to the project. This includes source code files you want Xcode to compile.
- Folders. If you have a folder of files that you manipulate as a whole—such as a folder of help files—you can simply add a reference to the folder to your project. This allows you to manipulate the folder in Xcode instead of touching each file individually. (To access any of the files individually from Xcode, you must also add a reference to the file to your project.)
- Frameworks. You can add a reference to each of the frameworks that your product links against. This gives you easy access to the framework's headers, directly in the project window.

When you create a project using Xcode's project templates, described in "Choosing a Project Template" (page 47), Xcode populates the project with a small set of default files required to build the associated product. For example, the figure below shows the contents of a new project created using the Carbon Application project template. This project builds a small C application with a NIB-based interface that links to the Carbon framework. The project contents have been expanded in the Groups & Files list to display its contents in outline view. Of course, keep in mind that the contents of a project vary depending on the project template and the products it creates.

Figure 5-2 The project contents in the Groups & Files list



The example project contains the following items:

- The Sources group contains implementation files; in this case the `main.c` file.
- The Resources group contains resource files for the application. This includes the `main.nib` file that defines the user interface, the `Info.plist` property list file, and the `Infoplist.strings` files containing strings used in the interface.
- The External Frameworks and Libraries group contains references to the frameworks that define the system interfaces used by the application's code. You can view a framework's header files by disclosing the contents of the framework in the Groups & Files list.
- The Products group contains references to the products created when the project's targets are built. A **product reference** is a special type of file reference that refers to the build system output for a particular target. A product reference lets you view your target's products right in the Groups & Files list. You can use the product reference to refer to the product in the same way you use a file reference to refer to a project file. Note, however, that the product reference does not actually refer to anything until you have built that target.

Xcode keeps a reference to each file, folder, and framework you add to your project. In this way, Xcode can find your files directly when it builds a product. However, Xcode also provides build settings for specifying general search paths for various items, such as headers and libraries. These include the Header Search Paths, Library Search Paths, and Framework Search Path build settings.

How Files Are Referenced

Xcode stores the location, or path, for each file, framework, and folder in a project. Xcode uses this path to locate the item. Xcode can store this as an absolute path or relative to another file system location. You choose the way that a given file, framework, or folder is referenced when you add it to the project. You can also change the reference type for an item in the file inspector. Xcode supports the following reference styles, each of which is available in the Reference Type menu:

- Relative to Enclosing Group. The path is relative to the folder associated with the file's group. If the file is not in a group or the group has no associated folder, the path is relative to the project's folder. This is the default setting for files in your project's folder.
- Relative to Project. The path is relative to the project's folder, regardless of whether the file is in a group with an associated folder.
- Relative to Build Product. The path is relative to the folder that contains the project's build products. This reference style is the default for items that are created by one of the project's targets.
- Relative to *source path*. The path is relative to a user-defined source path. You can define a source path in the Source Trees pane of Xcode Preferences. Note that this reference type is not available to you until you have defined at least one source tree.
- Absolute Path. The path is absolute from the root directory (/). This is useful in a limited set of circumstances, when you want to locate a file at a particular path. In most cases, you should use a relative path; absolute paths are fragile and break easily when you move projects between computers.

If a file is inside your project folder or its build folder (created by Xcode when it creates a new project), use one of the first three reference styles.

If Xcode can't find a file, folder, or framework at the path stored for it in the project, Xcode displays the item in red in the project window.

Adding Files, Frameworks, and Folders to a Project

If you created a new project using one of the project templates, or if you converted an existing project, your project will already have a number of groups and files in it, as well as frameworks, folders, and product references. Regardless of whether your project already includes files or was created completely empty, you will likely have to add files or frameworks to your project at some point.

You can add a file, folder, or framework to your project by dragging it from the Finder to the project source group or by choosing Project > Add to Project and using the navigation dialog to choose the item to add.

Adding Files and Folders

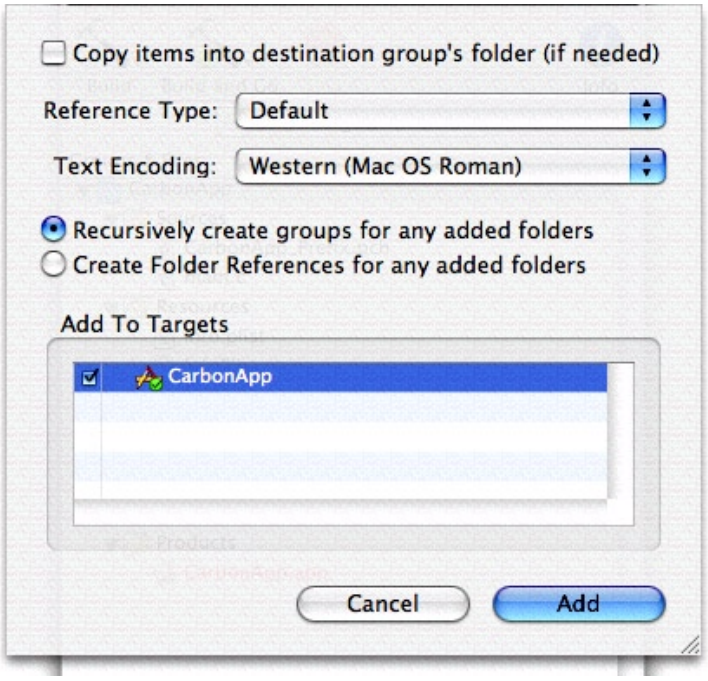
There are two ways for you to add files or folders to your project:

- In the project window Groups & Files list, select the group to add the files to, and choose Project > Add to Project. Use the resulting dialog to navigate to and choose the file or files to add. If you want to add all of the files in a given folder, you may simply choose the folder. The files or folders you add are placed after the items currently selected in the Groups & Files list, if any.
- Drag the icons for the files or folders from the Finder to the project window Groups & Files list. A line shows you where the files will be added.

As a shortcut, you can add a file that is open in an editor window to the project by selecting Project > Add Current File to Project. The editor window must have focus.

Once you have selected the file or files to add to the project, using either of the two methods described above, Xcode displays a dialog, shown below, that allows you to specify how the files are added to the project.

Figure 5-3 Adding files to a project



Here is what the dialog contains:

1. The "Copy items into destination group's folder (if needed)" option controls whether or not the files are copied into your project folder on disk. If you select this option, Xcode copies over any files that are not already present in the project folder. If the project folder contains subfolders for groups, then the files are copied into the appropriate subfolder.

2. The Reference Type menu specifies how the location of the file is stored. For a description of the various reference styles available to you, see "How Files Are Referenced" (page 77). Note that this menu does not contain any source paths until you have defined one or more source trees in the Xcode Preferences window. Once you have defined a source path, it appears at the bottom of the Reference Type menu and you can choose it for the files and folders you add.

3. The Text Encoding menu specifies the encoding for the file or files. This is the character set that Xcode uses to display and save a file. For more information on file encodings, see "Choosing File Encodings" (page 163).

4. The Add To Targets group allows you to add the file to one or more of the targets currently defined in your project. If the checkbox next to a target name is checked, the file is added to that target when it is added to the project. When you add a file to a target, that file is built when the target is built. You can specify which files are included in a target at any time; this option allows you to add the file to your project and to any necessary targets in one step.

The remaining options apply only if the selection of files to add to the project includes one or more folders. Xcode can add folders in two ways:

1. Group. Xcode recursively creates groups for the folder and its subfolders. Each of the files in these folders is added to the project and is placed in the group for the appropriate folder. If you choose to copy the files into the project's folder, Xcode duplicates the folder hierarchy. If you move a file to the folder outside of Xcode, Xcode does not add the file to the project.

   To add a folder as a group, select "Recursively create groups for any added folders."

2. Folder Reference. Xcode adds the folder itself to the project but not its contents. This is useful if you need to manipulate the folder as a whole but not the individual items within it. One example is a folder of help files that you edit outside of Xcode and that you want Xcode to move into the application's Resources folder when you build the application.

   To add a folder as a folder reference, select "Create Folder References for any added folders."

Adding Frameworks

Similar to adding files, you add frameworks to a project by doing either of the following:

- Select the group to add the framework to in the Groups & Files list, and choose Project > Add to Project. Use the resulting dialog to navigate to and choose the appropriate framework on the system—for example, `AddressBook.framework`.
- Drag the framework to the project window Groups & Files list.

After you have selected a framework, Xcode presents the same options described in "Adding Files and Folders" (page 78). The following options apply to frameworks:

- The Reference Type menu in the dialog specifies how the location of the framework is stored. For a description of the various reference styles available to you, see "How Files Are Referenced" (page 77).
- The Add to Targets group box allows you to add the framework to one or more of the targets currently defined in your project. If the checkbox next to a target name is checked, the framework is also included in that target when it is added to the project.
- The Text Encoding menu specifies the encoding used for the files in the framework. For more information on file encodings, see "Choosing File Encodings" (page 163).

Removing Files

You can remove any files, folders, or frameworks from your project by selecting them in the Groups & Files list and pressing Delete. You can also select the files to remove and choose Edit > Delete.

Xcode displays a dialog asking whether you wish to delete the actual files or just the project's references to them. If you choose Delete References, Xcode deletes only the project's references to those files; the files remain intact on the disk. If you choose Delete References & Files, Xcode deletes the references from the project, and deletes the referenced files from the disk. The files are deleted immediately; they are not moved to the Trash.

Source Trees

Source trees are root paths that can be used to define common access paths and locations for target outputs. A source tree defines a name and a location on the local file system. When you add files and folders to your project, you can specify their location relative to any source tree defined for your computer. Xcode stores the file reference relative to this source tree. Any other user who has the same source tree defined is able to work on the same project seamlessly, provided that the file actually exists at the source tree location on their computer as well.

Source trees let you keep common resources in locations other than the project folder of an individual project and still transfer projects back and forth between team members and their various computers without breaking the project's file references. This is particularly useful if you have a set of common files or resources that are used in a number of projects, and therefore cannot live in the project folder. Everyone working on a common project should have the same source trees defined; while the locations assigned to those source trees may differ, the names must be the same in order for Xcode to locate the necessary files and materials on the developer's computer.

Xcode supports global source trees; that is, any source trees that you define are available to all of your projects. Source trees are stored for each user, so if you have multiple developers using a single computer, you will have to define the source trees for each user, even though the location for those source trees is the same. Once you have defined a source tree, it is available to you from the Add Files dialog, to use when adding file, folder, and framework references to your project. You can also select the source tree from the Path Type pop-up menu in the file inspector, described in "Inspecting File, Folder, and Framework References" (page 161).

You can edit source trees in the Source Trees pane of Xcode Preferences. To open this pane, choose Xcode > Preferences and click Source Trees. To add a source tree, click the plus (+) button beneath the source tree table. Xcode adds an entry in the table. Add the following information to the entry:

- Setting Name is the name of the source tree. This name must be the same for all users who wish to use this same source tree to refer to common files.
- Display Name is the name that Xcode shows for the source tree in dialogs, inspectors, and anywhere else the source tree is used in the user interface. For example, this is the name used in the Path Type menu of the file inspector.
- Path is the full path to the files and other resources located using this source tree on the user's system. This path may vary from computer to computer, and from user to user.

To delete a source tree, select the source tree in the table and click the minus (-) button. To edit a source tree, double-click the entry for the source tree in the appropriate table column and type the changed text.

Referencing Other Projects

In addition to file, framework, and folder references, Xcode projects can contain a cross-project reference; that is, they can refer to another project outside of the current one. It is not always feasible or desirable to keep all related targets and products in a single project. However, you may still need to reference targets or products that reside in a different project. For example, you may have several applications that rely on a

common framework that resides in a different project. In this case, you can add a reference to the project containing the framework to the project containing the application. This reference, called a **cross-project reference**, lets you access the targets and products of the referenced project from your current project.

To create a reference to another project, choose Project > Add to Project and select the project bundle (the `.xcode` bundle) of the project you wish to reference. Xcode adds a reference to the source group for your current project, visible in the Groups & Files list. The project reference is indicated by the project icon. Clicking the disclosure triangle next to the project reference shows the product references that the other project contains. These product references can be added to targets in the current project.

You can relate targets in the current project to targets in the referenced project by creating a target dependency. You can add a dependency on a target in the referenced project in the same way that you would add a dependency to a target within the same project. See "Adding a Target Dependency" (page 239) to learn more about target dependencies.

For projects that use cross-project references, you should use a common build location; doing so ensures that Xcode can automatically locate products created by targets in those projects. For more on build locations, see "Build Locations" (page 301).

Organizing Xcode Projects

Some of the organizational decisions you make when you first set up a project—such as how many targets you need for your software development effort—affect your entire development experience. This chapter provides tips and tricks for organizing your software as you develop with Xcode. It also describes some of the features Xcode provides that let you group and organize information in the user interface for rapid and easy access. For instance, you can save commonly accessed locations as bookmarks or in the Favorites bar, or document project items by adding comments to them.

Software Organization Tips

The following are some general guidelines for organizing your software in Xcode. In subsequent sections, you'll get more detailed information to flesh out these tips.

- Follow standard software development practice to factor your software into logical units of reasonable size, which you can implement as applications, shared libraries, tools, plug-ins, and so on. In Xcode, each of these products requires one target.
- Put together projects and targets based on such factors as:
  - How do the products interact?
  - How many individuals (or teams) will be working on each task?
  - Do the products use different source code management systems?
  - Must the products run on different versions of Mac OS X?
- For smaller development tasks, it generally makes sense to keep targets for related products in a single project.
- For tasks that are reasonably separate, and especially if they are to be implemented by separate teams, use multiple projects.
- Use cross-project dependencies when targets in one project need to depend on targets in other projects.
- Use the information in "Build Locations" (page 301) to ensure that a target can access the build products of other targets when needed.

Dividing Your Work Into Projects and Targets

To develop software with Xcode, you are going to need at least one project, containing at least one target, and producing at least one product. Those are the basic structures you use for all your development.

Beyond that, there are no hard and fast rules for how you divide your work. For simple products or products that are closely tied together, you might use a single project with multiple targets. For large development tasks with many products, and especially with separate development teams, you'll want to use multiple projects and targets, perhaps connected with cross-project dependencies.

The following sections provide additional information and tips on organizing your software.

Identifying the Scope

In organizing your software, many decisions depend on the scope of the design goal and the number of products it requires. For example, if you're working alone on a simple application, you can create a project for the desired application type (such as a Cocoa or Carbon application) and get right to work. Here are some of the decisions you might face:

- If you decide to move some code to an internal library, you might add a target for the library.
- To take advantage of existing code in another project, you might choose to develop the existing code as a shared library and add a dependency so that the application project depends on the shared library project and makes use of its output.
- You might choose not to use a source code management (SCM) system, although even simple projects can sometimes benefit from such use.

Suppose, however, that you are working on a more complex software design, one that will be implemented by several individuals or development teams. Let's say you are asked to create a new application for "an easy-to-use recording studio." You may already have components of this application, such as a shared library for presenting music tracks. As you refine the product specification, you identify other common tasks that might belong in a shared library, tool, or companion application. You determine that the main application should rely on Apple technologies to provide certain features, such as burning CDs or making the application scriptable.

Eventually, you have identified a set of products to build, which gives you a good idea of the scope of the task. And that in turn can help you determine how to organize it into Xcode projects and targets.

Trade-offs of Putting Many Targets in One Project

You can help determine whether to put many or all of your targets in one project by considering some of the trade-offs involved. Here are some of the advantages of combining multiple targets in one project:

- Indexing works across all targets.

  Indexing information is required to access classes in the class browser, view symbols in the Project Symbols smart group, and to take full advantage of code completion. It is also necessary to use Command-double-click to jump to a definition and to use symbolic counterparts.
- You can easily set up dependencies between targets in the project.
- Anyone using the project has access to all its files.
- If you have access to extra CPUs on your network, you can use distributed builds to build a large project more rapidly.

Here are some of the disadvantages of putting all your targets in one project:

- All of the code is visible to any individuals or teams using the project, even if they're working on only one target or a small subset of the overall project.
- The project size may become unwieldy; this can cause Xcode to take a long time to perform operations such as indexing.
- All targets must use the same SCM system.
- All targets must build for the same Mac OS X system version (SDK).
- All targets use the same build styles (for example, you can't have variations on a Deployment build style for different targets).
- You can't use the graphical debugger to debug two executables in one project at the same time.

Trade-offs of Using Multiple Projects

There are also trade-offs in breaking up a software development task into multiple projects and targets. Here are some of the advantages of using multiple projects:

- You can use the project as a unit for dividing work among different individuals or teams. The separate projects allow you to segregate the code (for example, to limit access to confidential information).
- Each individual project can be of a more manageable size, and Xcode can perform indexing, building, and other operations more quickly.
- If projects need to share outputs, they can build into a common directory, as described in "Build Locations" (page 301).
- You can use cross-project dependencies to build other projects needed by the current project.
- Each project can use source code stored in a different SCM system. However, if individuals have physical access to other projects, it is still possible to look at SCM information from multiple projects that use the same SCM system.
- Each project can build for different Mac OS X systems (SDKs).
- Each project can define separate build styles.
- You can use the graphical debugger to debug two or more executables at the same time, one in each project. This is useful when products communicate directly or otherwise interact.

Here are some of the disadvantages of having multiple projects and targets:

- You won't have cross-project indexing, and thus you will have access only to symbols that are specifically exposed by other projects. This means, for example, that you can't automatically look up definitions in other projects unless you have physical access to them (not just to their end products).

  Similarly, you will not be able to take full advantage of other features that depend on indexing. That includes using code completion, using Command-double-click to jump to a definition, and using symbolic counterparts.
- Management of many smaller projects is likely to incur additional overhead. For example, to set up a target that depends on other targets in multiple projects, you'll first have to set up cross-project references.

  Similarly, use of multiple projects may require additional communication between teams.
- For an individual working on multiple projects, it may become unwieldy to switch between many open projects.

- Individual projects are smaller, and are less likely to be able to take advantage of distributed builds.

Organizing Files

Decisions about how to partition projects and their targets affect the organization of your entire software development effort. However, it is also important that your work in a project is organized and accessible to you. Particularly in larger projects, the number of files can be daunting. To help organize files into manageable chunks, Xcode lets you group them.

A **group** lets you collect related files together. A source group lets you group an arbitrary set of file, folder, and framework references in your project. A smart group, on the other hand, lets you group together files fitting a particular pattern or rule. This section shows you how to organize files using source and smart groups.

Organizing Files into Source Groups

In the Groups & Files list, source groups look like folders. However, they don't have to correspond to folders on the disk. You can instead arrange files into groups that make sense to you while working on them in Xcode. For example, in a project containing multiple targets, your project could store all the nib files in one folder and all of the implementation files in another folder on disk. But in the Groups & Files list, you could group the files according to target; that is, the nib files and implementation files for target A would be in one group, the nib files and implementation files for target B would be in another group, and so on. A group doesn't affect how a target is built.

Creating a New Source Group

You can create a new source group in any of the following ways:

- Create an empty group. Choose File > New Group and type the name.
- Create a group from existing items. Select the items you wish to group and choose File > Group. You can also Control-click the selected files and choose Group from the contextual menu.
- Create a new group based on an existing directory structure. Choose Project > Add to Project, select the folder, and check "Recursively create groups for added folders." This is described further in "Adding Files and Folders" (page 78).

Adding Files to a Group

You can add files to a group at any time by dragging the file icons to the group's folder in the Groups & Files list. A line appears to indicate where you are moving the files. Xcode automatically expands groups as you drag items onto them.

Deleting Groups

When you remove a group from your project, you can choose whether to remove the files in that group from the project or simply ungroup the files.

- To remove a group and the project's references to the files in that group, select the group, and choose Edit > Delete or press the Delete key.
- To remove a group and keep the files it contains,press Shift and choose File > Ungroup or Control-click the group and choose Ungroup from the contextual menu.

Using Smart Groups to Organize Files

Smart groups are also represented by folder icons in the Groups & Files list; however, they are colored purple to distinguish them from source groups. As was mentioned earlier, smart groups group files that adhere to a common pattern or rule. For example, you could define a smart group to collect all implementation files ending in `.m` or `.c`. When you make a change to your project that alters the set of files matching a smart group's rule—for example, adding a new `.c` file—the smart group automatically updates to reflect the change. You do not need to add files to a smart group yourself. In fact, Xcode does not allow you to drag files to a smart group.

Creating a New Smart Group

To create a new smart group, choose File > New Smart Group and select one of the following options

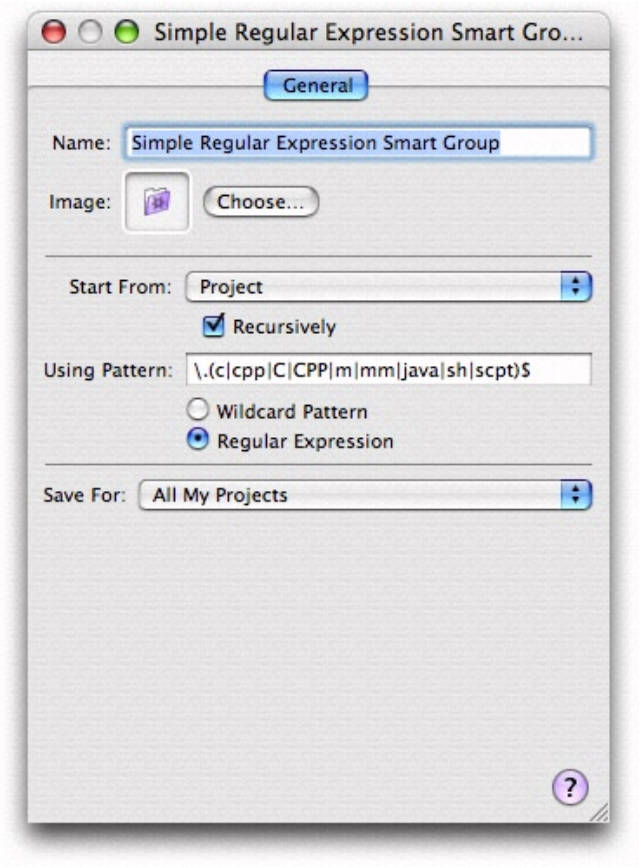
- Simple Filter Smart Group creates a smart group that collects files whose names contain a specified string.
- Simple Regular Expression Smart Group creates a smart group that uses a regular expression to specify the pattern that file names must follow.

Xcode adds a new smart group of the selected type to your project and brings up an Info window that allows you to configure the group. Xcode provides templates for each type of smart group; it uses these templates to configure new smart groups with a default set of options. For example, the default Simple Filter Smart Group collects all files with "*.nib" in their name. The default Simple Regular Expression Smart Group collects all C, C++, Objective-C, Objective-C++, Java, and AppleScript implementation files in your project.

Configuring a Smart Group

To configure a smart group, select the group in the Groups & Files list and open an Info window. The Info window for a smart group looks similar to the following figure:

Figure 6-1 Configuring a smart group



Here is what the smart group Info window contains:

1. The Name field specifies the name of the new smart group. By default, this is set to the type of the smart group, for example, "Simple Filter Smart Group."

2. The Image well displays the icon used for the smart group in the project window. The default image for a smart group is the purple folder icon, but you can also use a custom image to represent your new smart group. To change the icon, click the Choose button and navigate to the image file to use.

3. The settings that define how the smart group decides to include a file are those in the center of the smart group Info window. The Start From menu specifies the directory from which the smart group starts its search for matching files; by default, this is the project folder. If the Recursively option is enabled, the smart group also searches through subfolders of that folder.

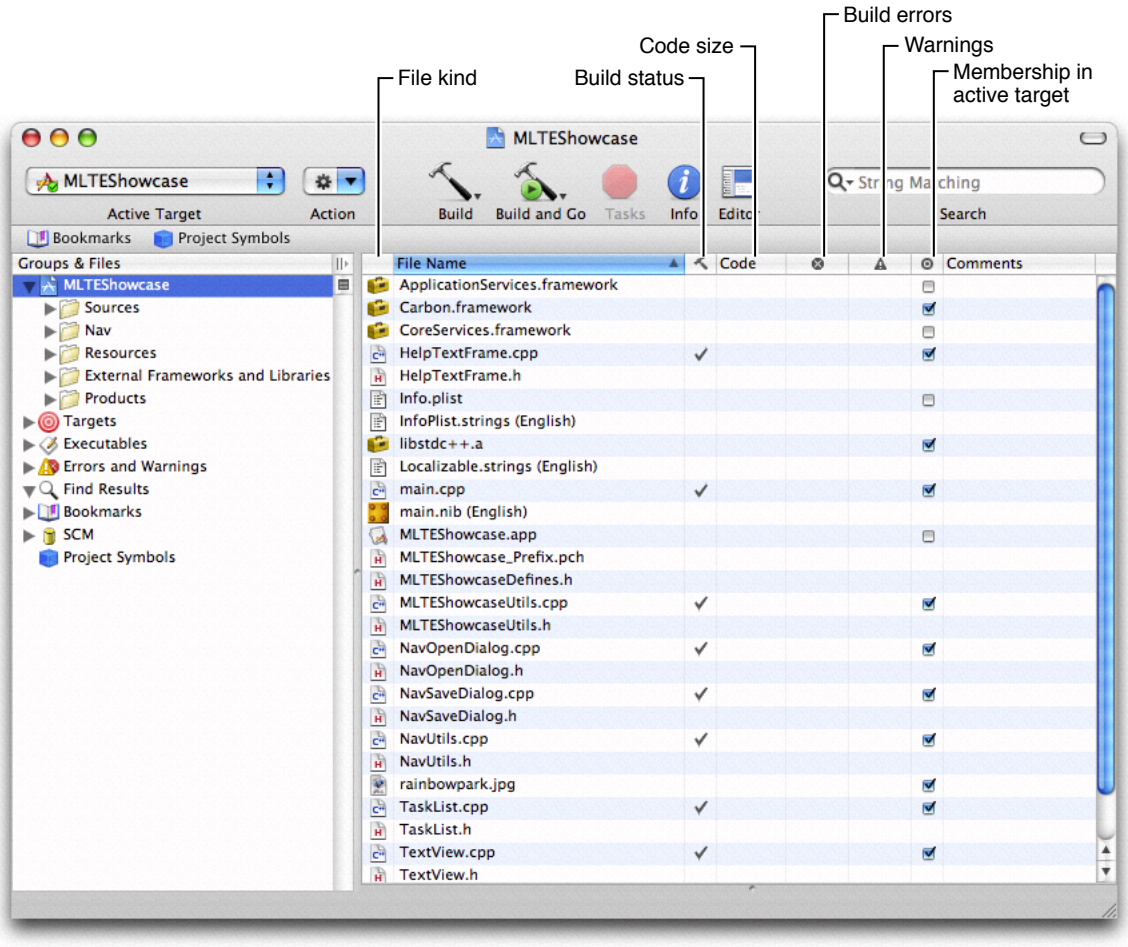
4. The Using Pattern field specifies the actual pattern that files must match in order to be included in the smart group. As mentioned, this pattern can be either a simple string that the smart group filters on or it can be a regular expression, depending on the value of the radio buttons beneath the field. For examples of each of these, look at the smart group templates. The Simple Filter Smart Group template uses the pattern `*.nib`. Any files with ".nib" in their name are included in this smart group. The Simple Regular Expression Smart Group, on the other hand, uses the regular expression `\.(c|cpp|C|CPP|m|mm|java|sh|scpt)$` to collect all implementation files ending in any one of these suffixes, regardless of the case of the filename and extension.

5. The Save For menu determines the scope for which this smart group is defined.You can have this new smart group appear in all of your projects, or you can specify that the smart group you have created remains specific to the current project.

Viewing Groups and Files

As you learned in "The Project Window" (page 55), you have two ways to view the contents of your project. You can view the groups and files in your project in outline view to see a hierarchical representation of your project organization. You can also view the groups and files in your project as a simple list, in the detail view. The detail view presents a flat list of all the files contained in the source group selected in the Groups & Files list. For example, if you select your project in the Groups & Files list, the detail view shows all of the files in the project. If you select an individual file in the Groups & Files list, only that file is displayed in the detail view. To view the contents of a source group in your project, select that group in the Groups & Files list. The following figure shows the project, selected in the Groups & Files list.

Figure 6-2 Viewing the contents of a group



All of the files in the project are displayed in the detail view. The detail view for any file shows the following information:

1. The first column, with an empty column heading, shows an icon indicating the type of the file. For example, a nib file is marked by the Interface Builder file icon. The icon for a C++ class file displays the characters "C++".
2. The File Name column displays the names of the files.
3. The column marked by the hammer icon displays the build status of each file. If a file has been changed since the active target was last built, this column displays a checkmark, indicating that the file needs to be built. If the file is up to date, this column is empty.
4. The Code column displays the size of the compiled code generated from the file.
5. The column marked by the error icon displays the number of errors in the file. If this column is empty, the file either contains no errors or has not yet been built.
6. The column marked by the warning icon displays the number of warnings for the file. If this column is empty, the file either has no warnings or has not yet been built.
7. The column marked by the target icon indicates whether the file is included in the active target. If the checkbox next to a file is checked, then the active target includes that file.
8. The column marked by the source code management (SCM) icon shows the current SCM status of the file.
9. The Path column shows the file system path to the file or folder.
10. The Comments column displays any note or other information that you have associated with the file in the Comments pane of the file inspector.

Not all of these columns are visible by default. You can choose which columns are shown in the detail view by using the View > Detail View Columns menu or by Control-clicking anywhere in the heading of one of the detail view's columns. This brings up a contextual menu from which you can choose the columns to display. For more information, see "The Detail View" (page 60).

Saving Commonly Accessed Locations

In any project, there are locations that you find yourself accessing again and again; files that you edit frequently, headers that you browse, even smart groups that you find yourself using often. Xcode lets you save locations that you access frequently and provides ways for you to quickly jump to these locations. You can bookmark project items. You can also use the Favorites bar to store locations. This section describes how to take advantage of bookmarks and the Favorites bar to provide quick access to project contents.

The Favorites Bar

The Favorites Bar in the project window lets you save commonly accessed items and return to them quickly. By default, the Favorites bar is hidden. To show the Favorites bar, choose View > Show Favorites Bar. To hide the Favorites bar, choose View > Hide Favorites Bar.

To add an item to the Favorites bar, simply drag it to the Favorites bar in the project window. You can save any of the same locations you can bookmark, including files, technical documentation, URLs, and so forth. In addition, you can add smart groups and source groups to the Favorites bar. The Groups & Files list can get quite long as you reveal the contents of more and more groups, scrolling the items at the bottom of the list out of view; you can add groups—including any of the built-in smart groups—to the Favorites bar to quickly jump to that group in the Groups & Files list. To reorder items in the Favorites bar, drag them to the desired location; dragging an item off of the Favorites bar deletes the item from the Favorites bar. To rename an item in the Favorites bar, Option-click the item and type the new name. This changes the name of the entry in the Favorites bar; it does not affect the name of the actual item that the entry represents.

To open a saved location, simply click or double-click it in the Favorites bar. If the item saved in the Favorites bar is a container, such as a group or folder, pressing the item (holding the mouse button down) brings up a pop-up menu that shows the items contents and lets you navigate to them. Each of the items in the Favorites bar serves as a proxy for the actual item. Thus, Control-clicking the item brings up a contextual menu that allows you to perform operations appropriate for the selected item.

Saving Commonly Accessed Locations as Bookmarks

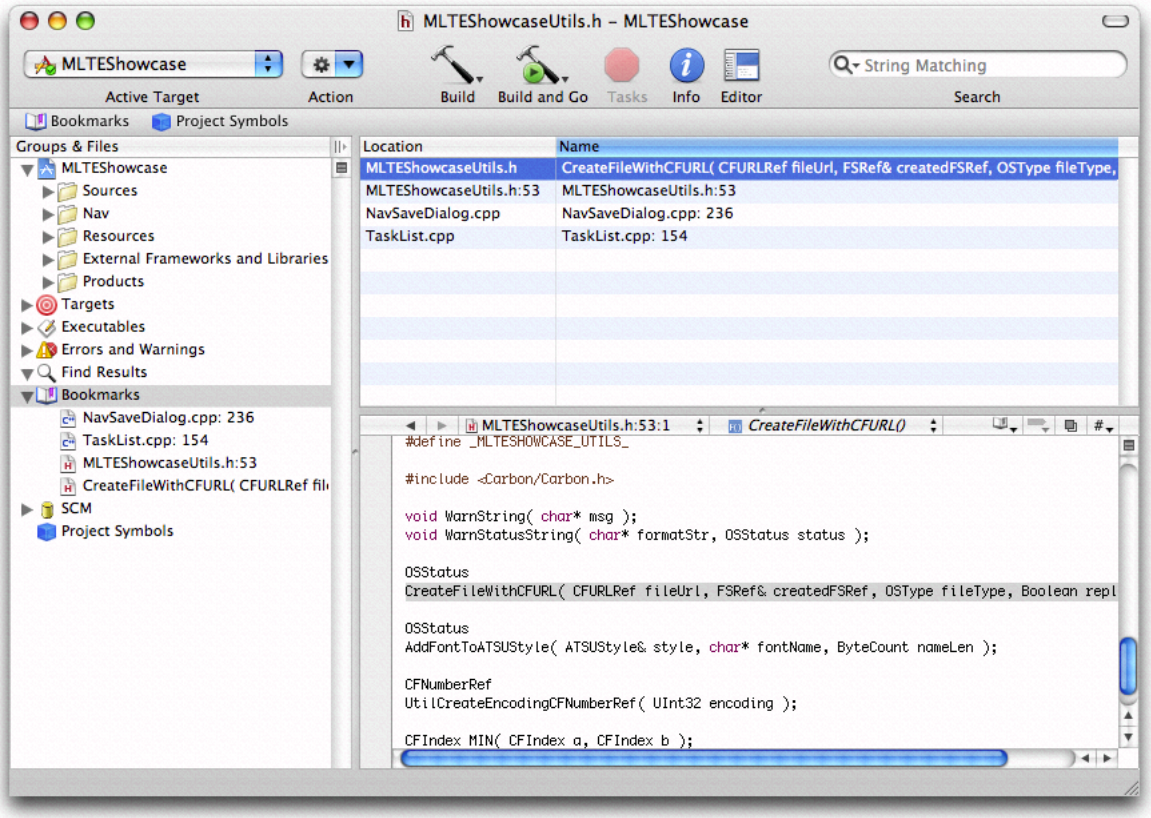
Another way that Xcode lets you organize your project and provide easy access to the information you need is with bookmarks. If you have files or locations in your project that you access often, you can bookmark those spots and return to those locations at any time simply by opening the bookmark.

To create a bookmark, open the location you want to bookmark and choose Find > Add to Bookmarks. Xcode prompts you for the name of the bookmark; you can enter a name to help you remember which location the bookmark marks, or you can use the default name suggested by Xcode. You can bookmark project files, technical documentation, URLs, and so forth.

You can see the bookmarks in your project in several ways:

- Bookmarks smart group. Select the Bookmarks group in the Groups & Files list to see the bookmarks in the detail view. The detail view shows the name and file of all of the bookmarks in your project. If the attached editor is open, selecting a bookmark opens that location in the editor. Otherwise, you can double-click the bookmark to open the bookmarked location in a separate editor window.
- Bookmarks window. To open this window, double-click the Bookmarks smart group or choose Window > Tools > Bookmarks. You can see all bookmarks in your project in this window; double-click any of these bookmarks to open the location.
- The bookmarks pop-up menu in the navigation bar of Xcode's editor. Choose any bookmarked location from this menu to open it in the editor, as described in "The Xcode Editor Interface" (page 169).

Figure 6-3 Viewing bookmarks



Adding Comments to Project Items

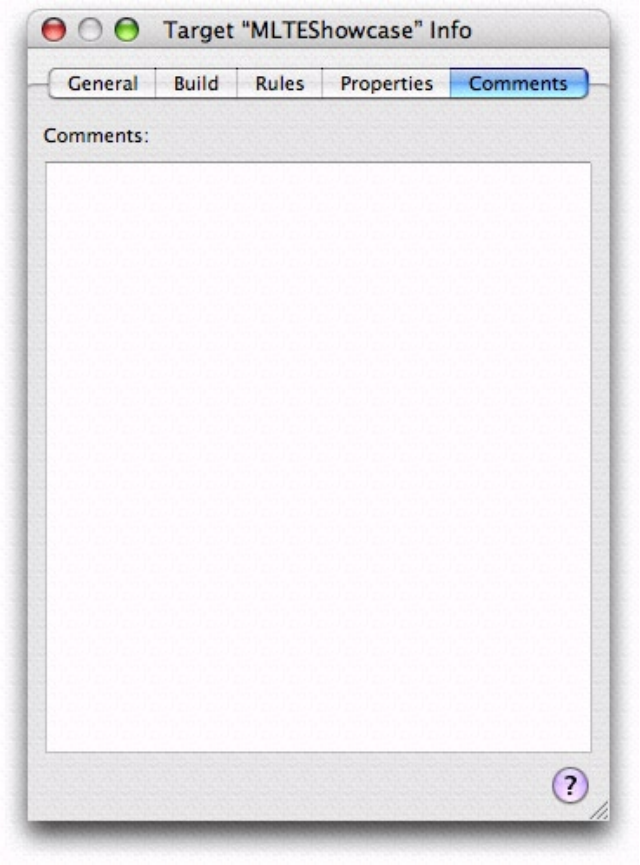
To help you keep track of your project contents, you can write comments and associate them with any of the items in your project. Xcode remembers these comments across sessions. In this way, you can document your project and its components, keep design notes, or track improvements you still wish to make. This is especially useful if you are working with large or complex projects that have many different pieces, or if you are working with a team of developers and have multiple people modifying the project.

For example, imagine a large project containing targets that build a suite of applications, a framework used by each of those applications, a command line tool for debugging, and a handful of plug-ins for the applications. With such a large number of targets it might be hard to keep track of exactly what each of the products created by those targets does. To make it easier to remember what each of them does, you could add a short description of the product that it creates to the Comments field for that target. This also aids others who may be working on the project with you; if they can read the comments, they can quickly get up to speed with the project and easily learn about changes made by other members of the development team.

To add comments to a project item:

- Select that item in the Groups & Files list or in the detail view and bring up an Info or inspector window.
- Click Comments to open the Comments pane, shown here.

Figure 6-4 The Comments pane



This pane contains a single text field into which you can type any information you wish. To add comments, click in the Comments field and type. You can also link to additional resources from the Comments pane; hypertext links, email addresses and other URLs are clickable in this field.

You can add comments to any project item other than smart groups, including projects, targets, files, and executables. You can view comments you have added to project items in the detail view and you can search the content of those comments using the Search field. If the Comments column is not already visible, choose View > Detail View Columns > Comments or Control-click anywhere in the column heading and choose Comments from the contextual menu.

Inspecting Project Attributes

Xcode tracks certain settings at the project-level. These settings include your choice of version control system, the version of Mac OS X to develop for, and the build styles available in the project. You can view and modify project-level settings in the project inspector.

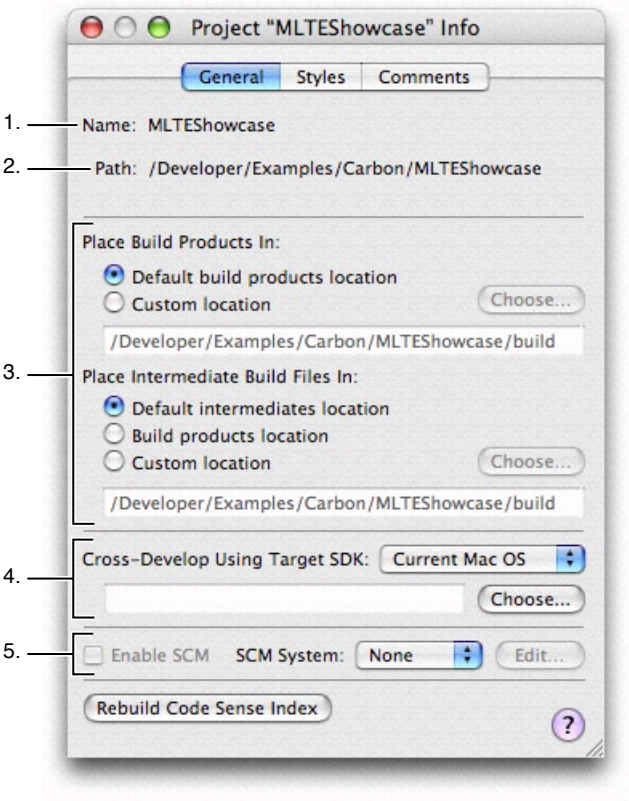
To open the project inspector, you can either:

- Select the project in the Groups & Files list and click the Info or Inspector buttons, or choose the Get Info or Show Inspector items from the File menu, as described in "Inspector and Info Windows" (page 73).
- Choose Project > Edit Project Settings.

The project inspector contains the following panes:

- General. This pane, described below, contains options that let you control various project-level settings, such as the Source Control Management (SCM) system used by the project or the minimum version of Mac OS X the project is built to run on.
- Styles. This pane contains all of the build styles defined for your project. A build style is a collection of build settings that are applied to one or more targets when you build; this allows you to vary the way in which a target is built. In the Styles pane, you can add, edit, and delete build styles. Build styles are described further in "Build Styles" (page 297).
- Comments. This pane lets you associate notes and other text with the project. The Comments pane is described further in "Adding Comments to Project Items" (page 92).

Figure 7-1 The project inspector



The General pane of the project inspector, shown here, contains the following information:

1. The name of the project, set when you first create the project using Xcode's project templates.
2. The location of the project folder in the filesystem.
3. The location at which the build products and intermediate files for the project's targets are placed. The options under the heading "Place Build Products In" specify the location where Xcode places the products created when building the project's targets. The options listed under "Place Intermediate Build Files In" specify where files generated in the course of building the product, but not included in the final product, are placed. See "Build Locations" (page 301) for more information.
4. The cross-development options let you choose the minimum version of Mac OS X to build your product for. This lets you target versions of the operating system other than the one you are currently developing on. Use the Cross-Develop Using Target SDK pop-up menu to specify which SDK to use. See "Using Cross-Development in Xcode" (page 333) for more information.
5. The source control management (SCM) system to use with the project. You specify an SCM system at the project level. In the General pane of the inspector, you can turn SCM on and off, as well as choose the particular SCM system to use with the project. See "Configuring Repository Access" (page 206) for more information.
6. The Rebuild Code Sense Index button lets you rebuild the symbolic index that Code Sense, described in "Code Sense" (page 104), uses to provides features such as code completion and symbol definition searches.

Finding Information in a Project

Being able to find information—both knowing how to leverage the user interface to locate items in a project, as well as knowing how to find information about your project—is critical to working effectively in Xcode.

The Xcode user interface gives you many different ways to location project information and items. "The Project Window" (page 55) describes the common paradigms of the Xcode user interface that let you find and manage project contents, including the Groups & Files list, which lets you organize and access the items in your project in an outline view, and the detail view, which lets you quickly filter your project contents. In addition, the Activity Viewer window lets you see additional information on Xcode operations, while the Info and inspector windows let you examine and modify items in your project.

Xcode also maintains a great deal of information about your project's contents, which it uses to assist you in the development process. The Xcode feature called Code Sense maintains an index that contains symbolic information for your project; Xcode uses this information as the basis for a number of features that let you browse the symbols in your project, view the class hierarchy of projects that use an object-oriented programming language, and search your project for symbol definitions.

As you are working on your software product, you often need to find information on system technologies. Xcode also includes a full-featured documentation viewer and installs a comprehensive suite of technical documentation. Using Xcode's documentation lookup features, you can quickly and easily search all or part of Apple's technical documentation for questions, keywords, or symbols.

This chapter describes how to use Xcode to find information about your project's contents and about Apple technologies. It covers:

- Searching your project for text, symbols, or using regular expressions.
- Using the Project Symbols smart group to find information on the symbols in your project.
- Viewing classes and class members for projects that include object-oriented code, using the class browser.
- Using the documentation window to browse Apple's technical documentation and search the Reference Library for keywords or symbols.

Searching in a Project

Xcode provides a number of ways to search for information in your project. You can search for text, regular expressions, or symbol definitions in a single file or across multiple files in your project. You can also easily substitute replacement text for one or more instances of matching text or symbols, either within a file or throughout the entire project.

This section describes how to use Xcode's project-wide search features to search through multiple files in your project and its included frameworks for text, regular expressions, and symbol definitions. This section also describes how to view search results. The single-file find is discussed further in "Searching in a Single File" (page 176). In addition, shortcuts for performing searches from an Xcode editor window are described in "Shortcuts for Finding Text and Symbol Definitions From an Editor Window" (page 178).

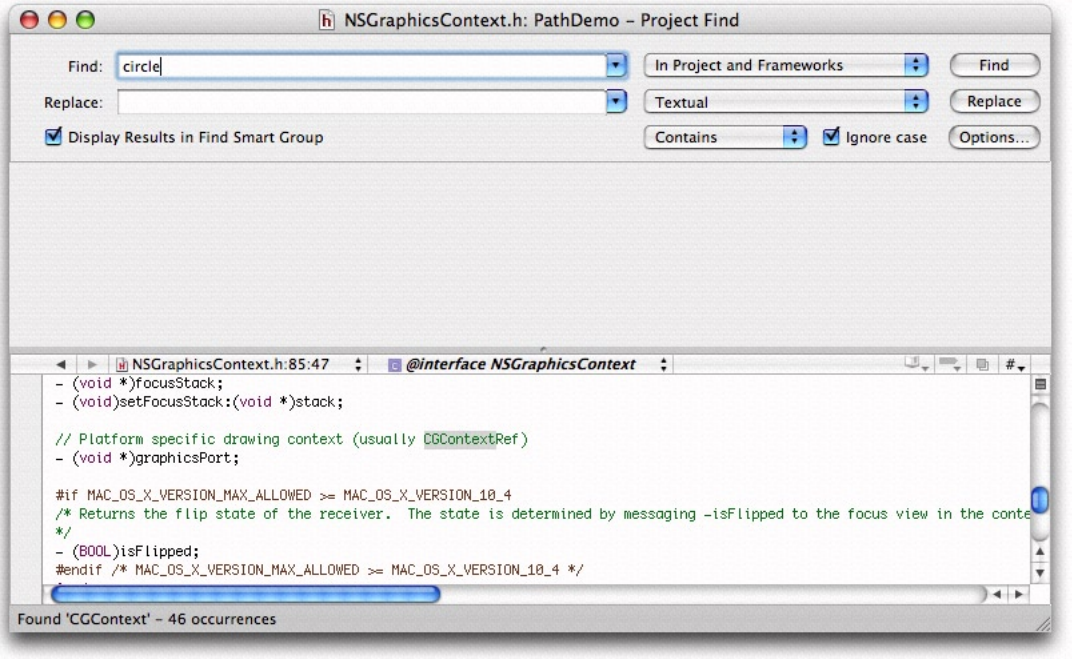
For more information on Code Sense, the technology that provides symbol definition searches, see "Code Sense" (page 104).

The Project Find Window

The Project Find window allows you to search for information in some or all of the files included in your project. Using the Project Find window, you can search your project for text, symbol definitions, or regular expressions. To open the Project Find window, choose Find > Find In Project. A window similar to the following appears.

Note: In the All-In-One project window layout, choosing Find > Find In Project opens the Project Find pane in the project page. The Project Find pane contains the same information as the Project Find window shown here.

Figure 8-1 The find window



Choosing What to Search For

Using the fields and menus at the top of the Project Find window, you can control what Xcode searches for.

1. The Find field specifies what to find. Xcode interprets this field differently, depending on the value of the pop-up menu to the right of the Replace field.
2. The pop-up menu to the right of the Replace field specifies the type of search; it contains the following options:
   - Textual finds any text that matches the text in the Find field.

- Regular Expression finds any text that matches the regular expression in the Find field.
- Definitions finds any symbol definition matching the symbol name in the Find field.

3. The pop-up menu to the right of the Display Results in Find Smart Group option controls how Xcode determines a match to the contents of the Find field. The available options are:
   - Contains. Choose this option to find text or symbol definitions that contain what is in the Find field.
   - Starts with. Choose this option to find text or symbol definitions that begin with the contents of the Find field.
   - Whole words. Choose this option to find text or symbol definitions that contain only what is in the Find field.
   - Ends with. Choose this option to find text or symbol definitions that end with the contents of the Find field.
4. The "Ignore case" option specifies whether or not the search is case sensitive.

As a shortcut, you can also perform a quick search of selected text or regular expressions in an editor window, as described in "Shortcuts for Finding Text and Symbol Definitions From an Editor Window" (page 178).

Specifying Which Files to Search

To control the scope of a search, use the pop-up menu to the right of the Find field in the Project Find window. This menu contains sets of search options that specify which projects and frameworks to search in. Xcode provides the following default sets of search options:

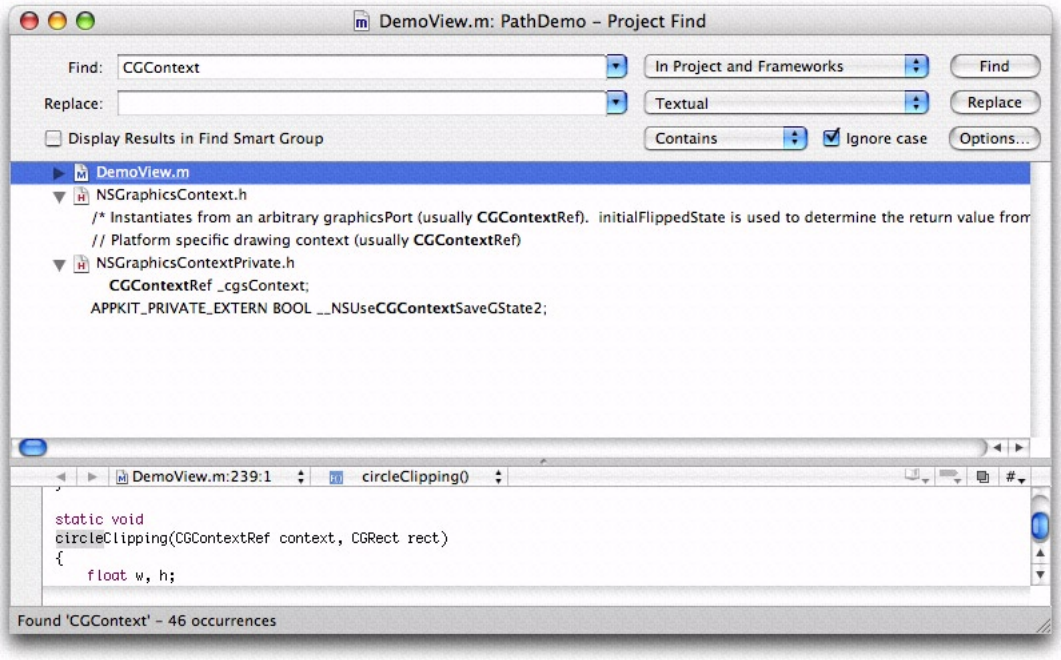
- In Project. Choose this option to search the files that are directly included in your project.
- In Project and Frameworks. Choose this option to search both the files and the frameworks included in your project.
- In Frameworks. Choose this option to search files that are in the frameworks included by your project.
- In All Open Files. Choose this option to search all open files.
- In Selected Project Items. Choose this option to search only in the currently selected project items.

You can further tailor these default sets of search options or define your own sets with the Batch Find Options window, described in "Creating Sets of Search Options" (page 101).

Viewing Search Results

When you perform a project-wide find, the results of the search are collected in the find results pane of the Project Find window; results are organized according to the file in which they appear, as shown in the figure below. You can view a particular search result by selecting it in the Project Find window; Xcode opens the file to the matching text and displays it in the attached editor. Double-click a search result to open it in a separate editor window.

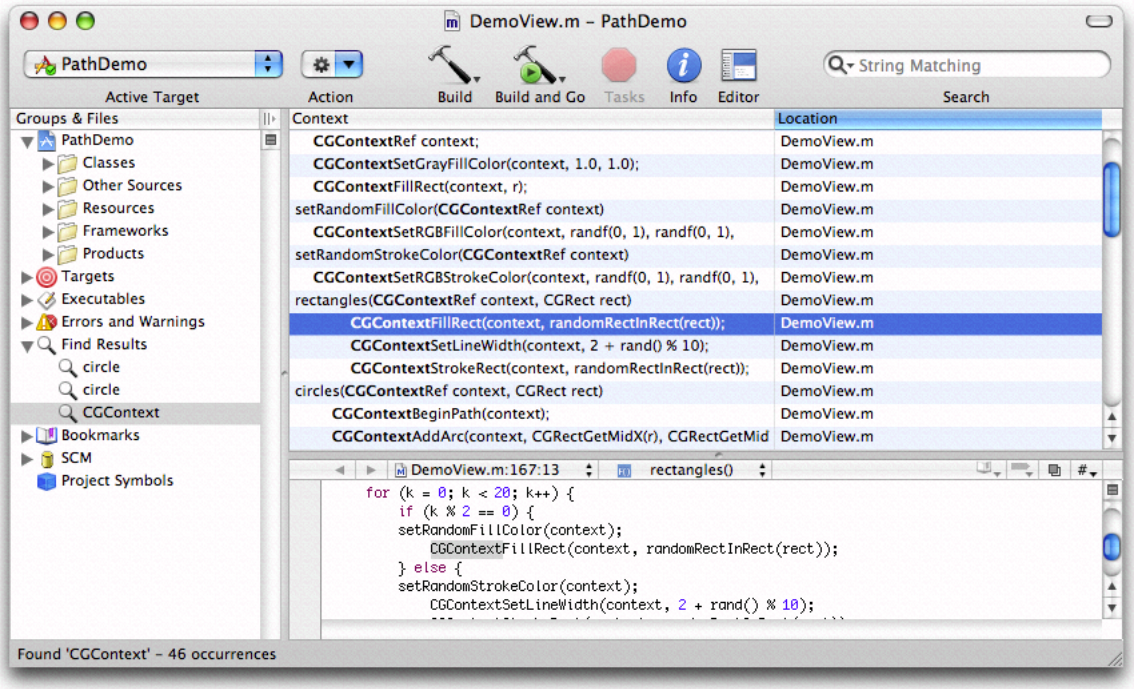
Figure 8-2 Find Results in the project find window



Each search, and its results, is also collected in the Find Results smart group that appears in the Groups & Files list of the project window. If you select the Display Results in Find Smart Group option, Xcode automatically brings the project window to the front and discloses the contents of the Find Results smart group when you perform a search, instead of showing the results in the Project Find window.

You can see all the searches you have performed by clicking the disclosure triangle next to the Find Results smart group in the Groups & Files list. To view the results of a given search, select that search in the Groups & Files list and, if necessary, open a detail view. The detail view shows all of the results for the selected search, as shown below. You can view the combined results of several searches by selecting those searches in the Groups & Files list. Double-clicking an item in the Find Results group in the Groups & Files list opens a Project Find window with the corresponding search specification as well as a detailed find results list. This allows you to rerun previous searches.

Figure 8-3 Search results in the project window



The detail view shows the context for each search result and the file in which the match occurs. The context of a search result is the surrounding text in which it appears for text searches, and the type and name of the matching symbol for a symbol definition search. To show or hide either of these two columns, use the View > Detail View Columns menu items or Control-click in any column header and choose the desired column from the contextual menu.

To view the source for a particular result, select the result in the detail view. If the project window or detail view has an attached editor, selecting a search result displays the source in the editor. You can double-click a search result to open the source for the result in a separate window.

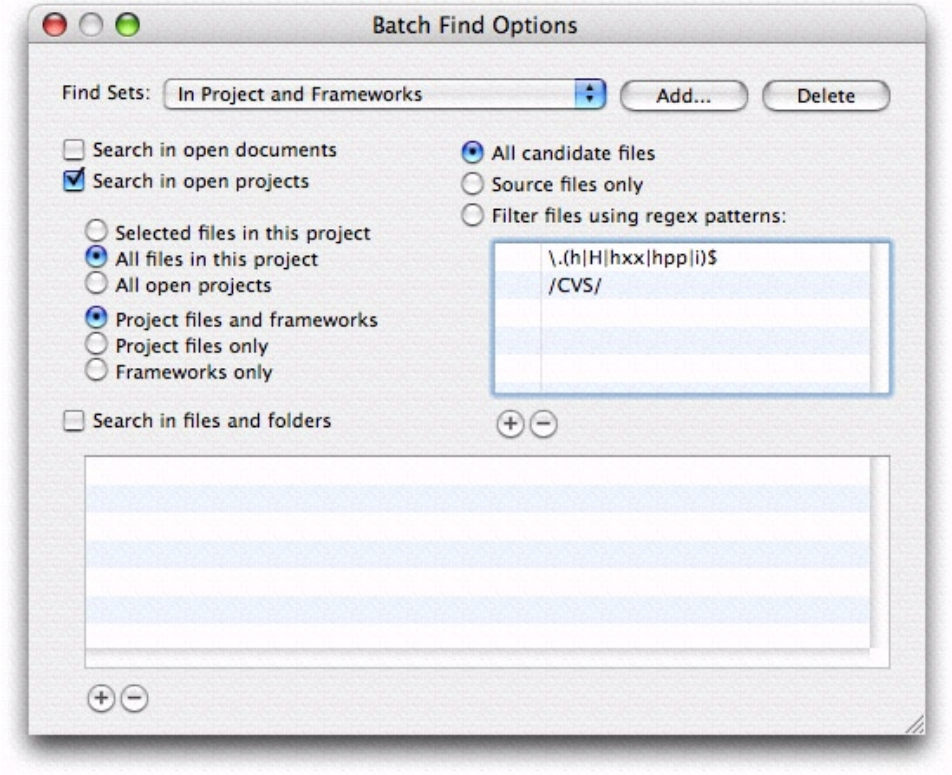
You can sort the results of a search according to the file in which they occur. Click the Location column heading to sort the detail view by location. You can also filter the search results using the Search field in the project window toolbar. Using the location of the search result as an example again, you can type all or part of a filename to see only those results that occur in the file of that name.

Creating Sets of Search Options

The Batch Find Options window lets you modify Xcode's default sets of search options, or define your own sets. This is particularly useful if you find yourself searching the same set of files over and over again. Instead of configuring the set of files to search each time, simply configure it once and save it as a search option set. To reuse the search option set, simply select it from the pop-up menu next to the Find button in the Project Find window.

To open the Batch Find Options window, click the Options button in the Project Find window. You should see a window similar to the following:

Figure 8-4 The Batch Find Options window



The Find Sets menu at the top of the Batch Find Options windows lists the available search option sets. To create a new set, click Add. Specify a name for the new set in the dialog that appears. Xcode creates a new set of search options with default values. To delete a set of search options, choose that set from the Find Sets menu and click Delete.

To edit a search option set, choose that set from the Find Sets menu and set the search options you wish to include. The Batch Find Options window provides the following options to control which files Xcode searches:

1. "Search in open documents" includes all files that are open in an editor in the search.
2. "Search in open projects" includes open projects in the search. You can further control which files in those projects are searched using the radio buttons below the "Search in open projects" option.

   The top set of radio buttons controls which projects are searched:

   - "Selected files in this project" searches the selected files in the current project.
   - "All files in this project" searches all files in the current project.
   - "All open projects" searches all files in all open projects.

   The bottom set of radio buttons lets you specify whether to include project or framework files in the search:

   - "Project files and frameworks" searches both the project's files and the files of any frameworks included in the project.

- "Project files only" confines the search to project files.
- "Frameworks only" confines the search to the frameworks included in the project.

3. "Search in files and folders" allows you to add specific files or directories to the search. Any files or directories listed in the table below this option are included in the search. To add an entry to this table, click the plus (+) button and choose a file or directory from the dialog that appears; or drag the file or directory from the Finder. You can also click in the table and press Return. To delete a file or directory from this table, select the entry and click the minus (-) button.

You can further restrict the files that are searched by a search option set using the radio buttons on the right side of the Batch Find Options window, next to the "Search in open documents" and "Search in open projects" options. You have the following options:

- "All candidate files" does not limit the searched files further. This searches all of the files in the search scope specified by the other options in the Batch Find Options window.
- "Source files only" limits the search to files containing source code.
- "Filter files using regex patterns" lets you filter the files to search using one or more regular expressions. These regular expressions are specified in the table beneath this radio button. To add an expression to this list, click the plus (+) button.

  Use the first column in this table to specify whether Xcode returns only those files that match a given regular expression or returns only files that do not match the regular expression. If this column is blank, Xcode does not use the regular expression. This column is blank by default for all regular expressions that you add; click in the column next to the regular expression to choose a different value.

Replacing Text in Multiple Files

You can use the Project Find window to replace some or all occurrences of the search string specified in the Find field. To replace text in multiple files:

1. Type the substitution text in the Replace field
2. Select one or more entries to replace. To choose which occurrences of the given search string to replace, do either of the following:
   - Select one or more entries to replace in the find results pane of the Project Find window.
   - Choose a search from the Find Results smart group in the project window. In the detail view, select one or more occurrences of the search string to replace.
3. Click the Replace button in the Project Find window.

If the contents of the Project Find window's find results pane are disclosed, Xcode uses the selection in the Project Find window to determine which occurrences to replace. If the find results group is closed, Xcode uses the selection in the detail view of the project window.

Viewing Project Symbols and Classes

Xcode includes a technology, called Code Sense, that maintains detailed information about the symbols in and utilized by your project to assist you in the development process. Code Sense uses the information in this symbol index, allowing you to browse the symbols and classes in your project, and perform symbol definition searches. It also uses this information to provide completion suggestions when editing source code, as described in "Code Completion" (page 189).

This section introduces Code Sense and discusses two of the features that it enables: the Project Symbols smart group and the class browser. In this chapter you will learn how you can use the Project Symbols smart group to view symbols and how you can take advantage of the class browser to find information on the classes defined in your project and its included frameworks. The other features provided by Code Sense are described in other parts of this document.

Code Sense

Code Sense makes it simple to find and view information about your code and to gain easy access to project symbols. Code Sense maintains detailed information about the code in your project and in libraries used by your project; this information is stored in a project index. Using this project index, Code Sense provides support for features such as the following:

- Code completion. Code completion, described in "Code Completion" (page 189), assists you in writing code by providing API suggestions from within the editor. Code completion uses the stored information about the symbols, as well as the contextual information from your source code, to offer a list of classes, methods, functions, or other appropriate symbols for the code you are working on.
- Class browser. Using the class browser, you can view the classes in your project and its included frameworks. From the class browser, you can easily access declarations, source code, and documentation for these classes and their members. The class browser is discussed in the section "Viewing Your Class Hierarchy" (page 107).
- Project Symbol smart group. The Project Symbol smart group allows you to view all of the symbols in your project directly in the project window. You can search these symbols or sort them according to type or name to quickly find the symbols you need. The Project Symbols smart group also gives you easy access to a symbol's definition. The Project Symbols smart group is described in "Viewing the Symbols in Your Project" (page 105).
- Searching. Xcode offers numerous ways to search for symbols and other information in your project. For instance, you can perform a project-wide search for symbol definitions, or you can select an identifier in an editor window and jump to its definition. These and other search features are described in "Searching in a Project" (page 97).

The project index used by Code Sense is created the first time you open a project. Thereafter, the index is updated in the background as you make changes to your project. Indexing occurs on a background thread, to keep it from interfering with other operations in Xcode. The index can be completely rebuilt, if necessary, by opening the General pane in the project inspector and clicking the Rebuild Code Sense Index button.

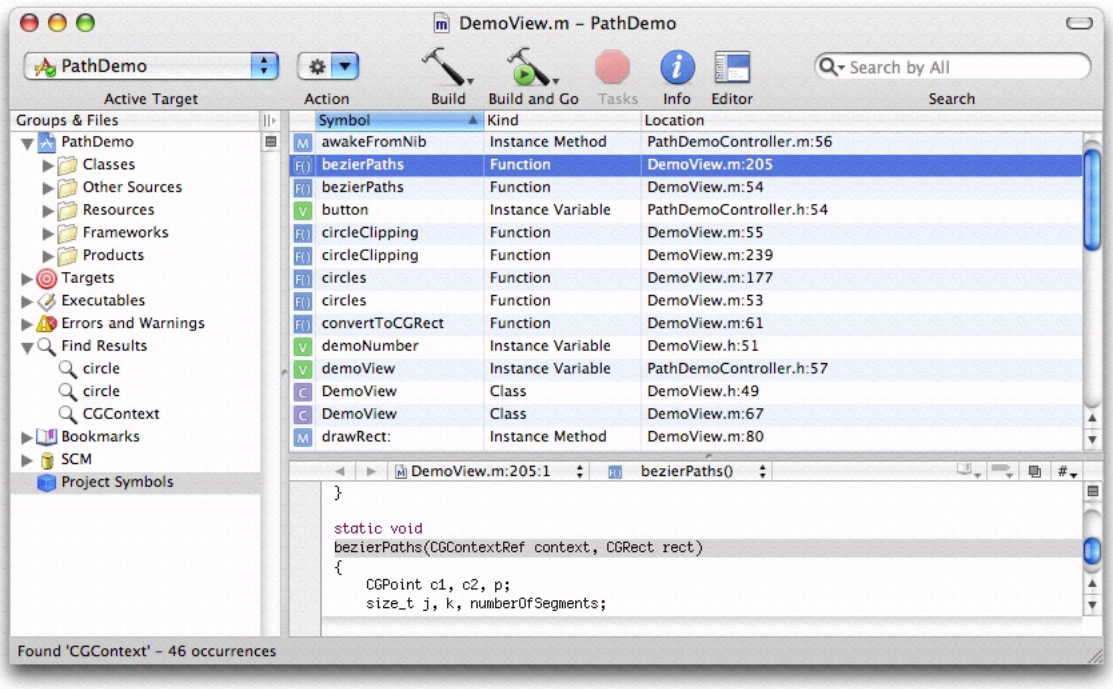
Indexing is enabled by default. You can disable indexing for all projects that you open. To do so, choose Xcode > Preferences, click Code Sense, and deselect the "Enable for all projects" checkbox in the Indexing section.

Note that if you turn off indexing, you will be unable to use those features that rely on the project index, such as code completion, the class browser, and the other features mentioned in this section. You can specify whether Xcode includes a particular file in the project index using the "Include in index" checkbox in the file inspector, described in "Inspecting File, Folder, and Framework References" (page 161).

Viewing the Symbols in Your Project

The Project Symbols smart group, one of the built-in smart groups provided by Xcode, allows you to view all of the symbols defined in your project. You can sort symbols by type, name, file, and file path, and you can search for symbols that match a string. To see the symbols defined in your project, select the Project Symbols smart group in the Groups & Files list. The detail view displays the symbols in your project. Here is what you see when you select the Project Symbols group for a project:

Figure 8-5 Viewing symbols in your project



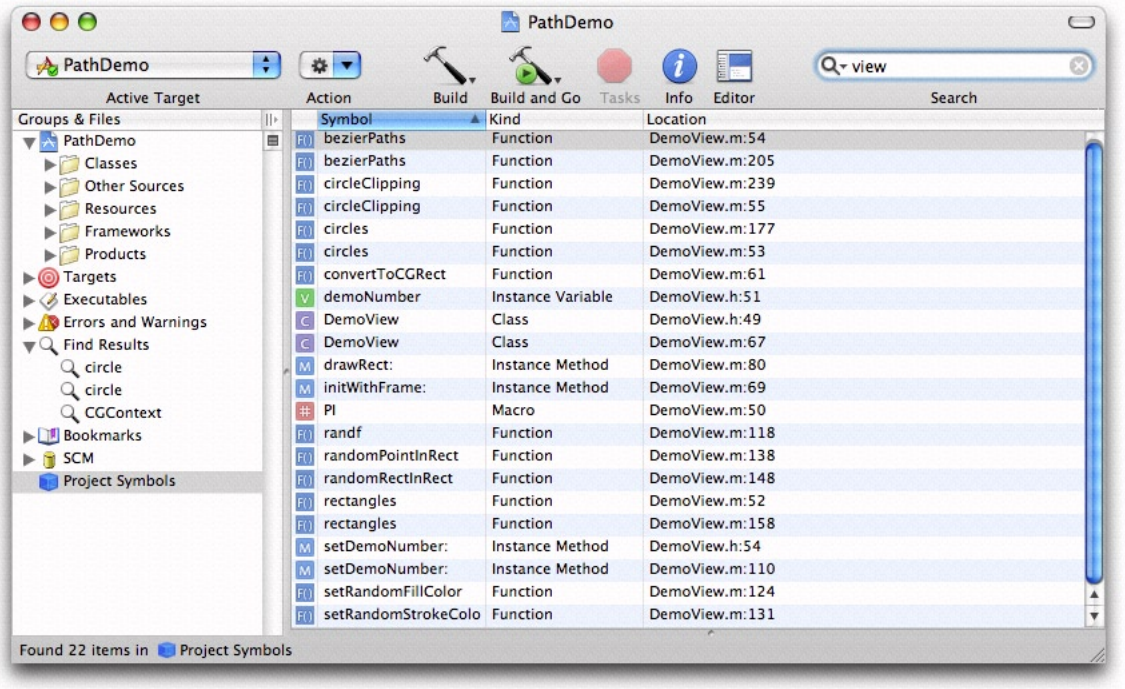
The detail view shows the following information for each symbol:

- An icon indicating the symbol type, such as structure, function, method, and so on. The icon is a visual cue that allows you to glance at a group of symbols and easily discern the type. The letter in the icon reflects the symbol type; for example, "M" for method, or "V" for variable. The color of the icon indicates the relative grouping of the symbols. For instance, purple icons indicate top-level elements such as Classes or Categories while orange icons represent basic types like structures, unions, typedefs and others.
- The symbol name.
- The symbol type. While the symbol type icon provides a handy visual cue as to the symbol type, the Kind column states the type explicitly.

- The file containing the symbol definition or declaration, and the line number at which the symbol appears.

To sort the symbols listed in the project window according to any of these categories, simply click the appropriate category heading. In addition, you can use the Search field in the project window toolbar to narrow the list of symbols to those matching a string or keyword. You can search the contents of any one of the categories—symbol name, symbol kind, or location—or you can search them all. In the PathDemo project, for example, you can search for all symbols declared in files pertaining to views by selecting "Search By Location" from the pop-up menu in the search field and typing "view." The symbols listed in the detail view are narrowed to include only those symbols defined in files whose names contain the word "view," as shown in the following figure. By default, the search field searches the content of all categories in the detail view.

Figure 8-6 Filtering the symbols in a project



You can configure which information is displayed in the detail view, as described in "The Detail View" (page 60).

To view the symbol definition, select the symbol in the detail view. If you have an editor open in the project window or detail view, the symbol definition appears there. If you prefer a separate editor window, double-click the symbol to open the file containing the symbol definition in a separate editor.

If you Control-click a symbol in the detail view, Xcode displays a menu that contains a number of useful commands. The commands available to you are:

- Reveal in Class Browser. If the selected symbol can be displayed in a class browser—for example, a class or a method—choosing this menu item opens the class browser window and selects the symbol in the browser. You can also reveal the selected symbol by choosing View > Reveal in Class Browser.

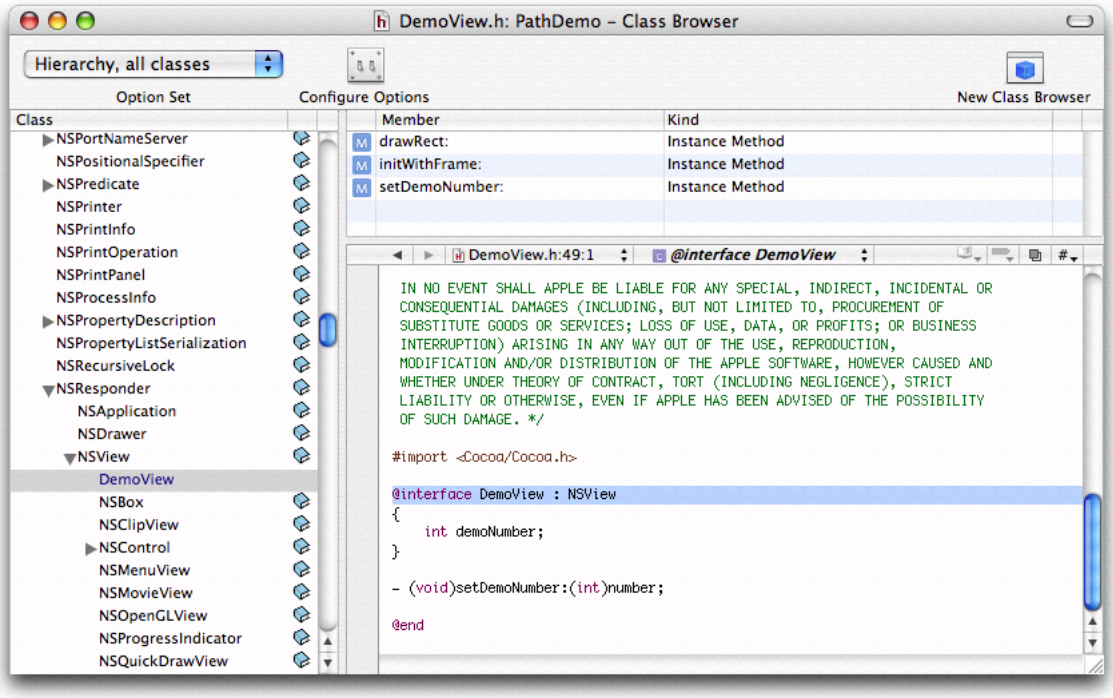
- Find Symbol Name in Project. When you choose this item, Xcode searches your project for the selected symbol name. It performs a textual search, just as if you had typed the symbol name into the Project Find window and selected "Textual" from the search type menu. The results of the search are displayed in the Find Results smart group.
- Copy Declaration for Method/Function. This command copies the declaration for the selected symbol to the pasteboard; you can then paste this declaration into any text field or editor using Command-V. This command only works for functions and methods.
- Copy Invocation for Method/Function. This command copies the invocation string for the selected symbol to the pasteboard, which you can then paste into any text field or editor. The invocation string is the same string that is inserted into your code when you select a code completion option. This command works only for functions and methods.

You can also reveal the file in which the selected symbol is defined in the Groups & Files list by choosing View > Reveal in Group Tree.

Viewing Your Class Hierarchy

If you are programming in an object-oriented language, you can view the class hierarchy of your project using the Xcode class browser. To open the class browser, choose Project > Class Browser. You can also open the class browser by selecting a symbol in the Project Symbols smart group and choosing View > Reveal in Class Browser. The following figure shows the class browser for a sample Cocoa application.

Figure 8-7 The Class Browser window



Classes and other top-level symbols—protocols, interfaces, and categories—are listed in the Class pane on the left side of the Class Browser window. When you select a class from this list, Xcode displays the members of that class in the table to the right of the Class pane. When you select a member name from this table, the declaration of that member item is displayed in the editor pane below. To see the item's definition, Option-click its name.

If the class browser does not list any classes, your project may not be indexed. To rebuild the index, select your project and open the inspector. Open the General pane and click Rebuild Code Sense Index.

A book icon beside a class or member's name indicates that documentation is available for that member. You can view this documentation by clicking the book icon.

The class browser uses fonts to distinguish between different types of classes and class members:

- Classes defined in your project's source code are in blue. Classes defined in a framework are in black.
- Members defined in the class are black. Inherited members are gray.

The Option Set pop-up menu and Configure Options button in the toolbar of the class browser window control which classes and class members are displayed in the browser. The New Class Browser button allows you to open additional class browser windows.

To see the file that a class or member in your project is declared in, select the class or member in the class browser and choose View > Reveal in Group Tree. This option is also available in the contextual menu for classes and class members. Xcode reveals the file in the Groups & Files list in the project window.

You can bookmark a class or class member by selecting it in the class browser and choosing Find > Add to Bookmarks or by choosing Add to Bookmarks from the contextual menu. Xcode creates a bookmark to the class or member's definition.

Choosing What the Class Browser Displays

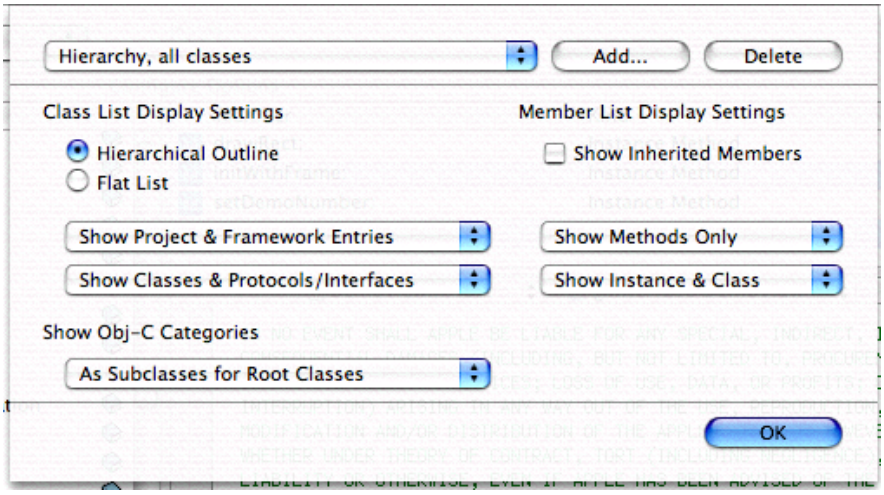
You choose which information Xcode displays in the class browser using the Option Set pop-up menu at the top of the browser window. This menu lets you switch between sets of display options. Xcode provides a few sets of predefined display options that allow you to choose:

- Whether to view all classes included in your project, or just those defined in your project
- Whether to view classes as a flat list or a hierarchical list

When you view classes as a hierarchical list, you can see the subclasses of a class by clicking the disclosure triangle next to that class.

To create your own set of display options or to make changes to an existing set, click the "Configure options" button. Xcode displays a dialog that lets you further refine which information is displayed in the class browser. The Class List Display Settings group, on the left half of the dialog, controls what is displayed in the Class list. The Member List Display Settings group, on the right, controls what is displayed in the members list.

Figure 8-8 The class browser options dialog



To choose how to display classes, use the radio buttons under Class List Display Settings:

- Choose Hierarchical Outline to view a hierarchical list of classes, where subclasses are listed under their superclasses.
- Choose Flat List to view an alphabetical list of classes, where all classes are at the same level and sorted alphabetically.

The pop-up menus under the Class List Display Settings options let you choose which items the class browser displays in the Class list. The first pop-up menu determines which classes the class browser shows:

- Show Project Entries Only. Shows only those classes defined in your project.
- Show Framework Entries Only. Shows only those classes defined in the frameworks that your project includes.
- Show Project & Framework Entries. Shows all of the classes defined in your project and in the frameworks that your project includes.

The second pop-up menu determines whether to show only classes, only protocols and interfaces, or classes, protocols and interfaces.

If you are programming in Objective-C, the Show Obj-C Categories menu lets you choose how Xcode displays Objective-C categories in the Class list:

- As Subclasses. The class browser lists categories under the classes that they extend.
- As Subclasses for Root Classes. The class browser lists categories under the root class of the classes that they extend.
- Always Merged into Class. The class browser lists all members of a class and any categories that extend that class together. The category does not appear as a separate entry in the Class list.

The Member List Display Settings let you control which items are included in the class members table in the class browser. You can choose the following:

- Whether to display members inherited from the class's superclass. To display inherited members, select the Show Inherited Members option.

  The inherited members that the class browser shows are limited by the scope of the selection in the first pop-up menu under Class List Display Settings. For example, if you have Show Project Entries Only selected, the class browser displays only inherited members from inherited classes in the project. To see all inherited members from all classes, choose Show Project & Framework Entries from the first pop-up menu in the Class List Display Settings group.
- Whether to display only methods, only data members, or both methods and data members, using the first pop-up menu in the section.
- Whether to display only instance members, only class members, or both instance and class members, using the second pop-up menu in the section.

Saving and Reusing Class Browser Options

You can save and reuse sets of class browser options. To reuse an existing set, choose it from the Options Set pop-up menu above the Class list. To create and delete sets, click the Configure Options button next to this pop-up menu.

- To create a new set of search options, click the Configure Options button, click Add, enter a name for your options set, and configure your options.
- To remove a set of search options, click the Configure Options button, choose the options set from the pop-up menu, and click Delete.

Viewing Documentation

Technical documentation is an important part of the software development process. As you develop a Mac OS X software product with Xcode, you're likely to use documentation to learn about the operating system and the technologies it supports, read about system frameworks, and look up individual API definitions.

Xcode includes its own documentation window, which you can use to view the technical documentation and other resources distributed as part of the ADC Reference Library. Xcode's documentation window gives you several ways to find documentation: you can browse documentation by title and category, search for symbol names, or perform a full-text search for a word or phrase. In addition, Xcode gives you access to the latest documentation available from Apple with downloadable documentation updates.

This section describes the documentation window and the ADC Reference Library content that is distributed with the Xcode Tools. It shows you how to use the API lookup and full-text search features to find Reference Library information, and how to obtain documentation updates.

Using the ADC Reference Library

Apple's ADC Reference Library is a complete collection of technical resources for developers, including documentation, sample code, release notes, technical notes, and technical Q&As. Xcode integrates this content into your development environment, letting you browse or search the ADC Reference Library from the documentation window.

Obtaining ADC Reference Library Content

The ADC Reference Library is available on Apple's developer website and as part of the Xcode Tools installation. You can automatically detect and download updates to the installed ADC Reference Library content through Xcode's documentation update mechanism, described in "Obtaining Documentation Updates" (page 118), or you can subscribe to the Developer Connection mailing to receive the entire ADC Reference Library on DVD each quarter. The Developer Connection mailing and Developer DVD Series are described further on the ADC website.

The Documentation package included with the Xcode Developer Tools installs a subset of the ADC Reference Library content on your local volume, at `/Developer/ADC Reference Library`. This content includes:

1. HTML versions of technical documentation, release notes, and API reference.
2. Category pages that let you browse documents by subject.
3. Full-text and symbol indexes that let you search all ADC Reference Library content.

This core documentation installation does not include PDF, sample code, technical notes, or technical Q&As. However, you can still access this content, referred to as "extended documentation," from Xcode. You can also find additional sample code, in the form of sample Xcode projects, at `/Developer/Examples` on your local volume.

Extended Documentation Locations

When you click a documentation link or search result in the Xcode documentation window, Xcode first looks for that content at `/Developer/ADC Reference Library`. If it does not find the requested content there, Xcode then looks at the extended documentation locations defined in Xcode's documentation preferences. If Xcode cannot find the requested content after checking the extended locations, it notifies you that it was unable to locate the specified documentation.

Xcode's default extended documentation location points to the full ADC Reference Library on Apple's developer website. You can see where Xcode looks for additional documentation in Xcode's preferences, in the Extended Locations table. You can also add your own locations. For example, if you have the full ADC Reference Library content on DVD, you can add a new location specifying that Xcode look for additional content on that DVD volume.

To add an alternative location for locating extended documentation:

1. Open Xcode Preferences by choosing Xcode > Preferences or typing Command-comma. Open the Documentation pane.
2. Under Extended Locations, click the '+' button.
3. Navigate to the volume or folder where the documentation is located and select the ADC Reference Library folder.
4. Click Add or press Return to add the new location to the Extended Locations table. Xcode adds the new documentation location to the list and makes it active. Click Apply or OK to apply your changes.

Xcode looks for content at the extended locations in the order in which they appear in the table, looking at the top entry first. You can change the order of a location by dragging its entry to the appropriate spot in the table. For example, if you want Xcode to look at the DVD copy of the ADC Reference Library before going to the web, add the DVD as an extended location and drag that entry to the top of the list.

If you don't want Xcode to look for documentation at a location that appears in the Extended Locations table, you can either remove the entry from the list or disable the location. To remove a location from the list, select the entry for that location and click the '-' button. To disable a location, deselect the checkbox in the On column next to the location. You cannot remove the default Web location, although you can disable it.

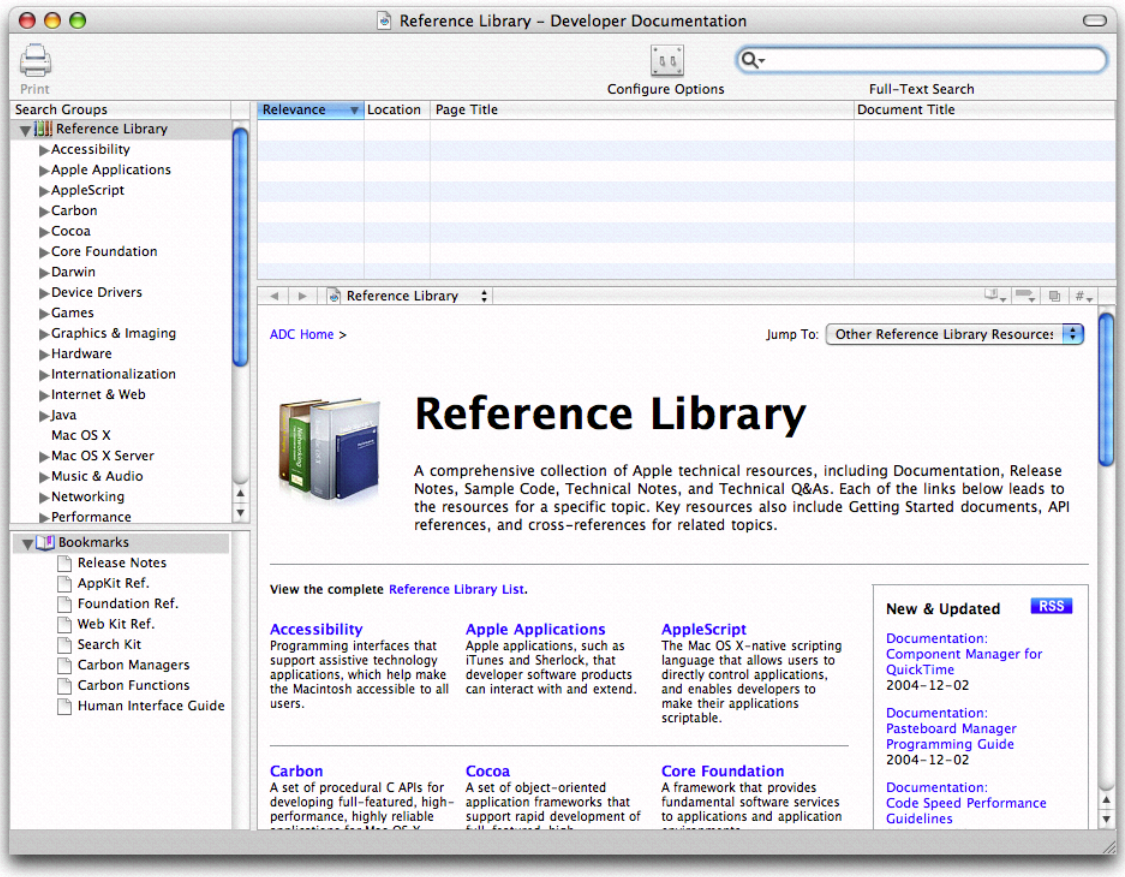
Browsing ADC Reference Library Content

Xcode has its own built-in documentation viewer, shown below, which allows you to find documentation quickly and easily. To open the documentation window you can:

- Select an item from the Help menu.
- Option–double-click a symbol name in an editor.
- Click a book icon next to a symbol name in the class browser.
- Select text in an editor window and choose Help > Find Selected Text in API Reference or Help > Find Selected Text in Documentation.
- Control-click the selected text in an editor window and choose Find Selected Text in API Reference or Find Selected Text in Documentation from the contextual menu.

Reference Library content is organized by category; you can see these categories in the Search Groups list in the documentation window, shown in Figure 8-9. Many of these categories have additional subcategories, which you can see by clicking the turn-down arrow next to the category name. When you select a category or subcategory in the Search Groups list, Xcode displays its contents in the documentation pane. Selecting the top-level Reference Library group, or a group with additional subcategories, displays a navigation page such as the one below that shows you the categories in this group with a brief description. From there you can further narrow the subject matter to look for specific documents.

Figure 8-9 The documentation window



When you perform a full-text or API reference search in Xcode, Xcode searches only in documents in the currently selected group and any groups it might contain. To search all of the ADC Reference Library content, select the Reference Library group in the Search Groups list.

If you prefer to have more than one document open and visible at a time, the documentation window lets you open a document in the documentation window, in a separate Xcode window, or in your preferred browser.

- To open the current document in a separate Xcode window, Control-click in the content area of the document and choose Open Page in New Window or Open Frame in New Window.
- To open the current document in your preferred browser, Control-click in the content area of the document and choose Open Page in Browser.

When following links that appear in the documentation, you can choose to open those links in the documentation window, in a separate editor window, or in your preferred browser.

- To open a link in the documentation window, click the link.
- To open a link in a separate Xcode window, you can either Control-click the link and choose Open Link in New Window from the contextual menu or simply Command-click the link.
- To open a link in your preferred browser, Control-click the link and choose Open Link in Browser or simply Option-click the link.

You can also copy a link or an image to the clipboard by Control-clicking the link or image and choosing Copy Link from the contextual menu.

Searching for Documentation

When you know the term you're looking for, searching is often the fastest way to find documentation. Xcode's documentation window supports two search modes:

- API reference search. This lets you quickly search the available reference documents for a symbol name.
- Full-text search. This lets you search the available documentation for a word or phrase. Xcode's full-text search supports simple search term matches, as well as more complex queries using Boolean operators and wildcard searches.

The current selection in the Search Groups list determines the scope of both API-reference and full-text searches. For full-text searches, the selection in the Search Groups list determines the documentation set that Xcode searches. For example, if you search for the term “button” with the Cocoa category selected in the Search Groups list, Xcode returns only documents from the Cocoa category that contain the word “button.”

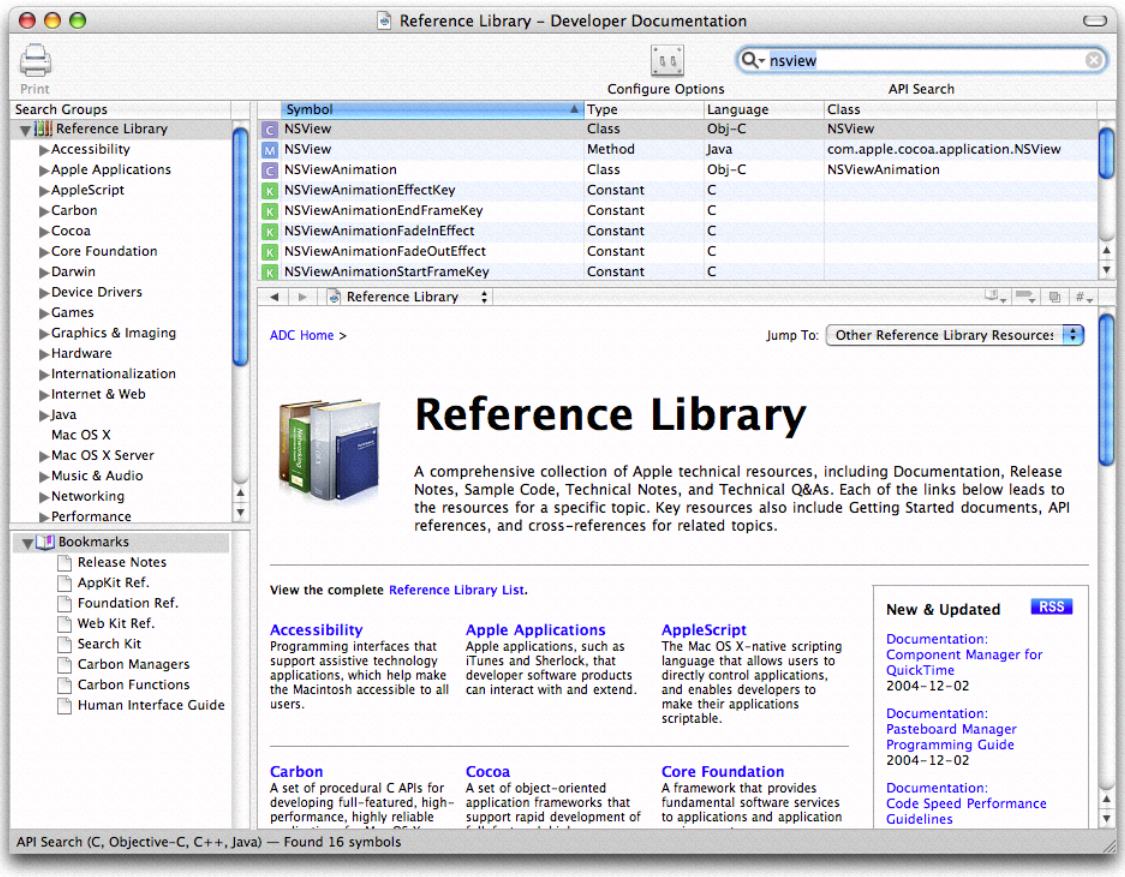
For API-Reference searches, the selection in the Search Group list determines the programming languages for which Xcode returns results. To search all available documentation, select Reference Library in the Search Groups list.

API Reference Search

When you are writing code, you often need to find documentation for a function, method, data type, or other symbol. Xcode's API reference search lets you quickly find the information you need.

To perform an API reference search in Xcode's documentation window, select API Search from the pull-down menu of the Search field and begin typing the name of the symbol. Xcode's API lookup supports type-ahead; as you type, the detail view displays all of the symbols whose names start with the string in the Search field, as shown in Figure 8-10. As a shortcut, you can also Option–double-click a symbol’s name in any Xcode editor window to open the documentation for that symbol in the documentation window.

Figure 8-10 API search in the documentation viewer



The current selection in the Search Group list determines the programming languages for which Xcode returns results. For example, if you select “Reference Library,” Xcode returns all matching symbols, regardless of the type or language. If you select Carbon in the Search Groups list and perform an API lookup, Xcode returns only matching C and C++ symbols.

You can also specify additional language filters for Xcode's API-reference search by clicking the Configure Options button in the documentation window toolbar. In the dialog that Xcode displays, choose which languages Xcode includes in API-reference searches. To include a language in the API search, select the checkbox in the On column next to that language. If you don't want Xcode to return results for a particular language, deselect the checkbox next to that language. These filters are applied to all API-reference searches regardless of the current selection in the Search Groups list. By default, all of the languages are enabled. Xcode supports API lookup for C, C++, Java, and Objective-C.

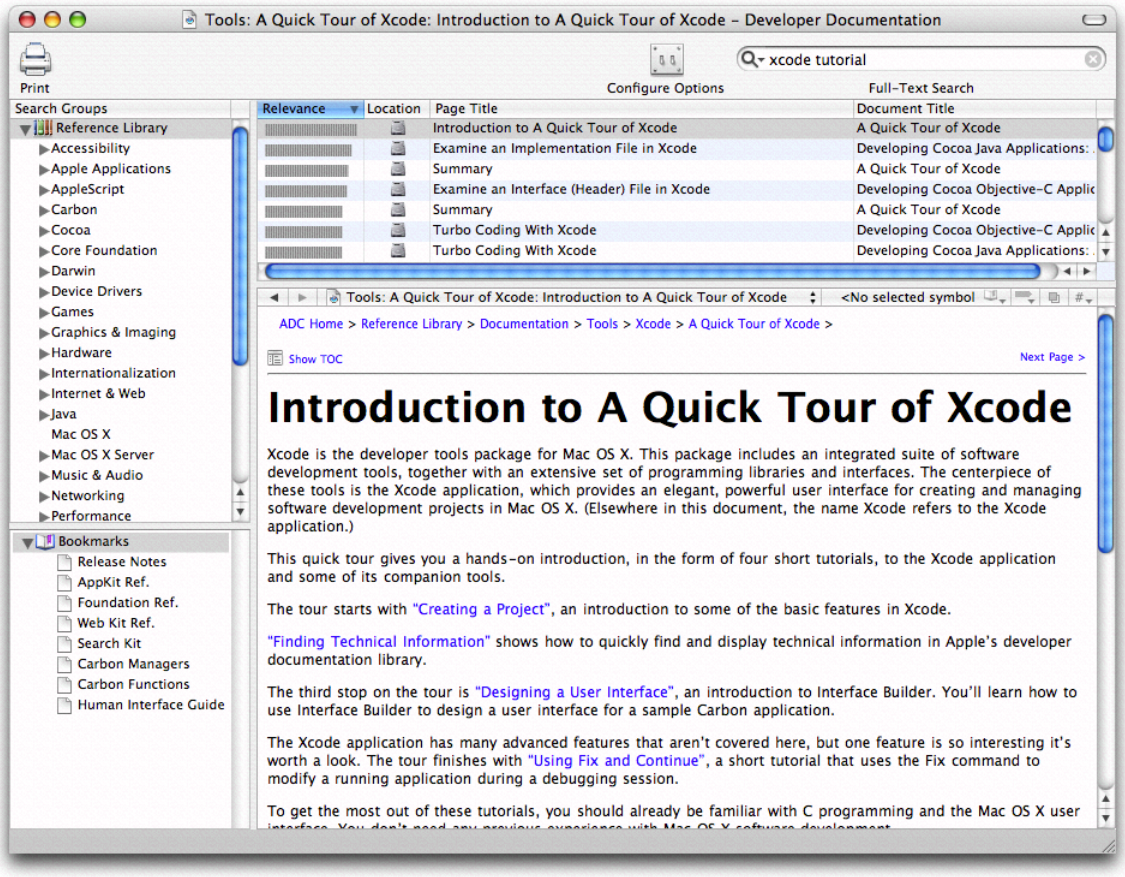
To view the documentation associated with one of the symbols returned in a search, select its name from the table. The documentation is displayed in the pane below the table view.

Full-Text Search

Xcode's full-text search lets you search the installed documentation for a word or phrase. You can enter a simple search term, such as “button,” or create more advanced queries using Boolean operators and wildcard characters. Xcode's full-text search uses Apple's Search Kit technology, described in *Search Kit Programming Guide*

To perform a full-text search choose Full-Text Search from the pull-down menu in the Search field of the documentation window. Type your query and press Return. Xcode searches the installed documentation for the given term or terms and displays the results in the documentation window, as shown in Figure 8-11. The table view displays the relevance, location, page title, and document title for each page returned by the search. To view a page returned as a search result, select that page in the search results table. The scope of the search is determined by the selection in the Search Groups list.

Figure 8-11 Results of a full-text search in the documentation window



By default, Xcode performs an exact-match search. That is, when you enter a search term, Xcode returns only those documents that contain the entire term. For example, searching for “button” returns documents containing the word “button,” but not documents containing only the word “buttons.” If you type a phrase, such as “bevel button,” Xcode returns those documents that contain both the word “bevel” and the word “button.”

Note: You can use a wildcard search, described in “Wildcard Search” (page 118), to search for variations on a term.

As a shortcut, you can also search the installed documentation for text in an editor window. Select the text to search for and choose Help > Find Selected Text in Documentation. Xcode performs a full-text search and returns all documents containing that text.

You can further refine your search by combining search terms using Boolean operators, searching for required terms, or using a wildcard search. The following sections describe the various queries you can construct for searching the developer documentation.

Constructing Queries Using Boolean Operators

If you wish to further restrict the parameters of your search, you can combine search terms using Boolean operators. You can create arbitrarily complex queries to narrow your search to fit your particular criteria.

Note: The smallest unit at which search results are evaluated is a single HTML file; in Apple's developer documentation, this typically corresponds to a section in a chapter, a group of function descriptions, or a class. If your query is too restrictive, you may not get any results at all.

Xcode supports the following Boolean operators, listed in order of precedence from highest to lowest:

()	logical grouping
!	NOT
&	AND
|	OR

For example, to find documents about the font panel in Mac OS X, you want to find documents that contain the phrase “font panel.” Just to be safe, you also want to catch any documents that don't contain the exact phrase “font panel,” but do contain both search terms “font” and “panel.” You can do that with the following query:

```
"font panel" | (font & panel)
```

Required Terms Search

Simpler than a Boolean query, a required-terms search lets you search for terms that must or must not appear in documents returned as a search result. A required terms search uses the following operators:

+	Indicates a term that must appear in any document returned
-	Indicates a term that must NOT appear in any document returned

For example, entering `+window` returns all documents containing the word “window,” similar to the behavior you get by simply entering “window” as a search term. However, if you enter `+window -dialog`, you will get all documents containing the word “window,” but NOT the word “dialog.”

Using Boolean operators to construct the previous query, you would write:

```
window & (!dialog)
```

Wildcard Search

If you are not sure exactly how a particular term appears in the documentation, you can use a wildcard search to include all variations of a search term in the search results. For example, if you are looking for all documentation about buttons in Mac OS X, you probably really want to see all documentation containing either the word “button” or the word “buttons.” Rather than have to specify each of these as separate terms, you can simply use the wildcard character to construct the following query, which returns all documents containing the word “button” or any word with the prefix “button.”

```
button*
```

You can use the wildcard character anywhere within a search string. Using a wildcard character at a location other than at the end of a search term may result in longer search times.

Finding Documentation for Command-Line Tools

To find documentation on a command-line tool, choose Help > Open man page. Use the “man page name” option to display documentation on a command-line tool. You can optionally specify a man page section; for example, `access(5)`. Use the “search string” option to find commands that are related to a keyword.

Working With Documentation Bookmarks

The Xcode documentation window also supports bookmarks, to provide easy access to documentation that you use frequently. To view the available bookmarks in the documentation window, click the disclosure triangle next to the Bookmarks group in the Search Groups list of the documentation window. Xcode includes a default set of bookmarks, but you can delete any of these default bookmarks or create your own.

To add a bookmark, choose Find > Add to Bookmarks. This bookmarks the page currently open in the documentation window and adds the bookmark to the Bookmarks group. You can also add a bookmark by dragging the document proxy icon in the titlebar of the documentation window to the Bookmarks group in the Search Groups list.

To open a bookmarked location, select the bookmark in the Search Groups list. To rename a bookmark, Option-click the name of the bookmark in the Search Groups list and type the new name.

Obtaining Documentation Updates

In addition to the full ADC Reference Library content available with the Xcode Tools DVD and Developer DVD Series, Apple also provides downloadable packages of the documentation installed on your local computer at `/Developer/ADC Reference Library`. Xcode automatically detects these updates as they become available. Documentation updates are released more frequently than the full Xcode Tools package, so this is a quick and easy way to stay up-to-date with the latest technical information from Apple.

Checking for Updates

Xcode automatically checks for a documentation update the first time you launch Xcode after installing a new version of the Xcode tools. Any time you access the documentation after the initial check, Xcode checks for an update if the interval specified in Xcode's Documentation preferences has passed since the last check. By default, Xcode checks for updates on a weekly basis, but you can choose a different interval.

To change the interval at which Xcode checks for documentation updates:

1. Open the Documentation pane of Xcode Preferences.
2. Choose an interval from the "Check for documentation updates" pop-up menu that appears in the Updates settings group. You can choose to have Xcode check for updates on a daily, weekly, or monthly basis. Click Apply or OK to apply your changes.

If you don't want Xcode to automatically check for updates at all—for example, if you use Xcode on a computer that does not usually have an internet connection—deselect the checkbox next to the "Check for documentation updates" menu.

To check for a documentation update immediately, click the Check Now button. The "Last check" field shows the date and time at which Xcode last successfully checked for updated documentation.

Installing an Update

When it checks for a documentation update, Xcode compares the version of the documentation on Apple's server to the version installed on your computer. If the documentation on the server is newer, Xcode notifies you that updated documentation is available. To download the documentation package, click Download.

Xcode launches your default browser application to download the documentation package. When the download is complete, the disk image automatically mounts. Double-click the documentation package to launch the Installer. The installer updates the local installation of the ADC Reference Library at `/Developer/ADC Reference Library`.

If Xcode is running when the installation is successfully completed, Xcode notifies you that the documentation has been updated. Click OK to reload the Search Groups and ADC Reference Library entry page in the documentation window. Otherwise, Xcode loads the new documentation when it next launches.

Controlling the Appearance of the Documentation Viewer

You can change the minimum font size used for viewing documentation. To do so, open the Documentation pane of the Xcode Preferences window. To enforce a minimum font size for documents displayed in the documentation window:

1. Under Universal Access, select the "Never use font sizes smaller than" option.
2. Choose a font size from the corresponding pop-up menu.

You can temporarily change the font size used to display an HTML file—even if its font size is controlled by a CSS file—by choosing Format > Font > Bigger or Format > Font > Smaller.

Design Tools

Xcode includes two separate design tools, with similar forms but different functions. Together, the tools allow you to model both the classes in your application and entities that represent your data. The class modeling tool allows you to understand and explore the classes in your project, whether they're written in Objective-C, C++, Java, or a mixture of those languages. The data modeling tool is like the class modeling tool in that you can, for example, lay out a diagram to represent your model visually, but different in that it deals with entities and the relationships between them, not with classes and hierarchies. You use the data modeling tool to define a schema for Core Data. The following chapters describe Xcode's design tools.

Overview of Xcode Design Tools

Xcode includes two separate design tools, with similar forms but different functions. Together, the tools allow you to model both the classes in your application and entities that represent your data. Although they are in some respects similar, class modeling and entity-relationship modeling are fundamentally different and serve different purposes—both are discussed below.

The two tools share some common functionality. They allow you to create models that form part of your project. They allow you to browse through the contents of the model using a set of table views and to visualize the contents in a diagram.

Control over how models are displayed is where Xcode differs from other design tools. Other IDEs that provide a graphic class browser typically give you little control over display—you see what it wants to show you. With Xcode, you have coarse-, medium-, and fine-grained control over what is displayed and how. You can edit the diagram in the same way you might in common graphics editors. For example, you can color, move, and align elements to arrange them how you wish; zoom in and out of the diagram; and choose whatever page and grid sizes you want. Moreover, your models never become stale.

Class Modeling

Class modeling allows you to understand and explore the classes in your project, whether they're written in Objective-C, C++, Java, or a mixture of those languages. You can get a bird's eye view of your project structure, look at relationships among classes, or scan quickly through class member and method types, parameters, and return values. You can use class modeling to augment or replace the Xcode class browser. The class model is saved with your project (you can even commit it to your repository), so other team members can get an overview of your code structure from the class model.

You can use the tool to visualize and browse class hierarchies not only in terms of the class relationships (subclass and superclass), but also in terms of what protocols (or interfaces in Java) a class implements, and what categories are present. You can even add comments to call out notable things about specific classes.

Unlike most other modeling tools, you control the set of files (groups, targets, and so forth) that are modeled, the position and layout of the classes in the diagram, and even what classes are shown. Moreover the Xcode class information is always up to date. Class models always represent the actual classes in files, groups, and targets in your project, and are automatically updated as you change your source code—even if you add, remove, or refactor classes.

Data Modeling

The data modeling tool is like the class modeling tool in that you can, for example, lay out a diagram to represent your model visually, but different in that it deals with entities and the relationships between them, not with classes and hierarchies. There is not necessarily a 1:1 relationship between entities and classes—for example, the same class may be used to represent more than one entity. For more about data modeling, see [Modeling Document].

More importantly, however, you use the data modeling tool to define what is in effect a source file—a schema for Core Data. The data model ultimately becomes part of your build product and is used by your application at run time. Instances of entities are typically stored in a persistent store (typically a file).

Why Are Modeling Tools Useful?

There are a number of reasons why modeling tools are useful. Some reasons apply to both the class and data models, some apply to one or the other.

A graphical representation of your project gives you a better conceptual overview of your project than raw XML or a mass of source files . In particular, with Xcode, it gives you a developer's eye view of your project, not a computer-generated representation . You can customize the view to see the information you need, not what the computer thinks is important or requires you to see . This is especially useful for communication between members of a team, or for homing in on a specific aspect of a project [see filtering]. In addition, the class modeling tool may also be useful for learning about functionality provided by existing libraries.

In terms of navigation, the browser view gives you an alternate means of navigating through your source, following relationships where appropriate. It provides a compact representation and summary of the classes or entities in your project, including their properties and behaviors, and (for the class browser) an easy way to get to relevant documentation.

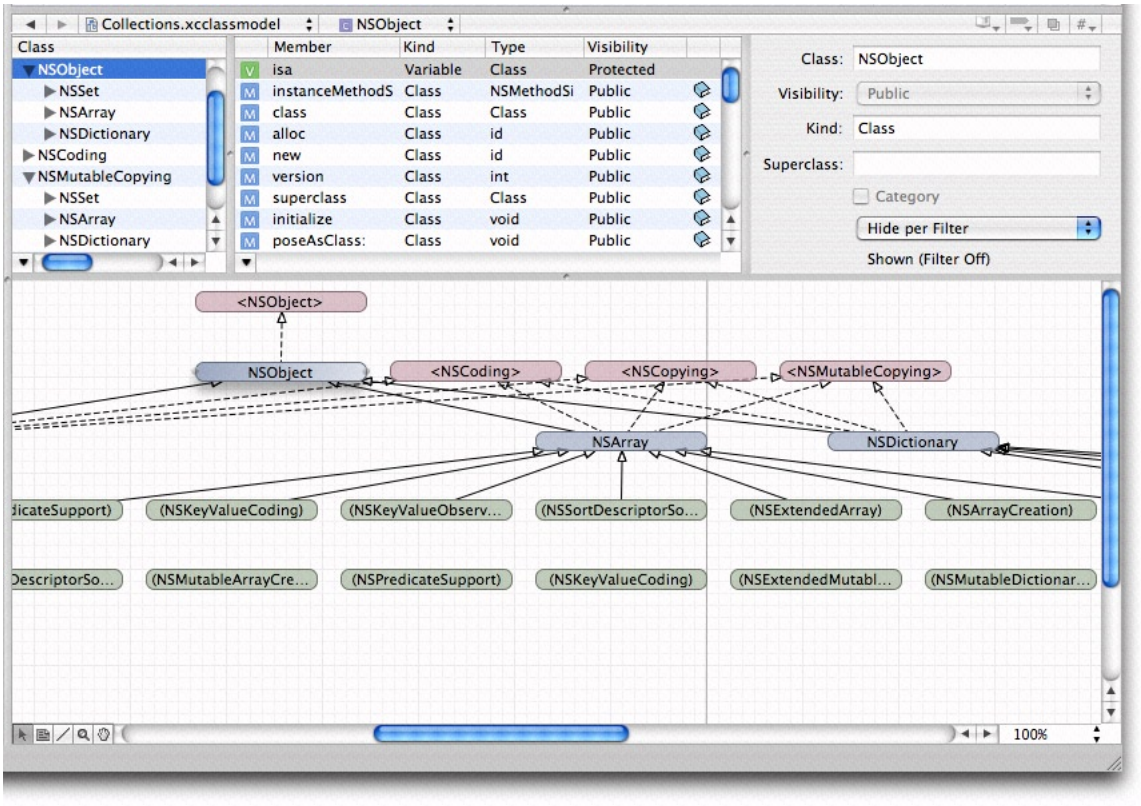
Both class and data models are project resources. A class model is stored with the project as it is an integral part of development process . A data model is stored as part of the project because it is essential to Core Data—it is also included in a target as it is compiled for deployment .

Common Features of the Xcode Design Tools

Although the class and data modeling tools are different, and have different purposes, they share common functionality and usage patterns. Both employ a browser that you can use to navigate the class or entity hierarchy, to view the properties of an individual class or entity or the properties of a collection of classes and entities, and to inspect attributes of a property. Both tools also have a diagram view that you can use to visualize their contents.

The diagram and browser views have different roles. The diagram view is typically best when you need a high-level overview of the project . The browser view gives you more detailed information , and it can be especially useful in data models when, for example, you want to edit several attributes simultaneously. When you have large collections of classes or entities, you can minimize the information shown in the diagram (for example, just class names and inheritance relationship lines) and get the detailed information from browser . The diagram view offers a variety of different configurations, so you can tailor your view to any need. Figure 10-1 shows the class browser and a class hierarchy diagram.

Figure 10-1 Browser view and diagram view for a class model



The Diagram View

The diagram view contains two important elements, rounded rectangles, which represent nodes, and lines. Class diagrams may also contain annotations (described in "Class Modeling With Xcode Design Tools" (page 137)).

Diagram Elements

The diagram view contains two important elements, "roundtangles" and lines. Class diagrams may also contain annotations (described in the "Class Modeling With Xcode Design Tools" (page 137)).

Nodes

Nodes are the base elements in the model. In the class model, they are classes, categories, and protocols (interfaces in Java); in the data model they are entities.

A node may be split into three sections: The title bar containing the name of the element (including the namespace or package for C++/Java), and two compartments. The compartments show attributes and relationships in data models and properties (instance variables) and operations (methods or member functions) in class models (see Figure 10-3 (page 127)).

Lines

The semantic meaning of a line is different in the class and data models, and it further depends on whether the line is solid or dashed, what sort of arrowhead is present, and what objects it connects. In class diagrams, you can only edit lines to or from annotations—they are created automatically based on the inheritance hierarchies defined by the code in your project.

Diagram Tools

The diagram view provides several tools, whose function should be familiar from other drawing packages. You select the tools from the palette in the bottom-left corner of the diagram view, shown in Figure 10-2.

Figure 10-2 Diagram tools

Arrow

You use the arrow tool to make selections and to move and resize graphic elements.

Text (Class Diagrams Only)

You use the text tool to annotate diagrams in a class model. To edit text in the text area, simply double click inside the element. You can use the Formatting menu, Font panel, Colors panel and so on to format the text as you wish. You can use the line tool to connect a text area to a specific class.

Line

You use the line tool to add a relationship in data model or to connect a comment to a specific class in a class model. To connect two elements, select the line tool, then drag from one end of the connection to the element at the other end. Note that in a data model, when you are establishing a relationship, you must make the connection from the source to the destination of the relationship.

Magnifying Glass

You use the magnifying tool to zoom into part of the diagram, or, by holding down the Option key, to zoom out . See "Zoom" (page 129) for other ways to zoom. To effect the zoom, you select the tool, then click inside the diagram.

Hand

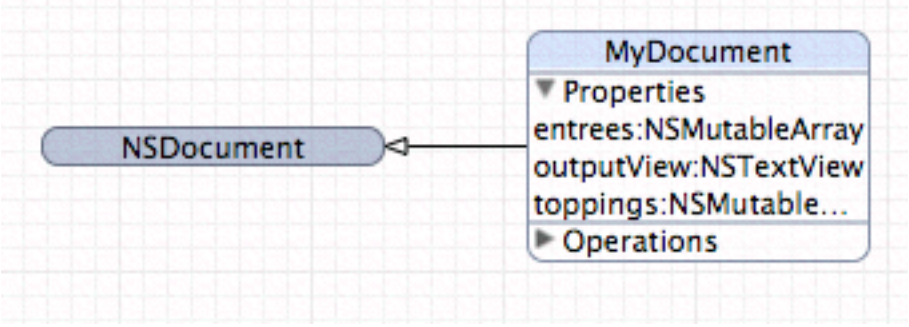
You use the hand tool to move the diagram if its bounds extend beyond the current view.

Roll-Up and Expansion

You can display a node and the compartments within it in a variety of ways:

- Rolled up, so that just the name of the class or entity is showing. This gives the most compact representation, with maximum information density in the diagram. (In Figure 10-3 the NSDocument node is rolled up.)
- Compartment titles showing. The titles are Attributes and Relationships in the data model, Properties and Operations in the class model. This gives a compact representation, but with easy access to detail .
- Compartments expanded. All the information in a compartment is visible but at the cost of screen real estate. (In Figure 10-3 the properties compartment of MyDocument is expanded, but the opertations compartment is not.)

Figure 10-3 A rolled up node and a partially expanded rolled down node



To roll up or roll down the node, use Design > Roll Up Compartments or Design > Roll Down Compartments respectively. To hide or expose compartment information, you use the disclosure triangle within a compartment, or Design > Expand Compartments or Design > Collapse Compartments .

Layout

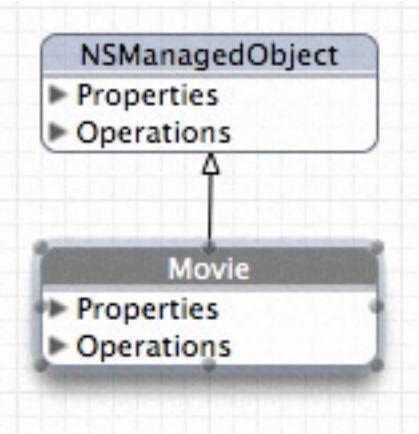
There are a number of options for moving and resizing elements; you can also constrain the way the elements can be moved and resized, and even prevent them from being moved and resized at all. Furthermore, you can zoom into and out of the diagram, and arrange the page layout as you wish.

Moving and Resizing Shapes

You can rearrange elements in a diagram to suit your needs—lines that join elements are updated appropriately. Use the arrow tool to select an element, and then simply drag it. You can move all the elements in the current selection (see "Multiple Selection" (page 129)) in the same way.

When you select a shape, "handles" appear around its edges (as shown in Figure 10-4). You can drag the handles to resize the shape as you wish.

Figure 10-4 Diagram view showing element handles



You can also automatically resize elements in several ways, using the Design > Diagram > Size menu: Make Same Width and Make Same Height resize the selected elements appropriately; Size to Fit resizes the selected elements so that they fully enclose their contents with minimal padding.

Alignment and Grid

You can use a variety of options to automatically align selected elements and to help you keep elements aligned . You can use the menu items in the Design > Diagram > Alignment menu to perform a number of operations, aligning specified edges or centers of a selection and aligning a selection in a row or column.

You can also use a grid to help keep elements aligned. By default, the diagram view displays a background grid, and move and resize operations are snapped to it. Using the Design > Diagram menu, you can turn the grid display on and off; you can also independently turn the snap-to-grid feature on and off.

Automatic Layout

The class modeler provides two ways to automatically lay out elements in a diagram, force-directed layout and hierarchical layout. You access them using the Design > Automatic Layout menu.

With hierarchical layout , parent elements are typically at the top of the diagram, children at the bottom. This gives a generally horizontal layout, and is generally better for deep hierarchies.

The force-directed layout tends to produce circular arrangements, with commonly referenced classes in the middle. This is usually better for shallow hierarchies. Note that the force-directed layout can take unbounded time to calculate—it is not recommended for very large collections.

You can apply the automatic layout feature just to selected items or to the whole diagram. The layout respects the current size of elements in the selection. If you expand or contract a node, then perform automatic layout again, the result is different from what it was prior to the change in size .

Locking

You can lock individual graphic elements in place using Design > Diagram > Lock, or the Lock contextual menu. If you subsequently apply automatic layout, locked elements are unaffected. To unlock an item, use Design > Diagram > Unlock, or the Unlock contextual menu.

Zoom

You can zoom into and out of the diagram in three different ways:

- Use the Design > Diagram menu to zoom in, out, and to fit.
- Use the pop-up menu to select a percentage zoom.
- Use the magnifying glass tool (click to zoom in, Option-click to zoom out).

Page Layout

If you move diagram elements outside the current diagram bounds (whether directly, or through applying automatic layout, or by unhiding elements), the page area automatically expands. Conversely, if you remove elements such that a page is left blank, the page area automatically contracts.

You can adjust the size of a page using the File > Page Setup menu. The page layout adjusts automatically to accommodate a change in page size.

Multiple Selection

You can use multiple selection in the diagram view to move a collection of elements in a flotilla drag, or for roll-up, expand all, and so on. You can make multiple selection in several ways:

- You can select a single element, then hold down the Shift key and click additional elements. Unselected elements are added to the current selection; selected elements are removed from the current selection.
- You can drag the background of the diagram to create a selection rectangle. Elements whose boundaries intersect with this rectangle are selected.

- You can select classes or entities in the browser—the browser selection and diagram selection are kept synchronized.

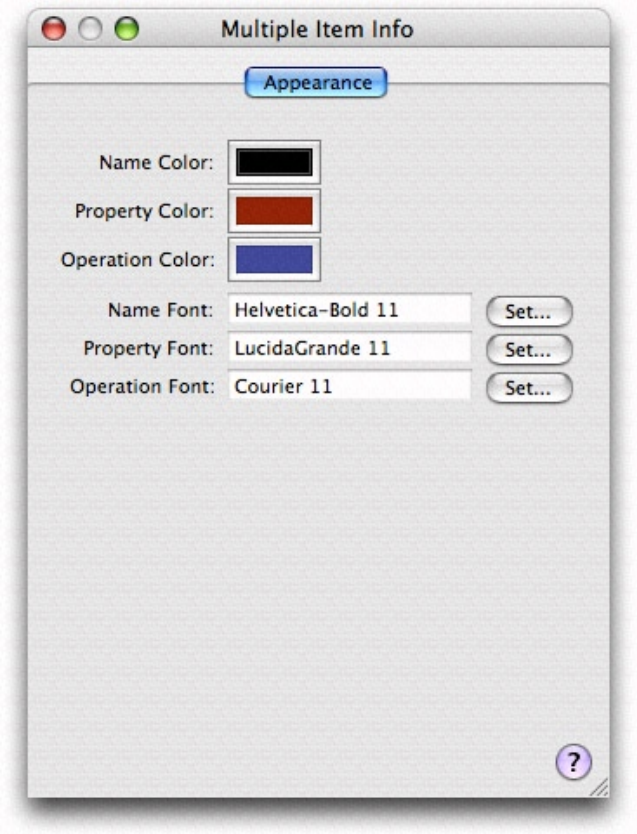
You can use Edit > Select All to select all elements in the diagram. Note that for items in the Diagram menu, clicking on the background (rather than a drawn element) is the equivalent of selecting all, but may be faster.

Colors and Fonts

The diagram view provides default coloring for various elements . By default, all text is black, and the title bar and outline of drawing elements are colored. In the data model the color is the same for all entities.

You change the background color of the title bar and color of the outline of elements by dropping a color swatch from the Color panel onto the element. You change the other color settings, and the font used for the title, property, and operations text, using the Appearance pane of the Info window (inspector). You can also select multiple elements and change their color and text settings simultaneously, as shown in Figure 10-5.

Figure 10-5 Appearance pane showing multiple selection



You can also change the default settings for the entire model using the Appearance pane—see "Info Window" (page 133).

The Browser View

The browser view gives you a different perspective on the whole of your model (note that all classes are shown in the browser view, even if they are hidden in the diagram view). It has three separate parts: two table view panes—the class/entity pane and the properties pane—and the detail pane. You can resize a pane by dragging the vertical divider. You can also hide a pane by resizing its width to zero—to hide the detail pane you must drag the divider on its left side most of the way to the right past its minimum size.

Figure 10-6 The Browser View

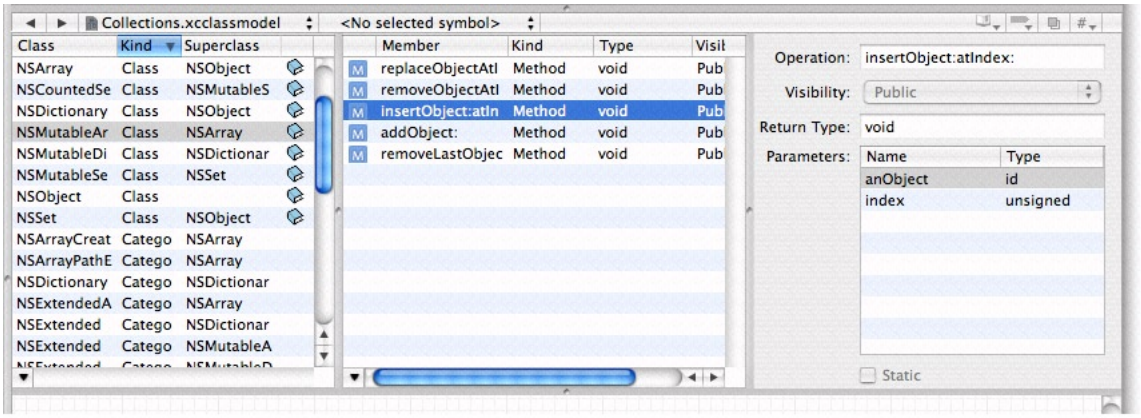


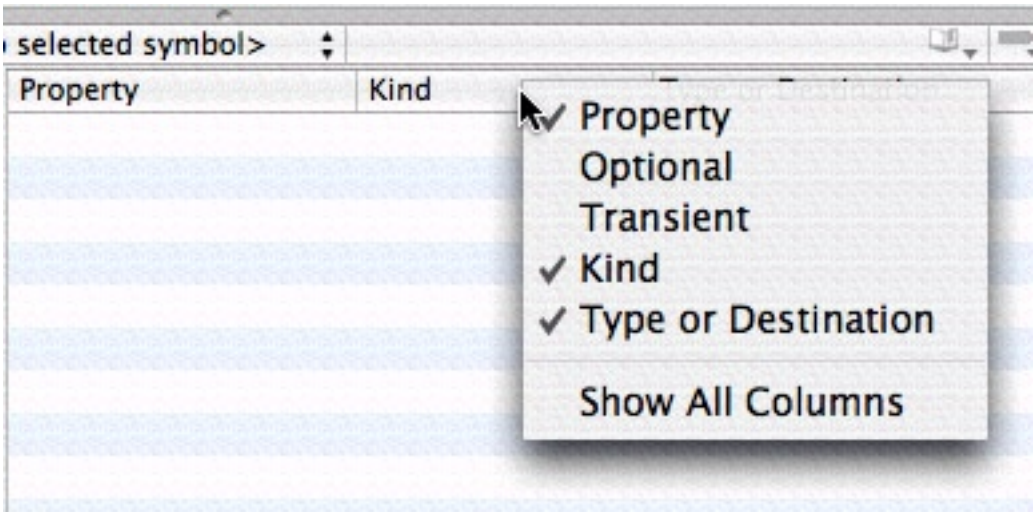
Table View Panes

The left-most pane displays and allows you to edit information about the classes or entities that are in the model. The middle pane displays and allows you to edit information about the properties of the currently selected class or entity. If you make a multiple selection in the class/entity table, the properties table shows the union of all properties of the selection.

As with most table views, you can rearrange and re-sort the columns. You rearrange the columns simply by dragging a header cell; you can change the sort order by clicking in header cells.

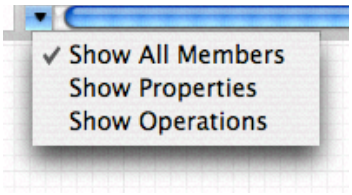
You can choose which columns to see by Control clicking table header cells to display a pop-up list that you can use to toggle the display of columns (see Figure 10-7). If you have a multiple selection, Show All Columns means that only the set of columns common to all members of the selection may be displayed, otherwise you get a specific set (dependent upon what you have selected).

Figure 10-7 Browser column options



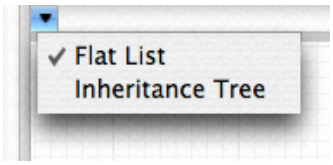
You can also choose which properties are shown by clicking the "v" button to the left of the horizontal scroll bar in the property table. This again displays a pop-up list that you can use to toggle the visibility of properties and operations, as shown in Figure 10-8.

Figure 10-8 Property list view options



Finally, you can display the class/entity list as a flat list or as an inheritance hierarchy. Click the "v" button to the left of the horizontal scroll bar to choose the view option you want as shown in Figure 10-9.

Figure 10-9 Class / Entity View Options



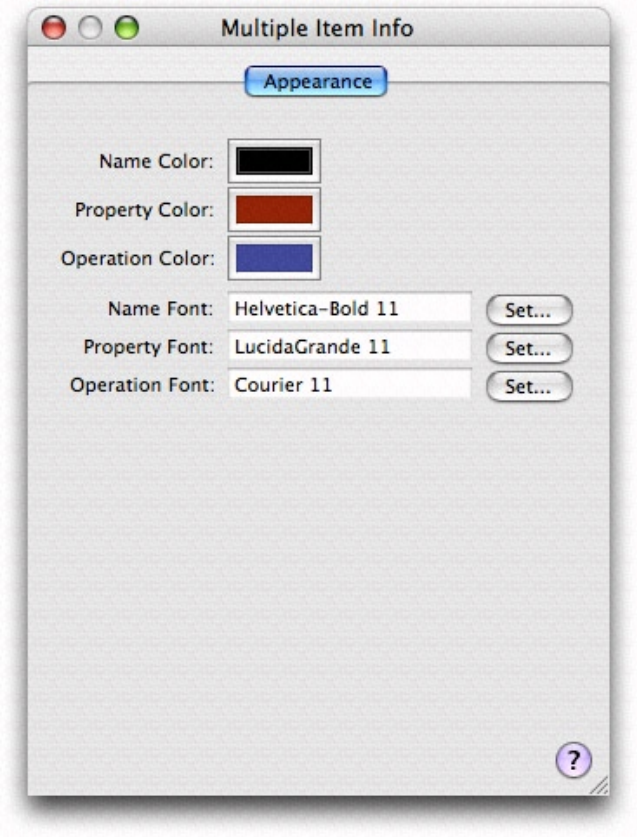
Detail Pane

The detail pane displays detailed information about whatever was last selected in either the entity/class or the property table. If you make a multiple selection, the editor shows the best representation it can of the union of the selected items. If you're using the data modeling tool, this makes it easy to apply changes to a number of entities or properties simultaneously.

Info Window

The Info Window (inspector) is different for data and class models . Both have Appearance panes and the class browser has a General pane with visibility settings and a Tracking pane —see "Indexing and Tracking" (page 138)). You can use the Appearance pane to set default colors and fonts for element names, properties, and operations. Figure 10-10 shows an Appearance pane with custom settings.

Figure 10-10 Appearance pane



Note: To use the Info window, you must make a selection within the browser or the diagram in the document (you can just click the background of the diagram view, for example), not in the Groups & Files browser (for a quick model there may not even be a file icon). If you select a model icon in the Groups & Files list and then choose Get Info, you get an Info window with General, File, and SCM panes.

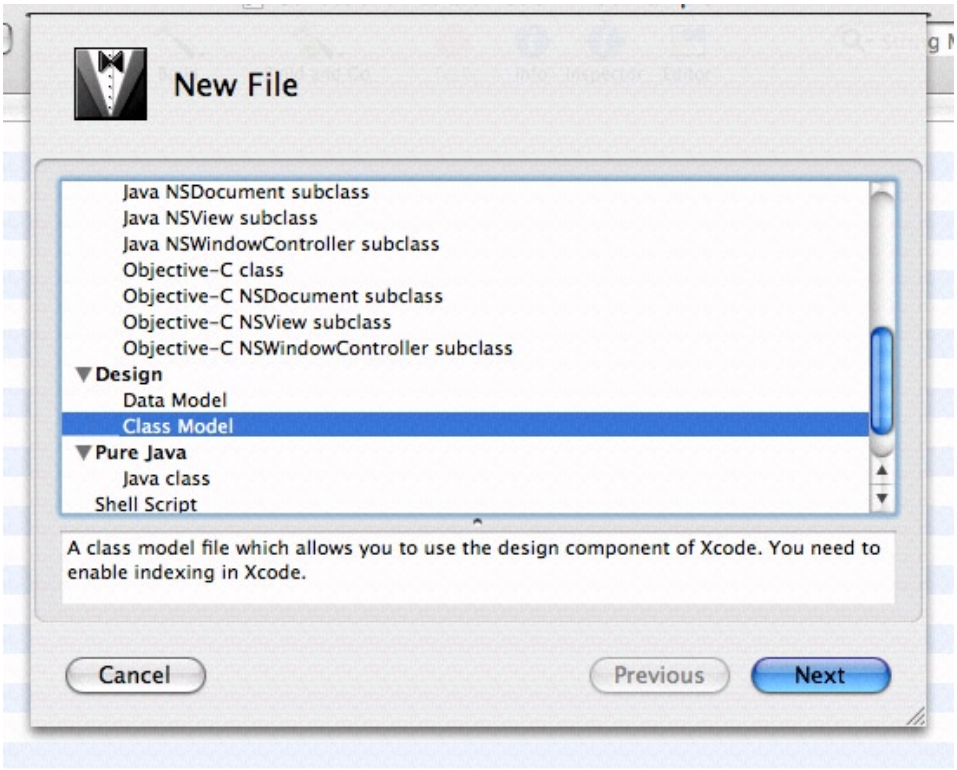
Workflow

The Xcode design tools offer a wide range of options and features to ease your workflow, from automatic page creation and deletion in the diagram view, to multiple selection editing in the browser. In addition, it is possible to add to the toolbar shortcuts for actions such as Add Entity, Attribute, and Quick Model.

Model Files

The usual way you create a new model file is with the File > New File command and the New File Assistant, shown in Figure 10-11.

Figure 10-11 New File Assistant



For class models, you can also create a quick model from the currently selected items using the Design > Class Model > Quick Model command (see "Creating a Quick Model" (page 137)).

Models are considered integral to the project , so you should typically add new class and data models to the project . The exceptions are quick models. Since they are initially considered temporary, you do not have to add them to the project immediately. If you decide to save a quick model, you can add it to the project at a later time. A data model, on the other hand, is a runtime resource (it is compiled, and deployed as part of application), and you should add new data models not only to the project, but also to the relevant target.

Note: Both class model and data model files are actually file packages. Make sure you take this into account when setting up source code management (SCM), copying, and so on.

You might find it useful to create several different class models in your project. Each model may present subsets of the data in different ways. Each gives a its own perspective on the project, so may be useful for different situations.

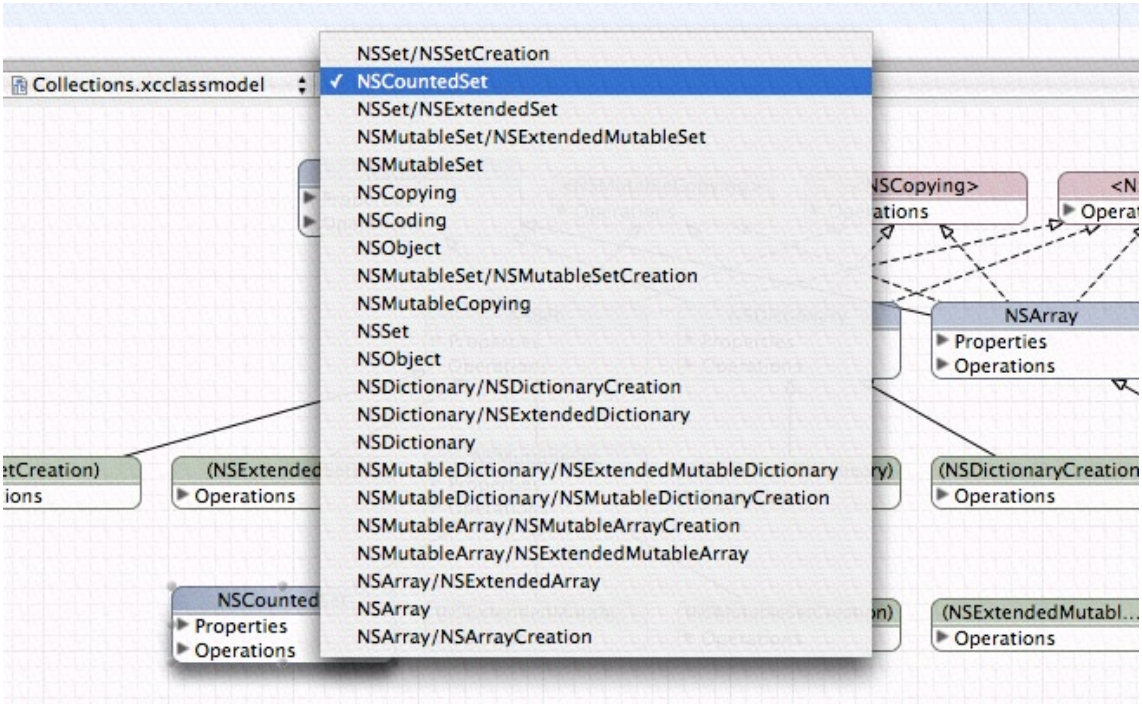
Navigation

You can use the browser and diagram views in conjunction for navigation—the selection in the two views is kept synchronized, so if you make a selection in the browser, the same item is selected in the diagram, and vice-versa.

If you want to see a large model in the diagram view, you can maximize the viewable area of a diagram in the main project window by hiding the toolbar, the navigation bar, the status bar, the Favorites bar, and the browser.

If you have a large class diagram, there are two strategies you can use to aid navigation. First, you can type to select classes or entities. As you type characters, the alphabetically "topmost" class or entity whose name has the prefix you typed is selected. Second, you can use the pop-up menu at the top of the document pane. The pop-up shows a list of elements . When you select an item from the pop-up menu (see Figure 10-12), the corresponding element is selected in the diagram and brought into view. This feature may be particularly useful if the browser is hidden.

Figure 10-12 Elements pop-up menu



The browser view, however, is useful when you have a large class diagram with all compartments rolled up, and you want to see more details about a given class, but don't want to make the diagram bigger. The browser also shows more information than is available in the diagram (parameters, return types and so on).

Contextual Menus

Most menu-based commands are also available from contextual menus associated with the relevant user interface element. You can Control-click a node for immediate access to operations that apply to it or its context, for example to expand compartments, or navigate to documentation. You can Control-click the diagram background to perform operations related both to the visual representation and (in the data model) to the model itself. For example, you can hide grid lines, zoom, set the alignment of drawing elements, and add entities.

Class Modeling With Xcode Design Tools

The Xcode class modeling tool helps you to explore and understand the classes in your project, whether they're written in Objective-C, C++, Java, or a mixture of those languages. It allows you to see class relationships (subclass and superclass relationships—including support for multiple inheritance in C++), protocols (or Interfaces in Java), and categories. In the diagram view, color and text coding help you to quickly distinguish between classes, categories, and protocols; and between project and framework code. The visibility (public, private, protected) of member functions and variables is shown appropriately. (If you are not familiar with any of these terms, you should consult suitable programming texts.)

You can use the tool as an index into your project. From within the tool, you can navigate to the source code of your own classes (both the declaration and implementation), to the declaration in framework classes (those for which you do not have source code), and to corresponding documentation. You can create models that persist as part of your project to communicate design details to other team members, and you can create temporary models (quick models) that serve to illuminate an immediate problem.

The tool's basic features and behavior are described in "Common Features of the Xcode Design Tools" (page 125). This chapter describes features and behavior that are unique to the class modeler.

Creating Models

Xcode allows you to create models in two ways, as quick models and as project class model files (model files you create with the New File Assistant). At first glance it may appear that the two sorts of class model are somehow different. It is important to realize that they are functionally the same but created in different ways and usually with a different immediate purpose in mind.

When you create a model, you add the source or the header files (or both), or containers (such as groups, targets, or projects, but not smartgroups, build phases, find results, and so on) that you want to contribute to the model. A model is dynamic, though. It is continuously updated in response to changes in your source code and the organization of your project. For this reason, you might add both a file and a group containing that file to a model, so that if the file is moved out of group it remains in the model (this strategy may be particularly useful early in a project's lifetime when groups are likely to change).

Note that since class model files depend on the project index, they can't survive outside the project. Note also that when you add a class to a model, its immediate superclass is implicitly added to the model (even if it's not in the project).

Creating a Quick Model

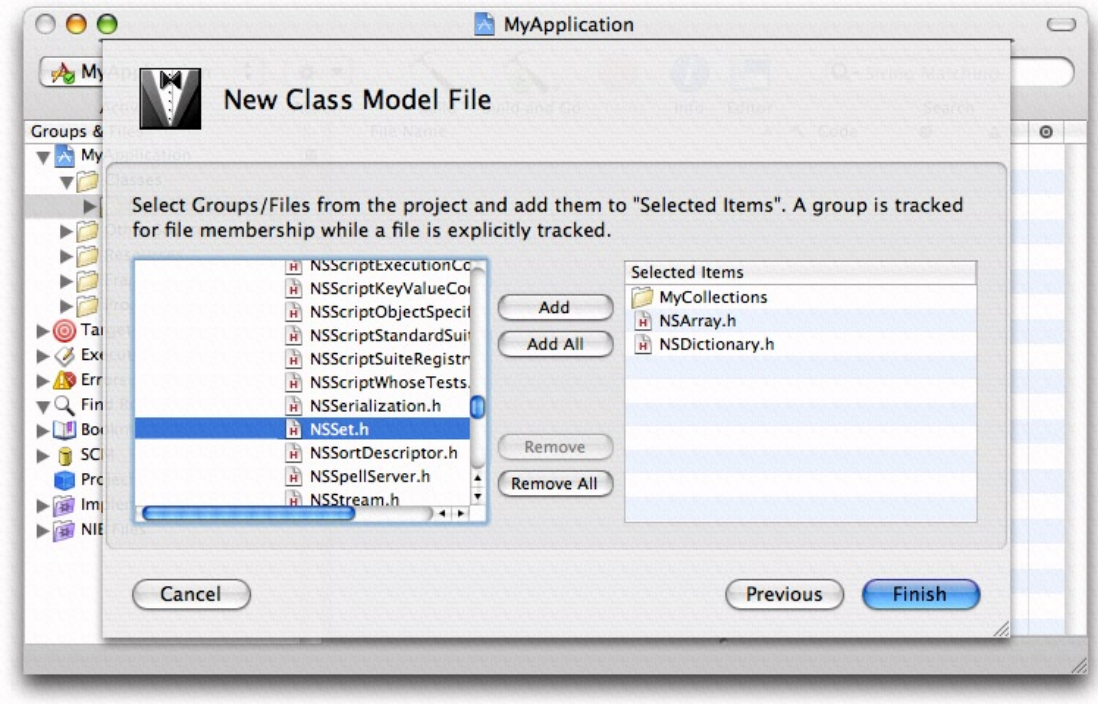
To create a Quick Model, select in the Groups & Files list the files and containers that you want to contribute to the model. Then choose Design > Class Model > Quick Model . Xcode displays the class browser and diagram.

A quick model is untitled and ephemeral. It does not appear in the project file browser, and if unchanged, it is closed without warning. If you make changes, however, you are prompted to save when the project is closed. You can also save the model at any point using Save or Save As if you decide you want to keep the model.

Creating a Class Model File

To create a class model file, choose File > New File and select Class Model from the Design group. You then name the file, and click Next. From the subsequent panel, shown in Figure 11-1, you select the files and containers that you want to contribute to the model.

Figure 11-1 Selecting groups and files to be in the model



When you click Finish, Xcode creates the model file, adds it to the project, and displays the class browser and diagram.

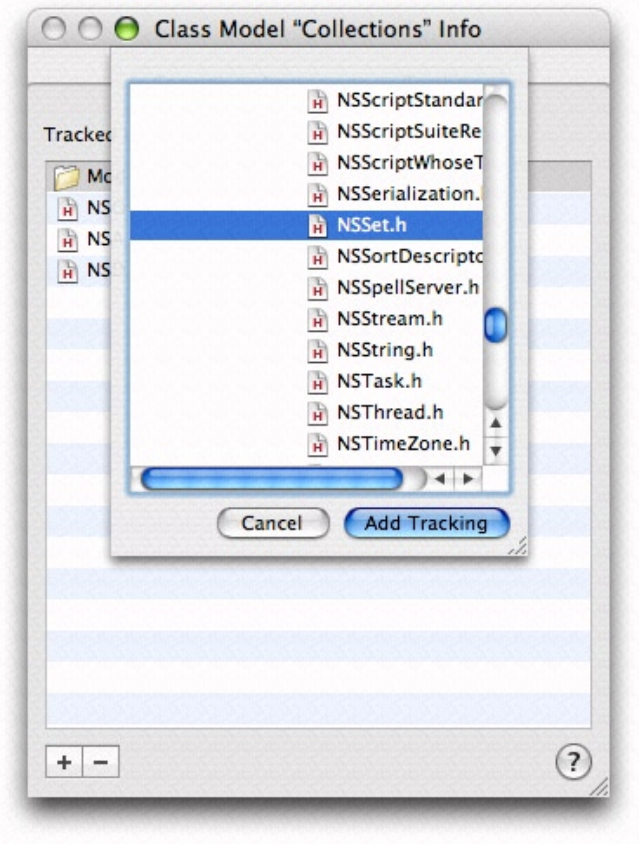
Indexing and Tracking

The tool uses the project indexer to track changes to your project. The class models always represent the actual classes in the files and groups in your project, and are automatically updated as you change your source code—even if you add, remove, or refactor classes. In order to function properly, therefore, the class model requires that the project indexer be enabled.

If the project indexing is not complete, the model pane simply displays the word “Indexing” until indexing is complete. If indexing is disabled, or you open a project on a read-only partition, you see an appropriate warning.

To change the list of tracked items that belong to the model you can use the Tracking pane of the Info window (inspector) as shown in Figure 11-2. Click the plus sign or minus sign (in the lower left of the pane) to add or remove files and groups respectively.

Figure 11-2 Adding a file in the Tracking pane



As you add and remove files from any project groups that make up a model, corresponding classes appear in and disappear from the browser and diagram as appropriate.

The Diagram View for Class Modeling

The diagram contains two important shapes, rounded rectangular nodes and lines. It may also contain annotations. Although you can change the layout and visual appearance of the nodes and lines (and optionally hide classes, properties and so forth), you can modify classes, the relationships between them, their properties, and so forth only by editing source files. For editing annotations, see “Annotations” (page 142).

Nodes in a Class Model

In class models, nodes can be classes, categories, or protocols (interfaces). The name is given in the title bar, and for C++/Java, includes the namespace or package. Nodes are color-coded to help you readily identify different types; different text styles help to further differentiate element and method types. Compartments within a node represent features of the class —properties for instance variables, operations for methods.

You can use the nodes for navigation. To go to your source files (or to the header file for system files), you can click the (>) symbol in the title bar; you can also use the Design -> Class Model menu or the contextual menu to navigate to any declaration, definition, or documentation that is available.

Text and Color Coding

The names of classes, categories, and protocols are represented differently: Class names appear unadorned; category names are surrounded by parentheses, and protocol names are surrounded by angle brackets. If an operation name is underlined, it is a class method .

By default, classes are represented in blue, categories in gray, and protocols (interfaces) in red. Project and framework classes are further differentiated by the saturation of the color (externals are dimmer) . You can change both the default colors and colors for individual nodes (see "Common Features of the Xcode Design Tools" (page 125)).

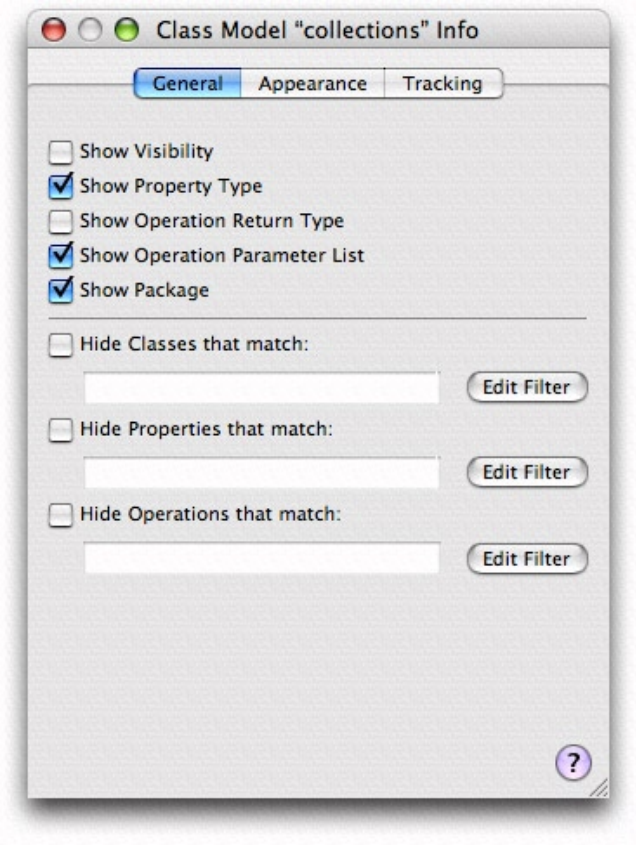
Compartments and View Options

Compartments within a node represent features of the class . The Properties compartment lists instance variables ; the Operations compartment lists methods—class method are underlined.

Within a node, you can display additional information. Using the General pane of the Info window (shown in Figure 11-3), you can choose whether or not to show:

- Visibility flags (public, private, and protected may be indicated by an icon in the compartment)
- Property type (for each property, shown after a colon in the compartment)
- Operation return types and parameter types
- Package information

Figure 11-3 Info window for a class model diagram



You have further control over what is displayed in the diagram—see "Filtering and Hiding" (page 142).

Lines

Lines indicate different things depending on whether they are solid or dashed, what sort of arrowhead is present, and what objects they connect.

A solid line with an open arrowhead:

- Denotes inheritance when it connects classes
- Specifies the class of which the category is a category when it connects a class and a category

A dashed line with an open arrowhead denotes implementation of a protocol (or interface in Java).

You cannot edit lines other than to or from annotations—they are created automatically based on the contents of your project.

Annotations

You can add annotations to the diagram to provide explanatory text using the text tool. You have full access to text styling options from the Format menu. You can connect a comment to a class using the line tool.

Filtering and Hiding

Sometimes diagrams contain more information than you want to see, and it can be useful to reduce clutter. Removing irrelevant classes makes it easier to concentrate on important ones—for example, you might remove NSObject from a class diagram to make it easier to see other relationships; or it may be that a tracked file contains definitions of several classes, and you are interested in only one of them. You might also want to hide other details, such as private variables, or instance variables whose name starts with an underscore.

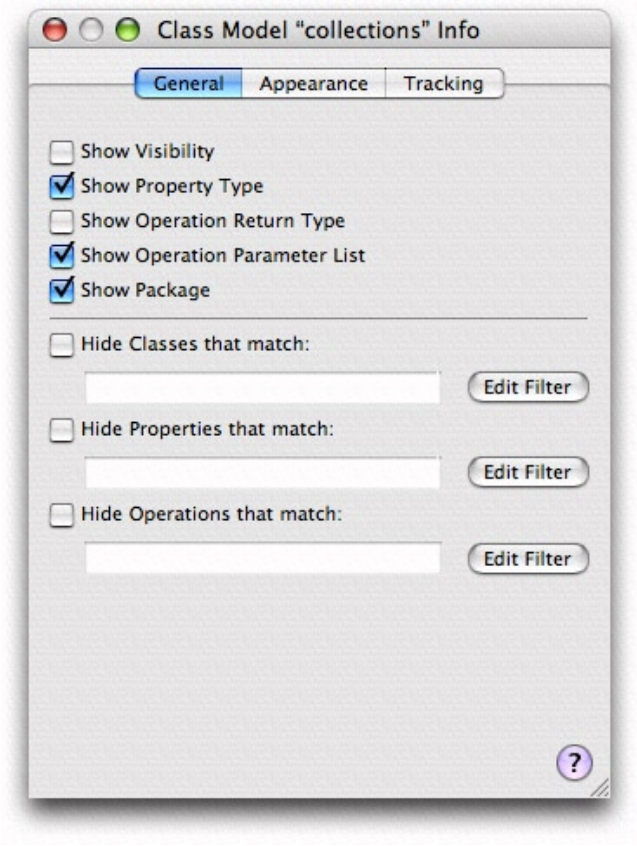
You can use a predicate to filter what classes and methods are shown in the diagram. On a class level, you can choose to use or override the filter. You can show a class if a filter would normally hide it, and vice versa.

Filtering and hiding settings affect only the diagram. The browser view still lists all the contents (you need a way to be able to select a class if it's hidden!). Filtering and hiding are different from tracking. Tracking determines whether or not files contribute to the model at all.

Filtering

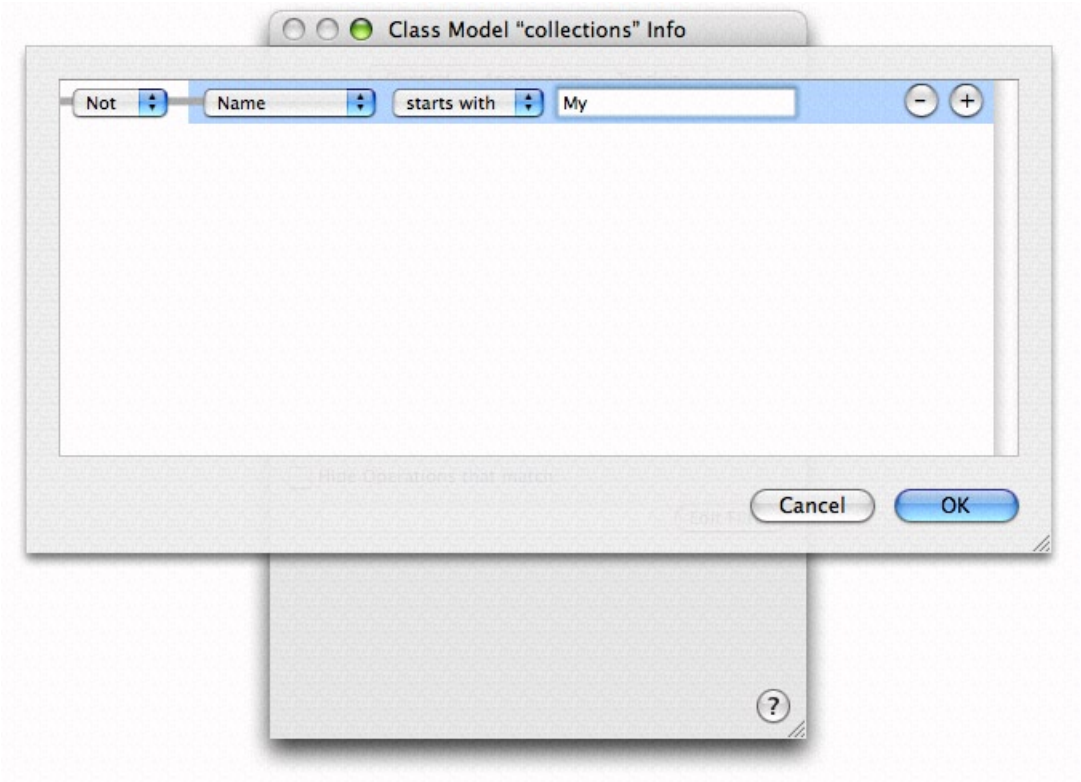
Filters apply as changes are made to files that contribute to the model. If you chose, for example, to hide all classes whose names begin with "XYZ", then in your header file you rename the class "XYZWidget" to "WXYWidget", a node for "WXYWidget" will appear in your diagram. To set up filtering, you use the General pane of the Info window, shown in Figure 11-4.

Figure 11-4 General pane of the Info window



You can set up independent filters for classes, properties, and operations, based on their name and kind. You can also toggle filters on and off as required. If you click Edit Filter, a sheet in which you can edit the filter appears, as shown in Figure 11-5. You can either enter a predicate directly into the appropriate text field or use the predicate builder (see "The Predicate Builder" (page 153)).

Figure 11-5 The filter editor

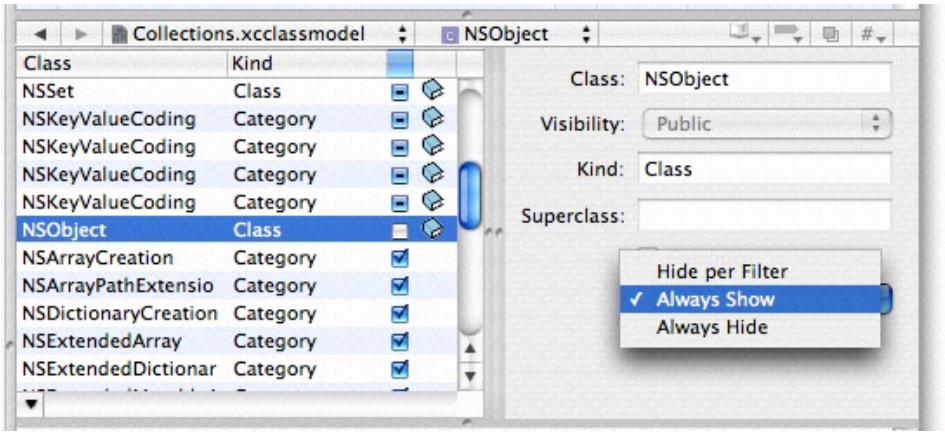


Note: Filters are labeled to hide classes, properties, or operations. Sometimes you want to show only a subset. You can apply a "NOT" expression to a predicate to invert its meaning. Therefore if you wanted to show only those classes whose names begin with "XYZ", you hide those whose names do not begin with "XYZ".

Hiding

Hiding allows you to override the filtered state of a class (or protocol, or category). You can specify that an element should follow the filtered setting, or be either always shown or always hidden. You can set the hiding state using the browser view, in either the Hidden column in the property pane or using the pop-up menu in the detail pane, as shown in Figure 11-6 .

Figure 11-6 Setting hiding in the detail pane



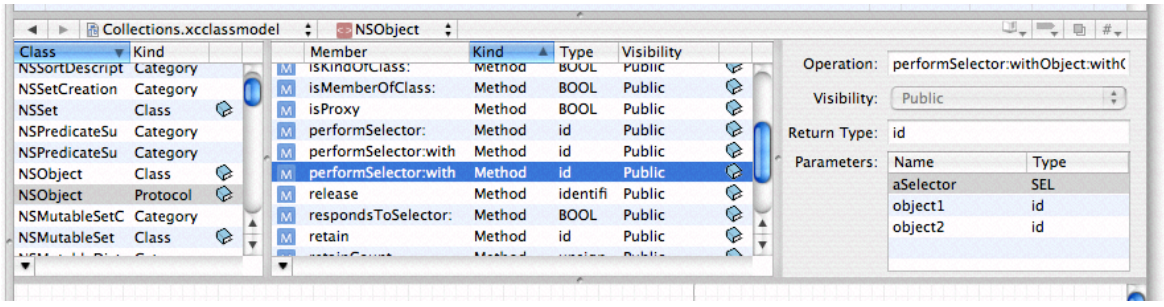
In the detail pane, you choose the setting from the pop-up menu. The browser has a three state checkbox you can use to modify the hiding setting. The states correspond to the same states defined in the pop-up menu.

Note: If you find you are making frequent changes to the hiding settings, you may consider creating different models to provide different perspectives. Models are lightweight, so do not consume significant system resources, and they use tracking so are always up to date.

The Browser View for Class Modeling

Most features of the browser view behavior are common to both the class model and the data model, and described in "Common Features of the Xcode Design Tools" (page 125).

Figure 11-7 The browser view in the class modeling tool



The table view in the left-most pane lists the classes, categories, and protocols in the model. The columns in the class list show the element name, the element type (class, category, or protocol), the hidden status, and a link to documentation if there is any associated with the element.

The table view in the properties pane shows summary information about the properties and methods associated with the current selection in the classes table, including the name, type, and visibility, and again a link to documentation if there is any associated with the element.

The detail pane shows information about the most recently selected element in the classes or properties table. You use the detail panel to set the hidden status of individual class elements. This setting affects what is displayed in the diagram view, as described in "Hiding" (page 144).

Data Modeling With Xcode

The purpose of the data modeling tool is to create a data model (or schema) for use with the Core Data framework. Rather than dealing with classes, instance variables, and methods, you use the tool to define entities, the attributes they have, and the relationships between them. For more about entity-relationship modeling, object modeling, why these are important, and definitions of terms (inverse relationship, optional attribute, and so on), see Object Modeling. For more about specific Core Data classes, see the relevant API reference documentation.

Ultimately, at runtime, the model is turned into an instance of NSManagedObjectModel with a collection of NSEntityDescription, NSAttribute, NSRelationship, and NSFetchRequest objects. In some respects this is analogous to the behavior of Interface Builder. With Interface Builder, you graphically create a collection of objects that are then saved in a file (a nib file) and recreated at runtime. As with user interface elements, it is possible to create a model directly in code at runtime; however it is typically easier to do so graphically using the appropriate tool. Similarly, just as it is possible to modify the user interface after it has been loaded, it is also possible to customize a model after it has been loaded. (Note that a model does have a constraint not shared with a nib file: once loaded, a model cannot be modified after it has been used.)

The tool's basic features and behavior are described in "Common Features of the Xcode Design Tools" (page 125). This chapter describes features and behavior that are unique to the data modeler.

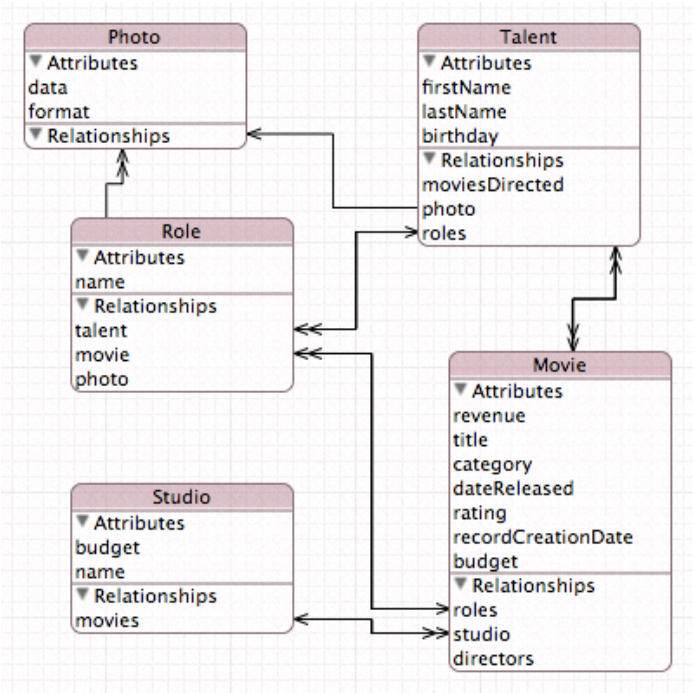
The Diagram View for Data Modeling

The diagram view for the data modeler contains the same sort of graphic elements as the class modeler, as illustrated in Figure 12-1 (page 148). The semantics of the elements, however, are different:

- Nodes are entities, not classes.
- Compartments within a node show attributes and relationships.
- Lines represent relationships between entities.

Arrowheads on lines also have meaning. A single arrowhead denotes a to-one relationship; a double arrowhead denotes a to-many. The direction of an arrow indicates the direction of the relationship—the arrow points to the destination entity. Figure 12-1 shows an example of the diagram view of a data model, with all compartments expanded.

Figure 12-1 Example diagram view of a data model



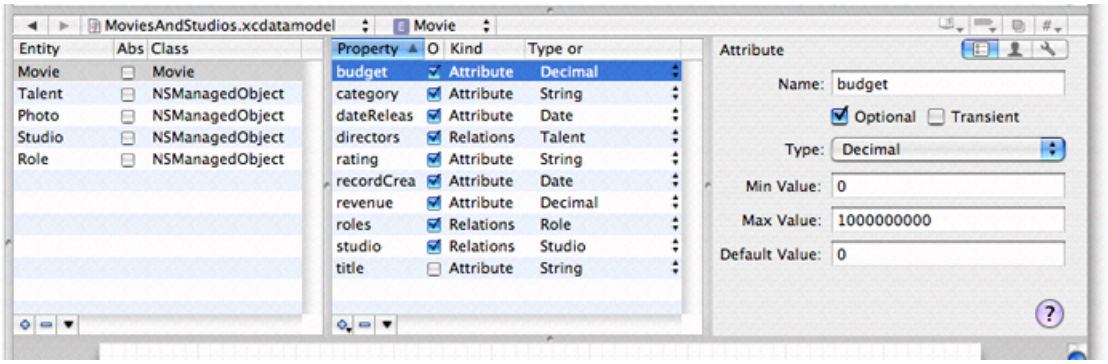
You can edit the model directly from the diagram using contextual menus. To add a new entity, you Control-click the background of the diagram. To add properties to an entity, you Control-click within its node. You can also delete entities and properties using the Delete key. Finally, you can use the line tool to establish new relationships. You select the line tool, then drag from the source node to the destination node.

Note that although relationships are typically implicitly bidirectional, relationships do not have to be modeled in both directions. If do you want to specify a bidirectional relationship, you must model both sides of the relationship—the reasons for this are given in the Core Data documentation. Moreover, within the model you must specify which relationships are the inverse of each other. To do this you need to use the model browser.

The Model Browser for Data Modeling

The data modeling tool browser's three parts are the entities pane, the properties or fetch requests pane, and the detail pane. The detail pane itself has three separate views: the General pane, the User Info pane, and the Configurations pane.

Figure 12-2 Example of a browser view for a data model



The Entities Pane

The table in the entities pane lists all the entities in the model, either as a flat list or in an inheritance hierarchy. The table has three columns, showing the entity name, the class used to represent the entity, and a checkbox that indicates whether the entity is abstract.

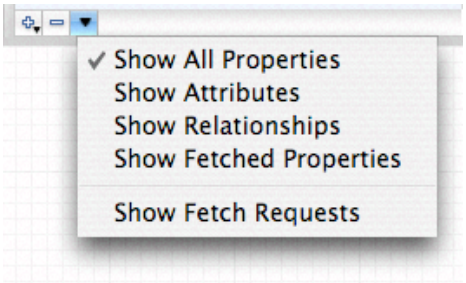
You can edit the entity and class names directly in the text field cells—double click the text to make it editable—and toggle the abstract setting of an entity by clicking the checkbox.

To add a new entity to the model, you click the plus sign to the left of the horizontal scroll bar, or choose Design > Data Model > Add Entity. You delete a selected entity or selected entities by clicking on the minus sign, or by pressing the Delete key.

The Properties Pane

The table in the properties pane lists the properties or fetch requests associated with the selected entities. You choose what features you want to view by choosing from the pop-up menu available from the button with the "v" to the left of the horizontal scroll bar, as shown in Figure 12-3.

Figure 12-3 Properties table options

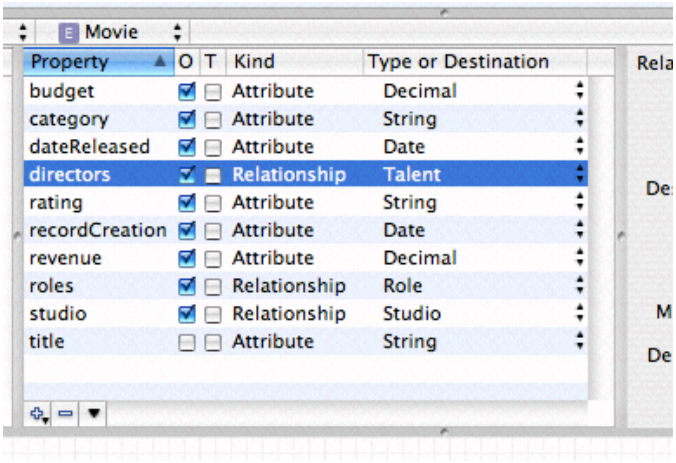


Note that the properties table shows the set of all properties of all entities selected in the entities table. Moreover, you can select and edit multiple properties at the same time. If several entities have a similar property, you can change them all simultaneously if you wish.

Properties View

The properties table has five columns showing the name of the property, a checkbox that indicates whether the property is optional, a checkbox that indicates whether the property is transient, the kind of property (attribute, relationship, or fetched), and the type (for example, date or integer if the property is an attribute) or destination entity (if the property is a relationship) of the property (see Figure 12-4).

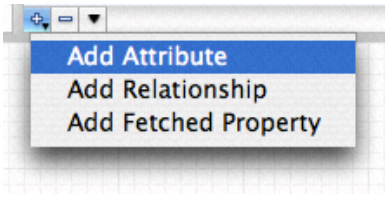
Figure 12-4 Properties view



You can edit most property values directly in the properties table—the exception is the property type ("Kind") which you specify when you first add the property. You typically edit the predicate associated with fetched properties from the detail pane, using the predicate builder(see "The Predicate Builder" (page 153)).

You add new properties using the pop-up menu from the plus sign to the left of the horizontal scroll bar (as shown in Figure 12-5), or by using the Design > Data Model menu. From the pop-up menu, you choose what sort of property you want to add—an attribute, a relationship, or a fetched property.

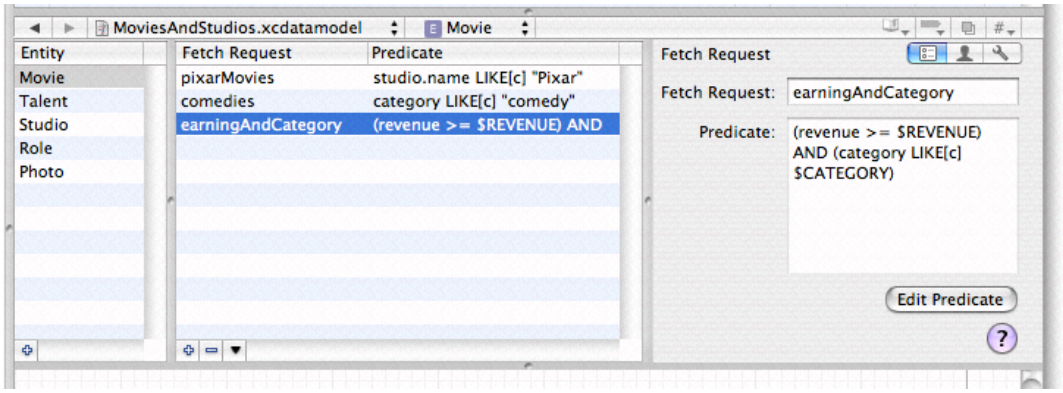
Figure 12-5 Adding a property



Fetch Requests View

The fetch requests view displays the fetch requests associated with an entity as shown in Figure 12-6. You add fetch requests using the plus sign button. You can edit the name of the fetch request and the predicate directly in the table view; however you typically construct the predicate graphically using the predicate builder from the detail pane.

Figure 12-6 Fetch requests view



When you add a fetch request to an entity, you are specifying that that entity is the one against which the fetch will be performed. For example, if you add a fetch request called “comedies” to the Movie entity, in code you would retrieve it from the model using:

```
NSFetchRequest *fetchRequest = [managedObjectModel
        fetchRequestTemplateForName:@"comedies"];
```

The returned fetch request’s entity is set to Movie. Since fetch requests are nevertheless general to the model, fetch request names must be unique across all entities. If you try to set a duplicate name, you get a warning sheet and you must choose a unique name before you can proceed.

The Detail Pane

The detail pane itself has three panes, the General pane, the User Info pane, and the Configurations pane. You choose which pane to display by clicking on the corresponding element in the segmented control in the upper right of the pane, shown in Figure 12-7.

Figure 12-7 Control for choosing the pane in the detail pane

General Pane

The general pane is different for entities, attributes (and for different types of attribute), relationships, and fetch requests. It changes automatically to the appropriate view depending on the last selection. Each view shows, and allows you to edit, details of the selected element.

- For entities, you can edit the entity name, the name of the class used to represent the entity, and the parent entity, and you can specify whether or not the entity is abstract.
- For attributes, you can specify the name and type, and whether it is optional or transient. When you specify the type, the pane updates to allow you to specify various constraints on the values the attribute may take. For example, for numeric and date attributes you can specify maximum, minimum, and default values, and for string attributes you can specify maximum and minimum length, a default value, and a regular expression that the string must match.

- For relationships, you can specify the name, cardinality, and destination of the relationship. You can also specify a delete rule, and—for to-many relationships—maximum and minimum counts.
- For fetched properties, you specify the name, the destination entity, and the predicate to be used for the fetch. You edit the predicate using the predicate builder by clicking the Edit Predicate button. For more details about the predicate builder, see "The Predicate Builder" (page 153).
- For fetch requests, you specify the name and the predicate. As with fetched properties, you edit the predicate using the predicate builder by clicking the Edit Predicate button—see "The Predicate Builder" (page 153).

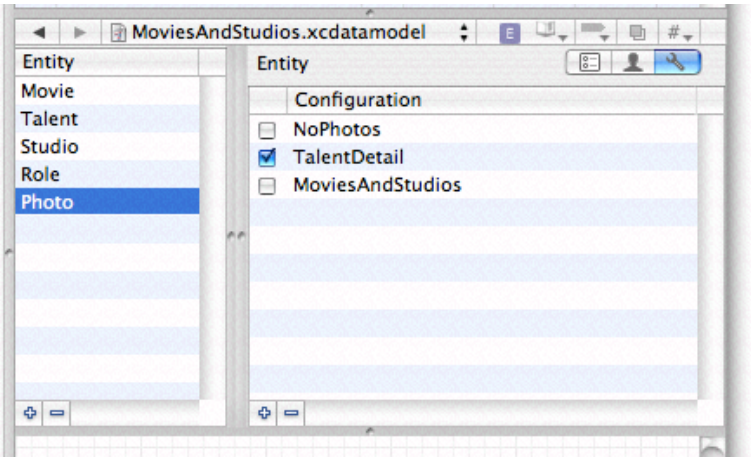
User Info Pane

The user info pane shows the info dictionary associated with the currently selected model element. Most elements in the model (entities, attributes, and relationships, but not fetch requests) may have an associated info dictionary that you can retrieve at runtime. The dictionary comprises key-value pairs. Using the info dictionary pane, you can specify any keys and string values you wish that may be of use in your application.

Configurations Pane

A configuration is a named collection of entities in the model. The configuration pane (show in Figure 12-8) therefore applies only to entities. You use it to add and remove configurations and to associate entities with configurations.

Figure 12-8 Configurations pane of the detail pane



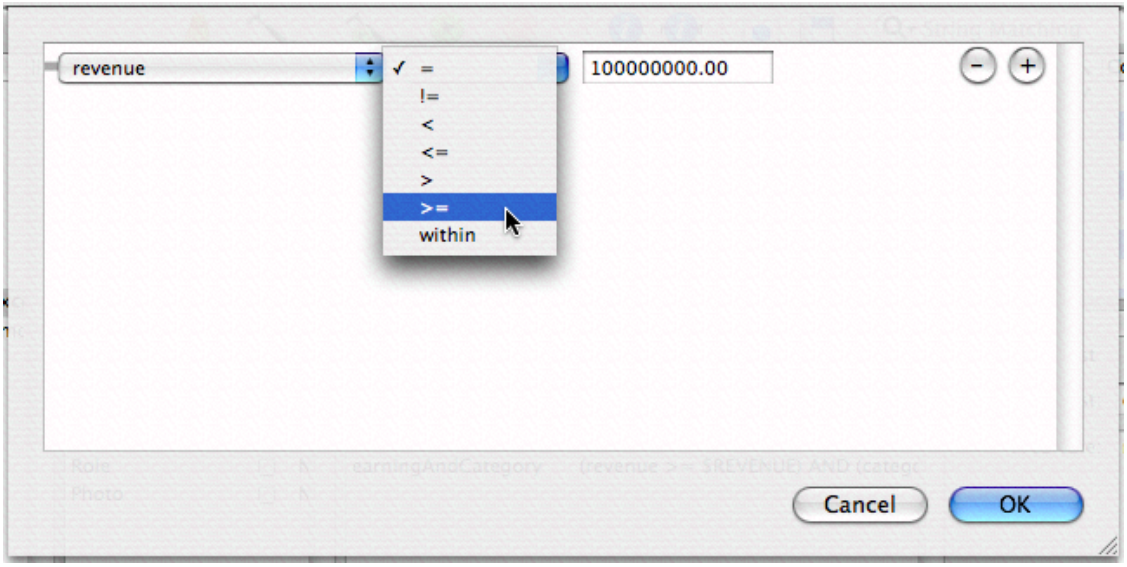
A model may have an arbitrary number of configurations. You add configurations using the plus sign button. Configurations appear in the list for all entities. The checkbox specifies whether or not the currently selected entity is associated with the given configuration.

The Predicate Builder

You use the predicate builder to create predicates for fetched properties and for fetch request templates. For more about predicates, see `NSPredicate` and for more about fetched properties, see `NSFetchedPropertyDescription`. Fetch request templates allow you to create predefined instances of NSFetchRequest that are stored in the model. You can either define all aspects of a fetch, or you can allow for runtime substitution of values for given variables. Fetch templates are associated with the entity against which the fetch will be made, that is, instances of which the fetch will return. For more about fetch templates, see `NSManagedObjectModel`.

With the predicate builder, you can build predicates of arbitrary complexity. The initial display shows a simple comparison predicate. The left-hand side is pop-up menu that allows you to choose the key used in a key path expression; the right-hand side is a text field that allows you to specify a constant value; and between them is a pop-up menu that allows you to choose a comparison operator.

Figure 12-9 Predicate builder



As with the rest of the modeling tool, the predicate builder simply provides a graphical means of defining a collection of objects that you could otherwise create programmatically. The code equivalent of the predicate `revenue >= 100000000` is as follows.

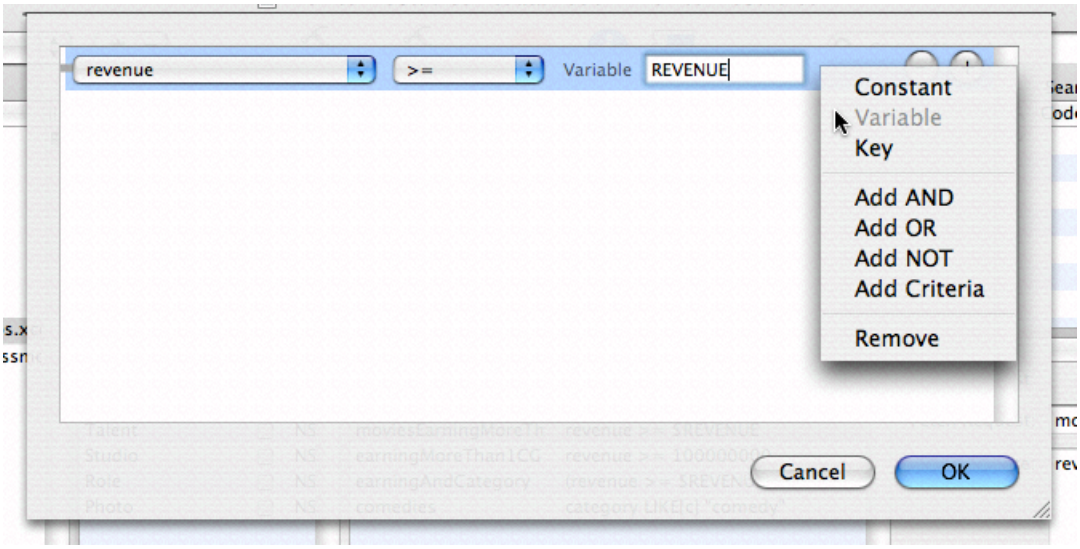
```
NSExpression *lhs = [NSExpression expressionForKeyPath:@"revenue"];
NSExpression *rhs = NSExpression *rhs = [NSExpression
expressionForConstantValue:[NSDecimalNumber  numberWithInt:100000000]];
NSComparisonPredicate  *predicate = [NSComparisonPredicate
    predicateWithLeftExpression:lhs
    rightExpression:rhs
    modifier:NSDirectPredicateModifier
    type:NSGreaterThanOrEqualToPredicateOperatorType
    options:0];
```

Right-Hand Side

In addition to a constant value, you can also define the right-hand side of a comparison predicate to be a variable or another key. This is necessary if you are creating either fetch request templates that require substitution variables or defining fetched properties and need to use the `$FETCH_SOURCE` variable in the predicate.

You change the type of the right hand side expression using the contextual menu shown in Figure 12-10 (you must Control-click "empty space" in the line of the criteria—for example, at the end or between the pop-up menus). This changes the constant value field into a variable field or a key pop-up menu as appropriate.

Figure 12-10 Right-hand side expression type



Left-Hand Side

The key pop-up menu, shown in Figure 12-11, displays only attributes of the entity with which the predicate is associated. To use a key path (that is, to follow relationships), choose the Select Key item in the key pop-up menu. This displays a browser, shown in Figure 12-12 (page 155), from which you can choose the key or key path you want.

Figure 12-11 Predicate keys

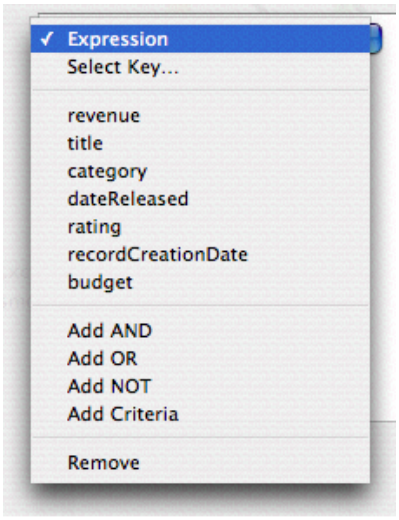
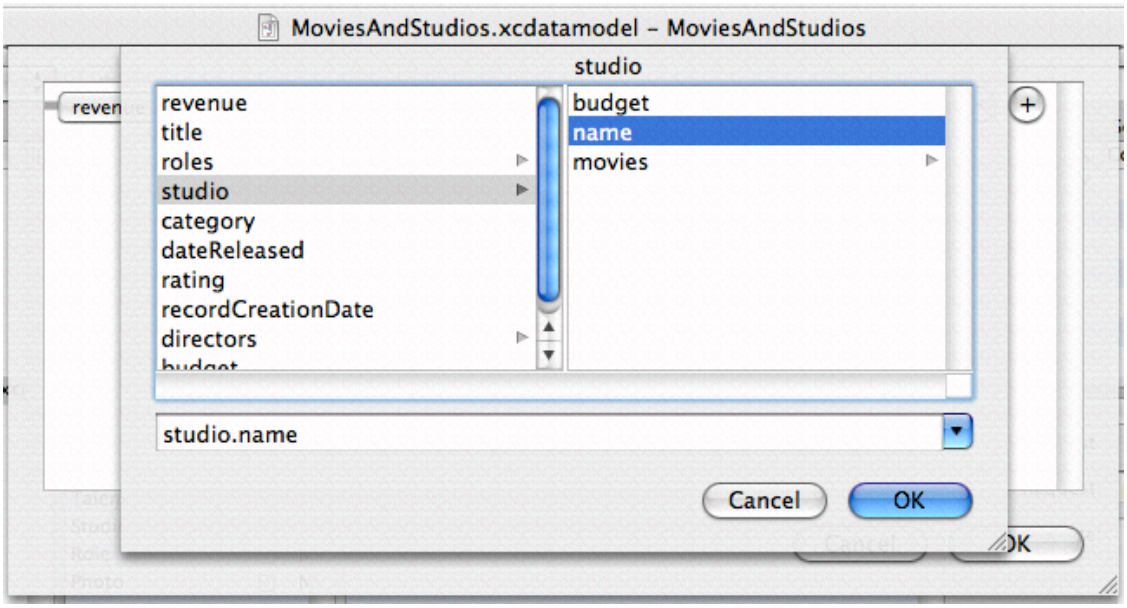


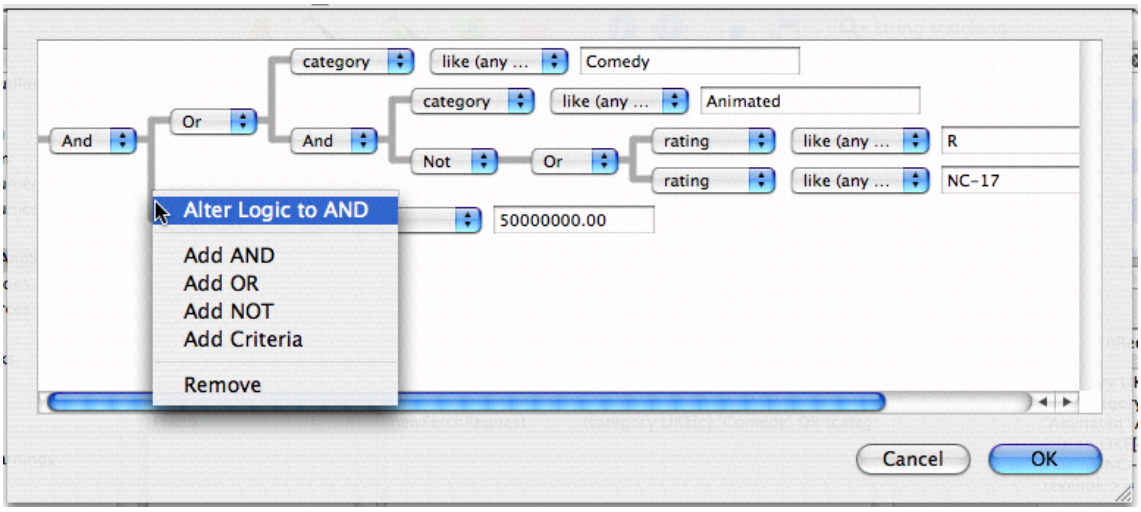
Figure 12-12 Adding a key path



Compound Predicates

You can add logical operators (AND, NOT, and OR) to create compound predicates of arbitrary depth and complexity. To add a specific logical operator, use the contextual menu, shown in Figure 12-13.

Figure 12-13 Creating a compound predicate



You can also add peer predicates by clicking the round button with the plus sign, or using the Add Criteria command in the contextual menu—these add an AND operator. You can change a logical operator using the pop-up menu. You can rearrage the predicate hierarchy by dragging. To remove a predicate, click the round button with the minus sign, or use the contextual menu. The predicate builder will try to rebuild the remaining predicate as it can, removing comparison operators where appropriate.

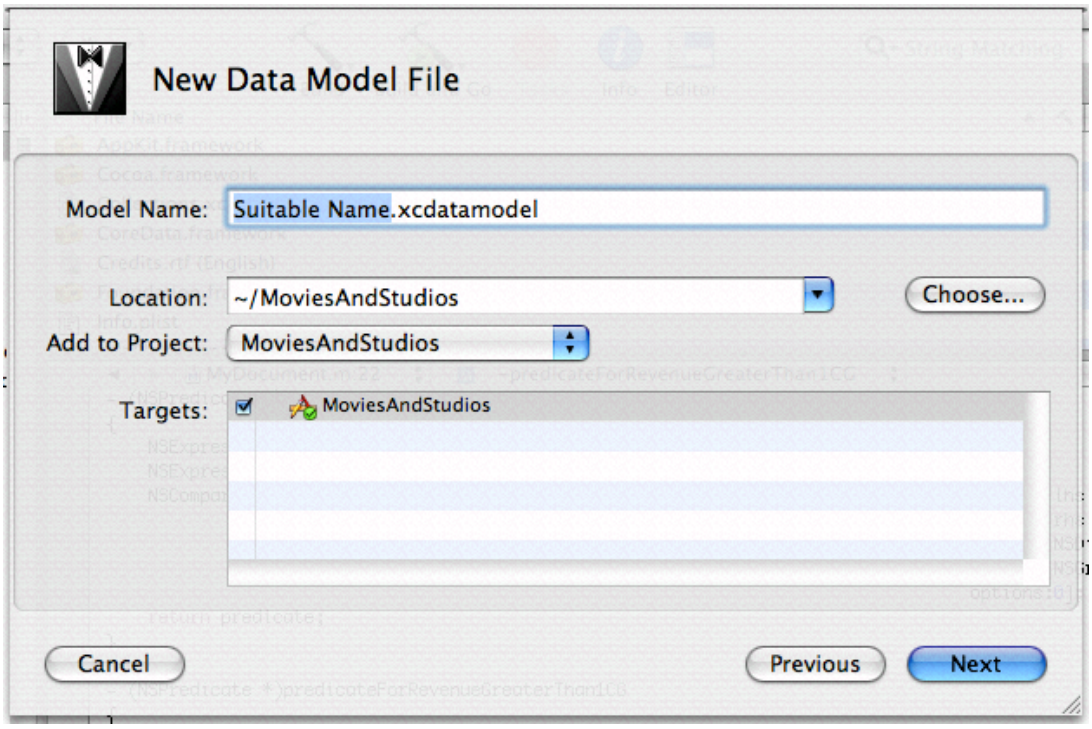
Workflow

The typical steps you take are defining your entities, specifying the attributes they have; specifying relationships between them; and adding business logic in the form of default values and value constraints. You may also define fetch templates for an entity. Although you can edit the model in the diagram view, it is more usual to do so in the browser, since it gives you more detail and greater flexibility. Creating a Managed Object Model Using Xcode provides a task-based approach to creating an entire model, from start to finish.

Creating a Model

If you create a Core Data–based project, a data model is automatically created for you and added to the project. If you need to create a new model, from the File menu choose New File and add a file of type Data Model (from the Design list). In the pane that appears (see Figure 12-14), give the file a suitable name and ensure that the file is added to your application target.

Figure 12-14 Creating a New Data Model File



Click Next, and in the following pane select any groups or files that you want to be parsed for inclusion in the model (if any), then click Finish.

Custom Classes

For each entity in the model, you specify a class that will be used to represent it in your application. By default the class is set to NSManagedObject, which is able to represent any entity. Typically, at the begnining of a project, you just use NSManagedObject for all your entities. Later, as your project matures, you define custom subclasses of NSManagedObject to provide custom functionality.

If you create a custom subclass of NSManagedObject to represent an entity, you typically implement custom accessor methods for the class's properties. This is generally tedious, repetitive work, so the data modeling tool provides menu items to automatically generate declarations and implementations for these methods and put them on the Clipboard so you can paste them into the appropriate source file.

Compiling a Data Model

A data model is a deployment resource. A data model must not only be a project file, it must be associated with the target that uses it . In addition to details of the entities and properties in the model, the model contains information about the diagram, its layout, colors of elements, and so on. This latter information is not needed at runtime. The model file is compiled to remove the extraneous information and make runtime loading of the resource as efficient as possible.

The model compiler is `momc` in `/Library/Application Support/Apple/Developer Tools/Plug-ins/XDCoreDataModel.xdplugin/Contents/Resources/`. If you want to use it in your own build scripts, its usage is: `momc` *`source destination`*, where *`source`* is the path of the Core Data model to compile, and *`destination`* is the path of the output mom file.

Editing Source Files

Much of the development process is actually spent writing source code. You spend a lot of time editing the files in your project, from authoring source code to modifying file characteristics, such as the file's encoding. Xcode includes a full-featured editor that supports a number of features that make coding easier. For example, syntax coloring helps you distinguish the various code elements in a source file. Syntax-aware indenting makes it simple to generate readable, well-formatted source code by automatically indenting source code, and matching braces, as appropriate for the current context.

Xcode's editor can display a great deal of information about the current file, such as line numbers, the location of breakpoints and build errors, and more. It supports a number of quick-access features, such as the ability to jump to any symbol definition or declaration in a file, jump between header and implementation files, and look up documentation for a symbol. In this way, you can quickly move around within or between files and find the information you need.

Using the symbol information available through Code Sense, Xcode's editor also supports code completion. As you type in a source code file, code completion suggests symbols appropriate for the current context. You can have Xcode automatically insert the symbol name and prototype, instead of typing all of the information yourself.

Xcode gives you many ways to open files and access information in an Xcode editor. You can choose to have the editor open as a standalone window, or you can use the attached editor to open files directly in the project, Build Results, Project Find, or Debugger windows.

Of course, many people are accustomed to the behavior of one particular text editor for authoring source code. They are extremely productive in this environment and prefer to keep using this editor, rather than learning the ins and outs of a new text editor. For this reason, Xcode also lets you specify an external program for opening and editing files in your project. In this way, you can manage project files and perform all other development tasks in Xcode while still opening and editing files in your usual editor. If you do decide to use Xcode's editor, you can use Xcode's built-in Metrowerks, BBEdit or MPW key binding sets for text editing operations to make your editing environment as familiar as possible.

The chapters that follow describe Xcode's editor, show how to open and access files with Xcode's editor or with an external editor, and show you how to take advantage of features such as code completion and syntax coloring to make the process of authoring source code easier and more efficient.

Inspecting File Attributes

For each file, framework, and folder added to your project, Xcode stores a number of important settings, such as the path to the file and the file's type. You can view and edit settings for file, framework, and folder references in the file reference inspector or Info window.

This chapter describes how to inspect file, folder, and framework references in your project. It also describes how to change the way in which Xcode handles a file by changing its type; and how to control the way a file is displayed and saved, by changing the file encoding and line ending.

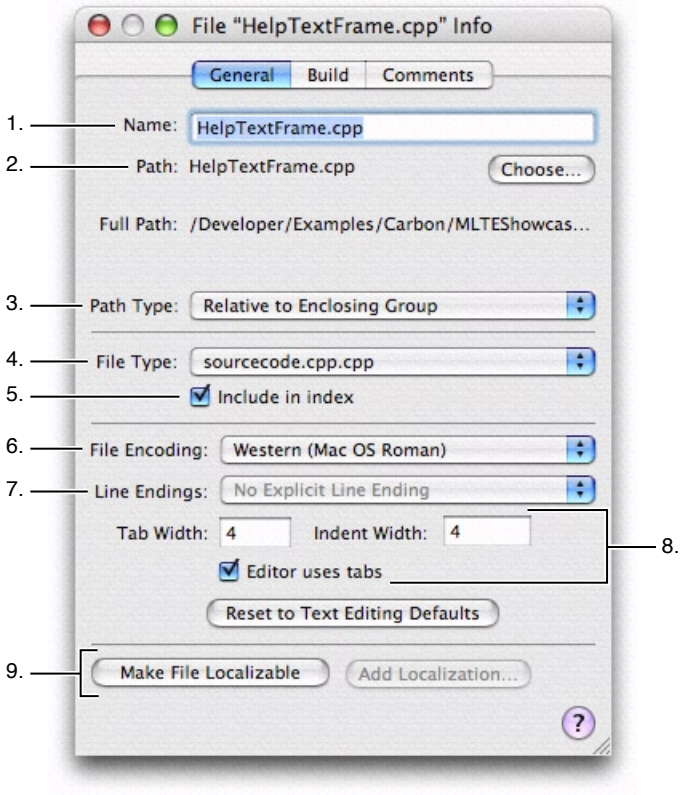
Inspecting File, Folder, and Framework References

You can view and edit settings for file, framework, and folder references in the file reference inspector or Info window. To bring up the Info window, click the Info button in the project window toolbar. If you select more than one file reference, Xcode displays one Info window that applies to all selected files. Attributes whose values are not the same for all selected files are dimmed. Changing an attribute's value applies that change to all selected file references.

The Info window for a file, folder, or framework reference or source group contains the following:

- The General pane, shown below, lets you modify a number of basic file attributes, including filename, location, reference style, and so forth.
- The Build pane lets you specify additional compiler flags for the file. The Build pane appears only when the file being inspected is a source code file. See "Per-File Compiler Flags" (page 294) for more information on the contents of this pane.
- The SCM pane lets you view SCM information for the file. This pane is only available for files under version control. See "Viewing Revisions" (page 219) for more information.
- The Comments pane lets you add notes, documentation, or other information about the file. See "Adding Comments to Project Items" (page 92) for more information on adding comments.

Figure 13-1 Inspecting a file



Here is what you can see and edit in the General pane of the file reference inspector:

1. The Name field displays the file's name. To rename the file, type the new name in this field.
2. The Path field shows the location of the file; that is, it shows the path to the file. To change this location, click the Choose button next to the path. You will get a dialog that lets you choose a new path.
3. The Path Type menu indicates the reference style used for the file. These reference styles are described in "How Files Are Referenced" (page 77). To change the way the file is referenced, choose a style from this menu.
4. The File Type menu lets you explicitly set the file's type, overriding the actual file type of the file. How to change a file's type is described further in "Overriding a File's Type" (page 164).
5. The "Include in index" checkbox controls whether Xcode includes the file when it creates the project's symbolic index. See "Code Sense" (page 104) for more information.
6. The File Encoding menu specifies the character set used to save and display the file. File encodings are discussed further in "Choosing File Encodings" (page 163).
7. The Line Endings pop-up menu specifies the type of line ending used in the file. Line endings are discussed further in "Changing Line Endings" (page 163).

8. The next several options control tab and indent settings for the individual file. The Tab Width and Indent Width fields control the number of spaces that Xcode inserts when it indents your code or when you press the Tab key when you edit the file in the Xcode editor. To change either of these values, type directly in the field.

   The "Editor uses tabs" setting indicates whether pressing the Tab key inserts spaces or a tab when you are editing this file in Xcode's editor. Controlling indentation in the editor is discussed further in "Setting Tab and Indent Formats" (page 186).

   The Reset to Text Editing Defaults restores the line ending, file encoding, tab and indent settings to Xcode's built-in defaults.

9. The Make File Localizable and Add Localization buttons at the bottom of the pane let you customize files for different regions.

Choosing File Encodings

The file encoding of a file defines the character set that Xcode uses to display and save a file. If you type a character that isn't in the file's encoding, Xcode asks whether you want to change the encoding. Xcode uses the default single-byte string encoding, if possible (usually Mac OS Roman), or Unicode if the file contains double-byte characters.

To change the file encoding for one or more files, select those files and open an inspector window. In the General pane of the inspector window, choose the desired file encoding from the File Encoding menu. Generally Unicode (UTF-8) is best for source files and Unicode (UTF-16) is best for `.strings` files. You can also change the file encoding of an open file by choosing an item from the Format > File Encoding menu. When Xcode next saves the file, it uses the new file encoding.

To choose the default file encoding for new files, open the Text Editing pane in the Xcode Preferences window and choose an encoding from the Default File Encoding menu.

Note: GCC, the compiler used by Xcode for C, C++, and Objective-C, expects its source files to contain only ASCII characters, with the exception that comments and strings can contain any characters. Also, some encodings use escape sequences to handle non-ASCII characters, and those escape sequences can cause unexpected results when GCC interprets them as ASCII. For example, some characters in the Japanese (Shift JIS) encoding look like `*/` and will end your comment before you intended. Unicode (UTF-8) avoids this confusion.

Changing Line Endings

UNIX, Windows, and Mac OS use different characters to denote the end of a line in a text file. Xcode can open text files that use any of these line endings. By default, it preserves line endings when it saves text files. However, other utilities and editors may require that a text file use specific line-ending characters. You can change the type of line endings that Xcode uses for a single file, or you can change the default line ending style that Xcode uses for all new or existing files.

To change an individual file's line endings, select the file in the project window and open the inspector. In the General pane of the inspector window, use the Line Endings menu to choose Unix Line Endings (LF), Classic Mac Line Endings (CR), or Windows Line Endings (CRLF).

To choose the default line endings used for all new or existing files, choose Xcode > Preferences, click Text Editing, and choose Unix (LF), Mac (CR), or Windows (CRLF) from the Line Encodings pop-up menus:

- The "For new files" menu lets you choose the default line encoding that Xcode uses for all new files.
- The "For existing files" menu specifies the type of line encoding that Xcode uses when saving existing files that you have opened for editing in Xcode. To have Xcode preserve line endings for existing files, choose "Preserve" from this menu; this is the default value for the menu.

Generally, you don't need to worry about line endings. If you find that you must change line endings from the defaults assigned by Xcode, keep these guidelines in mind when deciding which line endings to use:

- Most Mac OS development applications, including CodeWarrior and BBEdit, can handle files that use UNIX, Mac OS, or Windows line endings.
- Many BSD command-line utilities, such as `grep` and `awk`, can handle only files with UNIX line endings.

Overriding a File's Type

By default, Xcode uses the type stored for a file on disk to determine how to handle that file. A file's type affects which editor Xcode opens the file in, how the file is processed when you build a target that includes the file, and how Xcode colors the file when syntax coloring is enabled. You can change the way that Xcode handles a file by overriding the file's type.

The File Type menu in the General pane of the file inspector lets you explicitly set the file's type, overriding the actual file type of the file. The File Type menu lists all of the file types that Xcode is aware of; to set a file's type, choose it from this menu. Choosing Default For File discards any explicit file type set for the file in Xcode and reverts to using the type stored for the file on disk.

For more information on how Xcode determines how to process files of a certain type, see "Build Rules" (page 261) and "Adding Files to a Build Phase" (page 254). For more information on how Xcode chooses the editor to use for files of a certain type, see "Overriding How a File is Displayed" (page 193).

Opening, Closing, and Saving Files

Xcode gives you a number of ways to get to, and open, files in your project. This chapter describes how to open files in your preferred editor—by default, Xcode's editor—and describes a number of shortcuts for opening files by name or from an open editor.

Opening and Closing Files

There are many ways to open files in Xcode. From an open project, you can open any of your project files by clicking or double-clicking the file. Files open in the preferred editor for the file's type. If this is Xcode's editor, you have the option of opening file in a separate editor window or in the editor attached to most Xcode windows.

There are also many shortcuts for opening files. The Open Quickly command lets you type a path or filename to open a file. From an editor, you can open a file by name, jump to the header associated with an implementation file and vice versa, or jump to files included by the current file.

Opening Project Files

If you already have a project window open, you can open a file by selecting it in the Groups & Files list in the project window. If you have the editor open inside of the project window, a single click on the file name will open the file in the editor. Otherwise, double-clicking the file in the Groups & Files list opens the file in a separate editor window.

If the file's name is in red, Xcode cannot find the file. Select the file, open the inspector, and click the Choose button next to Path in the General pane.

If you have a code editor open and you have previously opened the file, you can choose the file's name from the pop-up menu that lists recently viewed files, in the navigation bar at the top of the code editor.

Opening Header Files and Other Related Files

You can quickly open a header or source file that's related to the file displayed in the editor.

To open the related header for an implementation file open in the editor, and vice versa:

- Click the Go to Counterpart icon in the navigation bar of the code editor, as described in "The Navigation Bar" (page 174)
- Choose View > Switch to Header/Source File.

For example, if `main.c` is in the editor, this opens `main.h`; if `main.h` is in the editor, it opens `main.c`.

You can also view all the files that the current file includes, as well as all the files that include the current file. To view the list of files that the file in the editor includes and that include this file, click the Included Files icon in the navigation bar of the code editor. A menu pops up with the name of the current file in the center. Above it are the names of the files that this file includes. Below it are the names of the files that include this file. To open one of the files, choose it from the menu.

Opening Files by Name or Path

In addition to shortcuts to open related files, Xcode also provides a shortcut that allows you to open files by name or path, without having to navigate to the file using the Open File dialog. To open a file by name, use one of the following shortcuts. You can open files using these shortcuts even if the file is not in your project.

- To open a file whose name appears in a code editor, select the name and choose File > Open Quickly.
- To open a file by typing its name, choose File > Open Quickly and enter the filename in the Path field.

Xcode first looks for the file in the current project and then searches a list of directories that it maintains for use with the Open Quickly command. A number of common directories—such as `System/Library/Frameworks`—are already included in this list by default. You can add your own commonly accessed directories to this list in the Opening Quickly pane of Xcode Preferences, described in "Opening Quickly Preferences" (page 410).

Open Quickly searches the directories in the order in which they appear in this pane. If Open Quickly doesn't display the file you expected, check whether a file with the same name as the one you wish to open exists in your project or at a location higher up in the list of directories to search.

If you know the path to a file, you can also use the Open Quickly command to open the file, without adding a new directory to the list of search paths, by choosing File > Open Quickly and entering the path. For example, to open the file `MyNotes.rtf` located in your Documents folder you would type `~/Documents/MyNotes.rtf`.

Closing Files

Files in a project remain open until you explicitly close the file or close the project. Open files in an editor appear in the pop-up menu of recently viewed files. To close a file, choose File > Close File *filename*.

Saving Files

Xcode indicates which files you've modified by highlighting their icons in gray in the Groups & Files list, detail view, and in the pop-up menu of recently viewed files. You can save your changes in a number of ways:

- To save changes to the current file, choose File > Save.
- To save a copy of a file, choose File > Save As. Xcode saves a copy of the file under the name you specify. If the file is part of your project, Xcode also changes the file reference in your project to refer to the copy.
- To make a backup of a file, hold the Option key and choose File > Save a Copy As. Xcode saves a copy of the file under the name you specify, but does not modify the file reference in your project, if one exists.
- To save all open files, choose File > Save All.

Xcode can also be configured to automatically save all changed files before beginning a build. To specify whether files are saved automatically when you build a target:

1. Choose Xcode > Preferences and click Building.
2. In the "For Unsaved Files" menu, choose Ask Before Building, Always Save, Never Save, or Cancel Build.

If you don't have write permission for a file, Xcode warns you when you try to edit it. You can choose to edit such files, but you can save your changes only if you have write permission for the containing folder. In this case, you can choose whether Xcode changes the file's permissions to make it writable.

To have Xcode change the file's permissions, choose Xcode > Preferences, click Text Editing, and select "Save files as writable" in Save Options. Otherwise, Xcode preserves the file's current permissions.

The Xcode Editor

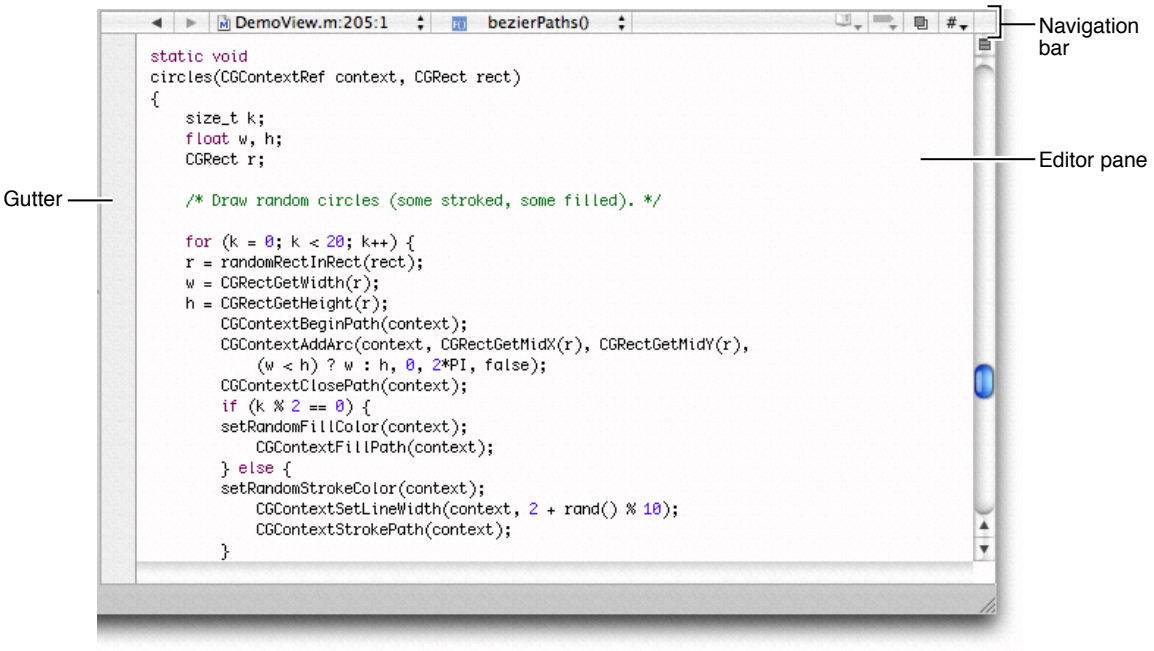
Xcode contains a full-featured editor for editing your project files. You have many options for using this editor to view and modify the files in your project; you can edit files in a dedicated editor window or use the editor attached to most Xcode windows. You can also choose whether to have multiple editor windows open at once, or use a single editor window for all files that you open.

This chapter describes Xcode's editor, shows how to open files in a standalone editor window or in an attached editor, and how to control the appearance of Xcode's editor.

The Xcode Editor Interface

When you edit a file in Xcode, you have the choice of using a standalone editor window or editing the file directly in any of the other Xcode windows, such as the project window, debugger window, build results window, and so forth. Regardless of your choice, Xcode uses a common interface for the editor. When you open a file in Xcode's editor, you see a view similar to that in Figure 15-1.

Figure 15-1 The Xcode Editor



Here's what the Xcode editor contains:

1. Gutter. The gutter displays file line numbers, as well as information about the location of breakpoints, errors, or warnings in the file. See "Displaying the Editor Gutter" (page 180) to learn more about the contents of the gutter, as well as how to show and hide the gutter.
2. Editor. The text editing pane displays the contents of the file.
3. Navigation bar. The bar along the top of the editor contains several menus and buttons that let you quickly see, and jump to, locations within the current file and in other files open in the editor. "The Navigation Bar" (page 174) describes the contents of the navigation bar and how to use it to navigate source code files.

Note: You can also modify the attributes of the file reference associated with a file open in the editor, as described in "Inspecting File, Folder, and Framework References" (page 161).

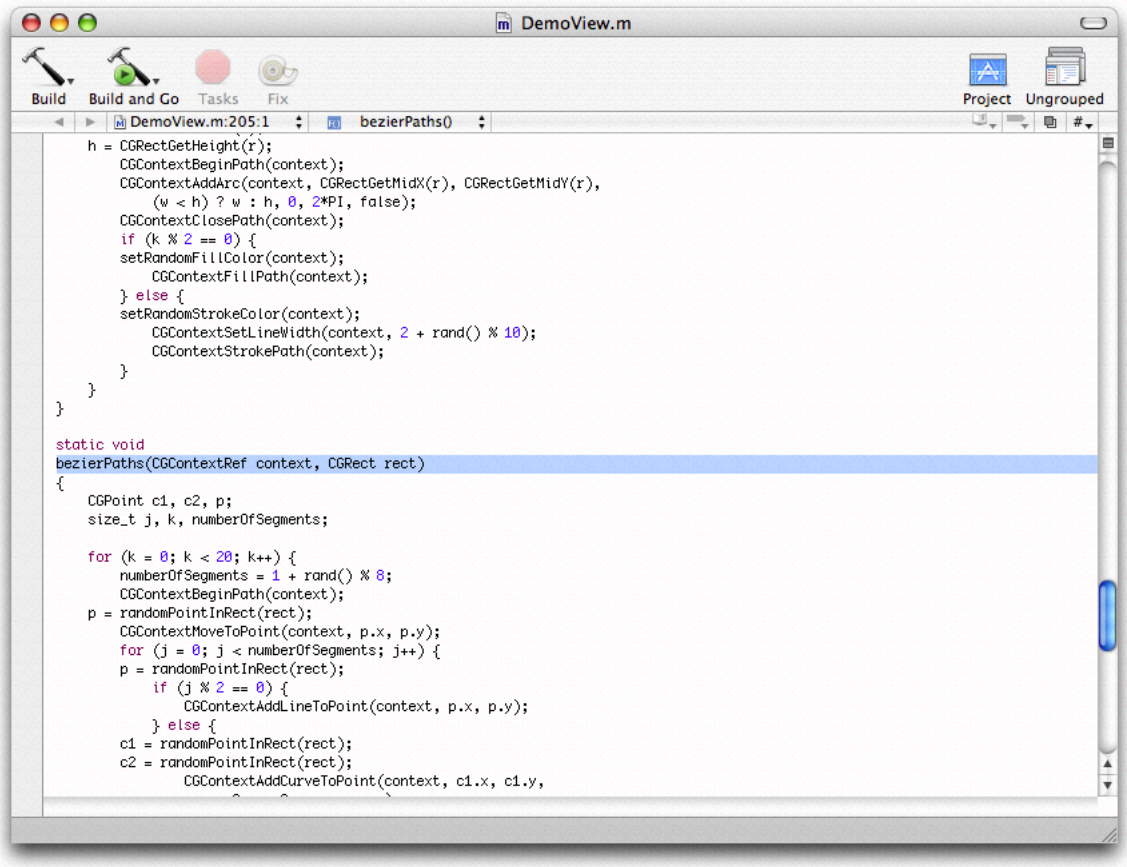
Editing Files in a Separate Editor Window

If you prefer, you can use a dedicated window for editing source files in Xcode. Regardless of your preference for whether Xcode automatically opens the attached editor in Xcode windows, you can always open a file in a separate editor by doing any of the following:

- Double-click the file in the Groups & Files list or the detail view in the project window.
- Select the file and choose View > Open in Separate Editor.
- Control-click the file and choose Open in Separate Editor from the contextual menu.

Figure 15-2 shows the Xcode editor in a separate window.

Figure 15-2 The Xcode editor in a standalone window



In addition to the basic editor interface, the standalone editor window also contains a toolbar and a status bar. The status bar is similar to the status bar of other Xcode windows, described in "The Project Window Status Bar" (page 64). Like the toolbar in other Xcode windows, the editor window toolbar provides easy access to common tasks. By default, it includes buttons to build, run, and debug the current target. It also contains the following two buttons:

- The Project button lets you quickly jump to the file in the project window. Clicking this button brings the project window to the front.
- The Grouped and Ungrouped buttons control whether opening a file, using any of the methods described above, opens a new standalone editor window for that file or opens the file in the current window. Clicking the button toggles the state. If the button is Grouped, indicated by the icon of a single window, double-clicking a file opens it in the current editor. If the button is Ungrouped, indicated by an icon of multiple layered windows, each file opens in a new editor window.

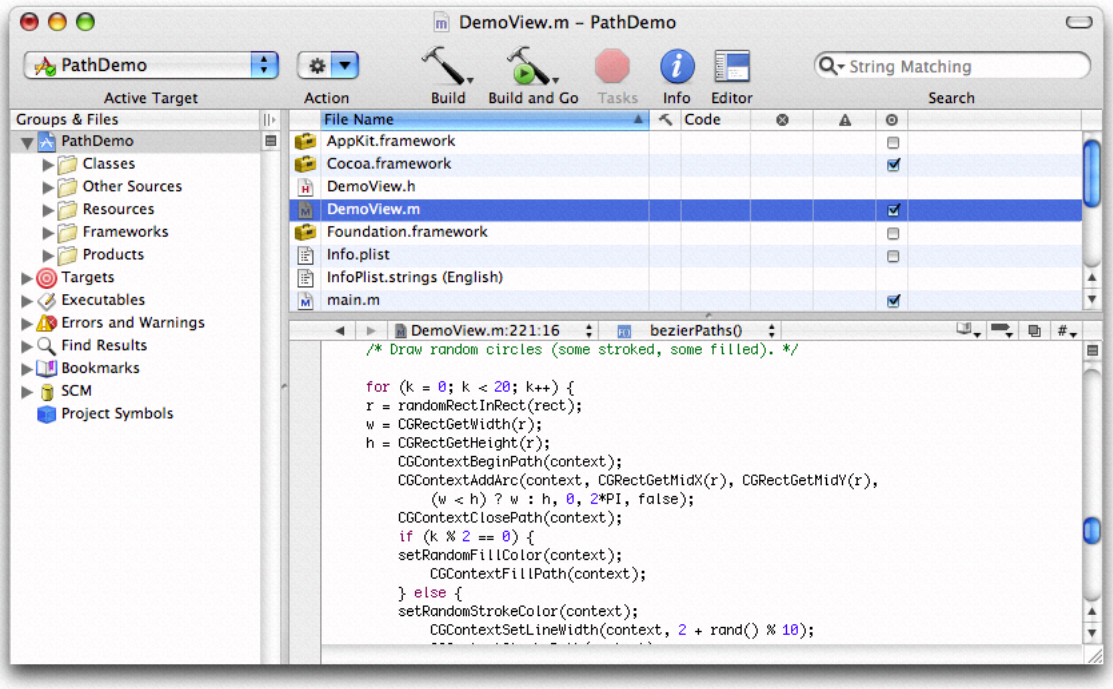
To preserve the state of any open editor windows when you close a project, choose Xcode > Preferences, click General, and select "Save window state" in the Environment options.

Using the Attached Editor

You can also edit your source files from within other Xcode windows, such as the project window and the debugger window. To open a file in the attached editor, make sure that the editor is visible in the window. If the editor is not already visible, you can open it by clicking the Editor button or choosing View > Zoom Editor In. This opens the attached editor to its maximum size. If the editor was already at its maximum size, clicking the Editor button or choosing View > Zoom Editor Out returns the attached editor pane to its previous size. To adjust the size of the attached editor to a different size, drag the separator to the size that you prefer.

Selecting a file, an error or warning, a bookmark, a find result or a project symbol opens the associated file in the editor as long as the editor is visible. You can also have Xcode automatically show the attached editor when you select one of these items in the detail view. To specify that Xcode automatically open the attached editor, select "Automatically open/close attached editor" in the Editing options in the General pane of the Xcode Preferences window.

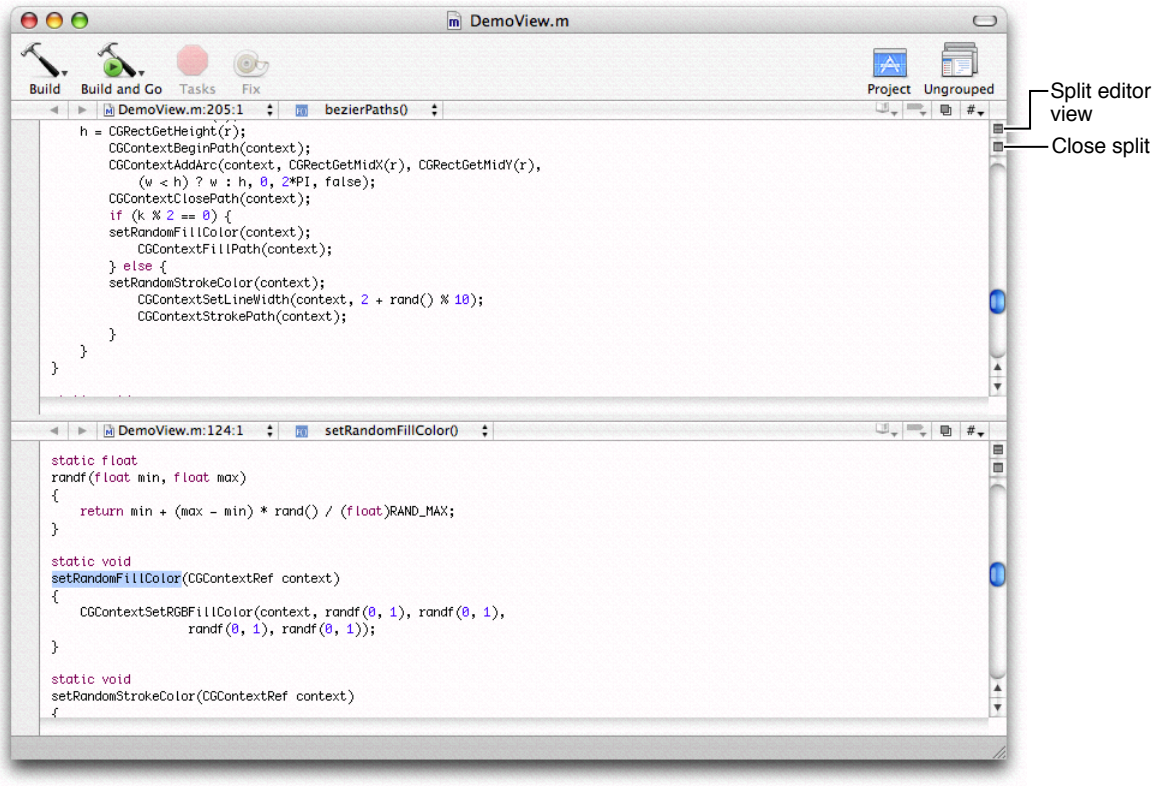
Figure 15-3 Editor in a project window



Splitting Code Editors

Xcode allows you to simultaneously view multiple files or multiple sections of the same file without increasing the number of open windows. It does this by splitting a code editor. The figure seen here shows an editor that has been split to display two parts of the same file.

Figure 15-4 Splitting a code editor



Note that you can split an editor whether that editor appears in a separate window or as an attached editor.

To split a code editor, make sure that the editor has focus and do one of the following:

- To split the editor vertically, choose View > Split *filename* Vertically, or click the split button. The split button—identical to the split button in the Groups & Files list, described in "Splitting the Groups & Files View" (page 59)—appears above the scrollbar of the editor window.
- To split a code editor horizontally, hold down the Option key and choose View > Split *filename* Horizontally, or Option-click the split button.

To close a split, choose View > Close Split View, or press the Close Split button. You can resize the panes of a split editor by dragging the resize control between them.

Navigating Source Code Files

Xcode provides many ways to find and navigate to information in a file and move between files in an editor. The navigation bar of the editor provides a number of menus that let you jump to related header or source files, move between open files, and jump to bookmarks, breakpoints, or other locations in the current file. Xcode's single file find lets you search the contents of a file in the editor. Xcode also provides a number of shortcuts for opening files, jumping to symbol definitions or declarations, or finding documentation, all from the editor.

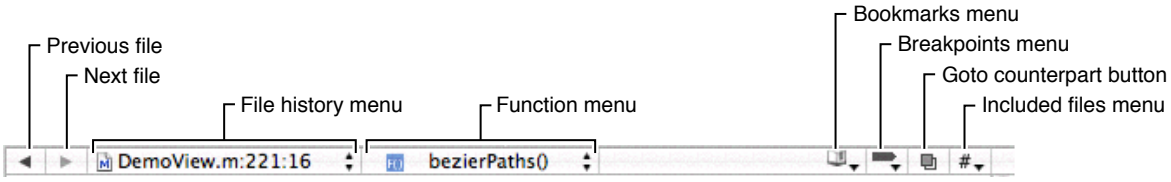
This section describes the contents of the navigation bar and shows you how to search within a file. It also shows some of the shortcuts you can use in Xcode's editor to find text and symbol definitions.

Some tricks for finding information from an editor are described in other chapters. "Opening Files by Name or Path" (page 166) describes shortcuts you can use to open a file whose name or path appears in an editor. "Searching for Documentation" (page 114) describes shortcuts you can use to jump to the documentation for a symbol whose name appears in the editor, or search the installed documentation for a word or phrase.

The Navigation Bar

The navigation bar contains a number of controls that you can use to move between open files, jump to symbols, and open related files. Figure 15-5 shows the navigation bar.

Figure 15-5 The navigation bar in the editor



Here is what the navigation bar contains:

1. The Previous and Next arrows move between open files in the editor.
2. The File History pop-up menu lists recently viewed files. Selecting a file from this menu displays that file in the editor, without having to repeatedly click Next or Previous.
3. The Function pop-up menu lists the function and method definitions in the current file. When you select a definition from this menu, the editor scrolls to the location of that definition. For information on how to configure the Function pop-up menu, see "The Function Pop-up Menu" (page 175).
4. The Bookmarks pop-up menu lists any bookmarked locations in the current file. When you select a bookmark from this menu, the editor scrolls to the location of the bookmark. See "Saving Commonly Accessed Locations as Bookmarks" (page 91) to learn more about bookmarks in your project.
5. The Breakpoints pop-up menu lists any breakpoints in the current file. Choosing a breakpoint from this menu scrolls the editor to the location at which the breakpoint is set. See "Breakpoints" (page 353) to learn more about breakpoints in Xcode.
6. The Go To Counterpart button opens the counterpart of the current file or jumps to the symbolic counterpart of the currently selected symbol. See "Jumping to a File's or Symbol's Counterpart" (page 176) for more information on the Go To Counterpart button.
7. The Included Files pop-up menu lists all of the files included by the file that is currently being edited, as well as all of the files that include the current file. Selecting a file from this list opens that file in the editor window. This menu is described more in "Opening Header Files and Other Related Files" (page 165).

The File History Menu

The File History pop-up menu lists all of the files that you have viewed in the current editor, with the current file at the top of the menu. To return to any of the these files, simply select it from the menu.

You can clear the File History menu by choosing Clear File History. This removes all but the file currently open in the editor from the editor's history list. By default, Xcode does not place a limit on the number of files that it remembers in this menu. However, you can limit the size of the file history menu by choosing the number of files you want remembered from the History Capacity menu. For example, if you choose 5 from this menu, Xcode only remembers the five most recently viewed files in the File History menu.

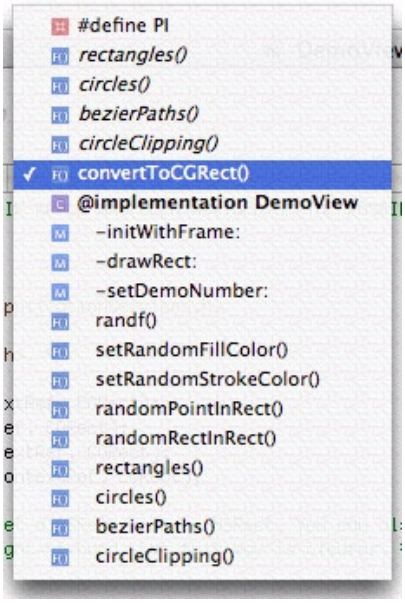
The Function Pop-up Menu

The function pop-up menu lets you jump to many points in your file, including any identifier it declares or defines. You can also add items that aren't definitions or declarations. The function pop-up menu is in the navigation bar, next to the File History menu. In this menu, you can see:

- Declarations and definitions for classes, functions, and methods
- `#define` directives
- Type declarations
- `#pragma` marks

To scroll to the location of any of these identifiers, select it from the menu. Figure 15-6 shows the function pop-up menu.

Figure 15-6 The function pop-up menu



By default, the contents of the function pop-up menu are sorted in the order in which they appear in the file. You can hold down the Option key while clicking the function pop-up menu to toggle the sort order of the items in the menu between alphabetical and the order in which they appear in the source-file.

You can also change the default behavior for the function pop-up menu. To choose which items appear in the function pop-up menu and the order they appear in, choose Xcode > Preferences, click Code Sense, and use the Editor Function Pop-up options:

- Select "Show declarations" to include declarations in the function pop-up menu. Otherwise, the function pop-up menu shows only definitions.
- Select "Sort list alphabetically" to have Xcode display the contents of the function pop-up in alphabetical order. Otherwise, the contents are sorted in the order in which they appear in the file.

Note: Syntax coloring must be enabled for the function pop-up menu to be available in the editor window.

To add a marker to a C, C++, or Objective-C source file and make that marker appear in the function pop-up menu, use the `#pragma mark` statement in your source code. For example, the following statement adds "PRINTING FUNCTIONS" to the function pop-up menu:

```
#pragma mark PRINTING FUNCTIONS
```

To add a separator to the function pop-up, use:

```
#pragma mark -
```

Jumping to a File's or Symbol's Counterpart

Clicking the Go To Counterpart button opens the related header or source file for the file currently open in the editor. For example, if the file currently open in the editor is `MyFile.c`, clicking this button opens `MyFile.h`, and vice versa. When your project contains files with the same name, Xcode gives preference to files located in the same folder as their counterparts. You can also open the current file's related header or implementation file by choosing View > Show Header/Source File.

Option-clicking the Go To Counterpart button displays the counterpart of the currently selected symbol—class, method, function, and so on—opening the corresponding file and scrolling to the appropriate section within it if necessary. If the selected symbol is a class, method, or function declaration, Xcode jumps to the definition for that item. If a class, function, or method definition is currently selected, Xcode jumps to the symbol's declaration. You can specify a key binding for the Switch to Symbolic Counterpart action in the Key Bindings pane of Xcode Preferences. For information on configuring key bindings for actions, see "Customizing Key Equivalents" (page 385).

By default, Xcode opens the file or symbol counterpart in the same editor; however, you can have Xcode open counterparts in a separate editor window. This makes it easy to view both a header and its implementation file, or a symbol declaration and its definition, at once. To have Xcode open counterparts in a separate window, open the General pane of Xcode Preferences and select "Open counterparts in same editor" in the Editing options.

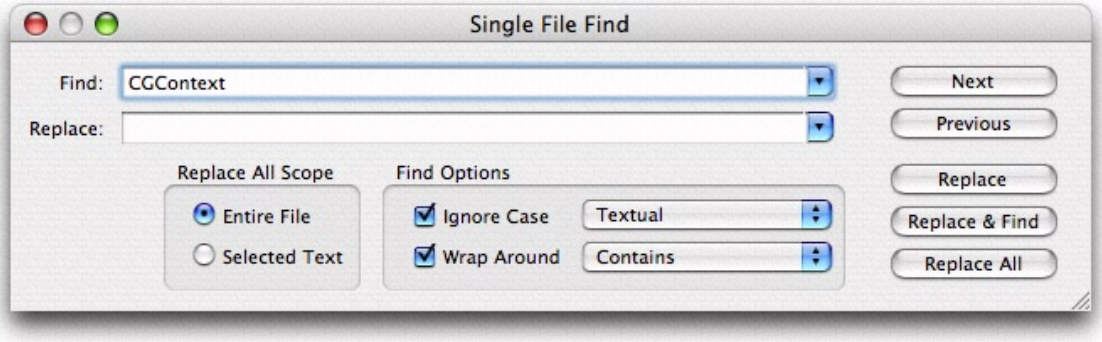
Searching in a Single File

When editing files in the code editor, it is fairly common to find that you need to make the same change in several places in the file. For example, when you rename a function, you also have to find all of the places where you call that function and change those calls to use the new name. Xcode provides a single-file find that allows you to search for and replace text within a single file. You can perform a simple textual search or search using regular expressions.

Specifying Search Terms

To search for text in a file that you have open in an Xcode editor, choose Find > Single File Find or press Command-F. Xcode displays the Single File Find window, shown here.

Figure 15-7 The Single File Find window



You can search using a text string or a regular expression; choose the appropriate search type from the pop-up menu next to the Ignore Case option in the Find Options group. Choosing "Textual" searches for text matching the string in the Find field. Choosing "Regular Expression" searches for text matching the regular expression in the Find field.

Type the text string or regular expression pattern to use for the search in the Find field of the Single File Find window. Xcode keeps track of search strings; to reuse a previous search string, click the arrow in the Find field and choose the string from the menu.

The other options in the Find Options group give you additional control over how the search is performed; these options are:

- Ignore Case. Use this option to ignore whether letters are uppercase or lowercase.
- Wrap around. Select this option to search the whole file; otherwise, Xcode searches from the current location of the insertion point to the end of the file.
- The pop-up menu next to the Wrap Around option specifies how Xcode determines a match to the search term in the Find field. Choose "Contains" to search for words that contain matching text in a substring, "Starts with" to search for words that begin with text matching the search term, "Whole words" to search for words that contain only text matching the contents of the Find field, or "Ends with" to search for words that end with matching text.

Use the Next and Previous buttons to continue searching for the same text in a file. Alternatively, you can choose Find > Find Next or hold down the Shift key and choose Find > Find Previous. Pressing Return finds the next (or first) match and dismisses the Single File Find window.

Replacing Text

You can use the Single File Find window to replace some or all occurrences of text matching the string or regular expression specified in the Find field. To search for and replace text in a file:

1. Open the Single File Find window and specify the search criteria, as described in the previous section.

2. Type the replacement text in the Replace field. As with search strings, Xcode keeps track of substitution strings; to reuse a previous substitution string, choose it from the menu in the Replace field.

Use the replace buttons in the bottom-right portion of the Single File Find window to perform the text substitution. The scope of the replacement varies, depending on the button you choose. Here are the buttons available to you:

- Replace substitutes the replacement text for the current selection.
- Replace & Find substitutes the replacement text for the current selection and then finds and selects the next occurrence of text matching the contents of the Find field.
- Replace All searches the entire file or selection and replaces all occurrences of text matching the contents of the Find field with the replacement text.

  If you choose Replace All, the Replace All Scope radio buttons control the scope of the search and replace operation. To search for and replace instances of the given search text throughout the entire file, select the Entire File option. To perform the search and replace operation within only the current selection, choose Selected Text.

Each of these buttons also has a menu item equivalent in the Find menu. To replace the current selection or to replace the current selection and find the next match, choose Find > Replace or Find > Replace and Find Next, respectively. To replace all occurrences of the search text, hold the Option key and choose Find > Replace All.

You can also replace the current selection and find the previous match in the file. To do so, hold the Shift key and either choose Find > Replace and Find Previous or click the Replace & Find button.

Shortcuts for Finding Text and Symbol Definitions From an Editor Window

Xcode provides a number of shortcuts for searching using text or other content that appears in an editor window. You can use these shortcuts to perform single-file and project-wide searches without going through the Single File Find or Project Find windows. When you use these shortcuts, Xcode performs the search using the same options specified the last time you used the Single File Find or Project Find windows. These windows are described in further detail in "Searching in a Single File" (page 176) and "Searching in a Project" (page 97).

To search a single file for text that appears in an editor window, select the text to search for, and choose Find > Find Selected Text.

To perform a project-wide search using the current selection in an editor window, use the shortcuts listed in Table 15-1.

Table 15-1 Shortcuts for performing a project-wide search using the current selection in the editor

Search project for	Choose
Selected text	Find > Find Selected Text in Project
Selected regular expression	Hold Option key and choose Find > Find Selected RegEx in Project

Search project for	Choose
Selected symbol definition	Find > Find Selected Definition in Project

You can also jump directly to the definition for a symbol identifier by doing either of the following:

- Command—double-click the symbol name.
- Select the symbol name and choose Find > Jump to Definition. The Jump to Definition menu item does not have a keyboard shortcut by default, but you can assign a keyboard shortcut to it in the Key Bindings pane of the Xcode Preferences window.

Each of the searches described in Table 15-1 (page 178) uses the last set of search options used when searching your project. If you want to perform a project-wide search using the current selection in an editor window, but do not want to use the last set of search options, you can open a Project Find window with the current selection by doing either of the following:

- To use the current selection as a search term, choose Find > Use Selection For Find. Xcode opens a Project Find window and places the contents of the current selection in the Find field.
- To use the current selection as a substitution string, choose Find > Use Selection for Replace. Xcode opens the Project Find window and places the contents of the current selection in the Replace field.

Controlling the Appearance of the Code Editor

Xcode gives you a great deal of flexibility to customize the appearance of the editor. You can change the fonts and colors used to display text in the editor to suit your own preferences. You can also control the amount of information that Xcode displays about file locations and contents. This section describes how to change the default font and text editing colors for Xcode editors, and how to use the gutter, page guide, and file history menu to locate information in a file.

Setting Default Fonts and Colors

You can change the font and colors used for text editing in Xcode in the Xcode Preferences window. Choose Xcode > Preferences and select Fonts & Colors. The Editor Font section displays the font used for text in the editor; to change this font, click the Set Font button.

Note: This font is also used for all text, regardless of its role, when syntax coloring is enabled, unless you specify otherwise. See "Setting Syntax Coloring" (page 183) for more information.

To change the colors used in the editor, use the Editor Colors options. To change the color of an item, click its color well and choose a new color. You can change the default color for the following elements:

- Text. This option controls the default color used for text in an editor. You can specify additional colors for text that represents particular code elements, such as strings. For a description of how to use syntax coloring in Xcode, see "Setting Syntax Coloring" (page 183).
- Background. This option specifies the background color used for editor windows.

- Selection. This option specifies the highlight color used to indicate selected text.
- Insertion Point. This option controls the color of the blinking insertion point character in an editor window.

Displaying a Page Guide

To help keep code lines no longer than a specified length, you can have Xcode display a guide line in every code editor at that column position in the file. To display a guide line, open the Text Editing pane of Xcode Preferences and select "Show page guide" in Display Options. Enter the location, in number of characters, at which you want the guide line displayed in the text field titled "Display at column." Xcode displays a gray line in the right margin of all open editors, at the specified column.

Xcode does not wrap your code lines when they reach the guide line. The line serves only as a guide.

Displaying the Editor Gutter

The gutter that appears on the left side of a code editor helps you quickly locate items in a file. This gutter can display:

- Line numbers for the current file. Line numbers make it easy to find a location in a file. Xcode does not show line numbers by default; to change this, open the Text Editing pane of Xcode Preferences and select "Show line numbers" in Display Options.
- Errors and warnings. To help you locate and fix problems in your code, Xcode displays error and warning icons next to the line at which an error or warning occurred. Clicking on the icon or pausing with the mouse over the icon displays the error or warning message.
- Breakpoints. You can use the gutter to set, remove, and otherwise control the breakpoints in a file. Xcode indicates the location of a breakpoint by displaying an arrow next to the line at which the breakpoint is set. For more information on using breakpoints, see "Breakpoints" (page 353).

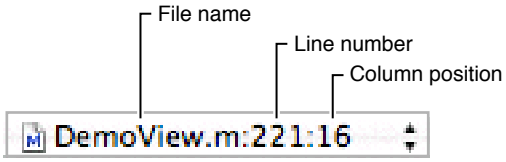
You can control the visibility of the gutter in a single editor or set the default behavior for all editors. To change the visibility of the gutter for all editors that you open, standalone and attached, open the Text Editing pane of Xcode Preferences and use the "Show gutter" option in Display Options. If this option is selected, as it is by default, the gutter is visible in all editors that you open. Otherwise, the gutter appears in all code editors only when you start debugging.

To show or hide the gutter in a particular instance of the editor, bring the window to the front and use the View > Hide Gutter and View > Show Gutter commands.

Viewing Column and Line Positions

As you've seen in "The Navigation Bar" (page 174), the File History pop-up menu in the navigation bar not only lets you move between currently open files, it also shows you your current location in the file. For the file currently open in the editor, the File History menu shows the name and the line number of the line containing the insertion point. You can also have the File History menu display the column position of the insertion point; that is, the offset of the insertion point, in number of characters, from the left margin of the editor. Figure 15-8 shows the location of the current insertion point in the File History menu.

Figure 15-8 Line and column positions in the File History pop-up menu



By default, Xcode does not display the column position of the insertion point. To change this, open the Text Editing pane of Xcode Preferences and select "Show column position."

Formatting and Syntax Coloring

Xcode provides a number of formatting and coloring options to help you keep your code well formed and readable. Syntax coloring makes it easy to understand the structure of your code by using different fonts or colors to identify different code elements, such as comments. With syntax-aware indenting, Xcode makes it simple to keep your code well-formed and neat by automatically indenting code and formatting it as appropriate for the current context. This chapter describes options for indenting code, matching parentheses, and using syntax coloring.

Setting Syntax Coloring

Xcode uses colors and fonts to distinguish among different types of code elements in a source code file. For example, you can display comments in green and keywords in boldface. Xcode maintains syntax coloring rules that specify a color and font for each code element you can customize. These rules apply to all languages for which Xcode supports syntax coloring.

Xcode supports syntax coloring for many different programming languages; to see the languages that it supports, use the Format > Syntax Coloring menu. Xcode uses a file's type to determine how to interpret and color the contents of the file.

By default, syntax-coloring is enabled for all files that you open in Xcode's editor. You can disable syntax-coloring, or customize the syntax-coloring rules, in the Fonts & Colors pane of Xcode Preferences. For more information on the Fonts & Colors preference pane, see "Fonts & Colors Preferences" (page 405). You can also enable or disable syntax-coloring for individual files open in an editor.

Controlling Syntax Coloring and Syntax Coloring Rules

To turn syntax coloring on and off for all files that you open, choose Xcode > Preferences, click Fonts & Colors, and use the Syntax Coloring option.

Note: Syntax coloring must be enabled for the function pop-up menu to be available in the editor window.

To set the color for a particular type of code element, select the code element from the Syntax Coloring pop-up menu and change its color and font. For example, to change the color used for strings, select Strings from this pop-up menu and click the color well to bring up the color palette. You can customize syntax coloring rules for the code elements listed in Table 16-1.

Table 16-1 Syntax coloring rules

Syntax Coloring Rule	Specifies font and color for
Comments	Comments in source code, denoted by // or enclosed between '/*' and '*/' character pairs.
Documentation Comments	Text of documentation using HeaderDoc or JavaDoc style markup.
Documentation Comment Keywords	Keywords used to identify documentation using HeaderDoc or JavaDoc style markup.
Strings	String constants in source code.
Keywords	Keywords in source code.
Characters	Character constants in source code.
Numbers	Numeric constants in source code.
Preprocessor	Preprocessor directives.

By default, all syntax-coloring rules use the same font—the font specified by the Editor Font option, described in "Setting Default Fonts and Colors" (page 179)—but different colors. You can, however, have Xcode change both font and color based on the code element. To use other fonts for other types of code elements, choose Xcode > Preferences, click Fonts & Colors, and select the "Allow separate fonts" option. Unless this option is enabled, Xcode does not allow you to change the font for an individual syntax-coloring rule.

To change the font used for a particular code element, choose that code element from the Syntax Coloring pop-up menu and click Set Font to open the Fonts window.

Xcode uses the syntax coloring rules in the Fonts & Colors pane to determine how to display files in its editor. You can, however, have Xcode preserve syntax coloring when copying and pasting, or when printing text from a code editor. These options are available in the Fonts & Colors pane of Xcode Preferences. To have syntax coloring appear when you print a file, select the "Use colors when printing" option. Otherwise, if you have specified different fonts for any of the code elements, Xcode uses those when printing, but uses only a single color.

To choose whether to preserve colors and fonts when copying code from an editor, use the "Copy colors and fonts" option. When this option is enabled, as it is by default, Xcode preserves both font and color information when it copies text to the clipboard.

Controlling Syntax Coloring for a Single File

You can control syntax coloring for individual files using the Format > Syntax Coloring menu. This menu lets you turn syntax coloring on or off for a file in an editor window and set the type of syntax coloring used for that file. By default, Xcode uses the file type of the current file to determine how to color the file's contents. However, you can specify that Xcode use syntax coloring appropriate for a particular language by choosing that language from the Format > Syntax Coloring menu.

To turn syntax coloring off for a file, choose Format > Syntax Coloring > None. To turn syntax coloring back on, using the type of the file to determine the appropriate syntax coloring, choose Format > Syntax Coloring > Default for file type.

Wrapping Lines

To keep all of your code visible in the editor, you can have Xcode wrap lines when they reach the right edge of a code editor. To turn on line wrapping for all files you open in Xcode, choose Xcode > Preferences, click Indentation, and select "Wrap lines in editor." Otherwise, Xcode does not move text to the next line until you insert a carriage return. To wrap lines for an individual file in an editor window, choose Format > Wrap Lines.

You can also have Xcode automatically indent wrapped lines, to visually distinguish them from other lines. Enter the number of spaces to indent lines by in the "Indent wrapped lines by" field.

Indenting Code

Xcode's editor supports syntax-aware indenting to make it simple to author neat and readable code. When you use syntax-aware indenting, Xcode automatically indents and formats your code as you type; pressing Return or Tab moves the insertion point to the appropriate level by examining the syntax of the surrounding lines. You can also choose to indent code manually.

This section shows you how to configure syntax-aware indenting, how to manually format text in the editor, and how to control the format of tabs and automatic indentation.

Syntax-Aware Indenting

Xcode gives you a number of ways to control how it automatically formats your code. You can control which characters cause Xcode to indent a line, what happens when you press the Tab key, and how Xcode indents braces and comments.

Syntax-aware indenting is not enabled by default; to turn it on, choose Xcode > Preferences, click Indentation, and select the "Syntax-aware indenting" option. For more information on the options available in the Indentation preferences pane, see "Indentation Preferences" (page 406).

Choosing What the Tab Key Does

When you use syntax-aware indenting, you usually press the Tab key to tell the editor to indent the text on the current line. But when you're at the end of the line, you may want to insert a tab character before, say, you insert a comment. To choose the circumstances when pressing the Tab key reindents a line, open the Indentation pane of Xcode Preferences and use the "Tab indents" menu in the syntax-aware indenting options. You can choose the following options:

- "In leading white space" indents only when the insertion point is at the beginning of a line or in the white space at the beginning of a line.
- "Always" indents when the insertion point is anywhere in the line.
- "Never" never indents the line.

To insert a tab character regardless of this option's setting, press Option-Tab. Similarly, to perform syntax-aware indenting, regardless of this option's setting, press Control-I.

Choosing How to Indent Braces

You can have Xcode automatically indent braces to help you easily see the level of nesting in your code and keep your code readable. In addition, to help you keep braces balanced, you can opt to have Xcode automatically insert a closing brace when you type an opening brace.

To choose how much an opening brace is indented when it appears on a line by itself, choose Xcode > Preferences, click Indentation, and use the "Indent solo '{' by:" field. If this field is greater than 0, Xcode automatically indents opening braces to the level of the previous line plus the specified number of characters. By default, the value of this field is 0.

To choose whether to insert a closing brace automatically when you type an opening brace, choose Xcode > Preferences, click Indentation, and use the "Automatically insert closing '}'" option.

Choosing Which Characters Reindent a Line

To choose which characters cause Xcode to a automatically indent a line whenever they're typed, choose Xcode > Preferences, click Indentation, and use the "Automatically indented characters" options.

Choosing How to Indent C++-Style Comments

You can choose how to indent C++-style (//) comments when they appear on lines by themselves. You cannot automatically indent C++-style comments that appear at the end of code lines.

To automatically indent C++-style comments that appear on lines by themselves, choose Xcode > Preferences, click Indentation, and use the "Indent // comments" option.

To align consecutive C++-style comments that appear on lines by themselves, choose Xcode > Preferences, click Indentation, and use the "Align consecutive // comments" option.

Both these options are on by default when syntax-aware indenting is enabled.

Indenting Code Manually

If you choose not to use syntax-aware indenting, you must do any indentation and formatting manually. When syntax-aware indenting is disabled, pressing Tab inserts a tab and pressing Return inserts a carriage return and moves the cursor to the same level as the previous line. You can also indent a block of text to the left or right by selecting the text and choosing Format > Shift Left or Format > Shift Right.

When syntax-aware indenting is turned off, Xcode may still indent newly added lines to the level of the previous line when you press Return. To turn this off, add the Return key to the key-equivalents list of the Insert Newline action in Key Bindings preferences. For information on configuring key bindings for actions, see "Customizing Key Equivalents" (page 385).

Setting Tab and Indent Formats

Whether you indent a line manually, or rely on Xcode's syntax-aware indenting, you can control the width of tabs and indents, as well as whether Xcode inserts Tab characters or spaces. You can specify default values for all files you open in Xcode, as well as customizing these settings for individual files.

Changing the Indent and Tab Width

You can set how many spaces to indent when the editor automatically indents or when you press the Tab key. To set the default indent or tab width for every file you open, open the Indentation pane of Xcode Preferences and change the "Tab width" or "Indent width" setting.

To override the default indent or tab width for one or more specific files, select the files in the Groups & Files list and open the inspector window. In the General pane, change the Indent Width or Tab Width setting.

If you change a file's default indent or tab width, those settings are in effect for everyone who views that file.

Using Spaces Instead of Tabs

The editor can insert a series of spaces instead of a tab whenever it indents code or you press Tab. This ensures that your code looks the same on other computers no matter how wide their tabs are set. However, this means that changing the width of tabs won't affect code you've already written.

To specify that the editor uses spaces instead of tabs, choose Xcode > Preferences, click Text Editing, and select "Insert 'tabs' instead of spaces" option. These options are saved in your own preferences but not in the file itself. When other people edit the file, their preferences for that file take effect.

You can also specify this setting on a per-file basis. To choose whether the editor uses tabs or spaces when editing a certain file, select the file in the Groups & Files list, open the inspector window, and select "Editor uses tabs."

Matching Parentheses, Braces, and Brackets

Xcode provides a number of ways to help you match pairs of delimiters (parentheses, braces, and brackets). Xcode assists you in the following ways:

- When you type a closing delimiter, Xcode causes its counterpart to blink.
- When syntax-aware indenting is enabled, Xcode can automatically insert a closing brace each time you type and opening brace, as described in "Choosing How to Indent Braces" (page 186).
- When you double-click any delimiter, Xcode selects the entire expression that it and its counterpart enclose. You can also choose to select the delimiters themselves.
- You can use the Format > Balance command to select the text surrounding the insertion point, up to the nearest set of enclosing delimiters.

You can further control Xcode's behavior when selecting text within a pair of enclosing delimiters in the Text Editing pane of Xcode Preferences. Use the following Editing Options:

- Select to matching brace. When this option is enabled, double-clicking a delimiter automatically selects the enclosed expression, including the delimiters themselves. This option is enabled by default when you turn on syntax-aware indenting.
- Omit braces in selection. When this option is enabled, double-clicking a delimiter selects the enclosed expression, but does not include the delimiters themselves in the selection. This option is disabled by default.

Code Completion

Xcode includes a feature, called Code Sense, which maintains a rich store of information about the symbols defined in your project and its included files. Xcode uses this feature as a basis for code completion. Xcode supports code completion for C, C++, Objective-C, Objective-C++, Java, and AppleScript.

When you are writing code, you often must type out or copy and paste long identifier names and lists of arguments. Code completion offers you a shortcut. As you type the beginning of an identifier or a keyword, Xcode suggests likely matches, based on the text you have already typed and the surrounding context of the file. This chapter describes how to use code completion and how to set code completion options. It also describes text macros, a feature that provides shortcuts for inserting common code constructs, using the code completion mechanism.

Note: Code completion relies on the symbolic index created for your project. If you disable indexing, code completion does not work.

Using Code Completion

As you type, Xcode builds a list of likely symbol names, based on the text you have already typed and the surrounding context of the file. If completion suggestions are available, Xcode indicates this by underlining the text that you have entered. You can view and select from the possible matches by:

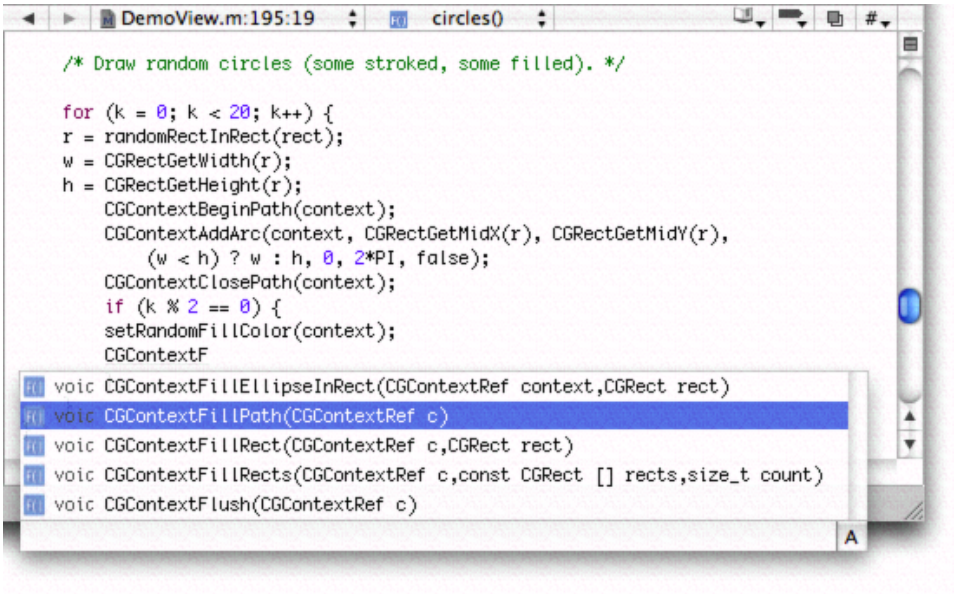
1. Bringing up the completion list and choosing a symbol name. Pressing the Escape key or choosing Edit > Completion List brings up a list of all of the possible matches for the current context. For functions and methods, this includes the return type as well as the function or method prototype. The button in the bottom-right corner of the completion list lets you toggle the sort order of the completion suggestions. By default, the list is sorted alphabetically, indicated by the letter "A" in this button. You can also choose to sort the list according to relevance ranking; click the button to change the sort order.

   You can choose the appropriate match from this list or continue typing to narrow the list further. To enter a symbol from the completion list, select it and hit Return or Tab. Xcode enters the remaining text for you and optionally inserts placeholders for any necessary function or method arguments.

   The completion list stays open until you select a completion suggestion or dismiss the list by typing the completion key or choosing Edit > Completion List again.

2. Cycling through the available completions. Instead of bringing up a list of all completions, can you choose to have Xcode insert the next available completion directly in your code. When you choose Edit > Next Completion or type Control-period, Xcode inserts and selects the first completion in the completion list, without displaying the entire list. Choosing Edit > Next Completion or typing Control-period again replaces the previous completion with the next completion in the list, and so forth. In this way, you can cycle through the available suggestions until you get the correct symbol name.

Figure 17-1 Using code completion



Xcode uses information about symbols and their scope within your code to determine the contents of the completion list. When there are syntax errors in the context, Xcode limits its scope analysis to the current code line.

Xcode also provides keyboard shortcuts for common code completion tasks. For example, to bring up the completion list, you can type Escape. To move to the next argument in a declaration that Xcode has completed for you, type Control–slash. To change the keyboard shortcuts associated with these commands, open the Key Bindings pane of Xcode Preferences, as described in "Customizing Key Equivalents" (page 385). In the Text Key Bindings pane, change the key sequences associated with the Code Sense actions. These are Code Sense Complete List, Code Sense Next Completion, Code Sense Previous Completion, Code Sense Select Next Placeholder and Code Sense Select Previous Placeholder.

Changing Code Completion Settings

The previous section describes the default interface for code completion. However, Xcode provides a number of options for customizing code completion behavior. Code completion settings apply to all projects that you open.

You can control how much or how little information Xcode gives you as you type. To control how code completion suggestions are made, use the following options:

- Select "Indicate when completions are available" to have Xcode indicate when it has suggestions for matching symbols by underling the text you have typed. This option is enabled by default. If you turn it off, completions are still available to you, but Xcode does not underline the text. You can open the completion list to see suggestions, or cycle through the available completions, as described in the previous section.

- Select "Automatically suggest on member call / access" to have Xcode automatically display the completion list when in the context of a function or method call, or when accessing the members of a data structure. If you choose this option, you can also specify the amount of the delay, after you stop typing, before Xcode displays the completion list. Specify the amount of time, in seconds, of this delay in the "Suggestion delay" field. By default, this delay is half a second.

Xcode also lets you control how functions and methods are completed. To have Xcode insert the name, as well as placeholders for the arguments to the function or method, select "Insert argument placeholders for completions."Otherwise, Xcode only inserts the name of the function or method. This option is on by default.

To choose how functions and methods are displayed in the completion list, use the "Show arguments in pop-up list" option. Select this option to have Xcode display functions and methods with their list of arguments in the completion list. Otherwise, Xcode only displays the function or method name. This option is on by default.

Text Macros

Using code completion to automatically complete symbol names saves you a lot of typing. In the course of writing source code, however, you still spend a lot of time typing the same basic code constructs—such as `alloc` and `init` methods in Objective-C programs, for example—over and over again. To help you with this, Xcode includes a set of text macros. Text macros let you insert common constructs and blocks of code with a menu item or keystroke, instead of typing them in directly.

You can insert a text macro in either of the following two ways:

1. Choose Edit > Insert Text Macro and then choose a text macro from one of the language-specific menus. Xcode provides built-in text macros for common C, C++, Objective-C, Java, and HTML constructs.

2. Type the completion prefix for the text macro and use code completion to insert the remaining text, just as you would complete a symbol name. Each text macro provided by Xcode has a completion prefix, a string that Code Sense uses to identify the text macro. When you type this string, Xcode includes the text macro in the completion list; you can select it from this list or cycle through the appropriate completions, as described in "Using Code Completion" (page 189).

The inserted text includes placeholders for arguments, variables, and other program-specific information. For example, choosing Edit > Insert Text Macro > C > If Block inserts the following text at the current insertion point in the active editor:

```
if (<#condition#>) {
     <#statements#>
}
```

Replace the placeholders `<#condition#>` and `<#statements#>` with your own code. You can cycle through the placeholders in a text macro in the same way you can cycle through function arguments with code completion. A text macro can also define one placeholder to be replaced with the current selection, if any. When you select text in the active editor and insert a text macro, Xcode substitutes the selected text for this placeholder. For the If Block text macro described above, Xcode substitutes the selected text for the `<#statements#>` placeholder. For example, if the current selection in the text editor is `CFRelease(someString);`, inserting the If Block text macro gives you the following:

```
if (<#condition#>) {
```

```
CFRelease(someString);
}
```

If there is no selection, Xcode simply inserts the <#statements#> placeholder, as in the previous example.

Some text macros have several variants. For example, the text macro for inserting an HTML heading has variants for the different levels of headings. For text macros that have multiple variants, repeatedly choosing that text macro from the Insert Text Macro menus cycles through the different versions of that macro. For example, choosing Insert Text Macro > HTML > Heading a single time inserts <$(h1)><#!text!#></$(h1)>; choosing it again inserts <$(h2)><#!text!#></$(h2)>.

Using an External Editor

Xcode lets you choose the editors used for the files in your project. You can use Xcode's built-in editor or use an external editor such as BBEdit. Generally, Xcode uses the filename extension to choose how to edit a file. For example, it edits an `.rtf` file with its own built-in RTF editor and a `.c` file with its own built-in source code editor.

You can temporarily change how a file is viewed, or permanently change how files of a certain type are viewed. To change how a file is viewed, you can do any of the following:

- To have files of a certain type always open in a different editor, change the preferred editor for that file type to that editor.
- To have files of a certain type always open in the application specified for them in the Finder, change the preferred editor for that file type to "Open With Finder."
- To temporarily force Xcode to treat a file as a different file type, and open it with the appropriate editor, use the Open As command.
- To temporarily force Xcode to open a file with the default application chosen for it in the Finder, use the Open With Finder command.

Note: HTML files are handled differently. If Xcode determines that an HTML file is documentation, Xcode assumes you want to view the file and displays the file with its built-in HTML viewer. Otherwise, Xcode assumes you want to edit the file and uses its built-in source code editor.

Overriding How a File is Displayed

You can temporarily override how Xcode displays a file. For example, you can choose to view a particular HTML file as plain text, so you can edit it instead of viewing it as rendered HTML.

To force a file to be displayed differently than is specified by the default rule for that file type, select the file in the Groups & Files list or detail view and choose an option from the File > Open As menu. You can also Control-click the file, and choose an option from the Open As menu in the contextual menu Xcode displays.

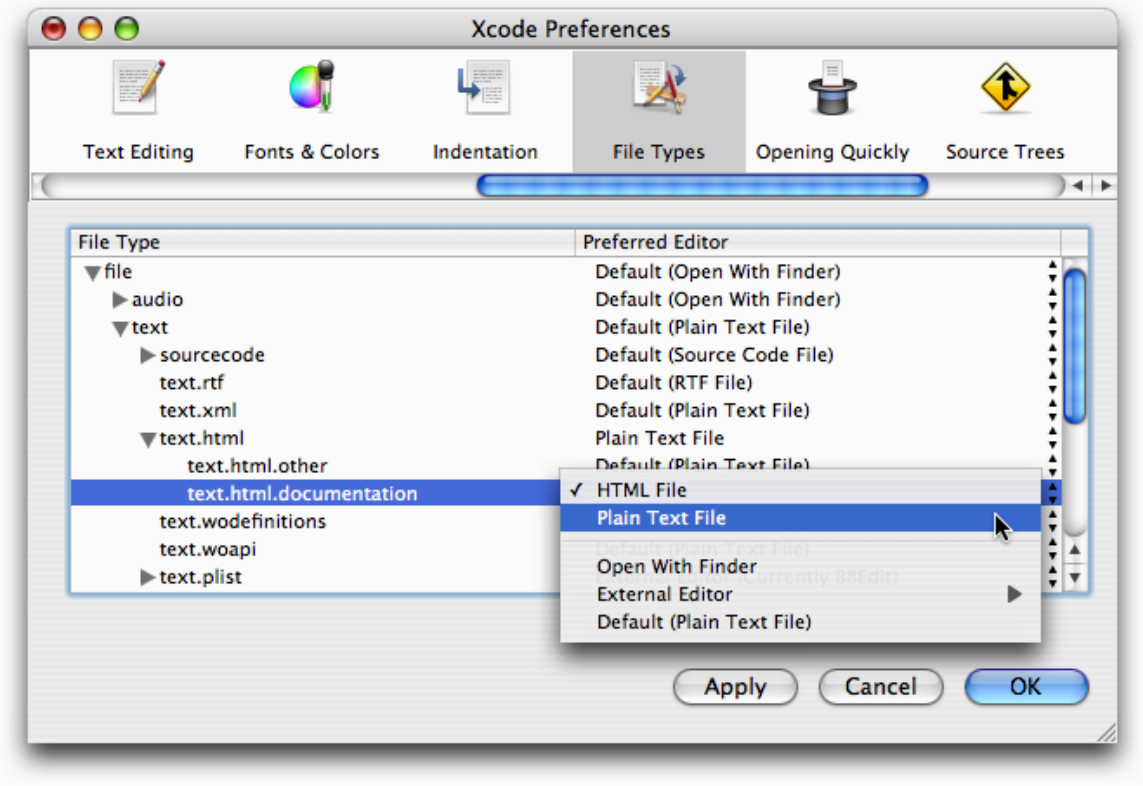
Changing the Preferred Editor for a File Type

You can permanently change how Xcode edits a particular type of file. In particular, you can specify how files of a certain type are treated and you can choose which editor is used to handle those files. For example, you can choose to view all HTML files as plain text, so you can edit them. Or you can choose to edit all your source files in BBEdit.

To see the file types that Xcode recognizes, choose Xcode > Preferences and click File Types. The File Types pane lists all of the folder and file types that Xcode handles and the preferred editor for each of those types. These file and folder types are organized into groups, from the most general to the most specific. Click the disclosure triangle next to an entry in the File Type column to reveal its contents.

To change the editor used for a particular file type, find the entry for the file type, click in the Preferred Editor column, and choose an option from the menu that appears. For example, to change Xcode to view all HTML files—including documentation files—as plain text, expand the following entries: file, then text, and then text.html. As mentioned earlier, Xcode already treats most HTML files as plain text by default; the value in the Preferred Editor column for the text.html entry is "Plain Text File," indicating the preferred editor for plain text files. However, the value in the Preferred Editor column for the text.html.documentation entry is "HTML File," which overrides the text.html setting. To make Xcode treat HTML documentation files as plain text, select text.html.documentation, click in the Preferred Editor column and choose "Plain Text File" from the pop-up menu, as shown here.

Figure 18-1 Changing how a file is viewed



With this change, Xcode uses the preferred editor for plain text files to open all HTML files. In the example shown in the previous figure, the preferred editor for plain text files is Xcode's default text editor.

You can also specify an external editor to use or have Xcode use the user's preferred application, as specified by the Finder, when opening files of a given type, as described in the following sections.

Opening Files With an External Editor

Xcode does not limit you to using its built-in editors to view and edit your files. You can specify an external editor of your choosing as the preferred editor for opening files of a given type. To choose an external editor for all files of a particular type:

1. Choose Xcode > Preferences, and click File Types.
2. Find the appropriate file type and click in the Preferred Editor column; a pop-up menu appears.
3. Select an option from the External Editor submenu. Currently, you can choose from the following options:
   - BBedit.
   - Text Wrangler.
   - SubEthaEdit.
   - `emacs`.
   - `xemacs`.
   - `vi`. Note that support for `vi` in Xcode is limited to opening the file in the editor.
   - Other. Choose this option to specify an external editor other than the ones specified earlier. When you select this option, a dialog that allows you to navigate to the application you wish to use as your external editor appears.

   Note that many of these external editors do not appear in the External Editor menu unless they are installed on your computer.

For example, to edit all your source files with BBEdit, open the File Types preference pane and expand these entries: file, then text. Select the source code entry, and choose External Editor > BBEdit from the pop-up menu that appears when you click in the Preferred Editor column.

There are some restrictions when you're using an external editor:

- When you build a project, Xcode lists modified files and asks you whether you want to save them. Files in BBEdit and Text Wrangler are listed, but files in other editors are not. You need to save those files yourself before starting a build.
- When you double-click a find result or a build error, most editors do not scroll to the line with the find result or error. BBEdit and Text Wrangler can.

To use `emacs` as an external editor, you must add these lines to your `~/.emacs` file:

```
(autoload 'gnuserv-start "gnuserv-compat"
          "Allow this Emacs process to be a server for client  processes." t)
(gnuserv-start)
```

Opening Files With Your Preferred Application

You can choose to open a file with the application chosen for it in the Finder. This lets you open files that Xcode cannot handle, or view a file using your preferred editor. If you edit a file in almost any other application, Xcode cannot save it for you before building a target. Some applications, such as Interface Builder and WebObjects Builder, communicate with Xcode and so can save your files before your project is built. Check the application's documentation to see if it can, too.

To always have Xcode use your preferred application to open files of a certain type:

- Choose Xcode > Preferences and click File Types.
- Find the appropriate file type and choose Open With Finder from the pop-up menu that appears when you click in the Preferred Editor column.

Note that you can only set this preference for file types that Xcode recognizes. To open files that Xcode cannot handle, or to temporarily override the settings in the File Types pane of Xcode Preferences and open a file using the Finder-specified application:

- In the Groups & Files list or detail view, Control-click the file and choose Open with Finder.

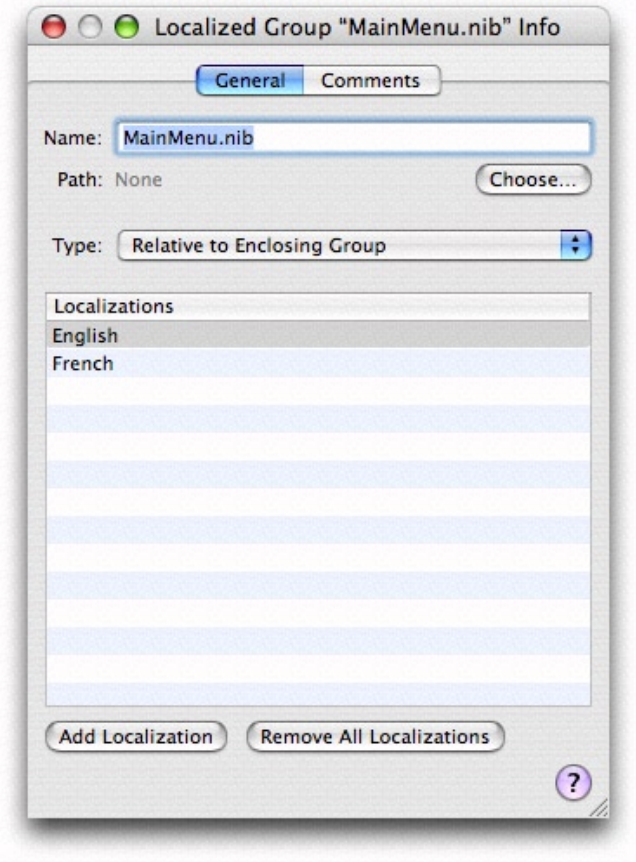
If you have the embedded editor open, single-clicking a filename still loads the file in the editor. But if you configure Xcode to use an external editor to edit the file, you cannot edit the file within Xcode. That is, the file is read-only in Xcode's built-in editor.

Customizing for Different Regions

Xcode lets you create applications, bundles, and frameworks that are customized for different regions. Generally, you'll start by creating a variant for one particular region, called the development region, and add more variants later.

In the Groups & Files list, a file customized for different regions appears as a localized group, which has a file icon with a triangle beside it. To see the file's variants, click the triangle. To add and remove variants, select the localized group, open the inspector window, and use the two buttons at the bottom of the General pane, as shown in the following figure.

Figure 19-1 Inspecting a localized group



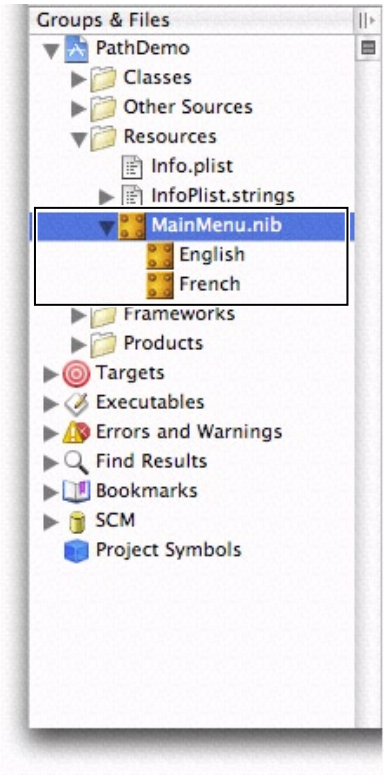
For more information on localizing your product for different regions, see *Internationalization Programming Topics*.

Marking Files for Localization

To mark files for localization, select the files, open the inspector, and click the Make File Localizable button. Xcode moves the files into the development region's `.lproj` folder. If a file was already in another `.lproj` folder, Xcode copies it to the development region's `.lproj` folder.

Xcode creates a localized group in the Groups & Files list, with the file's name and icon. To view the individual localization variants, click the disclosure triangle next to the localized group icon. The following figure shows the localized group for an application's main nib file in the Groups & Files list.

Figure 19-2 A localized group in the Groups & Files list



You can inspect any of the localized variants individually or you can inspect the localized group as a whole.

To remove files from localization, select the files, open the inspector, and click the Remove All Localizations button. Xcode moves the files from the development region's `.lproj` folder into the folder for nonlocalized resources. Other localized versions of the files are removed from the project but are not deleted from the disk.

Adding Files for a Region

To add files for a region, select the file or localized group for which you want to add another region, open the inspector window, and click the Add Localization button. Xcode queries you for the name of the localization region and copies the development region's version of the files to the new region's `.lproj` folder.

Version Control

Version control is a set of tools and procedures that streamline the safekeeping of files and their change histories. Version control also allows several developers to work on the same project at the same time. Xcode provides a user interface to the client programs of version control systems in order to facilitate working with source files maintained in a version control repository.

Version control systems use the client/server model: A server program manages the repository (a directory tree or a database) that holds the managed files; developers use client programs to communicate with the server and perform tasks such as retrieving files from the repository or submitting changes to them. Typically, the clients are command-line tools, although there are also clients that use a graphical user interface.

The following chapters show how to use Xcode to work on projects under version control. They provide a client-side perspective of manipulating source code and resources hosted in a version control repository. This document doesn't provide an exhaustive treatment of version control.

PART IV

Version Control

Overview of Version Control

Version control (also known as Source Control Management or SCM) is a set of tools and procedures to safeguard and manage files and changes made to them over time. A version control system frees you from having to manually manage access to the files that comprise a software project and track changes. Version control systems take care of these details, allowing you to concentrate on writing and testing code.

You should read this chapter if you're new to version control and are interested in adopting version control practices in your projects. After reading this overview, you'll understand how version control provides an infrastructure that improves the development experience for single developers and multiple developers working in the same project.

This chapter describes the essential aspects of version control systems. If you're familiar with version control, you can skip this chapter.

Have you ever wondered which files you modified to correct a problem in one of your projects? Or, how about undoing changes you made the previous day because you came up with a better solution to the problem? A version control system can give you answer to the first question quickly; you don't need to check modification dates or comments inside source files. It would also let you discard multifile changes so you can reimplement the solution to a problem without manually identifying and removing each change in every file you touched.

A version control system has three major parts: a repository, a client, and a server. The **repository** is a directory tree or database that contains the files managed by a version control system. The files stored in the repository are called **managed files**. Repositories can reside anywhere but are usually placed in a computer managed by a system administrator who sets access to the repository and ensures the persistence of its contents through regular backups.

The **client** is the program developers use to interact with a repository. The **server** is the process that actually modifies the repository. When a developer issues a command to the client, the client talks to the server process to carry it out.

Every developer authorized to access the repository can copy files from the repository into a local directory, also known as a **working copy**. This is where developers make changes to the project; they never work with the files in the repository. Developers normally don't have access to each other's working copies. This feature provides privacy and security because developers are unaware of what their peers are doing until they publish or **submit** their changes to the repository.

When a developer submits changes to a file in the repository, its **version number** (also known as the **revision number**) is incremented. The history of each file is recorded as a set of revisions, which you can retrieve and compare individually.

Version control provides several benefits, including:

- Centralized location of files

  When multiple developers work on a project concurrently, version control ensures that the project's official files are kept in a central location. As a result, the project's products can be built at any time without having to get the latest files from multiple locations.

- Complete history of every file

  Because all the changes made to each file in the repository are maintained in the repository, you can review the evolution of each file or an entire project since its inception. This information can be valuable when investigating the causes of software bugs.

- Change management infrastructure

  Version control systems don't allow developers to submit changes to the repository without describing the purpose of the change, which forces developers to document their work. This requirement saves time in the long run because it allows everybody to determine the reason for a particular change without having to find the person who performed the change. Version control also lets developers group changes in the way that best fits the them and their team. For example, developers can submit changes on a daily basis, whether the items they're working on are finished. This feature reduces the amount of data loss that can occur in case a developer's computer fails unexpectedly; however, it also reduces the stability of the project. Submitting changes to the repository only after a feature is completely implemented and tested is a better approach, especially for sensitive features that should be kept secret for some time.

Consider using version control when several developers work on one project at the same time. Single developers can also benefit from the structure that version control adds to the development process, such as revisions and change management.

Xcode provides a common interface to various version control systems, which include the open-source CVS (Concurrent Versions System) and Subversion, and Perforce. Xcode makes it easy to perform most version control tasks as you develop. It also tells you whether you've modified managed files in your local copy of the project. "Managing Projects" (page 205) and "Managing Files" (page 211) describe Xcode's version control interface in detail.

Managing Projects

Xcode provides an easy-to-use interface to your version control client tool; you can perform the most common version control tasks as you perform normal development tasks. However, there are some tasks you must perform using your client tool, such as adding projects to a repository and checking out projects. You seldom need to perform these tasks. For example, you need to check out a project to work on it for the first time on a particular computer. See "Using CVS" (page 427) and "Using Subversion" (page 431) for details.

This chapter describes the files that store project and user information for Xcode projects. You manage these files like any files that you modify directly during development under version control. This chapter also shows how to configure access to the repository a project is housed in.

Project Packages

Xcode saves project metadata and individual developer settings in the **project package**, which is named after the project and with the extension `.xcode`. (Earlier versions of Xcode used the `.pbproj` or `.pbxproj` extensions.) The project package is located in the project directory, for example, `Sketch/Sketch.xcode`. This package contains two files that you need to pay particular attention to in order to keep the project package in sync with the rest of the files in the project and to maintain personal project settings in the repository: The project file and the user file.

- The project file, which Xcode maintains in the project package under the name `project.pbxproj`, stores project-related information, such as the files that are part of the project, the groups in the Groups & Files list, build settings, target definitions, and so on. Xcode constantly writes out this file; therefore, most of the time its status is 'M' (for details on file status codes, see "Viewing File Status" (page 211)).

  If you make structural changes to the project, such as adding or removing files, you must commit this file along with the other changes (for example, when you commit a file you added to your working copy, you must also commit the project package). Otherwise, the project file may become out of sync with the rest of the project's files (source code files, resource files, and so on).

- The user file stores user-related information, such as bookmarks, the active build style, and so forth. Xcode maintains this file inside the project package as `<username>.pbxuser`. So, for the user name `clare`, the corresponding user file is called `clare.pbxuser`.

  Xcode adds your user file to a project package when you open the project if there isn't a user file for you inside the package. You should include your user file in your commits and updates if you want to safeguard your personal settings for the project in the repository or if you work on the project on more than one computer and want to use the same settings on all of them.

Configuring Repository Access

A **managed project** is one whose root directory is stored in a repository and whose access is controlled by a version control system. Before you can work on a managed project, you must check it out of the repository into a local copy.

Important: Xcode doesn't check out projects from a repository. You must use your client tool to create local copies of the projects you want to work on. See "Checking Out Projects From a CVS Repository" (page 430) and "Checking Out Projects From a Subversion Repository" (page 433) for details.

After you check out a project directory, you must open the project in Xcode and configure your repository-access settings. These include the name of the version control system that manages the repository, the path to the client tool, authentication information, and whether version control is active. These settings are saved in your user file in the project package. For more information on the project package, see "Project Packages" (page 205).

Follow these steps to configure your repository-access settings for a project:

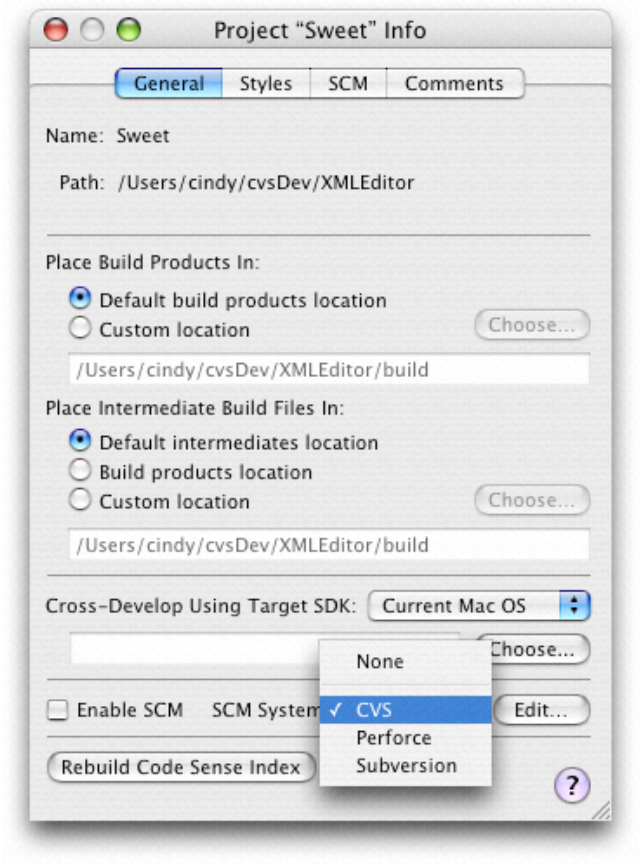
1. Open the project in Xcode.

   If you use a version control system that locks files in working copies, you may have to tell it that you intend to modify the project package before opening the project. For example, with Perforce, you would execute the following commands:

```
% cd ~/Working/Echo
% p4 edit Echo.xcode/...
% open -a Xcode Echo.xcode     # Or open it using the Xcode Open  command.
```

2. Choose your version control system from the SCM System pop-up menu in the General pane in the Project Info window, shown in Figure 21-1.

Figure 21-1 The SCM System pop-up menu

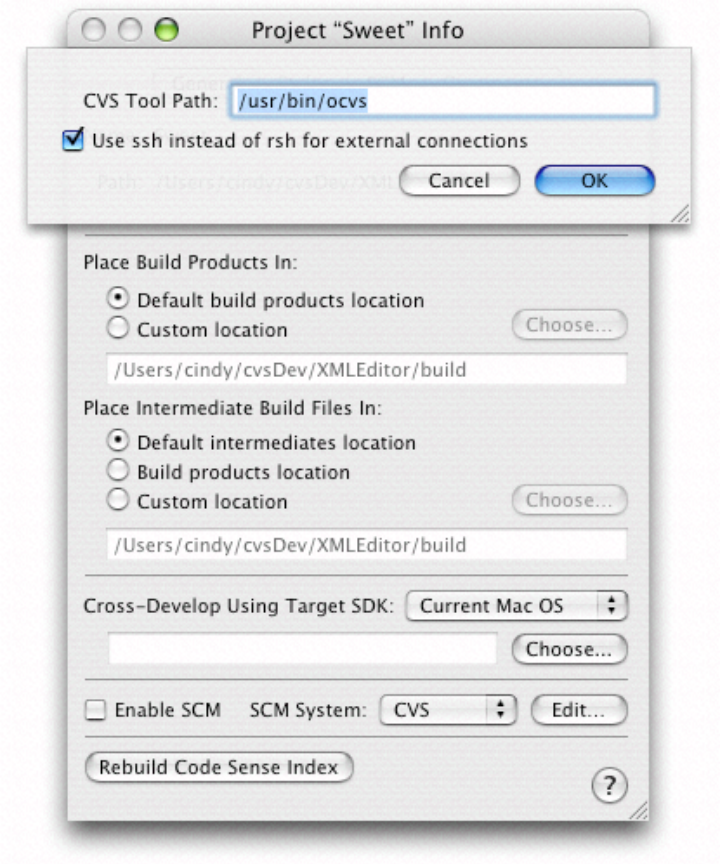


3. Tell Xcode how to use your client tool.

   Click Edit and enter the path to the client program in the client configuration dialog.

If you use SSH to access a CVS repository, select "Use ssh instead of rsh for external connections" as shown in Figure 21-2. See "Accessing a CVS Repository" (page 429) for further details.

Figure 21-2 Client configuration dialog for CVS

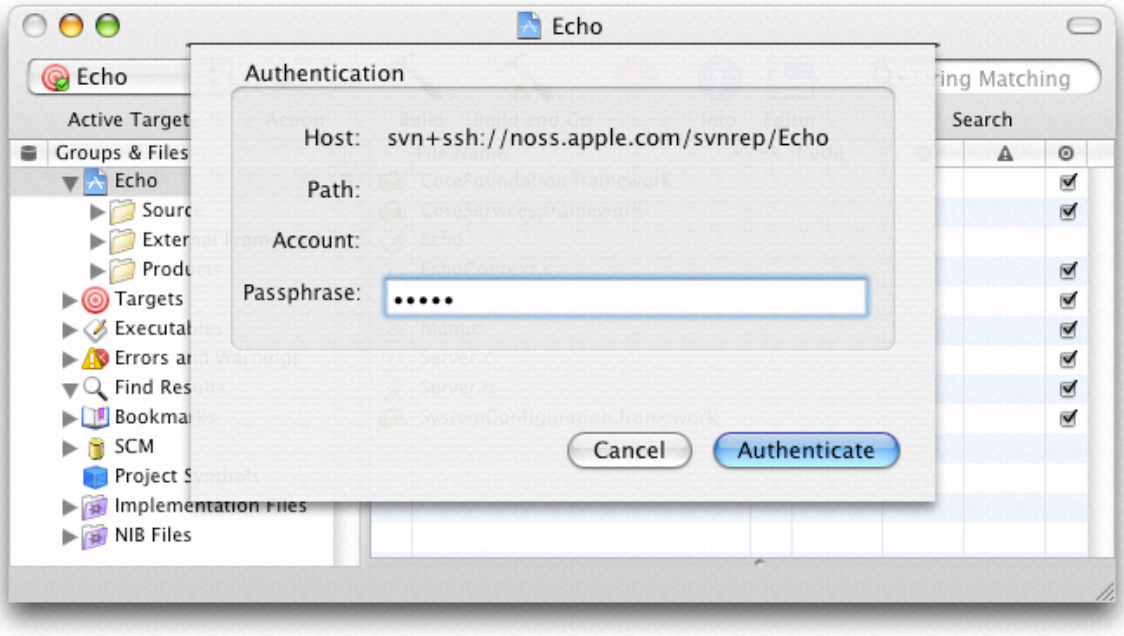


4. Activate version control for your copy of the project.

   Select Enable SCM in the General pane in the Project Info window.

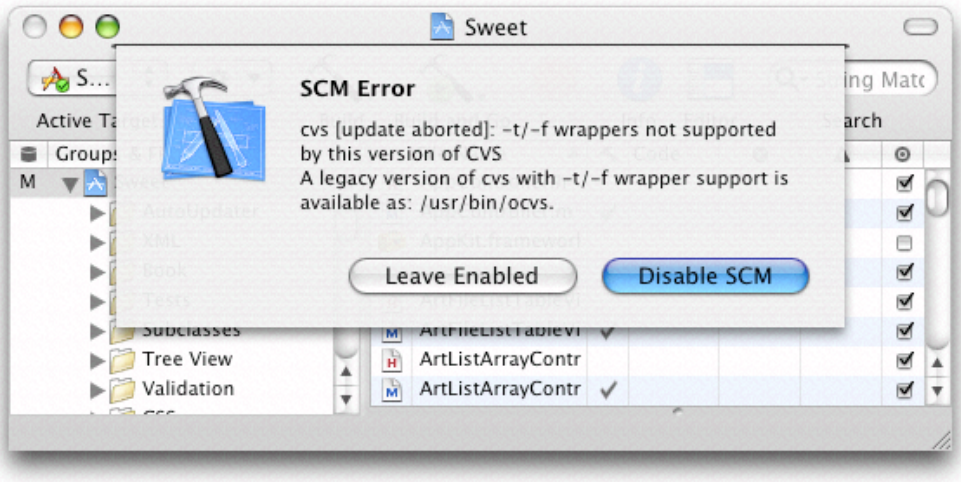
If you use SSH to access a Subversion repository, Xcode may ask you to enter your passphrase in the Authentication dialog, shown in Figure 21-3.

Figure 21-3 Authentication dialog for Subversion



If Xcode is unable to talk to your client, a dialog describing the problem appears. In it you may choose to disable SCM or to leave it enabled. For example, if you try to access a project that contains wrapped bundles with a version of CVS that doesn't support wrappers, the dialog shown in Figure 21-4 appears.

Figure 21-4 SCM Error dialog



5. Commit your user file to the repository.

This step is not required; however, saving your user file in the repository provides two main benefits:

- It ensures that you've correctly configured version control for your working copy of the project.
- It allows you to use the same personal settings for the project in any computer you use to work on it.

You now know how to set up version control in your projects. "Managing Files" (page 211) explains how to add version control operations to your development workflow.

Managing Files

Version control allows you to maintain a history of a project's development and lets you share projects with other developers. Xcode, in conjunction with your version control system, allow you to stay up to date with your team's progress. Through the Xcode user interface, you can perform most of the version control tasks needed to work on a software project successfully.

This chapter introduces common version control tasks and explains how to accomplish them in Xcode. It also provides the recommended workflow you should follow when working on managed Xcode projects.

Viewing File Status

As you work on a project, the version control status of its files in relation to the repository change. Xcode tells you which files you have changed in your local copy of the project, which files need to be updated to the latest version in the repository, and so forth.

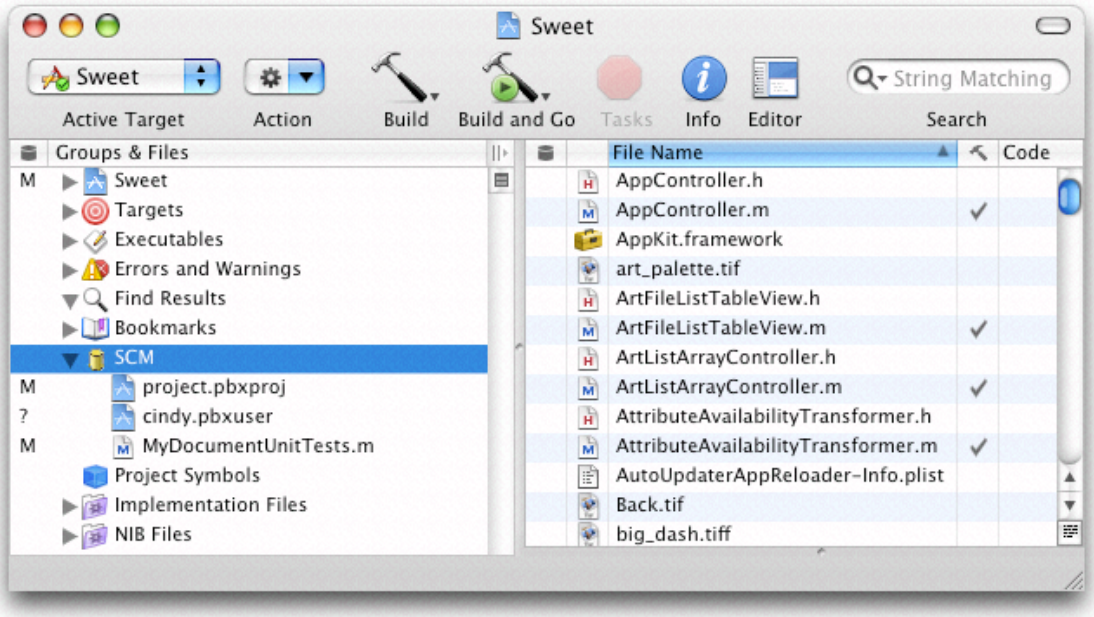
Xcode uses a one-letter code to represent the status of each file. Here's what each code means:

- Blank: The file is up to date with the latest version in the repository. You haven't changed your local copy of it.
- **?** Unknown: The file is not in the repository. See "Adding Files to the Repository" (page 213).
- **-** Dash: The file is in a directory that's not in the repository, or this is a directory that's not in the repository. To add a directory to the repository, you must use your client tool. After that, add the files in the directory using Xcode. If you don't use Xcode to add the files, Xcode will not add the files to the project file. In turn, when you commit your changes, Xcode will not notify other developers that files have been added to the project.
- **U** Update: The latest version of the file in the repository is newer than your version. To check for conflicts between your version and the latest revision and then get the latest revision if there are no conflicts, select the file in the detail view and choose SCM > Update To > Latest.
- **C** Conflict: Your changes may conflict with the changes in the latest version. To see conflicts, select the file in the detail view and choose SCM > Compare With > Latest.
- **M** Modified: The next time you commit your changes to the file, either by selecting it and choosing SCM > Commit Changes or by choosing SCM > Commit Entire Project, the modified version of this file is added to the repository.
- **A** To Be Added: The next time you commit your changes, this file is added to the repository.
- **R** To Be Removed: The next time you commit your changes, this file is removed from the repository.

You can view the version control status of files in several places: The SCM Group, the detail view, and the SCM Results window.

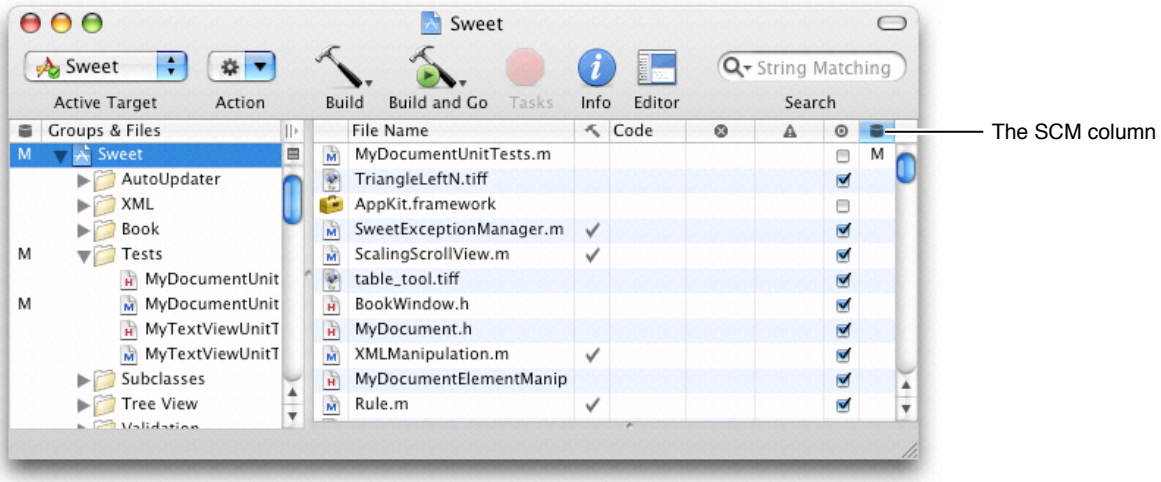
The SCM group in the Groups & Files list in the project window shows the status of all the files that differ from the latest version in the repository or for which you've specified a version control operation to be performed later, such as adding a new file to the repository. Figure 22-1 shows the SCM group.

Figure 22-1 The SCM group in the Groups & Files list



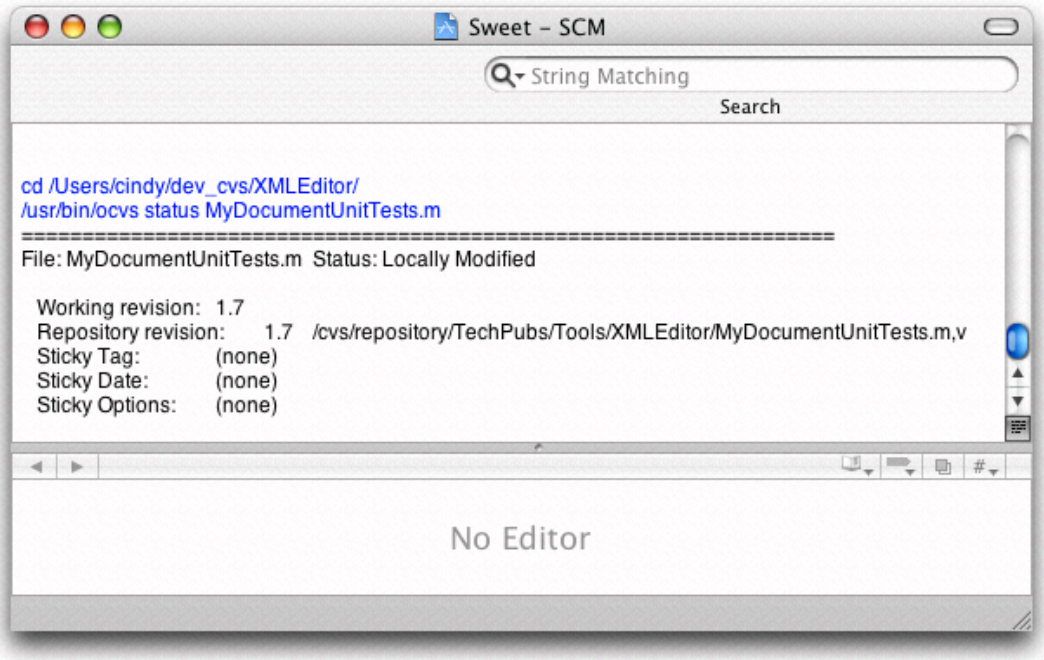
Xcode's detail view lists all the files in a project. When using version control, you can add the SCM column to the detail view by selecting the project group in the Groups & Files list and choosing View > Detail View Columns > SCM. The SCM column shows the status of each file in the project. Figure 22-2 shows the SCM column in Xcode's detail view.

Figure 22-2 The SCM column in Xcode's detail view



The SCM Results window provides the same information the SCM group does, plus an editor and the SCM results pane. The window appears when you double-click the SCM group or when you choose SCM > SCM Results. The button on the bottom-right corner of the top pane toggles between the file list pane and the SCM results pane. Figure 22-3 shows the SCM Results window with the SCM results pane and the editor pane.

Figure 22-3 The SCM Results and editor panes in the SCM Results window

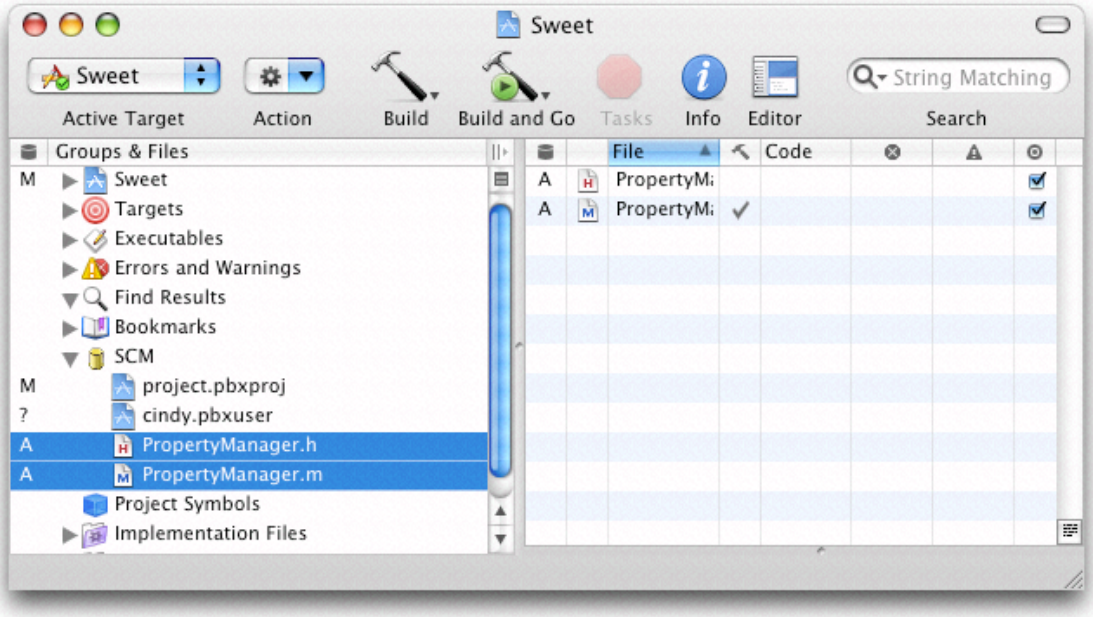


You can refresh the status of the files in your project anytime by choosing SCM > Refresh Entire Project.

Adding Files to the Repository

After you add a file to your local copy of a managed Xcode project, its status is ? (unknown). This means that the file is not part of the repository. If you want to add the file to the repository the next time you commit your changes, select the file in the project window or the SCM Results window and choose SCM > Add to Repository. The status of the file changes from ? to A. Figure 22-4 shows files in the SCM group to be added to the repository in the next commit.

Figure 22-4 Files to be added to the repository



When you commit file additions, you must commit the project file (`project.pbxproj`) as well, at the same time. This lets other developers know there's a new file in the project as soon as you commit the addition. If you don't commit the project file when you commit the file removal (that is, you select a file with a status of A, choose SCM > Commit Changes, and commit it without also selecting the project file), other developers will not be able to get the added file into their local copies of the project because Xcode wouldn't know that a file was added to the project.

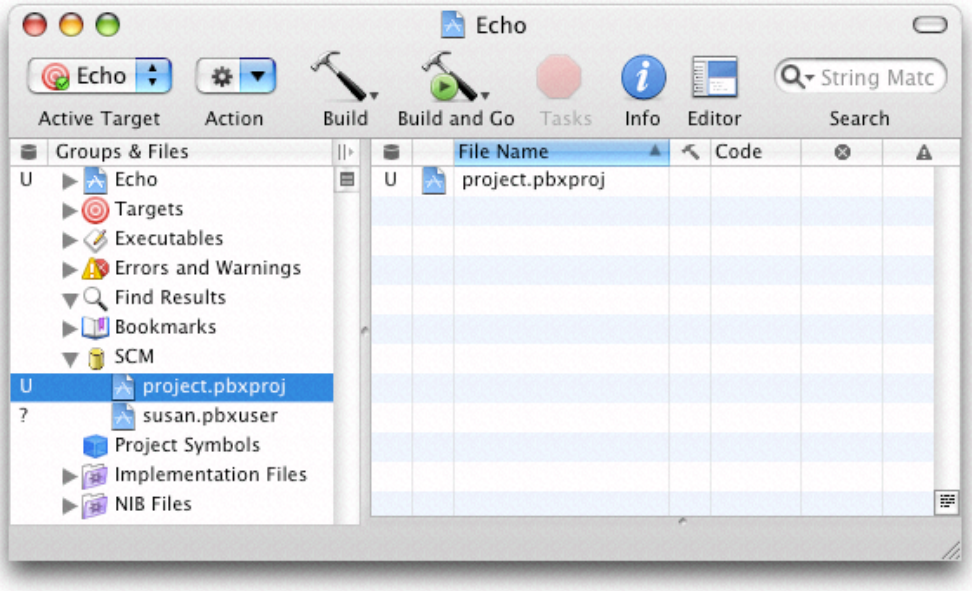
Updating Files

When a file in your local copy of a project becomes outdated, Xcode assigns it a status of U. This means that another developer has submitted changes to that file to the repository and your working copy doesn't include them. You can update your local copy of a file that needs updating one of two ways:

- Select the file in the project window and choose SCM > Update To > Latest.
- Choose SCM > Update Entire Project to update all files in the SCM group that have a status of U.

When the project file (`project.pbxproj`) has a status of U, you need to update the project file and reopen the project in Xcode. Figure 22-5 shows the SCM group of an Xcode project whose project file needs to be updated.

Figure 22-5 The SCM group in an Xcode project whose project file needs to be updated



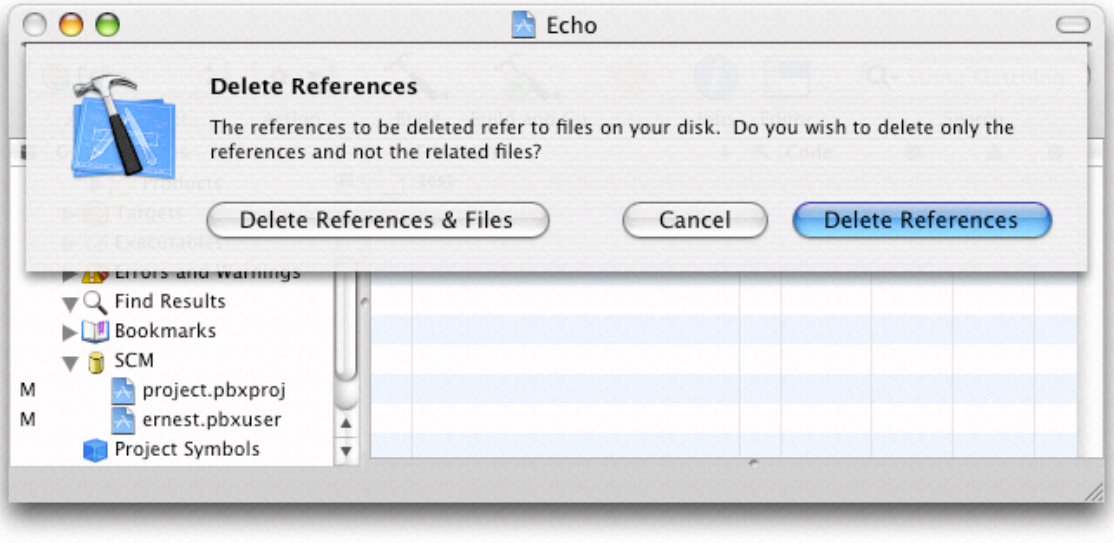
To update the project file, select `project.pbxproj` in the SCM group and choose SCM > Update To > Latest. Alternatively, you can choose SCM > Update Entire Project. After the update operation is completed, close the project and reopen it.

If you updated only the project file, when you reopen the project, the SCM group may display files with a status of U using red text. These files were added to the project after your last update. To get the new files into your working copy, select them in the project window and chose SCM > Update To > Latest. After the update operation is complete, choose SCM > Refresh Entire Project to refresh the status of the files in your working copy.

Removing Files From the Repository

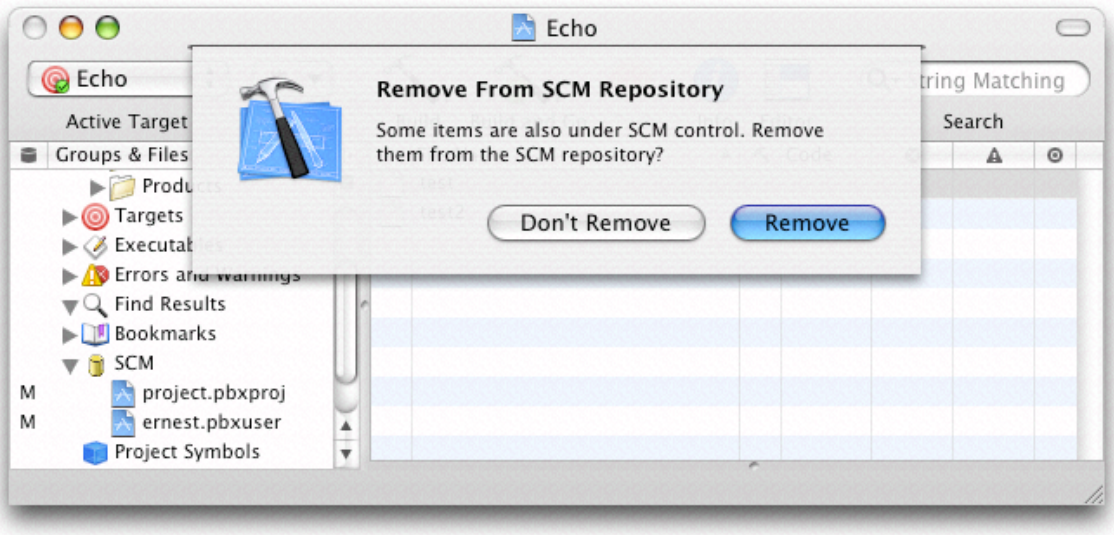
To remove a file from your local copy of a project and from the repository when you commit the operation, remove the file as you normally would; that is, select the file in the project window and choose Edit > Delete or press Command-Delete. The Delete References dialog, shown in Figure 22-6, appears.

Figure 22-6 The Delete References dialog



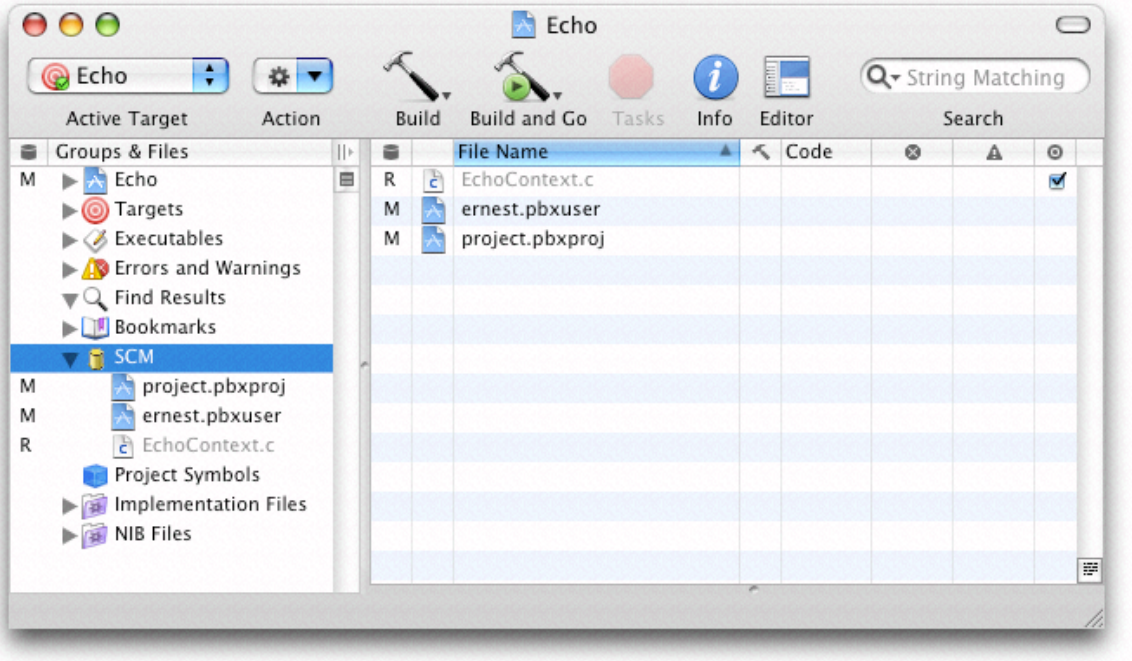
To remove the file from your local copy of the project, click Delete References & Files. The Remove From SCM Repository dialog appears, shown in Figure 22-7.

Figure 22-7 The Remove From SCM Repository dialog



To tell Xcode you want the file removed from the repository, click Remove. The file's status changes to R and the filename appears in gray, as shown in Figure 22-8.

Figure 22-8 File to be removed from the repository



When you commit the file-removal operation, you must commit the project file (`project.pbxproj`), at the same time. Other developers can then keep their project files in sync with the project directory by updating their local copies. If you don't commit the project file when you commit the file-removal operation (for example, you select a file with a status of R, choose SCM > Commit Changes, and commit it without also selecting the project file), Xcode notifies other developers that the file you removed needs to be updated. When they update their local copy with the repository, the file is removed from their local copy, but their copy of the project file still references the nonpresent file, and the file appears in red in the detail view. This may confuse developers, who will then have to find out why a file that's supposed to be in their project directories is missing.

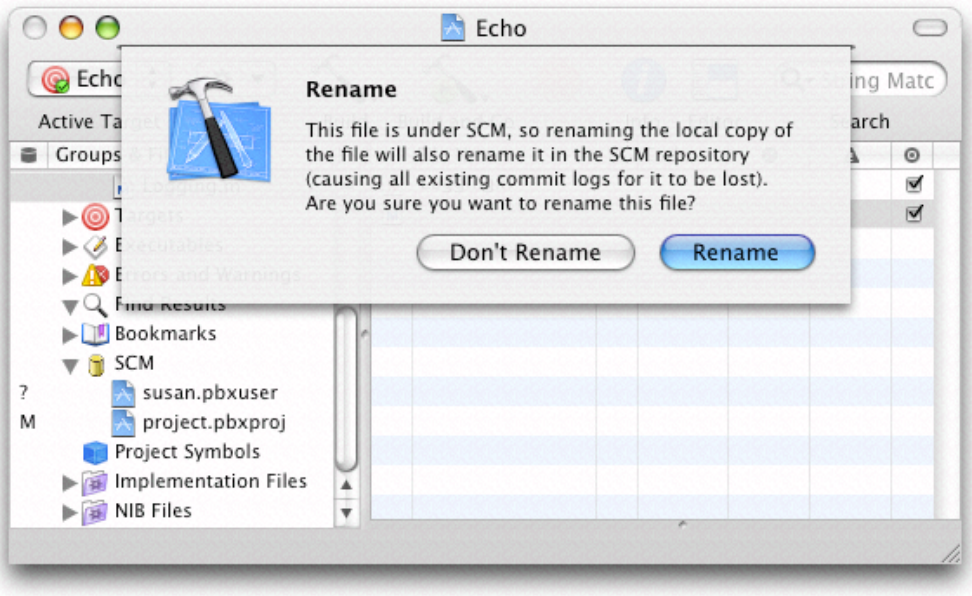
Renaming Files

Renaming a file produces two version control operations: the removal of the file under the old name and the addition of the file with the new name. Therefore, Xcode shows the R status next to the old name and the A status next to the new name.

Warning: When you rename a managed file, the change information for the file under the old name is unavailable under the new name.

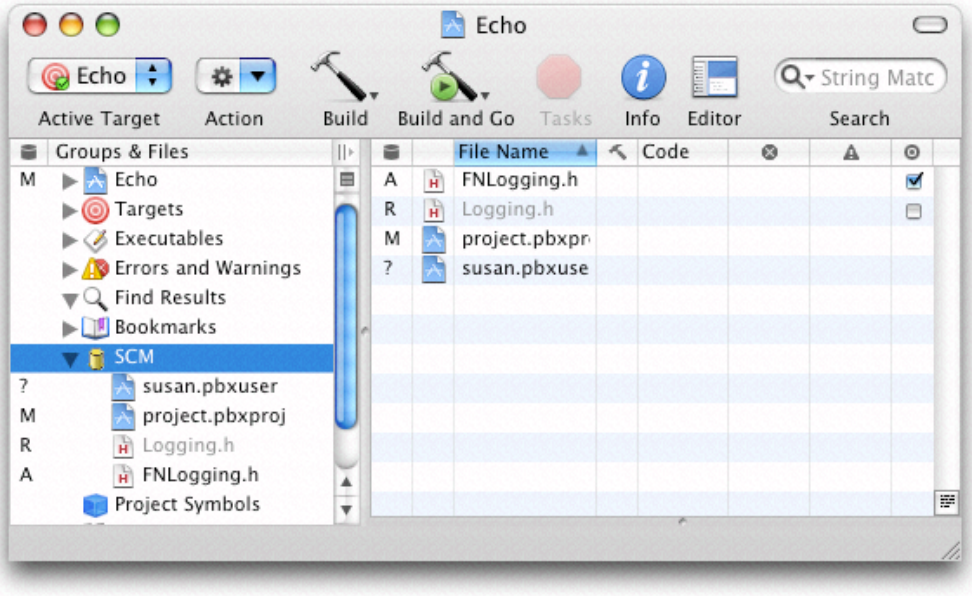
To rename a file, select the file in the project window and choose File > Rename. The Rename dialog appears, as shown in Figure 22-9.

Figure 22-9 Renaming a managed file



To proceed with the rename, click Rename. The SCM group in the Groups & Files list, as well as the detail view, show that the file under the old name will be removed and the file with the new name will be added, as shown in Figure 22-10.

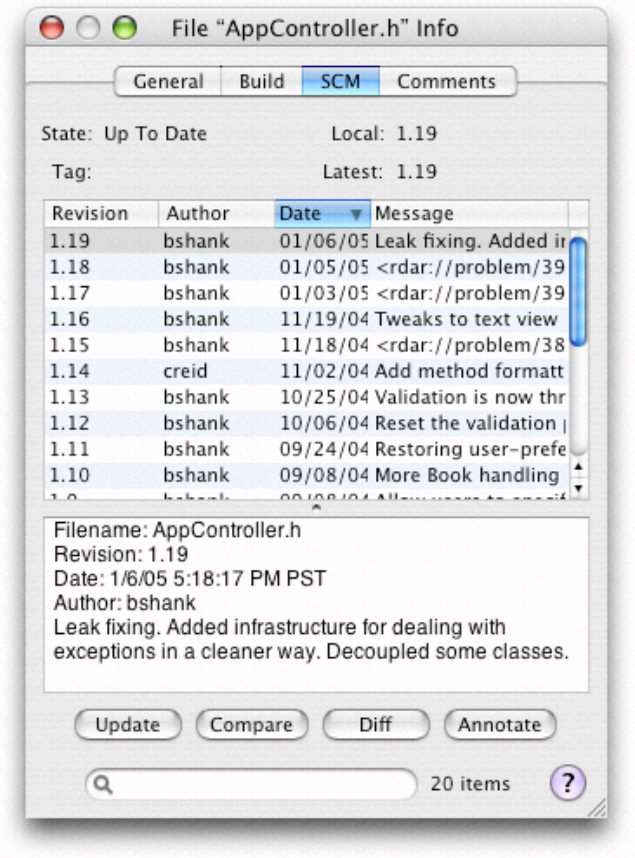
Figure 22-10 Uncommitted rename operation



Viewing Revisions

The SCM pane in the inspector or a file's Info window contains a list of each revision of the file. The information displayed in the list includes the revision number, the author, and a message about the changes made. The change message may be truncated in the list. To see the entire change message for a revision, select the revision in the revision list. The change message, as well as the other properties of the list, appears in the text field below the revision list. Figure 22-11 shows the revision of a file in an Info window.

Figure 22-11 Info window displaying the revisions of a file



You can compare any two versions of a file by selecting them in the list and clicking the Compare or Diff buttons, as appropriate. To update your copy of the file to a specific revision, select the revision and click Update.

Comparing Revisions

With Xcode you can compare different versions of a file in a project. This way, you can see changes made to a file from version to version. For example, you can compare your locally modified version of a file with the latest revision submitted by another member of your team. Or you can compare the two most recent revisions in the repository to see what has changed.

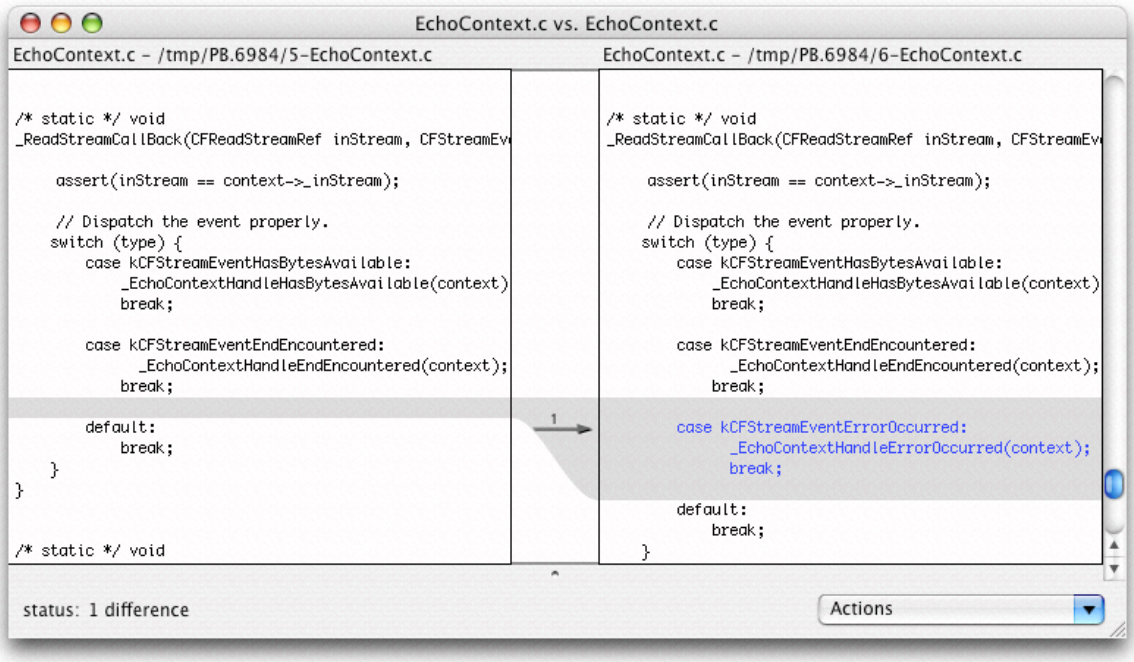
To compare your version of a file or project with a version in the repository, use the Compare With and Diff With commands in the SCM menu. You can also select any two revisions of a file in its Info window and compare them using the Compare and Diff buttons.

Xcode gives you a choice of tools to use when comparing files. The Compare With command lets you compare files using a visual tool, such as FileMerge. Alternatively, you can have Xcode perform the comparison using the differencing facility of your client.

The Compare Command

The Compare With command allows you to compare files using a visual tool. Figure 22-12 shows the result of comparing two revisions of a file.

Figure 22-12 Comparing two revisions of a file using FileMerge



To compare your version of a file with a revision in the repository:

1. Select the file in the project window.
2. Choose SCM > Compare With and select the revision to compare against:
   - Latest. Choose this option to compare a file with the latest version in the repository.
   - Base. Choose this option to compare a file with the version you checked out of the repository.
   - Revision. Choose this option to get the revision list for the selected file. This option is useful if you're not sure of the revision you want to compare against.
   - Specific Revision. When you know the revision you want to compare against, choose this option and enter the revision number in the dialog that appears.

- File. Choose this option to compare a file in your project with any file on disk. Choose the other file from the dialog that appears.

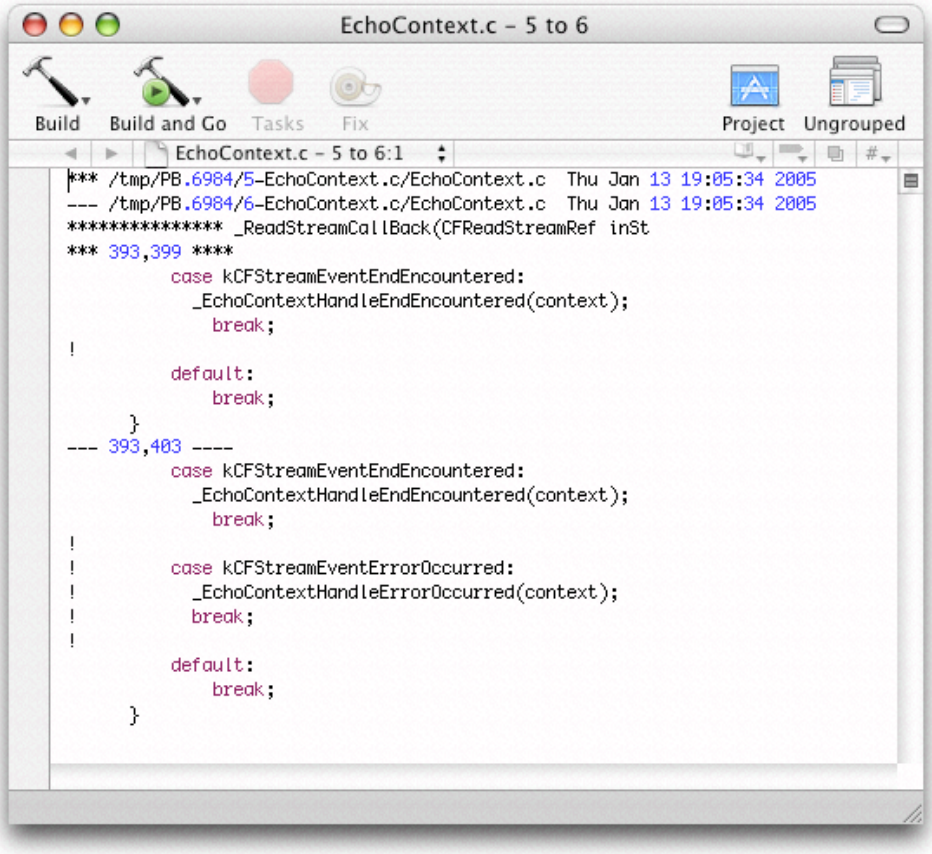
You can also compare any two revisions of a file in its Info window by doing the following:

1. Select the file, open its Info window, and click SCM.
2. Select the revisions you want to compare in the revision list and click Compare.

The Diff Command

Another way to compare revisions using Xcode is to identify the differences between them by using the differencing facility of your client tool. Xcode displays the output of the `diff` command in a separate editor window. Figure 22-13 shows an example of comparing two revisions of a file using `svn diff`.

Figure 22-13 Identifying differences between two revisions of a file



You can specify the format used in the comparison in the Differencing section in the SCM pane in the Xcode Preferences window. See "Specifying Comparison and Differencing Options" (page 222) for details.

To identify the differences between your copy of a file with a revision in the repository, select the file in the project window, choose SCM > Diff With, and choose a version to compare against. The options you can choose from are:

- Latest. Choose this option to compare a file with the latest revision in the repository.
- Base. Choose this option to compare a file with the revision you checked out of the repository.
- Revision. Choose this option to get the revision list for the selected file. This option is useful if you're not sure which revision you want to compare against.
- Specific Revision. When you know the revision you want to compare against, choose this option and enter the revision number in the dialog that appears.

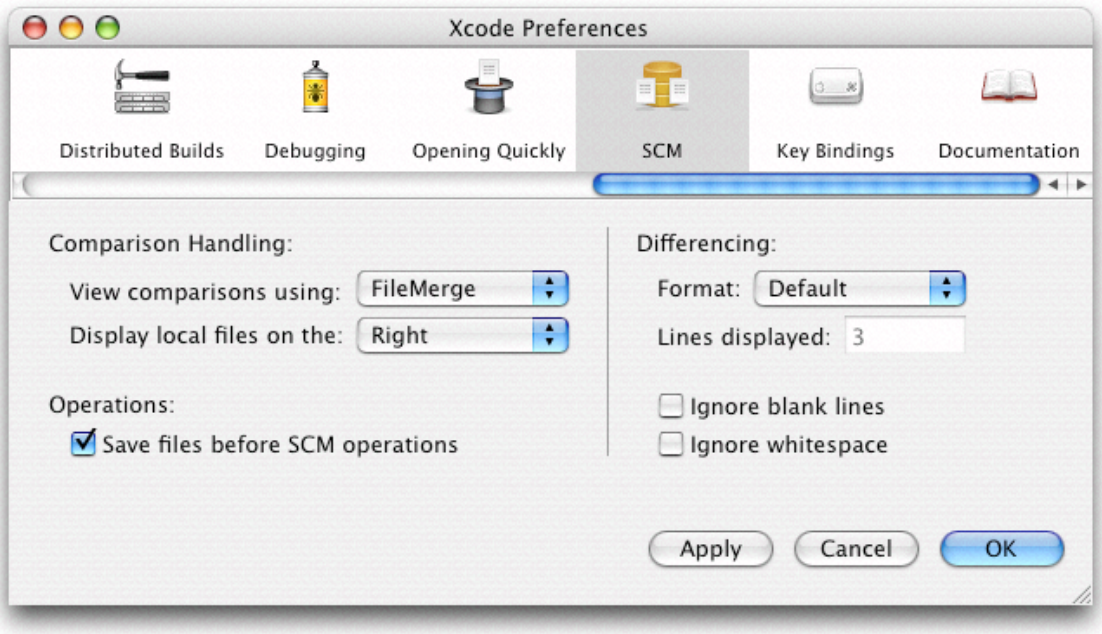
You can also identify the differences between any two revisions of a file in its Info window by doing the following:

1. Select the file, open its Info window, and click SCM.
2. Select the revisions you want to compare from the revision list and click Diff.

Specifying Comparison and Differencing Options

Xcode stores personal settings such as comparison and differencing preferences, whether Xcode saves open files before performing version control operations, and so forth, in your user file (`<username>.pbxuser`) in the project package (`<project_name>.xcode`). You specify those settings in the SCM pane in the Xcode Preferences window, shown in Figure 22-14. (See "Project Packages" (page 205) for more information on project packages.)

Figure 22-14 The SCM pane in the Xcode Preferences window



The Comparison Handling section of the SCM preferences pane allows you to specify how comparisons are performed:

- The "View comparisons using" pop-up menu lets you choose the tool you want Xcode to invoke when you execute the Compare command. You can choose between FileMerge, BBEdit, or Other to specify an application of your choosing.
- The "Display local files on the" pop-up menu specifies whether local files are displayed on the left or the right in FileMerge comparison windows.

The Operations section specifies whether Xcode saves files before performing version control operations through the "Save files before SCM operations" option.

The Differencing section lets you specify how you want the differencing performed:

- The Format pop-up menu specifies the output format used to display the results of the comparison. The possible formats are:
  - Default. This is the default output format used by your client's `diff` command.
  - Contextual. The output uses the context format, displaying differences between the two revisions with the number of lines of context specified in the Lines text field.
  - Side by Side. The output is a side-by-side comparison of the files.
  - Unified. Uses the unified format, which is similar to the context format but omits redundant lines of context.
- Ignore blank lines. Ignore changes that insert or remove blank lines.
- Ignore whitespace. Ignore whitespace when comparing lines.

Committing Changes

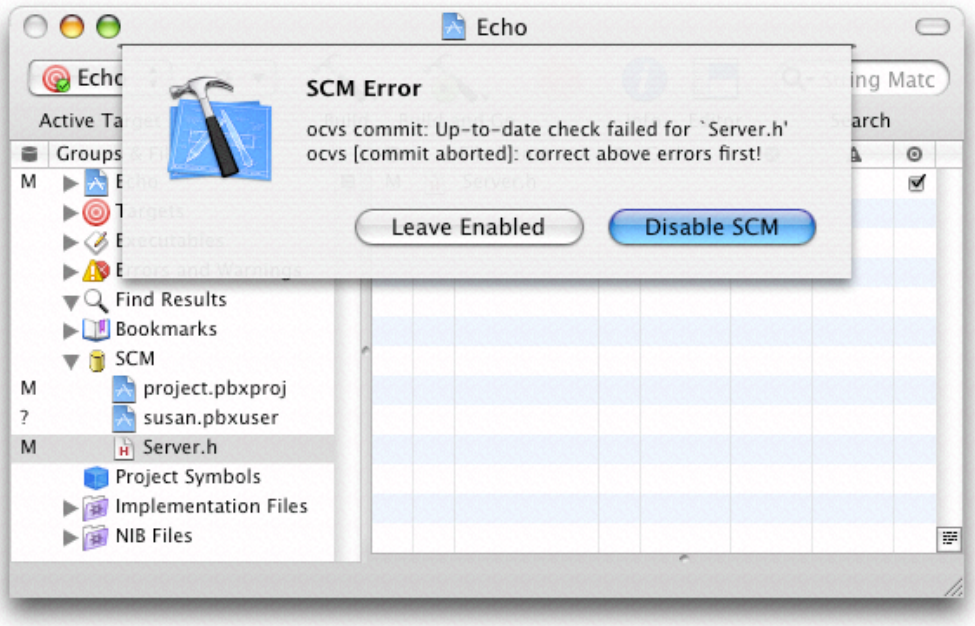
When you're done making changes to a file and you want to submit your modifications to the repository, you can tell Xcode to commit the file in one of two ways:

- Select the file in the project window and choose SCM > Commit Changes.
- Choose SCM > Commit Entire Project. This commits changes in every file listed in the SCM group with the status A, M, or R.

Both actions bring up a dialog with a text field in which you enter a message describing your changes. To execute the command, click Commit.

If your client tool encounters a problem during the commit process (for example, a file to be processed is outdated), Xcode displays a dialog showing your client's error message, as shown in Figure 22-15.

Figure 22-15 Dialog indicating that changes cannot be committed because there are files that need to be updated



You must correct the problem by, for example, updating any files with a status of U, before you can commit your changes.

Resolving Conflicts

A file with a status of C contains changes that clash with the latest revision in the repository. For example, you may have removed a method from a class definition that another developer published changes to before you had a chance to commit your own changes. To view how your version of a file in conflict differs from the latest revision, use the Compare or Diff commands. See "Comparing Revisions" (page 219) for details.

Version control systems cannot resolve conflicts. They can only make you aware of the presence of conflicts. In some cases, you may be able to resolve the conflict yourself. However, in the majority of cases (if you work in a team), you need to communicate with the person who published the changes that conflict with your own before determining the best way to resolve the conflict.

There are two ways of resolving a conflict between your version of a file and the latest revision in the repository:

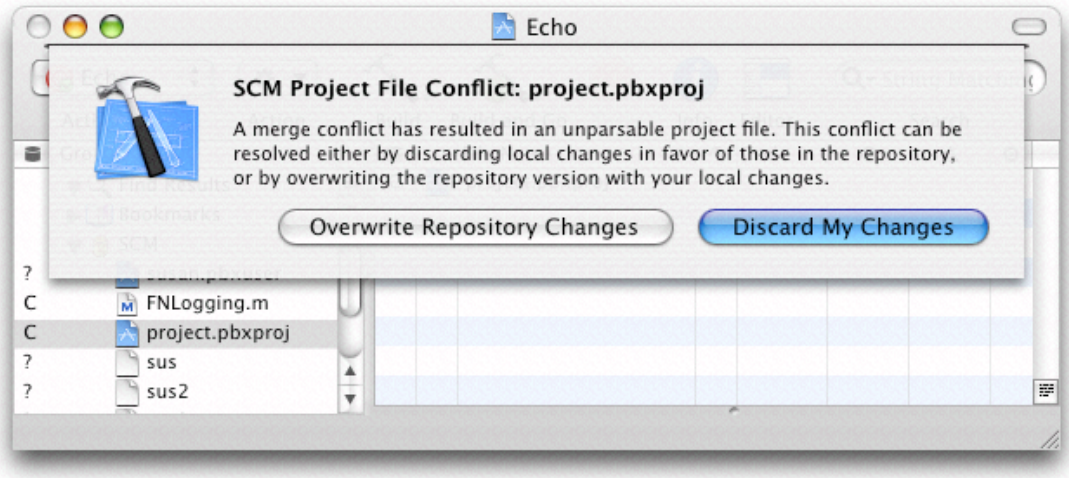
- Merge the changes published to the repository with your local changes and edit the resulting file as necessary:
  1. In the project window, select the file with the conflict.
  2. Choose SCM > Update To > Latest.
  3. Edit the file to resolve the conflicts.
  4. Save the file. If you're using Subversion, you must also choose SCM > Resolved.

- Discard your copy of the file in favor of the latest revision in the repository:

  Select the file in the project window and choose SCM > Discard Changes.

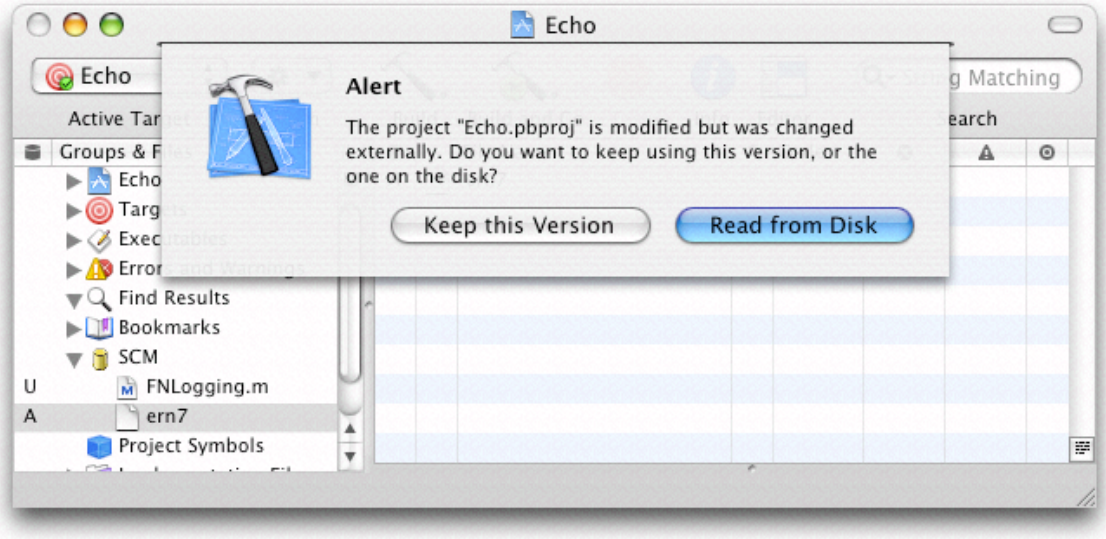
Conflicts in the project file (`project.pbxproj`) result from developers adding, removing, or renaming files from their local copies and committing those files without committing the project file at the same time. You cannot resolve conflicts in the project file. If your copy of the project file gets a status of C, choose SCM > Update Entire Project or select the project file in the project window and choose SCM > Update To > Latest. You may get a dialog like the one in Figure 22-16.

Figure 22-16 Unable to save project dialog



Click Discard My Changes to discard your modifications to the project file and use the latest version in the repository. If the dialog in Figure 22-17 appears, click Read From Disk to force Xcode to re-read the project file from disk (which is the latest version in the repository).

Figure 22-17 Dialog indicating that the project file has been changed by an application other than Xcode



If Xcode is unable to open your project due to a corrupted project file, you must use your client tool to update the project file to the latest version in the repository. See "Updating a Local Project File to the Latest Version in a CVS Repository" (page 430) and "Updating the Project File to the Latest Version in a Subversion Repository" (page 434) for more information.

Development Workflow

When working on a managed project, you have two major objectives: To maintain you local copy of the project up to date with the latest version in the repository and to keep the repository up to date with your changes. If you update your working copy regularly, you reduce the probability that the changes you make conflict with the changes other members of your team have published to the repository.

This section describes the workflow you should follow when working on a managed project with Xcode that allows you to keep your local copy of it up to date with the latest published changes and to submit your changes to the repository as appropriate to keep your teammates abreast of your work.

When working on a managed project, you should perform the following tasks on a regular basis—hourly, daily, weekly, or as convenient—depending on your team's needs and requirements:

- Update your local copy with the latest version in the repository.
- Make changes to your local copy of the project.
- Resolve conflicts.
- Publish your changes.

Update Your Local Copy

Before you start making changes to a project, you should make sure your local copy contains the latest changes your peers have published.

To determine whether your project is up to date, choose Refresh Entire Project from Xcode's SCM menu. You need to update any files that have a status of U. Files with a status of C are in conflict. You should examine the files in conflict before deciding whether to update them. See "Updating Files" (page 214) and "Resolving Conflicts" (page 224) for details.

Make Changes

To modify files all you need to do is edit them. (If you're using a version control system that locks files in your local copy, you need to choose SCM > Edit before you can save your changes to disk.)

Making structural changes to a project, such as adding or removing files, creating or removing groups in the Groups & Files list, and configuring build settings produce changes in the project file (`project.pbxproj`). You must commit the project file after making structural changes to a project, along with the files you operated on.

In general, you should make structural changes to a project only after updating it from the repository. You should then commit those operations immediately, along with the project file, before making additional modifications. Then inform the rest of your team so that they update their project files to the new version. This procedure helps reduce conflicts between your teammates' copies of the project file and the latest version in the repository, as well as keep the contents of their local project directories and their corresponding project files in sync. See "Project Packages" (page 205), "Updating Files" (page 214), and "Committing Changes" (page 223) for details.

Resolve Conflicts

Before you commit your changes to the repository, refresh the status of the project to determine whether conflicts exist. In Xcode, choose SCM > Refresh Entire Project. If the SCM group in the Groups & Files list doesn't contain files with a status of C, you can commit your changes. Otherwise, you must resolve the conflicts. See "Resolving Conflicts" (page 224) for details.

Publish Your Changes

After you've confirmed that your local modifications don't conflict with published changes, you may commit them to the repository. See "Committing Changes" (page 223) for more information.

The Build System

The build system is the part of the Xcode that is responsible for transforming the components of a project into one or more finished products. The build system takes a number of inputs and performs operations such as compiling, linking, copying files and so forth to produce an output—usually an application or other type software.

Xcode includes a powerful build system, that can create a wide variety of Mac OS X products, such as frameworks, libraries, applications, command-line tools and more. Using Xcode's predefined project and target templates you can build these products right out of the box. However, the Xcode build system is also flexible enough to allow you to customize the build process, to tailor it to meet the special needs of your projects or support your preferred workflow.

The following chapters show you how to build a product in Xcode, describe targets and the build system inputs that they organize, and show you how you can take advantage of build phases, build settings, and build styles to customize the build process. These chapters also describe a number of Xcode features that you can take advantage of to shorten the edit-build-debug cycle, such as distributed builds, precompiled prefix headers, and predictive compilation.

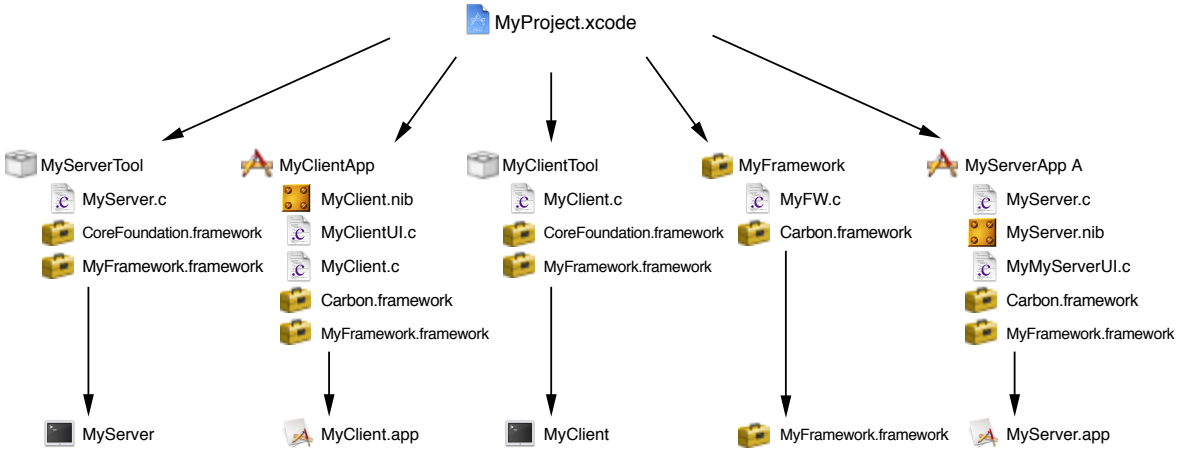
Targets

The organizing principle of the Xcode build system is the target. A **target** contains the instructions for building a finished product from a set of files in your project. Some common types of products are frameworks, libraries, applications, and command-line tools. Each target builds a single product. A simple Xcode project has just one target, which produces one product from the project's files. A larger development effort with multiple products may require a more complex project containing several related targets. For example, a project for a client-server software package may contain targets that create these products:

- A client application
- A server application
- Command-line tool versions of the client and server functionality
- A private framework that all the other targets use

Figure 23-1 shows the targets you may have in a project such as the one described above, and the products that those targets create.

Figure 23-1 Targets and products



When you initiate a build, Xcode builds the product specified by the current, or active, target and any targets on which the active target depends. In the Groups & Files list, the active target is marked by a checkmark in a round green circle. You can also see which target is active, as well as change the active target, in the Active Target pop-up menu in the project and Build Results windows, as described in "Building a Product" (page 301).

This chapter describes targets in Xcode and the information that they contain, and shows you how to create and modify targets.

Anatomy of a Target

A target is a blueprint for creating a product; the target organizes the inputs to the build system process—the source files and the instructions for processing them—required to create that product. These inputs are:

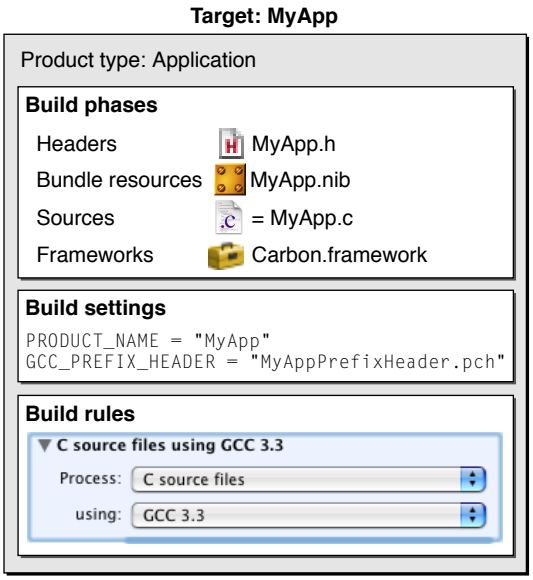
- **Files**. Each target contains a list of files that the build system takes as inputs and performs various operations on—such as compiling, linking, copying, and so forth—to arrive at the final product. Examples of source files are source code files, headers, resource files, and so forth.
- **Build phases**. Build phases organize the source files of a target according to the operations required to build a target's product. Each build phase consists of a list of input files and a task to be performed on each of those input files. Common build phases include compiling files, linking object files, and copying resource files.

  Xcode populates each new target with a default set of build phases. You can add or remove build phases to change the operations performed by Xcode when building the target. For further details on build phases, see "Build Phases" (page 249).
- **Build settings**. A build setting contains information on how to perform the operations required to build a product. For example, a build setting can specify command-line options for Xcode to pass to the compiler. Build settings can contain tool-specific options, paths to build files and product directories, and other information used by Xcode to determine how to perform build operations.

  Each target contains a list of build settings. Although you will most likely make most of your build setting changes at the target level, build settings can be modified in a number of other locations, such as build styles, as well. By modifying build settings at these other layers, you can quickly make a change and apply it to multiple targets, conditionally change the way a product is built, and so forth.

  Because build settings represent variable aspects of the build process, they are the most flexible means of customizing the build process. See "Build Settings" (page 267) for more information on using build settings.
- **Build rules**. A target's build rules determine how each source file in the target is processed. Each build rule consists of a condition—such as files matching the type ".c"—and an action. Typically, this action specifies the tool Xcode should invoke to process files that meet the condition; this is how Xcode determines the compiler to use for compiling source files. Build rules apply only to the Compile Sources and Build ResourceManager Resources build phases. Xcode defines a default set of build rules, but you can define your own custom build rules for a target. For more information on using build rules, see "Build Rules" (page 261).

Figure 23-2 A target



A target and the product that it creates are closely related; every target has an associated product type. When you create a new target from a target or project template, you choose the target's product type, as described in "Creating a New Target" (page 233).

Based on the product type, Xcode specifies initial values for certain product-specific build settings. For example, when you create a target that builds an application, Xcode assigns it the build setting specification INSTALL_PATH = `"/Applications"` based on the product type. Any subsequent changes you make to the target after creating it may override these default values. Note that the project and target templates contain additional configuration information that Xcode uses when it creates new targets.

Creating Targets

When you create a new project from one of the Xcode project templates, Xcode automatically creates a target for you. If, however, your project needs to contain more than one target—usually because you are creating more than one product—you can also add new targets to an existing project. This section shows you how to:

- Create a new target.
- Duplicate an existing target.
- Remove unused targets from the project.

Creating a New Target

If you are adding new targets to your project, chances are you've already made a number of decisions about product type, programming language, and framework. Xcode provides a number of target templates to support your choices. The selection of target templates is similar to the selection of project templates. The

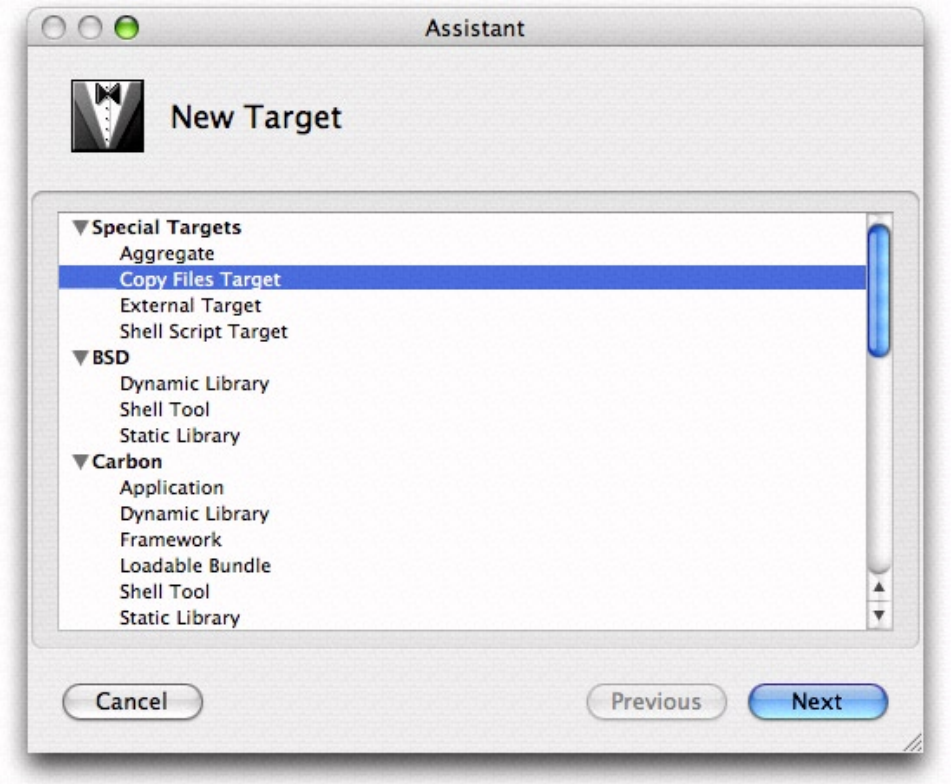
target specifies the target's product type, a list of default build phases, and default specifications for some build settings. A target template typically includes all build settings and build phases required to build a basic instance of the specified product. Unlike the project templates provided by Xcode, the target templates do not specify any default files; you must add files to the target yourself, as described in "Working with Files in a Target" (page 241).

You can create a new target and add it to an existing project in either of the following ways:

- Choose Project > New Target.
- Control-click in the Groups & Files list and choose Add > New Target from the contextual menu.

Xcode presents you with the New Target Assistant, shown here, which lets you choose from a number of possible target templates. Each target template corresponds to a particular type of product, such as an application or loadable bundle. Select one of these templates and click Next. Enter the name of the target; if more than one project is open, you can choose which project to add the new target to from the Add to Project menu. When you click Finish, Xcode creates a new target configured for the specified product type. Xcode also creates a reference to the target's product and places it in your project, although the product does not exist on disk until you build the target.

Figure 23-3 The New Target Assistant



Xcode provides the target templates listed in the following tables. Table 23-1 lists templates that create targets using Xcode's native build system. Because it performs all target and file-level dependency analysis for targets using the native build system, Xcode can offer detailed feedback about the build process and integration with the user interface for these targets.

Table 23-1 Xcode target templates

Target template	Creates
BSD	
Dynamic Library	A dynamic library, written in C, that makes use of BSD.
Shell Tool	A command-line utility, written in C.
Static Library	A static library, written in C, that makes use of BSD.
Carbon	
Application	An application, written in C or C++, that links against the Carbon framework.
Dynamic Library	A dynamic library that links against the Carbon framework.
Framework	A framework based on the Carbon framework.
Loadable Bundle	A bundle, such as a plug-in, that can be loaded into a running program.
Shell Tool	A command-line utility based on the Carbon framework.
Static Library	A static library, written in C or C++, based on the Carbon framework.
Cocoa	
Application	An application, written in Objective-C or Objective-C++, that links against the Cocoa framework.
Dynamic Library	A dynamic library that links against the Cocoa framework.
Framework	A framework based on the Cocoa framework.
Loadable Bundle	A bundle, such as a plug-in, that can be loaded into a running program.
Shell Tool	A command-line utility based on the Cocoa framework.
Static Library	A static library, written in Objective-C or Objective-C++, based on the Cocoa framework.
Java	
Application	An application, written in Java, and packaged as an application bundle.
Applet	A Java applet, built as a JAR file.
Tool	A command-line utility, written in Java and built as a JAR file.
Kernel Extension	
Generic Kernel Extension	A kernel extension
IOKit Driver	A device driver that uses the I/O Kit.

Xcode also supports targets that use Project Builder's Jam-based build system. Table 23-2 lists the templates that create targets using the Project Builder build system, called legacy targets. Note that Jam-based targets are supported only for compatibility with existing Project Builder projects. Where possible, you should use native targets instead. Legacy targets are discussed further in "Converting a Project Builder Target" (page 247).

Table 23-2 Legacy target templates

Template	Creates
Application	An application bundle.
Bundle	A standard bundle.
Cocoa	
Application	An application, written in Objective-C or Objective-C++, that links against the Cocoa framework
Framework	A framework.
Java	
Application	An application, written in Java, and packaged as an application bundle.
Applet	A Java applet, built as a JAR file.
Package	
Tool	A command-line utility, written in Java and built as a JAR file.
Kernel Extension	
Generic Kernel Extension	A kernel extension
IOKit Driver	A device driver that uses the I/O Kit.
Library	A dynamic or static library
Tool	A command-line utility.

In addition to the target templates described above, Xcode defines a handful of target templates that do not necessarily correspond to a particular product type. These targets are described in the next section.

Special Types of Targets

Xcode defines a handful of target templates that do not necessarily correspond to a particular product type. Instead, these targets can be used to:

- Build a group of targets together
- Build a product using an external build system
- Run a shell script

- Copy files to a specific location in the filesystem

Aggregate

Xcode defines a special type of target that lets you build a group of targets at once, even if those targets do not depend on each other. An **aggregate target** does not have an associated product and it does not contain build information, such as build rules. An aggregate target depends on each of the targets you want to build together. For example, you may have a suite of applications and want to build them at once for a final build. You would create an aggregate target and make it depend on each of the individual application targets; to build the entire application suite, just build the aggregate target.

An aggregate target may contain a custom Run Script build phase or a Copy Files build phase, but it does not contain any other build phases.

External

Xcode allows you to create targets that do not use Xcode's own native build system but instead use an external build tool that you specify. For example, if you have an existing project with a makefile, you can use an external target to run `make` and build the product.

An external target creates a product but does not contain build phases. Instead, it calls a build tool in a directory. With an external target, you can take full advantage of Xcode's editor, class browser, and source-level debugger. However, many Xcode features—such as ZeroLink and Fix and Continue—rely on the build information maintained by Xcode for targets using the native build system. As a result, these are not available to an external target. Furthermore, you must maintain your custom build system yourself. For instance, if you need to add files to an external target built using `make`, you must edit the makefile yourself.

Shell Script

A Shell Script target is an aggregate target that contains only one build phase, a Run Script build phase. Building a Shell Script target simply runs the associated shell script. Shell Script targets are useful if you have custom build steps that need to be performed. While Run Script build phases allow you to add custom steps to the build process for a single target, a Shell Script target lets you define a custom build operation that you can use with many different targets. For example, if your project has several targets that each use the files generated by a Shell Script target, you can make each of those targets depend upon the Shell Script target.

Copy Files

A Copy Files target is an aggregate target that contains only one build phase, a Copy Files build phase. Building a Copy Files target simply copies the associated files to the specified destination in the filesystem. Copy Files targets are useful if you have custom build steps that require files that are not specific to any other targets to be copied. While Copy Files build phases allow you to add a step to the build process for a single target that copies files in that target, a Copy Files target lets you copy files that are not specific to any one target. For example, if your project has several targets that require the same files to be installed at a particular location, you can use a Copy Files target to copy the files, and make each of the other targets depend upon the Copy Files target.

Duplicating a Target

There are a number of reasons why you might need to duplicate a target: you require two targets that are very similar but contain slight differences in the files or build phases that they include, or you have a complicated set of options that you build with and would prefer to simply start with a copy of a target that already contains those build settings.

Xcode allows you to duplicate a target, creating a copy that contains all of the same files, build phases, dependencies and build settings. You can create a copy of a target in the following way:

1. In the Groups & Files list, select the target you which to copy.
2. Choose Edit > Duplicate or Control-click and choose Duplicate from the contextual menu.

Removing a Target

If your project contains targets that are no longer in use, you can remove them from the project in the following ways:

1. Select the target to delete in the Groups & Files list.
2. Press the Delete key or choose Edit > Delete.

When you delete a target, Xcode also deletes the product reference for the product created by that target and removes any dependencies on the deleted target.

Target Dependencies

In a complex project, you may have several targets that create a number of related products. Frequently, these targets need to be built in a specified order. Returning to the example of the client-server software package created by the project shown in Figure 23-1 (page 231), you'll see that the client application, server application, and command-line tool targets each link to the private framework created by another target in the same project.

Before the application and command-line tool targets can be built, the framework target must be built. Because they require the private framework in order to build, each of the application and command-line tool targets is said to depend upon the target that creates the framework. You can use a **target dependency** to ensure that Xcode builds targets in the proper order; in this example, you would add a dependency upon the framework target to each of the application and command-line tool targets.

However, the applications and command-line tool in the client-server package must still be built individually. None of these targets requires the product created by any target other than the framework target. Xcode provides another mechanism for grouping targets that you want to build together, but that are otherwise unrelated; this is an aggregate target. The following sections show you how to add a target dependency and create an aggregate target, and gives an example of how you can use these tools to organize a software development effort with multiple products and projects.

Adding a Target Dependency

When you build a target with a dependency upon another target, Xcode makes sure that this other target is built and up-to-date before building the active target. That way, you can guarantee that when target A needs the product created by target B, target B is built before target A. In addition, if there are errors building target B, Xcode doesn't build target A.

You can view and modify a target's dependencies:

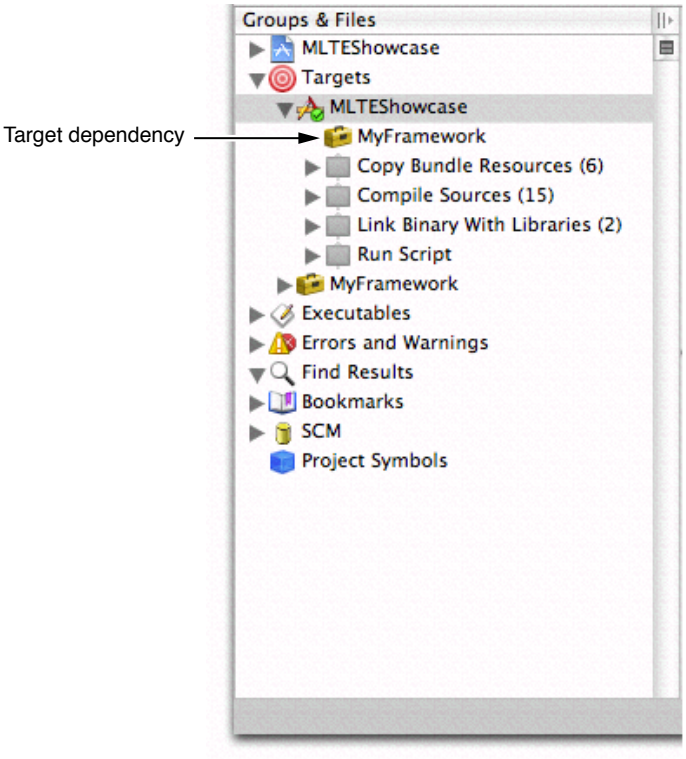
- In the General pane of the target inspector. The Direct Dependencies list in the bottom half of the General pane shows the other targets upon which the current target depends.

  To add a target dependency, click the plus sign button and select the target upon which you want the current target to depend in the resulting dialog. The list of targets includes all of the other targets in the current project, as well as the targets in any referenced projects. Targets in referenced projects are grouped according to the project to which they belong; click the disclosure triangle next to the project icon to see the targets in that project. For more information on referencing other projects, see "Referencing Other Projects" (page 81).

  To remove a target dependency, select it in the list and click the minus button.

- In the Groups & Files list. Click the disclosure triangle next to the target; the targets that the current target depends on are listed before the build phases in the target. You can make the current target depend on another target by dragging that target to the current target in the Groups & Files list. To delete a target dependency, select it in the Groups & Files list and press Delete or choose Edit > Delete. Figure 23-4 shows a target dependency in the Groups & Files list.

Figure 23-4 A target dependency in the Groups & Files list



Creating an Aggregate Target

To build several related targets at the same time, even if they aren't dependent on each other, create an aggregate target. As described in "Special Types of Targets" (page 236), an aggregate target does not produce a product itself and it does not contain build rules or information property list entries. Instead it exists so that you can make it dependent on other targets. To build a group of related targets, just build the aggregate target.

To create an aggregate target, choose Project > New Target and select Aggregate from the New Target Assistant. For each target you want to build with this aggregate target, add a target dependency to the aggregate target, as described in the "Adding a Target Dependency" (page 239).

Note that, while it does not contain any other build phases, an aggregate target can include a Run Script or Copy Files build phase.

An Example With Multiple Targets and Projects

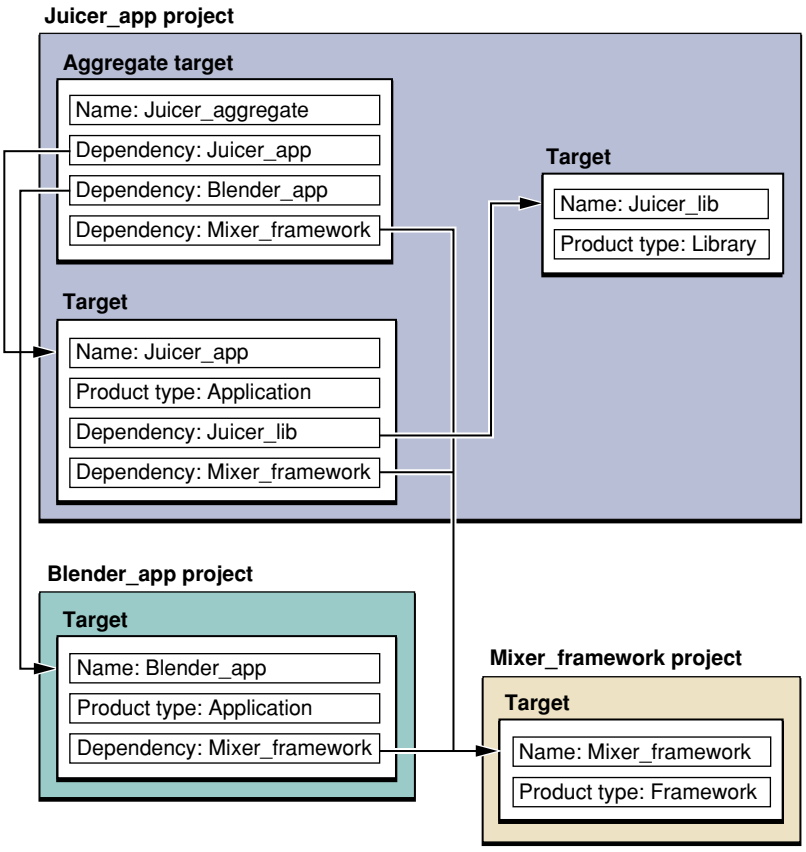
Suppose that your company has two teams working on separate applications, that one application includes an internal library, and that each application relies on a framework supplied by a third team. Figure 23-5 shows one way to set up your software development with three Xcode projects, using target dependencies, aggregate targets, and cross-project dependencies to relate the various products.

In Figure 23-5, the `Juicer_app` project contains the `Juicer_app` target, for building the Juicer application, and the `Juicer_lib` target, for building an internal library. The application target depends on the library target, and also has a cross-project dependency on the `Mixer_framework` target in the `Mixer_framework` project. Finally, the `Juicer_app` project contains the `Juicer_aggregate` target, as a convenience for building the entire suite of projects.

Note: Aggregate targets are typically used to build targets that don't otherwise depend on each other.

In Figure 23-5, the `Blender_app` project contains a target for building the Blender application. The Blender target also has a cross-project dependency on the Mixer framework.

Figure 23-5 Three projects with dependencies



Finally, the `Mixer_framework` project contains a target for building the Mixer framework, used by both the Juicer and Blender applications.

Given this combination of projects, targets, and dependencies, the following statements are true:

- Building the Juicer target builds the Juicer library, if it needs updating, and also builds the Mixer framework, if it needs updating.
- Building the Juicer aggregate target builds the Juicer application, which builds the Juicer library, if it needs updating. The aggregate target also builds the Blender application and the Mixer framework.
- Building the Juicer library does not cause any other targets to be built.
- Building the Blender target builds the Mixer framework, if it needs updating.

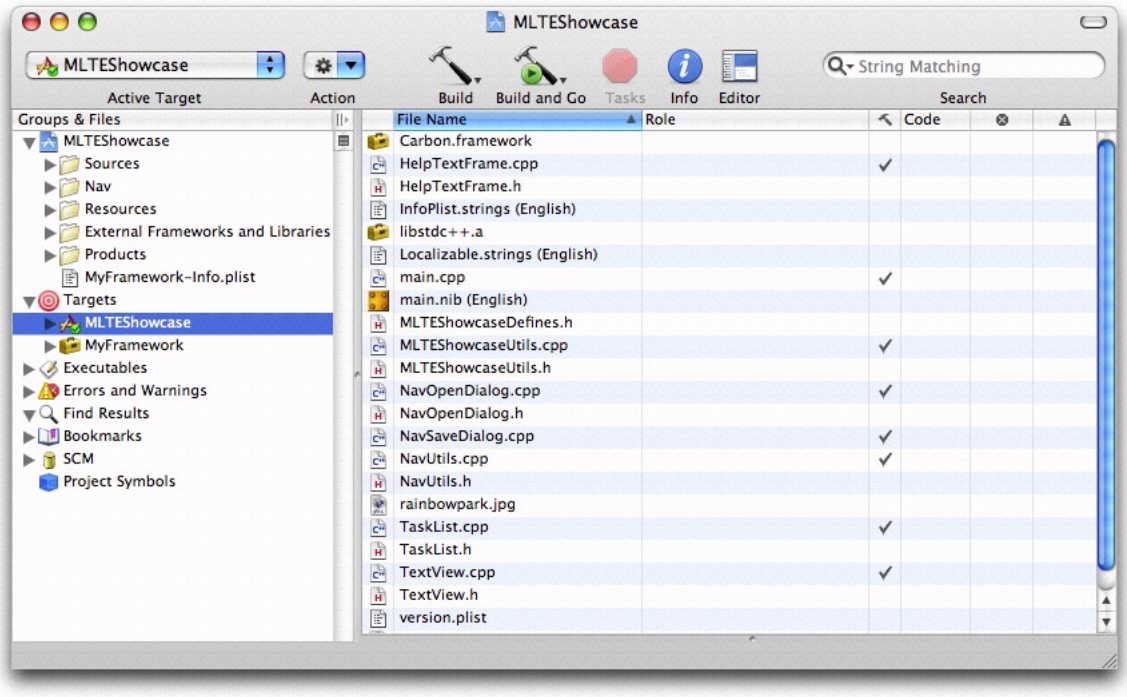
Working with Files in a Target

When you add files to a project, you can have Xcode also add those files to one or more targets. When you add files to a target in this way, Xcode automatically assigns them to the appropriate build phase, based on the file's type. This is the easiest way to add files to a target. However, you may have existing files in your project that you wish to add to a target, or find that you no longer need a target file. This section describes how to view the files in a target and shows you how to add and remove target files.

Viewing the Files in a Target

To view all of the files included in a target, select the target in the Groups & Files list; all targets are organized into the Targets smart group. The detail view shows all of the files and folders in the target, similar to what you see in the following figure. If one is not already open in the current window, open a detail view by choosing View > Detail.

Figure 23-6 Viewing targets in the project window



You can also see a target's files grouped according to the operation performed on those files during the build process. If you click the disclosure triangle next to a target, you will see the build phases for that target. Build phases, described in further detail in "Build Phases" (page 249), represent a task performed when the target is built. To see the files in a particular build phase, click the disclosure triangle next to the build phase or select the build phase and open a detail view.

Adding and Removing Target Files

You can change which files are included in a target from the project window. To add files to a target, you can:

- Drag the file reference or references to the appropriate build phase of the given target in the Groups & Files list. For native targets, Xcode does not let you drag a file to a build phase that does not accept that type of file as an input. For example, you cannot drag a NIB file to the Compile Sources build phase. See "Build Phases" (page 249) for more information on the available build phases and their files.
- Specify that a file be included in a target when you add that file to your project, as described in "Adding Files, Frameworks, and Folders to a Project" (page 78).

- To add a file to the active target, find the file in the detail view or in the Groups & Files list and select the checkbox in the Target column for that file. If the Target column is not visible, choose View > Detail View Columns > Target or View > Groups & Files Columns > Target.

To remove a file from a target, do the following in the Groups & Files list:

1. Click the disclosure triangle next to the target to reveal the target's build phases.
2. Click the disclosure next to the build phase to which the file belongs.
3. Select the file to remove from the target and click Delete or choose Edit > Delete.

To remove a file from the active target, find the file in the detail view or in the Groups & Files list and deselect the checkbox in the Target column for that file. Xcode removes the file from the target, but does not remove the file reference from the project.

Inspecting Targets

Like most other project items, targets have inspector and Info windows that allow you to view and modify target settings. As previously described in "Anatomy of a Target" (page 232), a target defines the instructions necessary to create a product. Each target has an associated set of files, tasks and settings that together constitute these instructions. You can edit and inspect many of these settings in the target inspector.

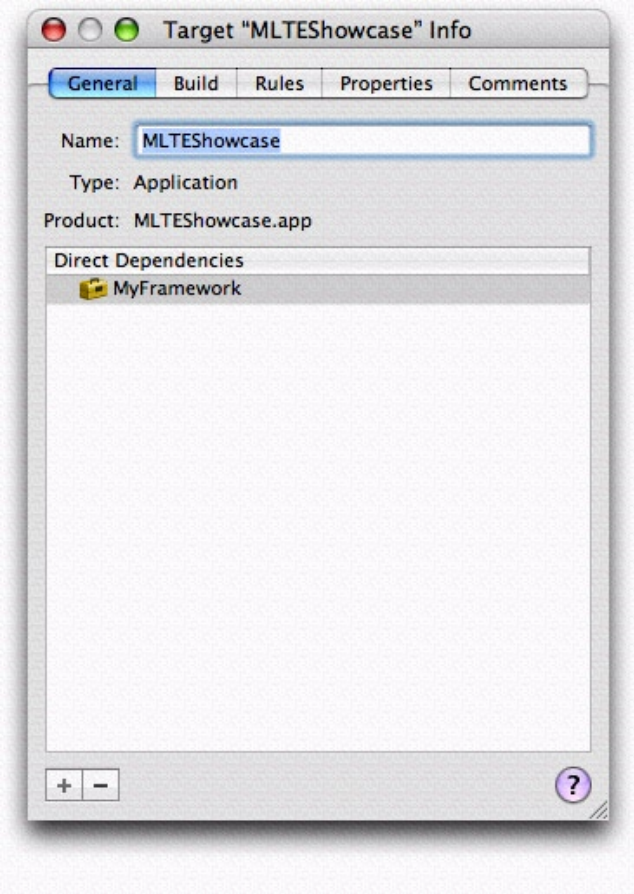
Note: To learn how to modify the files and build phases associated with a target, see "Working with Files in a Target" (page 241) and "Build Phases" (page 249), respectively.

The information available in the inspector window varies, depending on the type of target. The inspector window for native targets lets you view and edit all target settings and build information associated with that target. The inspector window for legacy and external targets contains only general information. To edit build information for either of these types of targets, you must use the target editor.

Inspecting Native Targets

The inspector window is the primary means of viewing and editing the information in native targets. Figure 23-7 shows the inspector window for a native target.

Figure 23-7 The Info window for a native target



This window contains the following panes:

- The General pane contains general information about a target, such as its name, the name of the associated product, and target dependencies. The General pane is described in "Editing General Target Settings" (page 245).
- The Build pane lets you view and edit build settings for the target. The Build pane is described in "Editing Build Settings in the Xcode Application" (page 277).
- The Rules pane displays the current system build rules, as well as any custom build rules defined for the current target. The Rules pane is described in "Build Rules" (page 261).
- The Properties pane lets you edit information property list entries for targets that create products requiring Info.plist files, such as applications and other bundles. If the target does not have an Info.plist file, this pane is not visible in the inspector. This pane is described in "Editing Information Property List Entries" (page 245).
- Comments. The Comments pane lets you associate notes or other documentation with the target. See "Adding Comments to Project Items" (page 92) for more information.

Inspecting Legacy and External Targets

You can view and edit only general target information in the inspector window for Jam-based Project Builder targets and external targets. You cannot configure build information for these targets. The inspector window for legacy and external targets contains the following panes:

- The General pane contains general information about a target, such as its name, the name of the associated product, and target dependencies. The General pane is described in "Editing General Target Settings" (page 245).
- Comments. The Comments pane lets you associate notes or other documentation with the target. See "Adding Comments to Project Items" (page 92) for more information.

If you have an existing Project Builder project with a Jam-based target, you can convert your target to use the Xcode native build system to edit build settings or build rules for the target in an inspector or Info window.

To edit the build settings for Jam-based and external targets in Xcode, use the target editor, described in "Editing Build Settings for Legacy and External Targets" (page 283).

Editing General Target Settings

In the General pane of the target inspector, you can edit basic target settings for any target in Xcode. The General pane contains the following settings:

- Name. This is the name Xcode uses to refer to the target. To rename the target, click in the text field and type the new name.
- Type. This is type of product created by the target. The product type is determined when you create the target; you cannot change it.
- Product. The name of the product created from the target; for example, "MyApp.app." The Product Name (PRODUCT_NAME) build setting specifies the product name, excluding any extension; in this example, "MyApp."By default, this is the same as the target name. To change the name of the product, you can change the name of the target or customize the Product Name build setting.

  To change the product extension—in this case, "app"—change the value of the Wrapper Extension (WRAPPER_EXTENSION) build setting.
- Dependencies. A list of the targets upon which the current target depends. See "Adding a Target Dependency" (page 239).

Editing Information Property List Entries

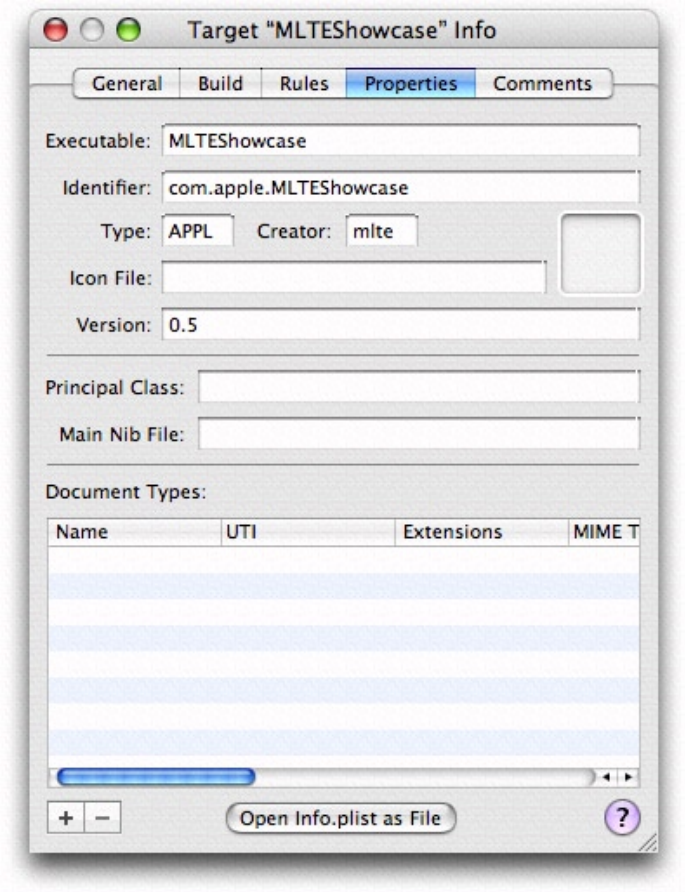
Information property list entries contain information used by the Finder and system software. This information ends up in a file called `Info.plist` that's contained within the product's bundle. If a product does not come in the form of a bundle, it has no `Info.plist` file.

The `Info.plist` file tells the Finder what the bundle's icon is, what documents it can open, what URLs it can handle, and so on. Unlike build settings, property list entries do not affect the build process but are simply copied directly into the bundle's `Info.plist` file at the end of the build process. For a list of entries used by the system and Finder, see *Runtime Configuration Guidelines*.

For Native Targets

The Properties pane of the target inspector, shown below, allows you to edit information property list entries for native targets. This pane is only visible for targets that create products with Info.plist files.

Figure 23-8 The Properties pane of the target inspector window



The top section of the pane allows you to edit basic information about the product, such as the name of the associated executable, the identifier, type and creator, version information, and an icon to associate with the finished product. Note that the name of the icon here must match the name of an icon file that is copied into the Resources folder of the product bundle.

The Principal Class and Main Nib File options are specific to Cocoa applications and bundles. The Principal Class corresponds to `NSPrincipalClass`. The Main Nib File field specifies the nib file that's automatically loaded when the application is launched. It corresponds to the information property list key `NSMainNibFile`.

The Document Types table allows you to specify which documents your finished product can handle. You can add and remove document types from this list using the plus and minus buttons. Here is what's in the Document Types table:

- Name is the name of the document type. For example, "Apple Sketch Document."
- Class is the subclass of NSDocument that this document uses. Use this field only if you're writing a document-based Cocoa application.

- Extensions contains a list of the filename extensions for this document type. Don't include the period in the extension. For example, "sketch" and "draw2".
- UTI contains a list of Uniform Type Identifiers (or UTIs) for the document. UTIs are strings that uniquely identify abstract types. They can be used to describe a file format or data type, but can also be used to describe type information for other sorts of entities, such as directories, volumes, or packages. For more information on UTIs, see the header file `UTType.h`, available as part of LaunchServices.framework in Mac OS X v. 10.3 and later.
- MIME Types contains a list of the MIME types for the document.
- OS Types contains a list of four-letter codes for the document. These codes are stored in the documents' resources or information property list files. For example, "sktc".
- Icon File is the name of the file that contains the document type's icon.
- Role describes how the application uses the documents of this type. You can choose from three values:
  - Editor means this application can display, edit, and save documents of this type.
  - Viewer means this application can display, but not edit, documents of this type.
  - None means this application can neither display nor edit documents of this type but instead uses them in some other way. For example, the Finder could declare an icon for font documents.
- Package specifies whether the document is a single file or a file package.

To edit a document type, click the type's line in the Document Types list and edit the document type information.

You may have additional keys that you need to include in your `Info.plist` file; for example, applications that include Apple Help help books need two additional `Info.plist` entries. To add these additional keys, you can edit the `Info.plist` file directly, by clicking the Open Info.plist as File button at the bottom of the Properties pane. You can refer to build settings in the `Info.plist` file. For example, `$(PRODUCT_NAME)` expands to the base name of the product built by the target.

You can set properties on multiple targets; simply select the targets in the project window and open an Info or inspector window. In the Properties pane, you can edit the values of properties that apply to more than one target.

For Legacy Targets

You cannot configure Info.plist entries for Jam-based Project Builder targets in the target inspector. To edit the Info.plist entries for Jam-based targets in Xcode, select the target in the Groups & Files list. If you have an editor open in the project window, Xcode displays the Project Builder target editor. To view this target editor in a separate window, double-click the target. Select Info.plist Entries in the left side of the target editor; Xcode displays the target's Info.plist entries in the target editor.

Converting a Project Builder Target

Xcode supports existing Project Builder targets that use the Project Builder build system, based on the Jam software build tool. Based on the information it maintains for a Jam-based target—build phases and build settings—Xcode creates a text-based Jamfile for the target and then invokes the `jam` command to perform the build. You can continue to use your existing Project Builder targets, or even create new Jam-based targets.

Although Xcode recognizes and can build Project Builder targets using the Jam-based build system, there are many advantages to converting to "native" targets that use Xcode's native build system. One advantage is shorter build times, particularly for incremental builds. In addition, many of the new Xcode features—such as ZeroLink, and Fix and Continue—work only with native targets.

Xcode's native build system performs its own dependency analysis and directly invokes the necessary build commands. The native build system takes advantage of the detailed information that the Xcode application maintains about a project's targets and relationships to provide finer control over the build process and better feedback from the build system than you can obtain from using an external tool.

To convert all Jam-based targets in your project to the native build system, choose Project > Upgrade All Targets in Project to Native.

To upgrade a single target to use the native build system, select that target in the Groups & Files list and choose Project > Upgrade *target name* to Native.

Xcode creates a new copy of each target, configures it to create the appropriate product type, and adds it to the Targets group. Your existing Jam-based targets remain unchanged. Xcode preserves all target dependencies when upgrading targets to the native build system.

Xcode generates a log file that shows you the results of the upgrade to native targets. The information in this log file includes the product type created by the upgraded target, locations of common support files—such as Info.plist files, and changes made to any build settings.

Build Phases

In Xcode a build phase represents a task to be performed on a set of files. Each build phase specializes in a specific kind of file, such as header files, source files, resource files, and frameworks and libraries. Also, each build phase performs specific operations on the files associated with it. That way, changes in the way files of the same kind are processed can be made by customizing the operations that the build phase performs.

This chapter provides an overview of build phases, explains how Xcode determines the order in which build phases should be processed to build a product, describes in detail some of the build phases available, and explains how build rules allow you to customize the Compile Sources build phase and the Build ResourceManager Resources build phase in a target.

Overview of Build Phases

A build phase collects a group of files and performs a set of operations on them in the process of building a product. To illustrate this, think of the tasks you would take to prepare for and deliver a presentation; many of the required tasks contain a set of inputs and outputs:

1. Specify the purpose of the presentation and identify the audience.

   This task has no prerequisites but produces a set of criteria that serve as the guiding principles for the presentation.

2. Gather the appropriate data, such as articles, reports, and surveys.

   Information gathering requires a set of criteria that specifies what to look for and where to look for it. The outcome are a set of documents containing the data gathered.

3. Purchase a business suit.

   This task has no inputs or outputs that directly relate to the presentation.

4. Analyze the data by collating facts, extrapolating trends, and summarizing representative opinions.

   This task uses the data gathered in task 2 and produces one document containing the information on which the presentation is based

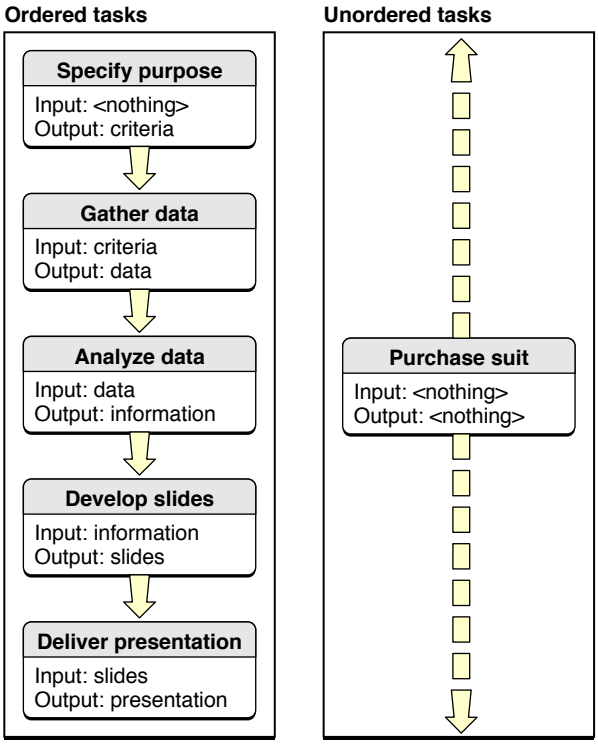
5. Develop a set of slides, including compelling illustrations, from the information produced by your analysis.

   This task requires the information produced in task 4 and produces a set of slides and illustrations, which could be presented using Keynote.

6. Deliver the presentation by showing the slides and illustrations in sequence and engaging the audience with a loud, clear voice and maintaining eye contact.

The order in which build phases are executed in a target depends on their inputs and outputs. Most of the tasks shown earlier contain a set of inputs and outputs that associate them with each other in an anterior/posterior relationship. Tasks that contain inputs that are the outputs of other tasks or outputs that are the inputs of other tasks are **ordered tasks**. These are tasks that are executed in the order determined by their inputs and outputs. **Unordered tasks**, on the other hand, are tasks with no inputs and outputs, tasks whose inputs are not the outputs of other tasks, or tasks whose outputs are not the inputs of other tasks. Figure 24-1 shows the presentation tasks shown earlier categorized as ordered and unordered.

Figure 24-1 Presentation tasks

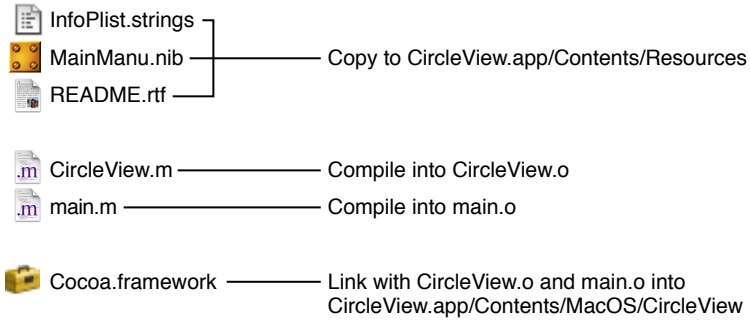


Most of the tasks illustrated depend on another task's outputs. The Specify Purpose task is the only task among the ordered tasks that doesn't depend on the output of any other task; therefore, it's performed first. The Purchase Suit task is the only unordered task; that is, it's neither the anterior nor the posterior of other tasks. Therefore, it can be executed at any time during the preparation of the presentation.

Build Phases in Xcode

To build an application, you typically compile source files and link them to system frameworks and libraries. Figure 24-2 shows some of the operations needed to build the CircleView application (the CircleView project is located in `/Developer/Examples/AppKit`).

Figure 24-2 Building an application

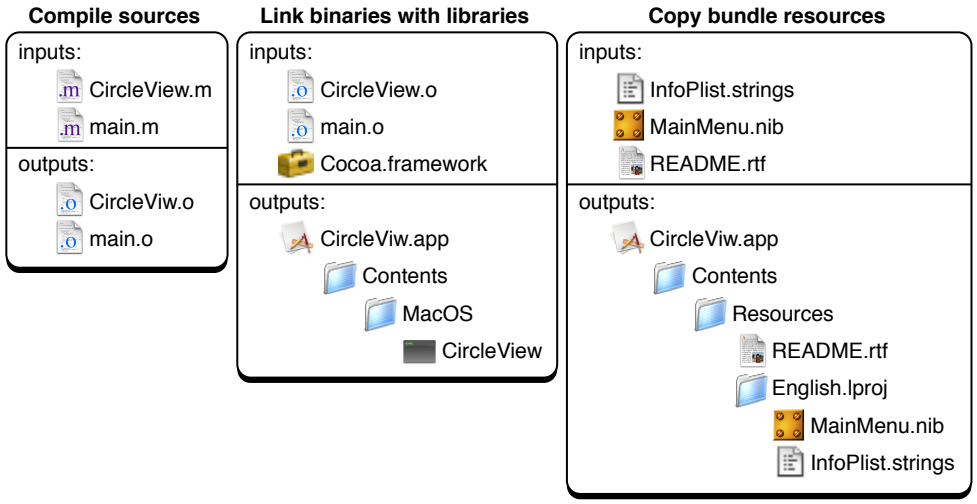


This simple project shows that there are many things to take into account when building a product. You have to copy resource files to the appropriate places, compile implementation files into object files, and link object files with the appropriate frameworks to produce a binary file. If you modify an implementation file, you would have to compile it, and link the generated object file with the other object files and the necessary frameworks.

If you analyze the files and the operations illustrated in Figure 24-2, you notice that the project contains three types of files: files that are copied or installed; files that are processed or compiled to produce intermediate files, such as object files; and frameworks that are linked with the object files to produce a binary file. To build a product, appropriate operations need to be performed on the files depending on their type.

Build phases associate groups of files with operations to be performed on them in order to build a product; for example, installing header files within frameworks, compiling source files, linking object files with system libraries or frameworks, and so on. As you add files to a project, Xcode associates them with the appropriate build phase, based on the files' type (specified by each file's extension). If you want to compile implementation files with a different compiler, you make the change once, at the build phase level (see "Compile Sources Build Phase" (page 256) and "Build Rules" (page 261) for details). Therefore, build phases help make the process of building an application, plug-in, library, or framework, understandable and easy to customize, as illustrated in Figure 24-3.

Figure 24-3 Building an application using build phases



A build phase operates on its inputs, which are the files associated with it (either by you or intrinsically by Xcode), and outputs, which are the files produced after the build phase is executed. Each build phase executes its task by invoking tools to perform the operations needed to accomplish the task.

Xcode offers several build phases, which are shown in Table 24-1. The name of the build phase reflects the task performed by that build phase.

Table 24-1 Build phases available in Xcode

Build phases	Description
Copy Headers	Installs header files with Public or Private roles in the appropriate locations in the product. In addition, header files associated to this build phase are added to the target's header map (`.hmap`) file in a way that makes it easier for the compiler to locate them; that is, you don't have to specify their path explicitly.
Copy Bundle Resources, Build Java Resources	Installs files by copying the associated files from the project directory to the appropriate locations in the product.
Copy Files	Installs files by copying the associated files from the project directory to a location in the product or to a specific location in the target file system, such as `/Library/Frameworks` or `/Library/Application Support`.
Compile Sources	Compiles source files into object files using a predefined tool, a tool you specify, or a build-rule script.
Run Script	Executes a shell script. You can use any scripting language whose scripts you can execute from the command-line, such as AppleScript, Perl, Python, and so on.
Link Binary With Libraries	Links object files with frameworks and libraries to produce a binary file.
Build ResourceManager Resources	Compiles `.r` files into resources that go into an application's resource fork.
Compile AppleScripts	Compiles `.applescript` files and places the resulting `.scpt` files in the product's `Contents/Resources/Scripts` directory.

Table 24-2 lists the build phases and their possible inputs and outputs.

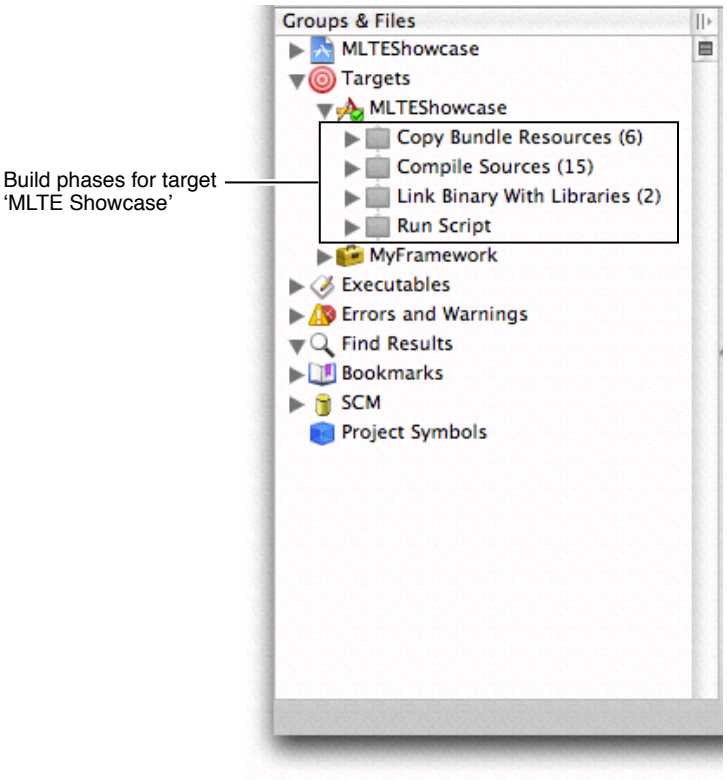
Table 24-2 Input files and output files of build phases

Build phase	Inputs	Outputs
Copy Headers	`.h` files	Copies in appropriate locations in the product.
Compile Sources	`.c`, `.m`, `.y`, `.l`, among other types of source files	`.o` files in target's build directory.
Link Binary With Libraries	`.framework`, `.dylib`, and the `.o` files in the target's build directory	Binary file, framework, or library in product destination directory.

Build phase	Inputs	Outputs
Copy Bundle Resources	`.nib` files, `.strings` files, image files, and others	Copies in the product's `Resources` directory.
Build ResourceManager Resources	`.r`, and `.rsrc` files	`Localized.rsrc` file in product's `Resources` directory.
Compile AppleScripts	`.applescript` files	`.scpt` files in the product's `Resources/Scripts` directory.

Each target has a set of build phases, separate from the build phases in other targets in your project. To view the build phases in a target, click the disclosure triangle next to the target in the Groups & Files list.

Figure 24-4 Viewing build phases



Adding and Deleting Build Phases

New targets—whether created through the New Target Assistant or as part of creating a new project—already include a set of default build phases. The default build phases for a target vary, depending on the type of product created by the target. For example, a target that builds an application typically includes a Copy Bundle Resources build phase to copy resources over to the application bundle; however, a target that builds

a shell tool does not include this build phase. The default build phases work for most simple targets; however, you can accommodate more complex products and build steps by adding your own build phases.To add a build phase:

- Select the target to add the build phase to; otherwise, Xcode adds the build phase to the active target.
- Choose a build phase from the Project > New Build Phase menu or from the Add > New Build Phase menu in the contextual menu that Xcode displays when you Control-click the target. Table 24-1 (page 252) shows the available build phases.

Xcode adds the new build phase after the currently selected build phase, or, if no build phase is selected, adds it as the last build phase in the target. Many types of build phases can only appear once in a target; for example, there can be only one Compile Sources build phase. If they already exist in the target, the menu items to add these build phases are dimmed. However, a target can contain multiple instances of the Copy Files and Run Script build phases.

To delete a build phase, select it in the Groups & Files list and press Delete or choose Edit > Delete. Deleting a build phase does delete the files in the build phase from the project or from the disk.

Adding Files to a Build Phase

When you add a file to a project, Xcode lets you choose whether to also add the file to any targets in the project. When you add files to a target in this way, Xcode automatically assigns the files to build phases, based on each file's type. To view the inputs to a particular build phase, you can do either of the following:

- Select the build phase in the Groups & Files list and, if necessary, open a detail view. The files assigned to the build phase are displayed in the detail view.
- Click the disclosure triangle next to the build phase in the Groups & Files list.

Intermediate files—files generated by Xcode in other build phases—are not listed. Xcode handles these intermediate files automatically.

If Xcode's default file placement is not sufficient for your needs, you can easily move files among build phases by dragging the file or files from their current build phase to the new build phase. You can also add files that are already in your project to a build phase by dragging the files from the project source group to the appropriate build phase in the Groups & Files list. To delete a file from a build phase, select the file and press Delete.

Processing Order

The order in which the build system executes build phases is important because some build phases produce files that are part of the inputs of other build phases. Therefore, the former must be executed before the latter. In native targets, dependencies between build phases determine the order of execution. In Jam-based targets, you must ensure that the build phases within a target are ordered appropriately.

In Native Targets

In some cases, a build phase inherently includes the outputs of other build phases as its inputs. For example, the outputs of the Compile Sources build phase (`.o` files) are part of the inputs of the Link Binary With Libraries build phase. This makes the Compile Sources build phase an antecedent of the Link Binary With Libraries build phase. Therefore, the order in which build phases are executed in a target depends on their inputs and outputs.

Most build phases have their inputs and outputs defined implicitly. However, Run Script build phases may have neither. In that case, the build system tries to run the associated script in the order specified within the target, but the actual point in the build process at which the script is run is undetermined. If you assign either input or output files to a Run Script build phase, the script's point of execution in the build process is determined by other targets having the build phase's outputs as their inputs, or the inputs of the build phase being the outputs of other build phases.

Figure 24-3 (page 251) illustrates this. Regardless of the order of the Compile Sources and the Link Binary With Libraries build phases in the target, the Compile Sources build phase is executed before the Link Binary With Libraries build phase because part the latter's inputs is constituted by the former's outputs.

In the Compile Sources build phase, the build system determines the order in which files are processed through the inputs and outputs of the target's build rules, in a similar way in which the order of build phases is determined.

In Jam-Based Targets

In Jam-based targets, the order of build phases within a target and the input files within the build phase determines the order in which the build system executes the build phases and processes each file within each build phase.

You must make sure that build phases that produce files required by other build phases are listed first within the target. For example, the Compile Sources build phase must always be listed before the Link Binary With Libraries build phase.

Within the Compile Sources build phase, you must ensure that source files that generate derived files are placed above dependent files. For example, if a target has Yacc (`.y`) and Lex files (`.l`) files and processing the Lex files requires the C (`.c`) files generated from the Yacc files, the Yacc files must be listed before the Lex files within the target.

Reordering Build Phases

Typically, there is no need to change the order of build phases in a target; the default arrangement works for the majority of cases. If, however, you find that you need to change the position of a build phase—for example, when adding custom build phases to a Jam-based target—you can reorder a build phase by dragging the build phase's icon to its new location in the target.

Compile Sources Build Phase

The Compile Sources build phase is one of the most customizable build phases (the other one being the Run Script build phase). The reason is that this build phase must handle a wide variety of input-file types. Xcode is preconfigured to process several types of source files, but you may have to compile source files that Xcode doesn't know about.

The feature that makes the Compile Sources build phase so flexible is its support of **build rules**. They specify the tool or script the build system invokes to process files in Compile Sources and Build ResourceManager Resources build phases when building a product. For details on build rules, see "Build Rules" (page 261).

Copy Files Build Phase

A Copy Files build phase allows you to copy and install files and resources of any type to specific locations as part of the build process. It complements the build phases that copy and install specific types of files, such as the Copy Headers build phase, which deals only with header files. You can have as many Copy Files build phases as you need in a target.

For example, using a Copy Files build phase, you can copy fonts to `/Library/Fonts`. Or, if you're developing a plug-in, a Copy Files build phase can copy the generated plug-in to the appropriate location. You can have as many Copy Files build phases in a target as you need.

To create a Copy Files build phase:

1. In the project window, click the disclosure triangle next to the target you want to add the build phase to and select the build phase after which to add the new build phase.
2. Choose Project > New Build Phase > New Copy Files Build Phase. Xcode adds the new Copy Files build phase after the build phase selected in the Groups & Files list.
3. Drag the files you want to copy from the Groups & Files list to the Copy Files build phase.

To configure the new Copy Files build phase, select it and open an Info or inspector window. You should see a window similar to the one shown here.

Figure 24-5 The Info window for a copy files build phase

Together, the Destination pop-up menu and the Path field specify the location to which Xcode copies the files in the Copy Files build phase. The Destination pop-up menu in the inspector window lets you choose from a number of standard locations. Table 24-3 shows the destination-location names you can choose in a Copy Files build phase for a framework called MyFramework and the resulting destination path. All the options, except Absolute Path and Products Directory, specify paths inside the generated bundle.

Table 24-3 Destination names and example destination paths of Copy Files build phases

Destination name	Destination path
Absolute Path	Anywhere.
Wrapper	`MyFramework.framework`
Executables	`MyFramework.framework/Versions/A/Resources`
Resources	`MyFramework.framework/Versions/A/Resources`
Java Resources	`MyFramework.framework/Versions/A/Resources/Java`
Frameworks	`MyFramework.framework/Versions/A/Frameworks`
Shared Frameworks	`MyFramework.framework/Versions/A/SharedFrameworks`

Destination name	Destination path
Shared Support	`MyFramework.framework/Versions/A/Resources`
Plug-ins	`MyFramework.framework/Versions/A/Resources`
Products Directory	The directory in which Xcode places the project's built products. See "Build Locations" (page 301).

The Path field specifies the path, relative to the location specified in the Destination menu, to the target directory. If you choose Absolute Path from the Destination pop-up menu, the Path field should contain the complete path to the destination directory for the files.

The "Copy only when installing" option lets you specify whether the build phase copies the files only in **install builds** of the product. That is, when using the `install` option of `xcodebuild` or when the Deployment Location (DEPLOYMENT_LOCATION) build setting is turned on. For more on `xcodebuild`, see "Building From the Command Line" (page 312).

Run Script Build Phase

A Run Script build phase lets you execute any commands you need to perform a task. You can archive files using `tar`, send mail, write messages to a log file, execute AppleScript scripts, and so on. You can use any of the shell languages available in your system. You can have any number of Run Script build phases in a target.

Before executing your script, Xcode assigns the values of most build settings to environment variables. In particular, it sets the environment variables listed in Table 24-4, as well as any build settings that are defined at the target and build style layers for the active target and build style. However, keep in mind that Xcode does not recognize changes made to those environment variables during the script's execution and that shell scripts are independent of each other. If you change the value of environment variables in one script, the change is not visible in another script. For details on the build settings that are reflected in the script's environment variables, see "Using Build Settings With Run Script Build Phases" (page 284).

To perform operations on intermediate files, you can use several environment variables that Xcode sets before executing your script. They are listed in Table 24-4.

Table 24-4 Environment variables that you can access from a Run Script build phase

Environment variable	Description
`ACTION`	The action being performed on the current target, such as "build" or "clean."
`BUILD_VARIANTS`	The variations—debug, profile or normal—that Xcode is creating for the product being built .
`PROJECT_NAME`	The name of the project containing the target that is being built.
`PRODUCT_NAME`	The name of the product being built, without any extension or suffix.
`TARGET_NAME`	The name of the target being built.

Environment variable	Description
TARGET_BUILD_DIR	The location of the target being built.
BUILT_PRODUCTS_DIR	The directory that holds the products created by building the targets in a project.
TEMP_FILES_DIR	The directory that holds intermediate files for a specific target.
DERIVED_FILES_DIR	The directory that holds intermediate source files generated by the Compile Sources build phase.
INSTALL_DIR	The location for the installed product.

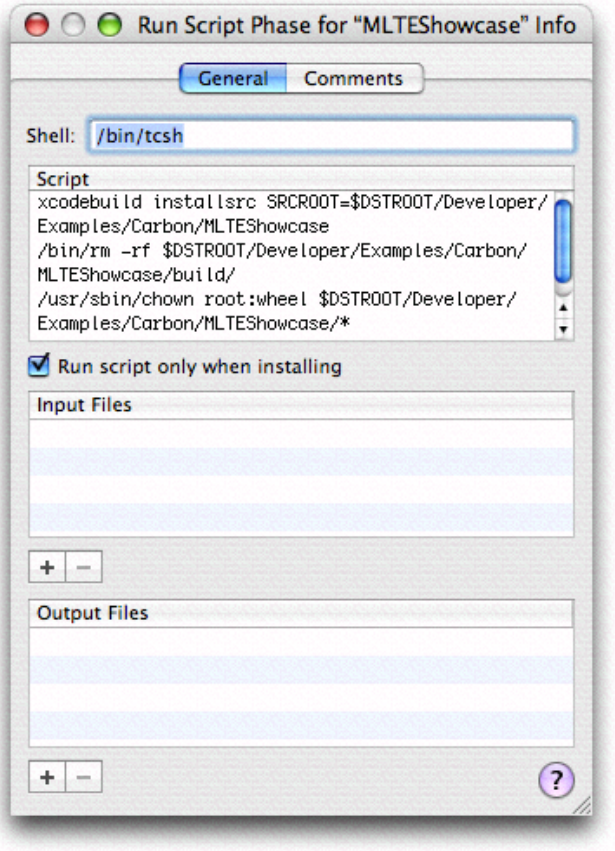
Keep in mind that the script executes using the permissions of the logged-in user. Xcode runs the script with the initial working directory set to the project directory.

To create a Run Script build phase:

1. In the project window, click the disclosure triangle next to the target you want to add the build phase to and select an existing build phase.
2. Choose Project > New Build Phase > New Shell Script Build Phase.

To configure the new Run Script build phase, select the build phase and open an Info or inspector window. You should see a window similar to the one shown here.

Figure 24-6 The Info window for a Run Script build phase



The Shell field specifies the path to the appropriate shell. Specify the script itself in the Script text field. You can type the contents of the script directly in the text field or invoke it from a file, as in the example shown above.

The "Run script only during deployment builds" option lets you specify that the script be run only during install builds; that is, when using the `install` option of `xcodebuild` or when the build settings Deployment Location (DEPLOYMENT_LOCATION) and Deployment Postprocessing (DEPLOYMENT_POSTPROCESSING) are on.

The Input Files and Output Files tables specify the names of input files and output files the script uses and produces. The Input Files table specifies the files that the script operates on; the Output Files table specifies the files that the script produces. To add an entry to either of these tables, click the "+" button below the table and type the name of the input or output file in the resulting text field. File paths are interpreted relative to the project directory.

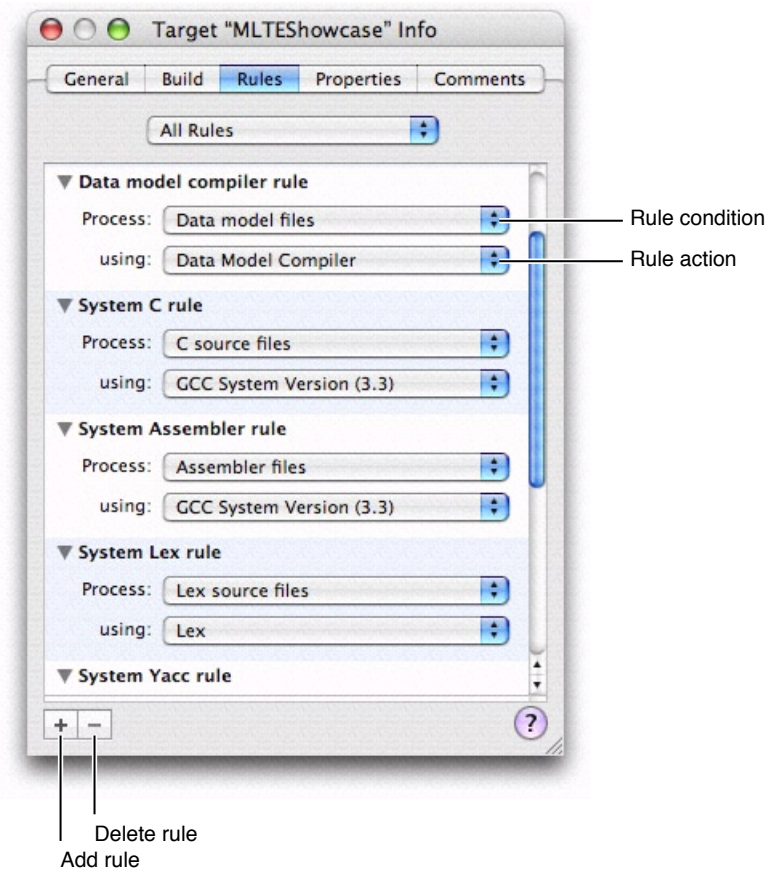
Xcode uses the input and output files to determine whether to run the script, and to determine the order in which the script is executed. Specifying input and output files ensures that Xcode runs the script only when the modification date of any of the input files is later than the modification date of any of the output files (reducing the time it takes to build your product), and that the files the script produces are included in the dependency analysis the build system performs before building your product. If you provide no outputs, Xcode runs the script every time you build the target.

Build Rules

Build rules specify how particular types of files are processed in Compile Sources build phases and specify the tool used to process files in Build ResourcesManager Resources build phases. For example, a build rule may indicate that all C source files be processed with the GCC 3.3 compiler.Each build rule consists of a condition and an action. The condition determines whether a source file is processed with the associated action. Usually, the condition specifies a file type.

Xcode provides default build rules that process C-based files, assembly files, Rez files, and so on. You can add rules to process other types of files to each target. You can see the build rules in effect for a target in the Rules pane of the target inspector, shown below.

Figure 24-7 The Rules pane of the Info window



There are two types of rules:

- System rules. These are predefined and unmodifiable, although you can override them. They include rules for processing C-based, Assembler, and Rez source files. See "System Rules" (page 262) for details. System rules are the same for all targets in a project.
- Target-specific rules. These are custom rules that you have defined for a particular target.

Target-specific rules can specify files that the system rules do not directly address or override the existing system rules. For example, there's a system rule for the processing of C-based source files, which means that `.c` and `.m` files are processed by the same rule. You can, however, add a target-specific rule indicating that `.m` files be processed by a different compiler. In addition, instead of specifying a particular type of file, you can set the rule's condition to a pattern that matches a set of files. See "Creating a Custom Build Rule" (page 262).

You can view all rules for a target, including available system rules, by choosing All Rules from the pop-up menu at the top of the Rules pane. To see only those build rules defined for the target, choose Rules Specific to Target.

A build rule's action typically specifies the tool or compiler to use when processing files that meet the given condition. But you can also specify a build-rule script. The default interpreter is `/bin/sh`. However, you can specify any script interpreter by entering `#!<interpreter_path>` as the first line of the script. When you use a build-rule script, you must specify the files the script produces as the build rule's output files. See "Creating a Custom Build Rule Script" (page 263).

When processing a source file, Xcode evaluates the build rules from top to bottom and chooses the first one whose condition matches the source file being processed. Because custom build rules appear above the built-in system rules, the custom build rules can override the system build rules.

System Rules

System rules are predefined rules in all targets and are used to process several well-known file types. Table 24-5 lists the system rules that Xcode provides.

Table 24-5 System rules

Rule	Inputs	Outputs
Data Model Compiler	`.xcdatamodel`	`.mom`
C	`.c`, `.m`	`.o`
Assembler	`.s`	`.o`
Yacc	`.y`	`.h`, `.c`
Lex	`.h`, `.c`	`.c`
Rez	`.r`	`.rsrc`
MiG	`.defs`	`.c`

Creating a Custom Build Rule

In addition to the system build rules, Xcode lets you define custom build rules on a per-target basis. Custom build rules allow you to change the way files of a particular type are processed or add support for types of files not directly addressed by the system rules.

To add a new build rule to a target, click the plus (+) button at the bottom of the Rules pane in the target inspector. Likewise, you can delete any rule—other than the system rules—by selecting the rule and clicking the minus (-) button.

To define a build rule's condition, choose a file type from the Process pop-up menu. You can also define rules that match arbitrary file names by choosing the last item in the pop-up menu ("Source files with names matching:") and specifying a filename pattern in the text field that appears. This field accepts the same kinds of file patterns that a Terminal shell accepts. For example, to match all files whose names start with a capital letter and end with a `.def` suffix, you specify `[A-Z]*.def` as the pattern.

You define the build rule's action by choosing one of the available compilers from the "using" pop-up menu. Xcode provides a number of built-in compilers for you to choose from. Alternatively, you can define your own custom script for processing files that meet the build rule's condition, as described in "Creating a Custom Build Rule Script" (page 263).

When processing a source file in the target, Xcode evaluates the build rules from top to bottom and chooses the first one whose condition matches the source file being processed. For this reason, you should put the most specific build rules above the more general ones. For example, a rule that matches only C++ files should appear above a rule that matches all C-like files—that is, C, C++, Objective-C, and Objective-C++ files. You can reorder custom rules by clicking in their background and dragging them. System rules cannot be reordered.

Creating a Custom Build Rule Script

Instead of choosing one of Xcode's built-in compilers to process the files specified by a build rule's condition, you can create a custom script to process those files. To do so, choose the "Custom script:" menu item. You can enter your script in the text field that appears, or you can store your script as a separate file in your project and invoke it from the text field using its project-relative path. In this case, you are actually defining a one-line script that calls the script in your project.

In addition to defining the script, you also need to tell Xcode the paths of any output files that the script produces. Enter the path to each separate output file that is produced by the script in the "with output files" table below the script text field. For each file, create a new row by clicking the plus (+) button. In this row, specify either the full (absolute) path or the project-relative path to the file. You can use any of the environment variables listed in Table 24-6 (page 264).

For example, suppose that you need to define a build rule to process files with the extension `.abcdef`. Also suppose that the build-rule script produces a `.c` file for each `.abcdef` file and that the generated `.c` files are placed in the default directory for intermediate files. In this case, the output-files specification could look like this:

```
$(DERIVED_FILES_DIR)/$(INPUT_FILE_BASE).c
```

The generated files are automatically fed back to the rule-processing engine. For example, continuing the `.abcdef` example rule, Xcode processes the `.c` files the script produces using the rule that processes `.c` files.

Execution Environment for Build-Rule Scripts

Before executing a build-rule script, Xcode sets environment variables to reflect the values of some build settings used to build the product. See "Using Build Settings With Run Script Build Phases" (page 284) for details. In addition, Xcode sets a few environment variables to values the script may need to access. Table 24-6 describes these environment variables.

Table 24-6 Environment variables for build-rule scripts

Variable	Description	Example
`INPUT_FILE_PATH`	Path to the source file being processed.	`/Users/me/Project/source.cpp`
`INPUT_FILE_DIR`	Directory containing the source file.	`/Users/me/Project`
`INPUT_FILE_NAME`	Filename of the source file being processed, including its extension.	`source.cpp`
`INPUT_FILE_BASE`	Base filename of the source file being processed.	`source`
`INPUT_FILE_SUFFIX`	Suffix of the source file being processed, including the leading period.	`.cpp`
`DERIVED_FILES_DIR`	Path of the directory for derived (intermediate) files.	`/Users/me/project/build/Project.build/Target.build/Derived-Sources`
`TARGET_BUILD_DIR`	Path to the directory into which the target's products are being built.	`/Users/me/Project/build`

Xcode runs build-rule scripts with the current working directory set to the project directory. This lets you access specific source files by using their project-relative paths.

Build Settings

A **build setting** is a variable that contains information used to build a product. For each operation performed in the build process—such as compiling Objective-C source files—build settings control how that operation is performed. For example, the information in a build setting can specify which options Xcode should pass to the tool—in this case, the compiler—used to perform that operation.

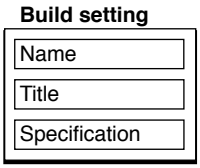
Build settings constitute the main method of customizing the build process. They represent variable aspects of the build process that Xcode consults as it builds a product.

This chapter explains how build settings are implemented and how you can take advantage of them to communicate with the build system. It explains the build setting syntax, the layers in which you can specify build settings, and how you can use them to customize the build process.

Overview of Build Settings

A build setting in Xcode has two parts: the name and the specification. The **build setting name** identifies the build setting and can be used within other settings; in that sense, it is similar to the names of environment variables in a command shell. The **build setting specification** is the information Xcode uses to determine the value of the build setting at build time. A build setting may also have a title, which is used to display the build setting in the Xcode user interface.

Figure 25-1 A build setting



The build system consults the value of build settings as it generates tool invocations. For example, to generate profiling code for all the source files compiled with `gcc`, you turn on the Generate Profiling Code (GCC_GENERATE_PROFILING_CODE) build setting. When Xcode creates the `gcc` command-line invocation to compile a source file, it evaluates build settings such as Generate Profiling Code to construct the argument list.

Build setting values may come from a number of different sources at build time. Xcode stores build setting definitions in dictionaries spread across several layers. Typically, you will be most interested in the target and build style layers; however, it is important to understand the various layers from which build setting values can be derived. See "Build Setting Layers" (page 269) for details.

Xcode has many built-in build settings that you can use to customize the build process. Furthermore, you can define your own build settings, which you can use to configure standard build settings across several targets or access within scripts in Run Script build phases to perform special tasks.

In addition to build settings, you can set **per-file compiler flags** that the build system uses when creating tool invocations. They allow you to make changes to how a file is compiled that do not affect any other files in the project. See "Per-File Compiler Flags" (page 294) for details.

Build Setting Syntax

Build setting names start with an uppercase letter or underscore character; the remaining characters can be uppercase letters, underscore characters, or numbers. For example, the build setting that specifies the name of a product is called PRODUCT_NAME. This restriction on the names of build settings helps identify build settings from other environment variables in shell scripts. The Xcode application doesn't allow you to define build settings whose names don't follow this convention. You can define build settings at the command-line layer that circumvent this restriction; for example, you can define a build setting called "project_name" in the command-line layer. Because build setting names are case-sensitive, however, the build system considers it a distinct build setting and doesn't override the value of the PROJECT_NAME build setting.

The Xcode application displays titles for most of its standard build settings in the project and target inspectors. For example, the title of the PRODUCT_NAME build setting is Product Name. `xcodebuild` doesn't use build setting titles.

The value of a build setting is determined by evaluating the build setting specification. A build setting specification can be a value such as a string, number and so forth; or it can reference the value of other build settings.

To reference a build setting value in a build setting specification, use the name of the build setting surrounded by parentheses and prefixed by the dollar-sign character ($). For example, the specification of a build setting that refers to the value of the Product Name build setting could be similar to `The name of this target's product is $(PRODUCT_NAME).`

Note: When referring to build setting values in build setting specifications, you must use the build setting name, such as PRODUCT_NAME, instead of its title.

In addition to the build settings provided by Xcode, you can add **user-defined build settings** to a project. A user-defined build setting is one that is not referenced by the build system. You can add them at the command line, build style, target, and environment levels. You can reference properly named user-defined build settings in build setting specifications and scripts in Run Script build phases. User-defined build settings don't have a title.

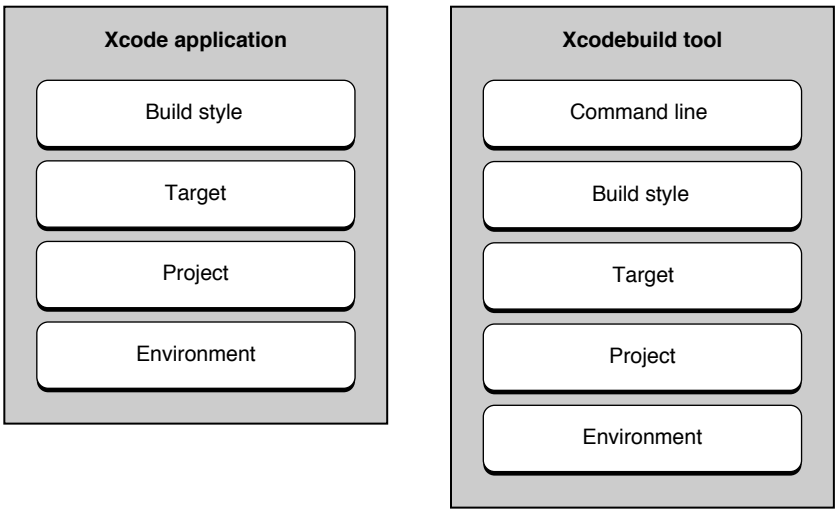
The following list describes some circumstances in which you may need to add user-defined build settings to a project:

- modify the value of build settings that are not displayed in the Xcode application
- specify a value that can be used by several build settings in different layers (see "Build Setting Evaluation" (page 271) for details)
- define a build setting you want to use in Run Script build phase scripts or a build-rule script

Build Setting Layers

Build setting specifications are defined in a number of layers depending on whether you build using the Xcode application or the `xcodebuild` tool. Figure 25-2 shows these layers.

Figure 25-2 Build setting layers



The specifications of build settings configured in higher layers override the specifications set at lower layers. The following sections describe each of the build system layers in detail:

1. Command-line layer (`xcodebuild` only)

   The `xcodebuild` tool's command invocation represents the highest layer in which you can configure build settings when building a product. The build settings configured in this layer are accessible only to `xcodebuild`.

   This layer gives you a way to customize builds done in batch mode from the command line without needing access to the Xcode application.

2. Build style layer

   The build style layer is available to both the Xcode application and the `xcodebuild` tool. It defines build setting variations for a build. These are changes to build setting definitions in the target layer and lower layers that allow you to produce different "flavors" of a product without having to create separate targets.

   Another use of the build style layer is to provide values for build settings to be shared by more than one target. All targets have access to the build settings in the active build style. When you have multiple targets with build settings that should be synchronized, a build style can be a convenient way to do so. See "Build Setting Evaluation" (page 271) for details.

   For more information on build styles, see "Build Styles" (page 297).

3. Target layer

This is the main layer for specifying how a product is built. Generally, it's the one that defines the largest number of build settings.

The Xcode user interface displays most of the build settings defined in this layer. However, the specifications displayed may not correspond to the values the build system obtains when building a product, unless you've customized the specification. For example, Installation Path (INSTALL_PATH) has a default value of `/usr/local/lib` for targets that produce a static library. This value is not shown in the target inspector; that is, the value shown for Installation Path is empty. But if you set the build setting to `$(INSTALL_PATH)/mylib` in the target inspector, at build time, INSTALL_PATH is `/usr/local/lib/mylib`.

Each target has its own set of build settings. That is, Xcode evaluates build settings for a target independently from the build settings defined in other targets. This is true even for dependent and aggregate targets.

For more information on targets, see "Targets" (page 231).

4. Project layer

   Xcode and `xcodebuild` use the Project layer to specify project-wide aspects, such as the location of the project itself. Build settings such as TEMP_DIR are specified in this layer. There is no mechanism for you to add build settings to this layer or modify the build setting specifications in it because you customize them in higher layers.

   Note: The SYMROOT, OBJROOT and SDKROOT build settings are exceptions. Xcode provides a user interface for modifying these build settings in the project inspector. The SYMROOT and OBJROOT build settings identify the build locations for build products and intermediate build files, respectively. The interface for setting these locations is described in "Build Locations" (page 301). The SDKROOT build setting specifies the path to the SDK to build against; "Using Cross-Development in Xcode" (page 333) describes how to choose an SDK in the project inspector.

5. Environment layer

   The environment layer is composed of environment variables that correspond to build setting names. You can use it to configure build settings that must apply to more than one project. This layer, however, cannot access build settings configured in any other layer. For example, defining an environment variable named `MY_PRODUCT_NAME` as `My Company $(PRODUCT_NAME)` results in an undefined-variable error, unless you also define an environment variable named `PRODUCT_NAME`.

   You must follow the syntax described in "Build Setting Syntax" (page 268) when defining the environment variables in your session.

   Note: If you associate additional compiler flags with a file, as described in "Per-File Compiler Flags" (page 294), those flags will always be used when the file is processed as part of a build. They cannot be overridden in any of the build setting layers.

Build Setting Evaluation

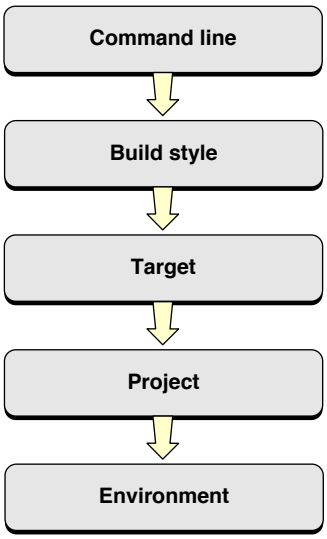
To take advantage of build settings in your project, you must understand how they are evaluated when Xcode builds your product. Placing build setting specifications in particular build setting layers provides you the ability to, for example, quickly change an aspect of a product when building from the command line, configure a product aspect only in one of a product's alternate "flavors" but not in all of them, or specify an aspect for all the projects that you work on.

This article describes how Xcode evaluates build settings. It provides details that help you determine where to place build setting specifications to greatly customize the build process.

Overview of Build Setting Evaluation

At build time, Xcode evaluates each build setting individually. Figure 25-3 shows the order in which the build system evaluates build settings.

Figure 25-3 Build setting evaluation precedence



Note: Remember that the build setting layers may not all be available when you build a product. If you build from the Xcode application, the command-line layer is not used. See "Build Setting Layers" (page 269) for details.

When the build system needs the value of a build setting, it starts at the highest build setting layer available to it and works down in the order shown in the following list:

1. Command invocation: Build settings defined in the `xcodebuild` invocation.
2. Build style: Build settings defined in the active build style.
3. Target: Build settings defined in the target being built.

4. Product: Build settings whose value depends on the type of product built by the target.

5. Environment variables: Build settings defined in the Xcode application environment or the shell from which `xcodebuild` is launched.

As soon as the build system finds a definition for the build setting it's looking for, it stops traversing the build setting layers. However, if the build setting specification found includes references to other build settings, it resolves them, which starts the traversing process again, as many times as necessary to compute the value of the original build setting.

If a build setting refers to itself (that is, the build setting specification includes a reference to the build setting being evaluated), the build system resolves the reference starting at the subsequent build setting layer. The following sections provides examples illustrating this process.

Evaluating a Build Setting Defined in Multiple Layers

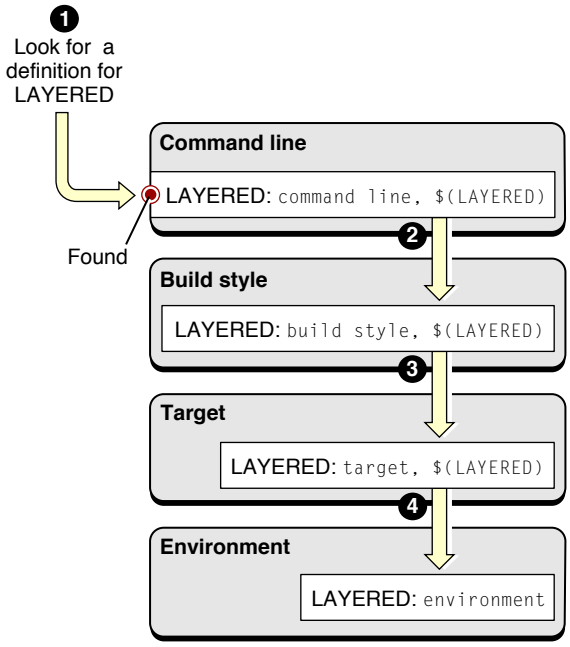
Imagine that you need to define a single build setting at every build setting layer (except the noncustomizable project layer) in order to build a product. This is a very unlikely case to be sure, but one that illustrates how the process works. Table 25-1 shows an example configuration for the LAYERED build setting throughout the build setting layers. This example assumes that the product is built from the command-line using `xcodebuild`:

Table 25-1 Configuration of the LAYERED build setting

Build setting layer	Build setting specification
Command line	`command line, $(LAYERED)`
Build style	`build style, $(LAYERED)`
Target	`target, $(LAYERED)`
Environment	`environment`

Figure 25-4 shows how the build system would evaluate the LAYERED build setting when building using `xcodebuild`.

Figure 25-4 Evaluation of the LAYERED build setting



Note: The project layer is omitted from Figure 25-4 because you cannot configure build settings in that layer.

To evaluate the LAYERED build setting, the build system does the following:

1. Looks for a definition for `LAYERED` in the command-line layer (the highest available to `xcodebuild`). It finds the specification `command line, $(LAYERED)`.
2. Resolves `$(LAYERED)` starting at the next layer down, the build style layer. At this layer, it obtains the specification `build style, $(LAYERED)`. Because this specification also references the value of the LAYERED build setting, the build system continues to look in lower layers for the build setting specification.
3. Resolves `$(LAYERED)` starting at the target layer, obtaining `target, $(LAYERED)`.
4. Resolves `$(LAYERED)` starting at the environment layer, obtaining `environment`.

   The evaluation of LAYERED stops here because there are no build setting layers below the environment layer. When all the references are resolved, the final value of the build setting is computed as `command line, build style, target, environment`.

This example uses `$(LAYERED)` to access the value of the same build setting at a lower layer. However, you can also use `$(value)` for that purpose. That is, replacing `$(LAYERED)` with `$(value)` in the any of the build setting specifications at the command-line, build-style, or target layers in the example above yields the same result.

The process of evaluating a build setting specification that references itself repeats recursively until the build system reaches the environment layer or until the build system finds a build setting specification that does not reference its own value.

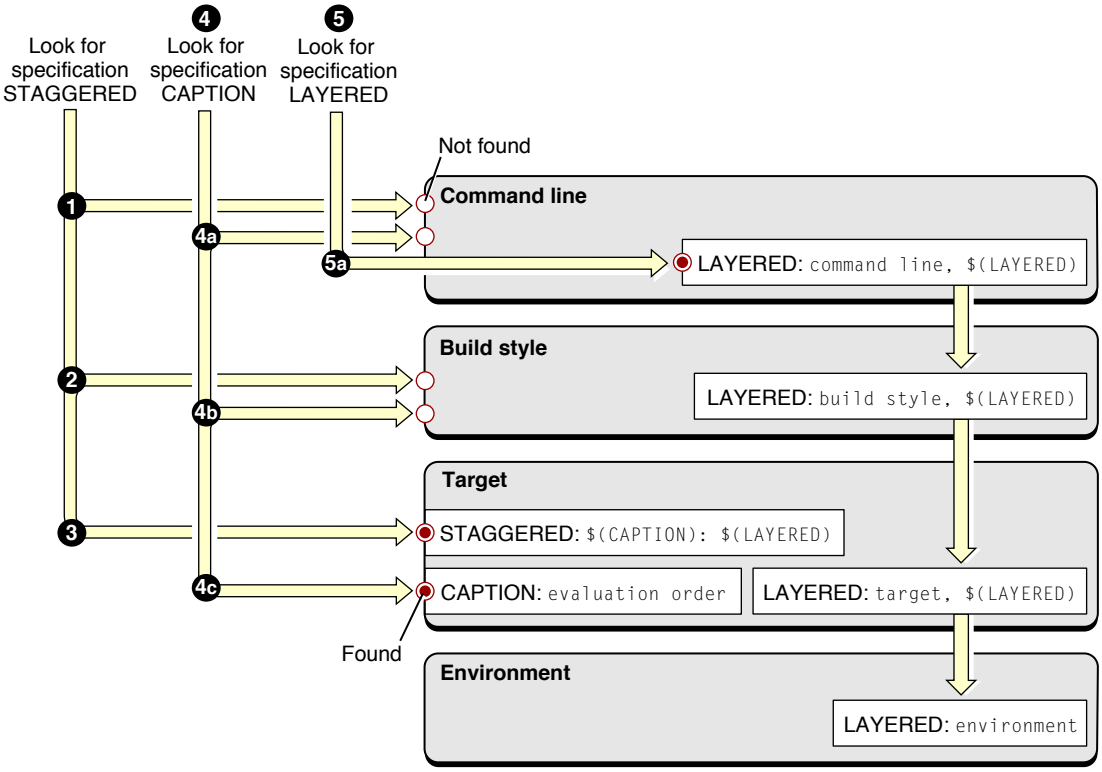
Evaluating a Build Setting Specification Using Several Values

The build system gives you a great deal of flexibility when configuring build settings. The following example illustrates how the build system evaluates the STAGGERED build setting, which is defined in the Target layer and references the values of several other build settings.

The value of the STAGGERED build setting is composed of data and a caption for the data, with the caption shown first and both elements separated by a colon (:) and a space. The specification for STAGGERED contains references to two other build settings: LAYERED (the data, explained in the previous section) and CAPTION (the caption for the data). It also contains static elements (the colon and space characters).

Figure 25-5 shows the evaluation of the STAGGERED build setting.

Figure 25-5 Evaluation of the STAGGERED build setting



These are the steps the build system takes to evaluate the STAGGERED build setting:

1. Look for a specification for STAGGERED in the command-line layer. None is found.
2. Look for a specification for STAGGERED in the build style layer. None is found.
3. Look for a specification for STAGGERED in the target layer.

   The build system finds a specification for STAGGERED: `$(CAPTION): $(LAYERED)`. Here is where the evaluation of the STAGGERED build setting begins.
4. Resolve `$(CAPTION)` starting at the command-line layer:

a. Look for a specification for CAPTION in the command-line layer. None is found.

b. Look for a specification for CAPTION in the build style layer. None is found.

c. Look for a specification for CAPTION in the target layer. The build system finds the specification `evaluation order`.

   The evaluation of CAPTION stops here because there are no references to other build settings. The final value of the CAPTION build setting is `evaluation order`.

5. Resolve `$(LAYERED)` starting at the command-line layer.

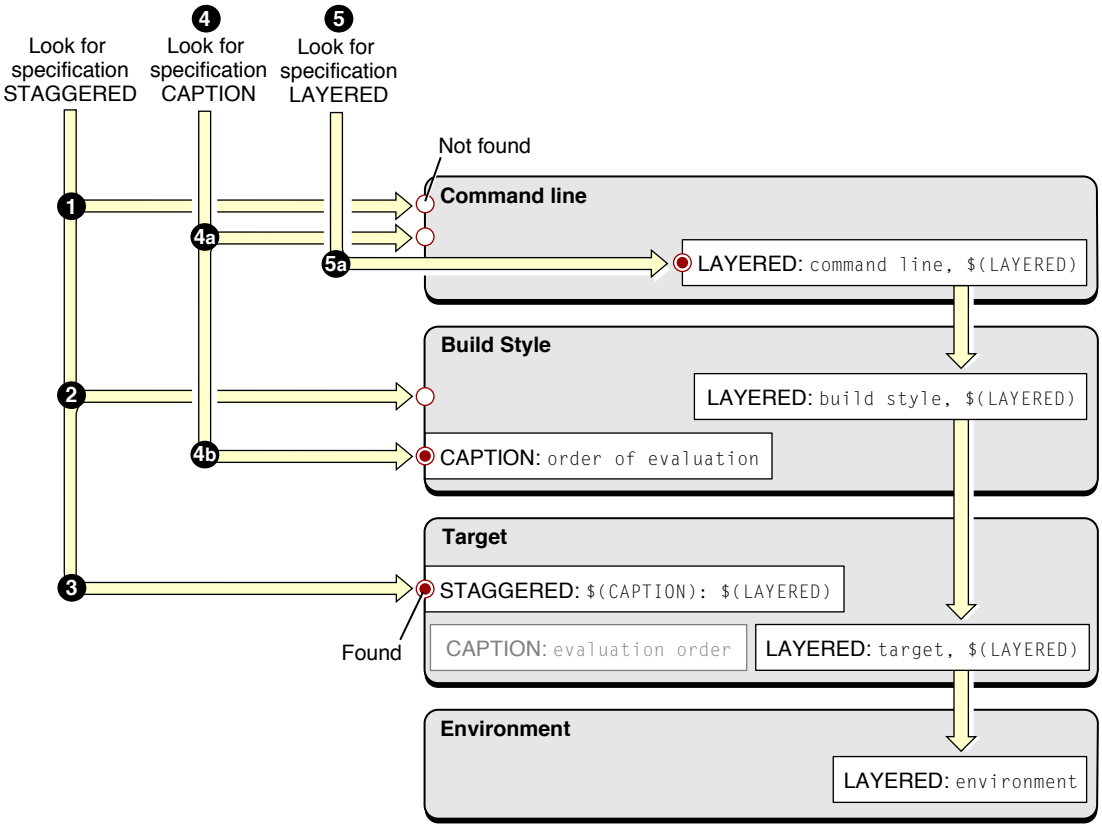
   a. Look for a specification for LAYERED in the command-line layer. The build system finds the specification `command line, $(LAYERED)`. The build system resolves the specification for LAYERED at this build setting layer as described in the previous section. The final value of the LAYERED build setting is `command line, build style, target, environment`.

6. Get the final value for the STAGGERED build setting by replacing the two references in the build setting's specification with their values: `evaluation order: command line, build style, target, environment`.

Knowing the precedence that the build system uses when evaluating build settings makes it easy to determine where to configure build settings to tailor the build process for special situations. Following the STAGGERED example, imagine you want to override the value of the CAPTION build setting at the build style layer. All you would have to do is configure the CAPTION build setting in the active build style with the appropriate value.

Figure 25-6 shows the effects of overriding CAPTION in the build style layer.

Figure 25-6 Evaluation of the STAGGERED build setting with CAPTION overridden in the Build Style layer



These are the steps the build system takes to evaluate the STAGGERED build setting after overriding CAPTION in the build style build setting layer:

1. Look for a specification for STAGGERED in the command-line layer. None is found.
2. Look for a specification for STAGGERED in the build style layer. None is found.
3. Look for a specification for STAGGERED in the target layer. The build system finds `$(CAPTION): $(LAYERED)`.
4. Resolve `$(CAPTION)` starting at the command-line layer:
    a. Look for a specification for CAPTION in the command-line layer. None is found.
    b. Look for a specification for CAPTION in the build style layer. The build system finds `order of evaluation`.

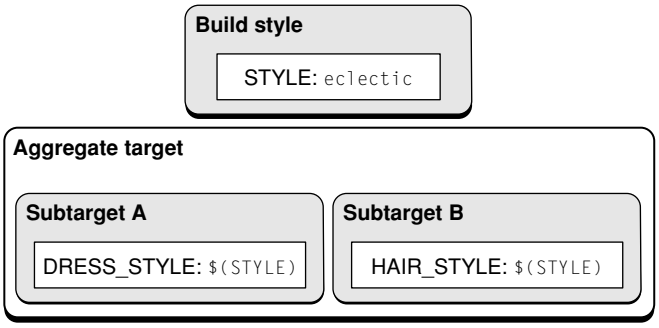
    The evaluation of CAPTION stops here because there are no references to other build settings in its specification. Even though CAPTION is configured in the target layer, that specification has been overridden in the build style layer; therefore, it's ignored. The final value of CAPTION in this example is `evaluation order`.
5. Look for a specification for LAYERED in the command-line layer. The build system finds `command line, $(LAYERED)`.

a. Resolve the specification for LAYERED at this build setting layer as described earlier in this section, obtaining `command line, build style, target, environment`.

6. Get the final value for the STAGGERED build setting by replacing the two references in the build setting's specification with their values: `order of evaluation: command line, build style, target, environment`.

A build style can define build settings that are shared among all the targets processed during a build. Figure 25-7 illustrates how two targets can use a build setting configured in the active build style.

Figure 25-7 Sharing build setting values among targets



When the aggregate target is built, DRESS_STYLE in target A and HAIR_STYLE in target B evaluate to `eclectic`.

Editing Build Settings in the Xcode Application

The Xcode application lets you access and edit build settings at the target and build style layers. It provides a convenient graphical user interface for changing build setting specifications. For native targets, the target inspector lets you see build setting specifications at the target layer. The project inspector lets you examine the build settings defined for the project's build styles. To view and edit build settings for legacy Project Builder targets and external targets, use the target editor.

Viewing Build Settings in an Inspector

You can view and edit build settings at the target level in the Build pane of the target inspector. To modify build settings at the build style level, use the Styles pane of the project inspector. The interface for modifying build settings in these two inspector window panes is almost identical.

Figure 25-8 The Build pane of a target Info window

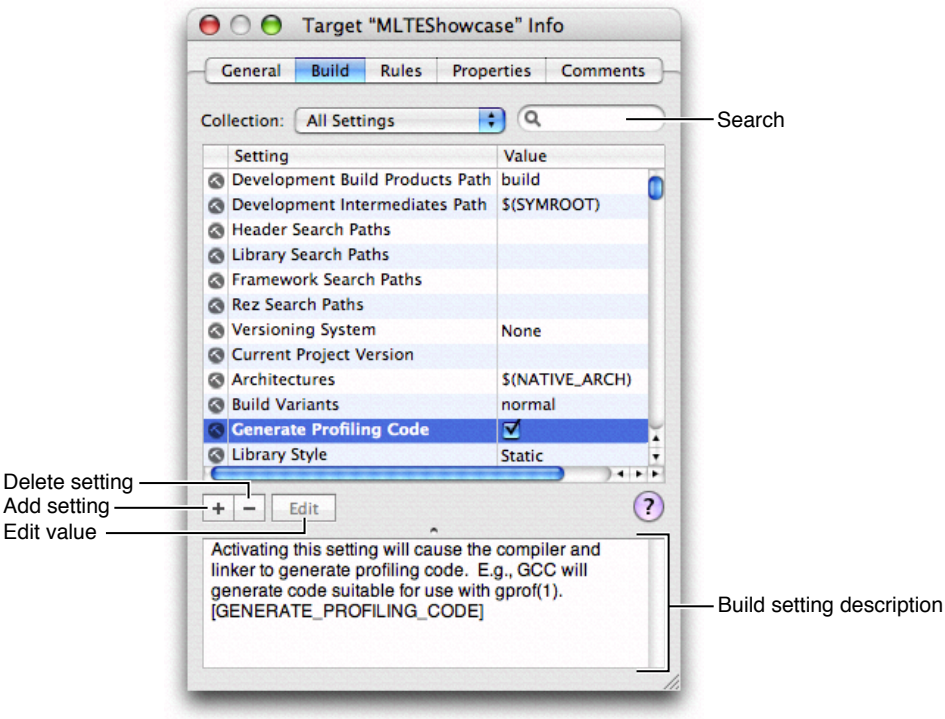


Figure 25-8 shows the Build pane of the target inspector. The target's build settings appear in a table in the Build pane. The columns show, from left to right:

- An icon indicating the type of setting. For example, settings that control warnings and errors are indicated by the warning icon.
- The Setting column contains the build setting title. This is a brief English-language description of the setting. To see the build setting name, open the help field, as described later in this section.
- The Value column contains the build setting specification. For Xcode's standard build settings, this may be a string, a pop-up menu of possible values, or an on/off value. This specification can also reference the value of other build settings.

If you do not know what a particular build setting does, the help field displays a longer description of the selected build setting. If the help field is not visible, drag the resize control below the build settings table to view the contents of the help field. If you want to know the name of the build setting, you can also find that in the help field.

You can search the table of build settings for a keyword or other text string using the search field at the top of the pane. As you type, Xcode filters the list to include only those settings that match the text in the search field. For example, you can find all build settings related to search paths by typing "search." Xcode searches both the build setting title and the build setting description in the help field.

If you know the title of the build setting you are looking for and keyboard focus is in the build settings table, you can simply start typing the build setting name to select that build setting.

Note that the inspector does not tell you all the build settings used when you press the Build button. The inspector shows only the build settings defined at the target or build style layer. To learn how to view all build settings used when you build, see "Finding Where a Build Setting is Defined" (page 284).

As mentioned earlier in "Build Setting Layers" (page 269), build settings defined in a build style have higher precedence than those defined in the target layer. If a build setting is defined in both the active build style and in the active target, the definition in the build style overrides that in the target. Xcode indicates this in the target inspector by crossing out settings that are overridden by the active build style. You can jump to the active build style's definition of that build setting by Control-clicking the build setting and choosing Jump to Active Build Style. Xcode opens the project inspector to the Styles pane and selects the build style's definition for the build setting. Build settings that are defined at the target level are shown in boldface in the target inspector.

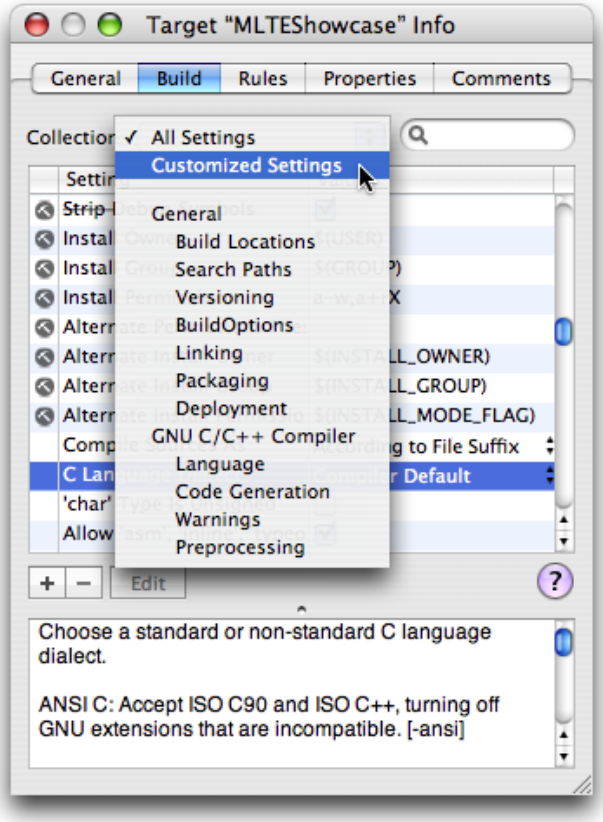
You can modify build settings for multiple targets at once; the Build pane of the target inspector supports multiple selection. To edit build settings for more than one target at a time, select those targets in the project window and open an inspector window.

For a list of the build settings available in the Xcode application's user interface see "Build Setting Names and Their Corresponding Titles" (page 286). For a complete list of Xcode build settings, see Xcode Build Settings or choose Help > Show Build Settings Notes.

Collections of Build Settings

For easy access to sets of related build settings, Xcode groups build settings into several different collections. The Collection pop-up menu above the build settings table contains a list of all available groups of build settings. To view the build settings in a group, choose that group from the pop-up menu, as shown in the following figure.

Figure 25-9 Choosing a build setting collection



Xcode provides groups for build settings that affect general aspects of the build process—such as the location at which build products are placed—as well as collections for tool-specific build settings that affect compiler options. Build setting collections have no effect on how the target is built.

To see all of the possible build settings and their definitions, if any, choose All Settings from the Collection menu. Note that tool-specific build settings appear in the target inspector only if the target contains files that are built using that tool. For example, if the target does not contain any resource files processed using the Rez tool, the Rez-specific build settings do not appear in the Collection menu.

The Customized Settings collection shows all of the build settings that are defined at the current target or build style layer. Like a smart group, the Customized Settings collection automatically updates to reflect changes to build settings at the current layer. For example, if you make a change to the value of a build setting in one of the other collections, the Customized Settings collection changes to include that build setting and its new definition.

Editing Build Setting Specifications

For standard build settings, the interface for modifying a build setting's specification in the target and project inspectors depends on the possible values for that build setting. These possible values are:

- String. If a build setting takes a string as its value, select the build setting row and click Edit. If the build setting takes a single string as its value, Xcode displays a text field; type the string into this field. You can also edit the string simply by clicking in the Value column for the build setting and typing the value.

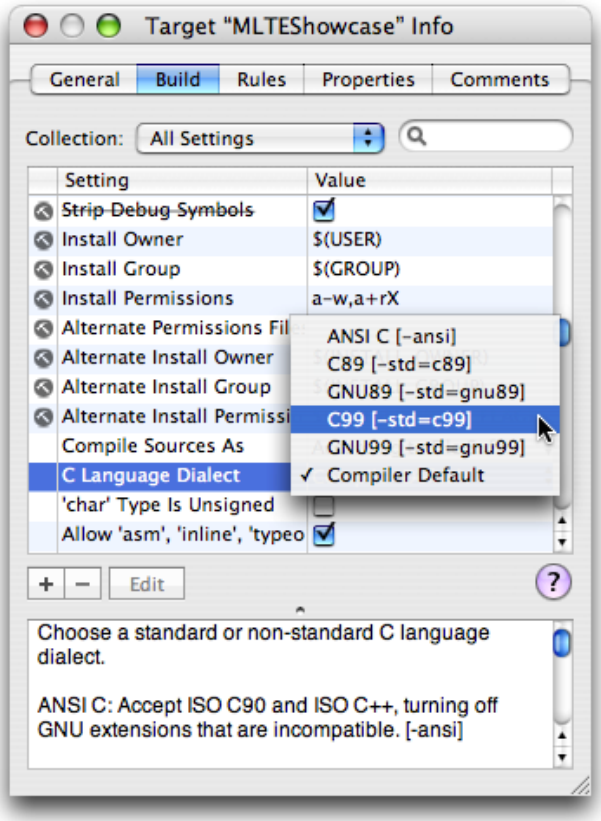
  If the build setting can contain one or more strings—for example, Header Search Paths—Xcode displays a table. To enter a string in this table, click the plus button and type the string into the table row added by Xcode. You can re-order the strings in the table by dragging the rows to the location at which you want them to appear.

  If you are entering file paths, you can simply drag the file or folder from the Finder into the text field or table, instead of typing the paths in yourself. Xcode will insert the path to the file.

- Boolean. If a build setting can have two states—enabled or disabled—the Value column contains a checkbox. A checkmark indicates that the build setting is enabled. To change the state of the build setting, click the checkbox.

- If a build setting has a finite number of possible values, Xcode displays a pop-up menu in the Value column. You can choose the desired value from this menu. For example, here is how you might select a build setting value from a menu:

Figure 25-10 Changing the value of a build setting



In addition, Xcode provides a custom user interface for modifying certain search path and architecture build settings, as described in "Editing Search Paths" (page 282) and "Creating Multi-Architecture Binaries" (page 283).

Adding and Deleting Build Settings

If you do not see the build setting you wish to modify, or if you want to define your own custom build settings, you can add build settings to the build settings table in the Build pane of the inspector window. To add a new build setting to a target or build style, click the plus (+) button below the build settings table. Double-click in the Setting column and type the name of the build setting, then double-click in the Value column and type the build setting specification.

To delete a build setting from a target, select that build setting and click the minus button (-).

Editing Search Paths

Xcode defines a number of build settings for specifying general search paths for files and frameworks used by targets in your project. These build settings are:

- Header Search Paths (HEADER_SEARCH_PATHS)

  This is a list of paths to folders to be searched by the compiler for included or imported header files when compiling C, Objective-C, C++, or Objective-C++ source files.

- Library Search Paths (LIBRARY_SEARCH_PATHS)

  This is a list of paths to folders to be searched by the linker for static and dynamic libraries used by the product.

- Framework Search Paths (FRAMEWORK_SEARCH_PATHS)

  This is a list of paths to folders containing frameworks to be searched by the compiler for both included or imported header files when compiling C, Objective-C, C++, or Objective-C++, and by the linker for frameworks used by the product.

- Rez Search Paths (REZ_SEARCH_PATHS)

  This is a list of paths to search for files included by Carbon Resource Manager resources and compiled with the Rez tool.

Xcode supports recursive search paths. For each path that you enter in one of these search path build settings, you can specify that Xcode search the directory at that path, as well as any subdirectories that directory contains.

You can edit search paths in much the same way that you edit other build settings that contain lists of strings, as described in "Editing Build Setting Specifications" (page 280). However, to specify that Xcode perform a recursive search for items at a particular path, select the checkbox next to that path in the Recursive column of the table that Xcode displays when you click Edit in the target or build style inspector. When this is selected, Xcode searches the directory at the specified location, and all subdirectories in it, for the header, framework, library or resource.

Note: Xcode does not search the contents of certain bundle structures and other special directories. These are `.nib`, `.lproj`, `.framework`, `.gch`, `.xcode`, CVS and `.svn` directories, as well as any directory whose name is in parentheses.

Xcode performs a breadth-first search of the directory structure at the search path, following any symlinks. Xcode searches locations in the order in which they appear in the search paths table, from top to bottom. It stops when it finds the first matching item, so if you have multiple files or libraries of the same name at the locations specified by a set of search paths, Xcode uses the first one it finds.

Note that adding very deep directory structures to a list of recursive search paths can increase your project's build time. The maximum number of paths that Xcode will expand a single recursive search path to is 1024.

Creating Multi-Architecture Binaries

Xcode can create multi-architecture (or "fat") binaries, which are executable files that can contain code and data for more than one CPU architecture. You can target multiple PowerPC-architecture CPUs with a single binary file. The Architectures (ARCHS) build settings lets you specify which architectures Xcode builds for. You can edit this build setting in the target inspector: in the Build pane, select the Architectures build setting and click Edit. Xcode displays a list of all of the supported architectures. Select the checkbox next to the architectures for which you want to build. You can build for any of following architectures:

- 32-bit architectures:
  - G3
  - G4
  - G5
- 64-bit architecture: G5

Xcode compiles for each architecture individually and creates a single fat file from these input files. For more information on multi-architecture files, see *Mac OS X ABI Mach-O File Format Reference*.

Editing Build Settings for Legacy and External Targets

You cannot configure build information for Jam-based Project Builder targets and external targets in the target inspector. To edit the build settings for Jam-based and external targets in Xcode, select the target in the Groups & Files list. If you have an editor open in the project window, Xcode displays the Project Builder target editor. To view this target editor in a separate window, double-click the target.

On the left side of the target editor, Xcode displays a number of groups of target settings appropriate for the current target. Selecting any of these groups displays its settings in the editor on the right side of the Target window.

To view the build settings for an external target, select the Custom Build Settings group. To view the build settings for a legacy Project Builder target, select the Expert View item in the Settings group. The target editor displays a table of build settings:

- The Name column contains the build setting name.

- The Value column contains the build setting specification.

Similar to the Build pane of the inspector window for native targets, you can add and delete build settings using the plus (+) and minus (-) buttons below the table. To edit a build setting, double-click in the appropriate column and type the build setting name or specification. The target editor for legacy targets also includes a simpler interface for a number of common build settings in the Simple View.

Using Build Settings With Run Script Build Phases

As Xcode constructs the command-line invocations for the various tools it uses to build a product, it accesses the appropriate build settings. Also, when it sets the environment variables for shell scripts in Run Script build phases, it resolves most build settings and sets environment variables with their values. Notable build settings that are not passed as environment variables to shell scripts are those geared specifically to tools such as `gcc`, `lex` and `yacc`. You can identify these build settings by examining the prefixes of build setting names. For example, Enable Trigraphs (GCC_ENABLE_TRIGRAPHS) contains the prefix GCC, indicating that it's exclusively tailored for the `gcc` tool; therefore, it isn't passed to shell scripts defined in Run Script build phases.

Note: Modifying the values of the environment variables that the build system sets for shell scripts in Run Script build phases has no effect on the value of the corresponding build setting. That is, shell scripts executed as part of the same build process after one in which you modify the environment variables get the unmodified values. This is the expected UNIX scoping behavior for environment variables.

Troubleshooting Build Settings

As you work on a project, you may need to determine where and how a build setting is defined. Because build settings can be identified by their name and their title (see "Build Setting Syntax" (page 268) for details), depending on where a build setting is defined (in the `xcodebuild` invocation, in an environment variable, or the Xcode application's user interface), you need to map between a build setting name and its corresponding build setting title.

This article provides tips on how to use the build system to help you find build setting specifications and troubleshoot build setting problems you may encounter.

Finding Where a Build Setting is Defined

During the development process you may need to find the top definition of a build setting to change its value during a build. That is, the specification that overrides all other specifications throughout the build setting layers (see "Build Setting Evaluation" (page 271) and "Build Setting Layers" (page 269) for details). This section showcases a technique you can use to quickly locate the top definition of a build setting.

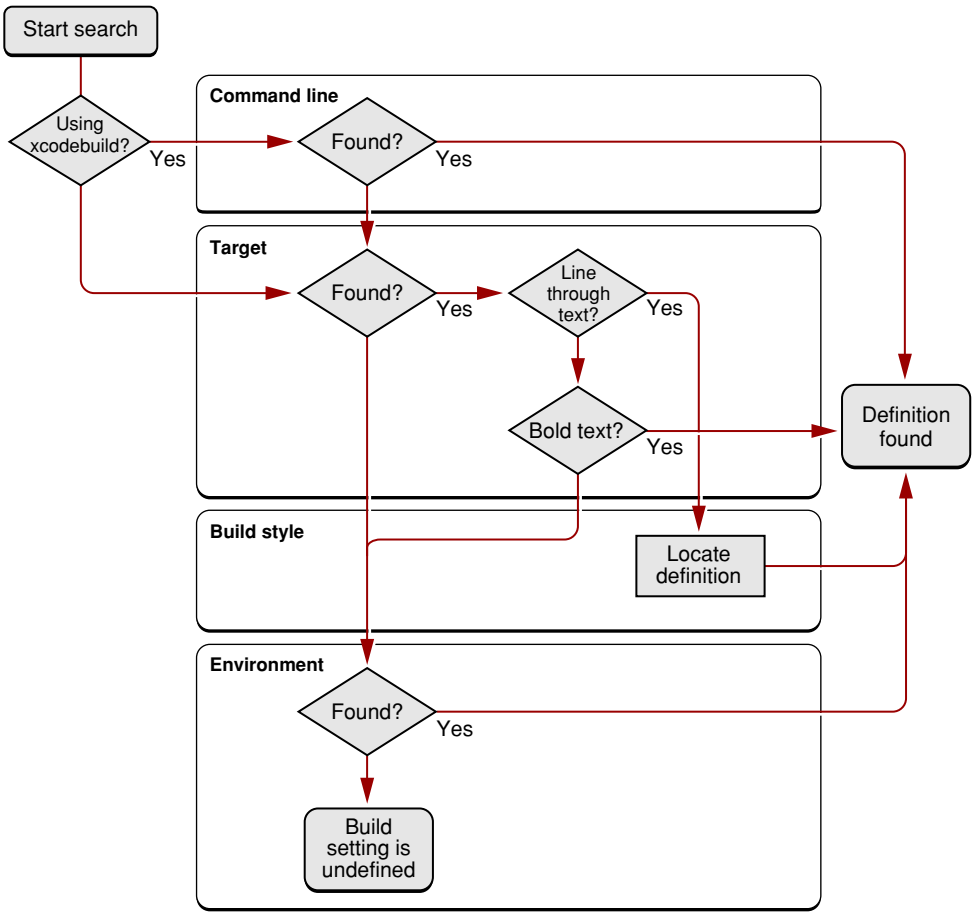
Look for the top definition of a build setting in these places:

- The `xcodebuild` invocation (when building using `xcodebuild`)
- The target you're interested in

- The active build style
- The output of the `env` command

Figure 25-11 shows the steps you would take to find the definition of a build setting when you know the build setting's name or title.

Figure 25-11 [Finding the definition of a build setting



1. If you're building using `xcodebuild`, look for the build setting name in the tool's invocation. If the build setting is part of the invocation, you found the definition.

2. Look for the build setting in the target you're interested in. You can locate a particular build setting by selecting the All Settings collection in the Build pane of the target inspector and entering the name or the title of the build setting you're looking for. See "Build Setting Names and Their Corresponding Titles" (page 286) for mappings of most build setting titles to the corresponding build setting names and vice versa.

    a. If the build setting has a line through it, it means that the build setting is overridden by the active build style. Look up the build setting in the active build style to find its definition. You can jump to the definition of the build setting in the active build style by Control-clicking the build setting and choosing Jump to Active Build Style.

b. If the build setting is displayed in bold text, it means that it's defined in the target and isn't overridden by the active build style. This is the top definition of the build setting.

c. If the build setting is displayed in nonbold text and doesn't have a line through it, the build setting is defined in the project layer or the environment layer (see "Build Setting Layers" (page 269) for details).

3. Look for the build setting in the build environment (the definitions of all the environment variables the Xcode application or `xcodebuild` have access to during the build process).

   a. Add a Run Script build phase to the target you're interested in and make it the first build phase to make it easier to locate the script's output.

   b. Add an invocation to the `env` command to the build phase's shell script.

   c. If you're building using the Xcode application, open the Build Results window, reveal the build log pane, and build the product. If you're building using `xcodebuild`, invoke the tool from your shell as you normally would. See "Building a Product" (page 301) for information on how to build in Xcode.

   d. Look at the build log (the detailed log in the Build Results window if using the Xcode application). The build environment is listed after the group of `setenv` invocations that set environment variables that reflect most of the build setting values for the current build. Search that group for the name of the build setting you're interested in. If the build setting is not in that group, the build setting is not defined for the target you are investigating.

      If the build setting is defined in the `setenv` group and the `env` group, you can override its value only in the target layer or above. If the build setting is defined only in the `env` group, look for the build setting definition among the environment variables defined in the user-configuration files for the logged-in user. If you find the build setting there, change its specification, log out, and log in.

Build Setting Names and Their Corresponding Titles

This section contains several tables that map build setting names to build setting titles and build setting titles to build setting names for the general and compiler-related build settings displayed in the Xcode user interface.

Table 25-2 maps the build setting names with the corresponding build setting titles of the general build settings.

Table 25-2 General build settings by build setting name

Build setting name	Build setting title
ALTERNATE_MODE	Alternate Install Permissions
ALTERNATE_PERMISSIONS_FILES	Alternate Permissions Files
ALTERNATE_OWNER	Alternate Install Owner
ALTERNATE_GROUP	Alternate Install Group

Build setting name	Build setting title
COPYING_PRESERVES_HFS_DATA	Preserve HFS Data
CURRENT_PROJECT_VERSION	Current Project Version
DEPLOYMENT_LOCATION	Deployment Location
DEPLOYMENT_POSTPROCESSING	Deployment Post-processing
DYLIB_COMPATIBILITY_VERSION	Compatibility Version
DYLIB_CURRENT_VERSION	Current Version
EXPORTED_SYMBOLS_FILE	Exported Symbols File
FRAMEWORK_SEARCH_PATHS	Framework Search Paths
FRAMEWORK_VERSION	Framework Version
GENERATE_PKGINFO_FILE	Force Package Info Generation
HEADER_SEARCH_PATHS	Header Search Paths
INFOPLIST_FILE	Info.plist File
INIT_ROUTINE	Initialization Routine
INSTALL_GROUP	Install Group
INSTALL_MODE_FLAG	Install Permissions
INSTALL_OWNER	Install Owner
INSTALL_PATH	Installation Path
LIBRARY_SEARCH_PATHS	Library Search Paths
LIBRARY_STYLE	Library Style
MACOSX_DEPLOYMENT_TARGET	Mac OS X Deployment Target
OTHER_LDFLAGS	Other Linker Flags
PREBINDING	Prebinding
PRIVATE_HEADERS_FOLDER_PATH	Private Headers Folder Path
PRODUCT_NAME	Product Name
PUBLIC_HEADERS_FOLDER_PATH	Public Headers Folder Path
REZ_SEARCH_PATHS	Rez Search Paths
SECTORDER_FLAGS	Symbol Ordering Flags

Build setting name	Build setting title
SKIP_INSTALL	Skip Install
UNSTRIPPED_PRODUCT	Unstripped Product
VERSIONING_SYSTEM	Versioning System
WARNING_LDFLAGS	Warning Linker Flags
WRAPPER_EXTENSION	Wrapper Extension
ZERO_LINK	Zero Link

Table 25-3 maps the build setting titles with the corresponding build setting names of the general build settings.

Table 25-3 General build settings by build setting title

Build setting title	Build setting name
Alternate Install Group	ALTERNATE_GROUP
Alternate Install Owner	ALTERNATE_OWNER
Alternate Install Permissions	ALTERNATE_MODE
Alternate Permissions Files	ALTERNATE_PERMISSIONS_FILES
Compatibility Version	DYLIB_COMPATIBILITY_VERSION
Current Project Version	CURRENT_PROJECT_VERSION
Current Version	DYLIB_CURRENT_VERSION
Deployment Location	DEPLOYMENT_LOCATION
Deployment Post-processing	DEPLOYMENT_POSTPROCESSING
Exported Symbols File	EXPORTED_SYMBOLS_FILE
Force Package Info Generation	GENERATE_PKGINFO_FILE
Framework Search Paths	FRAMEWORK_SEARCH_PATHS
Framework Version	FRAMEWORK_VERSION
Header Search Paths	HEADER_SEARCH_PATHS
Info.plist File	INFOPLIST_FILE
Initialization Routine	INIT_ROUTINE
Install Group	INSTALL_GROUP

Build setting title	Build setting name
Install Permissions	INSTALL_MODE_FLAG
Install Owner	INSTALL_OWNER
Installation Path	INSTALL_PATH
Library Search Paths	LIBRARY_SEARCH_PATHS
Library Style	LIBRARY_STYLE
Mac OS X Deployment Target	MACOSX_DEPLOYMENT_TARGET
Other Linker Flags	OTHER_LDFLAGS
Prebinding	PREBINDING
Preserve HFS Data	COPYING_PRESERVES_HFS_DATA
Private Headers Folder Path	PRIVATE_HEADERS_FOLDER_PATH
Product Name	PRODUCT_NAME
Public Headers Folder Path	PUBLIC_HEADERS_FOLDER_PATH
Rez Search Paths	REZ_SEARCH_PATHS
Skip Install	SKIP_INSTALL
Symbol Ordering Flags	SECTORDER_FLAGS
Unstripped Product	UNSTRIPPED_PRODUCT
Versioning System	VERSIONING_SYSTEM
Warning Linker Flags	WARNING_LDFLAGS
Wrapper Extension	WRAPPER_EXTENSION
Zero Link	ZERO_LINK

Table 25-4 maps the build setting names with the corresponding build setting titles of the compiler-related build settings.

Table 25-4 GNU C/C++ compiler build settings by build setting name

Build setting name	Build setting title
GCC_ALTIVEC_EXTENSIONS	Enable AltiVec Extensions
GCC_C_LANGUAGE_STANDARD	C Language Dialect
GCC_CHAR_IS_UNSIGNED_CHAR	'char' Type Is Unsigned

Build setting name	Build setting title
GCC_CW_ASM_SYNTAX	CodeWarrior-Style Inline Assembly
GCC_DEBUGGING_SYMBOLS	Level of Debug Symbols
GCC_DYNAMIC_NO_PIC	Generate Position Dependent Code
GCC_ENABLE_ASM_KEYWORD	Allow 'asm', 'inline', 'typeof'
GCC_ENABLE_CPP_EXCEPTIONS	Enable C++ Exceptions
GCC_ENABLE_CPP_RTTI	Enable C++ Runtime Types
GCC_ENABLE_FIX_AND_CONTINUE	Fix & Continue
GCC_ENABLE_OBJC_EXCEPTIONS	Enable Objective-C Exceptions
GCC_ENABLE_OBJC_GC	Enable Objective-C Garbage Collection
GCC_ENABLE_PASCAL_STRINGS	Recognize Pascal Strings
GCC_ENABLE_TRIGRAPHS	Enable Trigraphs
GCC_FAST_MATH	Relax IEEE Compliance
GCC_FAST_OBJC_DISPATCH	Accelerated Objective-C Dispatch
GCC_GENERATE_DEBUGGING_SYMBOLS	Generate Debug Symbols
GCC_GENERATE_PROFILING_CODE	Generate Profiling Code
GCC_INPUT_FILETYPE	Compile Sources As
GCC_MODEL_CPU	Target CPU
GCC_MODEL_PPC64	Use 64-bit Integer Math
GCC_MODEL_TUNING	Instruction Scheduling
GCC_NO_COMMON_BLOCKS	No Common Blocks
GCC_NO_NIL_RECEIVERS	Assume Non-nil Receivers
GCC_ONE_BYTE_BOOL	Use One Byte 'bool'
GCC_OPTIMIZATION_LEVEL	Optimization Level
GCC_PFE_FILE_C_DIALECTS	C Dialects to Precompile
GCC_PRECOMPILE_PREFIX_HEADER	Precompile Prefix Header
GCC_PREFIX_HEADER	Prefix Header
GCC_PREPROCESSOR_DEFINITIONS	Preprocessor Macros

Build setting name	Build setting title
GCC_REUSE_STRINGS	Make Strings Read-Only
GCC_SHORT_ENUMS	Short Enumeration Constants
GCC_STRICT_ALIASING	Enforce Strict Aliasing
GCC_TREAT_NONCONFORMANT_CODE_ERRORS_AS_WARNINGS	Treat Nonconformant Code Errors as Warnings
GCC_TREAT_WARNINGS_AS_ERRORS	Treat Warnings as Errors
GCC_UNROLL_LOOPS	Unroll Loops
GCC_WARN_ABOUT_MISSING_PROTOTYPES	Missing Function Prototypes
GCC_WARN_ABOUT_RETURN_TYPE	Mismatched Return Type
GCC_WARN_ALLOW_INCOMPLETE_PROTOCOL	Incomplete Objective-C Protocols
GCC_WARN_CHECK_SWITCH_STATEMENTS	Check Switch Statements
GCC_WARN_EFFECTIVE_CPLUSPLUS_VIOLATIONS	Effective C++ Violations
GCC_WARN_FOUR_CHARACTER_CONSTANTS	Four Character Literals
GCC_WARN_HIDDEN_VIRTUAL_FUNCTIONS	Hidden Virtual Functions
GCC_WARN_INHIBIT_ALL_WARNINGS	Inhibit All Warnings
GCC_WARN_NON_VIRTUAL_DESTRUCTOR	Non-virtual Destructor
GCC_WARN_INITIALIZER_NOT_FULLY_BRACKETED	Initializer Not Fully Bracketed
GCC_WARN_MISSING_PARENTHESES	Missing Braces and Parentheses
GCC_WARN_PEDANTIC	Pedantic Warnings
GCC_WARN_SHADOW	Hidden Local Variables
GCC_WARN_SIGN_COMPARE	Sign Comparison
GCC_WARN_TYPECHECK_CALLS_TO_PRINTF	Typecheck Calls to printf/scanf
GCC_WARN_UNINITIALIZED_AUTOS	Uninitialized Automatic Variables
GCC_WARN_UNKNOWN_PRAGMAS	Unknown Pragma
GCC_WARN_UNUSED_FUNCTION	Unused Functions
GCC_WARN_UNUSED_LABEL	Unused Labels
GCC_WARN_UNUSED_PARAMETER	Unused Parameters
GCC_WARN_UNUSED_VALUE	Unused Values

Build setting name	Build setting title
GCC_WARN_UNUSED_VARIABLE	Unused Variables
OTHER_CFLAGS	Other C Flags
OTHER_CPLUSPLUSFLAGS	Other C++ Flags
WARNING_CFLAGS	Other Warning Flags

Table 25-5 maps the build setting titles with the corresponding build setting names of the compiler-related build settings.

Table 25-5 GNU C/C++ compiler build settings by build setting title

Build setting title	Build setting name
Accelerated Objective-C Dispatch	GCC_FAST_OBJC_DISPATCH
Allow 'asm', 'inline', 'typeof'	GCC_ENABLE_ASM_KEYWORD
Assume Non-nil Receivers	GCC_NO_NIL_RECEIVERS
C Dialects to Precompile	GCC_PFE_FILE_C_DIALECTS
C Language Dialect	GCC_C_LANGUAGE_STANDARD
'char' Type Is Unsigned	GCC_CHAR_IS_UNSIGNED_CHAR
Check Switch Statements	GCC_WARN_CHECK_SWITCH_STATEMENTS
CodeWarrior-Style Inline Assembly	GCC_CW_ASM_SYNTAX
Compile Sources As	GCC_INPUT_FILETYPE
Effective C++ Violations	GCC_WARN_EFFECTIVE_CPLUSPLUS_VIOLATIONS
Enable AltiVec Extensions	GCC_ALTIVEC_EXTENSIONS
Enable C++ Exceptions	GCC_ENABLE_CPP_EXCEPTIONS
Enable C++ Runtime Types	GCC_ENABLE_CPP_RTTI
Enable Objective-C Exceptions	GCC_ENABLE_OBJC_EXCEPTIONS
Enable Objective-C Garbage Collection	GCC_ENABLE_OBJC_GC
Enable Trigraphs	GCC_ENABLE_TRIGRAPHS
Enforce Strict Aliasing	GCC_STRICT_ALIASING
Fix & Continue	GCC_ENABLE_FIX_AND_CONTINUE
Four Character Literals	GCC_WARN_FOUR_CHARACTER_CONSTANTS

Build setting title	Build setting name
Generate Profiling Code	GCC_GENERATE_PROFILING_CODE
Generate Debug Symbols	GCC_GENERATE_DEBUGGING_SYMBOLS
Generate Position Dependent Code	GCC_DYNAMIC_NO_PIC
Hidden Local Variables	GCC_WARN_SHADOW
Hidden Virtual Functions	GCC_WARN_HIDDEN_VIRTUAL_FUNCTIONS
Incomplete Objective-C Protocols	GCC_WARN_ALLOW_INCOMPLETE_PROTOCOL
Inhibit All Warnings	GCC_WARN_INHIBIT_ALL_WARNINGS
Initializer Not Fully Bracketed	GCC_WARN_INITIALIZER_NOT_FULLY_BRACKETED
Instruction Scheduling	GCC_MODEL_TUNING
Level of Debug Symbols	GCC_DEBUGGING_SYMBOLS
Make Strings Read-Only	GCC_REUSE_STRINGS
Mismatched Return Type	GCC_WARN_ABOUT_RETURN_TYPE
Missing Braces and Parentheses	GCC_WARN_MISSING_PARENTHESES
Missing Function Prototypes	GCC_WARN_ABOUT_MISSING_PROTOTYPES
No Common Blocks	GCC_NO_COMMON_BLOCKS
Non-virtual Destructor	GCC_WARN_NON_VIRTUAL_DESTRUCTOR
Optimization Level	GCC_OPTIMIZATION_LEVEL
Other C Flags	OTHER_CFLAGS
Other C++ Flags	OTHER_CPLUSPLUSFLAGS
Other Warning Flags	WARNING_CFLAGS
Pedantic Warnings	GCC_WARN_PEDANTIC
Precompile Prefix Header	GCC_PRECOMPILE_PREFIX_HEADER
Prefix Header	GCC_PREFIX_HEADER
Preprocessor Macros	GCC_PREPROCESSOR_DEFINITIONS
Recognize Pascal Strings	GCC_ENABLE_PASCAL_STRINGS
Relax IEEE Compliance	GCC_FAST_MATH
Short Enumeration Constants	GCC_SHORT_ENUMS

Build setting title	Build setting name
Sign Comparison	GCC_WARN_SIGN_COMPARE
Target CPU	GCC_MODEL_CPU
Treat Nonconformant Code Errors as Warnings	GCC_TREAT_NONCONFORMANT_CODE_ERRORS_AS_WARNINGS
Treat Warnings as Errors	GCC_TREAT_WARNINGS_AS_ERRORS
Typecheck Calls to printf/scanf	GCC_WARN_TYPECHECK_CALLS_TO_PRINTF
Uninitialized Automatic Variables	GCC_WARN_UNINITIALIZED_AUTOS
Unknown Pragma	GCC_WARN_UNKNOWN_PRAGMAS
Unroll Loops	GCC_UNROLL_LOOPS
Unused Functions	GCC_WARN_UNUSED_FUNCTION
Unused Labels	GCC_WARN_UNUSED_LABEL
Unused Parameters	GCC_WARN_UNUSED_PARAMETER
Unused Values	GCC_WARN_UNUSED_VALUE
Unused Variables	GCC_WARN_UNUSED_VARIABLE
Use 64-bit Integer Math	GCC_MODEL_PPC64
Use One Byte 'bool'	GCC_ONE_BYTE_BOOL

Table 25-6 shows GCC 4.0 build settings.

Table 25-6 GCC 4.0 build settings by build setting title

Build setting title	Build setting name
Auto-vectorization	GCC_AUTO_VECTORIZATION
Feedback-Directed Optimization	GCC_FEEDBACK_DIRECTED_OPTIMIZATION
Symbols Hidden by Default	GCC_SYMBOLS_PRIVATE_EXTERN
Inline Functions Hidden	GCC_INLINES_ARE_PRIVATE_EXTERN

Per-File Compiler Flags

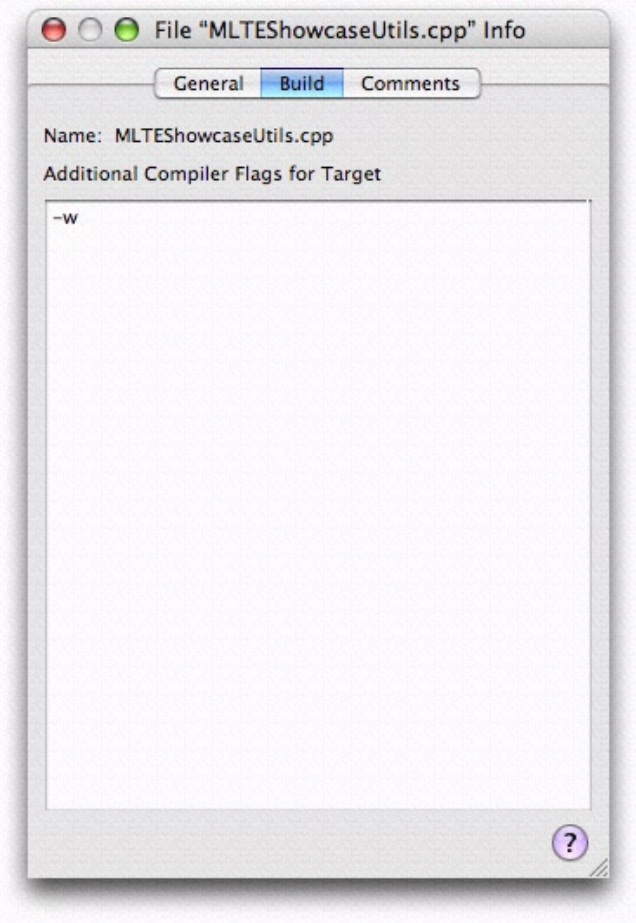
The build system provides facilities for customizing the build process for groups of files assigned to specific build phases. It includes the Other C Flags (OTHER_CFLAGS), Other C++ Flags (OTHER_CPLUSPLUSFLAGS), Other Warning Flags (WARNING_CFLAGS), and Preprocessor Macros (GCC_PREPROCESSOR_DEFINITIONS) build settings, among many others, to customize the invocation of the compiler when processing C-based

source files. However, sometimes it's necessary to specify compiler flags on a per-file basis. For example, in a project that treats warnings as errors in general, you may have a set of cross-platform source files where it's more convenient to allow some warnings to remain.

To set compiler flags for a file, select the source file and open an inspector or Info window. Click Build to open the Build pane, as shown in Figure 25-12. Type any compiler flags you wish to assign to the file in the Additional Compiler Flags for Target field. Multiple flags should be separated by spaces. You can use multiple selection to assign compiler flags to a subset of the files in a target. Note that the Build pane is only available for files containing source code.

For each target that a file belongs to, Xcode maintains a separate list of compiler flags assigned to a file. To specify compiler flags for use with a file when it is built as part of a particular target, select the file from the appropriate build phase in that target and open an inspector window, as described above. If you open an inspector window for the file from any other source or smart group in the project window, Xcode assumes you want to assign compiler flags to the file in the active target; thus, the Build pane is only available if the file is part of the active target.

Figure 25-12 The Build pane of the file inspector



When the build system constructs the command-line invocation for the tool that processes the source file, it adds the additional compiler flags assigned to the file to the invocation. The build system doesn't validate the flags you add; that is, it doesn't test whether the compiler actually supports the options you add, and it doesn't investigate whether the options you add conflict with the ones it generates. If you add a group of

compiler options and flags for a file and the file no longer compiles, you should remove all the flags you added and add them back one at a time, making sure the file compiles after you add each flag. You should also consult the tool's documentation to learn about possible conflicts.

The additional compiler flags specified for a file will always be used when the file is processed as part of a build and cannot be overridden in any of the build setting layers. See "Build Settings" (page 267) for information on build settings.

Build Styles

A build style is a variation on a target that allows you to override some of the build settings in a target without creating a whole new target. While a target contains a list of files, build rules, build phases, and build settings, a build style contains only build settings. You can apply the same build style to all targets in your project. Build styles are a flexible tool for quickly "tweaking" your product or for saving different groups of build settings to apply to a target depending on the current circumstances.

This article provides a general explanation of build styles, describes the predefined build styles you get when you create an Xcode project, and explains how to define your own build styles.

Overview of Build Styles

Build styles allow you to build two or more "flavors" or styles of a product without having to change build settings in the target or create separate targets for each product flavor. When you perform a build, the build settings defined in a build style modify or add to the group of build settings defined in the targets being built.

A common use of build styles is to build a given target differently to create development and deployment versions of a product. There are a handful of build settings—for example, optimization settings for the compiler—whose values are different, depending upon whether you are building a product for development purposes or to deploy to customers. In each situation, the target you build is identical in every other way—files, build phases, and build rules—because the product you want to generate for each is essentially the same. The only difference is in how the source files of the target are processed to build that product, the "style," if you will, in which the target is built.

If you were to create two targets—one for debugging and one for the final build—you would have to remember to keep both targets in sync. When you added a file to one target, you would have to remember to add it to the other, and so forth. With a build style, only the build settings defined in the build style are overridden. A build style does not override a target's build phases or build rules. It is much simpler to create build styles containing the build settings whose values you wish to change and then build with the appropriate build style.

Build styles are defined and used on a per-project basis. When you initiate a build, Xcode builds the active target, and any targets it depends on, with the active build style. Whatever build settings the active build style contains override any values assigned to those build settings in the target. See "Build Settings" (page 267) for details.

New Xcode projects contain two predefined build styles, Development and Deployment. You can edit those build styles or define new build styles of your own.

In addition to the Development and Deployment build styles present in all project types in Xcode, you may need to add a special build style to satisfy particular needs, such as configuring an environment to measure the performance of your application. In such a build style you may add build settings that include the addition of a library or framework that gathers and logs performance-related information. You may also need to target your application to specific Mac OS X versions. In that case, you may need to build your application slightly

differently for each version. For example, you can have build styles named Mac OS X 10.2 and a Mac OS X 10.3 that build a product tailored for Mac OS X version 10.2 and Mac OS X version 10.3, respectively, by setting Mac OS X Deployment target (MACOSX_DEPLOYMENT_TARGET), and any other build settings necessary, to the appropriate values.

Predefined Build Styles

A development build of a product may include debugging information to assist you in fixing bugs. However, this extra information can consume valuable space in a user's system. A deployment build should contain only the code necessary to run the application.

Some build settings tell Xcode to add debugging information to an executable or specify whether to optimize its execution speed. Other build settings turn on features such as ZeroLink and Fix and Continue, which are useful only during development.

All Xcode project templates include two build styles, the Development build style and the Deployment build style. By default, the Development build style turns on ZeroLink, Fix and Continue, and debug-symbol generation, among others, while turning off code optimization. The Deployment build style turns off ZeroLink and Fix and Continue. The code-optimization level is set to its highest by default, through the Optimization Level (GCC_OPTIMIZATION_LEVEL) build setting. Note that the Deployment build style doesn't turn on Deployment Location (DEPLOYMENT_LOCATION); therefore, the product is not copied to the location where it would be installed on a user's system. It also doesn't turn on Deployment Postprocessing (DEPLOYMENT_POSTPROCESSING), which specifies whether to strip binaries and whether to set their permissions to standard values.

For more information on building products for deployment, see "Building From the Command Line" (page 312).

Editing Build Styles

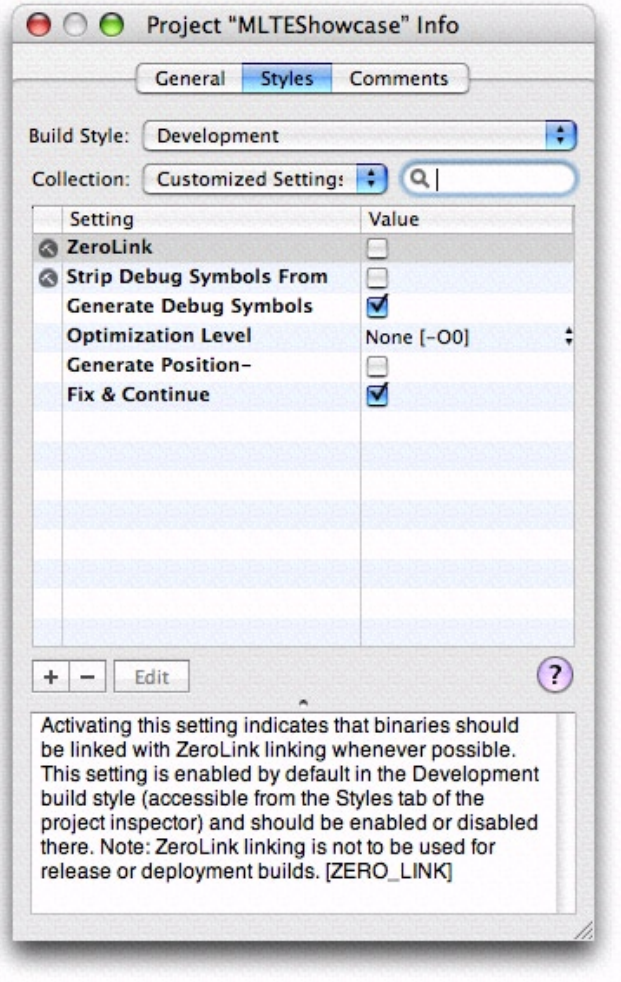
Generally, the two default build styles are enough for most people. You can use them as they are or add additional build settings to them. For example, you can specify that the Development build style display more compiler warnings.

If you're thinking of creating a new target, consider whether it might be best to create a new build style instead. In general, create a new target to create a new product, and create a new build style to modify how a target is built. If you're creating targets that differ only in their build settings, consider creating one target and several build styles. If you need to create targets that differ in other ways—such as build phases or information property list entries—you need to create separate targets for each.

Viewing Build Styles for a Project

To view the build styles in your project, select your project, open an Info window. and click Styles to open the Styles pane, shown in Figure 26-1. You can also choose Project > Edit Active Build Style to open the Styles pane of the project inspector and select the active build style.

Figure 26-1 The Styles pane



The build styles defined in your project are listed in the Build Style pop-up menu at the top of the Styles pane. To view the contents of a build style, select it from this menu. You can see the build settings defined in the build style and their specifications in the table below the Build Style menu.

The **active build style** is the build style applied to the active target, and to any targets it depends upon, when you build a project. Choose Project > Set Active Build Style or use the Active Build Style toolbar item in the Build Results window to change the active build style. You can also customize the project window toolbar to include the Active Build Style item; this allows you to change the active build style directly from the project window. To customize the toolbar, choose View > Customize Toolbar and drag the Active Build Style item to the toolbar.

Adding and Deleting Build Styles

Although the Development and Deployment build styles are sufficient in most cases, you may find that you wish to create your own build style. You can add and remove build styles from your project in the Styles pane of the inspector window. To create a new build style choose New Build Style from the Build Style pop-up menu. You can duplicate existing build styles as well.

To remove a build style from your project:

- Choose Edit Build Styles from the Build Style pop-up menu.
- Select the build style you want to remove in the resulting dialog and click the Delete button.

Modifying Build Settings in a Build Style

You can modify the value of a build setting in a build style in the same way that you change the value of a build setting in a target, described in "Editing Build Setting Specifications" (page 280).

To add a new build setting to the build style, click the plus (+) symbol in the lower left corner of the build settings table. Double-click in the Setting column and type the name of the build setting, then double-click in the Value column and type the value.

To delete a build setting from a build style, select that build setting in the build settings table and click the minus sign (-) in the lower-left corner.

Building a Product

To translate the source files and the instructions in a target into a product, you must build that target. You can build from the Xcode application or from the command-line, using `xcodebuild`. Building from the application provides detailed feedback about the progress of the build operation and integration with the Xcode user interface. For example, when you build from the Xcode application, you can easily jump from an error message to its location in a source file, make the fix and try the build again. Building from the command line lets you easily automate builds of a large number of targets across multiple projects.

This chapter describes how to initiate a build from the Xcode application or from the command line; describes build locations in Xcode and shows you how to create a shared build folder; and shows you how to view build status, errors, and warnings in Xcode.

Build Locations

When Xcode builds a target, it generates intermediate files, such as object files, as well as the product defined by the target. As you build software with Xcode, you need to know where Xcode places the output of a build. For example, suppose you have an application in one project that depends on a library created by a second project. When building the application, Xcode must be able to locate the library to link it into the application.

By default, Xcode places both the build products and the intermediate files that it generates in the build folder inside of your project directory. If the software you are developing is contained in a single project, this default location is probably fine. However, if you have many interdependent targets—particularly if these targets are divided across multiple projects—you'll need a shared build folder to ensure that Xcode can automatically find and use the product created by each of those targets.

Xcode lets you control where the results of a build are placed. In the Building pane of the Xcode Preferences window, you can specify where Xcode should put build products—that is, the products of a build—for all projects that you create. You can override this location on a per-project basis. Therefore, a useful approach is to set your Building preferences to support your most common behavior, then change that behavior in a particular project when you need a different behavior.

For example, if you are developing a group of related applications, libraries, plug-ins, and so on, you might set a shared build location in Xcode Preferences. Then the output of each project will end up in the same location, and can be accessed when building the other projects (such as the previously-mentioned application that needs to link to a separately-built library).

Xcode also supports the concept of an installation location, which is supported by the Deployment Location and Installation Path build settings. If you're building a Deployment build and you turn on the Deployment Location build setting and you supply a path for the Installation Path build setting, the built product is placed at the specified location.

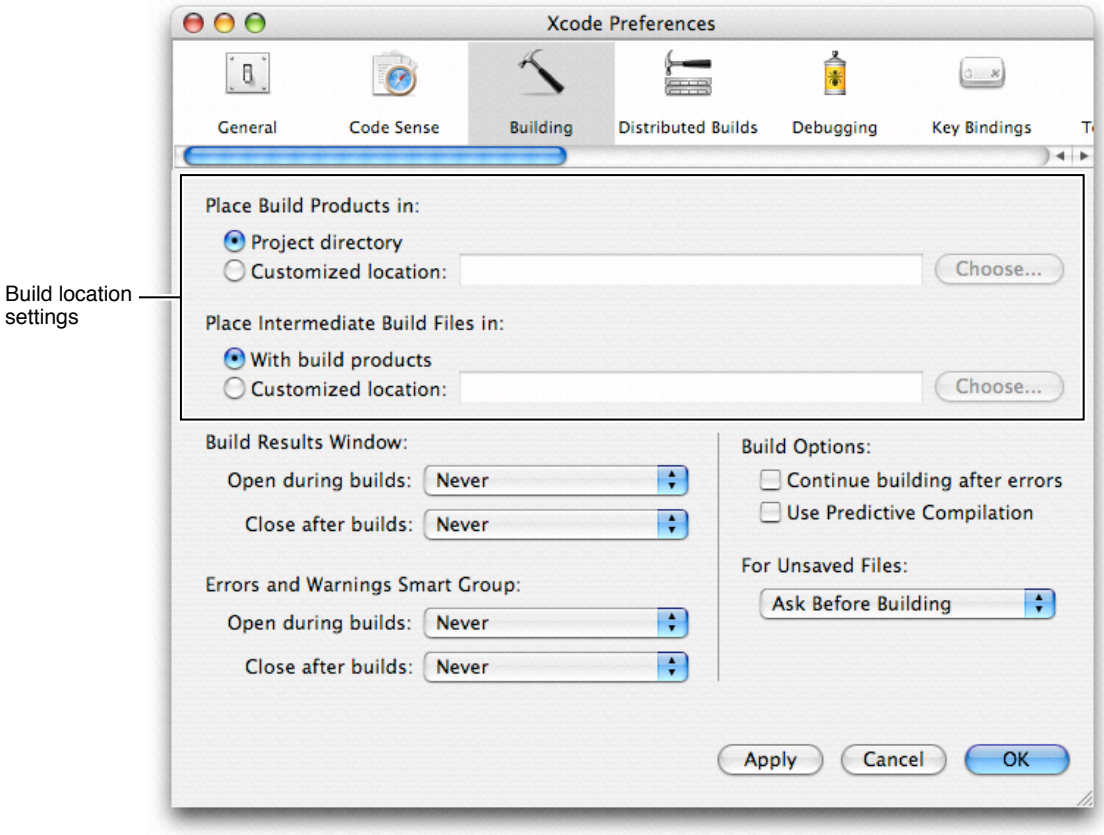
Changing the Default Build Location for All Projects

When you first start up Xcode as a new user, Xcode asks you to specify the directories in which it places the files generated by the build system, both intermediate files and built products. This default build location is used for all new projects that you create.

If you don't supply a build location, Xcode sets the path to the build folder to `/`, indicating a relative path in the project directory. Xcode creates a build folder in the project directory and places any build products in this build folder; by default, it also places intermediate files in a subdirectory of the build folder.

You can change the default build location used for projects that you create in the Building pane of the Xcode Preferences window. To open this pane, choose Xcode > Preferences and click Building. Figure 27-1 shows the Building pane of the Xcode Preferences window.

Figure 27-1 Specifying the default location for build results



To specify the default location for build products, use the "Place Build Products in" options. These are:

- Project directory. Xcode places build products in the `build` folder inside of the project directory. This option is set in Xcode by default.
- Customized location. Xcode places build products in the folder identified in this field. Type the full path to the folder, or click Choose and navigate to the folder you want to use.

Choose this option to create a shared build folder into which Xcode places the build products of all of your projects. When Xcode builds a target, it looks first in the project's build location for any files it depends upon. For example, given an application target that links against a library built by a target in different project, using a shared build folder, Xcode automatically finds the most recently built version of the library and links against it when it builds the application. Without a shared build folder, you must manually add the library's build folder into the application's library search path.

People who use the same project can set their own build folders. This means that every person on a team can have his or her own shared build folder.

Note: If you use a shared build folder, your products must have different names. Otherwise, products with the same name overwrite each other.

To set the default location used for the intermediate files—such as object files—generated by Xcode during a build, use the "Place Intermediate Build Files in" options. Choose one of the following:

- With build products. Xcode places the intermediate build files at the same location as the build products, whether this is in the project directory or in a separate folder that you have specified. This option is set in Xcode by default.
- Customized location. Xcode places intermediate files in the folder identified in this field. Type the full path to the folder, or click Choose and navigate to the folder you want to use.

Overriding the Default Build Location for a Project

Each time you create a new project, Xcode sets the build location for that project to the default build location specified in the Building pane of the Xcode Preferences window. You can, however, override this default build location on a per-project basis. This lets you choose a default build location that works best for most of your projects, and specify another location for the build results of individual projects as needed.

You can override the build location used for an individual project in the General pane of the project inspector. To override the default location for a project's build products, use the options under "Place Build Products In." These are:

- Default build products location. Xcode uses the default location for build products, specified in Xcode Preferences. This option is set by default for each project you create.
- Custom location. Xcode places the build products for the project in the folder specified in this field. Type the full path to the folder in the text field, or click Choose and navigate to the desired location.

To override the default location for a project's intermediate build files, use the options under "Place Intermediate Build Files In." These are:

- Default intermediates location. Xcode uses the default location for intermediate build files, as specified in Xcode Preferences. This option is set by default for each project you create.
- Build products location. Xcode places intermediate build files with the build products, whether this is the default location specified in Xcode Preferences or a separate location specified for the individual project.

- Custom location. Xcode places the intermediate build files for the project in the folder specified in this field. Type the full path to the folder in the text field, or click Choose and select the folder in the resulting dialog.

Building From the Xcode Application

You can perform all of the common build operations directly in the Xcode application. Building in the Xcode application, you can view build system output, see error and warning messages, and jump to the location of an error or warning in source files, all in a single window.

You can perform a full build of the active target and any targets on which it depends, compile a single file, or view the preprocessor output for a file. You can also remove the build products and intermediate files generated by the build system for a target.

Setting the Active Target and Build Style

When you build from the Xcode application, Xcode uses the current, or active, target to determine which product to create. Xcode also applies any build settings defined by the active build style to the target and its dependencies.

Before you start a build, make sure that the target you want to build is the active target. To make a target active, you can choose it from the Active Target pop-up menu or from the Project > Set Active Target menu.

To make a build style active, choose it from the Active Build Style pop-up menu in the Build Results window, or choose it from the Project > Set Active Build Style menu.

To learn more about configuring a target, see "Targets" (page 231); to learn how to configure a build style, see "Editing Build Styles" (page 298).

Initiating a Full Build

When it does a full build of a target, the build system performs all of the tasks specified by the build phases in that target, and in all targets that it depends on. After the initial build of the target, Xcode performs only those actions necessary to update changed files during subsequent builds. For example, if the only change to the target since the last time it was built was a minor edit to a single source code file, Xcode recompiles that file and relinks the object files to create the finished product.

There are several different ways you can initiate a build:

- To build the active target's product, choose Build > Build or click the Build button.
- To build the active target's product and run it if the build succeeds, choose Build > Build and Run, or click the Build and Run button.
- To build the active target's product and start the debugger if the build succeeds, choose Build > Build and Debug, or click the Build and Debug button.

You may encounter errors or warnings when building your target; see "Viewing Errors and Warnings" (page 309) for information on how to find and fix build errors.

If your product does not build properly and there are no error messages, make sure your files have correct dates. Files with invalid dates (before 1970) won't compile correctly.

Viewing Preprocessor Output

You can see the preprocessor output for a C, C++, or Objective-C source file in the active target. To do so, select that file and choose Build > Preprocess.

You can see a list of all `#define` directives in effect for a file by adding the flag `-dM` to the Other C Flags (OTHER_C_FLAGS) build setting in your target. Instead of the normal preprocessor output, Xcode shows all of the macros defined during the execution of the preprocessor (including predefined macros) when you choose Build > Preprocess. For more information on the C preprocessor, see *GNU C Preprocessor*.

Compiling a Single File

A full build can take a long time if you have a large project. You can compile a single file to ensure that it builds correctly without having to rebuild the entire target. To compile a single file

1. Open the file in an editor window or select the file in the Groups & Files list.
2. Choose Build > Compile or type Command-K. You can also Control-click in the editor or on the file in the Groups & Files list to bring up a contextual menu. Choose Compile from this menu to build the file.

The file must be part of the active target. Note that if you are trying to compile a file that is open in an editor, the editor must have focus for the Compile menu item to be available.

Cleaning a Target

As you learned in "Initiating a Full Build" (page 304), after the initial build of a target, Xcode performs only those actions required to update changed files during subsequent builds. You can, however, force Xcode to do a full rebuild of the target by cleaning that target and rebuilding.

When you clean a target, Xcode removes all of the product files, as well as any object files (.o files) or other intermediate files created during the build process. The next time you build, every file in every build phase is processed according to the action associated with that phase.

To clean only the active target, choose Build > Clean or click the Clean button. To clean all targets in your project, choose Build > Clean All Targets or click the Clean All button.

Viewing Build Status

During a build, you want to see how that build is progressing. Especially for long build operations, it is useful to know the status of that operation. When you build from the Xcode application, Xcode displays the build status in the project window status bar. The status bar message lets you know the operation currently being

performed, as in the figure below. It also displays the name of the target and build style used for the build. When the build is complete, Xcode displays the result of the build—whether the build succeeded or failed and whether there were any errors or warnings—on the right side of the status bar.

Figure 27-2 Build status message in the project window

You can click the build message in the status bar to see more detailed information about the build in the Build Results window. You can also click the progress indicator in the status bar during the course of the build to open the Activity Viewer, as described in "Viewing the Progress of Operations in Xcode" (page 74).

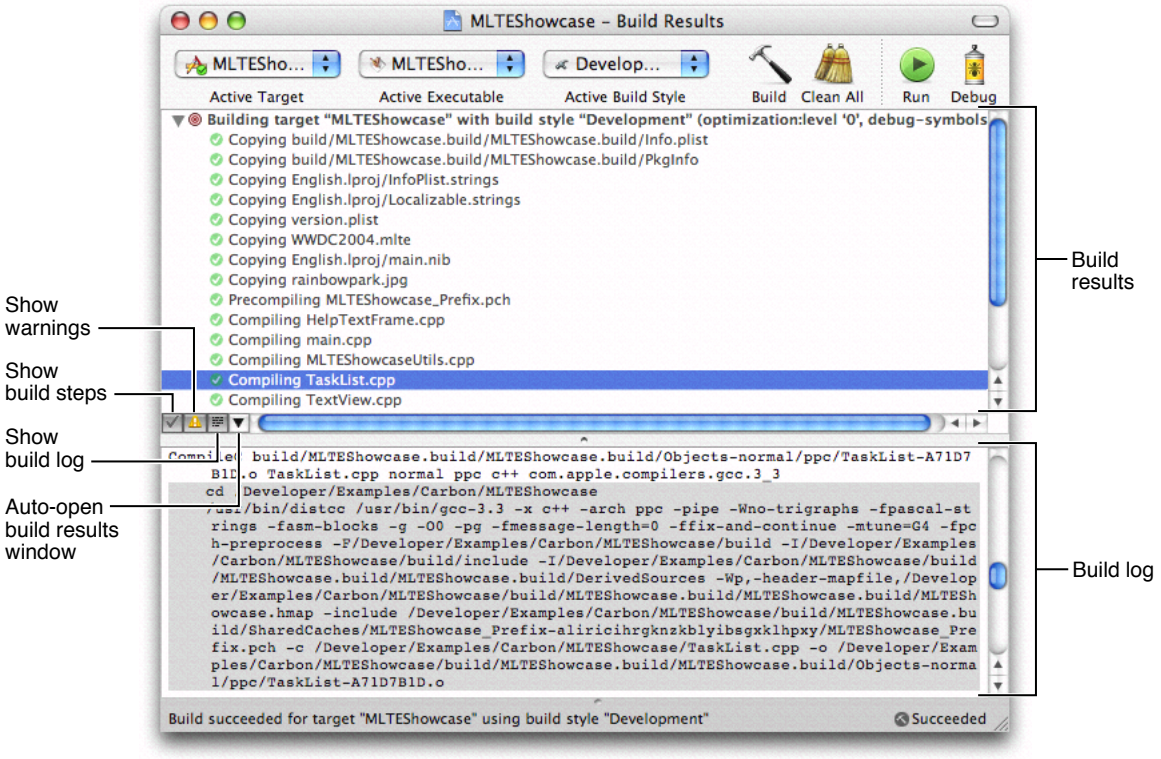
In addition, Xcode displays a progress indicator which shows the status of the build in its dock icon. If an error or warning occurs, Xcode indicates the number of errors or warnings with a red badge on the dock icon.

Viewing Detailed Build Results

The Build Results window lets you see a more detailed account of the progress of a build. It shows each step of the build process, as well as the full output of the build system, and can take you directly to the source of any errors or warnings. To open the Build Results window choose Build > Build Results or click the build result message—"Failed," "Succeeded" and so on—on the right side of the project window status bar. Figure 27-3 shows the Build Results window.

Note: If you are using the All-In-One project window layout, this opens the Build pane in the build page. You can also navigate to this page by clicking the hammer icon in the page control and clicking Build. The Build pane shows the same contents as the Build Results window.

Figure 27-3 The Build Results window



Here's what you see in the Build Results window:

- Toolbar. The toolbar of the Build Results window contains buttons that let you perform common build-related tasks, such as building, cleaning a target, and running or debugging the built product. By default, they also contain pop-up menus to let you change the active target, active executable, and active build style.
- Status Bar. The status bar displays the current status of the build. It contains the same build information as the status bar of the project window.
- Build results. This pane shows status output from the Xcode build system. You can control what information is shown here with the view options described below.
- View options. The buttons below the build results pane let you control how to view detailed build results. These options are as follows:
  - The build steps button, indicated by a checkmark icon, determines whether each of the individual build steps is shown as part of the detailed build results. Select this option to display each of the individual steps used to build the product, such as compiling and copying files. Otherwise, Xcode shows only those build steps that produce warnings or errors.

- The show warnings button, indicated by a warning icon, controls whether Xcode shows warning messages in the detailed build results. If this option is selected, Xcode displays both error and warning messages; otherwise, Xcode shows only error messages.
- The build log button, indicated by an icon showing console text, opens and closes the build log, described below. Select this option to reveal the build log; deselect it to close the build log.
- The arrow button brings up a menu that lets you choose when to display detailed build results. You can have Xcode automatically show and hide the Build Results window in the course of a build. To specify when detailed build results are shown, choose an item under "Temporarily Open During [*project name*] Builds." To specify when detailed build results are hidden, choose an item under "Temporarily Hide After [*project name*] Builds." These options are described further in "Specifying When Detailed Build Results are Shown" (page 308).

  Choosing an item from this menu affects only the current project. Changes made here do not persist across Xcode sessions. To set the default behavior of the Build Results window for all projects choose Open Global Build Preferences to open Xcode Preferences.

- Build log. The build log displays all of the commands used to build your target and the outputs of those commands, including compiler invocations. For example, you can see the flags passed to the compiler in this log. You can show and hide the build log using the build log button. When you select a build step in the top pane of the build window, Xcode selects the corresponding command in the build log.
- Editor. The Build Results window includes an attached editor. Selecting an error or warning message in the build results pane opens the file to the line containing the error or warning in the editor.

Specifying When Detailed Build Results are Shown

Having to open and close the Build Results window can get repetitive if you do it every time you build. If you know you always want to see the detailed build results when building, you can have Xcode automatically open the Build Results window when you start a build. Likewise, you can have Xcode automatically close the Build Results window when the build is complete. Xcode provides preferences to control the default behavior of the Build Results window. By default, Xcode does not automatically open or close the Build Results window when building.

To change Xcode's default behavior for showing and hiding detailed build results for all projects, choose Xcode > Preferences and click Building. Under "Build Results Window," use the following menus:

- To control when the Build Results window is shown, use the "Open during builds" menu
- To control when the Build Results window is hidden, use the "Close after builds" menu.

You can temporarily override this default behavior for an individual project using the pop-up menu in the Build Results window, as described in "Viewing Detailed Build Results" (page 306).

Choose one of the following options to control when the Build Results window is shown:

- Never. Xcode does not automatically open the Build Results window. This is the default value of this setting.
- Always. Xcode always automatically opens the Build Results window when a build starts.
- On Errors. Xcode automatically opens the Build Results window only if an error is encountered during the build.

- On Errors or Warnings. Xcode automatically opens the Build Results window when an error or warning occurs.

Choose one of the following options to control when the Build Results window is hidden:

- Never. Once opened, the Build Results window stays open until you close it or the project. This is the default value for this setting.
- Always. Xcode automatically closes the Build Results window immediately after the build stops, whether the build was successful or not.
- On Success. Xcode automatically closes the Build Results window when the build successfully completes and the product is built.
- On No Errors. Xcode automatically closes the Build Results window after the build is complete only if there are no errors.
- On No Errors or Warnings. Xcode automatically closes the Build Results window after the build is complete only if there are no errors or warnings.

Viewing Errors and Warnings

The Xcode application lets you easily see any errors or warnings generated during a build. You can view errors and warnings directly in the project window with the Errors and Warnings smart group, or you can view them in the Build Results window. You can also jump directly from the error or warning message to the location of errors or warnings in source files. This lets you quickly fix the problem and try the build again.

Viewing Errors and Warnings in the Project Window

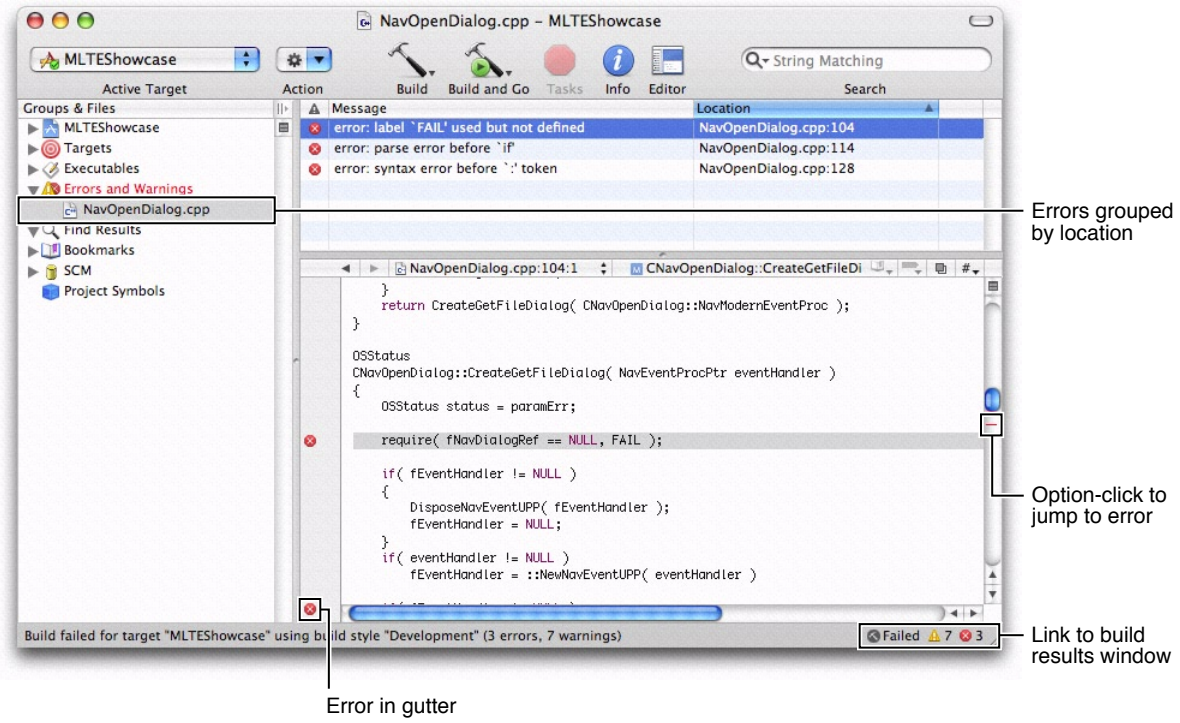
If you prefer to keep the project window in front of you while you are working, Xcode lets you view errors and warnings directly in the project window. You can view errors and warnings in the Errors and Warnings smart group in the Groups & Files list, as shown below. Before you build your project, the Errors and Warnings group is empty; it is populated as errors and warnings occur during the build.

To view errors and warnings, you can:

- Click the disclosure triangle next to the Errors and Warnings smart group. This displays the list of files in which errors or warnings occurred. To see the errors and warning messages for a particular file, select that file.
- Select the Errors and Warnings smart group in the Groups & Files list. This displays all errors and warnings generated during the last build attempt in the detail view.
- Click the error or warning icon that appears on the right of the project window or editor window status bar. This opens the Build Results window and selects the first error or warning in the build steps pane and in the build log.

You can also specify that Xcode automatically open the Errors and Warnings smart group in the Groups & Files view when an error or warning occurs during a build, as described later in this section.

Figure 27-4 Viewing errors and warnings in the project window



When you select the Error and Warnings group or any of its members in the Groups & Files list, the detail view to the right displays the error and warning messages and the location at which each occurred. The icon next to each item in the detail view identifies it as an error or a warning.

Similar to the Build Results window, you can have Xcode automatically show and hide the contents of the Errors and Warnings smart group. To control when Xcode hides and shows the contents of the Errors and Warnings smart group, choose Xcode > Preferences, click Building, and use the menus under "Errors and Warnings Smart Group."

The "Open during builds" menu lets you choose when Xcode automatically selects the Errors and Warnings smart group and discloses its contents in the Groups & Files list. It provides the following options:

- Never. Xcode does not automatically open the Errors and Warnings smart group, even if an error or warning occurs during a build. This is the default.
- Always. Xcode always automatically opens the Errors and Warnings smart group when a build starts.
- On Errors. Xcode automatically opens the Errors and Warnings group only if an error is encountered in the build.
- On Errors and Warnings. Xcode automatically opens the Errors and Warnings group if an error or warning occurs during the build.

The "Close after builds" menu gives you the following options for controlling when Xcode hides the Errors and Warnings group:

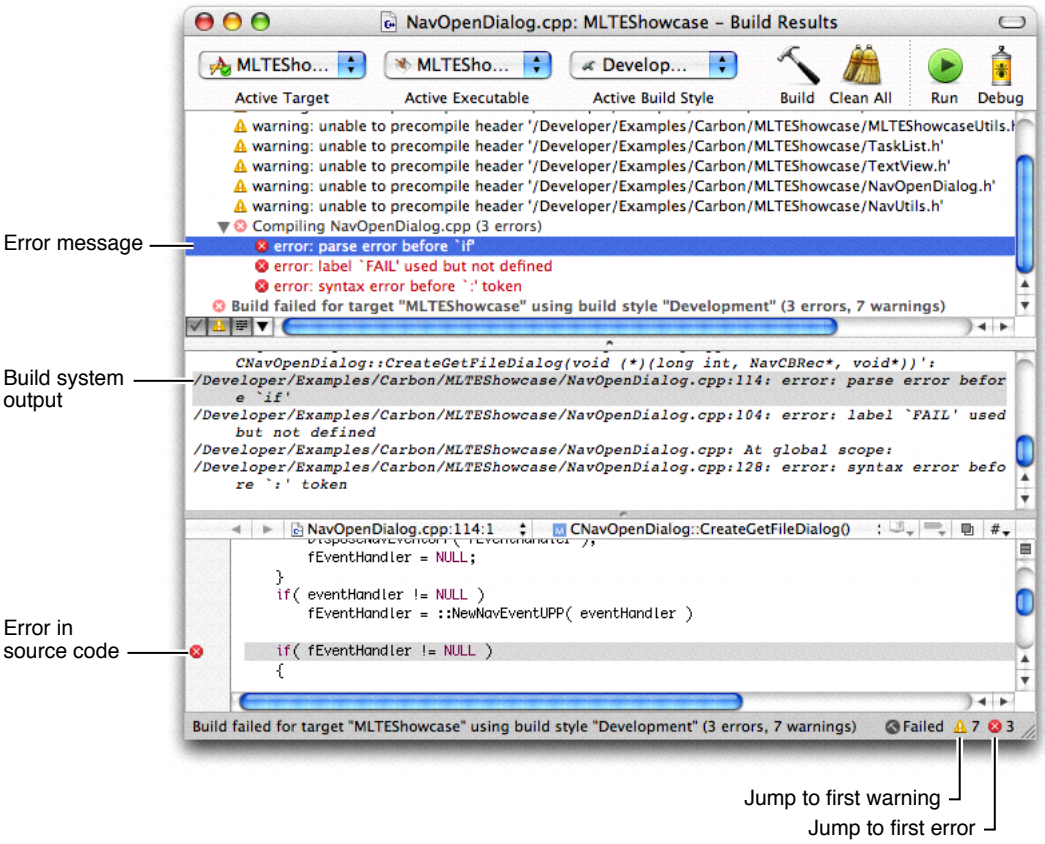
- Never. Once opened, the Errors and Warnings group stays open until you close it. This is the default.
- Always. Xcode automatically closes the Errors and Warnings group immediately after the build stops.

- On Success. Xcode automatically closes the Errors and Warnings group when the build is successfully completed and the product is built.
- On No Errors. If there are no errors, Xcode automatically closes the Errors and Warnings group after the build is complete.
- On No Warnings. If there are no errors or warnings, Xcode automatically closes the Errors and Warnings group after the build is complete.

Viewing Errors and Warnings in the Build Results Window

The Build Results window also displays all the warnings and errors that occur as Xcode builds your target. The example below shows an error in the Build Results window.

Figure 27-5 An error in the Build Results window



To navigate through warning and error messages even when the Build Results window is hidden, choose Build > Next Build Warning or Error (Command–equals-sign) and Build > Previous Build Warning or Error (Command–plus-sign). This highlights the next error or warning in the Build Results window. It does not, however, bring the Build Results window forward if it is hidden. If you are working in an editor window, choosing Build > Next / Previous Build Warning or Error highlights the line at which the error or warning occurred, opening the related file if it is not already open.

You can easily copy error and warning messages, so you can include them in email messages or other documents. To copy error and warning messages, select the messages and drag them to another document.

Viewing Source Code for an Error or Warning

From either the project window or the Build Results window, you can jump directly to the source code associated with an error or warning. In the project window, select the error or warning in the detail view; in the Build Results window, select the error or warning in the build steps pane. If the option to automatically disclose the attached editor is enabled, Xcode reveals the editor in the project or Build Results window and opens the source file to the location of the problem. Otherwise, you can double-click the error or warning to open the file in a separate editor window.

Xcode also displays error and warning icons in the gutter of the editor, next to the lines at which the errors and warnings occurred. This allows you to easily spot the errors and warnings in a file, either from the attached editor in the project or Build Results windows, or from a dedicated editor window. Marks in the scrollbar of an editor let you quickly scroll to the location of an error or warning; you can Option-click on a mark to jump to that location. If you pause with the mouse above an error or warning in the status bar, the detail view, or the gutter, Xcode displays a tooltip with the full text of the error message.

Controlling Errors and Warnings

You can choose what Xcode does when it first encounters an error while building a project. To choose whether Xcode stops building when it encounters an errors or continues to compile the next file in the target, choose Xcode > Preferences and click Building. Use the "Continue building after errors" checkbox in the Build Options.

Building From the Command Line

In addition to building your product from within the Xcode application, you can use `xcodebuild` to build a target from the command line. Building from the command line gives you additional flexibility compared to building from within the Xcode application that may be useful in certain circumstances. For example, using the `xcodebuild` tool, you can create a script that automatically builds your product at a specific time or build targets from multiple projects at the same time.

Note: If you need root privileges to install a product in its deployment location, you must build the product with the `xcodebuild` tool, because Xcode cannot grant you these privileges.

The `xcodebuild` tool reads your `.xcode` project bundle and uses the target information it finds there to build a product. However, there are differences between building within Xcode and building from the command line:

- When you build within Xcode, it uses the active target and build style. When you build from the command line, `xcodebuild` uses the first target in the project's target list and no build style, unless you specify a target or build style with a command-line option.
- If you run `xcodebuild` as the root user, the preferences you set in the Xcode Preferences window are not used. Preferences are stored per user, and there are no preferences stored for the root user (unless you logged in as root and used Xcode at some point).

To build a target using `xcodebuild`, use the `cd` command to change to your project's directory and enter the `xcodebuild` command with any command-line options you wish to specify. The project's directory contains your project's `.xcode` bundle. For example, if your project is in `~me/Projects/MyProj`, enter `cd ~me/Projects/MyProj`.

You can use `xcodebuild` to build a product suited for deployment and install the product in its final destination path. To do that, use the `xcodebuild` tool with the `install` option, which places the product in the distribution root specified by the DSTROOT, Installation Path (INSTALL_PATH), and Deployment Location (DEPLOYMENT_LOCATION) build settings. For example, to install a framework in `/Library/Frameworks`, configure the build settings as shown in Table 27-1.

Table 27-1 Build settings for installing a framework in the local domain

Build setting name	Value
DSTROOT	`/`
INSTALL_PATH	`$(LOCAL_LIBRARY_DIR)/Frameworks`
DEPLOYMENT_LOCATION	`YES`

The DSTROOT build setting can be set only in the `xcodebuild` command-line specification:

```
% sudo xcodebuild install -buildstyle Deployment DSTROOT=/
INSTALL_PATH=/Library/Frameworks  DEPLOYMENT_LOCATION=YES
```

See the `xcodebuild` man page for information on the available options and command usage. For details on framework placement, see Mac OS X Frameworks.

Linking

The Link Binary With Libraries build phase in Xcode projects links frameworks and libraries with object files to produce a binary file. Source files that use code in a framework or a library must include a reference to the appropriate programming interface contained in them.

Libraries and frameworks are linked to object files when building an executable file. However, this process is slow and can detract from the development experience. Xcode provides a feature, called ZeroLink, that eliminates the link step while you work on a project; see "Using ZeroLink" (page 320) for details.

Specifying the Search Order of External Symbols

The order in which frameworks and libraries are listed in the Link Binary With Libraries build phase specifies the order in which external symbols are resolved by the static linker at build time and the dynamic linker at runtime. When either of the linkers encounters an undefined external symbol, they look for the symbol starting with the first framework or library listed in the build phase.

When a program is built, the static linker replaces references to external symbols with the addresses of the symbols in the referenced libraries (this is called **prebinding**), or tells the dynamic linker to resolve the references when a program is loaded or when a symbol is referenced. Having the dynamic linker resolve references to external symbols at runtime provides the most flexibility, as a program can link with new versions of the symbols as they become available. However, this approach is not recommended for all situations, as linking with newer versions of a method or a function may cause problems during a program's execution.

In addition, how frameworks and libraries are listed in a Link Binary With Libraries build phase, tells the static linker the approach (or the semantics) to use when binding or resolving references to external symbols defined in libraries.

Placing static libraries after dynamic libraries in the Link Binary With Libraries build phase, ensures that the static linker binds references to external symbols defined in static libraries at build time, even when newer versions of the static libraries the application originally was linked with are present in the user's system.

When static libraries are listed before a dynamic library in the Link Binary With Libraries build phase, the static linker doesn't resolve references to symbols in those static libraries. Instead, those symbols are resolved by the dynamic linker at runtime. This may cause problems when the static libraries are updated, as the application links to the new, potentially incompatible versions of the symbols instead of the ones the developer intended.

For details on how symbols are resolved, see "Finding Imported Symbols" in "Executing Mach-O Files" in *Mac OS X ABI Mach-O File Format Reference* and the `ld` man page.

Preventing Prebinding

Mac OS X includes a prebinding mechanism used to speed-up application launch in programs that link against dynamic libraries. When a user installs an application or upgrades the operating system, a prebinding agent links the application against new versions of the dynamic libraries. Sometimes, however, you may want to prevent this behavior for specific applications.

To link the binary file, framework, library, or plug-in, so that prebinding is never done on it, you need to add the `-nofixprebinding` option to the linker invocation. To do this, add `-nofixprebinding` to the Other Linker Flags (OTHER_LDFLAGS) build setting. See the `ld` man page for more information.

Linking With System Frameworks

When linking with system frameworks (located in `/System/Library/Frameworks`), include only the umbrella header files in your source files and link only with the appropriate umbrella framework for your application. For example, in a Carbon application that uses the Address Book framework, you would include the following line in the header files of modules that access the Address Book programming interface:

```
#include <Carbon/Carbon.h>
```

You would also add `AddressBook.framework` to the list of files in the Frameworks & Libraries build phase.

Linking to a Dynamic Library in a Nonstandard Location

When you need to link with a custom version of a dynamic library but don't want to replace the standard version of the library, you can use the `-dylib_file` option of the linker to tell it where to find the nonstandard version of the library. Just add `-dylib_file` *standard_library_path* : *nonstandard_library_path* to the Other Linker Flags build setting, where *standard_library_path* is the path to the standard library and *nonstandard_library_path* is the path to the custom version of the library.

Reducing the Number of Exported Symbols

By default, Xcode builds binary files that export all their symbols. To reduce the number of symbols you want to export from a binary file, create an export file and set the Exported Symbols File (EXPORTED_SYMBOLS_FILE) build setting to the name of the file. For more information, see "Minimizing Your Exported Symbols" in *Code Size Performance Guidelines*.

Reducing Paging Activity

To help reduce your application's paging activity at runtime, you may specify an order file to the linker. You do this by setting the Symbol Ordering Flags (SECTORDER_FLAGS) build setting to `-sectorder __TEXT __text <order_file>`. For information on order files, see "Improving Locality of Reference" in *Code Size Performance Guidelines*.

Dead-Code Stripping

The static linker (`ld`) supports the removal of unused code and data blocks from executable files. This process (known as dead-code stripping) helps reduce the overall size of executables, which in turn improves performance by reducing the memory footprint of the executable. It also allows programs to link successfully in the situation where unused code refers to an undefined symbol, something that would normally result in a link error.

Dead-code stripping is not limited to removing only unused functions and executable code from a binary. The linker also removes any unused symbols and data that reside in data blocks. Such symbols might include global variables, static variables, and string data among others.

When dead-code stripping is enabled, the static linker searches for code that is unreachable from an initial set of live symbols and blocks. The initial list of live symbols and blocks may include the following:

- Symbols listed in an exports file; alternatively, the symbols absent from a list of items marked as *not* to be exported. For dynamic libraries or bundles without an exports file, all global symbols are part of the initial live list. See "Preventing the Stripping of Unused Symbols" (page 318) for more information.
- The block representing the default entry point or the symbol listed after the `-e` linker option, either of which identifies the specific entry point for an executable. See the `ld` man page for more information on the `-e` option.
- The symbol listed after the `-init` linker option, which identifies the initialization routine for a shared library. See the `ld` man page for more information.
- Symbols whose declaration includes the `used` attribute. See "Preventing the Stripping of Unused Symbols" (page 318) for more information.
- Objective-C runtime data.
- Symbols marked as being referenced dynamically (via the `REFERENCED_DYNAMICALLY` bit in `/usr/include/mach-o/nlist.h`).

Enabling Dead-Code Stripping in Your Project

To enable dead-code stripping in your project, you must pass the appropriate command-line options to the linker. From Xcode, you add these options in the Build pane of the target inspector; otherwise, you must add these options to your makefile or build scripts. Table 28-2 lists the Xcode build settings that control dead-code stripping. Enabling either of these build settings causes Xcode to build with the corresponding linker option.

Table 28-1 Xcode build settings for dead stripping

Linker option	Build setting
`-dead_strip`	Dead Code Stripping (DEAD_CODE_STRIPPING)
`-no_dead_strip_inits_and_terms`	Don't Dead-Strip Inits and Terms (PRESERVE_DEAD_CODE_INITS_AND_TERMS)

Table 28-1 lists the basic dead-code stripping options.

Table 28-2 Linker options for dead stripping

Linker option	Description
`-dead_strip`	Enables basic dead-code stripping by the linker.
`-no_dead_strip_inits_and_terms`	Prevents all constructors and destructors from being stripped when the `-dead_strip` option is in effect, even if they are not live.

You must recompile all object files using the compiler included with Xcode 1.5 or later before dead-code stripping can be performed by the linker. You must also compile the object files with the `-gfull` option to ensure that the resulting binaries can be properly debugged. In Xcode, change the value of the Level of Debug Symbols (GCC_DEBUGGING_SYMBOLS) build setting to All Symbols (`-gfull`).

Note: The GCC compiler's `-g` option normally defaults to `-gused`, which reduces the size of `.o` files at the expense of symbol information. Although the `-gfull` option does create larger `.o` files, it often leads to smaller executable files, even without dead-code stripping enabled.

Identifying Stripped Symbols

If you want to know what symbols were stripped by the static linker, you can find out by examining the linker-generated load map. This map lists all of the segments and sections in the linked executable and also lists the dead-stripped symbols. To have the linker generate a load map, add the `-M` option to your linker command-line options. In Xcode, you can add this option to the Other Linker Flags build setting.

Note: If you are passing this option through the `cc` compiler driver, make sure to pass this flag as `-Wl,-M` so that it is sent to the linker and not the compiler.

Preventing the Stripping of Unused Symbols

If your executable contains symbols that you know should not be stripped, you need to notify the linker that the symbol is actually used. You must prevent the stripping of symbols in situations where external code (such as plug-ins) use those symbols but local code does not.

There are two ways to tell the linker not to dead strip a symbol. You can include it in an exports file or mark the symbol declaration explicitly. To mark the declaration of a symbol, you include the `used` attribute as part of its definition. For example, you would mark a function as used by declaring it as follows:

```
void MyFunction(int param1) __attribute__((used));
```

Alternatively, you can provide an exports list for your executable that lists any symbols you expect to be used by plug-ins or other external code modules. To specify an exports file from Xcode, use the Exported Symbols File (EXPORTED_SYMBOLS_FILE) build setting; enter the path, relative to the project directory, to the exports file. To specify an exports file from the linker command line use the `-exported_symbols_list`. option. (You can also specify a list of symbols *not* to export using the `-unexported_symbols_list` option.)

If you are using an exports list and building either a shared library, or an executable that will be used with `ld`'s `-bundle_loader` flag, you need to include the symbols for exception frame information in the exports list for your exported C++ symbols. Otherwise, they may be stripped. These symbols end with `.eh`; you can view them with the `nm` tool.

Assembly Language Support

If you are writing assembly language code, the assembler now recognizes some additional directives to preserve or enhance the dead-stripping of code and data. You can use these directives to flag individual symbols or entire sections of assembly code.

Preserving Individual Symbols

To prevent the dead stripping of an individual symbol, use the `.no_dead_strip` directive. For example, the following code prevents the specified string from being dead stripped:

```
.no_dead_strip _my_version_string
.cstring
_my_version_string:
.ascii "Version 1.1"
```

Preserving Sections

To prevent an entire section from being dead stripped, add the `no_dead_strip` attribute to the section declaration. The following example demonstrates the use of this attribute:

```
.section __OBJC, __image_info, regular, no_dead_strip
```

You can also add the `live_support` attribute to a section to prevent it from being dead stripped if it references other code that is live. This attribute prevents the dead stripping of some code that might actually be needed but not referenced in a detectable way. For example, the compiler adds this attribute to C++ exception frame information. In your code, you might use the attribute as follows:

```
.section __TEXT, __eh_frame, coalesced, no_toc+strip_static_syms+live_support
```

Dividing Blocks of Symbols

The `.subsections_via_symbols` directive notifies the assembler that the contents of sections may be safely divided into individual blocks prior to dead-code stripping. This directive makes it possible for individual symbols to be stripped out of a given section if they are not used. This directive applies to all section declarations in your assembly file and should be placed outside of any section declarations, as shown below:

```
.subsections_via_symbols

; Section declarations...
```

If you use this directive, make sure that each symbol in the section marks the beginning of a separate block of code. Implicit dependencies between blocks of code might result in the removal of needed code from the executable. For example, the following section contains three individual symbols, but execution of the code at `_plus_three` ends at the `blr` statement at the bottom of the code block.

```
.text
.globl _plus_three
_plus_three:
addi r3, r3, 1
.globl _plus_two
_plus_two:
addi r3, r3, 1
.global _plus_one
_plus_one:
addi r3, r3, 1
blr
```

If you were to use the `.subsections_via_symbols` directive on this code, the assembler would permit the stripping of the symbols `_plus_two` and `_plus_one` if they were not called by any other code. If this occurred, `_plus_three` would no longer return the correct value because part of its code would be missing. In addition, if `_plus_one` were dead stripped, the code might crash as it continued executing into the next block.

Using ZeroLink

ZeroLink speeds application development time by eliminating the link process from development builds. Instead, Xcode generates an application stub that contains the full paths to the object files that make up the application. At runtime, each object (`.o`) file is linked as it's needed. This works only when running your application within Xcode. You cannot deploy applications using ZeroLink.

To turn ZeroLink on or off, use the ZeroLink (ZERO_LINK) build setting. ZeroLink is enabled by default in the Development build style. If you build with this build style, you automatically get ZeroLink functionality. See "Build Styles" (page 297) for more information on using build styles. ZeroLink works only for native targets.

The following sections explain how you can customize ZeroLink to further reduce application launch times and identify issues you must keep in mind when using ZeroLink.

Customizing ZeroLink

ZeroLink postpones the linking of object files until the last moment possible. However, there are some symbols that, by default, are always resolved (that is, the corresponding object files are linked against the application and the code is executed). These symbols are static initializers in C++ and `+load` methods and categories in Objective-C. You can tell ZeroLink not to search the object files of your application for these symbols to reduce the application's launch time.

There are situations that require the initialization of objects before a program's `main` function is called. For example, a class may declare global variables that can be accessed by other code before the class has a chance to initialize them. In Objective-C, the `+initialize` method may be executed too late (see "Initializing a Class Object" in *The Objective-C 2.0 Programming Language*. The purpose of static initializers in C++ and `+load` methods in Objective-C is to provide developers with a mechanism to initialize variables at the earliest possible point during a program's launch process. In Objective-C–based applications, categories are also loaded before `main` is called.

When you build an application, the static linker adds the standard entry-point function to the main executable file. This function sets up the runtime environment state for the kernel and the application before calling `main`, which involves calling static initializers for C++ code and loading categories and invoking `+load` methods for Objective-C code.

When using ZeroLink, you can further reduce the launch time of the application by postponing the execution of static initializers and `+load` methods, and the loading of categories. But you must be certain that code in your application doesn't rely on static initializers or `+load` methods being called before `main` or on categories being loaded before `main` is called. Otherwise, your application may crash or behave unexpectedly.

There are three linker options you can use to customize ZeroLink in your project. To use these options, add them to the Other Linker Flags (OTHER_LDFLAGS) build setting:

- `-no-run-initializers-before-main`: If an application contains C++ code, the linker looks for static initializers at launch time before calling `main` in all the object files that make up the application and, if it finds any, links the object files containing them into the application. This may slow down application launch. If you're developing an application using C++ and it doesn't depend on static initializers being run before `main`, use this option to prevent ZeroLink from scanning object files in search of static initializers before calling `main`.

  Keep in mind that if there's code that requires that static initializers be run before `main`, your application could crash. This specially true for applications that use static initializers to register code with a registry. If the sole purpose of the static initializers is to perform the registration (that is, if no other code would ever trigger the execution of the static initializers by accessing a global variable, for example), no registration would take place. For example, when reading a serialized file, the reader reads the name of the class of each serialized object and tells the class to reconstruct an instance from the data. In C++ there is no infrastructure to look up a class by name. Instead, a common idiom is for each class capable of serializing and deserializing instances of itself to register its class name and deserialization method address with the reader using a static initializer.

- `-no-load-categories-before-main`: If an application contains Objective-C code, the linker looks for categories at launch time before calling `main` in all the object files that make up the application and, if it finds any, links the object files containing them into the application. As with `-no-run-initializers-before-main`, this may slow application launch. If your application doesn't depend on categories, use this option to prevent ZeroLink from scanning object files in search of categories before calling `main`.

Some developers rely on the use of categories in the implementation of their class hierarchies. In such a model, the implementation of a class spans one or more categories of that class. Using this design model, a program may run incorrectly when the categories that implement additional class functionality are not loaded. With ZeroLink, classes can be loaded when need because there's a direct reference to them, but ZeroLink cannot load categories dynamically. Therefore, if your application depends on categories, it may crash or behave unexpectedly if you use the `-no-load-categories-before-main` flag.

- `-no-run-load-methods-before-main`: Similarly to `-no-load-categories-before-main`, ZeroLink scans object files for `+load` methods in Objective-C code before `main` is invoked to link them to the application. Using this option prevents the scanning of object files for `+load` methods before ZeroLink calls `main`. `+load` methods in Objective-C are similar to static initializers in C++: They are executed early during application launch, before main is called, to perform tasks that must be performed at the earliest possible time. However, the order of execution of `+load` methods is not guaranteed. If your code depends on `+load` methods to be run before `main`, your application may crash or run incorrectly if you use this flag.

When building an application that uses static libraries (`.a` files), each static library is linked to produce a bundle. At runtime, each bundle is loaded on demand. If your application starts slowly while using ZeroLink and has a large number (100 or more) of object files, you can try adding an intermediate static library target, containing the relatively stable parts of your source code. This gives you the best of both worlds: static (build time) linking for stable code and dynamic (runtime) linking for code that changes frequently.

If you want to view information about the loading and linking of object files as your application runs, set the `ZERO_LINK_VERBOSE` environment variable to any value. The information appears in run log of the application or `stderr`.

Caveats When Using ZeroLink

These are some things you should keep in mind when using ZeroLink:

- ZeroLink doesn't support the use of private external symbols; that is symbols declared as `__private_extern__`.

  Private external symbols are visible only to other modules within the same Mach-O file as the modules that contain them. If you use a private external symbol in your project while ZeroLink is turned on, you get an unknown-symbol error when your code tries to access it. For example, if you have the definition `__private_extern__ int my_extern = 800;` in a source file and the declaration `extern int my_extern;` in another source file, when the second module accesses `my_extern`, your application exits with the following log output:

  ```
  ZeroLink: unknown symbol '_my_extern'
  MyApplication has exited due to signal 6 (SIGABRT)
  ```

  For more information on private external symbols, see "Scope and Treatment of Symbol Definitions" in "Executing Mach-O Files" in *Mac OS X ABI Mach-O File Format Reference*.

- When setting breakpoints on methods, you must use full method specifiers, not just selectors. For example, if your project has a class named MyClass with an instance method called `myMethod`, you must specify a breakpoint on `myMethod` like this:

  ```
  -[MyClass myMethod]
  ```

- If you get an error similar to this one:

```
dyld: /Users/user_name/MyApp/build/MyApp.app/Contents/MacOS/MyApp  Undefined
symbols:
Foundation undefined reference to _objc_exception_set_functions  expected to be
 defined in
/System/Library/PrivateFrameworks/ZeroLink.framework/Versions/A/Resources/libobjc.A.dylib
```

delete
`/System/Library/PrivateFrameworks/ZeroLink.framework/Versions/A/Resources/libobjc.A.dylib`.

You would get this error if you install the Xcode tools in a system with a prerelease version of Mac OS X. The `libobjc.A.dylib` file contains a developmental copy of the Objective-C runtime. It's not necessary for normal development.

Optimizing the Edit-Build-Debug Cycle

Xcode offers a number of features you can take advantage of to decrease build time for your project. For example, you can use distributed builds to shorten the time it takes to build your whole project or multiple projects. ZeroLink and predictive compilation, on the other hand, improve turnaround time for single file changes, thereby speeding up the edit–build–debug cycle. All of these features reduce the amount of time you spend idle while waiting for your project to build.

This chapter describes the following features:

- Precompiled prefix headers let you decrease the amount of time spent building compiling each source file in a target by specifying a single header file that includes all of the headers commonly used by the target's files and compiling this header a single time.
- Predictive compilation reduces the time required to compile single file changes by beginning to compile a file while you are still editing it.
- Distributed builds can dramatically reduce build time for large projects by distributing compiles to available machines on the network.

Other features that you can use to optimize the edit-build-debug cycle include:

- Fix and Continue, described in "Using Fix and Continue" (page 373), improves your debugging efficiency by allowing you to make changes to your application and see the results of your modification without stopping your debugging session.
- ZeroLink, described in "Using ZeroLink" (page 320), shortens build time by eliminating the linking step for development builds.

Using a Precompiled Prefix Header

A precompiled header is a file in the intermediate form used by the compiler to compile a source file. Using precompiled headers, you can significantly reduce the amount of time spent building your product. Often, many of the source code files in a target include a subset of common system and project headers. For example, each source file in a Cocoa application typically includes the `Cocoa.h` system header, which in turn includes a number of other headers. When you build a target, the compiler spends a great deal of time repeatedly processing the same headers.

You can significantly reduce build time by providing a prefix header that includes the set of common headers used by all or most of the source code files in your target and having Xcode precompile that prefix header.

If you have indicated that Xcode should do so, Xcode precompiles the prefix header when you build the target. Xcode then includes that precompiled header file for each of the target's source files. The contents of the prefix header are compiled only once, resulting in faster compilation of each source file. Furthermore, subsequent builds of the target can use that same precompiled header, provided that nothing in the prefix header or any of the files on which it depends has changed. Each target can have only one prefix header.

When Xcode compiles your prefix header, it generates a variant for each C language dialect used by your source files; it stores these in a folder in your project's build directory. It also generates—as needed—a variant for each combination of source header and compiler flags. For example, you may have per-file compiler flags set for some of the files in your target. Xcode will create a variant of the precompiled header by precompiling the prefix header with the set of compiler flags derived from the target and the individual source file. As Xcode invokes the compiler to process each source file in your target, the compiler searches this directory for a precompiled header variant matching the language and compiler flags for the current compile. The first precompiled header variant that is valid for the compilation is used.

Xcode automatically regenerates the precompiled header whenever the prefix header, or any of the files it depends on are changed, so you don't need to manually maintain the precompiled header.

Creating the Prefix Header

To take advantage of precompiled headers in Xcode, you must first create a prefix header. Create a header file containing any common `#include` and `#define` statements used by the files in your target.

Note: You can use a prefix header to include a common set of header files for each source file in your target without precompiling the prefix header.

Do not include anything that changes frequently in the prefix header. Xcode recompiles your precompiled header file when the prefix header, or any of the headers it includes, change. Each time the precompiled header changes, all of the files in the target must be recompiled. This can be an expensive operation for large projects.

Because the compiler includes the prefix header file before compiling each source file in the target, the contents of the prefix header must be compatible with each of the C language dialects used in the target. For example, if your target uses Cocoa and contains both Objective-C and C source files, the prefix header needs to include the appropriate guard macros to make it compatible with both language dialects, similar to the example shown here:

```
#ifdef __OBJC__
#import <Cocoa/Cocoa.h>
#endif
#define MY_CUSTOM_MACRO 1
#include "MyCommonHeaderContainingPlainC.h"
```

Configuring Your Target To Use the Precompiled Header

Once you have created the prefix header, you need to set up your target to precompile that header. To do this, you must provide values for the two build settings described here. You can edit these build settings in the Build pane of the target inspector or Info window. The settings you need to change are:

- Prefix Header (GCC_PREFIX_HEADER). Change the value of this build setting to the project-relative path of the prefix header file. If you have a precompiled header file from an existing project, set the prefix header path to the path to that file.
- Precompile Prefix Header (GCC_PRECOMPILE_PREFIX_HEADER). Make sure that this option is turned on. A checkmark is present in the Value column if this option is enabled.

You must provide values for these settings in each target that uses a precompiled prefix header, even if those targets use the same prefix header.

By default, Xcode precompiles a version of the header for each C-like language used by the target (C, C++, Objective-C, or Objective-C++). The C Dialects to Precompile (GCC_PFE_FILE_C_DIALECTS) build setting lets you explicitly specify the C dialects for which Xcode should produce versions of the precompiled header.

Sharing Precompiled Header Binaries

It is possible to share a precompiled header binary across multiple targets, provided that those targets use the same prefix header and compiler options. In order to share a precompiled header binary, each individual target must have the same prefix header specified. To use the same prefix header for multiple targets, set the value of the Prefix Header build setting for each target to the path to the header.

The PRECOMP_DESTINATION_DIR build setting specifies the location of the directory to which Xcode writes the precompiled header binary. For each target, set this location to a common directory. When Xcode invokes GCC to compile each source file in a target, GCC searches this common directory for the appropriate header binary. For any targets that use the same prefix header and compiler options, GCC uses the same precompiled header binary when it builds those targets.

This build setting is not exposed in the Xcode user interface. To change the value of this build setting, add a new entry in the build settings table that appears in the Build pane of the target inspector. Enter PRECOMP_DESTINATION_DIR as the name of the build setting and enter the path to the directory you want to use for precompiled binaries in the Value column for the build setting.

If you specify the same prefix header for multiple targets, but do not specify a common location for the precompiled binary, Xcode precompiles the prefix header once for each target as it is built.

Controlling the Cache Size Used for Precompiled Headers

Xcode caches the precompiled header files that it generates. To control the size of the cache devoted to storing those files, use the `BuildSystemCacheSizeInMegabytes` user default. In the Terminal application, type:

```
defaults write com.apple.xcode BuildSystemCacheSizeInMegabytes defaultCacheSize
```

Specifying 0 for the cache size gives you an unlimited cache. 200 MB is the default cache size set by Xcode. If the cache increases beyond the default size, Xcode removes as many precompiled headers as is necessary to reduce the cache to its default size when Xcode is next launched. Xcode removes the oldest files first.

Restrictions

To take advantage of Xcode's automatic support for precompiled headers you must:

- Use GCC 3.3 or later. Xcode uses GCC 3.3's PCH mechanism to create precompiled headers. PCH is not available with previous versions of GCC.
- Use a native target. Xcode's automatic support for precompiled prefix headers using PCH is only available for targets that use Xcode's own native build system. Xcode automatically handles many of the restrictions upon using precompiled headers with GCC. If you are using an external target to work with another

build system—such as `make`—you can still use precompiled headers, but you must create and maintain them yourself. For more information on using precompiled headers with GCC, see *GNU C/C++/Objective-C 3.3 Compiler*. Jam-based targets can also use precompiled prefix headers, but they are limited to the PFE (persistent front end) mechanism introduced with GCC 3.1. PFE is no longer recommended.

- Use one and only one prefix header per target.
- Set the Prefix Header and Precompile Prefix Header build settings for every target that uses precompiled headers.

Distributing Builds Among Multiple Computers

Building a product involves many small operations. Many of these operations—such as compiling source files—can be performed in parallel, decreasing the total amount of time it takes to build your product. If you have a dual-processor computer, Xcode automatically uses both processors. However, the greater the number of processors available to you, the greater the number of build tasks you can run in parallel. Distributed builds give you the ability to distribute build tasks among multiple computers on a network. When you use distributed builds, Xcode distributes as many build operations as possible among the computers available for that purpose.

How Distributed Builds Work

Xcode uses `distcc` to manage distributed builds. The `distcc` client manages the setup and distribution of build tasks. The server process (`distccd`) manages communication with the remote computers hosting the build tasks. The server process runs on both the local, or client, computer and on the remote computer.

When you initiate a build, Xcode invokes the `distcc` client on your local computer. For each source module that needs to be compiled, the client process connects to the server process running on the local machine and gives it the information required to distribute that task. Namely, the client process tells `distccd` the operation to perform and gives it any necessary arguments, a list of files to copy to the remote host (input files) and a list of files to copy from the remote host (output files). The server process broadcasts a work request for that operation. When an available computer responds, `distccd` sends the inputs to that computer. When the compile is complete and the remote computer returns the results, `distccd` places the output files in the appropriate locations in the file system and returns the results to the client process.

On a remote computer accepting build operations, the server process listens for requests for assistance. When it receives a request, it creates a connection with the client computer, obtains the inputs required for the compilation, and invokes `gcc`. When the compile is complete, `distccd` sends the results—the generated `.o` files, `stderr` and `stdout`—back to the client computer.

If the attempt to distribute a build task fails—for example, if communication with the remote host is lost, or the remote host cannot execute the compile—the compilation is performed on your local computer.

Xcode only distributes compilation of individual source modules. Your local computer still performs all of the build setup, linking, and product packaging. Preprocessing is also done on your local computer; this is done to avoid the problem of ensuring that all machines participating in a build have exactly the same version of all headers used in the build. For each source file, the `distcc` client invokes GCC to generate a `.i` file containing the preprocessed source; this is the input file sent to the remote host.

Computers that are shared respond to requests for assistance as they are able. Xcode determines how best to use available machines for a build, based on processing capacity. A computer's processing capacity is determined through a combination of availability and processing power.

You can use precompiled headers with distributed builds. If a precompiled header is present, the path to that header is listed in the preprocessed file generated from each source file. Before it sends a file to a remote computer, `distcc` checks for this path and, if found, sends the precompiled header to the remote computer.

Requirements for Using Distributed Builds

Use of distributed builds is subject to the following constraints:

- You must use GCC 3.3 or later.
- Machines that tasks are distributed to must be running the same version of the compiler, operating system, and Xcode.
- Distributed builds work with native targets. Jam-based targets or targets using another external build system are not compatible with distributed builds.
- Distributed builds support C, C++, Objective-C and Objective-C++.
- Distributed builds are enabled on a per-user basis, for all projects and targets built by that user.

Discovering Available Computers

To enable distributed builds on your computer, choose Xcode > Preferences and click Distributed Builds. To use other computers on the network for your builds, select "Distribute builds to."

Xcode uses Bonjour to automatically discover computers that are set up to broadcast their availability. The Distributed Builds preference pane lists the computers currently available for sharing build tasks. Services discovered through Bonjour display the Bonjour name of the computer on which `distccd` is running.

By default, Xcode will use any computers that are available when you begin a build. You can restrict which computers are used by selecting "trusted computers only" from the "Distribute builds to" menu. When you choose this option, Xcode distributes builds only to computers that you explicitly designate as "trusted." A computer is trusted if the checkbox in the Trusted column next to its entry is selected.

Each entry in the Distributed Builds preference pane represents a single computer. However, shared computers with dual processors run two instances of `distccd`, each of which can accept build tasks. The Max Connections column displays the number of processors of, and therefore the number of possible connections with, each available computer.

Computers to which tasks are distributed to must be running the same version of the compiler, operating system, and Xcode. The Status column shows whether a shared computer is compatible or not. If a computer goes to sleep, its services disappear from the preference pane, and that computer is no longer available for performing compilations.

Sharing a Computer

To allow other computers to use your computer for their builds, select the "Share my computer for building with" option in the Distributed Builds pane of the Xcode Preferences window and choose a priority for sharing your computer from the menu.

Don't share your computer if you are using it to host a distributed build as well. Because precompilation and the distribution of build tasks are done on your local machine, sharing it for others to distribute build tasks to can significantly slow down your own build.

Distributed Builds and Firewalls

To distribute builds across a firewall, the firewall must allow traffic on ports `3632` and `7264`. Some firewalls allow Bonjour traffic, making shared computers behind the firewall visible in the Distributed Builds preference pane. However, the distributed build will not work. When Xcode attempts to distribute build tasks and receives no response from the computers behind the firewall, it times out and builds the project on the local machine.

To allow traffic on ports `3632` and `7264` on computers with a firewall enabled, use the Firewall pane of Sharing Preferences.

Getting the Most Out of Distributed Builds

Using Xcode to distribute builds across multiple computers can greatly decrease the time it takes to build your product. To get the most benefit from using distributed builds, you should consider the following:

- Networking speed. The distributed build system sends files to be built on other computers; those machines, in turn, send back the resulting object files. To see a significant reduction in build time, your network must be fast enough that the cost of transferring files between machines is minimal. This occurs around 100 Mbit/s (megabits per second). For this reason, using distributed builds over a wireless network does not show any build speed benefits.
- Number of available computers. As stated at the beginning of this article, the more processors you have available to you, the more build tasks can be performed in parallel and the more significant the reduction in build time that you see. If you have a limited number of computers available for sharing—especially if those computers do not have significant processing capacity, whether due to limited availability or slow processor speed—you may not see any noticeable improvements in build performance. The overhead of managing distribution of build tasks may overcome the benefit you get from building on additional machines. You may find that you get better build performance from utilizing other optimizations, such as precompiled headers or parallel builds, on your local computer.
- Differing processing capacity between the computers managing the build and those hosting the build tasks. If you have a number of different computers available to you, of varying speed, you will get better performance if you use the faster computers for building and the slower computer to manage the build and farm out build tasks. For example, if you have a G4 and a G5 available, on a fast network, host the project on the G4 and make the G5 available for shared builds.

Predictive Compilation

Predictive compilation is a feature introduced to reduce the time required to compile single file changes and speed up the edit-compile-debug cycle of software development. If you have predictive compilation enabled for your project, Xcode begins compiling the files required to build the current target even before you tell Xcode to build.

Predictive compilation uses the information that Xcode maintains about the build state of targets that use the native build system. Xcode keeps the graph of all files involved in the build and their dependencies, as well as a list of files that require updating. At any point in time, Xcode knows which of the files used in building a target's product are out of date and what actions are required to bring those files up to date. A file can be updated when all of the other files on which it depends are up to date. As files become available for processing, Xcode begins to update them in the background, even as you edit your project.

Xcode will even begin compiling a source code file as you are editing it. Xcode begins reading in and parsing headers, making progress compiling the file even before you initiate a build. When you do choose to save and build the file, much of the work has already been done.

Until you explicitly initiate a build, Xcode does not commit any of the output files to their standard location in the file system. When you indicate that you are done editing, by invoking one of the build commands, Xcode decides whether to keep or discard the output files that it has generated in the background. If none of the changes made subsequent to its generation affect the content of a file, Xcode commits the file to its intended location in the file system. Otherwise, Xcode discards its results and regenerates the output file.

You can turn on predictive compilation by selecting "Use Predictive Compilation" option in the Building pane of the Xcode Preferences window.

Predictive compilation works only with GCC 3.3 or later and native targets. All predictive compilation is done locally on your computer, regardless of whether you have distributed builds enabled. On slower machines, enabling predictive compilation may interfere with Xcode performance during editing.

To conserve battery power, Xcode turns off predictive compilation on laptop machines running under battery power.

Using Cross-Development in Xcode

Using Xcode, you can develop software that can be deployed on, and take advantage of features from, different versions of Mac OS X, including versions different from the one you are developing on.

When you install the developer tools shipped with Mac OS X v10.3, you can also install SDKs, or Software Development Kits, which are complete sets of header files and stub libraries as shipped in previous versions of Mac OS X. In order to take advantage of cross-development, you must install the SDKs for the OS versions you plan on targeting. You then specify which version (or SDK) of Mac OS X headers and libraries to build with. You can also specify the earliest Mac OS X system version on which the software will run.

In some cases, Apple may seed an SDK for an upcoming version of the operating system, allowing you to prepare your application to work with future versions of the Mac OS before they have been released to the general public.

Important: Cross-development in Xcode requires native targets. For more information on how to upgrade existing Jam-based targets to use the native build system, see "Converting a Project Builder Target" (page 247).

To set up your Xcode project for cross-development, take the following steps:

1. Choose an SDK to develop for. Select your project in the Groups & Files list and open an Info window. In the General pane, choose the SDK—for example, Mac OS X version 10.2.7—from the "Cross-Develop Using Target SDK" pop-up menu. When you choose an SDK, Xcode builds targets in your project against the set of headers corresponding to the specified version of Mac OS X, and links against the stub libraries in that SDK. This allows you to build products on your development machine that can be run on the system targeted by the SDK. Your software can use features available in system versions up to and including the one you select. The default value is to build for the current operating system.

   You can also type a path directly in the text field or click the Choose button to select an SDK other than a Mac OS X SDK, or to choose a Mac OS X SDK that is stored in a nonstandard location.

2. Choose a deployment version of the Mac OS. If your software must run on a range of operating system versions, choose a Mac OS X deployment operating system for each individual target that requires one. The deployment operating system identifies the earliest system version on which the software can run. By default, this is set to the version of Mac OS X corresponding to the SDK version. If no SDK is specified and the Mac OS X Deployment Target build setting is not set, the compiler targets Mac OS X version 10.1 by default.

   To set the deployment version for a target, select the target in the Groups & Files list and open an Info window. Click Build to open the Build pane.

   Find the Mac OS X Deployment Target setting and choose a deployment operating system from the pop-up menu in the "Value" column. If this build setting is not visible, select Deployment from the Collections menu.

3. For each target, supply a prefix file that takes into account the selected SDK. To use an umbrella framework header from an SDK as your prefix file, add the appropriate `#include <Framework/Framework.h>` directive to your target's prefix file, instead of setting a Prefix Header path to the umbrella framework header directly.

There is a lot more to successfully developing software for multiple versions of the Mac OS. For more information see *Cross Development* in Tools Documentation.

Debugging

Once a product has been designed and an initial version built, you spend time testing it, to find and correct bugs. Xcode contains a full-featured source-level debugger that lets you step through your code line by line, set breakpoints, and view variables, stack frames, and threads. The following chapters introduce executable environments, and show you how to use them to run your program under different conditions from within Xcode. They also describe Xcode's graphical debugger, and how to use it to examine program data and control execution of your code; as well as how to use Fix and Continue to make changes to your source code and continue debugging.

PART VI

Debugging

Executable Environments

An **executable environment** defines how a product is run when you run it from Xcode. The executable environment tells Xcode which program to launch when you run or debug, as well as how to launch it. Xcode automatically creates an executable environment for each target that produces a product that can run on its own. However, you can create your own executable environments for testing products such as plug-ins or frameworks. You can also set up multiple custom executable environments for testing your program under varying sets of circumstances, or use a custom executable environment to debug a program you do not have the source to. This chapter describes how to view the executables in your project and how to configure an executable environment.

Executable Environments in Xcode

The executable environment defines:

- What executable file is launched.
- Command-line arguments to pass to the program upon launch
- Environment variables to set before launching the program.
- Debugging options that tell Xcode which debugger to use and how to run the program under the debugger.

Generally, you do not have to worry about executable environments. If you are creating a target that produces a product that can be run by itself—such as an application—Xcode automatically knows to use the application when you run or debug the target.

However, if you have a product that can't be run by itself—such as a plug-in for a third party application—you need to create your own custom executable environment. This custom executable environment specifies the program to launch when you run or debug, such as the third-party application that uses your plug-in.

Even if your target creates a product that can run on its own, you may also wish to customize the executable environment associated with it, in order to pass different arguments to the executable or test it with different environment variables.

Executable environments defined for a project are organized in the Executables group in the Groups & Files list of the project window. To see these executables, select that group in the Groups & Files list or click the disclosure triangle next to the group. Executable environments that you define are stored in your project's user file—that is, in the `.pbxuser` file in the project bundle. As a result, each developer working on a project needs to define their own executable environments. When you run or debug in Xcode, Xcode launches the program specified by the active executable, as described in the next section.

Setting the Active Executable

The **active executable** is the executable environment that Xcode uses when you click Run or Debug (or choose one of the corresponding menu items). Xcode tries to keep the active target and the active executable in sync; if you set the active target to a target that builds an executable, Xcode makes that executable active. Otherwise, the active executable is unchanged. If you use a custom executable environment to test your product, you have to make sure that the active executable is correct for the target you want to build and run or debug.

The active executable is indicated by the blue (selected) button in the detail view. To change the active executable yourself, you can:

- Select the executable in the detail view. Click the radio button next to the executable to select it.
- Choose the executable from the Active Executable pop-up menu. By default, this menu appears in the toolbar of the Build Results window, but not in the project window toolbar. To add this menu, choose View > Customize Toolbar and drag the menu to the toolbar.

Creating a Custom Executable Environment

Many targets create a product that can be run by itself, such as an application or command-line tool. When you create such a target, Xcode adds an entry to your project's executable list that points to the target's product, and it knows to use that executable environment when you run or debug the target.

Sometimes, though, you have a product that can't run by itself, such as a plug-in or a framework. Even if your product can run by itself, you may want to run the product under different conditions to test it. For example, you may want to test a command-line tool by passing it different flags. Or you may have an application that performs differently depending on the value of an environment variable.

In these cases, you need to create a custom executable environment. You may have several executable environments for exercising the product of a single target. For example, you could have several applications that test different aspects of a framework. Or you could have several lists of command-line arguments and environment variables that test different aspects of a command-line tool.

To create a custom executable, choose Project > New Custom Executable. Xcode displays an assistant in which you can specify:

- Executable Name is the name used to identify the executable environment in Xcode.
- Executable Path is the path to the executable to launch.
- The Add To Project menu lets you choose which of the currently open projects to add the custom executable to.

When you click Finish, Xcode adds the new executable environment to the project. You can change the program that Xcode launches when using this executable environment—along with other executable settings—in the inspector, as described in "Editing Executable Settings" (page 339).

Editing Executable Settings

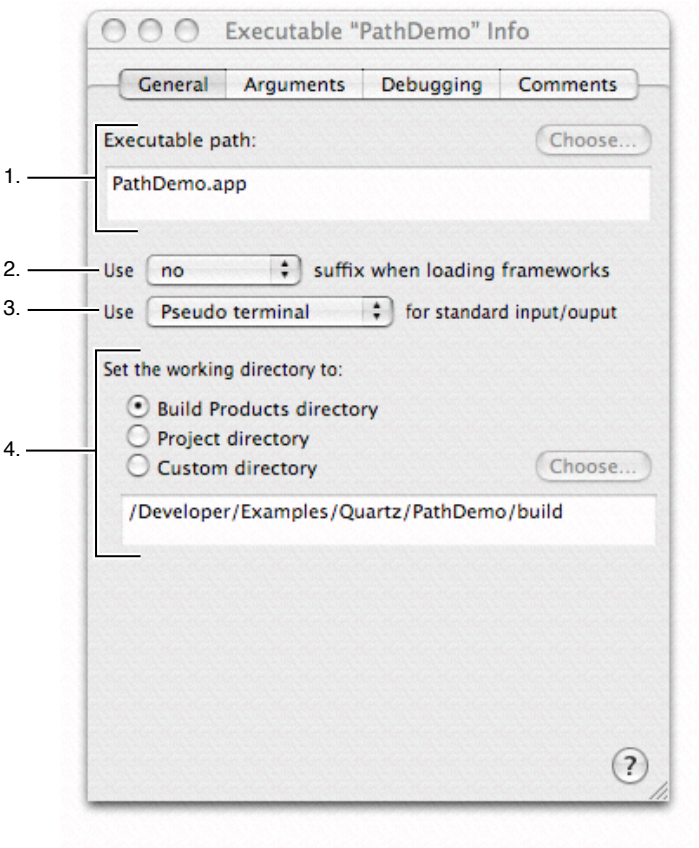
Executable environments contain a number of settings that give you control over how your product is run when you launch it from Xcode. To configure an executable, select it in the Groups & Files list or in the detail view and open an inspector or Info window. The executable inspector contains the following panes:

- The General pane lets you view and edit basic information about the executable environment, such as the executable to use, the working directory, and so forth.
- The Arguments pane lets you specify arguments to pass to the executable on launch, as well as any environment variables for Xcode to set before launching the executable.
- The Debugging pane lets you specify which debugger to use when debugging the executable, as well as a number of other debugging options. The options in this pane are described in "Configuring Your Executable for Debugging" (page 346).
- The Comments pane lets you add notes or other arbitrary text to associate with the executable.

General Settings

The General pane, shown here, lets you edit basic information about an executable environment.

Figure 31-1 The General pane of the Info window for an executable



The General pane contains the following:

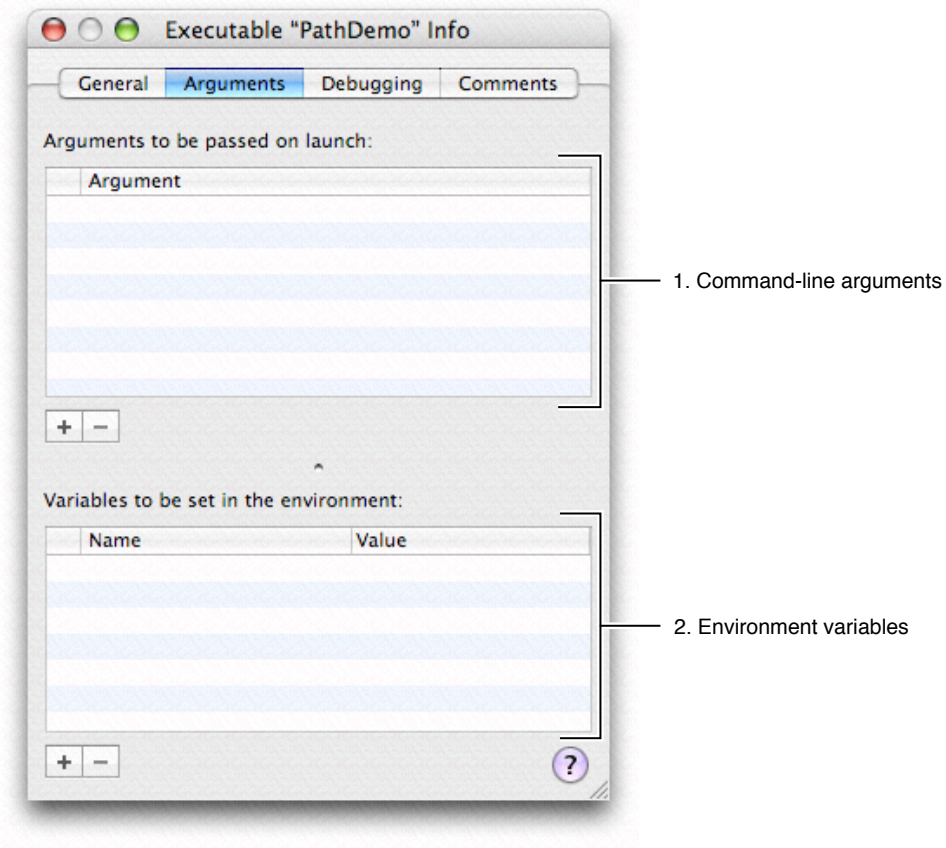
1. The "Executable path" field shows the name and location of the executable that Xcode launches when you run or debug. When you create a custom executable environment, you specify the executable to run when that executable environment is active. However, you can change the executable associated with an executable environment at any time. To specify a different executable, either type the path to the new executable directly in the text field or click the Choose button and navigate to the executable in the dialog.
2. The first pop-up menu from the top lets you control which framework variant is used when loading frameworks used by the executable. Xcode tells the dynamic linker to look for frameworks with the suffix specified in this pop-up menu. This lets you test with debug versions of many system frameworks.
3. The second pop-up menu from the top specifies the device used for standard input and output when running the executable.
4. The options under "Set the working directory to" let you specify the working directory used when running the executable. By default, Xcode uses the Build Products directory, which is set for the current project, as described in "Build Locations" (page 301).

Setting Command-Line Arguments and Environment Variables

If you have a command-line tool that takes certain arguments, you can assign those arguments to the executable environment you run the tool with and Xcode will pass those arguments to your tool when you run or debug. To test your command-line tool under different conditions, you can create multiple executable environments, each with different arguments. Changing your test environment becomes as simple as changing the active executable.

The Arguments pane of the inspector and Info window for an executable lets you specify arguments to pass to the executable, as well as environment variables that Xcode sets before launching the executable. Figure 31-2 shows the Arguments pane of the executable environment Info window.

Figure 31-2 Arguments and environment variables in the Info window for an executable



The Arguments pane of the inspector window includes:

1. A table titled "Arguments to be passed on launch." This table contains a list of command-line arguments that Xcode passes to the executable on launch. The arguments table contains two columns: the Argument column contains the argument and the Active column contains a checkbox that enables or disables the use of that argument.

    - To add a new argument, click the plus-sign button. Xcode adds an empty entry to the table.
    - To edit an argument line, double-click in the Argument column and type the argument. To disable or enable the argument, use the Active checkbox; when this checkbox is selected, Xcode passes the given argument to the executable when it is launched. To reorder the arguments list, drag the argument line to its new location in the list.
    - To remove an argument, single-click to select that argument in the table and click the minus-sign button.

2. The table titled "Variables to be set in environment" specifies environment variables that Xcode sets before it launches the executable. These environment variables are available to your program when it's running. They can be accessed with such BSD system calls as `getenv`.

    The environment variables table contains three columns: the Name column contains the variable name, the Value column contains the value of the environment variable, and the active column contains a checkbox indicating whether or not the given environment variable is used.

- To add a new variable, click the plus-sign button. Xcode adds an empty entry to the table.
- To edit a variable's name, double-click in the Name column and type the name of the variable. To edit a variable's value, double-click in the Value column and type the value of the variable. To disable or enable the variable, use the Active checkbox; when this checkbox is selected, Xcode sets the environment variable before launching the executable.
- To remove a variable, select it and click the minus-sign button.

Running a Development Product

Once you have built a target's product, you can test that product by running it from within Xcode. To build a development version of the selected target's product and run it if the build succeeds, click the Build and Run button or choose Build > Build and Run. To run the active executable, click Run or choose Debug > Run Executable.

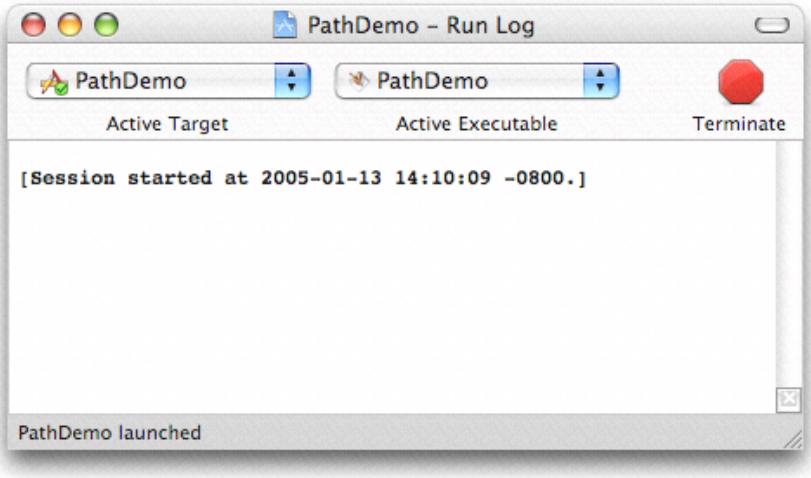
If the Build and Run button is not available, try the following:

- Build the selected target.
- Make sure the active executable is set correctly. See "Setting the Active Executable" (page 338).

The Run Log

Many programs print messages to `stdout`, as well as logging debugging messages to the console or `stderr`. When you run your program in Xcode—with the Run or Build and Run commands—you can see this output in the Run Log window. In addition, if you are creating a command-line program that takes input from `stdin`, you can use the Run Log window to interact with your program. To open the Run Log window, choose Debug > Run Log. Figure 31-3 shows the Run Log window.

Figure 31-3 The Run Log window



Note: When you run your program in Xcode's debugger, you must use the Standard I/O window to communicate with your program on `stdin`. You can view debugging output to the console or to `stderr` in the Console Log window.

Running in Xcode's Debugger

While running your program can expose bugs in its operation, it does not help you pinpoint the source of the problem in your source code. For that, you need to run your program in the debugger. The purpose of a debugger is to allow you to pause the execution of your program, examine its contents, and locate and fix problems in your code. Xcode's debugger provides a graphical user interface to `gdb` for debugging C, C++, Objective-C, and Objective-C++ programs; and communicates directly with the Java virtual machine to debug Java programs.

This chapter describes how to run your program in the debugger in Xcode; this includes configuring debugger options for an executable and enabling debugging facilities, as well as launching your program in the debugger and viewing debug output.

Generating Debugging Information

Before you can take advantage of Xcode's source-level debugger, the compiler must collect information for the debugger. To generate debugging symbols for a product, enable the Generate Debug Symbols (GCC_GENERATE_DEBUGGING_SYMBOLS) build setting and build the product. This setting is enabled by default in the Development build style provided by Xcode. If you use this build style as the active build style when you build your target, the compiler generates the necessary debugging information.

To view the build settings set in the Development build style:

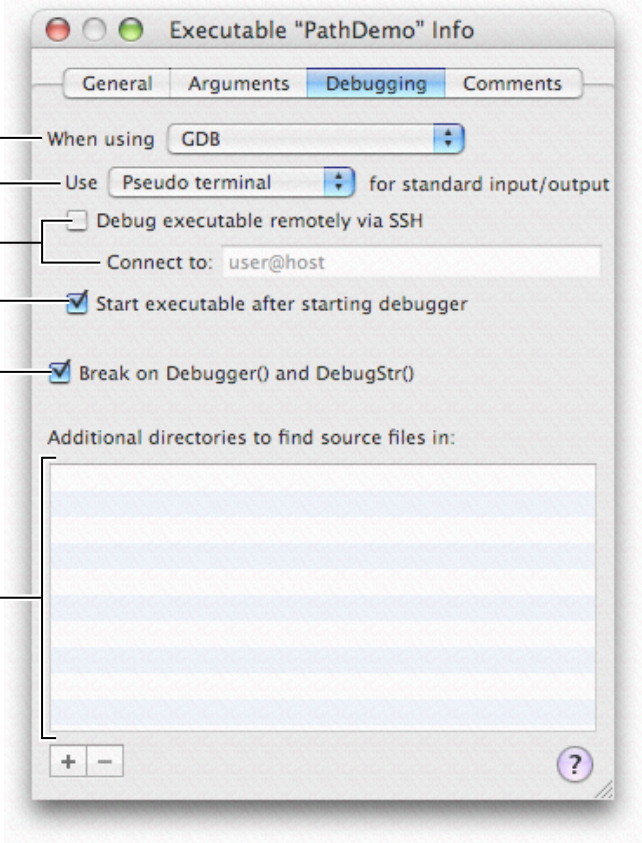
1. In the project window select the project and bring up the inspector window.
2. Click Styles to bring up the Styles pane.
3. Choose Development from the Build Style pop-up menu at the top of the pane. You should see the Generate Debug Symbols setting in the table of build settings. Make sure that this setting is turned on; if it is, a checkmark appears in the Value column for this setting. Otherwise, turn on the setting by clicking the checkbox in the Value column.

You can use the Active Build Style pop-up menu to change the active build style. This menu is available by default in the toolbar of the Build Results window; otherwise, you can add it to the toolbar of other project windows by choosing View > Customize Toolbar and dragging the Active Build Style menu into the toolbar. For more information on build styles, see "Build Styles" (page 297).

Configuring Your Executable for Debugging

When you start your program in the debugger, Xcode launches it with the command-line arguments, environment variables, and other settings specified in the executable environment. The executable environment also contains a number of debugger-specific options that control which debugger Xcode uses, as well as how Xcode launches and communicates with the program. These options are set in the Debugging pane of the executable inspector, shown in Figure 32-1.

Figure 32-1 The Debugging pane of the Info window for an executable



Here is what the Debugging pane contains:

1. When using. This pop-up menu lets you choose the debugger used when you debug the executable. The next several options below this menu pertain only to the selected debugger.

2. Use [*device*] for standard input/output. This pop-up menu lets you specify how standard input and output are handled when debugging the executable. By default, Xcode uses a pseudo terminal device to display `stdin` and `stdout` in the Standard I/O Log window, described in "Debugging a Command-Line Program" (page 350). You can also choose System Console from this menu to have Xcode redirect your program's output to the system console (`console.log`).

3. Options for remote debugging. The "Debug executable remotely via SSH" option lets you set up the executable for remote debugging using GDB in the Xcode debugger window. The "Connect to" field specifies the host machine on which the executable is run. See "Remote Debugging in Xcode" (page 379).

4. Start executable after starting debugger. This option lets you control whether Xcode starts the executable running immediately after loading it in the debugger. If this option is enabled, Xcode loads the executable in the debugger but does not start it until you press "Restart." This allows you to perform operations in the debugger—such as setting breakpoints—before the executable runs.

5. Break on Debugger() and DebugStr(). This option tells Xcode to set the USERBREAK environment variable, which suspends execution of your program on calls to the Core Services framework debugging functions `Debugger` or `DebugStr`.

6. Additional directories to find source files in. This table lets you specify additional folders in which Xcode can look for source files corresponding to the symbol information in the executable code. To add a directory, click the plus-sign button and type the path to the directory in the Source Directory field. You can also drag folders to the table from the Finder; Xcode will insert the path to the folder. To remove a directory, select it in the table and click the minus-sign button.

Starting Your Program in the Debugger

You can start a debugging session using the buttons available in the toolbar of most Xcode windows, or using the menu items in the Build and Debug menus. To build a development version of the active target's product and start the debugger if the build succeeds, choose Build > Build and Debug, or click the Build and Debug button. To load an executable that has already been built, choose Debug > Debug Executable or click the Debug button.

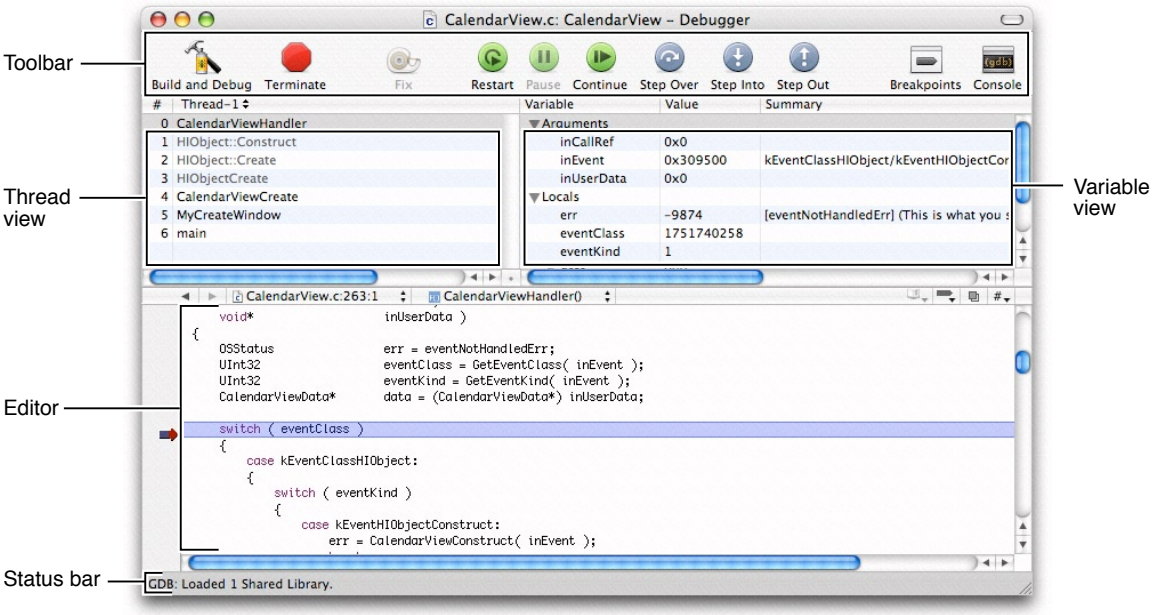
When you start a debugging session, Xcode uses the active executable for the current project to determine which executable file to load. It loads that program in the debugger, sets any environment variables specified by the executable environment, and starts the program, passing any specified command-line arguments. You can debug only one executable in a project at a time. You can, however, have multiple projects open at once, each with its own instance of the debugger. In this way, you can debug multiple executables at a time. If, for example, you have a project that builds both client and server executables, you can debug them both at once by creating an empty project and adding a custom executable that points to the client or server product. You can then run the client executable under the debugger from one project and the server executable from the other. "Creating a Custom Executable Environment" (page 338) describes how to create a custom environment.

To choose which debugger is used, edit the debugging settings for the executable, as described in "Configuring Your Executable for Debugging" (page 346).

The Debugger Window

Xcode automatically opens the Debugger window, if necessary, when you start a debug session. You can also display the debugger window at any time by choosing Debug > Debugger. Figure 32-2 shows the Debugger window.

Figure 32-2 The debugger window



Here's what the debugger window contains:

- Toolbar buttons for stepping through code and displaying related windows.
  - The Build and Debug and the Debug buttons load the project's active executable in the debugger. When a debugging session is active, the Terminate button is available; this button stops the active debugging session.
  - The Fix button compiles a single file fix and modifies your executable to run the changed code without stopping the current debugging session. For more information on how to use this feature, see "Using Fix and Continue" (page 374).
  - The Pause, Continue, and Restart buttons control execution of the program.
  - The Step Into, Step Over, and Step Out buttons step through lines of code.
  - The Breakpoints button opens the breakpoints window, which allows you to create and view breakpoints, described further in "Breakpoints" (page 353).
  - The Console button opens the debugger console.
- The status bar displays the current status of the debugging session. For example, in the window shown above, Xcode indicates that GDB has just finished loading symbols for a single shared library.
- The Thread view on the upper-left side of the debugger window displays the call stack of the current thread. The pop-up menu above this view lets you select different threads to view when debugging a multi-threaded application.
- The Variables view on the upper-right side of the debugger window shows the variables defined in the current scope and their values. This section also shows the current state of all processor registers when the disassembly view is enabled (see "Viewing Disassembled Code and Processor Registers" (page 369) for more information).

- The editor displays the source code you are debugging. When execution of your program is paused, the debugger indicates the line at which execution is paused by displaying the PC indicator, which appears as a red arrow. The line of code is also highlighted. You can change the color used to highlight the currently executing statement by choosing Xcode > Preferences, selecting Debugging, and choosing a new color from the Instruction Pointer Highlighting color well.

You can change the layout of the debugger window by choosing Debug > Toggle Debugger Layout. In this alternate layout, the Variables view is on the left side of the debugger window, under the Thread view, and the editor is on the right side of the window.

Troubleshooting

If the Build and Debug button is not available, try the following:

- Build the selected target.
- Associate an executable with the selected target. See "Executable Environments" (page 337).

If the debugger does not display source for a file, try the following:

- Make sure you have the source. Apple's frameworks and many third-party libraries don't include source code.
- Make sure debugging is enabled for the target.
- If the file is in your project's Groups & Files list, make sure its name is not in red, which means it can't be found.
- If the file is not in the Groups & Files list and your target might need to process it, add the file to the project. See "Adding Files, Frameworks, and Folders to a Project" (page 78).
- If the file is for a library or framework that was built for you, do one of the following:
  - Place the source in the same location used by the person who built the library or framework. When someone builds a debuggable binary, the compiler stores the paths of its source files in the binary.
  - Add the file's directory to the source directories list. In the project window, select the executable and open an Info window. Click Debugging and enter the file's directory in the table titled "Additional directories to find source files in."

Lazy Symbol Loading

Lazy symbol loading reduces the initial memory footprint for debugging large applications by reading debugging symbols only when they are needed. When the "Load symbols lazily" option is enabled—as it is by default in Xcode's Debugger Preferences pane—the `gdb` debugger reads a minimal amount of symbol information to support symbolic breakpoints and calling functions in the debugger console. To set a file and line breakpoint Xcode gives hints to `gdb` about the symbols needed to set the breakpoint. When a stack trace is generated, `gdb` automatically reads the full debugging symbols as needed, providing line number and file information to Xcode. When lazy symbol loading is disabled, Xcode reads the full debugging symbols when the debugger starts up.

The Console Window

Xcode's graphical interface for GDB, the GNU debugger, and for the Java debugger lets you perform most necessary debugging tasks. You may, however, encounter situations—such as working with watchpoints in GDB—that require you to interact directly with the debugger on the command line. Using the Console Log in Xcode, you can:

- View the commands that Xcode sends to `gdb` or the Java command-line debugger.
- Send commands directly to `gdb` or the Java command-line debugger.
- View the debugger output for those commands.
- See debugging messages printed to stderr by your program or by system frameworks.

To open the Console Log, click the Console button in the toolbar of Xcode's Debugger window or choose Debug > Console Log. To enter commands, click in the console window and type at the gdb or JavaBug prompt. To get help with GDB and Java debugging commands, enter `help` at the console. To learn more about command-line debugging with GDB, see *Debugging With GDB* in Tools Compilers & Debuggers Documentation.

To make the Console Log easily readable, Xcode lets you choose the text colors and fonts used in the console window. You can use different fonts and colors for the text you type in the console, the text the debugger writes to the console, and the debug console's prompt. To change the colors used for text in the Console window, choose Xcode > Preferences, click Debugging, and use the Fonts and Colors options. See "Debugging Preferences" (page 400) for more information.

Debugging a Command-Line Program

If you are debugging a command-line program that requires input from `stdin`, you must use the Standard I/O Log to communicate with your program when it is running in the debugger. To open the Standard I/O Log window, choose Debug > Standard I/O Log. This window is only available when your program is running under the debugger.

Xcode and Mac OS X Debugging

Many of the subsystems in Mac OS X include debugging facilities that you can use to help you in your debugging tasks. You can use most of these debugging facilities along with Xcode. Many debugging facilities are enabled or disabled by setting an environment variable; you can modify the executable environment to set these environment variables from Xcode. Xcode also includes several options for enabling specific debugging options, such as libgmalloc (Guard Malloc), loading debug library variants, and stopping on Core Services debugging functions (described in "Stopping on Core Services Debugging Functions" (page 357)). For more on the many debugging facilities available in Mac OS X, see *TN2124: Mac OS X Debugging Magic*.

Using Debug Variants of System Libraries

Many Mac OS X system frameworks include debug versions, in addition to the production version. These library variants are identified by their _debug suffix. Debug variants of the system frameworks usually include debugging symbols, extra assertions, and often extra debugging facilities. You can modify the executable environment to have Xcode use the debug variants for libraries that your program loads. To use the debug variant of a library, open the inspector for the executable environment that you use to run your program. In the General pane, choose "debug" from the menu "Use [`suffix`] suffix when loading frameworks."

Using Guard Malloc in Xcode

Xcode also integrates Guard Malloc (libgmalloc) into the debugger interface. Guard Malloc helps you debug memory problems by causing your program to crash on memory access errors. Because Guard Malloc causes your program to crash, you should use Guard Malloc with the debugger. When a memory access error occurs and your program crashes, you can look at the stack trace, determine exactly where the error occurred, and quickly jump to the location of the problem.

To enable debugging with Guard Malloc from Xcode, choose Debug > Enable Guard Malloc, before starting the debugging session. You can also use Guard Malloc with gdb from the command line, by setting the DYLD_INSERT_LIBRARIES environment variable, as described in the man page for `libgmalloc`.

Guard Malloc has a number of additional options available. You can take advantage of these by setting the appropriate environment variables on the executable. In the inspector window for the executable, open the Arguments pane and add the environment variables to the environment variables table at the bottom of the window. See the man page for `libgmalloc` for additional details and information.

Controlling Execution of Your Code

The purpose of the debugger is to help you identify and resolve problems in your code by analyzing the internal operation of your program. Using the debugger, you can halt the execution of your program when you encounter a problem and inspect the program in its current state. The debugger can stop your program when certain events occur or when a particular line of code is reached. It also lets you execute individual machine instructions or lines of code, pausing after each to examine the contents of your program.

This chapter shows you the various facilities that the Xcode debugger provides for pausing and then resuming execution of your program. It describes how to use breakpoints to stop at a particular line of code or at a function or method call, as well as how to stop when your program throws an exception. It also describes how to step through code, to view changes effected in your program one line or machine instruction at a time.

Breakpoints

Breakpoints let you pause the execution of your program in the debugger whenever certain events occur in your code. You can set a breakpoint to stop your program when:

- Execution reaches a specific line of code. When you set a breakpoint for a particular line number in a source code file, the debugger stops your program just before it executes any of the code on that line.
- A particular function or method is called. You can specify the name of a function or method to set a breakpoint at the entry to that function or method; this is referred to as a symbolic breakpoint.

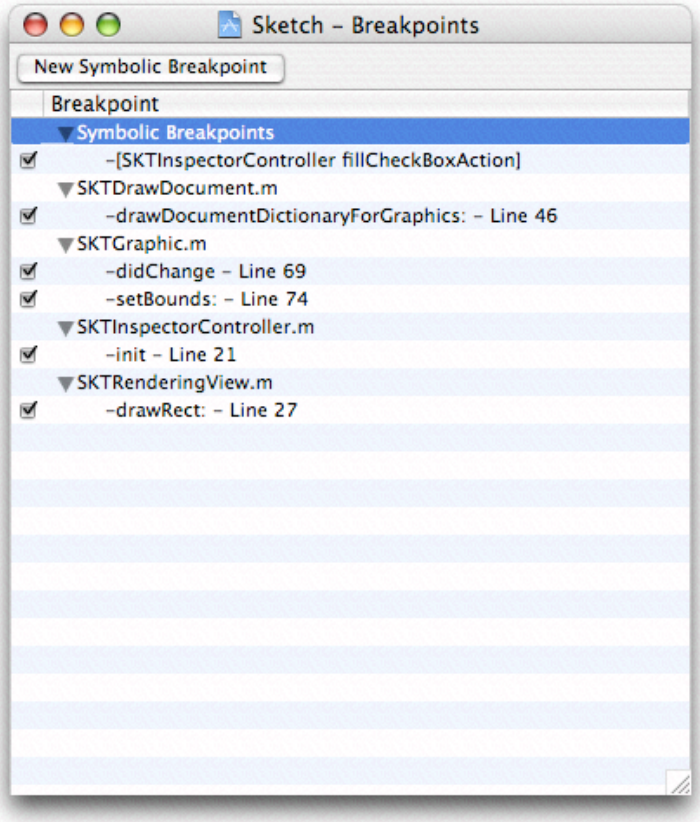
In Xcode, you can set and view breakpoints for a project from the code editor or from the Breakpoints window. Xcode stores all your breakpoints when you close your project and restores them when you open it again. Breakpoints are stored per-project.

GDB also supports special kinds of breakpoints that let you halt execution of your program when a particular event, such as the loading of a certain library, occurs; or when the value of an expression changes. To set breakpoints of this nature—called catchpoints and watchpoints—you must use the command line interface, described in "The Console Window" (page 350).

The Breakpoints Window

The Breakpoints window lets you view and modify all of the breakpoints set in the current project. This includes function and method breakpoints, as well as breakpoints associated with a particular line of code in a source file. To open the Breakpoints window, click the Breakpoints button in the toolbar of the debugger window or choose Debug > Breakpoints. Figure 33-1 shows the Breakpoints window:

Figure 33-1 The Breakpoints window



Breakpoints on a function or method are assigned to the Symbolic Breakpoints group. File line breakpoints are grouped by the file in which they appear; Xcode displays the line number at which each breakpoint is set and the surrounding context of the line. The checkbox next to each breakpoint indicates whether that breakpoint is enabled. When a breakpoint is enabled, the debugger stops when it encounters the specified line or function.

You can delete, disable, or reenable any existing breakpoint in your project in the Breakpoints window. You can also create new symbolic breakpoints. However, you can only add a new breakpoint to a specific line of code from the editor.

To view the source code for a breakpoint, double-click the breakpoint in the Breakpoints window. This opens the source file in which the breakpoint is set or the function is defined in a separate editor window.

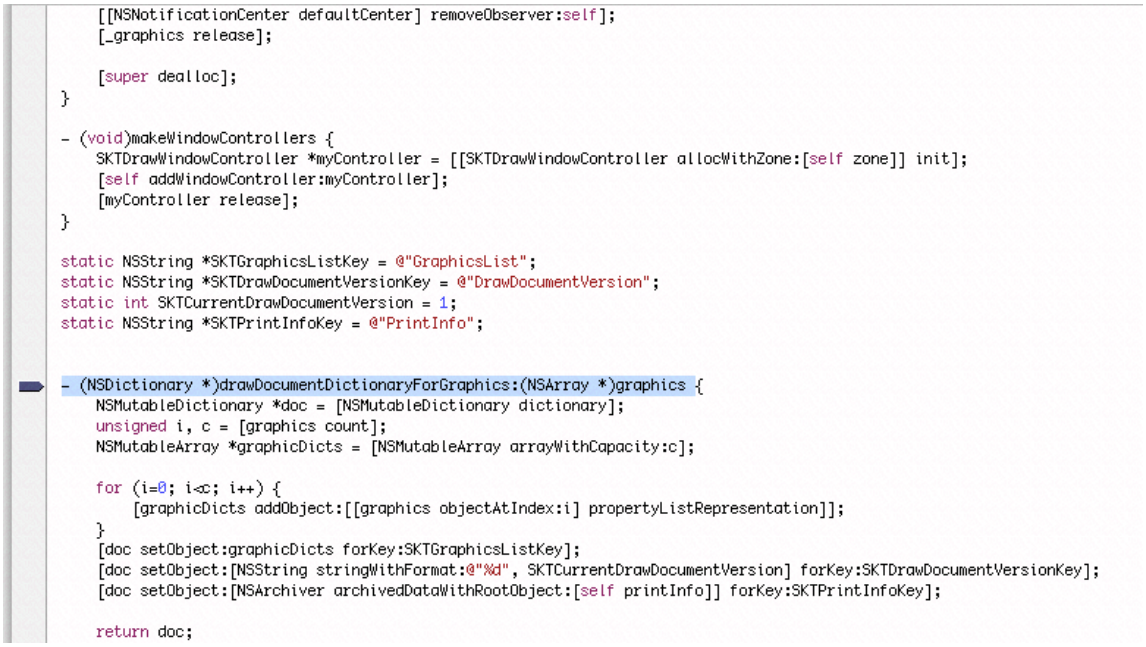
Adding Breakpoints

Xcode's debugger lets you add a breakpoint to a specific line of code or to a function or method. You can set a breakpoint on a line of code directly in the editor. To set a symbolic breakpoint, use the Breakpoints window.

Setting a Breakpoint at a Line of Code

To add a file line breakpoint, click the margin beside the line of code in the code editor or place the insertion point in the line and choose Debug > Add Breakpoint at Current Line (or type Command-\). Xcode adds a breakpoint at the current line and displays it in the gutter of the editor. Breakpoints appear as a dark arrow in the gutter, pointing to the specified line of code, similar to the following:

Figure 33-2 A breakpoint in a gutter



```
    [[NSNotificationCenter defaultCenter] removeObserver:self];
    [_graphics release];

    [super dealloc];
}

- (void)makeWindowControllers {
    SKTDrawWindowController *myController = [[SKTDrawWindowController allocWithZone:[self zone]] init];
    [self addWindowController:myController];
    [myController release];
}

static NSString *SKTGraphicsListKey = @"GraphicsList";
static NSString *SKTDrawDocumentVersionKey = @"DrawDocumentVersion";
static int SKTCurrentDrawDocumentVersion = 1;
static NSString *SKTPrintInfoKey = @"PrintInfo";


- (NSDictionary *)drawDocumentDictionaryForGraphics:(NSArray *)graphics {
    NSMutableDictionary *doc = [NSMutableDictionary dictionary];
    unsigned i, c = [graphics count];
    NSMutableArray *graphicDicts = [NSMutableArray arrayWithCapacity:c];

    for (i=0; i<c; i++) {
        [graphicDicts addObject:[[graphics objectAtIndex:i] propertyListRepresentation]];
    }
    [doc setObject:graphicDicts forKey:SKTGraphicsListKey];
    [doc setObject:[NSString stringWithFormat:@"%d", SKTCurrentDrawDocumentVersion] forKey:SKTDrawDocumentVersionKey];
    [doc setObject:[NSArchiver archivedDataWithRootObject:[self printInfo]] forKey:SKTPrintInfoKey];

    return doc;
```

You can easily move a breakpoint associated with a specific line of code to another line by dragging the breakpoint arrow to the new line within the code editor.

Once you've set a breakpoint at a particular line, you can see that breakpoint in the Breakpoints window as well, as described in the previous section. However, you cannot set a new file line breakpoint from the Breakpoints window.

Setting a Breakpoint on a Function or Method

To set a breakpoint at a function or method, use the Breakpoints window. You can open the breakpoints window by clicking the Breakpoint button or choosing Debug > Breakpoints. To add a new breakpoint, click the New Symbolic Breakpoint button. Xcode adds an item to the Symbolic Breakpoints group; type the name of the function or method at which to set the breakpoint in this field. For example, to stop whenever something in your code calls `malloc`, enter `malloc`.

When you set a breakpoint on an Objective-C method, you must include the brackets and a plus or minus sign. For example, to stop whenever a Cocoa exception is raised, enter `-[NSException raise]`.

Deleting Breakpoints

If you no longer need a breakpoint, you can delete it from the project using the Breakpoints window or, if the breakpoint is set on a specific line of code, from within the code editor. To remove any breakpoint in your project—symbolic or line number—open the Breakpoints window, select the breakpoint you want to delete, and choose Edit > Delete or press the Delete key.

To remove a breakpoint that is set on a line of code, you can also do either of the following in the editor:

- Click the breakpoint marker—the arrow symbol—in the gutter of the editor.
- Place the insertion point in the line and choose Debug > Remove Breakpoint at Current Line.

Xcode removes the breakpoint marker from the breakpoint gutter and deletes the breakpoint from the Breakpoints window.

Disabling and Reenabling Breakpoints

In the course of debugging a program, you may find that you don't currently want the debugger to use a particular breakpoint that you have set, but you don't want to delete it entirely, either, in case you want to use it again at a later time. Instead of deleting the breakpoint from your project, you can simply disable it. This renders the breakpoint inactive, but retains all of the information for that breakpoint, so you can enable it again later. A disabled breakpoint appears in the breakpoint list and is saved in the project, but Xcode doesn't stop at it. The breakpoint remains disabled until you enable it again.

You can disable or reenable any breakpoint in your project—whether a symbolic breakpoint or a line number breakpoint—from the Breakpoints window. To disable or reenable a breakpoint, open the Breakpoints window and click the checkbox beside the breakpoint you want to modify. A breakpoint is enabled when this checkbox is selected.

You can also disable or reenable file line breakpoints from the code editor by Control-clicking the breakpoint in the gutter and choosing Disable Breakpoint or Enable Breakpoint from the contextual menu. Alternatively, Command-clicking the breakpoint arrow in the gutter toggles the state of the breakpoint between enabled and disabled. A disabled breakpoint on a line of code appears as a light grey arrow in the breakpoint gutter of the code editor.

Stopping on C++ Exceptions

If you are debugging C++ code, you can specify that the Xcode debugger stop on `catch` and `throw`. You can enable this feature using the following two menu items in the Debug menu:

- Stop on C++ throw. Enable this option to halt execution when an exception is thrown.
- Stop on C++ catch. Enable this option to halt execution when an exception is caught.

Stopping on Core Services Debugging Functions

The Core Services framework includes routines, such as `Debugger` and `DebugStr`, that break into the debugger with a message. If your code contains calls to these functions, you can tell Xcode's debugger to stop when it encounters these functions.

You can enable this feature for an individual executable, as described in "Configuring Your Executable for Debugging" (page 346), or for all executables in the current project using the Debug menu. To enable stopping on calls to `Debugger` and `DebugStr` from the Debug menu, choose Debug > Stop on Debugger()/DebugStr(). If this feature is already enabled, choosing the menu item a second time disables it.

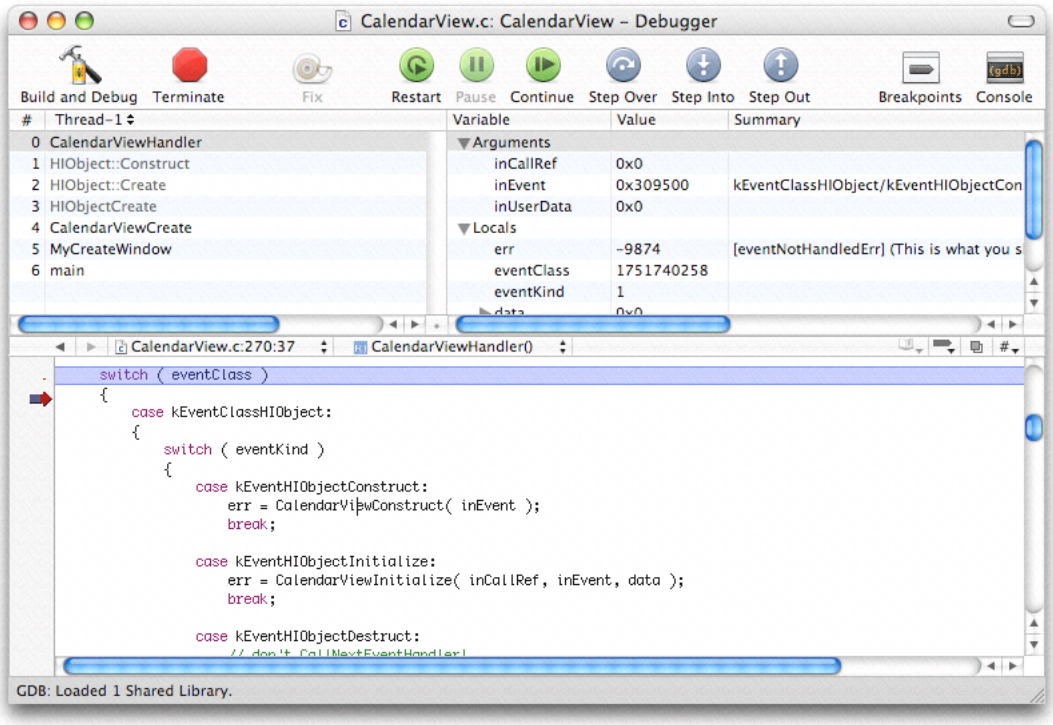
Xcode sets the USERBREAK environment variable to 1, which causes these functions to send a SIGINT signal to the current process, breaking into the debugger.

Stepping Through Code

Once execution of your program has been suspended in the debugger—for example, after hitting a breakpoint—you have a number of options for resuming execution. You can simply resume execution of your program from its current location, continuing until the program exits or the debugger encounters another breakpoint, or you can step through your program's code. When you step through code, you tell the debugger to execute just one more code line or machine instruction and pause the program once more. This lets you examine changes in your program in detail and pinpoint problems.

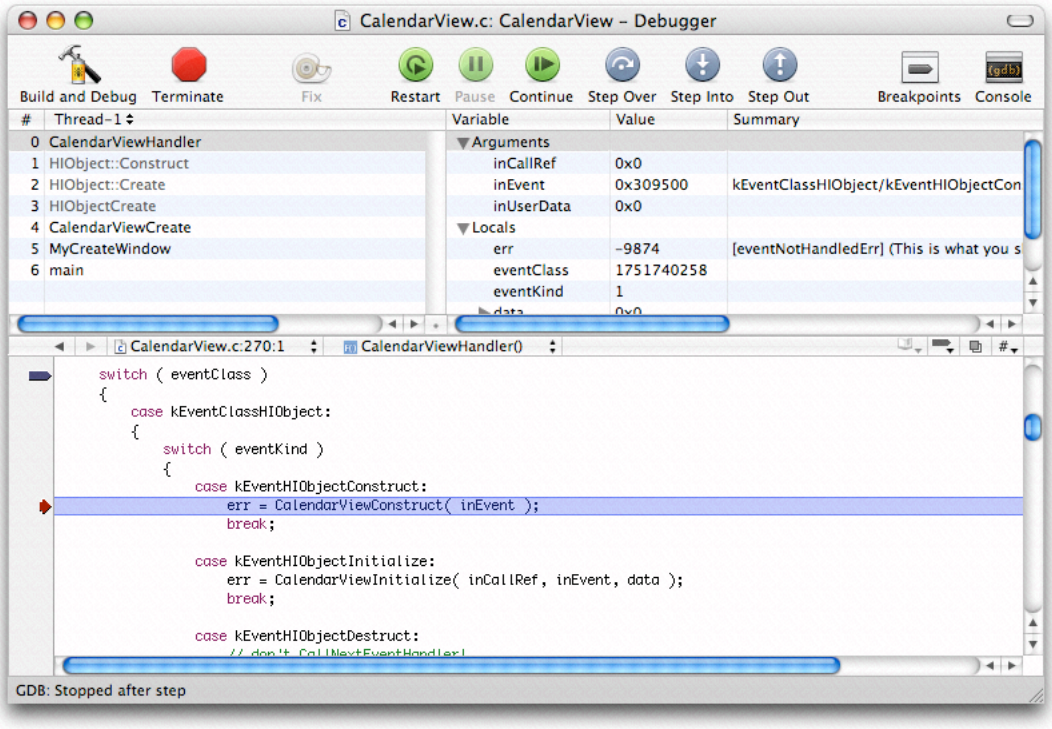
The Xcode debugger provides toolbar buttons and menu items to let you control the execution of your code. The following example uses the CalendarView sample application to illustrate how to step through code in the debugger using some of these commands. In Figure 33-3 the execution of the CalendarView program is halted at a breakpoint set in the `CalendarViewHandler` function.

Figure 33-3 Execution of a program stopped at a breakpoint



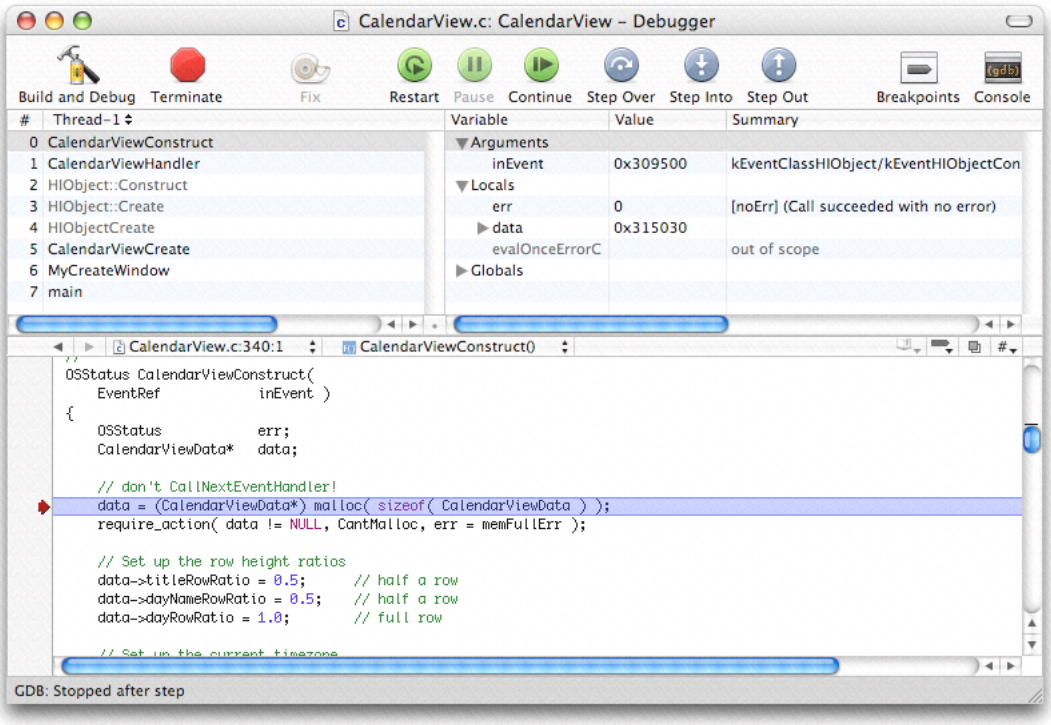
The line of code at which execution is paused—the line at which the breakpoint is set—is highlighted in the editor. The red arrow next to the code line, the PC indicator, shows the position of the program counter. You can see the chain of function calls from which the `CalendarViewHandler` function was invoked in the Thread view, located in the top left of the debugger window. The arguments and local variables of the function or method selected in the thread list are visible in the Variable view, to the right of the thread list in the debugger window. To go to the next line of code, staying within the current function, press Step Over.

Figure 33-4 Stepping over a line of code



The PC indicator has advanced to the next line of code, where it is again paused. The next line of code has a function call. To go to the next line of code, stepping into the function if possible, press Step Into.

Figure 33-5 Stepping into a function



The PC indicator is now paused at a line in the `CalendarViewConstruct` function, which was called from the line of code in the `CalendarViewHandler` function that you stepped into. The local variables and arguments for `CalendarViewConstruct` are shown in the variable display and the call chain now lists this function above the `CalendarViewHandler` function. To go to the end of the current function and back to the statement of the `CalendarViewHandler` function from which it was called, press Step Out.

You can also use the Step Into Instruction and Step Over Instruction commands to step through individual machine instructions. See "Viewing Disassembled Code and Processor Registers" (page 369) for more information.

All of the commands in this section also have equivalent menu items in Xcode's Debug menu. You can change the location at which execution will start when you continue the program or resume stepping through code by dragging the PC indicator to the appropriate line of code.

Stopping and Starting Your Program in the Debugger

In addition to the commands demonstrated in the previous section, you can further control the execution of your program in the debugger using the following commands:

- To force-quit your application and restart it, press Restart.
- To pause your program while it's running, press Pause.
- To continue it again, press Continue.
- To force the application to quit immediately, press Terminate.

Xcode provides buttons in the Debugger window toolbar, as well as menu items in the Debug menu, for each of these commands.

Examining Program Data and Information

When execution of your program is paused, Xcode displays a great deal of information on the current state of your program. Xcode's graphical debugging interface makes it easy to examine the call stack, and view variables and their values. This chapter describes how to view variables, expressions, and disassembled code. It also describes how to use the Memory Browser to view the contents of memory.

Viewing Stack Frames

For each function call that your program makes, the debugger stores information about that call in a **stack frame**. These stack frames are stored in the **call stack**. When execution of your program is paused in the debugger, Xcode displays the call stack for the currently running process in the Thread view display on the left of the debugger window, with the most recent call at the top.

Selecting any function call in the call stack displays the stack frame for that function. The stack frame includes information on the arguments to the function, variables defined in the function, and the location of the function call. Xcode displays the frame's variables in the Variable list and displays its currently executing statement in the code editor with a red arrow. If a stack frame is grayed out, no source code is available for it.

In the pop-up menu above the call stack in the debugger window, Xcode displays all the threads for your application. To view a thread, choose it from the pop-up menu. Xcode displays its call chain in the Thread list.

Viewing Variables in the Debugger Window

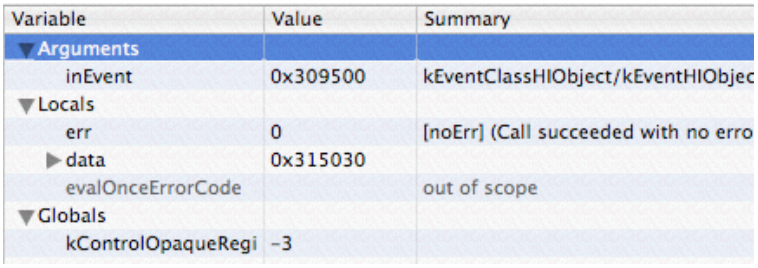
The variables view shows information—such as name, type and value—about the variables in your program. Variables are displayed for the stack frame that is currently selected in the debugger window. The variables view, shown in Figure 34-1, appears in upper-right portion of the Debugger window by default. The variables view can have up to four columns:

1. The Variable column shows the variable's name.
2. The Type column displays the type of the variable. This column is optional. To display it, choose Debug > Variables View > Show Types.
3. The Value column shows the variable's contents. If a variable's value is in red, it changed when the application was last active. You can edit the value of any variable; the changed value is used when you resume execution of your program.
4. The Summary column gives more information on a variable's contents. It can be a description of the variable or an English language summary of the variable's value. For example, if a variable represents a point, its summary could read "(x = *x value*, y = *y value*)." You can edit the summary of a variable by

double-clicking the Summary column or choosing Debug > Variables View > Edit Summary Format. For a description of how you can format variable summaries, see "Using Custom Data Formatters to View Variables" (page 364).

You can choose which columns the debugger shows in the variables view; Xcode remembers these columns across debugging sessions.

Figure 34-1 The variables view



Variables in the variables view are grouped by category; to view variables in any of these groups, click the disclosure triangle next to that group. These groups are:

- The Arguments group contains the arguments to the function that is currently selected in the call stack.
- The Locals group contains the local variables declared in the function that is currently selected in the call stack.
- The Globals group shows global variables and their values. By default, there are no global variables in this section; you must select those variables you want to track in the Globals Browser, described in "Using the Globals Browser" (page 367).
- The File Statics group shows file statics, if any. This group is not shown if none are present.

To view the contents of a structured variable, click the disclosure triangle beside the variable's name. You can also use a data formatter to display a variable's contents in the Summary column, as described in "Using Custom Data Formatters to View Variables" (page 364), or you can view a variable in its own window. Viewing a variable in its own window is particularly useful for viewing the contents of complex structured variables. To open a variable in its own window, double-click the variable's name or select it and choose Debug > Variables View > View Variable in Window.

Using Custom Data Formatters to View Variables

Xcode allows you to customize how variables are displayed in the debugger by specifying your own format strings for the Value or Summary columns. In this way, you can display program data in a readable format. Xcode includes a number of built-in data formatters for data types defined by various Mac OS X system frameworks. You can edit these format strings or create your own data formatters.

Working With Data Formatters

The menu item Debug > Variables View > Enable Data Formatters lets you turn data formatters on and off. If a checkmark appears next to the menu item, data formatters are enabled; select the menu item again to turn data formatters off. Data formatters are enabled by default.

If you are debugging heavily threaded code, where more than one thread is executing the same code, data formatters may cause threads to run at the wrong time and miss breakpoints. To avoid this problem, disable data formatters.

Writing Custom Data Formatters

You can provide your own data format strings to display data types defined by your program. To edit the format string associated with a variable value or variable summary, double-click in the appropriate column of the variables view. You can also choose Debug > Variables View > Edit Summary Format.

Data format strings can contain:

- Literal text. You can add explanatory text or labels to identify the data presented by the format string.
- References to values within a structured data type. You can access any member of a structured variable from the format string for that variable. The syntax for doing so is *%pathToValue%*, where *pathToValue* is the . -delimited path to the value you want to access in the current data structure.
- Expressions, including function or method calls. The syntax for an expression is *{expression}*. To reference the variable itself in the expression, use $VAR. For example, to display the name of a notification—of type `NSNotification`—you can use *{(NSString *)[$VAR name]}:s*.

When it displays the data format string, Xcode replaces the member reference or expression with the contents of the Variable view's Value column for the value obtained by evaluating the reference or expression. You can, however, specify that Xcode use the contents of any column in the Variables view—Variable name, Type, Value, or Summary. To do so, add *:referencedColumn* after the expression or member reference, where *referencedColumn* is a letter indicating which Variables view column to access. So, the syntax for accessing a value in a structured data type becomes *%pathToValue%:referencedColumn*. Table 34-1 shows the possible values for referencing variable display columns.

Table 34-1

Reference	Variables view column
n	Variable (shows the variable name)
v	Value
t	Type
s	Summary

The following example uses the `CGRect` data type to illustrate how you can build format strings using member references and expressions. (Note that Apple provides format strings for the `CGRect` data type, so Xcode's debugger already knows how to display the contents of variables of that type). The `CGRect` data type is defined as follows:

```
struct CGRect { CGPoint origin; CGSize size; }; typedef struct CGRect CGRect;
```

Assuming that the goal is to create a format string that displays the origin and size of variables of type `CGRect`, there are many ways you can write such a format string. For example, you can reference to members of the `origin` and `size` fields directly. Of course, each of these two fields also contain data structures, so simply referencing the values of those fields isn't very interesting; the values you actually want are in those data structures. One way you can access those values is to simply include the full path to the desired field

from the `CGRect` type. For example, to access the height and width of the rectangle, in the `height` and `width` fields of the `CGSize` structure in the `size` field, you could use the references *%size.height%* and *%size.width%*. An example format string using these references might be similar to the following:

height = %size.height%, width = %size.width%

You could write a similar reference to access the x and y-coordinates of the origin. Or, if you already have data formatter for values of type `CGPoint` that displays the x and y coordinates of the point in the Summary column of the Variables view—such as "*(%x%, %y%)*"— you can leverage that format string to display the contents of the `origin` field in the data formatter for the `CGRect` type. You can do so by referencing the Summary column for `CGPoint`, as in the following format string:

origin: %origin%:s

When Xcode evaluates this format string, it accesses the `origin` field and retrieves the contents of the Summary column for the `CGPoint` data type, substituting it for the reference to the `origin` field. The end result is equivalent to writing the format string *origin: (%origin.x%, %origin.y%)*.

You can combine this with the format string for the size field and create a data format string for the `CGRect` type similar to the following:

origin: %origin%:s, height = %size.height%, width = %size.width%

Given a rectangle with the origin (1,2), a width of 3, and a height of 4, this results in the following display: **origin: (1, 2), width=3, height=4**. You can also write a data formatter to display the exact same information using an expression, such as the following:

origin: {$VAR.origin}:s, height = {$VAR.size.height}, width = {$VAR.size.width}

When Xcode evaluates this expression for a variable, it replaces $VAR with a reference to the variable itself. Of course, using an expression to perform a simple value reference is not necessary. Another example of an expression in a format string is *{(NSString *)[$VAR name]}:s*, to display the name of a notification, of type `NSNotification`.

When you specify a custom format string for a variable of a given type, that format string is also used for all other variables of the same type. Note, however, that you cannot specify a custom format for string types, such as `NSString`, `char*`, and so on. Custom format strings that you enter in Xcode's debugger window are stored at `~/Library/Application Support/Apple/Developer Tools/CustomDataViews/CustomDataViews.plist`.

In addition to supplying custom format strings to display variables in the debugger, you can also write your own code that constructs descriptions for variables displayed in the debugger. These functions can be packaged as a bundle that is loaded into the process being debugged and can be invoked from format strings.

Using a Different Display Format to View a Variable

You can view the value of a variable in a variety of formats, including hexadecimal, octal, and unsigned decimal. To display a variable's value in a different numeric format, select the variable in the variables view and choose the numeric format from the Debug > Variables View menu. Choose Debug > Variables View > Natural to use the default format for a variable's value, based on the type of that variable.

You can also cast a variable to a type that's not included in the menu. For example, a variable may be declared as `void *`, but you know it contains a `char *` value. To cast a variable to a type, select the variable, choose Debug > Variables View > View Value As and enter the type.

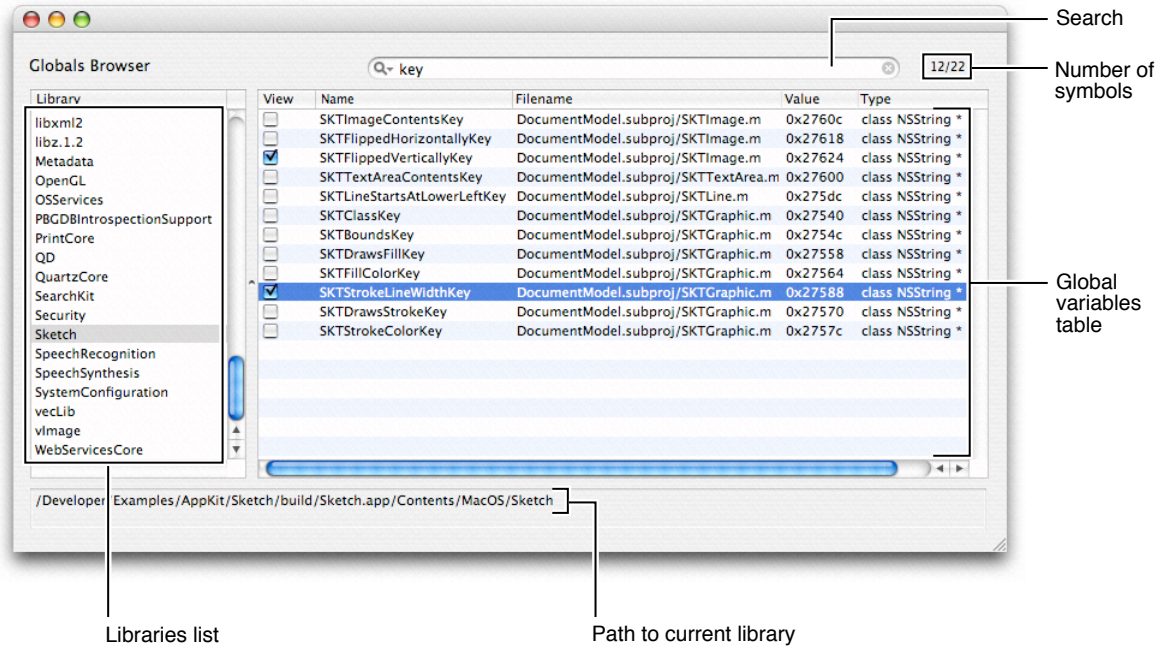
In addition to changing the display format used for a variable's contents in the Value column, Xcode lets you track the value of the variable as an expression or see the contents of the variable in memory. To open the Expressions window and add a variable to it, select the variable and choose Debug > Variables View > View Variable As Expression. The Expressions window is described further in "Using the Expressions Window" (page 368). To see the contents of a variable in memory, using the Memory Browser window, select the variable and choose Debug > Variables View > View As Memory. See "Browsing the Contents of Memory" (page 369) for more information on the Memory Browser window.

You can also access any of the menu items described in this section from the contextual menu displayed when you Control-click a variable.

Using the Globals Browser

By default, Xcode does not display global variables in the variables view. The variables view contains a Globals group, but it is empty. You can choose which global variables to display in the Globals section of the debugger's variable view using the Globals Browser, shown in Figure 34-2. The Globals Browser lets you search for global variables by library.

Figure 34-2 The Globals Browser



You can open the Globals Browser by choosing Debug > Tools > Global Variables. The debugger must be running and execution of the program being debugged must be paused for this item to be available. If you attempt to disclose the contents of the Globals group in the variables view, Xcode automatically opens the Globals Browser.

The list on the left of the Globals Browser window, titled "Library," lists the available libraries, including system libraries and your own libraries. To see a library's global variables, select that library in the list; the global variables defined by that library are shown in the table to the right. In the globals table, you can see:

1. The name of the global variable.
2. The file in which the global variable is defined.
3. The current value of the global variable.
4. The type of the global variable.

You can use the search field at the top of the Globals Browser window to filter the contents of the global variables table. To the right of the search field, Xcode displays the number of global variables currently visible, as a fraction of the total number of global variables in the currently selected library.

To add a global variable to the Globals list in the debugger window's variable view, select the checkbox in the "View" column next to the global variable. You can remove the global variable from this Globals list by deselecting this checkbox at any time.

When you select a library in the Library list, the full path to that library is displayed below the list.

Using the Expressions Window

The Expressions window lets you view and track the value of an expression, including a global value or a function result, over the course of a debugging session. To open the Expressions window, choose Debug >Tools > Expressions. Type the expression you wish to track in the Expression field. Xcode adds the expression, evaluates it, and displays the value and summary for that expression. The display format of the value and summary information is determined by any data formatters in effect.

The expression can include any variables that are in scope at the current statement and can use any function in your project. To view processor registers, enter an expression such as `'$r0'`, `'$r1'`.

You can also add a variable to the Expressions window by selecting the variable in the variables view and choosing Debug > Variables View > View As Expression. To remove an expression, select it and press Delete.

Here are some tips on using the Expressions window:

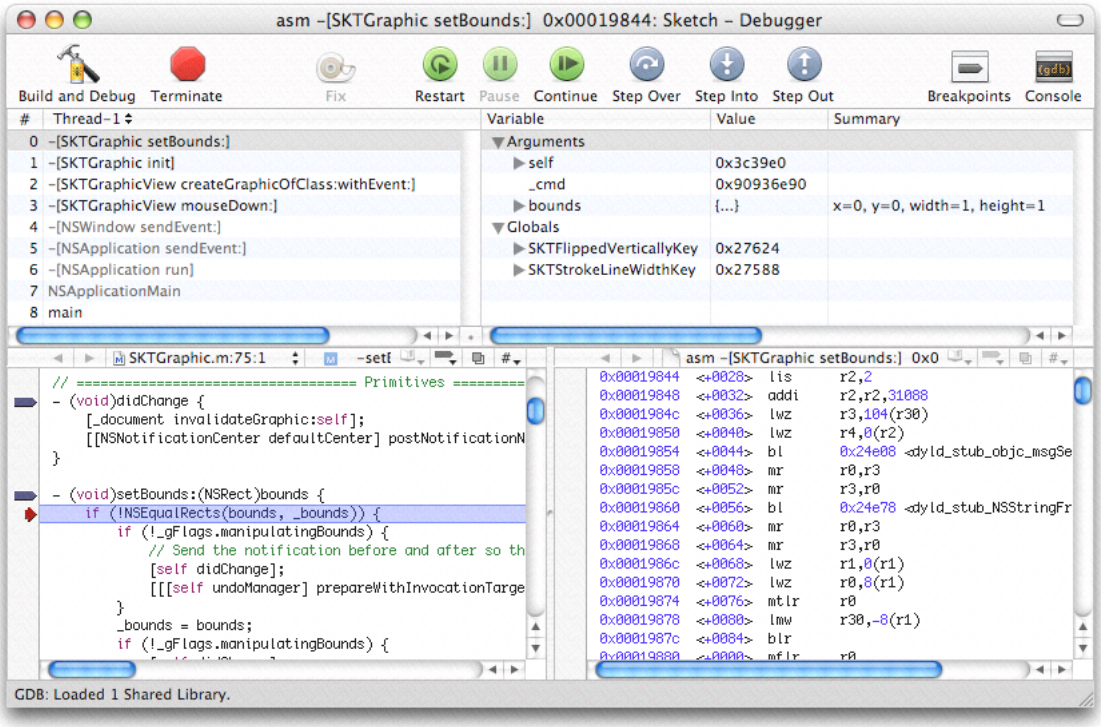
- The expression is evaluated in the current frame. As the frame changes, the expression may go in and out of scope. When an expression goes out of scope, Xcode notes this in the Summary column.
- Always cast a function to its proper return type. The debugger doesn't know the return type for many functions. For example, use `(int)getpid()` instead of `getpid()`.
- Expressions and functions can have side effects. For example, `i++`, increments `i` each time it's evaluated. If you step though 10 lines of code, `i` is incremented 10 times.

Viewing Disassembled Code and Processor Registers

The disassembly view of the debugger window allows you to observe disassembled code. The editor in the debugger window supports three states: Code, Disassembly, and Code and Disassembly. To change the state of the disassembly view choose Debug > Toggle Disassembly Display. The mark next to this menu item indicates the current state of the disassembly view: a line for Disassembly, a checkmark for Code and Disassembly, and no mark for Code.

When there's no source code available for the function or method selected in the Thread list, Xcode displays the disassembled code in the editor. Figure 34-3 shows disassembled code in the Xcode debugger.

Figure 34-3 Viewing disassembled code in the debugger

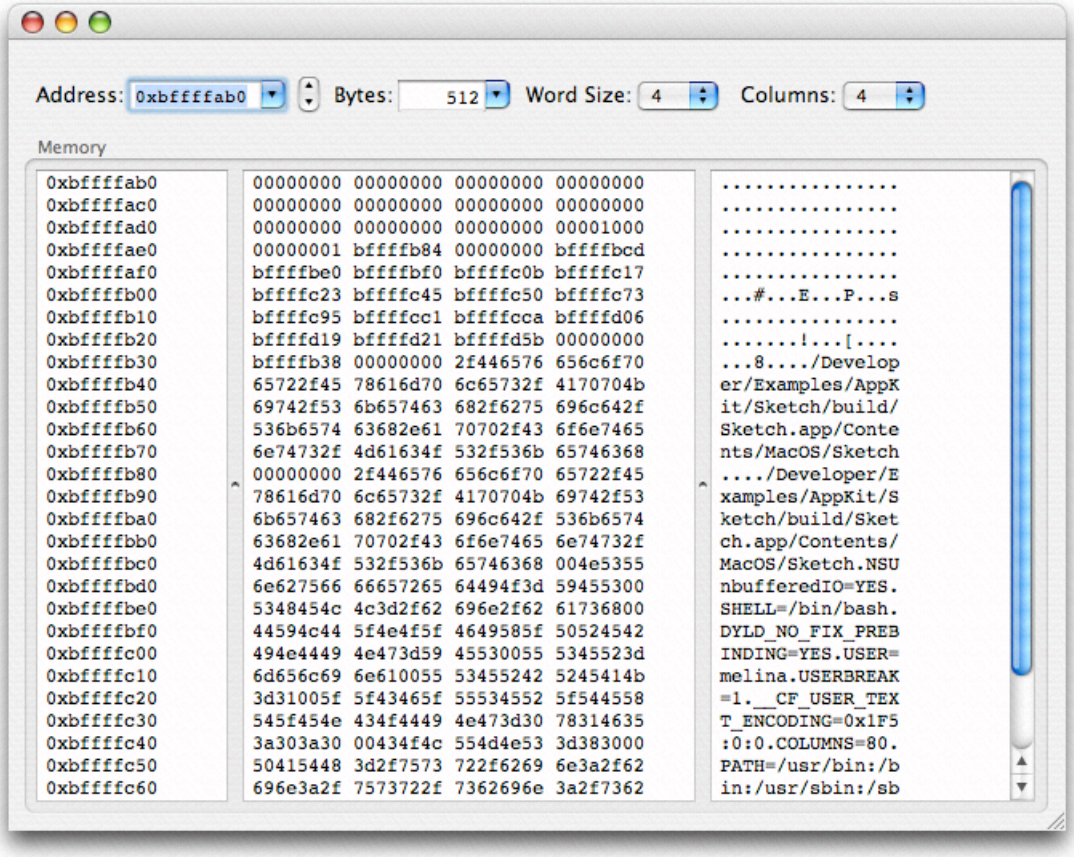


When disassembly view is enabled, the Variables list contains a Registers group containing all the processor registers.

Browsing the Contents of Memory

When execution of the current program is paused in the debugger, you can browse the contents of memory using the memory browser. To open the memory browser, shown here, choose Debug > Tools > Memory Browser. You can also open the Memory Browser to the location of a particular variable; in the Debugger window, select the variable and choose Debug > Variables View > View As Memory.

Figure 34-4 The memory browser window



In the Memory Browser, you can see:

- A contiguous block of memory addresses.
- The contents of memory at those addresses, represented in hexadecimal.
- The contents of memory at those addresses, represented in ASCII.
- The Address field controls the starting address of the memory displayed in the table below. You can enter an address, a variable name or expression. Hexadecimal values must be preceded by `0x`.
- The arrows next to the Address field "step" through memory; clicking the up arrow shows the previous page of memory, while clicking the down arrow returns the next page of memory.
- The Bytes field controls the number of bytes displayed in the memory browser. You can choose one of the options from the pop-up menu in the Bytes field or type a number directly into the field. However, the number of bytes will be rounded up to the next full row of memory fetched.
- The Word Size and Columns menus control how the memory is divided up and displayed. The Word Size menu specifies the size, in bytes, of a single chunk of memory. The Columns menu controls how many units, of the size specified by the Word Size menu, are displayed for an address.

Shared Libraries Window

The Shared Libraries window lets you see which libraries have been loaded by an executable running in Xcode's debugger. To open the Shared Libraries window, choose Debug > Tools > Shared Libraries.

The Module Information table lists all of the individual libraries that the executable links against. In this table, you can see the name and address of each shared library, as well as which symbols the debugger has loaded for that library. The Starting column shows which symbols the debugger loads by default for a given library when the current executable is running. The Current Level column shows which symbols the debugger has loaded for the library during the current debugging session. When an entry has a value in the Address and Current Level columns, the library has been loaded in the debugging session.

The path at the bottom of the window shows where the currently selected library is located in the file system. You can quickly locate a particular library by using the search field to filter the list of libraries by name. You can add and delete libraries from this list using the '+' and '-' buttons.

Using the Shared Libraries window you can also choose which symbols the debugger loads for a shared library. This can help the debugger load your project faster. You can specify a default symbol level for all system and user libraries; you can also change which symbols the debugger loads for individual libraries.

For any shared library, you can choose one of three levels of debugging information:

- All loads all debugging information, including all symbol names and the line numbers for your source code.
- External loads only the names of the symbols declared external.
- None loads no information.

You can specify a different symbol level for system libraries and user libraries. User libraries are any libraries produced by a target in the current project. System libraries are all other libraries.

By default, the debugger loads only external symbols for system and user libraries and automatically loads additional symbols as needed, as described in "Lazy Symbol Loading" (page 349). Disabling the "Load symbols lazily" option, described in "Debugging Preferences" (page 400), changes the default symbol level for User Libraries to "All." This is a per-user setting and affects all executables you define. You can also customize the default symbol level settings for system and user libraries on a per-executable basis, using the Default Level pop-up menus in the Shared Libraries window.

For some special cases—applications with a large number of symbols—you may wish to customize the default symbol level for individual libraries when running with a particular executable. To set the initial symbol level to a value other than the default, make a selection in the Starting column. While debugging you can increase the symbol level using the Current Level column. This can be useful if you need more symbol information while using GDB commands in the Console. Clicking Reset sets all of the starting symbol levels for the libraries in the Module Information table back to the Default value.

Using Fix and Continue

A very powerful feature of Xcode is the ability to modify your executable while it is running and see the results of your modification. This feature is implemented through the Fix command in the Xcode interface.

The Fix command is not intended as a replacement for building your project regularly. Instead, it is a convenience feature for viewing the effect of small changes without restarting your debugging session. This feature is useful in situations where it takes a significant amount of time to reach the place in your application's execution cycle that you want to debug. You can use the feature to learn more about potential bug fixes or to see the immediate results of code changes.

About the Fix Command

The Fix command in Xcode is a way to modify an application at debug time and see the results of your modification without restarting your debugging session. This feature can be particularly useful if it takes a lot of time to reach your application's current state of execution in the debugger. Rather than recompile your project and restart your debugging session, you can make minor changes to your code, patch your executable, and see the immediate results of your changes.

Important: Use of the Fix command is subject to certain requirements and restrictions, which are listed in "Restrictions on Using the Fix Command" (page 375).

GDB and the Fix Command

The process of fixing source files while debugging is tricky. GDB must manipulate your executable while it is running, inserting new code while not disturbing the state of your program execution. The actual process involves compiling your code with special flags and rewriting portions of your binary to call the new code when appropriate. At all times, the Fix command modifies the in-memory image of your executable. It does not permanently modify the files of your original application binary.

When it receives a patched binary, GDB compares that binary against the code in the application's original binary. GDB checks for several modifications that cannot be patched into a running application. If it detects any of these modifications, it reports back to Xcode that it could not incorporate the patch. If this occurs, you can stop debugging and rebuild your application or continue debugging without the patch.

If GDB does not report any problems with your patch, it integrates the patched code into your application's memory image. It does this by making the following modifications:

- GDB modifies functions in your original binary to jump to any patched versions.
- GDB modifies any static or global variables in the patched file to point back to the versions in the original binary.

Debugging With Patched Code

After patching your application, you should be able to continue debugging your code as before. The next time you encounter a patched function, you should see the changes you made appear in the debugger window. However, there are some caveats to be aware of when working with patched code.

If you patch a function that is on the stack, you may not see the results of that patch immediately. GDB is capable of patching the function that is currently at the top of the stack. However, if you patch a function that is further down the calling chain, the patch does not take effect until the next time you call it. Thus, the function must return and be called again before it receives the patch. Until that time, the function on the stack continues to execute the original code.

While your program is paused in the debugger, you can move the program counter around and resume execution from any point in the current function. This feature lets you rerun a patched function from the beginning, or from any point, to account for the changes you made.

If you quit your debugging session for any reason, you must rebuild your program to acquire any changes made by patching. The effects of the Fix command are only applicable to your executable while it is active in the debugger. The reason is that the command does not modify your program's compiled object files. Instead, it creates temporary object files and loads them into the memory space of your process dynamically. When you quit the debugger, GDB discards the temporary object files containing the patches. Recompiling your project recreates your application's original object files from the patched code.

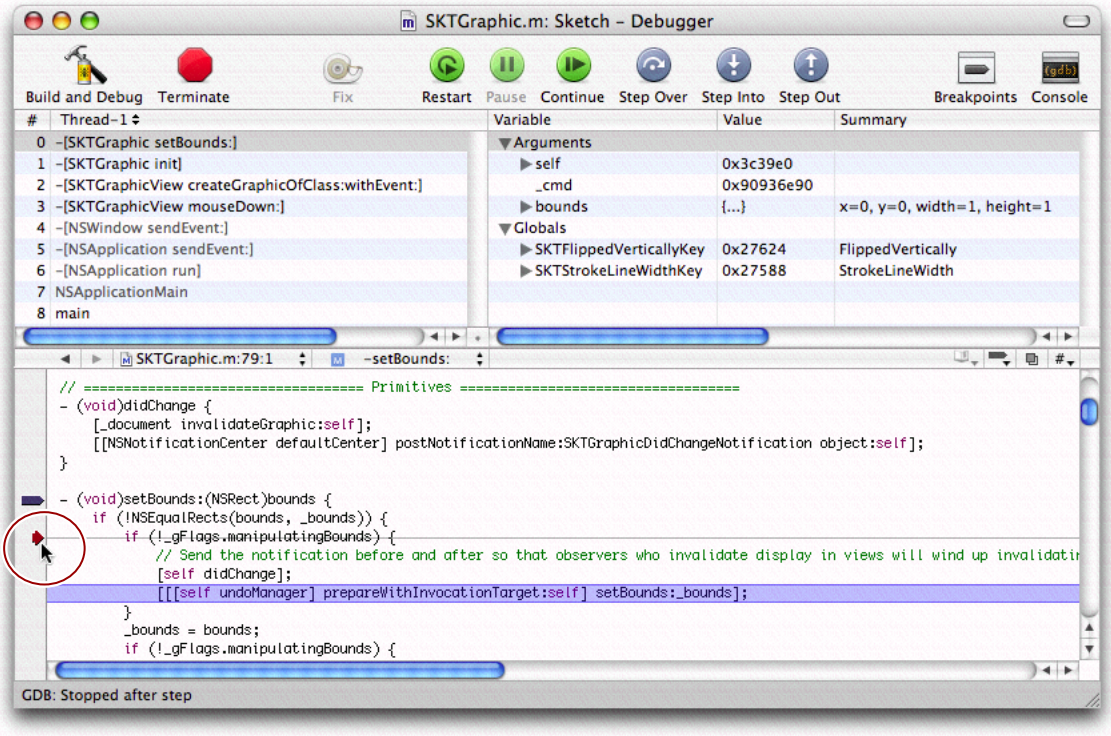
Using Fix and Continue

If you are running your executable in the debugger and you make changes to your source code, you can patch your executable by doing either of the following:

- Click the Fix button.
- Choose Debug > Fix.

If you change more than one source file, you must fix each file separately. Xcode compiles the changes, patches the executable to use the new code, and resumes execution from the location at which the program was halted. If the changes to your code appear before the line at which execution is set to resume, you will not see the effect of your change until that code is called again. However, you can get around this by manually altering the location at which execution resumes. When your program is paused, you can drag the PC indicator—the red arrow pointing to the line of code where execution is paused—to the location at which you want to resume execution, as shown here.

Figure 36-1 Changing the position of the program counter



Xcode automatically locates the correct target to use when creating the fix bundle. For example, if you are debugging an application suite that relies on a framework you created, you can make a change in the framework code. When you click the Fix button, Xcode automatically uses the correct framework target, instead of the target associated with the running executable, to create the fix. Xcode will also follow cross-project references to targets in other projects.

Restrictions on Using the Fix Command

The Fix command in Xcode is a powerful way to make small changes to a source file without restarting your debugging session. Although powerful, there are some things you need to do before you can take advantage of the Fix command. The command itself is enabled only when you are actively debugging an executable. In addition, you must make sure you build your program using the following settings:

- Build using native targets
- Compile your code with GCC version 3.3 or later
- For C++ developers, link your code using ZeroLink
- Build your code without optimizations
- Build your code with debugging symbols enabled

In general, if you build using the Development build style provided by Xcode, the correct settings are used. However, if you try to patch your executable and get a file-not-found error, check your target's build settings and make sure that debugging symbols are turned on, as described in "Generating Debugging Information" (page 345). You can also check the build log to make sure that the patch bundle was created.

The Fix command works on only one file at a time. If you make changes to multiple source files, you must fix each file separately before continuing with your debugging session.

Important: If you attempt to fix multiple files, pause your application until you finish integrating all of the patches.

Restrictions Reported by GDB

There are many types of code changes that cannot be patched. The GDB debugger reports an error if it cannot integrate any of your changes due to a known restriction. If GDB reports one of these errors, you must rebuild your program and restart your debugging session or continue debugging the program without the changes.

GDB recognizes the following changes to your code and reports an error if you try to include them as part of a fix:

- Changes to the number or type of arguments in a function or method that is currently on the stack
- Changes to the return type or name of a function or method that is currently on the stack
- Changes to the number or type of local variables in a function or method that is currently on the stack
- Changes to the type of global or file-static variables
- Symbol type redefinitions, that is, changing a function to a variable or a variable to a function.

Additional Restrictions

In addition to the restrictions reported by GDB, there are additional restrictions that GDB currently does not check. If you attempt to include any of these changes in a patch, your application may crash or exhibit other undefined behavior when it encounters the code. The solution is to avoid using the Fix command for the change. Instead, rebuild your program and restart your debugging session.

The following is a list of changes that cannot be included as part of a fix:

- Changes to a nib file
- Changes to the definition of a structure or union
- The addition of a new Objective-C class
- The addition or removal of class instance variables
- The addition or removal of class methods
- The addition or removal of methods to an Objective-C category. These methods are not registered with the Objective-C runtime and thus cannot be called. (You can fix existing methods in a category.)
- Any reference to an unresolved external variable or function. Link errors of this nature cannot be resolved by the dynamic linker.

- The addition of a static variable across multiple patches in one session. GDB maintains a copy of the static in each patch; however, because there is no original to refer to, each variable remains separate from the others, which can lead to unpredictable results.
- The addition of a function to one file when it is called from a different patched file. New functions are private to the patch file in which they appear.
- The addition of a new `try` block or the addition of a `catch` handler to an existing block
- The addition of a C++ template class specialization
- Changes to functions that require two-level namespaces during linking to prevent known symbol conflicts across different libraries. GDB supports patching two-level namespace binaries but currently does so using flat namespace conventions.

Important: Be aware that other conditions may also cause patched code to fail or exhibit other undefined behavior. If you encounter such a problem, you should rebuild your program and start a new debugging session.

Supported Fixes

Although there are many restrictions to what you can fix, there are also some features that are explicitly supported by the Fix command, including the following:

- Storing pointers to a patched function still works. GDB inserts code to jump from the old function to the newly-patched function. Thus, despite your code having a pointer to the original function, using that pointer executes the patched version.
- You can add new file-local static variables to a file. One caveat to adding such variables is that GDB does not execute any initialization code associated with them. Thus, if you declare a new class instance as a global static, GDB does not execute any constructors or static initializers for the instance. Also, there are limitations on how those variables behave after multiple patches. See "Additional Restrictions" (page 376) for more information.
- You can add new C++ classes as part of a patch.

Remote Debugging in Xcode

Xcode's integrated debugger supports remote graphical debugging when debugging with GDB. Remote debugging lets you debug a program running on another computer. This is good for programs that you cannot easily debug on the host on which they are running. For example, you may be trying to debug a full-screen application, such as a game, or a problem with event handling in your application's GUI. Interacting with the debugger on the same computer interferes with the execution of the program you are trying to debug. In these cases, you have to debug the program remotely.

With Xcode's remote graphical debugging, you can debug a program running on a remote machine within the Xcode debugger, as you would any local executable, without resorting to the command-line.

Note: Standard input (stdin) does not work with remote debugging; if you have a command-line tool that requires user input, you must use gdb's command-line interface to debug your program remotely.

This chapter introduces remote debugging in Xcode and walks you through enabling remote debugging in Xcode. To set up your project for remote debugging, you must perform the steps described in the following three sections:

1. "Configuring Remote Login" (page 379) describes how to configure your local computer and the remote host to allow remote login using SSH public key authentication.
2. "Creating a Shared Build Location" (page 381) describes how to set up a shared build directory that both computers can access via the same path.
3. "Configuring Your Executable for Remote Debugging" (page 381) describes how to configure the executable of the program you wish to debug for remote debugging in Xcode.

Configuring Remote Login

Remote debugging in Xcode relies on SSH public key authentication to create a secure connection with the remote computer. To facilitate authentication, Xcode integrates with `ssh-agent`. This lets you use encrypted private keys for added security without having to re-enter your passphrase each time Xcode establishes a connection to the remote host. If you already use a third party utility to set up the environment variables used by `ssh-agent`, Xcode attempts to use those settings. Otherwise, Xcode uses its own agent for authentication.

Before starting a remote debugging session, you need to be able to login to the remote computer. To do this, you must:

1. Enable remote login on the computer that will host the program being debugged. In the Sharing pane of System Preferences, under "Services," select Remote Login. This allows access to the remote computer via SSH.

2. Ensure that you can connect to the remote host using SSH public key authentication. If you are unsure whether you are using SSH public key authentication, you can test this by logging in to the remote computer with `ssh`. If you are prompted for the user's password, you are not using public key authentication. If you are prompted for a passphrase—or for nothing at all—you are already using public key authentication.

If you are not set up to log in to the remote host using SSH public key authentication, you need to create a public/private key pair, and configure the local and host computers to use it. You can do so with the following steps:

1. Generate a public / private key pair using `ssh-keygen`. On the command line, type the following line:

   ```
   ssh-keygen -b 2048 -t dsa
   ```

   This generates 2048-bit DSA keys. You should see output similar to the following:

   ```
   Generating public/private dsa key pair.
   Enter file in which to save the key (/Users/admin/.ssh/id_dsa):
   /Users/admin/.ssh/id_dsa already exists.
   Overwrite (y/n)? y
   Enter passphrase (empty for no passphrase):
   Enter same passphrase again:
   Your identification has been saved in /Users/admin/.ssh/id_dsa.
   Your public key has been saved in /Users/admin/.ssh/id_dsa.pub.
   The key fingerprint is:
   ```

   Note: Do not leave the passphrase empty; if you do so, your private key will be unencrypted.

2. Copy the public key to the `authorized_keys` file on the remote computer. This file is usually stored at `~/.ssh/authorized_keys`. If the `authorized_keys` file already exists on the remote computer, be careful not to overwrite the file. You can add the public key, which is stored in the file you specified to `ssh-keygen` (`id_dsa.pub` by default), by entering the following on the command line:

   ```
   cat id_dsa.pub >>! ~/.ssh/authorized_keys
   ```

3. Make sure that the `authorized_keys` file is not readable by anybody else. Change the permissions on the file by entering the following on the command line:

   ```
   chmod go-rwx ~/.ssh/authorized_keys
   ```

4. Test the connection by logging in to the remote computer using `ssh`. From the command-line, type "`ssh` *username@hostname*". Ensure that you are not asked for the user's password. If you did not leave it empty in Step 1, you should be prompted for your passphrase, as in the following example:

   ```
   Enter passphrase for key '/Users/admin/.ssh/id_dsa':
   ```

If you are debugging a GUI application, you must be logged into the remote computer as the same user that you connect to using `ssh`. This user must have permission to read the build products.

Creating a Shared Build Location

For remote debugging to work, both computers—both the local computer running Xcode and the remote host running the program you are debugging—must have access to your project's build products and intermediate files via the same access path. You can do this in either of two ways:

- Create a single shared location. This is easiest with a network home directory, although you can use any shared folder that both computers can access. In Xcode, set the build products and intermediates location to this shared folder. If necessary, create symlinks to the build folder on the remote host so the path to the build products is the same on both computers. For details on how to set the location of the build folder in your Xcode project, see "Build Locations" (page 301).
- Copy the files to the remote host. Alternatively, you can copy the build products and intermediate files over to the remote host after each build, although this is considerably less convenient. These must be located at the same path on the remote computer.

Note: When debugging an application built with ZeroLink, it is essential that the build products be accessible using the same path on the remote computer.

Configuring Your Executable for Remote Debugging

Once you have configured both computers to allow for remote login and set up a common build products location, the last step is configuring the executable you want to debug remotely. You may consider creating a separate custom executable environment for remote debugging. Using separate executable environments, you can specify different options for debugging remotely and debugging locally and easily switch between the two modes.

To configure an executable for remote debugging:

1. Select the executable and open an inspector or Info window. Click Debugging to open the debugging pane.
2. Make sure that "GDB" is selected in the "When using" pop-up menu at the top of the Debugging pane. Select the option for "Debug executable remotely via SSH." In the Host field, type in the user you will log in as and the address of the remote host; for example, "`admin@cowpuppy.apple.com`."
3. From the "Use <*device*> for standard input /output" menu, choose "Pipe."

To start a remote debugging session, make sure the active executable is correctly set, then build and debug your product as normal. Before it launches the executable, Xcode displays an authentication dialog that asks you to type in your passphrase. After you have authenticated once, Xcode does not prompt you for your pass phrase again until the next time you initiate a remote debugging session after restarting Xcode.

If you are experiencing problems debugging on the remote host, look in the debugger console for error messages. To view the debugger console, choose Debugger > Console Log.

Customizing Xcode

The Xcode development environment does its best to provide an easy, intuitive interface for the most common development tasks that you face. However, there are many different factors that affect your requirements for your development environment. Luckily, Xcode is also a very flexible tool, providing many different ways to customize the development process.

The following chapters describe many of the ways in which you can customize Xcode to make it a more productive and custom-tailored environment for your development. Some features are of particular use to developers who are familiar with BBEdit, CodeWarrior, or MPW, but most should be useful to any developer.

In particular, these chapters show you how to customize Xcode's user interface, change user settings with Xcode Preferences, and add functionality to the Xcode application using the User Scripts menu. In addition, many of the chapters that appear in previous sections of this document also describe ways in which you can use Xcode to customize your development environment. It does not describe how to extend the Xcode application.

Xcode offers many opportunities for customization, including:

- Customizing the build process. Xcode provides many different ways for you to customize the behavior of the build system. The Copy Files and Run Script build phases, described in "Build Phases" (page 249), let you add your own operations to the build process for a target; "Build Rules" (page 261) let you customize the way in which files in a target's build phases are processed. You can use Shell Script targets to add reusable custom operations to the build process; external targets let you build using an external build tool of your own choice. See "Special Types of Targets" (page 236). You can also invoke `xcodebuild` from shell scripts to automatically build one or more products.
- "Setting Command-Line Arguments and Environment Variables" (page 340) shows how to set environment variables that are available to your executable when running in the Xcode development environment.
- "Using Smart Groups to Organize Files" (page 87) describes how to use smart groups to organize the files of a large project.
- "Project Window Layouts" (page 64) describes how to customize the configuration of the project window and other Xcode windows.
- "Customizing Key Equivalents" (page 385) shows you how to set Command-key equivalents for menu items and keyboard equivalents for common editing tasks.
- "Using Scripts To Customize Xcode" (page 415) describes how to use shell commands and scripts to customize your programming environment.

Customizing Key Equivalents

Xcode lets you change the Command-key equivalents for its menu items and lets you change the keyboard equivalents for common editing tasks, such as paging through a document or moving the cursor. You can choose a pre-defined set of key bindings for menu items and other tasks, or create your own set. The pre-defined sets include sets that mimic BBEdit, Metrowerks CodeWarrior, and MPW.

To work with key bindings, choose Preferences from the Xcode menu, then click Key Bindings. Figure 38-1 shows the Key Bindings preferences pane, with the key equivalents for the Xcode menu visible. Menu item key equivalents do not require the Command key.

You can use the pop-up menu and buttons to choose a predefined set of key bindings, copy one of the supplied sets and delete sets you have created. Use the Duplicate button to copy the set that is currently selected in the Key Binding Sets pop-up menu. You cannot edit any of the Xcode's preexisting key bindings sets; to customize key bindings, duplicate an existing set and modify the copy. Use the Delete button to delete the currently selected set. You can't delete a supplied set, only a set you have created or copied.

Figure 38-1 The Key Bindings pane in the Xcode Preferences window

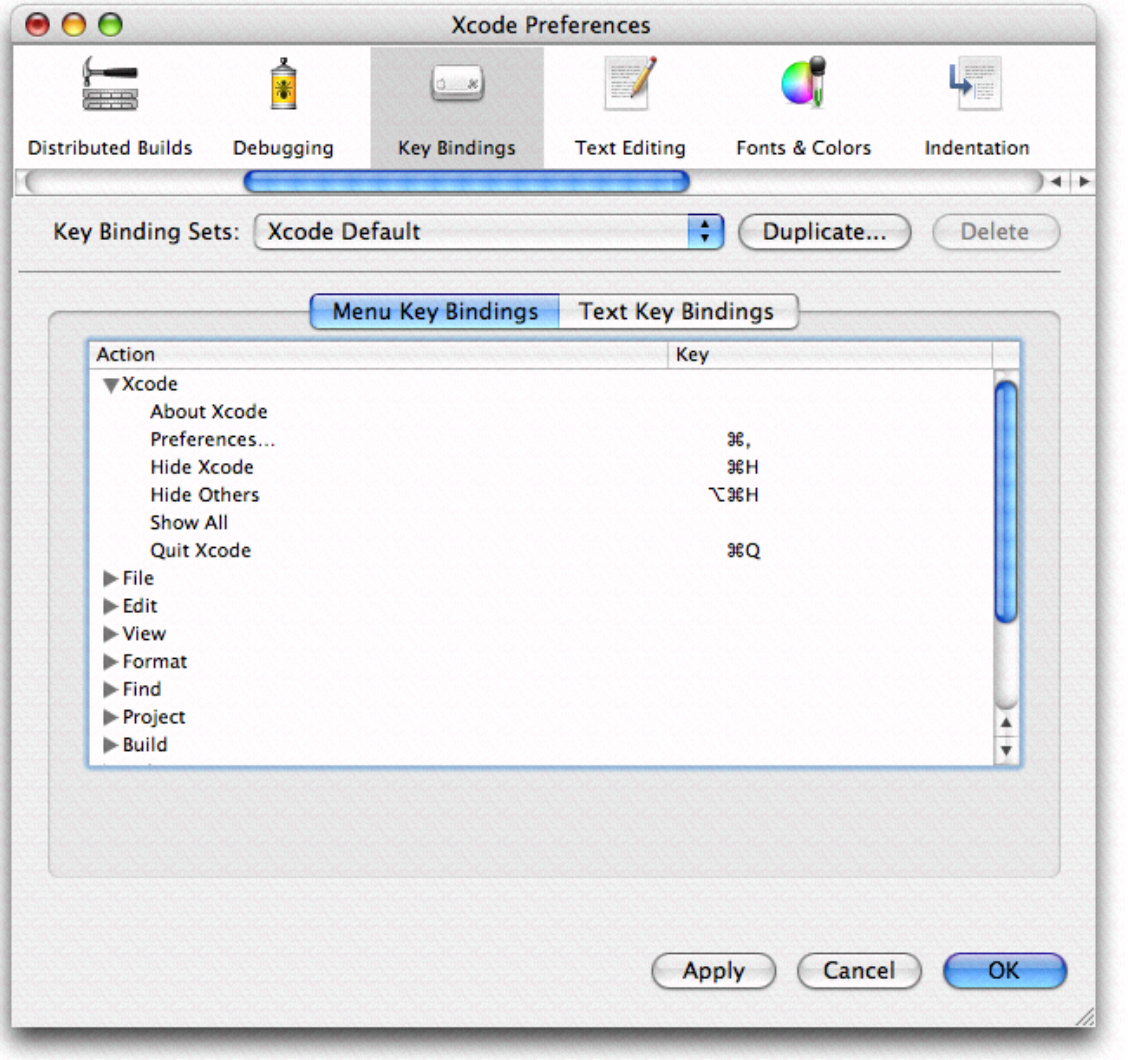


Figure 38-2 shows the supplied sets provided in the Key Binding Sets pop-up menu, including sets that mimic the menu and shortcut equivalents of Metrowerks CodeWarrior, and other popular Macintosh IDEs and text editors.

Figure 38-2 Supplied sets of key bindings

Customizing Command-Key Equivalents for Menu Items

The Menu Key Bindings pane provides access to almost all of the menus and menu items in Xcode. The Text Key Bindings pane, which provides access to key equivalents for common editing tasks, is described in "Customizing Keyboard Equivalents for Other Tasks" (page 389).

Xcode currently lists key equivalents using the traditional menu glyphs shown in Figure 38-3 (not all glyphs are shown).

Figure 38-3 Some of the glyphs for available key equivalents

Command	⌘	Shift	⇧
Option	⌥	Control	^
Left Arrow	←	Home	↖
Right arrow	→	End	↘
Up arrow	↑	Page up	⇞
Down arrow	↓	Page down	⇟
Backspace	⌫	Return	⏎
Delete	⌦	Enter	⌅
Escape	⎋		

The following steps describe how to create a custom set of key bindings, based on the supplied CodeWarrior set, and how to add a menu item key equivalent. In this example, the equivalent automatically opens the release notes, an option available from the Help menu.

1. Navigate to the Key Bindings pane in the Preferences window:

   Xcode > Preferences, then click Key Bindings.

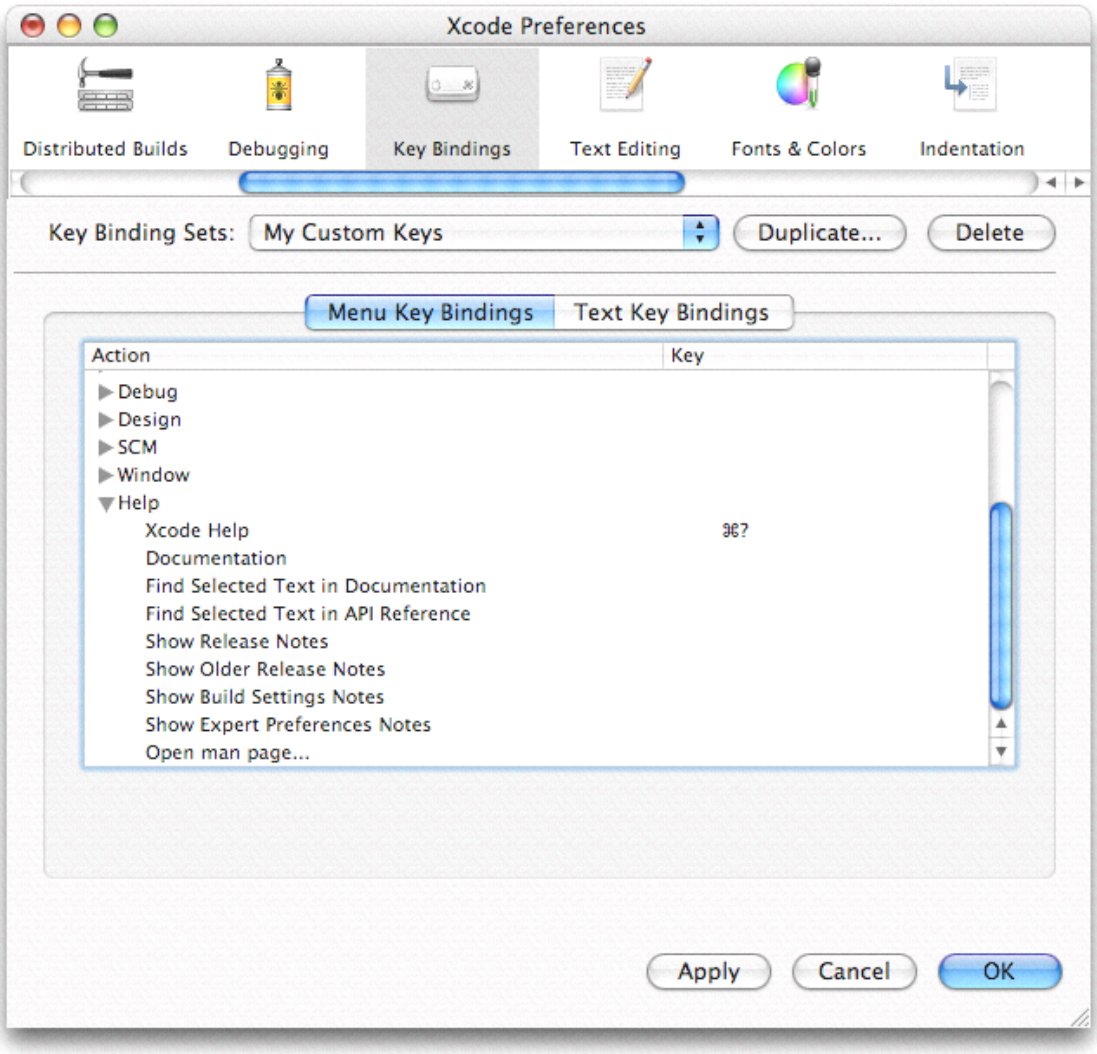
2. Choose Metrowerks Compatible in the Key Binding Sets pop-up menu, as shown in Figure 38-2.

3. Click the Duplicate button to create a copy of that set. When prompted for a name for the set, type `My Custom Keys`.

4. Click the Menu Key Bindings tab.

5. Scroll to the Help menu and click the disclosure triangle in the Action column to open the menu. Figure 38-4 shows the Help menu actions.

Figure 38-4 The Help menu commands



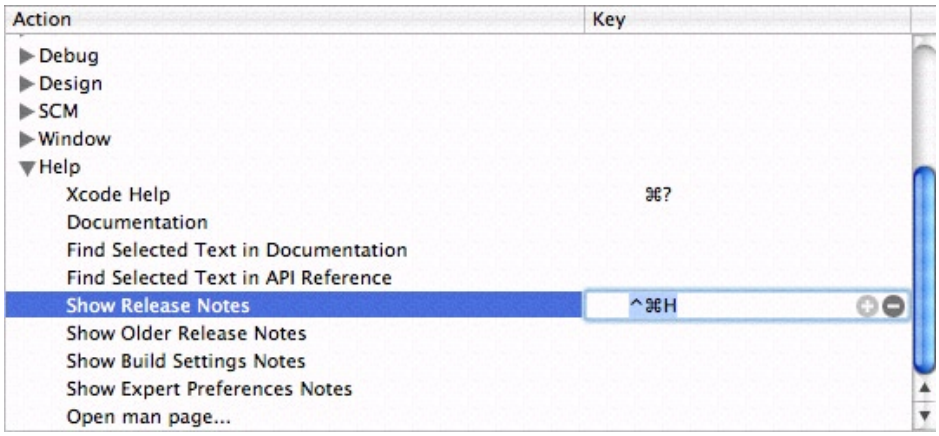
6. Double-click in the Key column next to the Show Release Notes menu item to open an editing field, then type Command-Control-H (holding the keys down simultaneously). The result is shown in Figure 38-5.

   If you try to assign a key equivalent that is already assigned to another action in the current key binding set, Xcode displays a message indicating which action it is assigned to below the key bindings table.

   Note that Xcode displays the letter "h" in its capitalized form. Whether or not you include the Shift key as part of a menu key equivalent, Xcode shows letters as they appear in menus (as capitals).

You can use the - button (shown in Figure 38-5) to clear a menu key equivalent. You can use the + button to assign multiple equivalents to a single action (where you can use any of the equivalents to initiate the action).

Figure 38-5 Editing the menu key equivalent for the Show Release Notes menu item



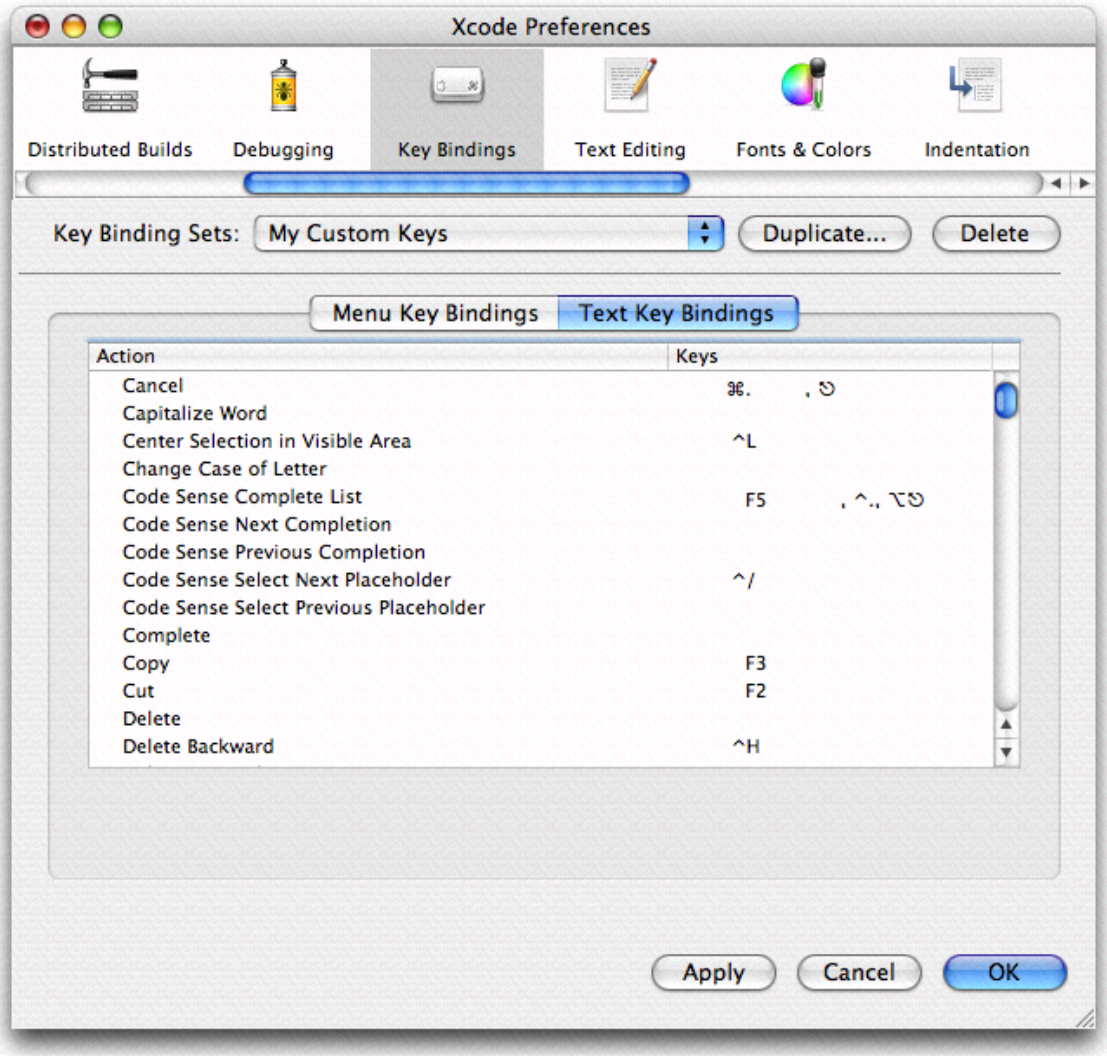
7. You can repeat the previous step to add or change other menu key equivalents.
8. Click the Apply button to apply your changes.
9. You can now type Command-Control-H to choose the Show Release Notes menu item, which opens the release notes from the Help menu. If you open the Help menu, you will see the keystroke glyphs shown in Figure 38-5 next to the Show Release Notes menu item.

Customizing Keyboard Equivalents for Other Tasks

You can customize keyboard equivalents for tasks such as editing and formatting text, cursor movement, and project navigation using steps similar to those described for menu items in "Customizing Command-Key Equivalents for Menu Items" (page 387). You can start by copying one of the predefined sets of key bindings shown in Figure 38-2 (page 386). In addition to its default settings, Xcode provides sets that are compatible with BBEdit, Metrowerks CodeWarrior, and MPW.

Figure 38-6 shows the contents of the Text Key Bindings pane, with some of the key equivalents for Text Editing actions visible. The current set is the My Custom Keys set, created in "Customizing Command-Key Equivalents for Menu Items" (page 387).

Figure 38-6 Text Key Bindings in the Preferences window

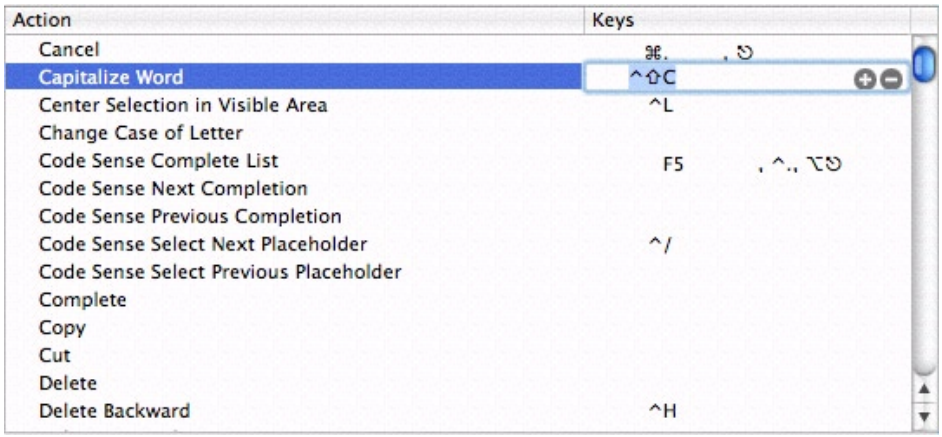


The following steps show how to set a shortcut for the Capitalize Word action:

1. Navigate to the Key Bindings pane in the Preferences window: Xcode > Preferences, then click Key Bindings.
2. Choose My Custom Keys (created previously in "Customizing Command-Key Equivalents for Menu Items" (page 387)) in the Key Binding Sets pop-up menu.
3. Click the Text Key Bindings button.
4. Double-click in the Keys column next to the Capitalize Word item to open an editing field, then type Control-Shift-C (holding the keys down simultaneously). The result is shown in Figure 38-7.

You can use the - button (shown in Figure 38-5) to clear a keyboard equivalent. You can enter more than one key combination for an action by clicking the + button.

Figure 38-7 Editing the Text Editing shortcut for Capitalize Word



5. You can repeat the previous step to add or change other keyboard equivalents for editing actions (or other actions not shown here).
6. Click the Apply button to apply your changes.
7. You can now type Control-Shift-C to capitalize the currently selected word in a project.

Xcode Preferences

Another simple way in which you can customize your work environment in Xcode is through Xcode's preferences. Xcode provides preferences that let you enable and disable features, customize the behavior of operations in Xcode, and provide default values for project and file-level settings.

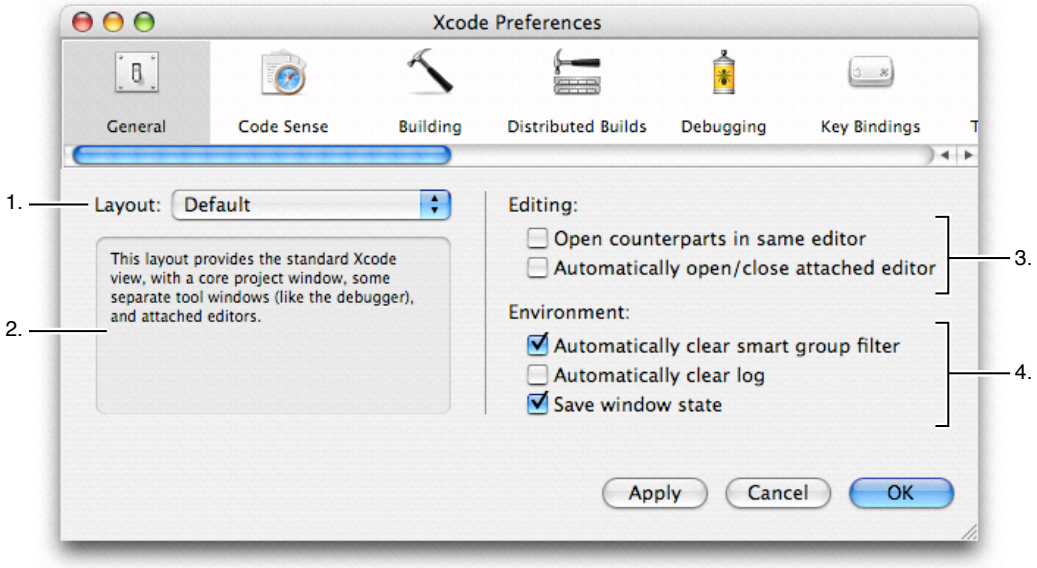
To open the Xcode Preferences window, choose Xcode > Preferences. The Xcode Preferences window contains options for controlling many different parts of the Xcode application. This chapter describes each of the preference panes in Xcode Preferences and the options that they contain.

You can open the preference pane for a particular group by clicking its icon. Note that any changes you make in a preference pane do not take effect until you click OK or Apply.

General Preferences

The General pane of the Xcode Preferences window lets you control general environment settings for the Xcode application, such as your project window configuration and windowing preferences. Figure 39-1 shows the General pane of Xcode Preferences.

Figure 39-1 General Xcode Preferences



Here is what the pane contains:

1. Layout menu. This menu lets you choose the project window configuration for all open projects. See "Project Window Layouts" (page 64) for a description of the available layouts.

2. Layout description. This gives you a brief explanation of the layout selected in the Layout menu.

3. Editing options. These options control Xcode's windowing policy for editor windows. See "The Xcode Editor Interface" (page 169) to learn more about Xcode's editor. The options are:

    a. "Open counterparts in same editor" controls how Xcode displays files when jumping to a related header or source file, or to a related symbol definition or declaration. If this option is selected, Xcode always opens file and symbol counterparts in the current editor window; otherwise, it opens the a separate editor to display the counterpart. By default, this option is disabled. See "Opening Header Files and Other Related Files" (page 165) describes how to jump to a file's or symbol's counterpart.

    b. "Automatically open/close attached editor" controls when Xcode shows or hides the editor pane attached to the project, Build Results, Project Find, or Debugger windows. If this option is selected, Xcode will automatically show the attached editor for a window when you select an editable item. By default, this item is disabled. See "Using the Attached Editor" (page 172) for more information on attached editors.

4. Environment options. These control general user interface settings across many different Xcode windows. These are:

    a. "Automatically clear smart group filter" controls whether Xcode clears the contents of the Search field in the toolbar when you select a different item in the Groups & Files list. If this option is selected, Xcode clears the contents of the Search field when you change the currently selected smart group. Otherwise, the contents of the Search field are preserved. This option is enabled by default. See "Searching and Sorting in the Detail View" (page 61) for more information on Search field.

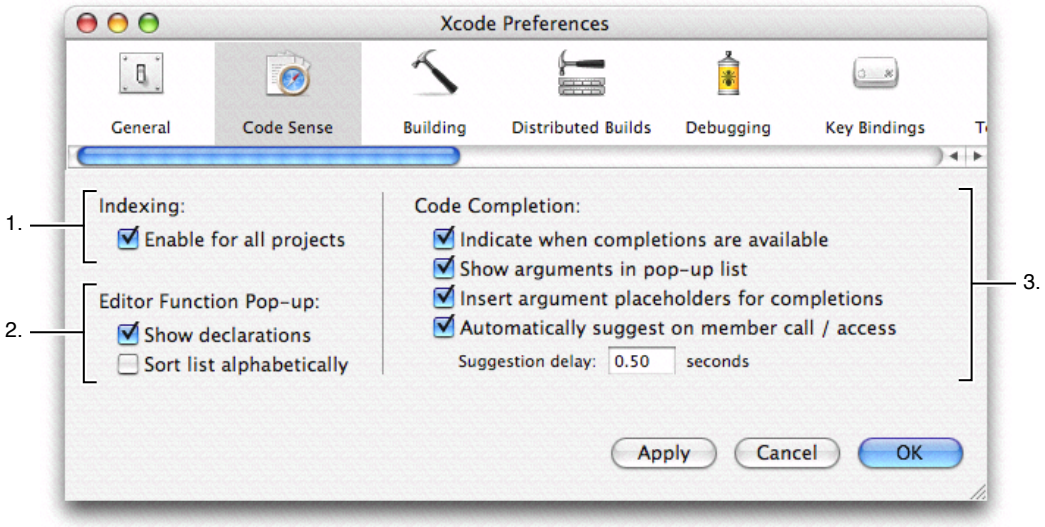
    b. "Automatically clear log" controls whether Xcode clears the contents of the run and console logs between sessions. If this option is selected, Xcode automatically clears the contents of the Run Log and Console Log windows each time you relaunch an executable in Xcode. Otherwise, information from the current session is appended to the existing log. This option is disabled by default.

    c. "Save window state" controls whether Xcode saves the state of the windows in a project across sessions. If this option is selected, Xcode saves the state of a project's open windows when you close the project and restores those windows when you next open the project. Otherwise, Xcode does not save the state of a project's windows; reopening a project opens only the project window. By default, this option is enabled.

Code Sense Preferences

The Code Sense pane of the Xcode Preferences window contains options for controlling project indexing and Xcode's interface for code completion. Figure 39-2 shows the Code Sense pane of Xcode Preferences.

Figure 39-2 Code Sense Preferences



Here is what the pane contains:

1. Indexing. The "Enable for all projects" option lets you turn indexing on and off for all projects. If this option is selected, Xcode generates a symbolic index for each project that you open. Xcode uses the information in this index to provide features such as code completion, symbol definition searching, and more.

   Disabling this option turns indexing off for all projects and discards any existing project index. Features that rely on this index, such as the Project Symbols smart group, will not work. This option is enabled by default. See "Code Sense" (page 104) for more information.

2. Editor Function Pop-up. These options let you control the information and appearance of the function pop-up menu that appears in the navigation bar of Xcode's editor. The function pop-up is described further in "The Function Pop-up Menu" (page 175). These options are:

   a. "Show declarations" controls whether Xcode shows function and method declarations in the function pop-up menu. If this is selected, the function pop-up menu shows both declarations and definitions; otherwise it displays only definitions. This option is enabled by default.

   b. "Sort list alphabetically" controls how Xcode sorts the contents of the function pop-up menu. If this option is selected, the items in the menu appear in alphabetical order. Otherwise, they are sorted in the order in which they appear in the file. This option is disabled by default.

3. Code Completion. These options control Xcode's code completion interface. Code completion, described in "Code Completion" (page 189), suggests matches for symbol names and keywords as you type. The options are:

   a. "Indicate when completions are available" controls whether Xcode shows you when it has suggestions for completing the current text. If this option is selected, Xcode underlines the text you have entered in the editor when there are suggestions for matching symbol names or keywords. Otherwise, Xcode does not underline the text, although completion suggestions are still available to you, as described in "Using Code Completion" (page 189). This option is enabled by default.

b. "Show arguments in pop-up list" controls whether Xcode displays arguments for functions and methods in the completion list. When this option is selected, Xcode displays the return type and arguments for functions and methods in the list of completion suggestions. Otherwise, Xcode shows only the symbol name. This option is on by default.

c. "Insert argument placeholders for completions" controls whether Xcode inserts the arguments to a function or method when you insert a completion suggestion. If this option is enabled, inserting a function or method using code completion also inserts placeholders for arguments. Otherwise, Xcode only inserts the symbol name. This option is on by default.

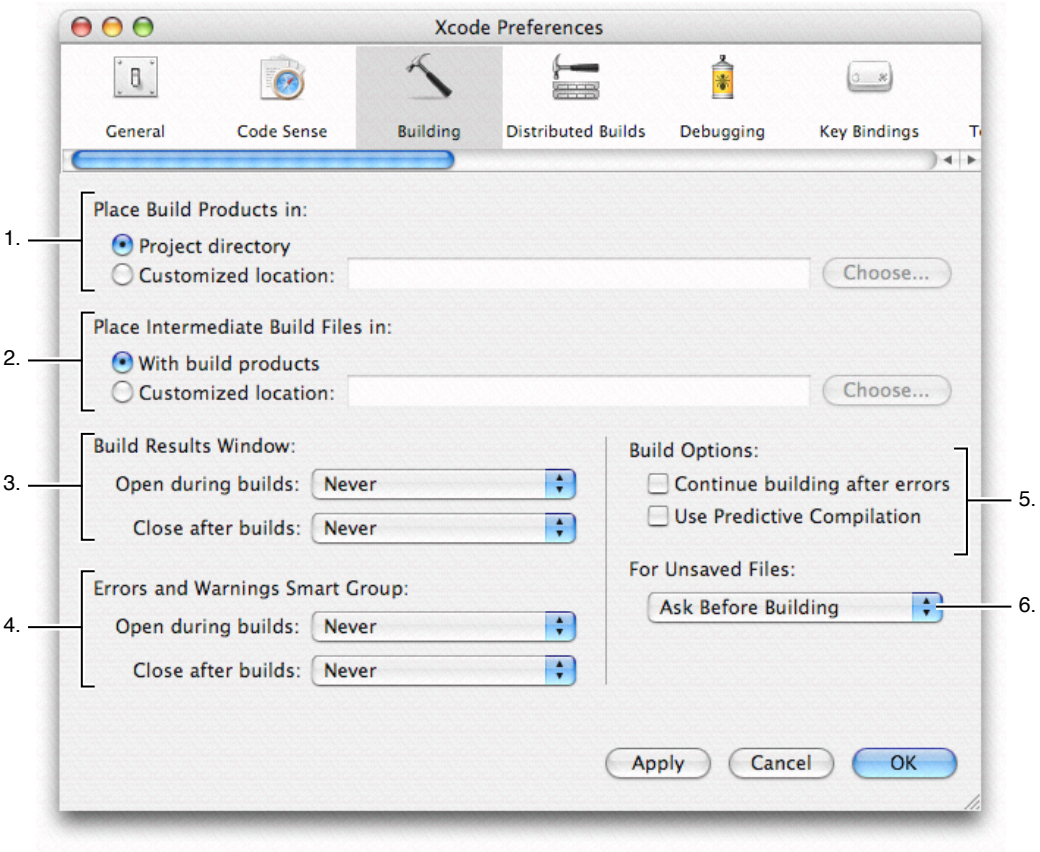
d. "Automatically suggest on member call / access" controls whether Xcode automatically shows completion suggestions when accessing class members. When this option is enabled, Xcode automatically displays the completion list in the context of a method call or field access. Otherwise, you must bring up the completion list yourself, as described in "Using Code Completion" (page 189). This option is off by default.

e. "Suggestion delay" specifies the amount of time, in seconds, after you stop typing, before Xcode automatically displays the completion list. This delay only applies if "Automatically suggest on member call / access" is enabled. By default, this delay is set to half a second; to change it, type the new value in the field.

Building Preferences

The Building pane of the Xcode Preferences window contains options for setting default build locations and controlling the display of the Build Results window. Figure 39-3 shows the Building pane of Xcode Preferences.

Figure 39-3 Building Preferences



Here is what the pane contains:

1. Place Build Products In. This setting controls the location at which Xcode places the output generated when you build a target—such as an application, tool, and so forth—for all new projects that you create. The default value for this option is "Project directory"; for all development builds of a target, Xcode places the product that is generated in the build folder in your project directory.

   To specify a different folder for build products, select "Customized location"; Xcode places build products in the folder specified in the associated text field. You can type the path to that folder in the field or click Choose to locate the folder using the standard navigation dialog.

   You can also override this setting for individual projects. For information on build locations in Xcode, see "Build Locations" (page 301).

2. Place Intermediate Files In. This setting controls the location at which Xcode places the intermediate files generated by the Xcode build system when you build a target—such as object (`.o`) files—for all new projects that you create. The default value for this option is "With build products"; for all builds of a target, Xcode places the intermediate files that are generated in the location specified for build products. This is the location indicated by the "Place Build Products In" option, described above, either in the Building preferences or at the project level.

   To specify a different folder for build products, select "Customized location"; Xcode places intermediate files in the folder specified in the associated text field. You can type the path to that folder in the field or click Choose to locate the folder using the standard navigation dialog.

You can also override this setting for individual projects. For information on build locations in Xcode, see "Build Locations" (page 301).

3. Build Results Window. These pop-up menus control when Xcode opens the Build Results window or pane. These menus are:

    a. Open during builds. This menu lets you specify when (or whether) Xcode automatically opens the Build Results window or pane when you build a target. By default, the value of this option is "Never" and Xcode does not automatically open the Build Results window.

    b. Close after builds. This menu lets you specify when (or whether) Xcode automatically closes the Build Results window or pane after a build completes. By default, the value of this option is "Never" and Xcode does not automatically close the Build Results window.

    These menus, and their possible values, are described further in "Specifying When Detailed Build Results are Shown" (page 308).

4. Error and Warnings Smart Group. These pop-up menus control when Xcode selects the Errors and Warnings smart group in the Groups & Files list and discloses its contents before bringing the project window to the front. These menus are:

    a. Open during builds. This menu lets you specify when (or whether) Xcode automatically shows the contents of the Errors and Warnings smart group when you build a target. By default, the value of this option is "Never" and Xcode does not automatically open the Errors and Warnings smart group.

    b. Close after builds. This menu lets you specify when (or whether) Xcode automatically shows the contents of the Errors and Warnings smart group after a build completes. By default, the value of this option is "Never" and Xcode does not automatically close the Errors and Warnings smart group.

    These menus, and their possible values, are described further in "Viewing Errors and Warnings" (page 309).

5. Build Options. These options let you choose how Xcode reacts to build errors and whether Xcode tries to speed up the build process by using Predictive Compilation. The options are:

    - "Continue building after errors" controls what Xcode does when it encounters a build error. If this option is selected, Xcode continues trying to build the next file in the target after encountering an error. Otherwise, Xcode stops the build. By default, this option is disabled.
    - "Use Predictive Compilation" turns Xcode's Predictive Compilation feature on and off. When this option is selected, Xcode tries to shorten build time by beginning to compile source files in the background, even before you initiate a build. Otherwise, Xcode does nothing until you hit Build. By default, this option is disabled. Predictive Compilation is described further in "Predictive Compilation" (page 331).

6. For Unsaved Files. This menu determines what Xcode does with files that have unsaved changes when you start a build. The options are:

    - Ask Before Building. If there are files containing unsaved changes when you click Build, Xcode displays a dialog asking you what to do before it starts the build. You can choose to save or discard the changes and then continue the build, or you can cancel the build operation. This is the default value for the For Unsaved Files menu.
    - Always Save. Xcode automatically saves all unsaved changes before it begins building.

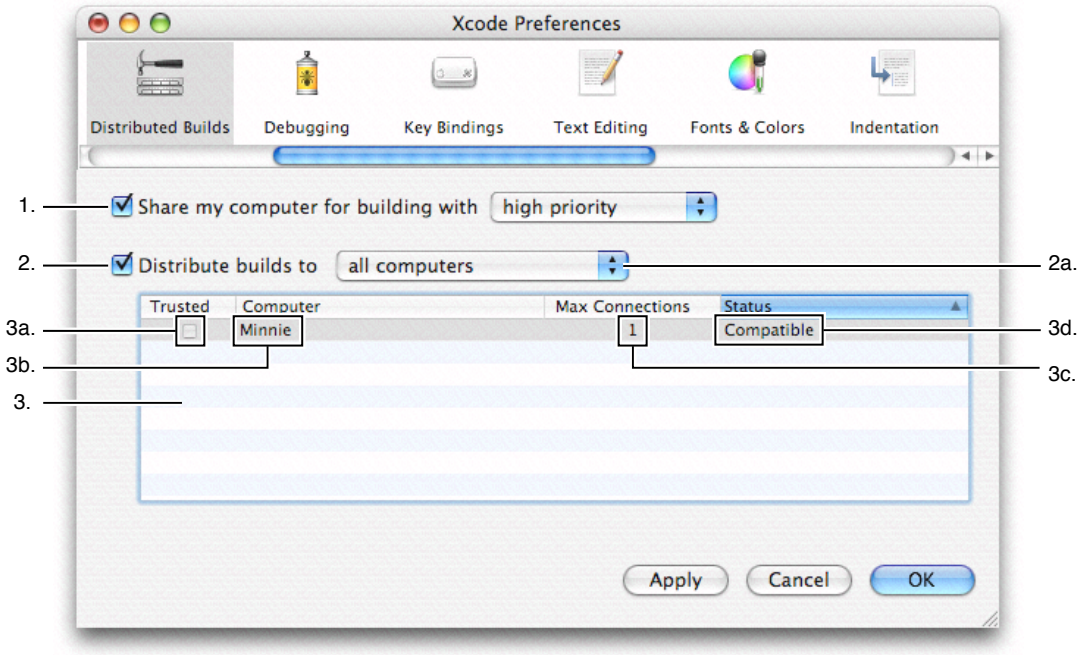
- Never Save. Xcode automatically discards all unsaved changes before it begins building.
- Cancel Build. If there are files containing unsaved changes, Xcode cancels the build operation.

For more information on building in Xcode, see "Building a Product" (page 301).

Distributed Builds Preferences

The Distributed Builds pane of the Xcode Preferences window contains options for distributing build tasks to other computers on your network. Figure 39-4 shows the Distributed Builds pane of Xcode Preferences.

Figure 39-4 Distributed Builds Preferences



Here is what the pane contains:

1. "Share my computer for building with." Selecting this option makes your computer available to other computers on your local network for performing build tasks. This option is disabled by default.

   If you enable this option, the pop-up menu specifies the priority assigned to build tasks that are distributed to your computer. This determines the amount of processing time allocated for that task. You can choose high, low, or medium priority. The default value is "high priority."

2. "Distribute builds to." This option turns distributed builds on and off. If this option is selected, Xcode looks for available computers on your local network and distributes build tasks to them. Otherwise, all building is done on your local computer. This option is off by default.

3. Table of available computers. Xcode automatically discovers computers on the local network that are available for building. This table shows all of the computers available on the current network. For each available computer, the table shows:

    a. Whether the computer is trusted. You can restrict which computers Xcode distributes builds to, using the pop-up menu described in item 4. If you do so, you must indicate which of the available computers you "trust;" that is, which of the computers you allow Xcode to use for building. A computer is trusted if the checkbox in the Trusted column is selected.

    b. The name of the computer. The Computer column displays the Bonjour name of the computer that is available for building.

    c. The number of builds the computer can handle. Dual-processor machines can use both processors for building. The Max Connections columns shows the number of processors, and therefore the maximum number of connections to, the computer.

    d. Whether the available computer is compatible. To distribute build tasks to another computer, that computer must be running the same versions of Mac OS X, GCC, and Xcode as you are running on your computer. The Status column indicates whether the computer broadcasting its services is compatible with yours

4. The "Distribute builds to" pop-up menu lets you restrict the computers that Xcode distributes builds to. You can choose:

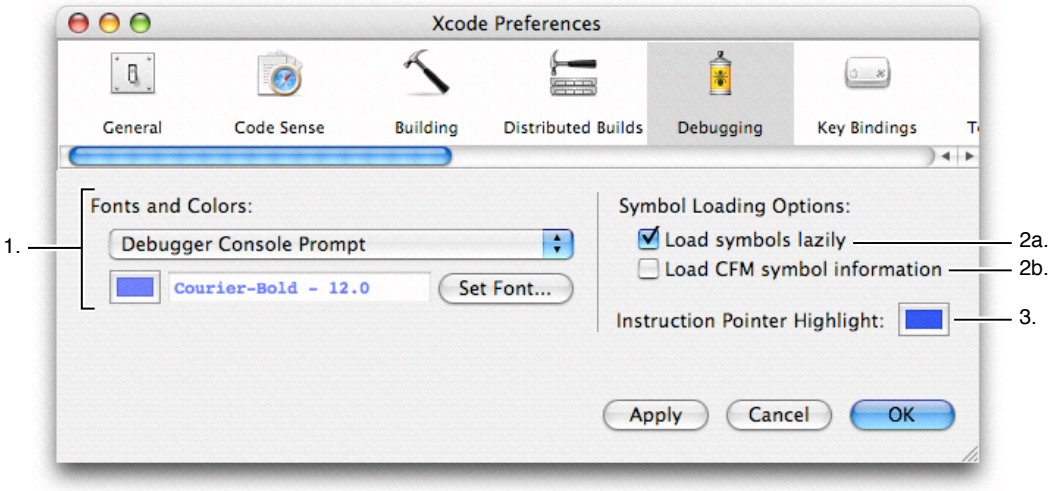
    - All computers. Xcode distributes build tasks to any of the available computers, listed in the table. By default, Xcode distributes builds to any available computer.
    - Trusted computers only. Xcode distributes builds only to those computers marked "Trusted" in the table.

For more information about distributed builds in Xcode, see "Distributing Builds Among Multiple Computers" (page 328).

Debugging Preferences

The Debugging pane of the Xcode Preferences window contains options for controlling the appearance of Xcode's debugger window and specifying how the debugger loads debugging symbols. Figure 39-5 shows the Debugging pane of Xcode Preferences.

Figure 39-5 Debugging Preferences



Here is what the pane contains:

1. Fonts and Colors. These options let you control the color and font used for text in the debugger console and standard I/O windows. To see or change the color or font for a text item, select it form the pop-up menu. To change the font for an item, click Set Font and choose a new font from the Fonts window. To change irs color, click the color well and choose a new color from the Colors window.

2. Symbol Loading Options. These options let you control which symbols Xcode's debugger load and when. These options are:

    a. "Load symbols lazily" controls when Xcode loads debugging symbols for an executable when you run it in Xcode's debugger. If this option is selected, Xcode loads symbols only as they are needed. Otherwise, Xcode loads all symbols for the executable and its libraries when you launch it in the debugger. This option is enabled by default. You can further customize which symbols are loaded in the Shared Libraries window. See "Shared Libraries Window" (page 371) for more information.

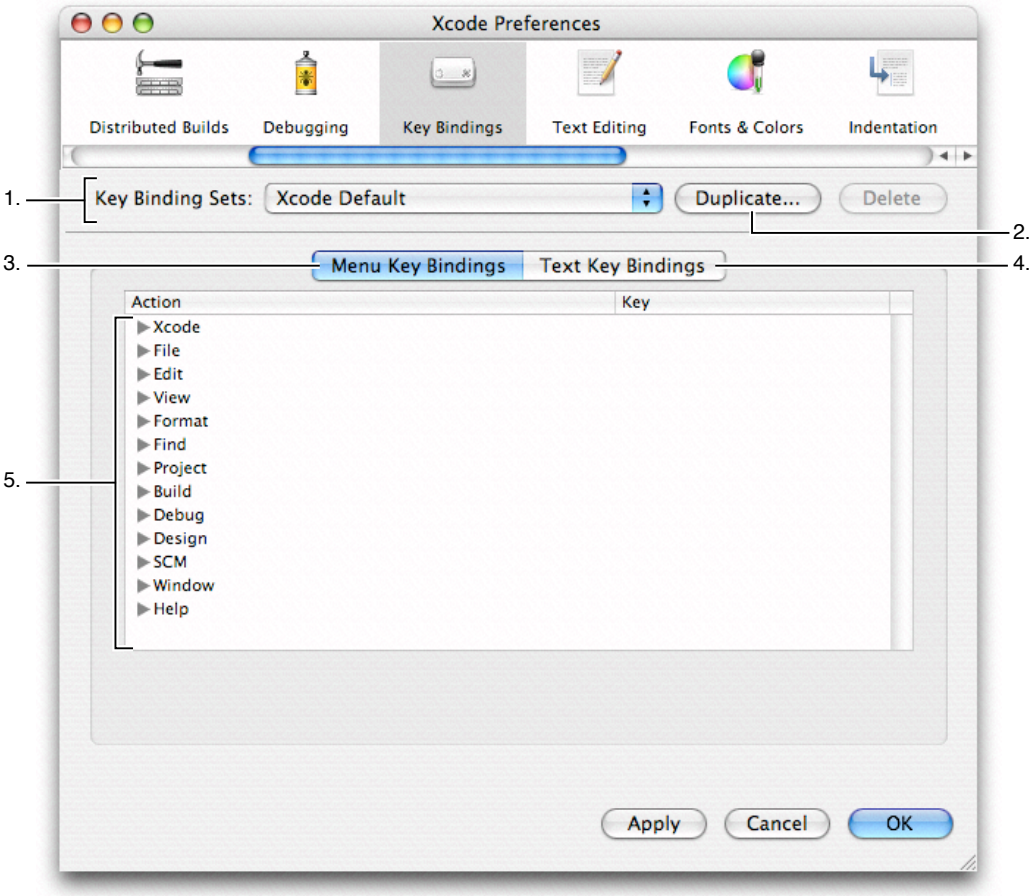
    b. "Load CFM symbol infomration" specifies whether Xcode loads symbol information for CFM-based binaries.

3. Instruction Pointer Highlight. This option controls the color used to highlight the location of the instruction pointer in the debugger window when execution of the current program is stopped. To change this color, click the color well and choose the new color from the Colors window. For more information on the instruction pointer and controlling execution of your code, see "Controlling Execution of Your Code" (page 353)

Key Bindings Preferences

The Key Bindings pane of the Xcode Preferences window lets you see and customize the list of Xcode commands and their keyboard shortcuts. Figure 39-6 shows the Key Bindings pane of Xcode Preferences.

Figure 39-6 Key Bindings Preferences



Here is what the pane contains:

1. Key Bindings Sets. This menu lets you choose which set of key bindings are in effect. Xcode provides four predefined sets: Xcode Default, BBEdit Compatible, Metrowerks Compatible, and MPW Compatible. You can also add your own custom sets of key bindings.

2. Duplicate. You cannot edit any of the built-in key bindings sets. To create your own set of custom key bindings, click this button to create a copy of the current set and edit that copy.

3. Menu Key Bindings. This pane lists the key bindings for menu items in Xcode.

4. Text Key Bindings. This pane lists the key bindings for text editing actions in Xcode's editor.

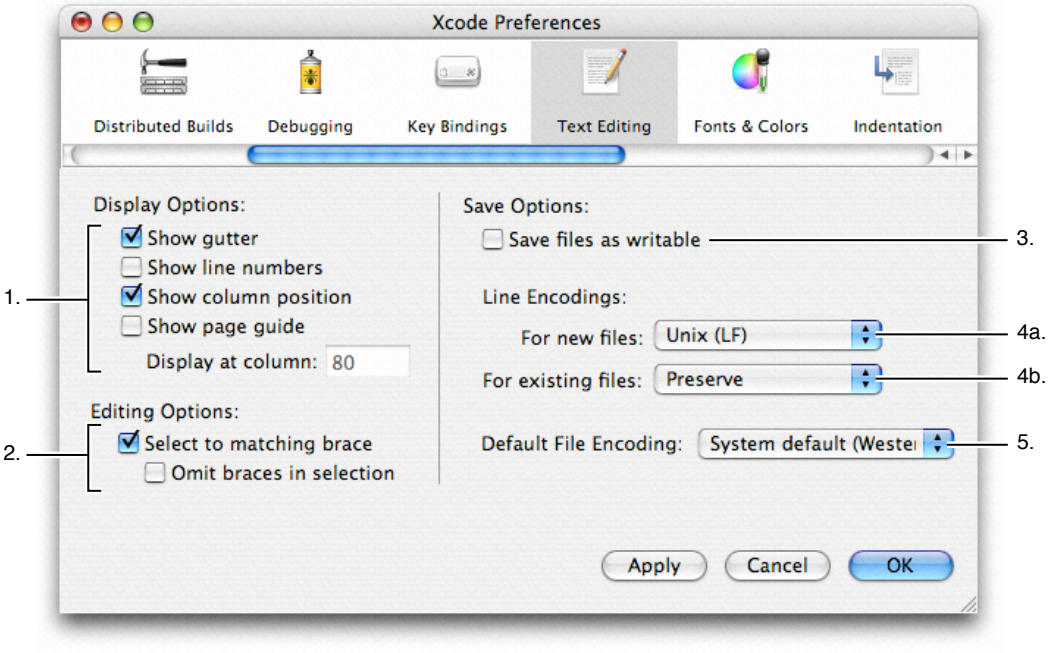
5. Key bindings. The key bindings table lists all of the available key bindings in Xcode. The Action column lists the Xcode action—a menu item or text editing command—and the Key column lists the keyboard shortcut for that action. To edit the key binding for a command, double-click in the Key column and type the key combination. You can assign more than one keyboard shortcut to an action. To add additional key combinations, click the plus (+) button.

For more information on customizing key bindings, see "Customizing Key Equivalents" (page 385).

Text Editing Preferences

The Text Editing pane of the Xcode Preferences window lets you control display options for the Xcode editor, as well default settings for saving files that you edit in Xcode's editor, such as file encoding, and so forth. Figure 39-7 shows the Text Editing pane of Xcode Preferences.

Figure 39-7 Text Editing Preferences



Here is what the pane contains:

1. Display Options. These options control the appearance of Xcode's editor, whether it appears as a separate window or as a pane attached to another Xcode window. See "Controlling the Appearance of the Code Editor" (page 179) to learn more about changing the appearance of Xcode's editor. The options are:

    a. "Show gutter" controls when Xcode displays the gutter in the editor. The Xcode gutter shows information about the current file such as the location of breakpoints, line numbers, and the location of errors or warnings. If this option is selected, Xcode always shows the gutter in all open editors; otherwise, it shows the gutter only when debugging. By default, this option is enabled. See "Displaying the Editor Gutter" (page 180) for more information.

    b. "Show line numbers" controls whether Xcode shows a file's line numbers in the editor gutter. If this option is selected, Xcode shows line numbers for a file whenever the editor gutter is visible. By default, this item is disabled. See "Displaying the Editor Gutter" (page 180) for more information

    c. "Show column position" controls whether Xcode shows the current position of the cursor in the function pop-up of the editor. If this option is selected, Xcode shows the current position of the insertion point, as the offset from the left margin of the line. This is measured in number of characters. By default, this item is disabled. See "Navigating Source Code Files" (page 173) for more information.

d. "Show page guide" controls whether Xcode displays the page guide in the editor. If this option is selected, Xcode displays a grey guide line in the editor to show you the right margin of the editor. This is just a guide, and does not actually affect the margin width in the editor. By default, this item is disabled. See "Displaying a Page Guide" (page 180) for more information.

e. The "Display at column" field controls the column position at which Xcode displays the page guide discussed in the previous item. This position is specified in number of characters. To change the position at which the page guide is shown, enter a new number in the field. See "Displaying a Page Guide" (page 180) for more information.

2. Editing Options. These options control Xcode's selection behavior for source code. The options are:

   a. "Select to matching brace" controls whether Xcode automatically selects text contained in braces when you double-click the brace. If this option is selected, double-clicking a brace, bracket, or parenthesis in a source code file automatically selects the text up to, and including, the matching brace. By default, this option is enabled. See "Matching Parentheses, Braces, and Brackets" (page 187).

   b. "Omit braces in selection" controls whether Xcode includes the braces themselves in text selected by double-clicking a brace, bracket, or parenthesis. If this option is selected, double-clicking a brace, bracket, or parenthesis selects the text between the braces, but not the braces themselves. By default, this item is disabled. See "Matching Parentheses, Braces, and Brackets" (page 187) for more information on attached editors.

3. Save Options. The Save Options on the right side of the Text Editing pane let you specify how Xcode stores files that you edit in the Xcode editor. The next several items in this list describe these options.

   "Save files as writable" controls the permissions that Xcode uses for files that it saves. If this option is selected, Xcode adds write permission to files that you edit and save in Xcode. Otherwise, Xcode preserves permissions for files as they are on disk. Files that you create in Xcode already have write permission. This option is disabled by default. See "Saving Files" (page 166) for a description of the available layouts.

4. Line Encodings. These menus control the default line endings used for files in Xcode. You can use Unix, Windows, or Mac line endings for files that you open and edit in Xcode; the type of line endings used for a file can affect which file editors and other tools can interpret the file. See "Changing Line Endings" (page 163) for more information on line endings. The menus are:

   a. For new files. This menu lets you choose the type of line endings used for files that Xcode creates. You can choose Unix, Mac, or Windows line endings. The default value for this setting is Unix.

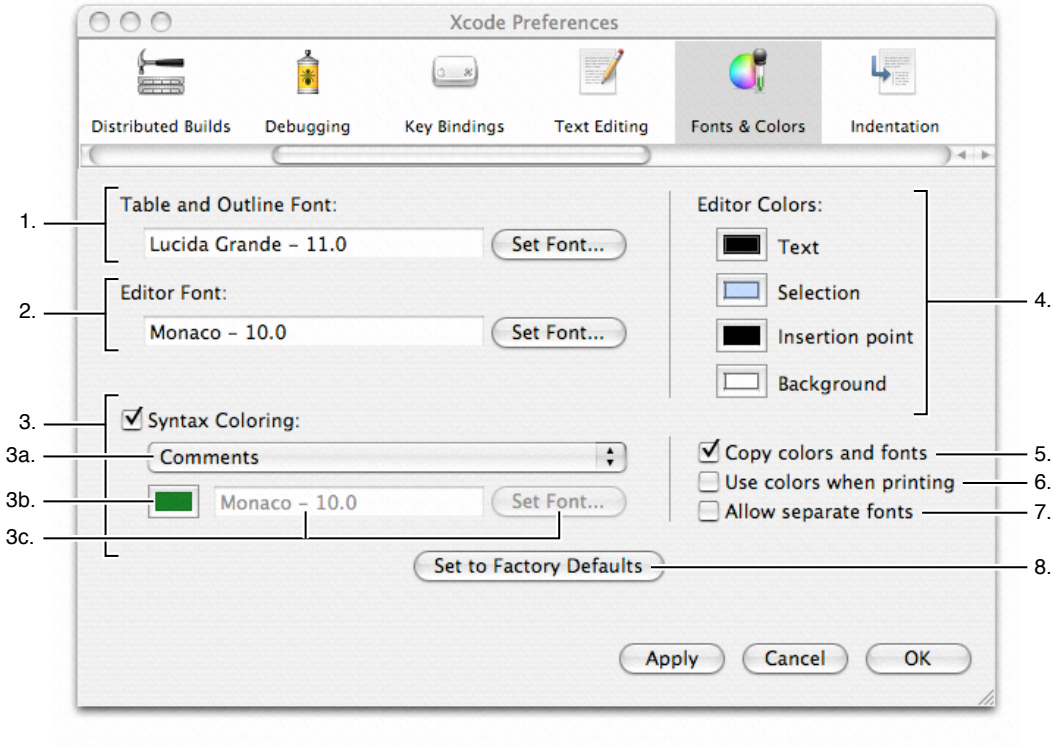
   b. For existing files. This menu lets you choose the type of line endings used for preexisting files that you open and edit in Xcode. If you choose Unix, Mac, or Windows from this menu, Xcode saves all files that you open and edit in Xcode with line endings of this type, changing them the next time it saves the file, if necessary. If you choose Preserve from this menu, Xcode uses whatever type of line endings the file already has. The default value for this setting is Preserve.

5. Default File Encoding. This menu lets you choose the default file encoding that Xcode uses for new files that you create in Xcode. By default, this is set to "System default;" Xcode uses the same default file encoding as the system. You can choose any of the file encoding supported by your Mac OS X system from this menu. See "Choosing File Encodings" (page 163) for more information.

Fonts & Colors Preferences

The Fonts & Colors pane of the Xcode Preferences window controls the font used in Xcode's user interface, and the font and font colors used for text in Xcode's editor, including syntax coloring used to display different code elements in source code files. Figure 39-8 shows the Fonts & Colors pane of Xcode Preferences.

Figure 39-8 Fonts & Colors Preferences



Here is what the pane contains:

1. Table and Outline Font. This option controls the font used to display text in the Groups & Files list and detail views in Xcode. The name and size of the current font is shown in the Table and Outline Font field. To change this font, click Set Font and choose a new font from the Font window. See "The Project Window and its Components" (page 55) for more information on the Groups & Files list and the detail view.

2. Editor Font. This option controls the font that Xcode uses for plain text in the Xcode editor. The name and size of the current font is shown in the Editor Font field. To change this font, click Set Font and choose a new font from the Font window. You can change the color used for plain text in the editor using the Text color well, described in item 4a. See "The Xcode Editor" (page 169) for more information on the Xcode editor.

3. Syntax Coloring. This option, and the options below it, control syntax coloring for code elements in source code files in Xcode's editor. If this option is selected, Xcode uses different fonts and / or colors for each of the different code elements listed in the pop-up menu when it displays source code files in the editor. This option is enabled by default. See "Setting Syntax Coloring" (page 183) to learn more. The options are:

a. The syntax coloring pop-up menu shows the various code elements that Xcode's syntax coloring supports. To see or modify the font and font color for a particular type of code element, choose it from this menu.

b. The syntax coloring color well controls the font color used for the code element currently selected in the pop-up menu described in item 3a. To change this color, click the color well and choose a new color from the Colors palette.

c. The syntax coloring font field displays the name and size of the font used to display the code element currently selected in the pop-up menu indicated in item 3a. To change the font for the current item, click the Set Font button and choose a font in the Fonts window. All code elements use the font specified by the Editor Font option described in item 2, unless the "Allow separate fonts" option is selected, as described in item 7.

4. Editor Colors. These options control the colors used for various elements in Xcode's editor. To change the color for an item, click its color well and choose a new color from the Colors window. See "Controlling the Appearance of the Code Editor" (page 179) for a description of how to control the editor's appearance. You can change the colors used for the following items:

   a. Text. This option controls the color used for plain text in the editor. When syntax coloring is off, this is the font color used for all text in the editor. When syntax coloring is enabled, this is the color used for all text that does not constitute one of the code elements described in item 3a.

   b. Selection. This option controls the color used to highlight selected text in the editor.

   c. Insertion point. This option controls the color used for the insertion point in the editor.

   d. Background. This option controls the color used for the background of the editor window or pane.

5. "Copy colors and fonts" controls whether Xcode preserves syntax coloring when copying and pasting from an editor. If this option is selected, Xcode copies color and font information to the clipboard when you copy or cut text from an Xcode editor. This option is enabled by default.

6. "Use colors when printing" controls whether Xcode preserves syntax coloring information when printing files from and editor. If this option is selected, Xcode prints source code files using the colors and fonts specified by the syntax coloring rules. This option is disabled by default.

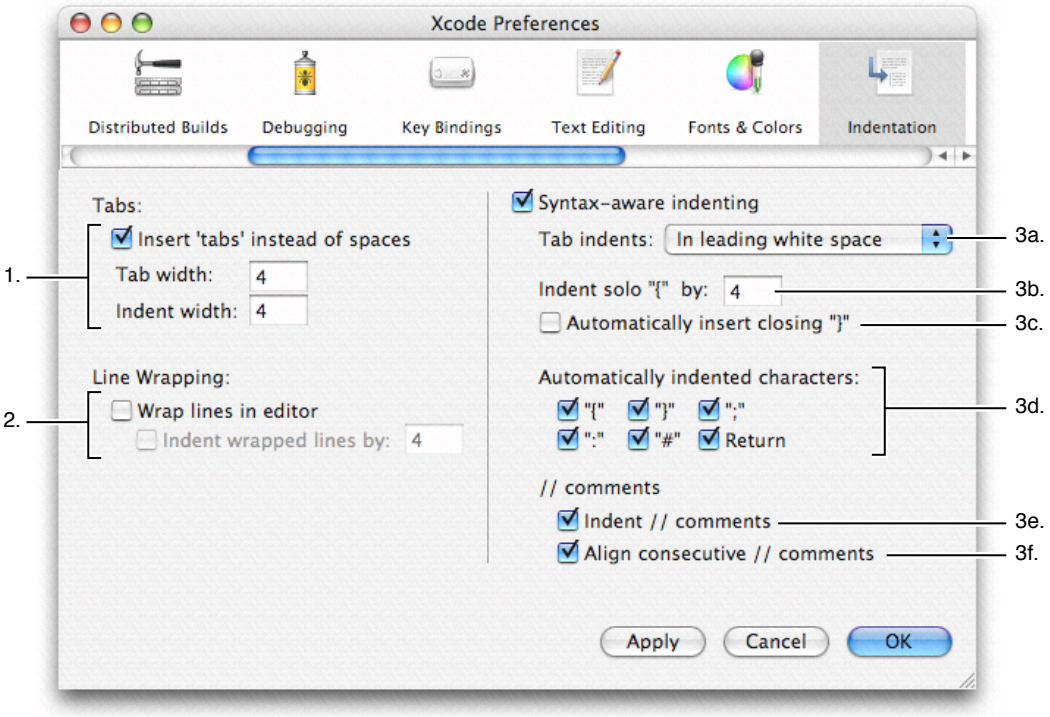
7. "Allow separate fonts" controls whether Xcode lets you set fonts individually for the code elements used for syntax coloring, as described in item 3. If this option is selected, you can set the font for each code element separately. Otherwise, Xcode uses the font specified by the Editor Font option (described in item 2) for all text in an editor, regardless of the code element. This option does not affect the colors used for various code elements when syntax coloring is on. This option is disabled by default.

8. Set to Factory Defaults. This button returns all of the font and color options in the Fonts & Colors pane to their original settings.

Indentation Preferences

The Indentation pane of the Xcode Preferences window controls formatting options for files in Xcode's editor. Figure 39-9 shows the Indentation pane of Xcode Preferences.

Figure 39-9 Indentation Preferences



Here is what the pane contains:

1. Tabs. These options control how Xcode inserts space into files when editing files. See "Setting Tab and Indent Formats" (page 186) for more information. These options are:

    a. "Insert 'tabs' instead of spaces" controls whether Xcode inserts tab characters when you press the Tab key in the editor. If this option is selected, pressing the Tab key inserts a Tab character. Otherwise, Xcode inserts space characters. This option is enabled by default.

    b. Tab width. This option specifies the default width, in number of characters, used to display tabs in the editor. To change the width of a tab, type a number in the text field. You can override this setting for individual files, as described in "Inspecting File Attributes" (page 161).

    c. Indent width. This option specifies the default width, in number of characters, used to indent lines in the editor. To change the indentation width, type a number in the text field. You can override this setting for individual files, as described in "Inspecting File Attributes" (page 161).

2. Line Wrapping. These options control how Xcode wraps lines in files displayed in the editor. These options do not affect line breaks or other information stored with the file, simply how the file is displayed onscreen. See "Wrapping Lines" (page 185) for more information. These options are:

    a. "Wrap lines in editor" controls whether Xcode wraps lines in an editor. If this option is selected, Xcode wraps text to the next line when it reaches the outer edge of the text editing area onscreen. Otherwise, Xcode only moves text to the next line when a return or new line characters is inserted. This option is disabled by default.

b. "Indent wrapped lines by" controls how Xcode indents text that it wraps to the next line in an editor. If this number is greater than 0, Xcode indents wrapped text by the specified number of characters, as a visual indication that the text has been wrapped (as opposed to being moved to the next line by the insertion of a carriage return or new line character). This option is only available if "Wrap lines in editor is enabled." To change the amount by which lines are indented, type a new number in the field.

3. Syntax-aware indenting. This option, and the options below it, control automatic formatting options for source code in Xcode editors. If this option is selected, Xcode assists you in writing source code by automatically inserting formatting information appropriate for the current context. This option is disabled by default. See "Indenting Code" (page 185) to learn more. The options are:

   a. Tab indents. This menu controls when pressing Tab in the editor inserts an indentation. You can choose the following:

      In leading white space. Pressing Tab inserts an indentation only at the beginning of a line or following a space. If syntax-aware indenting is enabled, this is the default value for the Tab indents setting.

      Never. Pressing Tab never causes an indentation.

      Always. Pressing Tab always causes an indentation.

   b. "Indent solo "{" by" controls the amount by which a "{" character on a line by itself is indented. If this number is greater than 0, Xcode automatically indents a left brace that appears on a line by itself (that is, a left brace that is preceded by a new line or carriage return) by the specified number of characters. The default value of this field is 0.

   c. "Automatically insert closing "}"" controls whether Xcode automatically inserts a matching right brace when you type an opening brace. If this option is selected, typing an opening brace causes Xcode to insert a matching closing brace. If syntax-aware indenting is enabled, this option is off by default.

   d. "Automatically indented characters." These options control which characters trigger Xcode to automatically cause an indentation. When any of the following options is selected, typing that character in an editor causes Xcode to indent the current line or the following line. By default, each of these characters is selected (when syntax-aware indenting is enabled).

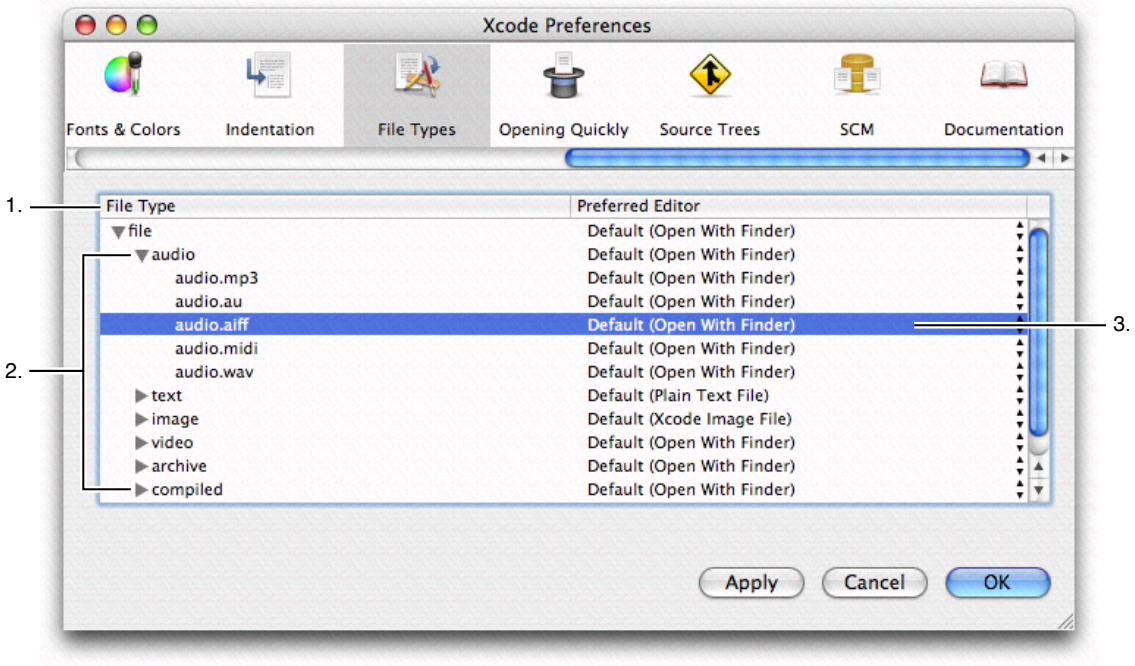
   e. "Indent // comments" controls whether Xcode automatically indents C++-style comments. If this option is selected, Xcode automatically indents comments beginning with / /. This option is enabled by default when syntax-aware indenting is enabled.

   f. "Align consecutive //comments" controls whether Xcode automatically indents consecutive C++-style comments to the same level. This option is enabled by default when syntax-aware indenting is enabled.

File Types Preferences

The File Types pane of the Xcode Preferences window displays all of the file types recognized by Xcode and the default application for opening files of that type. Figure 39-10 shows the File Types pane of Xcode Preferences.

Figure 39-10 File Types Preferences



The file types are shown in the table in the File Types pane. This table contains the following information:

1. File type. The File Type column contains entries for each file type that Xcode recognizes. For example, audio.mp3.

2. File type categories. File type entries are organized into groups, from most general to most specific. For example, the `audio.mp3` and `audio.aiff` file types belong to the "audio" group, which belongs to the "file" group. In this way, you can control the default editor used for an entire class of files. To see the file types in a group, click the disclosure triangle next to that group.

3. Preferred Editor. The pop-up menu in this column lets you choose the preferred editor for a file type. You can choose:

    - Open With Finder. This opens files and folders of this type with the application chosen for it in the Finder.
    - External Editor. This lets you choose a specific application to use for opening files and folders of this type.
    - Default. This opens files and folders of this type in the default editor assigned to it by Xcode. For file types that Xcode can open and display, this is usually the Xcode editor. For example, the default preferred editor for the `text.rtf` file type is Default (RTF File); Xcode opens files of this type in its own editor and interprets them as RTF files.

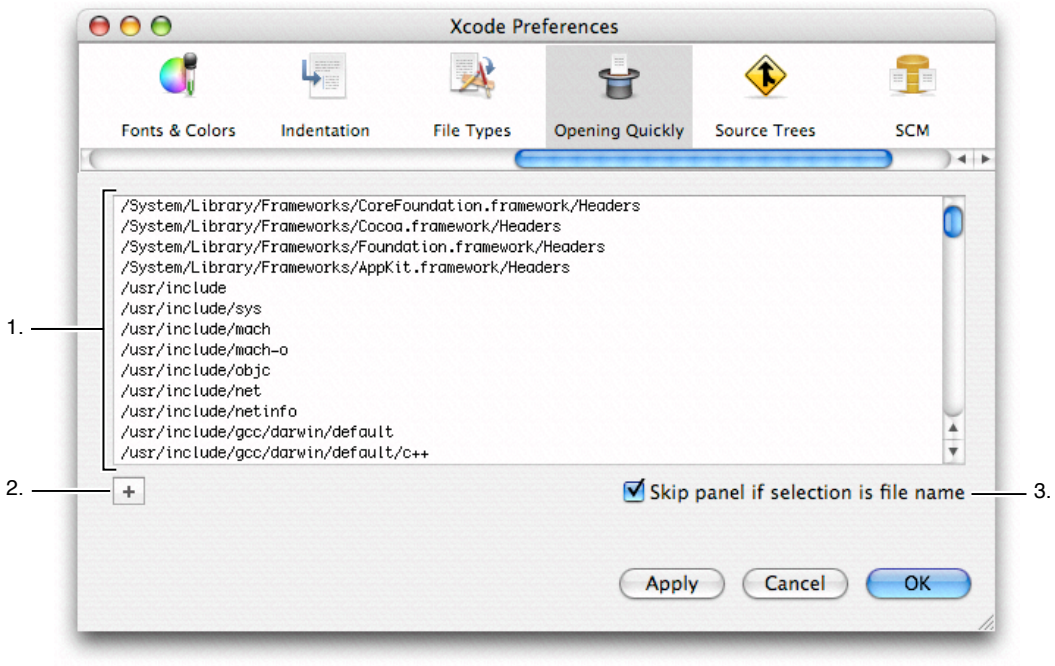
    For file types that Xcode's editor can display, this menu also contains items for choosing how the Xcode editor interprets them. For example, Xcode's editor usually interprets and displays `text.sdef` files as AppleScript dictionaries. However, you can choose Plain Text File from the Preferred Editor menu to force Xcode to display files of this type as plain text files in the editor.

For more information on choosing how to open and display files in Xcode, see "Using an External Editor" (page 193). To learn how to override a file's type, see "Overriding a File's Type" (page 164).

Opening Quickly Preferences

The Opening Quickly pane of the Xcode Preferences window contains the list of directories that Xcode uses to find files with the Open Quickly command. Figure 39-11 shows the Opening Quickly pane of Xcode Preferences.

Figure 39-11 Opening Quickly Preferences



Here is what the pane contains:

1. The list of directories that Xcode uses to find and open files, one directory per line. Paths to a number of common directories—such as `System/Library/Frameworks`—are already included in this list by default. Open Quickly searches the directories in the order in which they appear in this list.

2. The plus (+) button. Click this button to add a directory to this list by navigating to it with the Open File dialog. You can also add a file or directory to the list by dragging it from the Finder.

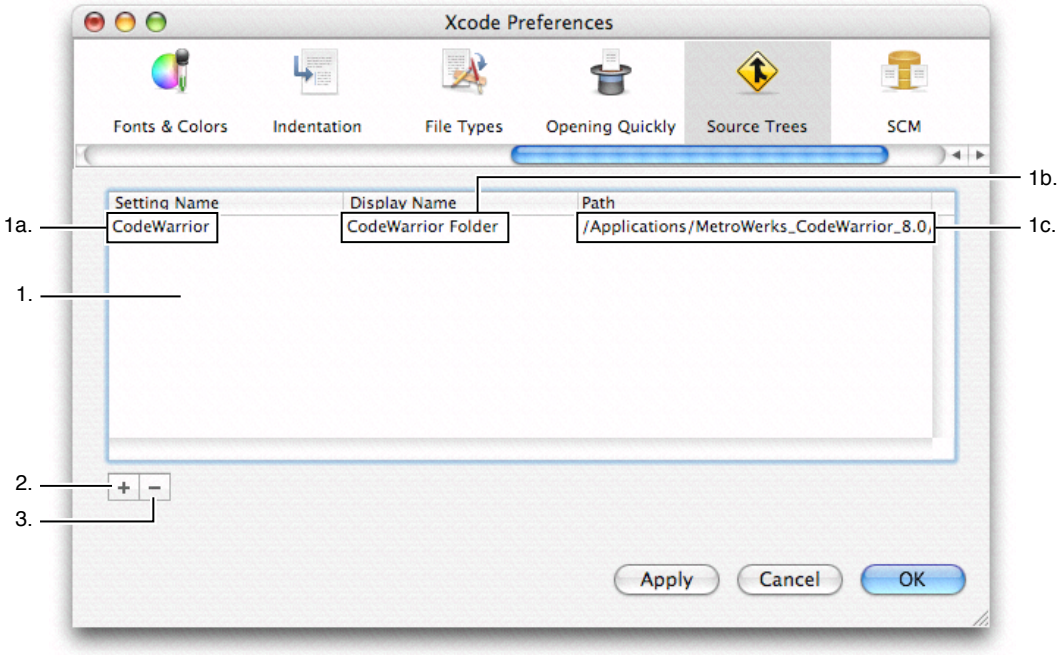
3. Skip panel if selection is file name. This option controls the behavior of the Open Quickly feature when you select a file name in an editor and choose File > Open Quickly. If this option is enabled, Xcode skips the Open Quickly dialog and searches for the selected file name. Otherwise, Xcode displays the Open Quickly dialog, and uses the selected filename to fill out the Path field. This option is enabled by default.

For more information on using the Open Quickly command, see "Opening Files by Name or Path" (page 166).

Source Trees Preferences

The Source Trees pane of the Xcode Preferences window lets you define source trees for all projects that you open. A source tree defines a common access path or output location for the files used or generated by your project. Figure 39-12 shows the Source Trees pane of Xcode Preferences.

Figure 39-12 Source Trees Preferences



Here is what the pane contains:

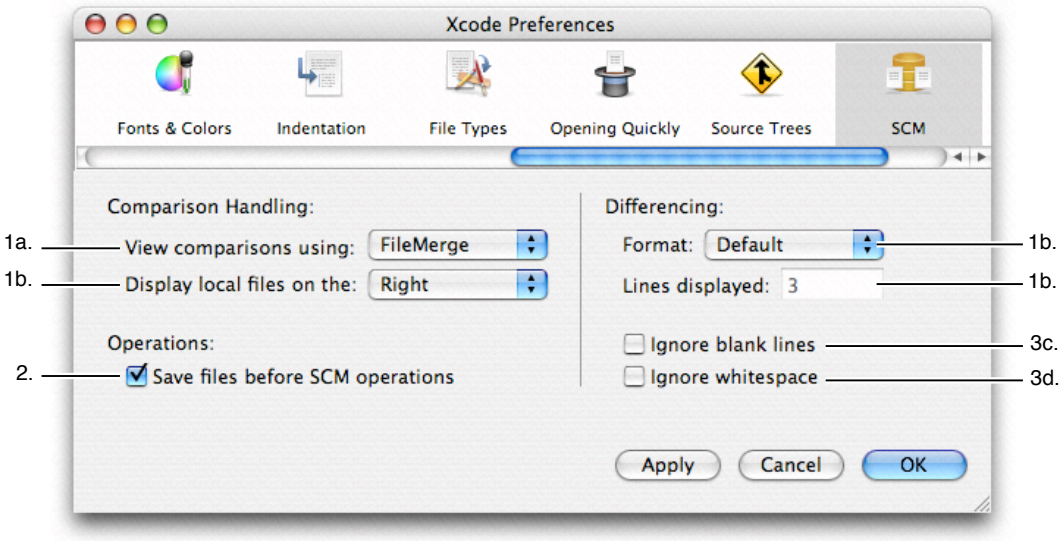
1. Source trees table. This table lists all of the source trees that you have defined. The columns are as follows:
    a. Setting Name is the name used to identify the source tree. This name must be the same for all users accessing items with the source tree.
    b. Display Name is the name that Xcode uses to display the source tree in the user interface. This can be any string you like.
    c. Path is the path to the root directory of the source tree on your local filesystem.
2. Plus (+) button. This button adds a new source tree entry to the table. You must add a new entry before you can edit it. In addition to using this button to add an entry, you can also drag a folder from the Finder to the source trees table.
3. Minus (-) button. This button deletes the selected source tree from the table.

Any source tree in this table is available to any of your projects. To learn more about source trees, see "Source Trees" (page 81).

SCM Preferences

The SCM preferences pane contains options that let you control how Xcode performs version control and file comparison operations. Figure 39-13 shows the SCM pane of Xcode Preferences.

Figure 39-13 SCM Preferences



Here is what the pane contains:

1. Comparison Handling options. These options control how Xcode compares files using the Compare With command, described in "The Compare Command" (page 220). These options are:

    a. View comparisons using. This menu lets you choose the application used to compare files.

    b. The "Display local files on the" menu lets you choose the side of the screen on which you want the local version of the file to appear when comparing file versions.

2. Operations. The "Save files before SCM operations" option specifies whether Xcode automatically saves changed files before performing any version control operations. This option is on by default.

3. Differencing options. These options control how Xcode performs file comparisons using the `diff` command. These options are:

    a. Format. This menu lets you choose the format in which output from the diff command is displayed. For a list of the possible formats, see "Specifying Comparison and Differencing Options" (page 222).

    b. Lines displayed specifies the number of lines of context displayed around each difference.

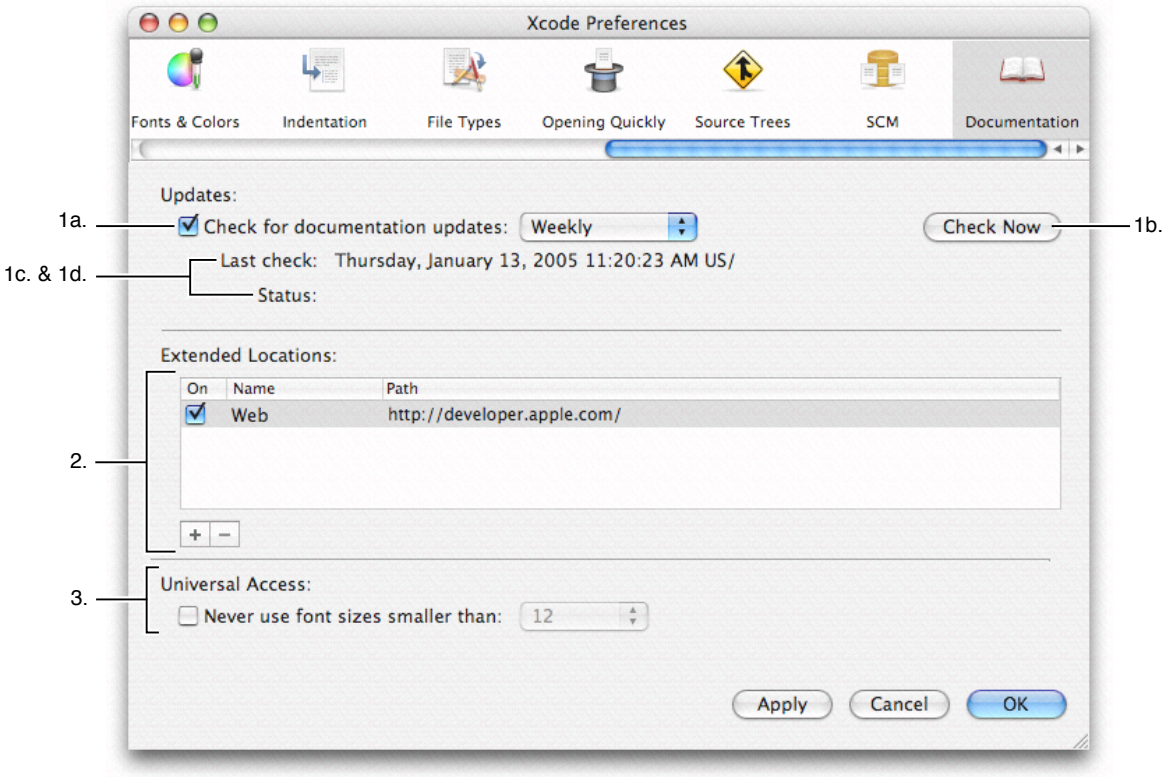
    c. Ignore blank lines indicates whether blank lines are skipped when differencing files.

    d. Ignore whitespace tells Xcode whether to ignore whitespace when differencing files.

Documentation Preferences

The Documentation pane of the Xcode Preferences window controls options for viewing and updating technical documentation with Xcode's documentation window. Figure 39-14 shows the Documentation pane of Xcode Preferences.

Figure 39-14 Documentation Preferences



Here is what the pane contains:

1. Updates. These options control how Xcode checks for updates to the installed developer documentation. These options are:

    a. Check for documentation updates. This option determines whether Xcode automatically checks for updated documentation. If this option is selected, Xcode automatically performs a periodic check for the presence of an updated documentation package on Apple's developer website. Otherwise, Xcode checks only when you tell it to. This option is enabled by default.

    The pop-up menu specifies how often Xcode checks for updates. You can have Xcode check once every day, week, or month. By default, Xcode is set to check on a weekly basis.

    b. Check Now. This button lets you manually check for available updates. Clicking this button causes Xcode to perform the check immediately.

    c. Last check. This displays the date and time at which Xcode last successfully performed a check for updated documentation.

d. Status. This displays the status of the last check.

For more information on obtaining documentation updates, see "Obtaining Documentation Updates" (page 118).

2. Extended Locations. This table lets you specify alternate locations for Xcode to use to locate documentation. To add an entry to this table, you can click the plus (+) button or drag a directory from the Finder. Xcode searches for documentation in these locations from top to bottom. For each location that you add, the Extended Locations table shows the following:

- On. This column shows whether or not the current location is actively being used by Xcode when it looks for documentation. A location is being used if the checkbox in this column is selected. Otherwise, Xcode skips the location when it looks for documentation. When you add a new location, it is on by default.
- Name. This column contains the display name for the location. This name has no effect on how Xcode looks for documentation; it is there to help you identify the location.
- Path. This column contains the full path to the folder containing documentation.

For more information on using extended documentation locations, see "Extended Documentation Locations" (page 111).

3. Universal Access. These options control how Xcode displays documentation in its documentation window. The "Never use font sizes smaller than" option specifies whether Xcode enforces a minimum font size for HTML pages that it displays in the documentation window. When this option is selected, Xcode uses the minimum font size—specified in the pop-up menu—to display all text that uses a smaller font size. This option is disabled by default.

Using Scripts To Customize Xcode

Xcode provides a number of mechanisms for working with scripts and customizing your work environment, including:

- The ability to execute text in a text editor window as a shell command or series of commands
- The automatic execution of a startup script when Xcode is launched
- The automated creation of an extensible User Scripts menu with menu items that execute shell scripts
- A number of built-in script variables and utility scripts you can use in menu scripts or other scripts
- A shell-file build phase that lets you add the execution of shell script files to the build process for a target. Build phases described in "Build Phases" (page 249).

Executing Shell Commands

Xcode provides a keyboard equivalent for executing any shell command that appears in a text editor window. To use this feature, you select the command text and press Control-R. The results appear below the command in the text editor window, autoscrolling if necessary to show the output.

Xcode creates a new shell each time you execute a command, so there is no shared context between different executions. For example, if you execute a command that changes the directory, the next command you execute will not execute in that directory. To overcome this, you can select two commands together and press Control-R.

You might recognize the similarity between this feature and using the Enter key to execute commands in MPW. One way to take advantage of this feature is to keep a file of commonly used commands and execute them as needed. Or you might use an empty text file as a scratch area to type and execute commands.

You can also add custom menu items to execute frequently used shell scripts. Any scripts you execute can take advantage of many special script variables and built-in scripts defined by Xcode. For more information, see "The Startup Script and the User Scripts Menu" (page 415).

The Startup Script and the User Scripts Menu

Xcode provides a number of scripting mechanisms to enhance your productivity and customize your work environment. Xcode can run a startup script every time it is launched. This script can perform whatever custom actions you want to happen on each startup, but is also responsible for creating custom menus in its menu bar, specifically for launching your own specialized scripts. When you launch Xcode, it looks for a script named `StartupScript` and executes it if found. Xcode first looks for a StartupScript file at

`~/Library/Application Support/Apple/Developer Tools/`; if none is found, Xcode falls back to the default StartupScript file installed at `/Library/Application Support/Apple/Developer Tools/`. This script can use any shell, as long as it starts with a standard `#!` comment to identify the shell.

Xcode provides a default startup script and menu definition files that together add a User Scripts menu to the Xcode menu bar. The User Scripts menu is identified by the script icon in the Xcode menu bar, shown Figure 40-1. A **menu definition file** is a special kind of shell script that contains one or more menu definitions and some associated script statements. A **menu definition** uses variables and built-in scripts defined by Xcode to add menus or menu items to the User Scripts menu (or to other menus). Choosing one of the resulting menu items causes the associated script statements to be executed.

The default menu definition files add items to the User Scripts menu that let you open files, perform searches, add comments to your code, sort text, and even add HeaderDoc templates that can help you document your header files. And you can use these files as examples to help write additional menu definitions.

You can customize Xcode's existing StartupScript and User Scripts menu, or you can override them entirely, by placing your own versions at `~/Library/Application Support/Apple/Developer Tools/`.

The following sections describe how Xcode creates the User Scripts menu and provide examples of how you can take advantage of the startup script and the User Scripts menu to customize your Xcode environment. For a full description of the available variables and how they are used in scripts, see "Menu Script Reference" (page 421).

How Xcode Creates the User Scripts Menu

When you install Xcode, the following files are installed in `/Library/Application Support/Apple/Developer Tools/`:

- the default version of the script `StartupScript`; this version is a Perl script that builds the User Scripts menu
- a `Scripts` folder containing a number of menu definition files; these are simply shell scripts that define menu items and the commands to execute when those items are chosen. Menu definition files are interpreted in UTF-8 encoding, which is a strict superset of ASCII, so plain ASCII is fine too

You can override either or both of these default files by placing your own StartupScript file and Scripts directory at `~/Library/Application Support/Apple/Developer Tools`. Note that doing so entirely overrides the default StartupScript or Scripts folder installed with Xcode. You might find it easier to create copies of Xcode's default StartupScript file and Scripts folder and modify these. If you do so, be aware that you will not automatically see bug fixes or improvements made to the default scripts installed at `/Library/Application Support/Apple/Developer Tools/`, as Xcode will load the scripts you have installed at `~/Library/Application Support/Apple/Developer Tools`. To work around this, you can compare the files in the two locations (using `diff` or a similar comparison tool) and update your copies of the scripts as needed.

When you launch Xcode, the following steps take place:

1. Xcode looks for a StartupScript file finds the file `~/Library/Application Support/Apple/Developer Tools/StartupScript` and, if found, executes it. If no StartupScript file is found at the user location, Xcode looks in the default location, `/Library/Application Support/Apple/Developer Tools`. The StartupScript script is invoked with no arguments or input and its output is discarded.

2. The startup script looks for directories in `~/Library/Application Support/Apple/Developer Tools/Scripts`. If no Scripts directory exists at this location, Xcode then looks in the default location, `/Library/Application Support/Apple/Developer Tools/Scripts`. If the script finds any directories in the `Scripts` folder, it creates a new menu corresponding to that directory. If Xcode finds an image file named `menuIcon.tiff` at the top level of the directory, it uses the image as the menu's title in Xcode's menu bar. Otherwise, it uses the textual name of the directory as the menu's title.

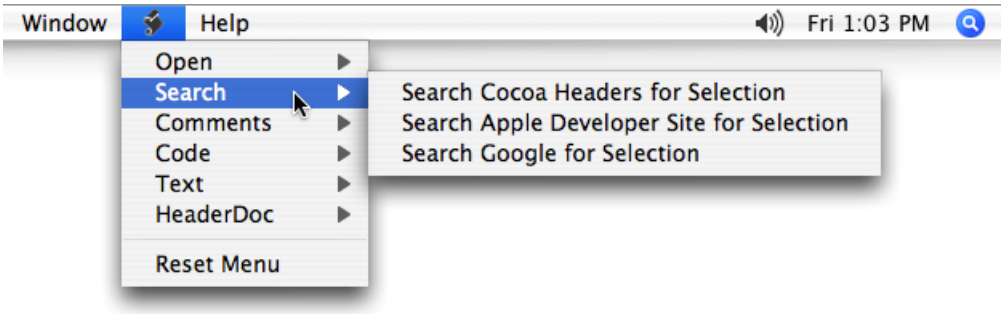
   Take a look inside the `Scripts` directory. Navigate through the `10-User Scripts` directory and see how the scripts are categorized and divided. Those with the extension `.sh` are shell files, while those with the extension `.pl` are Perl files.

3. For each valid menu definition file Xcode finds within each directory, the startup script adds corresponding menu items to the appropriate menu.

   - Files with a numeric prefix in their names are added in ascending order.
   - Files that do not have a prefix are added alphabetically, after files that do have prefixes.
   - Files that have a numeric prefix followed immediately by three dashes are interpreted as a request for a menu separator.
   - The User Scripts menu supports a menu hierarchy. The name of each immediate subdirectory of `~/Library/Application Support/Apple/Developer Tools/Scripts` becomes the name of the corresponding menu. The immediate subdirectories of each of those directories become submenus of the corresponding menu.Menu definition files within the subdirectories are added as commands in the corresponding submenu. You can specify where subdirectory and submenu names appear in the menus by adding numeric prefixes to the filenames.

Figure 40-1 shows the default User Scripts menu that results from the scripts that ship with Xcode. The Search submenu is open, showing several menu items. In this case the menu items do not have key equivalents, though some items in other submenus do have key equivalents.

Figure 40-1 The User Scripts menu



How to Add an Item to the User Scripts Menu

You can easily customize your work environment by adding menus, submenus, and menu items to the menu bar to handle frequently performed operations. To do so, you create a menu definition file, name it appropriately to specify its location in the User Scripts menu (or a menu of your own creation), and place it in the appropriate location in the `Scripts` directory.

For a simple example, the following steps describe how to copy one of the menu definition files from the Sort submenu and use it to add two new menu items to the Sort submenu. That section also describes how the startup script determines where to insert menu items (either into the main User Scripts menu or into a submenu), and how to insert a menu separator.

1. Duplicate the file `10-sort.sh` in `~/Library/Application Support/Apple/Developer Tools/Scripts/10-User Scripts/50-Sort`.
2. Rename the new file `20-reverse_sort.sh`.
3. Open the file in Xcode.
4. Change the line `# %%%{PBXName=Sort Selection}%%%` to `# %%%{PBXName=Reverse Sort Selection}%%%`. This line provides the name for the first new menu item.
5. The five lines starting with `# %%%{PBXNewScript}%%%` actually define a second menu item. Change the line `# %%%{PBXName=Sort File}%%%` to `# %%%{PBXName=Reverse Sort File}%%%`. This line provides the name for the second new menu item.
6. Change the line `sort <&0` to `sort -r <&0`. This tells the sort command to reverse the result of comparison, so that lines with greater key values appear earlier in the output instead of later.
7. Quit Xcode, then launch it again. The items Reverse Sort Selection and Reverse Sort File now appear in the Sort submenu of the User Scripts menu. Choosing the items performs sorts that are reversed from those performed by the original menu items.

The section "Using Variables in a Menu Definition Script" (page 418) takes a closer look at the special variables you worked with in this example.

How to Remove Items From the User Scripts Menu

You can remove items from the User Scripts menu by removing their menu definition files from `~/Library/Application Support/Apple/Developer Tools/Scripts`. If you don't need the User Scripts menu at all, you can either move the entire `Scripts` directory, or move the `StartupScript` file, from `~/Library/Application Support/Apple/Developer Tools`.

Using Variables in a Menu Definition Script

Xcode provides a number of variables you can use in menu definition scripts to get information from, or pass information to, Xcode. These special variables start with `%%%{` and end with `}%%%`. You can use them to specify the menu name, key equivalent, input treatment, output treatment, and arguments for the script. They can also control whether the script's output is displayed incrementally or all at once when the script is finished. (For a full description of the available variables and other options you can use in scripts, see "Menu Script Reference" (page 421).)

Listing 40-1 shows the original menu definition file `10-sort.sh`, which was referred to in "How to Add an Item to the User Scripts Menu" (page 417). The numbered lines in this script are described below.

Listing 40-1 The menu definition file 10-sort.sh

```
#! /bin/sh                                    -1-
```

```
#
# sort.sh - alphabetically sorts the lines of the selection or file
#
# -- PB User Script Info --
# %%%{PBXName=Sort Selection}%%%        -2-
# %%%{PBXInput=Selection}%%%            -3-
# %%%{PBXOutput=ReplaceSelection}%%%    -4-
# %%%{PBXKeyEquivalent=}%%%             -5-
#
# %%%{PBXNewScript}%%%                  -6-
# %%%{PBXName=Sort File}%%%
# %%%{PBXInput=AllText}%%%
# %%%{PBXOutput=ReplaceAllText}%%%
# %%%{PBXKeyEquivalent=}%%%

#                                       -7-
echo -n "%%%{PBXSelection}%%%"
sort <&0
echo -n "%%%{PBXSelection}%%%"
```

1. This line identifies the shell for the menu definition. A menu definition can use any shell, as long as it starts with a standard `#!` comment to identify the shell. As mentioned previously, menu definition files are interpreted in UTF-8 encoding, which is a strict superset of ASCII, so plain ASCII is fine too.

2. The built-in variable in this line (`%%%{PBXName=Sort Selection}%%%`) is the first of four built-in variables the script uses in its first menu definition. The variable specifies the name of the menu or menu item; if you don't specify a name, the name of the menu file is used.

3. `%%%{PBXName=Selection}%%%` specifies that the script for this menu item should take its input from the current selection; if you don't specify an input location, the script gets no input. The possible input options are described in "Specifying Where to Get Input" (page 421).

4. `%%%{PBXOutput=ReplaceSelection}%%%` specifies that the script's output should replace the current selection (that is, the sorted text should replace the original, unsorted text); if you don't specify an output location, the output is discarded. The possible input options are described in "Specifying Where to Place Output" (page 422).

5. `%%%{PBXKeyEquivalent=}%%%` specifies the key equivalent for the menu item; in this case, there is no key equivalent specified. The characters you use to specify key equivalents are listed in "Specifying the Menu Item's Key Equivalent" (page 421).

6. This line, containing the variable `%%%{PBXNewScript}%%%`, starts a second menu definition. It is similar to the first definition, except that it gets all the text of the current document as its input (`%%%{PBXInput=AllText}%%%`) and replaces all the text with its output (`%%%{PBXOutput=ReplaceAllText}%%%`). For related information, see "Placing Multiple Menu Items in One Script" (page 422).

7. Both menu definitions use the same three lines at the bottom of the file to perform the sort operation. The variable `%%%{PBXSelection}%%%` specifies an exact selection within the output. Inserting this variable before and after the sort operation (by echoing it, with `-n` to omit a trailing newline character) results in Xcode selecting the entire output text—that is, selecting the sorted text.

Working With Built-in Utility Scripts

In addition to the variables described in "Using Variables in a Menu Definition Script" (page 418), Xcode provides several useful built-in utility scripts. These scripts can be used in menu definition files or in other scripts you write.

To use one of these scripts, you preface it with the expansion variable `%%%{PBXUtilityScriptsPath}%%%`, which specifies the location of the script. For example, the following statement displays a dialog to get input from the user and places the result in the variable `STRING`. The original text displayed in the dialog is "DefaultString".

```
STRING = `%%%{PBXUtilityScriptsPath}%%%/AskUserForStringDialog "DefaultString"`
```

In addition to the `AskUserForStringDialog` script, Xcode provides built-in scripts to:

- Choose an existing file or folder
- Choose a new file
- Choose an application
- Add a menu item from a menu definition file, or from any script file
- Add a submenu
- Add a menu separator
- Remove a custom menu item or remove all custom menu items from a menu

For details, see "Built in Utility Scripts" (page 424).

Additional Customization With Scripts

As described previously, when you launch Xcode, it looks for a file named `StartupScript` located at `~/Library/Application Support/Apple/Developer Tools/` and executes it if found. Xcode ships with a default `StartupScript` and a number of menu definition files that together create and populate the User Scripts menu.

These simple features clearly provide many options for customizing your Xcode environment:

- You can modify the User Scripts menu by deleting unused menu definition files or adding new ones you write.
- You can modify `StartupScript` to, for example, call additional scripts you write. Or you can replace `StartupScript` completely.
- Xcode provides many built-in script variables and utility scripts you can use in menu scripts or other scripts. You can even add items to other menus or create new custom menus and menu items.
- Many Xcode build settings can be accessed from scripts.
- Your scripts can use Perl or other languages.
- Your shell scripts can execute AppleScript scripts, using the `osascript` command.

Menu Script Reference

The following sections describe the features and terminology you can use in working with menu scripts. Topics covered include menu script definition variable expansion, pre-execution variable expansion, and special user output script markers.

Menu Script Definition Variable Expansion

In addition to the standard script statements in a menu script definition, Xcode parses certain special directives from the file content to allow control over various menu script options. These directives can specify the menu name, key equivalent, input treatment, output treatment, and arguments for the script. They can also control whether the script's output is displayed incrementally or all at once when the script is finished. These special variable expansions start with "%%%{" and end with "}%%%". Any recognized directives will be parsed and deleted from the script text as the file is first being read in (even so, all the example scripts put these directives in shell comments). The following directives are supported.

Specifying the Menu Item's Name

`%%%{PBXName=`*`menu-title`*`}%%%` sets the name of the script currently being defined to *`menu-title`*. If not set, the menu item's name is the file name.

Specifying the Menu Item's Key Equivalent

`%%%{PBXKeyEquivalent=`*`key-equiv`*`}%%%` sets the key equivalent for the menu item to *`key-equiv`*. If not set, the menu item will have no key equivalent. A *`key-equiv`* begins with characters specifying the modifiers and ends with a character that will actually be the key equivalent. Modifier characters are:

- `@` is Command
- `~` is Option
- `^` is Control
- `$` is Shift

Most key equivalents should begin with `@`. Modifier characters are recognized until the first non-modifier character. The rest is the actual key. If the key is also a modifier character, precede it with a `\`. For example `@b` is Command-B, and `@~\@` is Command-Option-@. Remember that a menu script definition file must be Unicode (UTF-8) if it contains non-ASCII characters (such as function keys) as key equivalents. Note that you can insert a function key into a file by pressing Control-Q and then the function key (such as Home, F5, or Page Up).

Specifying Where to Get Input

`%%%{PBXInput=`*`input-treatment`*`}%%%` specifies where the script gets its input for `stdin`. If not set, the script gets no input. The possible values for *`input-treatment`* are:

- `None` uses no input
- `Selection` uses the selected text as input

- `AllText` uses all the text in the window as input

Specifying Where to Place Output

`%%%{PBXOutput=`*`output-treatment`*`}%%%` specifies where to send the output from the script. If not set, the output is discarded. The possible values for *`output-treatment`* are:

- `Discard` discards any output
- `ReplaceSelection` replaces the current selected text with the output
- `ReplaceAllText` replaces the complete text with the output
- `InsertAfterSelection` inserts the output after the currently selected text
- `AppendToAllText` appends output to the end of the complete text
- `SeparateWindow` shows output in a separate window (currently an alert panel)
- `Pasteboard` puts the output on the pasteboard

Specifying Script Arguments

`%%%{PBXArgument=`*`arg`*`}%%%` specifies an argument to pass to the script. You can use this expansion zero or more times, each one adds a new argument to pass to the script.

Specifying How to Display the Menu Item's Output

`%%%{PBXIncrementalDisplay=`*`flag`*`}%%%` specifies whether to display the script's output as it arrives. If *`flag`* is `YES` and the output goes to the text view (that is, not to a separate window or the clipboard), then the output is displayed as it arrives. If *`flag`* is `NO`, the output is displayed after the script finishes. The default is `NO`.

Placing Multiple Menu Items in One Script

`%%%{PBXNewScript}%%%` signals the beginning of a new script. Use this directive if you want to define more than one menu item, each with its own settings, in a single script. When this directive is encountered, all the previous directives are considered complete, a menu command is created, and everything is reset to start specifying settings for a new menu item.

Pre-Execution Script Variable Expansion

Menu Scripts can also contain a variety of special variables that will be expanded by Xcode each time the script is executed. Several variables are supported.

Getting Text From the Active Window

These variables are replaced by text in the active window:

- `%%%{PBXSelectedText}%%$` is replaced by the selected text in the active text object.

- `%%%{PBXAllText}%%%` is replaced by the entire text in the active text object.

The text is expanded verbatim with no quoting. In most shells this would be fairly dangerous because the selection might include single or double quotes or any number of other special shell characters. One safe way to use this in a Bourne shell script, for example, is to have it expand within "here-doc" style input redirection like so:

```
cat << EOFEOFEOF
%%%{PBXSelectedText}%%%
EOFEOFEOF
```

The above script would simply print the selected text to the standard output.

Getting Information on the Active Window's Contents

These variables are replaced by information on the text in the active window:

- `%%%{PBXTextLength}%%%` is replaced by the number of characters in the active text object.
- `%%%{PBXSelectionStart}%%%` is replaced by the index of the first character in the selection in the active text object.
- `%%%{PBXSelectionEnd}%%%` is replaced by the index of the first character after the selection in the active text object.
- `%%%{PBXSelectionLength}%%%` is replaced by the number of characters in the current selection in the active text object.

Getting the Pathname for the File in the Active Window

`%%%{PBXFilePath}%%%` is replaced by the path to the file for the active text object, if it can be determined. This may not be accurate. Xcode tries to find the file path first by walking up the responder chain looking for an NSWindowController that has an NSDocument. If it finds one it will use the document's `fileName`. If it does not find one, it will use the `representedFilename` of the window, if it has one.

Note that this implies that sometimes this will expand to nothing and sometimes it may expand to a file name that is not really a text file containing the text of the active text object. In Xcode text file editors, this works correctly, in other text areas in Xcode (like the build log or any text field) it will not do anything reasonable.

Getting the Pathname for the Utility Scripts

`%%%{PBXUtilityScriptsPath}%%%` is replaced by the path to the folder that contains a number of built in utility scripts and commands that can be used from user scripts to provide functionality such as using a dialog to ask the user for a string or to ask the user to choose a folder or file, or to add to the menu bar of the host application. See below for descriptions of the available utility scripts.

Special User Script Output Markers

When a User Script is done executing, Xcode scans the output for certain special markers. Currently only one marker is supported.

`%%%{PBXSelection}%%%` specifies an exact selection within the output. By default Xcode will set the selection to be an insertion point after all the newly inserted output text. But if the output contains one or two instances of this special marker, it will use them to determine the selection. If there is one such marker, it identifies an insertion point selection. If there are two, all the text between them is selected. The markers are removed from the output.

Built in Utility Scripts

Xcode provides several useful utility scripts that are built-in to Xcode itself. These scripts can be used in menu definition file scripts or in MPW-style worksheet content. To use one of these scripts, use the `%%%{PBXUtilityScriptsPath}%%%` expansion variables. For an example, see "Working With Built-in Utility Scripts" (page 420).

Specifying a String

`AskUserForStringDialog [`*`default-string`*`]`

Displays a dialog in the active application and returns the string that the user enters. If supplied, *`default-string`* is the initial contents of the text field.

Choosing an Existing File or Folder

`AskUserForExistingFileDialog [`*`prompt-string`*`] AskUserForExistingFolderDialog [`*`prompt-string`*`]`

Displays a standard open dialog and returns the path of the file or folder that the user chooses. If supplied, *`prompt-string`* is the prompt in the dialog. Otherwise a default prompt is used.

Choosing a New File

`AskUserForNewFileDialog [`*`prompt-string`* `[`*`default-name`*`]]`

Displays a standard save dialog and returns the path of the new file. If supplied, *`prompt-string`* is the prompt in the dialog. Otherwise, a default prompt is used. If supplied, *`default-name`* is the default name for the new file

Choosing an Application

`AskUserForApplicationDialog [`*`title-string`* `[`*`prompt-string`*`]]`

Displays an application picker dialog and returns path of the application the user chose. If supplied, *`title-string`* is the title for the dialog. Otherwise, a default title is used. If supplied, *`prompt-string`* is the prompt in the dialog. Otherwise, a default prompt is used.

Adding a Menu Item From Any Script File

`SetMenu add script` *`script menu-title key-equiv input-treatment output-treatment index`* `[`*`menu-path`* `...]`

Adds a new menu item to an existing menu in Xcode. The menu item has the name *menu-title*, and the key equivalent *key-equiv*. (Use " " for no key equivalent.) When the user chooses this command, it invokes the script *script*, getting its input from *input-treatment* and placing its output in *output-treatment*. The new item is inserted at *index* in the menu identified by menu-path. If you don't specify *menu-path* the item appears in the main menu bar. *menu-path* contains the titles of menus and submenus that lead to the desired menu.

index is a zero-based index starting at the end of all the original items in the menu. For example, the index `0` in the File menu would generally be the first item after Print (usually the last item in the File menu of an application.) Index `2` would be after the second custom item in a menu. Use negative indices to count from the end of a menu. Index `-1` means at the end, and Index `-2` means right before the last item.

The *key-equiv*, *input-treatment*, and *output-treatment* arguments use the same syntax as the values of the menu definition file directives `PBXKeyEquivalent`, `PBXInput`, and `PBXOutput` respectively. For example, if *input-treatment* is `Selection`, the selected text is the input for the new menu item's script.

This is the most complicated form of the `SetMenu` command. Usually it is better to use the next form in conjunction with menu script definition files.

Adding a Menu Item From a Menu Definition Script

```
SetMenu add scriptfile script-path index [menu-path ...]
```

Adds new menu items to an existing menu in Xcode. The items are read from *script-path*. See the menu script definition files notes above for details on the file format. Details such as the menu titles key equivalents, and input and output treatment are defined within the file. The new items are inserted at the given *index* in the menu specified by *menu-path*. If you don't specify *menu-path* the item appears in the main menu bar. *menu-path* contains the titles of menus and submenus that lead to the desired menu.

index is a zero-based index starting at the end of all the original items in the menu. For example, the index `0` in the File menu would generally be the first item after the Print (usually the last item in the File menu of an application.) Index `2` would be after the second custom item in a menu. Use negative indices to count from the end of a menu. Index `-1` means at the end, and Index `-2` means right before the last item

Adding a Submenu

```
SetMenu add submenu submenu-name index [menu-path ...]
```

Adds a new submenu to an existing menu in Xcode. The submenu's title is *submenu-name*. Initially, it has no items. The new submenu is inserted at *index* in the menu specified by the *menu-path*. If you don't specify *menu-path* the item appears in the main menu bar. *menu-path* contains the titles of menus and submenus that lead to the desired menu.

index is a zero-based index starting at the end of all the original items in the menu. For example, the index `0` in the File menu would generally be the first item after the Print (usually the last item in the File menu of an application.) Index `2` would be after the second custom item in a menu. Use negative indices to count from the end of a menu. Index `-1` means at the end, and Index `-2` means right before the last item

Adding a Menu Separator

```
SetMenu add separator index [menu-path ...]
```

Adds a new separator to an existing menu in Xcode. The new separator is inserted at *index* in the menu specified by the *menu-path*. If you don't specify *menu-path* the item appears in the main menu bar. *menu-path* contains the titles of menus and submenus that lead to the desired menu.

index is a zero-based index starting at the end of all the original items in the menu. For example, the index `0` in the File menu would generally be the first item after the Print (usually the last item in the File menu of an application.) Index `2` would be after the second custom item in a menu. Use negative indices to count from the end of a menu. Index `-1` means at the end, and Index `-2` means right before the last item

Removing a Custom Menu Item

```
SetMenu remove item index [menu-path ...]
```

Removes a custom item from an existing menu in Xcode. The custom item at *index* in the menu specified by the *menu-path*. If you don't specify *menu-path* the item is removed from the main menu bar. *menu-path* contains the titles of menus and submenus that lead to the desired menu.

index is a zero-based index starting at the end of all the original items in the menu. For example, the index `0` in the File menu would generally be the first item after the Print (usually the last item in the File menu of an application.) Index `2` would be after the second custom item in a menu. Use negative indices to count from the end of a menu. Index `-1` means at the end, and Index `-2` means right before the last item

Only items and submenus added by a `SetMenu` command may be removed by the `SetMenu remove item` command. You cannot remove Xcode's real menu items.

Removing All Custom Menu Items From a Menu

```
SetMenu remove all [menu-path ...]
```

Removes all custom items from an existing menu in Xcode. All custom items in the menu identified by the given *menu-path* are removed. If you don't specify *menu-path* the item is removed from the main menu bar. *menu-path* contains the titles of menus and submenus that lead to the desired menu.

This command applies to items and custom submenus, but does not recurse into original submenus. For example, if you added an item to the File menu and you added a whole submenu called My Scripts to the main menu bar, `SetMenu remove all` removes the My Scripts submenu, but does not remove the custom item in the File menu. `SetMenu remove all File` removes the custom item from the File menu.

Only items and submenus added by a `SetMenu` command can be removed by the `SetMenu remove all` command. You cannot remove Xcode's real menu items.

Using CVS

This appendix provides instructions on performing a few essential version control operations using the CVS client tool. It doesn't offer detailed guidance on using CVS. Consult the CVS documentation for in-depth explanations.

The cvs and ocvs Tools

CVS is installed in Mac OS X. Max OS X version 10.4 includes two versions of CVS: the legacy version (version 1.10), which supports wrappers and the current version (version 1.11.18), which doesn't support wrappers. CVS with wrapper support is required when you work on projects that contain bundles, such as nibs, which are directories that group a set of files as a discrete package (see *Bundle Programming Guide* for details). When you use CVS with wrappers, bundles are compressed into a single file before they are placed in the repository and decompressed when they are checked out to local copies.

In Mac OS X v10.4, the CVS tool that supports wrappers is `/usr/bin/ocvs`, (this is the same version of the tool that shipped on earlier versions of Mac OS X). The current version of the CVS tool is `/usr/bin/cvs`. It incorporates fixes to security problems in earlier versions of CVS. Xcode is initially configured to use `ocvs`.

You must use the `ocvs` tool when working on projects hosted on repositories created using CVS 1.10 or earlier. Also, you must host new projects containing bundles in repositories created using CVS 1.10 or earlier. You should host new projects that don't use bundles in repositories created using CVS version 1.11.18 or later.

Creating a CVS Repository

Most developers don't need to worry about creating or managing repositories. This task is usually handled by system administrators. However, if you're a member of a very small team or the sole programmer in your organization, you may have to create and maintain the repository that holds your company's source code. Or you may create a local repository to experiment with version control in your computer.

To create a CVS repository, create the CVS root directory (the directory that contains the repository) and initialize the repository. If you want others to access the repository, you should create a group containing their user names and assign the group to the root directory.

Creating the cvsusers Group

To create the CVS users group, execute these commands in Terminal:

```
% sudo nicl . -create /groups/cvsusers                    // 1
% sudo nicl . -append /groups/cvsusers gid 600            // 2
```

```
% sudo nicl . -append /groups/cvsusers users <user1> <user2>  ... <userN>      // 3
% lookupd -flushcache                                                          // 4
% memberd -r                                                                  // 5
```

This is what the commands do:

1. Creates the `cvsusers` group.
2. Assigns an ID number to the `cvsusers` group. You can use any unused group ID number.
3. Assigns a list of user names to the `cvsusers` group.
4. Flushes the directory information cache.
5. Resolves the memberships of the new group.

Creating the Root Directory

To create the repository's root directory, execute the following commands:

```
% sudo mkdir /cvsrep                                                          // 1
% sudo chgrp cvsusers /cvsrep                                                 // 2
% sudo chmod g+w /cvsrep                                                      // 3
```

This is what the commands do:

1. Creates the `/cvsrep` directory.
2. Assigns the `cvsusers` group to the directory.
3. Gives write permission to the `cvsusers` group for the directory.

Initializing the Repository

To initialize the repository, execute the following commands:

```
% setenv CVSROOT /cvsrep         # 'export CVSROOT=/cvsrep' in bash
% sudo cvs init
```

Important: Users who want to access your repository must set the `CVSROOT` environment variable to the path to your repository in their login scripts. In addition, every user must execute the `cvs login` command before the user can use the `cvs` tool to access the repository.

Accessing a CVS Repository

You can access local repositories (that is, repositories that reside on the same computer on which you develop) directly. For repositories located in remote computers, however, CVS uses secure connection methods, such as SSH. See the CVS documentation for information on other secure connection methods.

This section shows how to configure environment variables to enable SSH-based access to a CVS repository. For information on how to set up the required private and public keys, see "Configuring Your SSH Environment" (page 435).

To access a CVS repository through SSH, you need to configure the `CVSROOT` and `CVS_RSH` environment variables for your user account:

1. Set `CVSROOT` so that it defines the user under which to log in, the computer the repository resides in, and the path to the repository's root directory, using the following format:

   ```
   :ext:<user>@<host><repository_path>
   ```

   For example:

   ```
   setenv CVSROOT :ext:ming@server.apple.com:/cvsrep
   ```

2. Set `CVS_RSH` to `ssh`:

   ```
   setenv CVS_RSH ssh
   ```

For more information on using SSH to access a remote CVS repository, consult the CVS documentation.

Importing Projects Into a CVS Repository

To add a project directory to a CVS repository, use the `cvs import` command, which operates on the current working directory. You must have the `CVSROOT` environment variable set to the CVS root directory (see "Creating a CVS Repository" (page 427)). The command's syntax is:

```
cvs import -m "<import_comment>" <module_name>  <tag> start
```

For example, to import the project directory `/Developer/Examples/Networking/Echo`, you issue the following commands in Terminal:

```
% cd /Developer/Examples/Networking/Echo
% cvs import -m "Echo added to repository" Echo Echo_1  start
```

Before adding a project directory to a repository, you should move or delete the `build` directory if it resides in the project directory. You should also move or delete any other directories you don't want to add to the repository. Otherwise, changes to files in those directories are tracked by your version control system and added to the repository.

Checking Out Projects From a CVS Repository

To check out a project in a CVS repository, use the `cvs checkout` command. The command's syntax is:

```
cvs checkout <module_name>
```

For example:

```
% cd ~/src
% cvs checkout Echo
```

Updating a Local Project File to the Latest Version in a CVS Repository

When Xcode is unable to open a project because its project file (`project.pbxproj`) is corrupted, you must use the `cvs update` command to get the latest version from the repository. For example, to update the project package in a project named Grape using `cvs`, you execute the `cvs update` command on the project file, as shown here:

```
% cvs update -C Grape.xcode/project.pbxproj
(Locally modified project.pbxproj moved to .#project.pbxproj.1.4)
U Grape.xcode/project.pbxproj
```

To do the same using `ocvs`, execute the following commands:

```
% mv Grape.xcode/project.pbxproj /tmp
% ocvs update Grape.xcode/project.pbxproj
ocvs update: warning: Grape.xcode/project.pbxproj was lost
U Grape.xcode/project.pbxproj
```

Using Subversion

This appendix provides instructions on performing a few essential version control operations using the Subversion client tool. It doesn't offer detailed guidance on using Subversion. Consult the Subversion documentation for in-depth explanations.

Installing the Subversion Software

This section shows how to install the server and client software of the Subversion version control system in a computer using Fink. Fink is an open-source project that simplifies the installation of UNIX software in several platforms, including Mac OS X.

To install Fink on your computer, go to http://fink.sourceforge.net and follow the download instructions.

To install the Subversion server software, execute these commands in Terminal:

```
% sudo apt-get update                                                    // 1
Hit us.dl.sourceforge.net 10.3/release/main Packages
Hit us.dl.sourceforge.net 10.3/release/main Release
...
% sudo apt-get install svn                                               // 2
Reading Package Lists... Done
Building Dependency Tree... Done
...
```

This is what the commands do:

1. The `apt-get update` command updates Fink's list of available packages.
2. The `apt-get install svn` command installs the `svn` package, which contains the Subversion server software.

To install the Subversion client software, execute the following command:

```
% sudo apt-get install svn-client
```

Creating a Subversion Repository

Most developers don't need to worry about creating or managing repositories. This task is usually handled by system administrators. However, if you're a member of a very small team or the sole programmer in your organization, you may have to create and maintain the repository that holds your company's source code. Or you may create a local repository to experiment with version control in your computer.

To create a Subversion repository, create its root directory (the directory that contains the repository) and initialize the repository. If you want others to access the repository, you should to create a group containing their usernames and assign the group to the root directory.

Creating the svnusers Group

To create the Subversion users group, execute these commands in Terminal:

```
% sudo nicl . -create /groups/svnusers                                          // 1
% sudo nicl . -append /groups/svnusers gid 700                                  // 2
% sudo nicl . -append /groups/svnusers users <user1> <user2>  ... <userN>       // 3
% lookupd -flushcache                                                          // 4
% memberd -r                                                                   // 5
```

This is what the commands do:

1. Creates the `svnusers` group.
2. Assigns an ID number to the `svnusers` group. You can use any unused group ID number.
3. Assigns a list of user names to the `svnusers` group.
4. Flushes the directory information cache.
5. Resolves the memberships of the new group.

Creating and Initializing the Root Directory

To create the repository's root directory, execute the following commands:

```
% sudo svnadmin create /svnrep                                                 // 1
% sudo chgrp svnusers /svnrep/db                                               // 2
% sudo chmod -R g+wx /svnrep/db                                                // 3
```

This is what the commands do:

1. Creates and initializes the `/svnrep` repository directory.
2. Assigns the `svnusers` group to the repository's data directory.
3. Gives write and execute permissions to the `svnusers` group for the data directory.

Accessing a Subversion Repository

Subversion uses URLs (Uniform Resource Locators) to identify repositories. Using URLs, you can work with several Subversion repositories at a time. To access a local repository, you use a URL like the following:

```
file://<repository_root>/<project_path>
```

For repositories located on remote computers, Subversion offers a variety of options; one of them is SSH. To access a repository on a remote computer using SSH, use a URL like the following:

```
svn+ssh://<computer_name>/<repository_root>/<project_path>
```

Before you can access a remote repository using SSH, you have to configure your SSH environment. See "Configuring Your SSH Environment" (page 435) for details. Consult the Subversion documentation for information on other access methods.

Importing Projects Into a Subversion Repository

To add a project directory to a Subversion repository, use the `svn import` command. Its syntax is:

```
svn import -m "<import_comment>" <source>  <repository>
```

For example, to import the project directory `/Developer/Examples/Networking/Echo` into a local repository, you issue the following commands in Terminal:

```
% svn import -m "Echo added to repository" /Developer/Examples/Networking/Echo
  file:///svnrep/Echo
Adding         /Developer/Examples/Networking/Echo/EchoContext.c
Adding         /Developer/Examples/Networking/Echo/main.c
...
Committed revision 1.
```

Before adding a project directory to a repository, you should move or delete the `build` directory if it resides in the project directory. You should also move or delete any other directories you don't want to add to the repository. Otherwise, changes to files in those directories are tracked by your version control system and added to the repository.

Checking Out Projects From a Subversion Repository

To check out a project in a Subversion repository, use the `svn checkout` command. Its syntax is:

```
svn checkout <repository> <target>
```

For example:

```
% svn checkout file:///svnrep/Echo ~/src/Echo
A  /Users/ernest/src/Echo/EchoContext.c
A  /Users/ernest/src/Echo/main.c
...
Checked out revision 1.
```

Updating the Project File to the Latest Version in a Subversion Repository

When Xcode is unable to open a project because its project file (`project.pbxproj`) is corrupted, you must use the `svn revert` command to restore the project file to the version you last obtained from the repository. For example, to update the project package in a project named Sketch, you execute the `svn revert` command on the project file, as shown here:

```
% svn revert Sketch.xcode/project.pbxproj
Reverted 'Sketch.xcode/project.pbxproj'
```

Configuring Your SSH Environment

This appendix explains how to configure SSH access from one computer (the server) to another (the client) for a single user. This allows you to connect securely from your workstation to the computer where your repository is located.

1. The server's administrator must create a user account for you on that computer. Make sure you can log in to the server.

```
% ssh ernest@server.apple.com
ernest@server.apple.com's password:
Last login: Thu Sep 30 15:56:52 2004 from xx.xx.xx.xx
Welcome to Darwin!
```

2. If it doesn't already exist, create the `.ssh` directory in your home directory in the server computer.

```
% mkdir ~/.ssh
% exit
```

3. Using the `ssh-keygen` command, create a private and public key pair and store it in your home directory in the client computer:

```
% ssh-keygen -t dsa
Generating public/private dsa key pair.
Enter file in which to save the key (/Volumes/Athene/ernest/.ssh/id_dsa:
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has ben saved in /Volumes/Athene/ernest/.ssh/id_dsa.
Your public key has been saved in /Volumes/Athene/ernest/.ssh/id_dsa.pub.
The key fingerprint is:
##:##:##:##:##:##:##:##:##:##:##:##:##:##:## ernest@work.apple.com
% cd ~/.ssh
% ls
id_dsa    id_dsa.pub    known_hosts
```

4. Using the `scp` command, copy the public key file (`id_dsa.pub`) to your home directory in the server as `authorized_keys` (unless the `authorized_keys` file already exists there):

```
% scp id_dsa.pub ernest@server.apple.com:~/.ssh/authorized_keys
ernest@server.apple.com's password:
id_dsa.pub                                100%  613     1.2MB/s    00:00
```

   If the `authorized_keys` file if it already exists, add your public key to it using a text editor.

5. Ensure you can connect to the server using your passphrase:

```
% ssh ernest@server.apple.com
Enter passphrase for key '/Users/ernest/.ssh/id_dsa':
Last login: Thu Sep 30 16:06:45 2004 from xx.xx.xx.xx
Welcome to Darwin!
```

Document Revision History

This table describes the changes to *Xcode 2.0 User Guide*.

Date	Notes
2006-11-07	Moved document to legacy because 2.0 is not the current version of Xcode.
2005-04-29	TBD
	New document that describes how to develop software with Xcode. Replaces Xcode Help, "Xcode Build System," "Customizing Xcode," "Xcode Source Control Management," and "Xcode Workflow."

Document Revision History