# Xcode Project Management Guide

**Tools & Languages: IDEs**

**2010-07-02**

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

An Xcode 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.

This document introduces the various parts of a project, shows you how to create projects, and describes how to organize the contents of a project. This document also describes the project window, Xcode's interface for performing project management tasks, and shows how to use that interface to find and discover information in Xcode.

You should read this document if you plan on developing software products for iOS OS or Mac OS X. To get the best out of this document, you should be familiar with basic software development concepts, such as object-oriented programming, compilation, and debugging. You should also be familiar with Objective-C, the main programming language used in Apple platforms.

**Software requirements:** This document is written for Xcode 3.2.3 and later.

## Organization of This Document

This document contains the following chapters, which are divided in two parts:

"Part I: Project Organization" (page 13) describes how Xcode projects are organized and how to find information in them.

- "Overview of an Xcode Project" (page 15) describes the contents of an Xcode project and gives an overview of the information required to develop software with Xcode.
- "Creating Projects" (page 21) shows you how to create a project or import CodeWarrior projects and describes the available project templates.
- "Files in Projects" (page 25) discusses the files in a project, describes how Xcode references project files, shows you how to add files, frameworks, and folders to your project, and describes how to use source trees and cross-project references.
- "Searching Files and Projects" (page 37) describes how to use Xcode to find information about your project's contents.
- "Viewing Project Symbols and Classes" (page 47) shows how to find information about the classes defined in a project and its included frameworks.
- "Localizing Files" (page 55) describes the process of internationalizing your product by localizing some of its files.
- "Using the Organizer" (page 57) describes the Organizer and shows how you can use it to organize and work on multiple projects on one window, including non–Xcode projects.

"Part II: Product Development" (page 61) describes how to perform the major development tasks, including static analysis to find bugs in your code early and building your product.

- "Building Products" (page 63) shows how to build products in Xcode and how to take advantage of multiple CPUs during a build. It also describes how to target releases of a platform different from the one for which you're developing your product.
- "Analyzing Code" (page 89) describes how to use the static analyzer to identify and fix code flaws.
- "Defining Executable Environments" (page 101) describes how to view the executables in your project and how to configure an executable environment.
- "Running Programs" (page 109) shows how to run your programs after you've built them in Xcode.

## See Also

These documents provide overview or additional information about developing software products for Apple platforms:

- *A Tour of Xcode* provides a hands-on introduction to the development of software products for Mac OS X.
- *The Objective-C Programming Language* describes the Objective-C programming language and runtime environment.

# Part I: Project Organization

Part I of this document describe how Xcode projects are organized, how to search files and projects, how to use the Organizer to manage multiple projects, and how to localize files to internationalize products.

# Part I: Project Organization

# Overview of an Xcode Project

To carry out the development process, Xcode relies on certain key components. It uses projects to organize these components. 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.

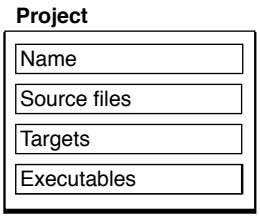
## Components of an Xcode Project

A project contains and organizes everything you need to create one or more software products. In your Xcode project, you:

- Organize build system inputs for building a product.
- Maintain information on items within it 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 1-1 shows a simplified representation of a project and its essential pieces.

**Figure 1-1** The key components of a project




### Source 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 source files—using Xcode's built-in text editor or an external editor—and organize files into a target (described next) 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 Projects" (page 25) describes how Xcode stores these references and discusses the files in a project in more detail.

## 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 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.

## Executables

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 what program to launch when you run or debug from within Xcode and how to launch the program. The executable environment lets you tell Xcode what command-line arguments to pass, what environment variables to set, what 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.

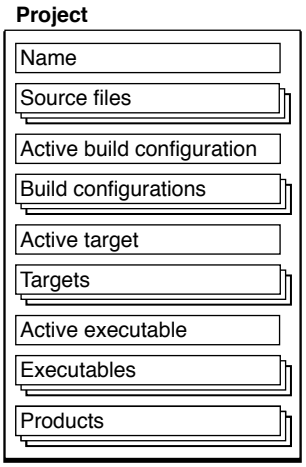
Even if your product generates an executable that can run on its own, you may want to customize the executable environment to specify command-line arguments for Xcode to pass to the program on launch, environment variables to set, and so forth. "Defining Executable Environments" (page 101) describes 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.

## Product Information

In addition to the fundamental building blocks of the development process, an Xcode project also maintains a great deal of information about the items in your project and their current state. Figure 1-2 shows a representation of a project with this additional information.

**Figure 1-2** An Xcode project




An 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, projectwide symbol searching, and more. Groups and other features for organizing project contents are described in Project Organization; projectwide searches and other features for finding information in your project are described in "Searching Files and Projects" (page 37).
- Projectwide 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 click the Build button. It also stores the active architecture, which is the architecture for which all products in the project are built.

The settings that an Xcode project tracks include:

- **Build configurations.** A build configuration is a named collection of build settings. You can define different build configurations for different circumstances—such as development or release—and switch between them to alter how the products in a project are built. In this way, you can rapidly try variations on a build. The project defines the list of build configurations; each individual target contains its own definition of the build settings used to build the target with that configuration. See "Build Configuration Overview" in *Xcode Build System Guide* for more information about build configurations.
- **Targets.** A target serves as the blueprint for building a product. A target mainly identifies the product's source and the operations to perform on them. It also specifies the SDK (the set of header files, libraries, and frameworks) against which the source files are compiled and linked to produce the product's binary file.
- **Active target and build configuration.** The active target is the target that gets built when you initiate a build in Xcode. Xcode uses the active build configuration to select the appropriate configuration of the active target and each target it depends upon, when building. See "Setting Build Factors" (page 75) 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.

A project can have multiple targets and multiple executables. However, there can be only one active target, one active build configuration, one active architecture, 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 the debugger. If you want to debug both applications at once with the graphical debugger, you have to build them in separate projects.

# The Project Directory

When you create a 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 `.xcodeproj`.

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

- **Source files.** 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 subdirectories within the project directory, including directories for localized resources, as described in "Localizing Files" (page 55).

  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 Projects" (page 25).
- **Build folder.** When you build a target, Xcode generates a number of files, including the target's finished product. By default, Xcode creates the build directory in the project directory to hold the files that it creates. The build directory can, however, reside at any location in the file system. For more information on the build directory, see "Build Locations" (page 63).
- **Nonproject files.** Nonproject files are files that reside in the project directory but that are not used to build your product. If you use SCM, these files are part or source-control operations made on the project directory. See "Managing Files Under Source Control" in *Xcode Source Management Guide* for more information.

# The Project Info Window

Xcode maintains information about a project at several levels, including project, target, and file. Information kept at the project level comprises general information, project build settings, project build configurations, and project comments. You use the **Project Info window** to view and edit this information.

To open the Project Info window, do one of the following:

- Double-click the project group in the Groups & Files list.
- Select the project in the Groups & Files list and click the Info button.
- Select the project in the Groups & Files list and choose File > Get Info.
- Choose Project > Edit Project Settings.

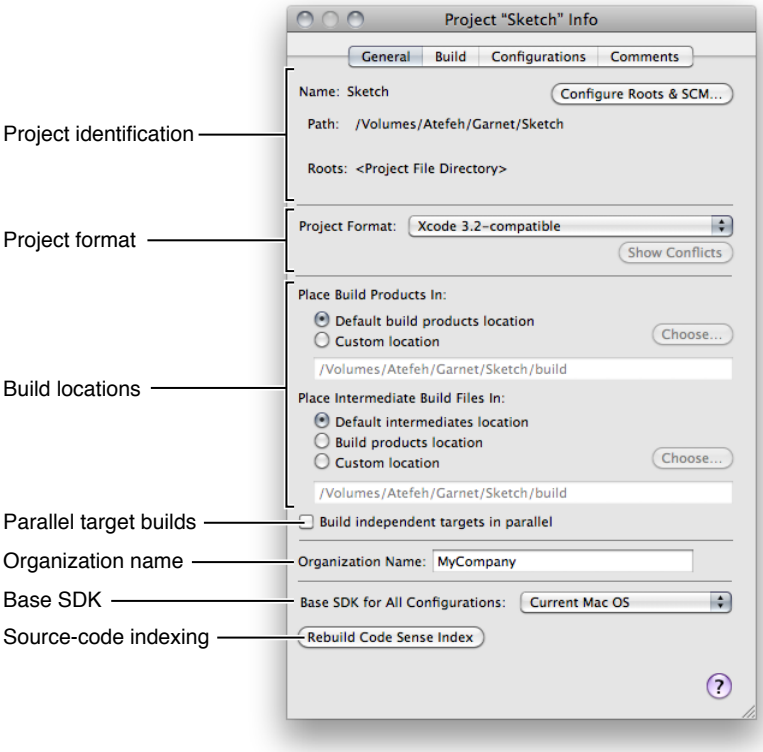
The Project Info window contains the following panes:

- **General.** This pane contains general project settings that affect all the project's files and targets. For details, see "General Project Attributes" (page 19).
- **Build.** This pane lets you define project build configurations. For more information, see Build Settings and "Build Configuration Overview" in *Xcode Build System Guide*.
- **Configurations.** This pane contains a list of build configuration names, which may be used by the project and its targets. Build configurations are described further in "Build Configuration Overview" in *Xcode Build System Guide*.
- **Comments.** This pane lets you add textual notes to the project. See "Adding Comments to Project Items" in *Xcode Workspace Guide* for more information.

## General Project Attributes

Figure 1-3, shows the General pane of the Project Info window.

**Figure 1-3** General pane of the Project Info window




The General pane contains the following information:

- **Project identification.** Specifies the name of the project and its main location. It also identifies the project roots, directories that define the project hierarchy. A **project root** is a directory containing a project package (see "The Project Directory" (page 18)), source files, and other project files. Simple projects have

a single project root. Complex projects, made up of two or more projects that share products or resources, can contain multiple project roots. Each project root has its own SCM configuration, which allows you to work on projects made up of multiple projects with subprojects stored in different SCM repositories.

- **Project format.** Specifies the Xcode release with which the project must remain compatible. See "Choosing the Project Format" (page 23) for details.
- **Build locations.** 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 63) for more information.
- **Parallel target builds.** Specifies whether to build independent targets in parallel. To learn more, see "Building in Parallel" (page 82).
- **Organization name.** Specifies name of the organization to which copyright is attributed in header files created for the project.
- **Base SDK.** Identifies the base SDK used to build the project's targets, which specifies the minimum version of the operating system to build your product for (you can override this setting at the target level). See "Setting Build Factors" (page 75) and "Building for Multiple Releases of an Operating System" (page 85) for more information.
- **Source-code indexing.** Allows you to rebuild the symbol index. See "Symbol Indexing" (page 47) for details.

# Creating Projects

As soon as you know what product you are working on, you need an Xcode project. If the product is new, you can create an Xcode project 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 22).

This chapter shows you how to create a project and describes the available project templates.

## 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 (Objective-C, Objective-C++, or C) you plan to use. You also decide which Apple technologies and frameworks to use.

After you've resolved these issues, you'll find that Xcode provides a wide variety of project templates to support your goals. The New Project dialog groups these templates under several product-type groups, such as applications, frameworks, Automator actions, and so on. Two additional templates create an empty project and a project that uses an external build system.

**iOS:** For information about the iOS project templates, see About iPhone OS Development and "Creating an iPhone Project" in *iOS Development Guide*.

**Mac OS X:** To learn about the Mac OS X products you can create, see Software Development Overview.

The project template you choose specifies a default target and 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.

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.

**Importing projects:** Xcode doesn't import Project Builder and ProjectBuilderWO projects. For these projects, you must create an Xcode project and add the older project's files to it.

# Specifying the Project Name

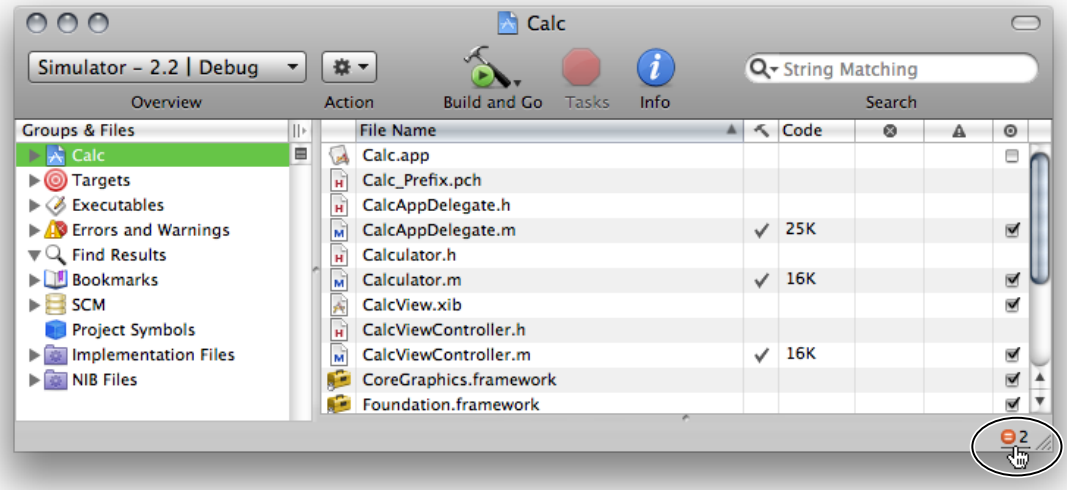
The name of your project may consist of uppercase and lowercase letters (a–z), numbers (0–9), dashes (-), and underscores (_). See *Uniform Type Identifiers Overview* for more information.

# Opening and Closing Projects

To open any project, choose File > Open, navigate to the project directory, and choose the `.xcodeproj` file package you want to open. To open a project you've recently used, choose the project from the Recent Projects submenu in the File menu.

Xcode 3.2 can open projects that use the Xcode 2.4 and later project formats. If the opened project uses features that are not supported in Xcode 3.2, the project window displays an incompatibility notification in the project window status bar, as shown in Figure 2-1. To see details about the incompatibility, click the notification. See "Viewing Project Format Conflicts" (page 23) for more information.

**Figure 2-1** Status bar notification of project format conflicts




To close a project, close the project window.

# Choosing the Project Format

You may work in a team in which some team members may need to use an Xcode release different from the one you use; for example, some developers may need to use Xcode 2.4 while you may want to use Xcode 3.2 to take advantage of features that are not available in earlier releases. If you use Xcode 3.2 to work on the same projects that an Xcode 2.4 user also works on, you have to make sure that you don't use Xcode 3.2 features on those projects; otherwise, your Xcode 2.4–using colleague may have trouble working on or even opening the projects.

To help keep shared projects usable by developers using different Xcode releases, Xcode allows you to specify the release with which a project must remain compatible. This feature is based on project formats. A **project format** tells Xcode how to store the project configuration into the project file inside the project package (see "The Project Directory" (page 18) for details). A **project configuration** is the set of development features, project attributes, and project and target build settings used in a project.

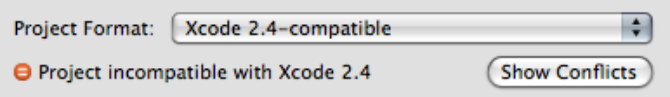
Project formats allow you to specify an Xcode release with which a project must be compatible. This feature lets you use a later release of Xcode to work on a project created with an earlier release while ensuring that the project remains compatible with the earlier release.

**Note:** Another approach to working on a development team that must use multiple Xcode releases is to install multiple Xcode releases on your computer. For example, you can have Xcode 2.4 and Xcode 3.2 on your computer, allowing you to work on Xcode 2.4–based projects and Xcode 3.2–based projects side by side. See *Xcode Installation Guide* for more information.

The rest of this section describes how to choose a project format for a project and how to view and resolve conflicts that arise between the project configuration and the chosen format.

You specify the project format for a project in the General pane of the Project Info window (Figure 1-3 (page 19)). The **Project Format menu** lists the project formats Xcode supports. When you choose a project format that doesn't support the project configuration (for example, when a build setting isn't supported by the chosen format), Xcode notifies you of the incompatibility as shown in Figure 2-2.

**Figure 2-2** Incompatibility between project format and project configuration



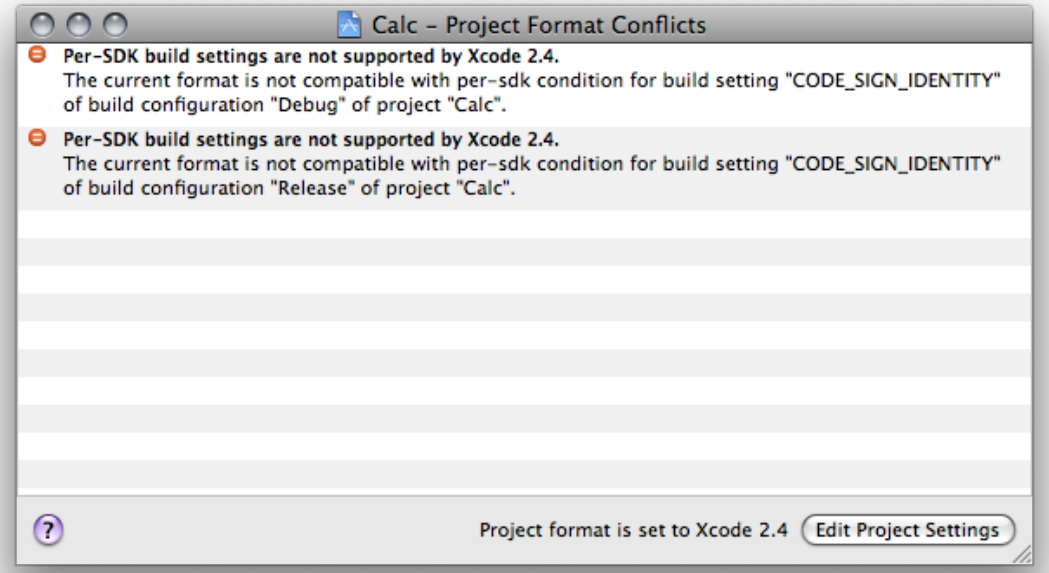

To view the details of the incompatibility, click the Show Conflicts button (this button is available only when there's at least one conflict). See "Viewing Project Format Conflicts" (page 23) for more information.

**Important:** When using cross-project references, ensure that the referenced projects use the same project format as the referencing project. If the project formats differ, you may encounter build errors.

## Viewing Project Format Conflicts

When a project uses Xcode features that are not supported by the chosen project format, Xcode displays the unsupported features (or conflicts) in the **Project Format Conflicts window**, shown in Figure 2-3.

**Figure 2-3** The Project Format Conflicts window




This window appears when you click the Show Conflicts button in the Project Info window or the project-conflict notification in the status bar (see "Choosing the Project Format" (page 23) for details). You have two options for solving these conflicts:

- Change the project format to one that supports the feature that's causing the conflict.

  Other developers working on the same project must use a release of Xcode that supports the same project format.
- Change the project configuration so that it doesn't use the feature that's causing the conflict.

# Files in Projects

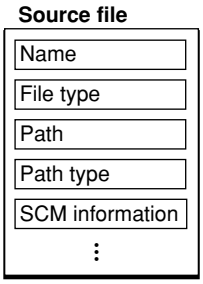
The files in a project are the fundamental building blocks from which you create your end product. Files contain the source code that you write and serve as the inputs to the build system for creating a product. They can also hold notes, performance metrics, and the like to aid you in the development 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 3-1 shows the information that Xcode tracks for source files.

**Figure 3-1** A source file




In the illustration, 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 29) describes the various ways in which Xcode stores paths.

You can view and edit these file attributes in the File Info window.

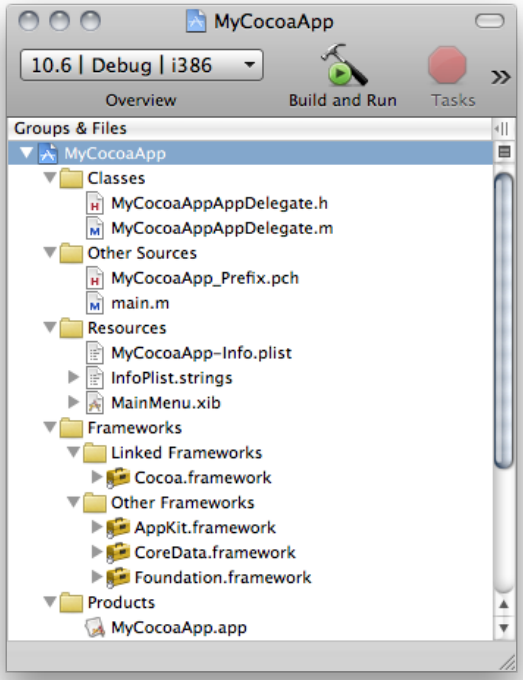
# The Files in a Project

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:

- **Source 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 want to edit using the Xcode text 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 project templates, described in "Choosing a Project Template" (page 21), Xcode populates the project with a small set of default files required to build the associated product. For example, Figure 3-2 shows the contents of a project created using the Cocoa Application project template. The project contents have been expanded in the Groups & Files list to display its contents in outline view. Keep in mind that the contents of a project vary depending on the project template and the products it creates.

**Figure 3-2** The project contents in the Groups & Files list




The example project contains the following items:

- The Classes and Other Sources groups contain source-code files.
- 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 Frameworks 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 special type of file reference, called a **product reference**, 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.

To learn more about the Groups & Files list, see "Project Window Components" in *Xcode Workspace Guide*.

Xcode keeps a reference to each file, folder, and framework you add to your project. Through these references Xcode finds your files directly when it builds a product. 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 Paths build settings.

# Managing Files and Folders in a Project

If you created a project using one of the project templates or if you converted an existing project, your project already has a number of groups with files, folders, and product references. At some point you may need to add or remove files or folders from a project. The following sections show how to accomplish these tasks.

## 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.

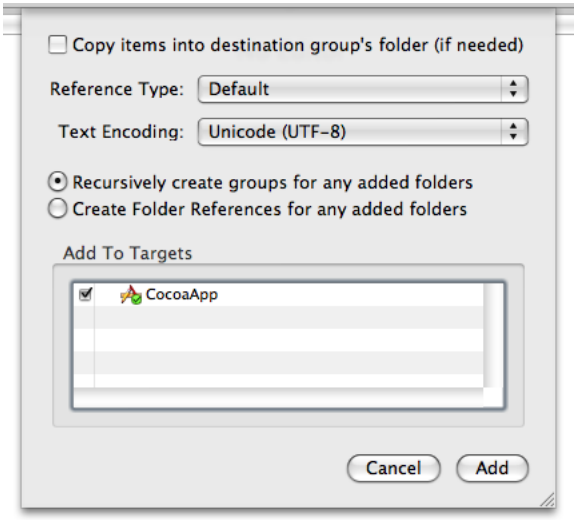
  Use the resulting dialog to navigate to and choose the file or files to add. If you want to add all of the files in a given folder, you may simply choose the folder. The files or folders you add are placed after the items currently selected in the Groups & Files list, if any.
- Drag the icons for the 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 choosing Project > Add Current File to Project

The editor window must have focus.

After you have selected the file or files to add to the project, using either of the two methods described earlier, Xcode displays a dialog, shown in Figure 3-3, that allows you to specify how the files are added to the project.

**Figure 3-3** Specifying how to add files to a project




Here is what the dialog contains:

- 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.
- The Reference Type menu specifies how the location of the file is stored. For a description of the reference styles available to you, see "How Files Are Referenced" (page 29). Note that this menu does not contain any source paths until you have defined one or more source trees in Source Trees preferences (for source-tree details, see "Source Trees" (page 35)). After 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.
- 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" in *Xcode Workspace Guide*.
- 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, as a group or by folder reference.

- **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."
- **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."

## 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 Info window. 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.

## Removing Files

You can remove any files or folders 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 may display a dialog asking whether you want to move the files to the Trash or just delete 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 in your file system. If you choose Also Move to Trash, Xcode deletes the references from the project and moves the referenced files to the Trash.

# Renaming a Project

When you create an Xcode project the name you choose for it is replicated in several files within the project directory, including the project package (see "The Project Directory" (page 18) for information about the project package), source files (such as application-delegate source files in Cocoa-application projects), info-plist files, and the built product. Other than having the project name within their names, the files do not have a structural relationship with each other. That is, changing the name of the project package does not result in a change to the name of built product. Therefore, renaming a project in a satisfactory way requires a set of very focused changes. Xcode's project-rename command changes the name of a project and attempts to effect that change in the appropriate build settings, targets, and files.

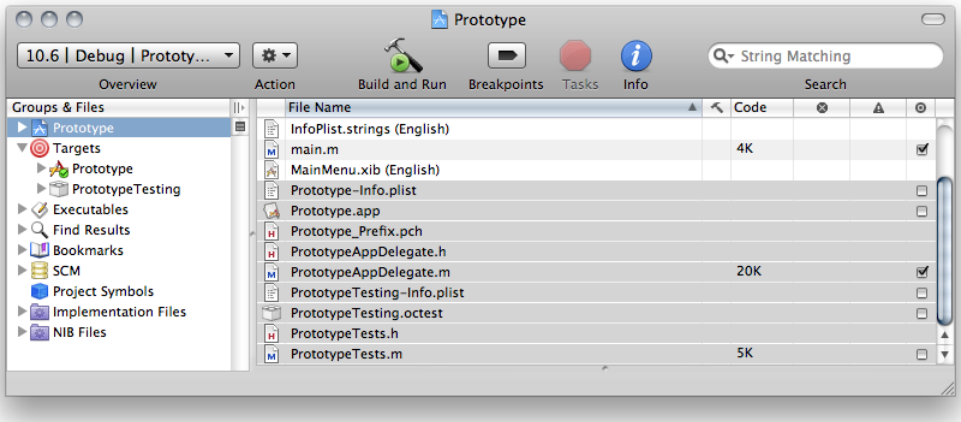
**Software requirements:** The project-rename command is available in Xcode 3.2 and later.

To rename a project, choose Project > Rename.

When you rename a project, Xcode renames all targets that contain the old name. As part of renaming a target, Xcode renames prefix, info-plist, and product files.

Figure 3-4 shows a project named Prototype with two targets containing that name. The files that also contain "Prototype" in their name are highlighted in the detail view.

**Figure 3-4** Project to be renamed




When you execute the project-rename command, Xcode displays the project-rename dialog, shown in Figure 3-5. This dialog lists the items over which the rename operation takes place; you can deselect items you don't want renamed. The dialog also lets you indicate whether to create a snapshot before renaming the project. You can later examine the snapshot to get a list of the files Xcode modified as part of the operation. For more on snapshots, see "Snapshots".

**Figure 3-5** The project-rename dialog

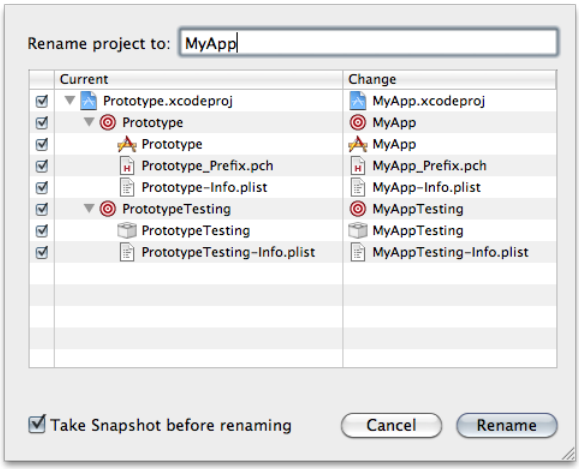
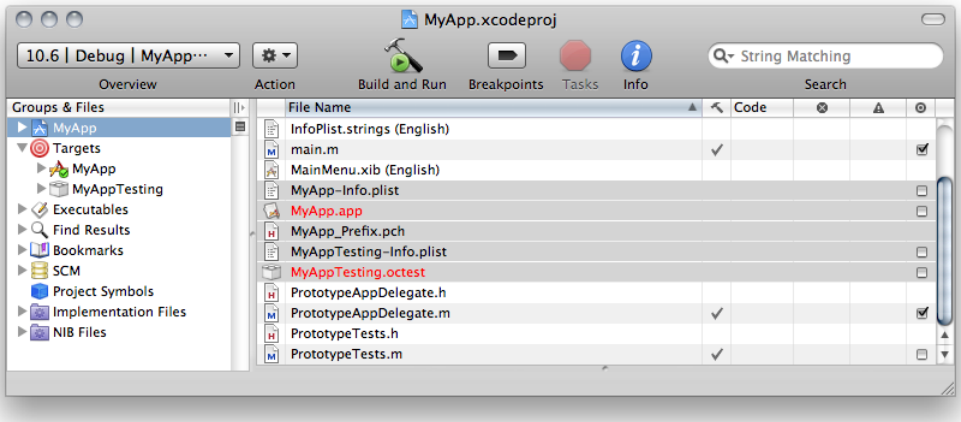
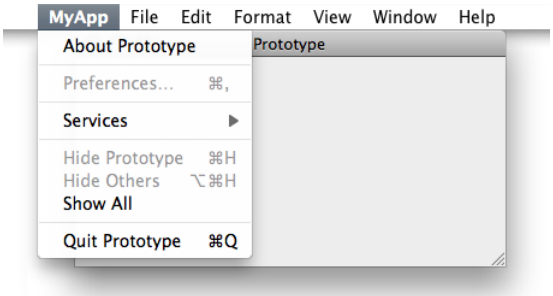



Figure 3-5 shows the project after Xcode completes the rename operation. Note that not all the files with "Prototype" in their name are renamed.

**Figure 3-6** Project on which a project-rename operation has been performed




After Xcode completes the project-rename operation, you should review all aspects of your project to ensure that the operation was completed as expected and to complete the task in places that Xcode did not modify but that should be updated, such as user-interface elements.




Keep in mind that project templates use build settings to propagate the value of properties, such as the product name, throughout the project. For example, an application's info-plist file uses the value of the Product Name build setting to define the application's bundle-name and bundle-identifier properties. When the info-plist file contains template-based definitions for these properties, changing the value of the Product Name build setting updates the values of those properties accordingly. However, if the properties have been customized by removing `PRODUCT_NAME` from their definitions, renaming a project does not change the value of those properties.

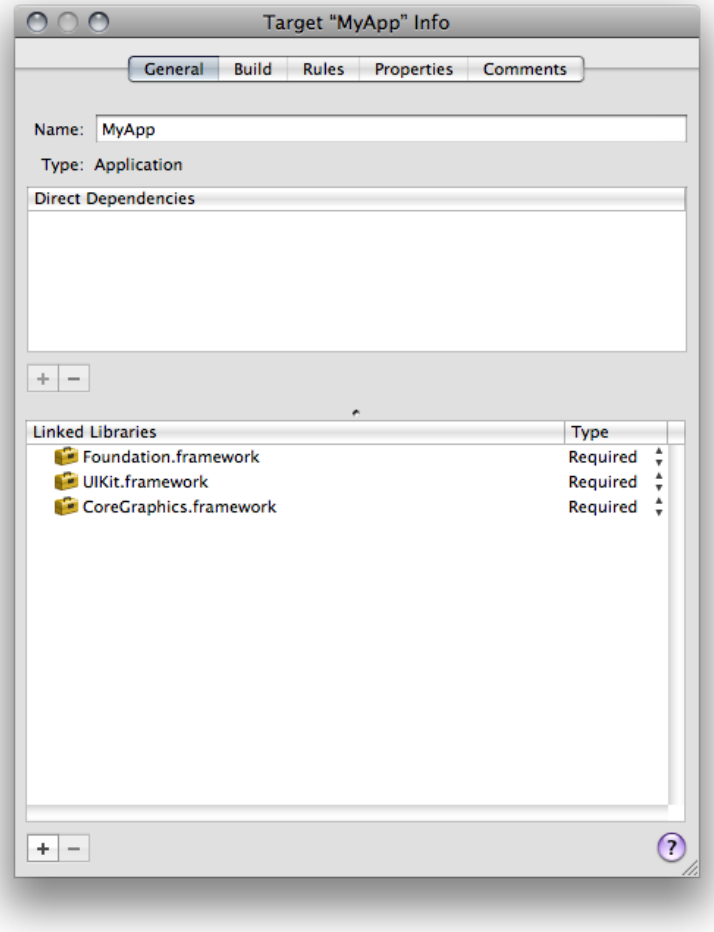
# Linking Libraries and Frameworks

Each of your project's targets specifies the frameworks and libraries against which its source files are linked. You can link against standard libraries or external libraries. Standard libraries or frameworks are those provided by the active SDK. External libraries or frameworks are those not provided by the active SDK.

To add a library to a target:

1. Open the Info window for the appropriate target and click General.

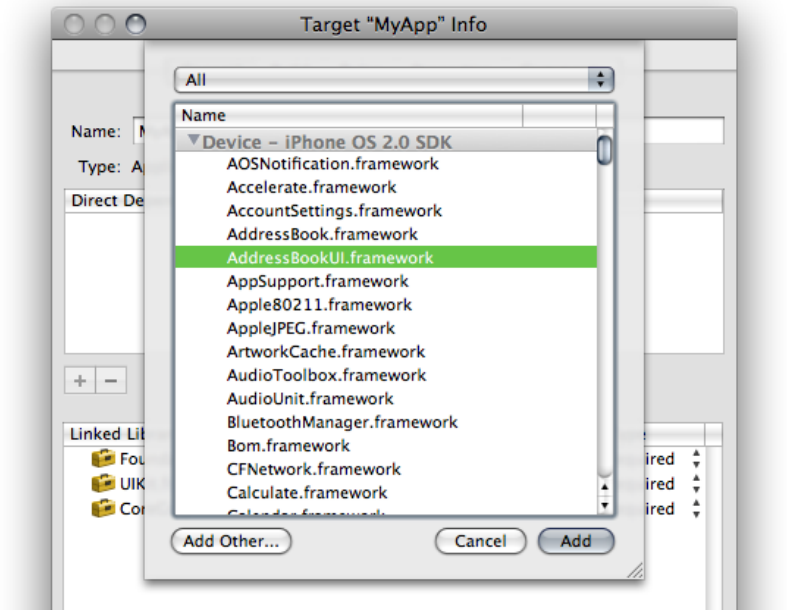
   The Linked Libraries list in the General pane lists the libraries against which Xcode links the target's source files.




2. Click the Add (+) button below the Linked Libraries list.

3. Choose the library to add.

To add a standard library, choose the library from the dialog.




**Note:** The dialog displays libraries and frameworks that belong to the target's Base SDK build setting, which is normally defined at the project level in the General pane of the Project Info window.

To add libraries from another SDK, first change the Base SDK build setting to the one that contains the desired library.

To add an external library, click Add Other.

After choosing an external library, Xcode presents the same options described in "Adding Files and Folders" (page 28). The following options apply to external libraries:

- 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 29).
- The Add to Targets group box allows you to add the library to one or more of the targets currently defined in your project. If the checkbox next to a target name is checked, the library 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 a framework. For more information on file encodings, see "Choosing File Encodings" in *Xcode Workspace Guide*.

**Important:** When using external libraries, you must ensure Xcode can locate the appropriate headers using the Header Search Paths build setting. See "Finding Header Files" (page 78) for details.

4. Choose whether the library is required or optional.

Required libraries must be present on the host computer for the product to load. Optional or weak-linked libraries may or may not be present for the product to load; however, before accessing any of the library's symbols at runtime, you must ensure that the optional library is present on the host.

In addition to the Target Info window, you can use the detail view to view the libraries your product links against:

1. In the Groups & Files list, reveal the target's build phases.
2. Select the Link Binary With Libraries build phase.

   The detail view shows the libraries your product uses. The Role column indicates whether the library is required or optional at runtime.

# Source Trees

A **source tree** is a root path that can be used to define a common location 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 users who have the same source tree defined are able to work on the same project seamlessly, provided that the file also exists at the source tree location on their computers.

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; even though 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.

The source trees that you define are available to all your projects; that is, Xcode supports CodeWarrior-style global source trees. Because source trees are stored for each user, 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. After 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 Info window, described in "Viewing File Information" in *Xcode Workspace Guide*.

You can edit source trees in Source Trees 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, Info windows, 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 Info window.
- **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 new 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 package (the `.xcodeproj` file package) of the project you wish to reference. (You may also drag the project group in the Groups & Files list of another project or a project package in the Finder to the current project's project group.) Xcode adds a reference to the source group for your current project, visible in the Groups & Files list. The project reference is identified by the Xcode 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 Target Dependencies" in *Xcode Build System Guide* 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 63).

# The Xcode Cache

Xcode places its persistent caches in a secure location in your file system. You can get to this location using the `getconf` command:

```
> cd `getconf DARWIN_USER_CACHE_DIR`
```

This location is known as `<Xcode_Persistent_Cache>`.

If the Xcode cache grows too much, you can delete it using the Empty Caches command. Keep in mind that the Xcode cache may include precompiled headers for your projects, and building those projects may take longer after the cache has been emptied.

To empty the Xcode persistent cache, choose Xcode > Empty Caches.

# Searching Files and Projects

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

Xcode gives you many ways to locate project information and items. *Xcode Workspace Guide* 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 lets you see additional information on Xcode operations, while Info windows let you examine and modify items in your project.

"Opening Files by Filename or Symbol Name" in *Xcode Workspace Guide* describes shortcuts you can use to open a file whose name or path appears in an editor. "Searching Documentation" in *Xcode Workspace Guide* 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.

This chapter describes how to perform text-based searches on files and projects, using string match or regular expressions. It also shows how to use the Project Symbols smart group to find information on the symbols in a project.

## Searching in a File

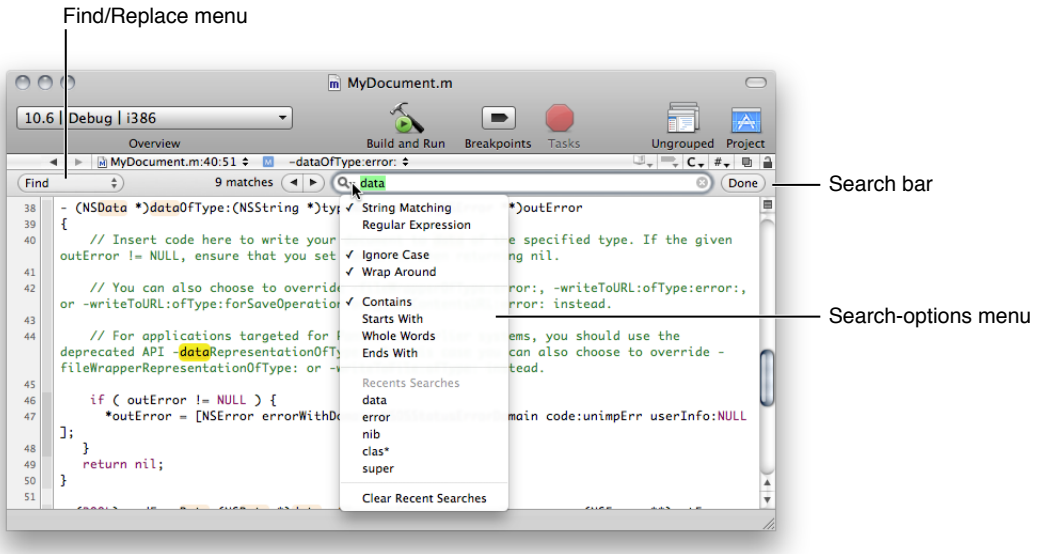
This section shows how to search text within a file. Xcode uses the same mechanism to let you search and replace text in the text editor and the property-list editor.

### Searching Text

To search for text in a file that you have open in the editor, choose Edit > Find > Find.

The search bar appears below the editor toolbar, as shown in Figure 4-1.

**Figure 4-1** The search bar on a text editor window




The Find/Replace pop-up menu lets you choose between Find and Find & Replace. For more on Find & Replace, see "Replacing Text" (page 39).

You can search using a text string or a regular expression; choose the appropriate search type from the search-options menu in the search field:

- **String Matching.** Searches for text matching the string in the search field.
- **Regular Expression.** Searches for text matching the regular expression in the search field.

  Xcode uses the ICU library for regular expression matching (see http://icu-project.org/userguide).

Type the text string or regular expression pattern to use for the search in the search field. To minimize your typing, Xcode keeps track of search strings; to reuse a previous search string, choose it from the Recent Searches group in the search-options menu. You can clear the recent-searches list by choosing Clear Recent Searches from the search-options menu.

The other options in the search-options menu give you additional control over how the search is performed; these options are:

- **Ignore Case.** Select 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.
- **Contains.** Searches for words that contain matching text in a substring.
- **Starts with.** Searches for words that begin with text matching the search term.
- **Whole words.** Searches for words that contain only text matching the contents of the Find field.
- **Ends with.** Searches for words that end with matching text.

Use the Next and Previous buttons (to the left of the search field) to continue searching for the same text in a file. You can also use menu commands. Choose Edit > Find > Find Next to move to the next match and Edit > Find > Find Previous to move to the previous match.

**Table 4-1** Search commands

| Command | Action |
| --- | --- |
| Move to next match | Press Return. |
| Dismiss the search bar | Press Escape. |

## Replacing Text

You can replace some or all occurrences of text matching the string or regular expression specified in the search field.

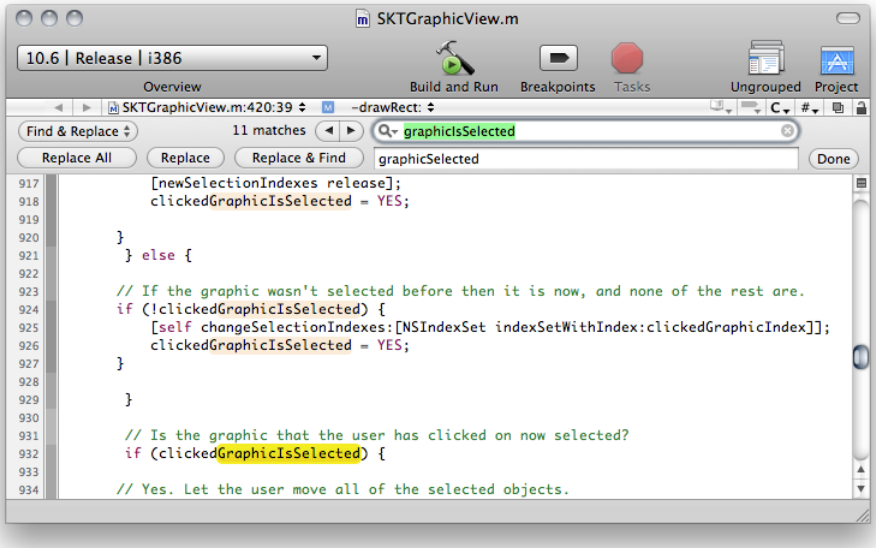
**Tip:** Xcode refactoring provides a more efficient way of renaming a symbol throughout multiple files. See "Refactoring" for details.

To search for text in a file and replace it with other text, do one of the following:

- Choose Edit > Find > Find and Replace.
- In the search bar, choose Find & Replace from the Find/Replace menu.

The search & replace bar appears, as shown in Figure 4-2.

**Figure 4-2** The search & replace bar on a text editor window



Use the replace buttons 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 All.** Searches the entire file or selection and replaces all occurrences of text matching the contents of the Find field with the replacement text.

  Holding down Option changes the search & replace scope to the selected text.
- **Replace.** Substitutes the replacement text for the current match.
- **Replace & Find.** Substitutes the replacement text for the current match and then finds and selects the next match.

Each of these buttons also has a menu item equivalent in the Edit > Find menu: Replace, Replace All, Replace and Find Next, and Replace and Find Previous.

**Table 4-2** Search & replace commands

| Command | Action |
| --- | --- |
| Replace the current match and move to next match | Press Return. |
| Dismiss the search & replace bar | Press Escape. |

# Searching in a Project

Xcode provides a number of ways to search for information in a 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 projectwide search features in Xcode 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 and shortcuts for performing searches from an Xcode editor window are discussed further in “Navigating Source Files” in *Xcode Workspace Guide*.

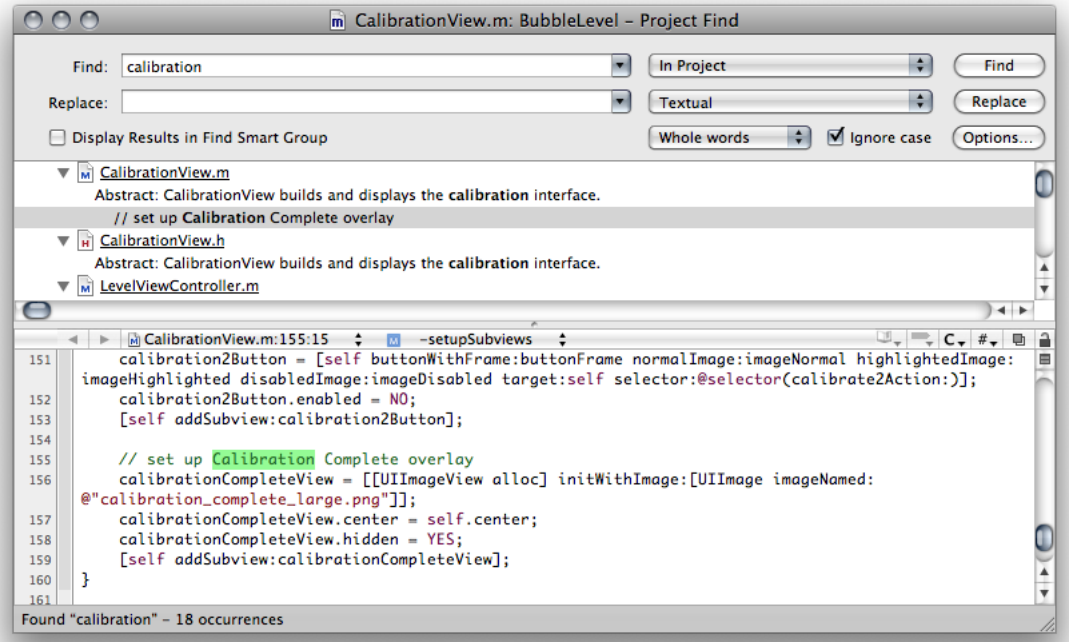
For more information on Code Sense, the technology that provides symbol definition searches, see "Symbol Indexing" (page 47).

## 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 Edit > Find > Find In Project. A window similar to the one in Figure 4-3appears.

**Note:** In the All-In-One project window layout, choosing Edit > 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 in Figure 4-3.

**Figure 4-3** The Project 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.

- 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.
- 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. Xcode uses the ICU library for regular expression matching; for more information, see the http://icu.source-forge.net/userguide.

    **Important:** When replacing text using regular expressions, instead of the dollar (`$`) sign to specify references to capture-group text, use the slash (`\`) character.

    For example, to specify capture group 0 in the replacement text, use `\0`, not `$0`.

  - **Definitions.** Finds any symbol definition matching the symbol name in the Find field.

- **Symbol.** Finds code that uses the symbol identified in the Find field.

  **Important:** Xcode uses Spotlight to find symbols. Therefore the volume on which the source code you're searching through must be indexed by Spotlight. If the source code is stored on a disk image, you must mount the disk image and tell Spotlight to index the volume it represents using this command:

  ```
  > sudo mdutil -i on /Volumes/<volume_name>
  ```

- 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.
- 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" in *Xcode Workspace Guide*.

## 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.** Search the files that are directly included in your project.
- **In Project and Frameworks.** Search the files and the frameworks included in your project.
- **In Frameworks.** Search files that are in the frameworks included by your project.
- **In All Open Files.** Search all open files.
- **In Selected Project Items.** 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 42).

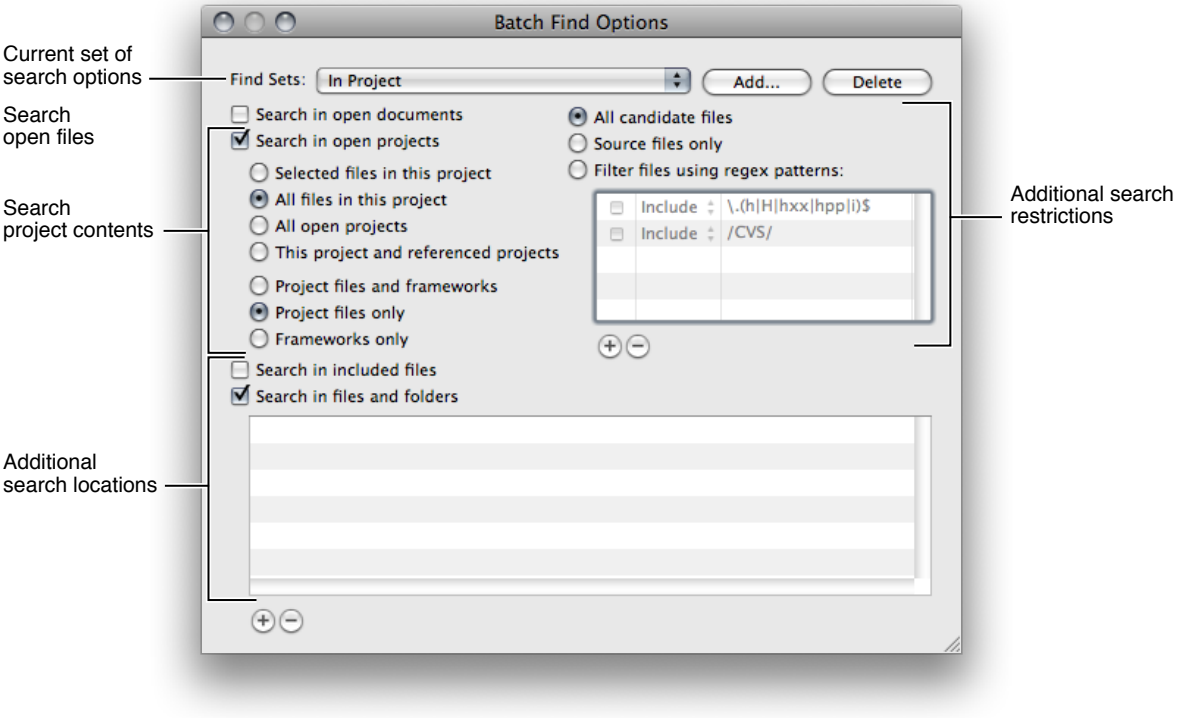
## Creating Sets of Search Options

The Batch Find Options window lets you further tailor the scope of the search. You can further narrow which files and projects are searched, filter the list of files to be searched according to a regular expression, and add directories to the list of locations to search.

You can modify default sets of search options or define your own sets. Defining your own set 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 choose 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 one in Figure 4-4.

**Figure 4-4** 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:

- "Search in open documents" includes all files that are open in an editor in the search.
- "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 and bottom set of radio buttons lets you specify whether to include project or framework files in the search.

- "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. 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.

  The first column in the table specifies whether to use the corresponding regular expression. The second column specifies whether to return the files that match a given regular expression or the files that do not match the regular expression.

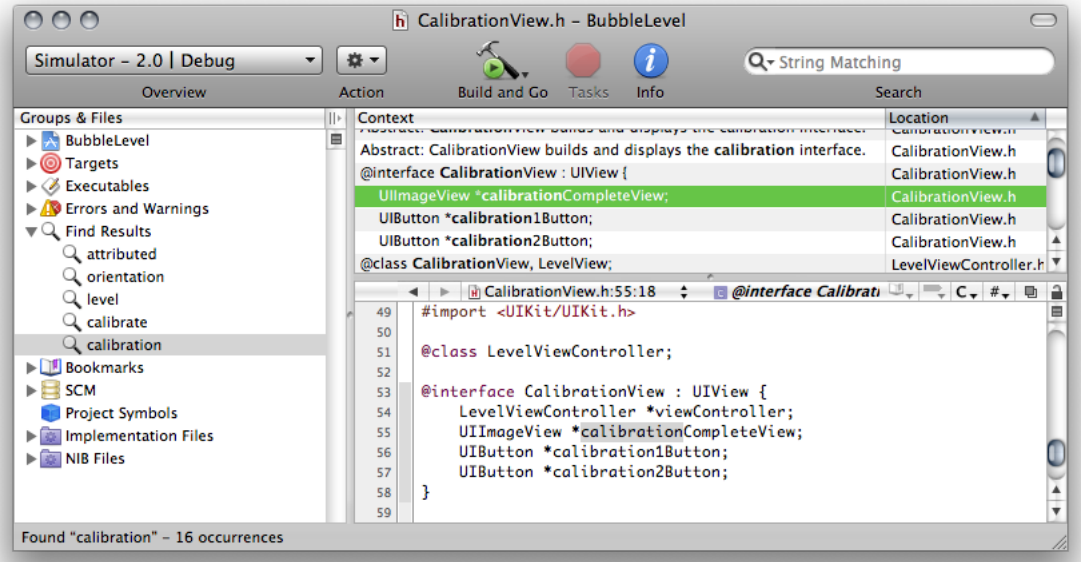
## Viewing Search Results

When you perform a projectwide find, the results of the search are listed below the search criteria; results are organized according to the file in which they appear. You can view a particular search result in the file it was found in 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.

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, as shown in Figure 4-3 (page 41). 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 in Figure 4-5. 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 4-5** Search results in the project window




To delete a search result from the Groups & Files list, select the result and press Delete.

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.

## 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.

# Shortcuts for Finding Text and Symbol Definitions

Xcode provides a number of shortcuts for searching that use text or other content that appears in the text editor. You can use these shortcuts to perform single-file and projectwide 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 File" (page 37) and "Searching in a Project" in *Xcode Project Management Guide*.

To search the current project for text that appears in the text editor, select the text to search for, and choose Edit > Find > Find Selected Text.

To perform a projectwide search using the current selection in the text editor, use the shortcuts listed in Table 4-3.

**Table 4-3** Shortcuts for performing a projectwide search using the current selection in the editor

| Search project for | Choose |
|---|---|
| Selected text | Edit > Find > Find Selected Text in Project |
| Selected symbol definition | Editor shortcut menu > 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 Edit > Find > Jump to Definition.

Each of the searches described in Table 4-3 (page 46) uses the last set of search options used when searching your project. If you want to perform a projectwide search using the current selection in the text editor, 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 the search term, choose Edit > 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 Edit > Find > Use Selection for Replace.

  Xcode opens the Project Find window and places the contents of the current selection in the Replace field.

# Viewing Project Symbols and Classes

Xcode maintains detailed information about the symbols in and utilized by your project to assist you in the development process. Xcode 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 "Completing Code" in *Xcode Workspace Guide*.

This chapter shows how to use the Project Symbols smart group to view symbols and how to take advantage of the class browser to find information on the classes defined in a project and its included frameworks.

## Symbol Indexing

Symbol Indexing makes it simple to find and view information about your code and to gain easy access to project symbols. A project's symbol index 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, Xcode provides support for features such as the following:

- **Code completion.** Code completion, described in "Completing Code" in *Xcode Workspace Guide*, assists you in writing code by providing completion 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 in the Class Browser" (page 50).
- **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 48).
- **Searching.** Xcode offers numerous ways to search for symbols and other information in your project. For instance, you can perform a projectwide 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 40).

Xcode creates the a project's symbol index 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 Info window and clicking the Rebuild Code Sense Index button.

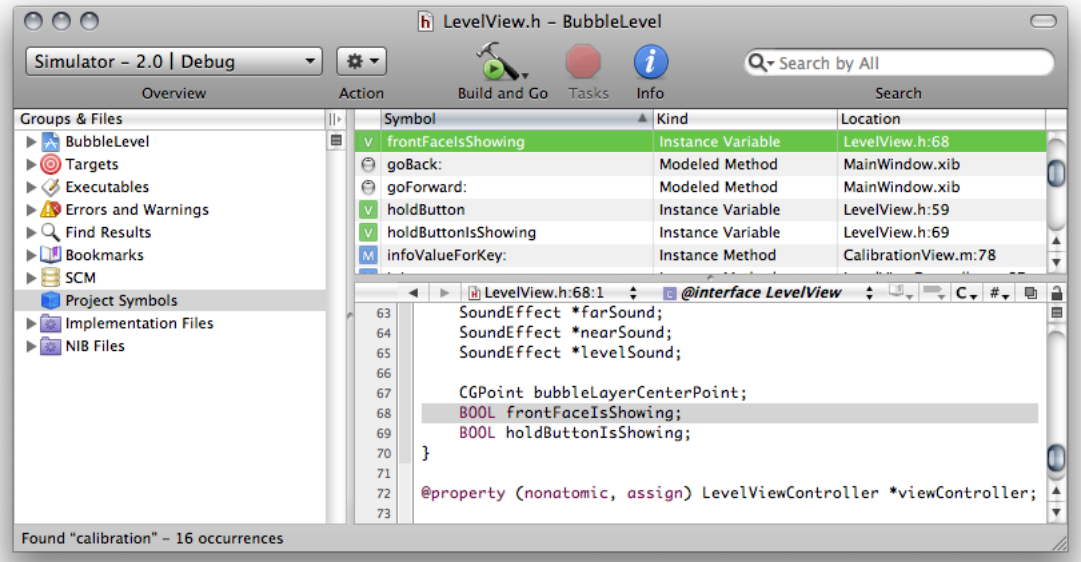
Indexing is enabled by default. You can disable indexing for all projects that you open. To do so, choose Xcode > Preferences, click Code Sense, and deselect the "Enable for all projects" option in the Indexing section.

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 Info window, described in "Viewing File Information" in *Xcode Workspace Guide*.

# 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. When you select the Project Symbols group for a project, you get a view like the one shown in Figure 5-1.

**Figure 5-1** Viewing symbols in your project



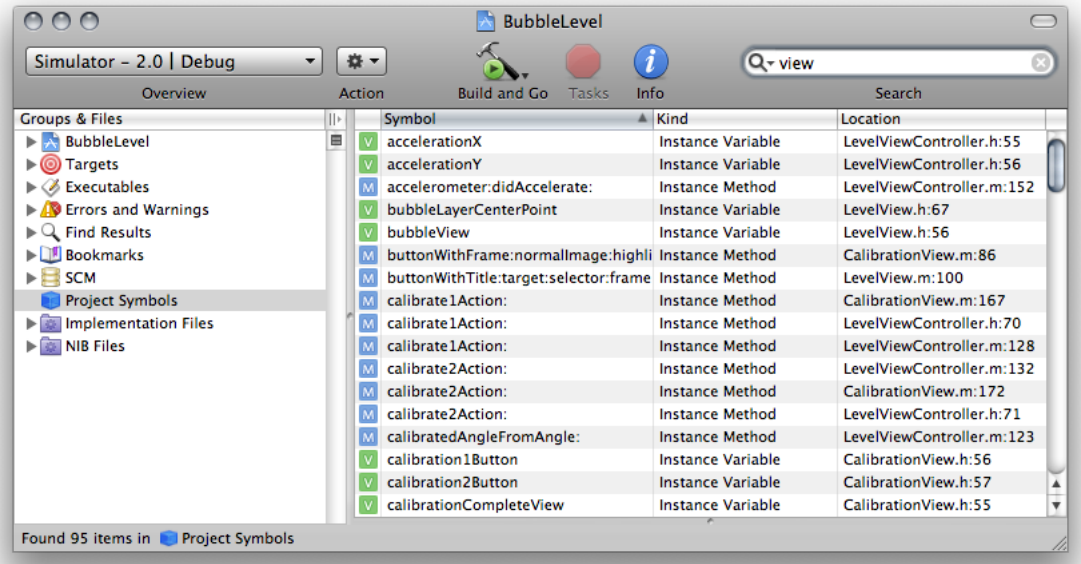

The detail view shows the following information for each symbol:

- An icon indicating the symbol type, such as structure, function, or method. 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 BubbleLevel project, for example, you can search for all symbols declared in files pertaining to views by choosing "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 Figure 5-2. By default, the search field searches the content of all categories in the detail view.

**Figure 5-2** Filtering the symbols in a project




You can configure which information is displayed in the detail view, as described in "The Detail View" in *Xcode Workspace Guide*.

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 the symbol's shortcut (or contextual) menu, which 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 clipboard; you can then paste this declaration into any text field or editor using Command-V. This command works only for functions and methods.
- **Copy Invocation for Method/Function.** This command copies the invocation string for the selected symbol to the clipboard, 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.

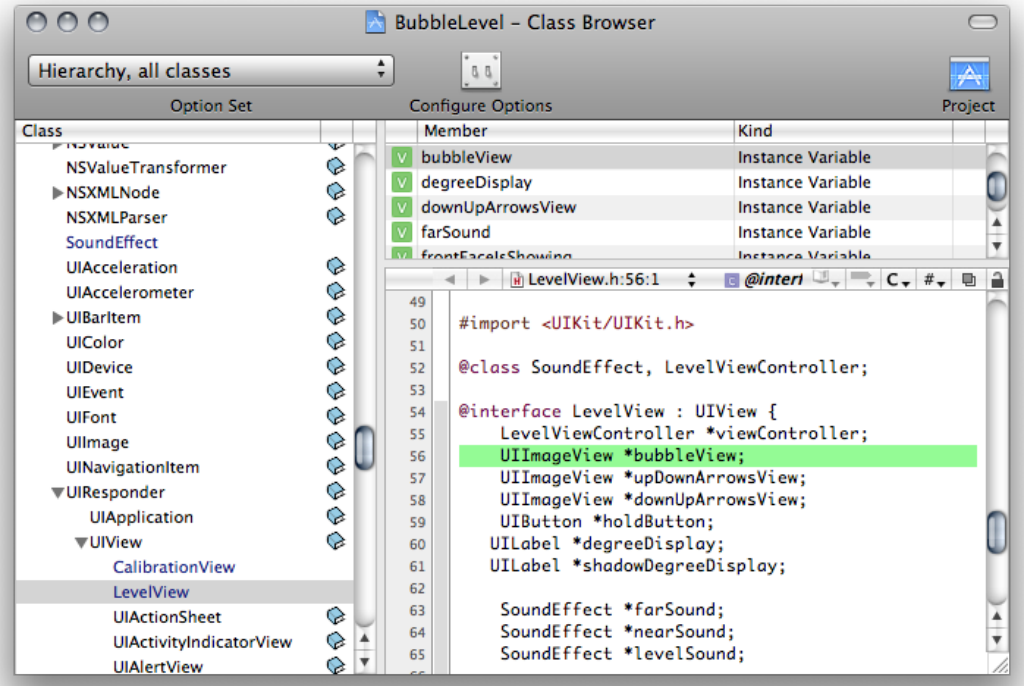
**Note:** The availability of these commands in a symbol's contextual menu depends on the how you configure the class browser. See "Choosing What the Class Browser Displays" (page 52) for details.

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 in the Class Browser

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. Figure 5-3 shows the class browser.

**Figure 5-3** The class browser




Classes and other top-level symbols—protocols, interfaces, and categories—are listed in the Class pane on the left side of the class browser. 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 Info window. 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.

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 shortcut menu. Xcode creates a bookmark to the class or member's definition.

## Choosing What the Class Browser Displays

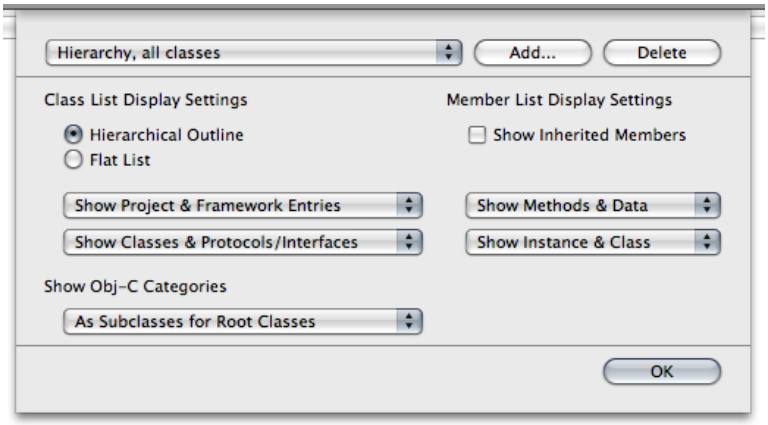
You choose which information Xcode displays in the class browser using the Option Set pop-up menu at the top of the browser window. This menu lets you switch between sets of display options. Xcode provides a few sets of predefined display options that allow you to choose:

- 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, shown in Figure 5-4, 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 5-4** 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 a new set of search options, click the Configure Options button, click Add, enter a name for your options set, and configure your options.

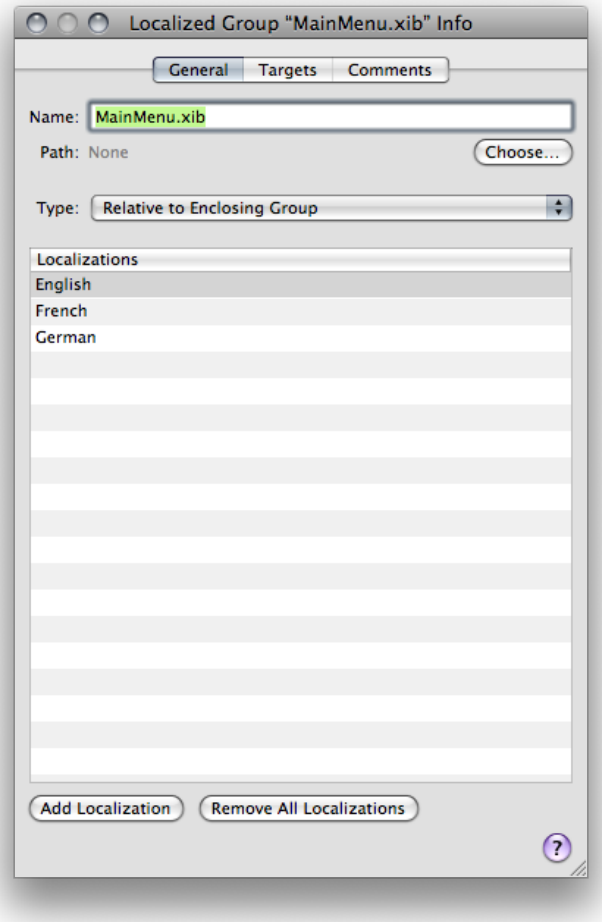
To remove a set of search options, click the Configure Options button, choose the options set from the pop-up menu, and click Delete.

# Localizing Files

Xcode lets you create applications, bundles, and frameworks that are customized for different locales. Generally, you start by creating a variant for one particular locale, called the development locale, and add more variants later.

In the Groups & Files list, a file customized for different locales appears as a localized group, which has a file icon with a disclosure triangle beside it. To see the file's variants, click the triangle. To add and remove variants, select the localized item, open the Info window, and use the two buttons at the bottom of the General pane, as shown in Figure 6-1.

**Figure 6-1** The Info window for a localized item



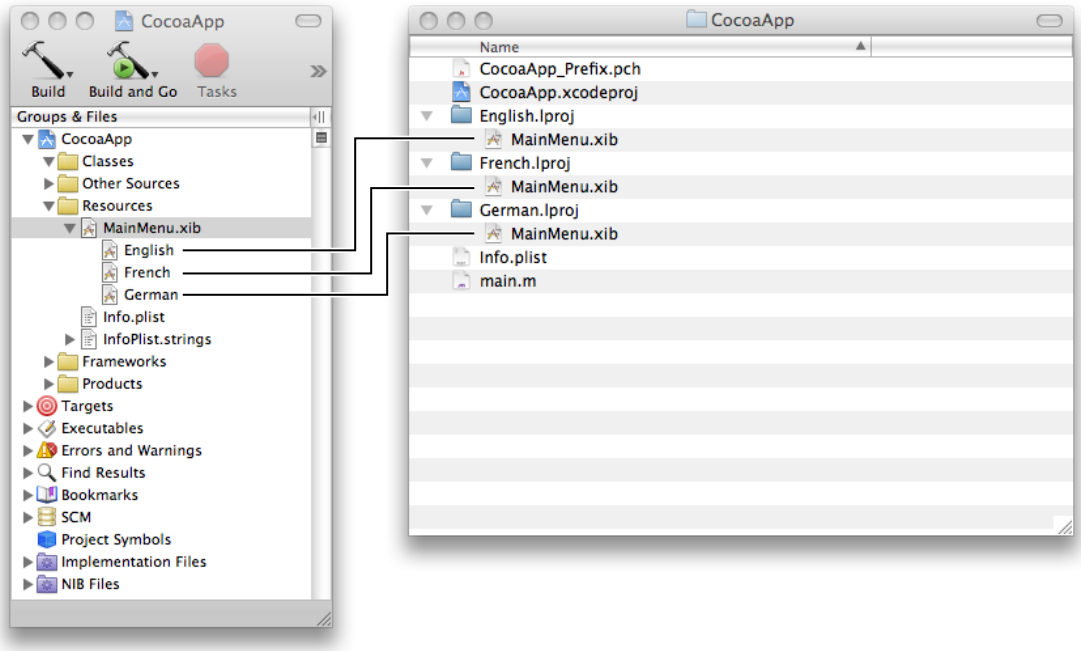

For more information on localizing your product for different regions, see *Internationalization Programming Topics*.

# Marking Files for Localization

To mark files for localization, select the files, open the Info window, and click the Make File Localizable button. Xcode moves the files into the locale's `.lproj` folder. If a file was already in another `.lproj` folder, Xcode copies it to the locale's `.lproj` folder.

Xcode creates a localized group in the Groups & Files list, with the file's name and icon. To view the individual locales, click the disclosure triangle next to the localized group icon. Figure 6-2 shows the localized group for an application's main nib file in the Groups & Files list.

**Figure 6-2** A localized item in the Groups & Files list and the project directory




You can inspect of 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 File Info window, and click the Remove All Localizations button. Xcode moves the files from the locale'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 file system.

# Adding Files for a Locale

To add files for a locale, select the file or localized group for which you want to add another locale, open the File Info window, and click the Add Localization button. Xcode queries you for the name of the locale and copies the development locale's version of the files to the new locale's `.lproj` folder.

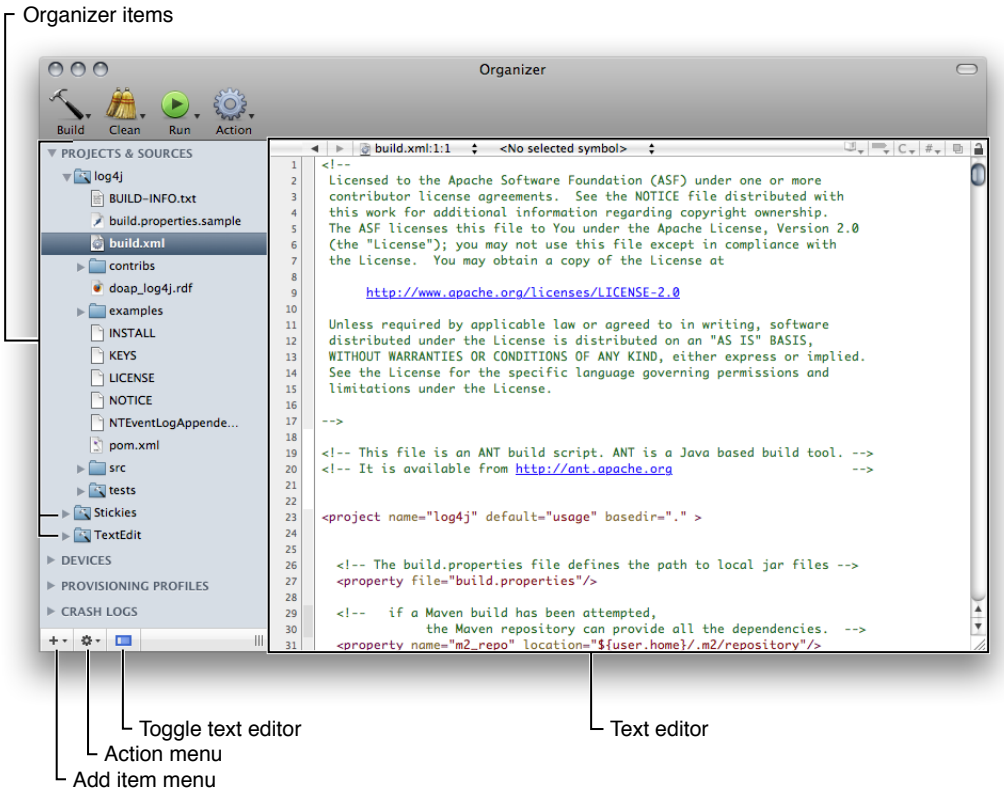
# Using the Organizer

The Xcode Organizer allows you to access frequently used directories (containing projects or other resources) from a single window, which reduces the amount of Finder-based navigation you need to perform to get to those files. The Organizer also streamlines your development workflow by letting you assign tasks to the directories it displays. To build a product, you don't have to open the corresponding project in an Xcode project window. And you can manage directories containing Xcode projects as well as directories with projects that use other build systems in the Organizer.

You use the Organizer by adding Organizer items to it. An **Organizer item** represents a directory in your file system (you can think of these items as symbolic links or folder references). You can perform many of the functions the Finder provides, such as moving or deleting files and directories. However, the real power of the Organizer comes from its support of Organizer actions. An **Organizer action** is a predefined or custom task that Xcode performs on a directory. See "Using Organizer Actions" (page 58) for details.

To open the Organizer, shown in Figure 7-1, choose Window > Organizer.

**Figure 7-1** The Organizer




The Organizer can work with projects that use any build system that can be operated through shell scripts, including Xcode projects, and Make-based and Ant-based projects.

To add an item to the Organizer, perform any of these actions:

- Drag a folder from a Finder window into the Organizer.
- From the Add Item menu in the bottom-left corner of the Organizer, choose one of these options:
  - **New File.** Creates a empty text file.
  - **New Folder.** Creates an empty directory.
  - **New From Template.** Creates a directory with predefined content.

To remove an item from the Organizer:

1. Select the item in the Organizer.
2. Choose Remove From Organizer from the Action menu.

To change the directory to which an Organizer item points:

1. Select the item in the Organizer.
2. Choose Assign New Location from the Action menu.

To create a snapshot of an Organizer item:

1. Select the item in the Organizer.
2. Choose Make Snapshot from the Action menu.

# Using Organizer Actions

In addition to providing a single place from which to access projects and other resources that you use often, the Organizer allows you to assign actions to the directories it displays. For example, to build a product needed by your project, you can create an Organizer item that points to its project directory. When you need a fresh copy of the product, you can build it from the Organizer. This allows you to get what you want, the product, with minimal distractions. That is, for Xcode projects, you don't have to open the project in a project window and build it; and, for other types of project, you don't have to use legacy targets in Xcode, or issue commands in a Terminal window to execute build scripts.

The Organizer supports four types of actions: build, clean, run, and general. These actions are accessible through four Organizer-action toolbar items with pop-up menus: Build, Clean, Run, and Action.

- **Build actions** generate a product.
- **Clean actions** delete product files and intermediary build files.
- **Run actions** launch executable files.
- **General actions** can perform any type of task.

To perform an Organizer action, you select the object on which you want to perform the action (the **action object**) and choose the action from one of the Organizer-action pop-up menus. Although the Organizer provides actions for directories that use build systems it recognizes, you may have to edit those actions or define custom actions to build a product correctly. "Managing Organizer Actions" (page 59) explains how to implement custom actions.

# Managing Organizer Actions

Organizer actions have two main attributes: a defining directory and a root directory specifier. The **defining directory** is the directory upon which the action is "attached." The **root directory specifier** represents the desired directory for the action.

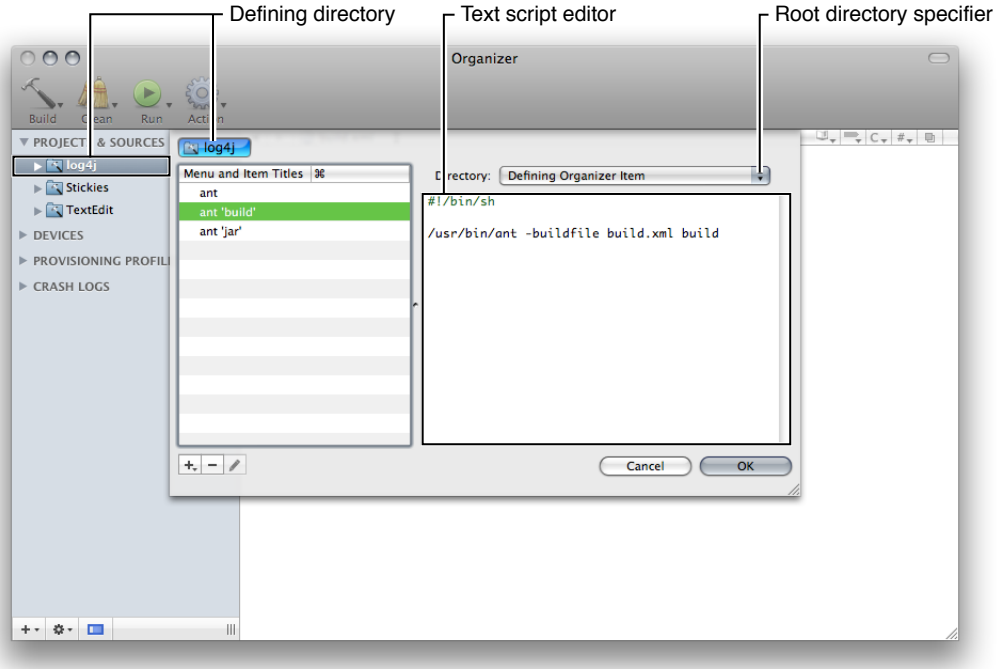
To define an Organizer action:

1. Select the directory in the Organizer to which you want to attach the action.
2. Choose Edit Actions from the appropriate Organizer-action pop-up menu.

   The Organizer action editor (Figure 7-2) appears.
3. Add and implement the action in the action editor.

   To implement actions, you can use shell scripts, AppleScript scripts, or Automator workflows.

**Figure 7-2** The Organizer action editor



The Directory pop-up menu in the action editor contains several options for designating the action's root directory. Table 7-1 lists the available action directory choices and the corresponding action root directory when the action is invoked.

**Table 7-1** Root-directory-specifier choices and resulting root directories

| Root directory specifier | Root directory |
| --- | --- |
| Selection | The directory selected in the Organizer. |
| Top Level Organizer Item | The directory pointed to by the enclosing Organizer item. |
| Defining Organizer Item | The directory that defines the action. |
| Home Directory | The user's home directory. |
| File System Root | The system root (/) directory. |

# Searching Organizer Items

The Organizer allows you to search individual Organizer items. Xcode displays the search results in a window similar to the Project Find window. You can perform textual searches and regular expression searches.

# Editing Text Files in the Organizer

The Organizer lets you edit text files using the Xcode text editor. You can display or hide the editor pane by clicking the text editor toggle button in the bottom-left corner of the Organizer.

**Note:** Code Sense is not available in the Organizer.

# Part II: Product Development

Part II of this document describes essential product development tasks, mainly building products and performing static analysis to find bugs in code before testing the built product This part also describes how to configure executable environments to tailor the testing process.

# Building Products

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:

- Shows how to view and edit build settings and per-file compiler flags
- 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 directory
- Explains how to build projects with multiple targets efficiently using parallel builds
- Shows how to view build status, errors, and warnings

**Prerequisites:** To take the most out of this chapter, you must be familiar with the essential Xcode workflows for creating and building applications. *A Tour of Xcode* shows how to create and build a simple application.

.

## Build Locations

When Xcode builds a target, it generates intermediate files, such as object files, as well as the product described 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.

There are two build-location settings that you can specify to customize the location of build products and intermediate build files. These are build products directory and intermediate build files directory, respectively.

By default, Xcode places both the build products and the intermediate files that it generates in the `build` directory inside of your project directory. If the software you are developing is contained in a single project, this 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 directory to ensure that Xcode can automatically find and use the product created by each of those targets.

**Note:** Within the build directory, Xcode maintains separate subdirectories for each build configuration defined by the project.

Xcode also supports the concept of an installation location, using the Deployment Location (`DEPLOYMENT_LOCATION`) and Installation Directory (`INSTALL_PATH`) build settings. If you're building a product for release, turn on the Deployment Location build setting and supply a path for the Installation Directory build setting. This places the built product at the specified location.

For information about the build settings that specify build locations and how the build products directory and intermediate build files directory settings affect them, see *Xcode Build Setting Reference*.

## Setting Default Build Locations

When you first start up Xcode, it asks you to specify the directories in which it places the files generated by the build system, both intermediate files and built products. These build locations are used for all new projects.

Each project can specify a value for its build products directory and intermediate build files directory. However, new projects defer to the default values for these settings, specified in Building preferences (see "General Project Attributes" (page 19)). Therefore, setting custom build locations in Building preferences, sets the build locations for new projects. For example, to set shared build locations for all the projects you create, set those locations in Building preferences. And, for projects that don't need to share those locations, you can customize their build location, as described in "Setting Project-Specific Build Locations" (page 64).

See "Building Preferences" in *Xcode Workspace Guide* for details.

## Setting Project-Specific Build Locations

Each time you create a project, Xcode sets the build locations for that project to the default build locations specified in the Building pane of Xcode preferences. You can, however, override these default build locations on a per-project basis. By taking advantage of this feature, you can choose default build locations that works best for most of your projects, and specify other default build locations for individual projects as needed.

To override the default location for a project's build products, use the build locations settings in the Project Info window.

- **Place Build Products In.** Sets the project's build products directory.
- **Place Intermediate Build Files In.** Sets the project's intermediate build files directory.

To learn how to change the default build location used for projects that you create, see "Setting Default Build Locations" (page 64).

# Managing Build Settings

Build settings are variables that tell the Xcode build system how to build your product. Build settings provide a powerful and flexible mechanism for customizing the build process.

In the Xcode application, you can define build settings at two levels:

- **Project level:** The build settings defined at the project level specify build aspects that apply to all the targets contained in the project.
- **Target level:** A target inherits the build settings specified by the containing project. The target, however, may also override build settings defined by the project. In other words, the build settings at the target level are the ultimate determinant of how a product is built. (When you use the `xcodebuild` command-line tool, you may override target build settings by defining them in the tool invocation.)

To learn more about build settings, see Build Settings.

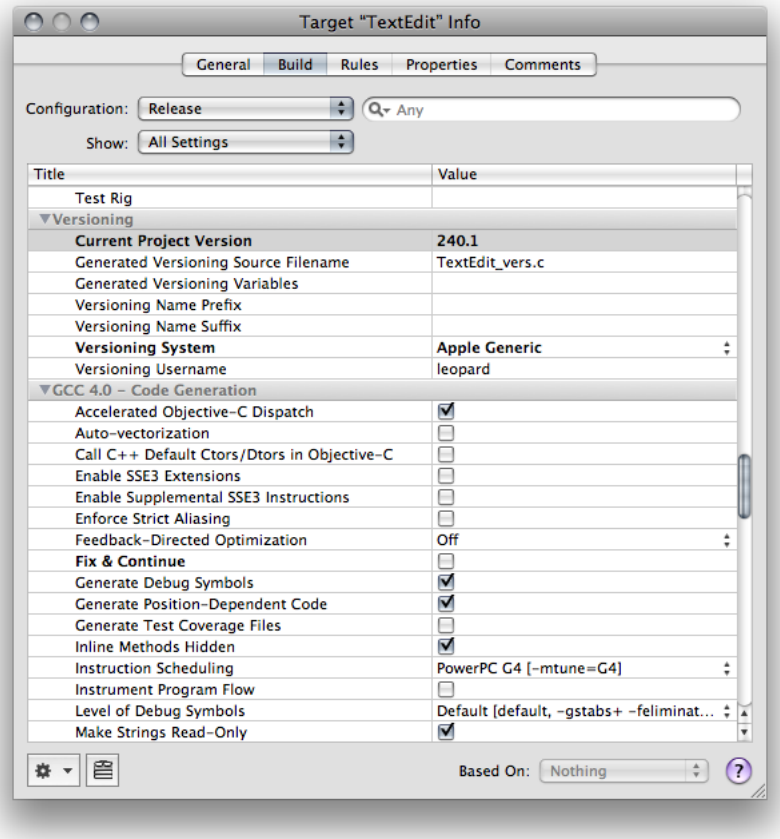
The Xcode application lets you access and edit build settings at the target and project layers. It provides a convenient graphical user interface for changing build setting specifications, the Build pane of Info windows for targets and projects. This section shows how to view and edit build settings using the build settings editor.

**Note:** Build settings defined at the command line (using `xcodebuild`) and per-file compiler flags are not reflected in the Build pane.

## Viewing Build Settings

You can view and edit build settings at the target and project levels in the Build pane of the Info window for targets and projects. Figure 8-1 shows the Build pane for a target.

**Figure 8-1** Build settings for a target




These are the components of the Build pane:

- **Configuration menu.** The Configuration pop-up menu indicates which of the target's build configurations is currently displayed. Choosing Active from this menu displays the build settings defined for the active build configuration; for more information, see "Build Configuration Overview" in *Xcode Build System Guide*. The build settings defined for the active configuration appear in the build settings table.
- **Search field.** The search field lets you search the build settings table for a keyword or other text string. As you type, Xcode filters the list to include only those build settings that match the text in the search field. For example, you can find all build settings related to search paths by typing `search`.
- **Show menu.** The Show pop-up menu lets you filter the list of build settings shown in the build settings table.
- **Build settings table.** The build settings table contains two columns:
  - **Title.** The Title column shows build setting titles, which are brief English-language descriptions for the build settings. You can display build setting names instead by Control-clicking anywhere in the table and choosing Show Setting Names from the shortcut menu.

    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.
  - **Value.** The Value column shows build setting values. You can display build setting specifications by Control-clicking anywhere in the table and choosing Show Definitions from the shortcut menu.

> **Note:** Xcode uses **bold text** to indicate build settings specified at the current level—that is, at the level that you are currently viewing, either the project or the target. Build settings that are not in bold text are specified at lower layers.

- **Action menu.** The action menu lets you add or remove build setting specifications, including conditional build setting specifications.
- **Research Assistant button.** The Research Assistant button opens the Research Assistant, which displays information about the build setting selected in the build settings table. To learn more about the Research Assistant, see "Using the Research Assistant" in *Xcode Workspace Guide*.
- **Based On menu.** The Based On menu specifies the build configuration file upon which the active build configuration is based. (When the project contains no configuration files, this pop-up menu is dimmed.) When you base a build configuration on a configuration file, Xcode sets the default specifications of the build settings in the build configuration to the corresponding specifications in the configuration file.

You can modify build settings for multiple configurations of a target at once; to do so, choose All Configurations from the Configuration menu, as described in "Managing Build Configurations" in *Xcode Build System Guide*.

You can also modify build settings in multiple targets at once; the build settings table 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 Info window.

The build settings table supports copy and paste, as well as drag and drop, of build settings. If you have already configured a group of settings for a target, you can reuse them by selecting those build settings and dragging them into a text file, or by copying and pasting them between Info windows.

For a list of the available build settings, see *Xcode Build Setting Reference*.

## Editing Build Setting Specifications

The interface for modifying a build setting's specification in the build settings editor varies according to the possible values for that build setting. Depending on the value, Xcode may display, for example, a text field, a table, a checkbox, or a pop-up menu. This list describes the interface to edit the possible build setting value data types:

- **String.** If a build setting takes a string as its value, double-click the Value cell to edit it.

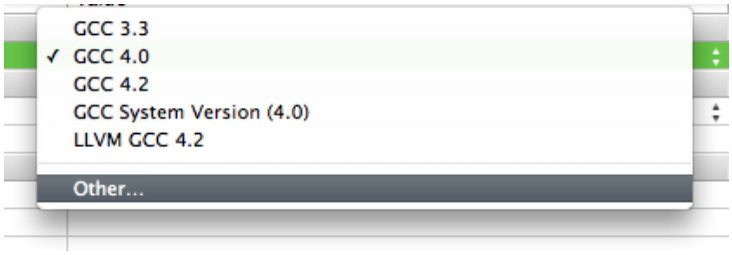
  If you are entering a file path, you can simply drag the file or folder from the Finder into the text field.
- **List.** If a build setting can have a list of items as its value, the Value column displays the value as a space-separated list. You can add or remove items from a lis of values by double-clicking anywhere in the row and using the plus and minus buttons in the dialog that appears.

  If you are entering file paths, you can simply drag the file or folder from the Finder into the list, instead of typing the paths in yourself.

> **Note:** Because Xcode interprets spaces as item separators, you must not use the list control to edit values that contain items with spaces. To edit a value containing spaces in one or more of its item, you must enter the value as a space-separated list using the string control. That is, single-click the Value cell and enter the items. Put quotation marks (either ' or ") around the items that contain spaces.

- **Boolean.** If a build setting can have two states—on or off—the Value cell contains a checkbox. A checkmark indicates that the build setting is turned on.
- **Choice.** If a build setting has a finite number of possible values, the build setting's Value cell contains a pop-up menu with the possible values.

  For some build settings, in addition to the standard values shown in the value pop-up menu, you may specify a custom value by choosing Other from the value list.




## Adding and Deleting Build Settings

If you do not see the build setting you want to modify, or if you want to define custom build settings, you can add build settings to the build settings table in the Build pane of the Info window for a target or project.

To add a build setting definition to a target or project, choose Add User-Defined Setting from the Action menu in the bottom-left corner of the window.

To remove a build setting from a target or project, select the build setting definition to remove and choose Delete Definition at This Level from the Action menu.

If the build setting definition that you deleted is defined at a lower layer, Xcode continues to display that build setting in the build settings table. However, Xcode displays the build setting in nonbold text, indicating that it's defined elsewhere.

## Editing Conditional Build Settings

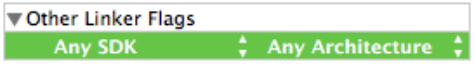
Conditional build settings let you specify the conditions under which you want particular build setting specifications to apply. For example, when building using a particular SDK or for a particular platform. For details, see "Conditional Build Settings" in *Xcode Build System Guide*.

To add a conditional build setting definition:

1. In the build settings table, select the build setting to which you want to add a conditional definition. See "Viewing Build Settings" (page 65) for details on how to find the appropriate build setting.
2. From the Action menu, choose Add Build Setting Condition.

The build setting conditions appear below the build setting, as shown in Figure 8-2.

**Figure 8-2** Build setting conditions




3. From the Any SDK and Any Architecture pop-up menus, choose the conditions under which you want the build setting specification to apply.

   For example, if you want the build setting to apply to any architecture and any release of the iPhone OS Device SDK, choose "Any iPhone OS Device" from the Any SDK pop-up menu (Figure 8-3). To specify a particular release of the iPhone SDK—for example, the one for iPhone OS 2.2—choose "Simulator - iPhone OS 2.0."

**Figure 8-3** Conditional build setting

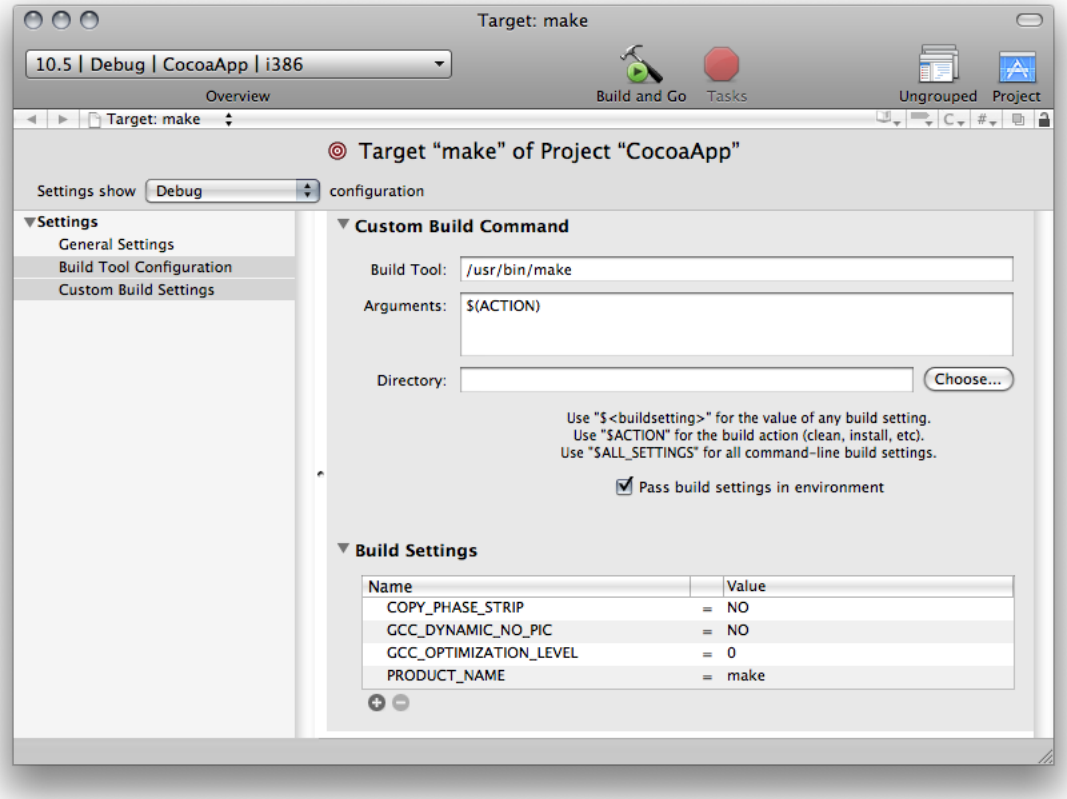

**Note:** The build settings table shows only the SDK and architecture conditions. It doesn't show the variant condition described in "Conditional Build Settings" in *Xcode Build System Guide*.

## Editing Build Settings for Legacy and External Targets

You cannot configure build information for Jam-based Project Builder targets and external targets in the build settings table. To edit the build settings for Jam-based and external targets in Xcode, select the target in the Groups & Files list. If you have a text 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 External Target Info window (shown in Figure 8-4), 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 Info window.

**Figure 8-4** External Target Info 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 Custom Build Settings 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.

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.

# Per-File Compiler Flags

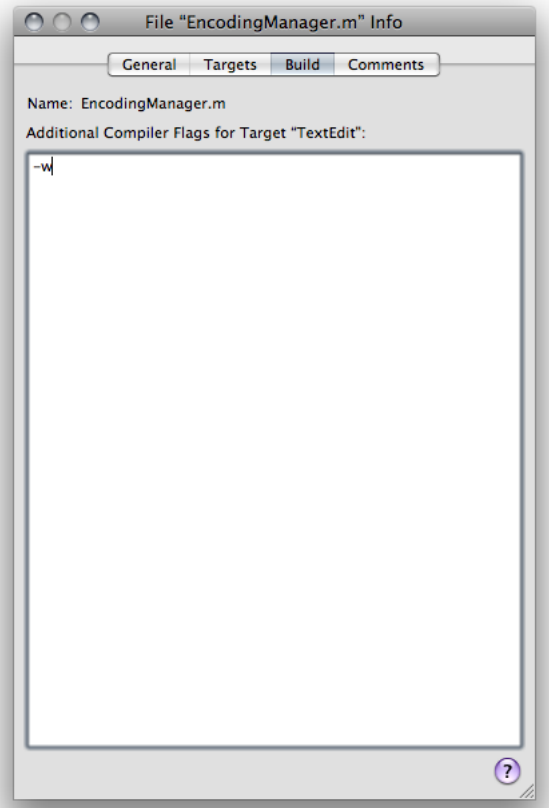
The build system provides facilities for customizing the build process of source files of particular types. It includes the Other C Flags (`OTHER_CFLAGS`), Other C++ Flags (`OTHER_CPLUSPLUSFLAGS`), Other Warning Flags (`WARNING_CFLAGS`), and Preprocessor Macros (`GCC_PREPROCESSOR_DEFINITIONS`) build 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 for particular files. For example, in a project that treats warnings as errors in general, you may have a set of cross-platform source files whose warnings you want the compiler to ignore.

To view the compiler flags for a file:

1. Select the source file in the Groups & Files list.
2. Choose File > Get Info to open the File Info window.
3. Display the Build pane (Figure 8-5).

**Figure 8-5** File compiler flags




**Note:** The Build pane of the File Info window is available only for files containing source code.

In the Additional Compiler Flags for Target field, you can enter any compiler flags you want to assign to the file. Use spaces to separate flags.

You can use multiple selection to assign compiler flags to a subset of the files in a target.

For each target that a file belongs to, Xcode maintains a separate list of compiler flags assigned to the 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 the File Info window, as described earlier. If you open an Info window for the same file from any other 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 available only if the file is part of the active target.

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 for a file that produce build errors or warnings, you should remove all the options you added and add them back one at a time, making sure the file compiles after you add each option, to determine which option is not supported. You should also consult the tool's documentation to learn about possible conflicts.

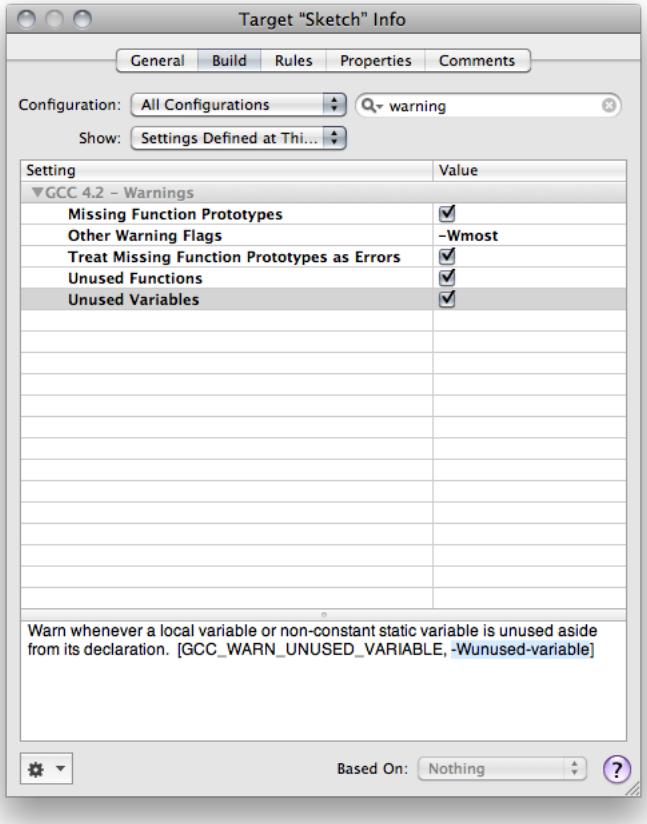
The additional compiler options specified for a file are always used when the file is processed as part of a build, and these compiler options cannot be overridden in any of the build setting layers. (Build setting layers are described in "Build Setting Evaluation" in *Xcode Build System Guide*.)

You can use file compiler options to negate compiler options generated by the build system. For example, you may have turned on the Unused Variables build setting for your target, but want to deactivate unused-variable warnings for only one of the target's source files.

To negate the effect of a build setting on a particular file:

1. Clean and build the target to ensure there are no build errors or warnings.

2. Using the build-setting descriptions in the Build pane of the project or target Info window, determine the compiler option that corresponds to the build setting in question.




3. Add the negative form of the compiler option, by prepending it with `-Wno-`, to the compiler flags of the file whose build-setting effect you want to suppress. For example, to negate the `-Wunused-variable` compiler option (which the build system generates when Unused Variables is turned on), add `-Wno-unused-variable` to the file's compiler options.

4. Build the product. If there are build errors, the compiler option just added may not be supported by the compiler.

# 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`)

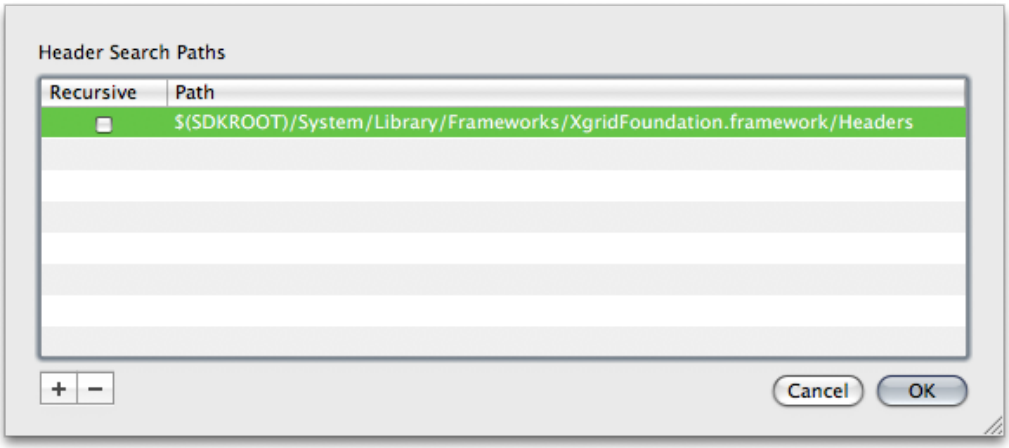
  This is a list of paths to search for files included by Carbon Resource Manager resources and compiled with the Rez tool.

For additional information on these build settings, see *Xcode Build Setting Reference*.

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 67). 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 list editor, shown in Figure 8-6. Xcode appends `**` to the search path. When you build, Xcode searches the directory at the specified location, and all subdirectories in it, for the header, framework, library, or resource.

**Figure 8-6** Header Search Paths list



**Note:** Xcode does not search the contents of certain bundle structures and other special directories. These are `.nib`, `.lproj`, `.framework`, `.gch`, `.xcodeproj`, `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 symbolic links. 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 expands a single recursive search path to is 1024.

# Building with Xcode

Building targets 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 Build Factors

Before you start a build task, you should ensure that the build factors are set appropriately. A **build factor** is a setting that defines a particular aspect of a build. There are five build factors:

- **Active target.** Identifies the target (and dependent targets) that's built when you execute the Build command. See "Targets" for details.
- **Active SDK.** Specifies the build environment, which includes the core and additional frameworks and libraries against which the targets' source code is compiled and the command-line tools used to build the product.
- **Active configuration.** Specifies the build configuration to use when building the active target and its dependent targets. See "Build Configuration Overview" in *Xcode Build System Guide*.
- **Active architecture.** Specifies the architecture for which the product is built, such as i386, x86_64, or ArmV6.
- **Active executable.** Specifies the executable environment to use for running the product after it's built successfully. See "Setting the Active Executable" (page 102) for details.

You can set a project's build factors (except the active SDK) using the Project menu or the Overview toolbar menu.

# Building a Target

After setting the active target and build configuration, you're ready to build the target's product. In a build, the target being built is known as the current target. The first time you build a target, the build system creates all the intermediate files, such as object files, needed to build the product. In subsequent builds of the same target, the build system processes only the files that have changed since the target's previous build. 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, the build system recompiles that file and relinks the object files to create the finished product. That is, changes to the contents of the file and changes to build settings that affect the way a source file is built cause the build system to rebuild the file.

To build the active target, execute one of the following commands from the Build menu:

- **Build.** Builds the target.
- **Build and Analyze.** Builds the target and performs static analysis on the target's source files. For more information about static analysis, see "Analyzing Code" (page 89).
- **Build and Run.** Builds the target and launches the built product.
- **Build and Run – Breakpoints Off.** Builds the target and launches the built product with its breakpoints turned off. Execution continues through breakpoints.
- **Build and Debug – Breakpoints On.** Builds the target and launches the built product with its breakpoints turned on. Execution stops when the first active breakpoint is reached.

You may also build targets individually, regardless of which is the active target, by choosing the appropriate build command from the target shortcut menu in the Groups & Files list.

**Note:** If you want to force Xcode to rebuild a build file the next time you build the target, you can touch the file, to mark it as changed. Similarly, if Xcode has marked the file as needing to be rebuilt, but you know you haven't made any substantive changes to the file, you can untouch the file; the build system does not rebuild the file the next time the target is built. To touch or untouch a file, click in the build status column (indicated by the hammer icon) next to that file in the detail view. If a checkmark appears in this column, the file is rebuilt the next time you build a target to which it belongs. Keep in mind that this feature does not work with all build systems.

Xcode builds the current target in two major stages:

- **Target dependencies.** The build system builds the targets on which the current target depends.
- **Build phases.** The build system processes the current target's build phases.

Building a target produces build messages, which include errors, warnings, and static-analyzer messages. See "Viewing Build Results" (page 78) for information on how to view build messages, and how to address errors and warnings.

If your product does not build properly and there are no error messages, make sure your files have correct dates. Files with invalid dates (that is, dates before 1970) won't compile correctly.

## Viewing Preprocessor Output

You can see the preprocessor output for a C, C++, or Objective-C source files in the active target. To do so, select the files and choose Build > Preprocess.

Preprocessor output can also identify the `#define` directives (including predefined macros) in effect for a target's source files. To include these directives, add `-dM` to the Other C Flags (`OTHER_CFLAGS`) build setting. For more information on the C preprocessor, see *GNU C 4.0 Preprocessor User Guide*.

## Compiling Files

A full build can take a long time if you build target with many source files. You can compile a subset of a target's source files to ensure that they build correctly without having to build the entire target. To compile a set of target files:

1. Select the target files (they must be part of the active target).
2. Choose Build > Compile.

## Viewing Assembly Code

To see the assembly code output by the compiler for a source code file, select that file and choose Build > Show Assembly Code. Xcode compiles the file, and any additional files required to build it, and displays the assembly code generated by the compiler in an editor. You show assembly code for multiple files by selecting them in the project window.

## Removing Build Products and Intermediate Build Files

As you learned in "Building a Target" (page 76), after the initial build of a target, the build system 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, the build system removes all 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 the active target, choose Build > Clean.

The Clean Target dialog appears. It contains two options:

- **Also Clean Dependencies.** Cleans the targets on which the target depends.
- **Also Remove Precompiled Headers.** Removes the precompiled header binaries used by the target.

To clean all targets in your project, choose Build > Clean All Targets.

## Finding Header Files

All headers in your project are automatically accessible to your source code. You can also specify additional search paths at which to find headers using the Header Search Paths (`HEADER_SEARCH_PATHS`) build setting, described in "Search Paths" (page 73).

Note that you do not have to add headers to your target to include them in your source code. You should add header files to your target—as part of the Copy Header Files build phase—only if they are embedded in your product, as with a framework.

By default, Xcode searches user paths before system paths. This means that, by default, your project headers have precedence over system headers. Thus, if your project defines a header file named `String.h`, source code in your project that includes `String.h` incorporates your project's version of the header, not the system version. You can override this behavior by turning off the Always Search User Paths (`ALWAYS_SEARCH_USER_PATHS`) build setting.

When Always Search User Paths is inactive, project headers can be incorporated into translation units only by surrounding the header filename with quotation marks. (A **translation unit** is a single source file with its imported/included files). (A translation unit is a source file with its imported/included header files.) For example, `#include "String.h"` incorporates the `String.h` header file defined in the project into generated translation unit. Furthermore, system headers can be incorporated into translation units only by surrounding the header filename with angle brackets. For example, `#include <String.h>` incorporates the `String.h` system header file into the generated translation unit.

## Viewing Build Status

During a build, you want to see how that build is progressing. Especially for long build processes, it is useful to know the status of that process. Xcode displays the build status in the project window status bar. The status bar message lets you know the operation currently being performed. When the build is complete, Xcode displays the result of the build—namely, whether the build succeeded or failed and whether there were any errors or warnings—on the right side of the status bar.

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 Tasks in Xcode" in *Xcode Workspace Guide*.

In addition, Xcode displays a progress indicator that 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.

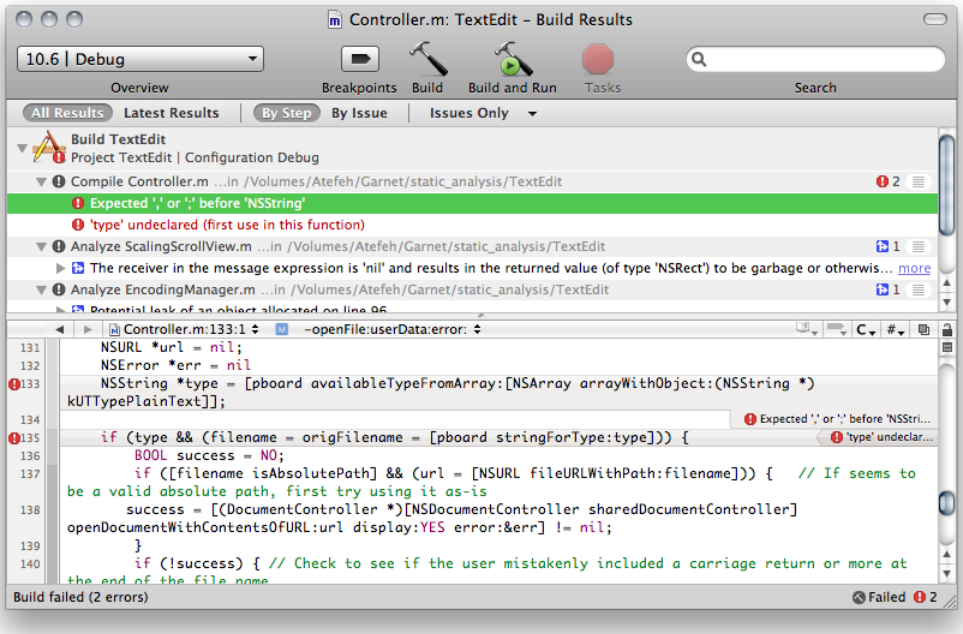
## Viewing Build Results

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 viewer, choose Build > Build Results

Figure 8-7 shows the Build Results window.

**Figure 8-7** Build Results window




**Note:** In the all-in-one project window layout, the build results appear in the Build pane of the project window.

These are the major user-interface elements of the Build Results window:

- **Build results toolbar.** The build results toolbar lets you specify the results you want to view.
- **All Results button.** Displays the results generated since the last clean was performed. Xcode remembers the results of multiple builds, which allows you to gauge your progress solving issues across several builds.
- **Latest Results button.** Displays the build results of the last build.
- **By Step button.** Displays the results by build operation. For example, each source-file compilation is a build step.
- **By Issue button.** Displays the results grouped by issue type when there are problems with the build, such as build errors and warnings, or static analyzer messages.
- **Messages menu.** The pop-up menu in the third section of the toolbar lets you filter the build results. Its options are:
  - **All Messages.** Displays all build results, which includes status messages, errors, warnings, and static-analyzer messages.
  - **Issues Only.** Displays errors, warnings, and static-analyzer messages.
  - **Errors & Warnings Only.** Displays errors and warnings.
  - **Errors Only.** Displays errors.
  - **Analyzer Results Only.** Displays static-analyzer messages.

- **Build results.** The build results pane build messages from the build system.
- **Editor pane.** The editor pane displays the file corresponding to the selected message in the build results pane.

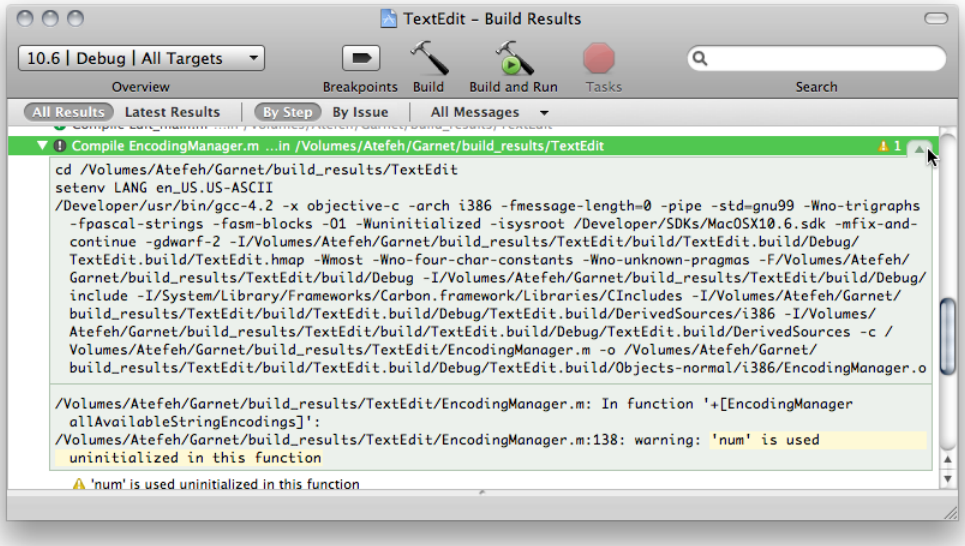
To navigate through warning and error messages even when the Build Results window is hidden, use these commands:

- Build > Next Build Warning or Error
- Build > Previous Build Warning or Error

These commands select the next/previous error or warning in the Build Results window. They do not, however, bring the Build Results window forward if it is hidden. If you are working in the text editor, using these commands selects the line at which the error or warning occurred and opens the related file if it is not already open.

When viewing build results by step, you can display a step's transcript by clicking the Show Transcript button (the button containing thin horizontal lines) on the right side of the build step. Figure 8-8 shows the transcript of a build step. To hide the transcript, click the Hide Transcript button (the button containing a triangle pointing up).

**Figure 8-8** Displaying a build-step transcript in the Build Results window




You can copy build transcripts to text files (such as an email messages) for further analysis:

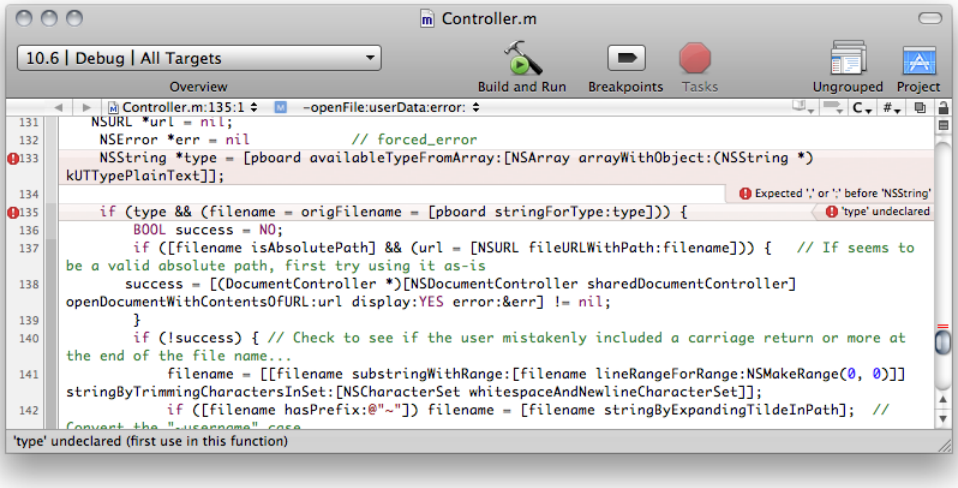
- To copy a build-step transcript, select the build step and drag it to the desired location. Dragging a build step to the Desktop creates a text clipping.
- To copy the entire build transcript, choose Edit > Select All and drag the selection to the desired location.

Instead of dragging, you can use the Copy and Paste commands in the Edit menu to copy the transcript to the desired location.

## Viewing Build Messages in the Text Editor

If you start a build from a text editor window, Xcode displays errors and warnings for the edited file directly in the editor. The text editor displays build messages as message bubbles (see "Viewing Project Messages" in *Xcode Workspace Guide*) next to the source code line that caused the problem, as shown in Figure 8-9.

**Figure 8-9** Text editor displaying build error messages




To indicate whether to display build messages in the text editor and to specify the type of messages to display, use the View > Message Bubbles menu.

# Building with xcodebuild

In addition to building your product from 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 `xcodebuild`, you can create a script that automatically builds your product at a specific time.

The `xcodebuild` tool reads your `.xcodeproj` project package and uses the target information it finds there to build a product.

To build a target using `xcodebuild`, use the `cd` command to change to your project's directory and invoke the `xcodebuild` command with the appropriate options. The project's directory contains your project's `.xcodeproj` bundle. For example, if your project is in `~me/Projects/MyProj`, enter `cd ~me/Projects/MyProj`.

## Building Projects Created with Early Versions of Xcode

The project file format, and the extension used for the project bundle, changed for Xcode 2.1. `xcodebuild` can build projects created by Xcode 2.0 or earlier (identified by the `.xcode` project bundle extension), as well as projects that have been upgraded. If the project directory contains both a `.xcodeproj` bundle and a `.xcode` bundle, `xcodebuild` uses the `.xcodeproj` bundle by default (to override this behavior use the `xcodebuild -project` option). If no `.xcodeproj` project package exists, `xcodebuild` opens the `.xcode` project package and creates an upgraded copy of it in memory. It uses this copy to build the project. The upgraded copy is not saved back to the file system.

## Building with the xcodebuild Tool Versus with Xcode

When you build within Xcode, it uses the active target and build configuration; `xcodebuild` uses the first target in the project's target list unless you specify a target (`xcodebuild -target`). You can specify which build configuration to use with the `-configuration` option. If you do not specify a build configuration, `xcodebuild` uses the target's default build configuration.

If you run `xcodebuild` as the root user, the preferences you set in Xcode preferences under another computer user 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).

# Building in Parallel

Normally the build system builds a target's dependencies sequentially; such builds are called sequential target builds). That is, each dependency's build process is completed before starting the build process for another dependency.

To shorten the build process, the build system can take advantage of parallel target builds. With parallel target builds, the build system builds target dependencies in parallel instead of one at a time.

The only build phase that the build system carries out in parallel is Compile Sources. All other build phases are performed linearly. This means that before and after the Compile Sources build phase, the build system processes a target's dependencies sequentially. To learn more about the order in which build phases are processed, see "Build Phase Processing Order" in *Xcode Build System Guide*.

You can activate parallel target builds for a project in the Project Info window. In the General pane, select "Build independent targets in parallel."

- Project editor > General > Build independent targets in parallel

**Important:** Before activating parallel target builds for a project, ensure that the targets to be built in parallel do not have dependencies (direct or indirect) on each other. An example of an indirect dependency is a target that uses a source file generated during the build process of another target. With sequential target builds, this hidden dependency may be avoided by building the target that generates the source file first. However, with parallel target builds, the order in which the targets are built is undetermined, which may produce successful and unsuccessful builds of the same target with no changes made to it.

# Building for Release

A product built for release to customers differs from a product built for debugging and development purposes. For example, you typically want your release product to be more highly optimized and to have unneeded symbols stripped to reduce code size. To build a product suited for deployment and install that product in its final destination path, there are a handful of build settings that you need to set before you build. These build settings are listed in Table 8-1.

**Note:** Many of these build settings are set in the default Release build configuration provided by Xcode, as described in "Predefined Build Configurations" in *Xcode Build System Guide*.

**Table 8-1** Build settings for installation builds

| Build setting | Value |
| --- | --- |
| Installation Build Products Location (`DSTROOT`) | Identifies the distribution root at which the build system places products built during an installation build. |
| Installation Directory (`INSTALL_PATH`) | Identifies the location under the distribution root at which the build system places products during an installation build. |
| Deployment Postprocessing (`DEPLOYMENT_POSTPROCESSING`) | Specifies whether the build system performs a number of steps designed to prepare the executable for release. These steps are described later in this section. |
| Deployment Location (`DEPLOYMENT_LOCATION`) | Specifies whether the build system places the product in the location specified by the `DSTROOT` and `INSTALL_PATH` build settings. |

To build a release version of a product and install it, you can do either of the following:

- Use `xcodebuild` from the command line with the `install` option.
- In the appropriate build configuration in the Xcode application, turn on the Deployment Location and Deployment Postprocessing build settings, set Installation Build Products Location to `/`, and set Installation Directory to the directory where you want to place the built product.

For example, to install a framework in `/Library/Frameworks`, configure the build settings as shown in Table 8-2.

**Table 8-2** Build settings for installing a framework in the local domain

| Build setting name | Build setting specification |
|---|---|
| DSTROOT | / |
| INSTALL_PATH | $(USER_LIBRARY_DIR)/Frameworks |
| DEPLOYMENT_LOCATION | YES |

To build and install this framework in its final destination path using `xcodebuild install`, enter the following on the command line:

```
% sudo xcodebuild install -configuration Release DSTROOT=/
INSTALL_PATH=/Library/Frameworks DEPLOYMENT_LOCATION=YES
```

**Note:** `INSTALL_PATH` must be set to a valid path to install in the distribution root. If `INSTALL_PATH` is not set or if it is set to the empty string, the build system assumes that the product is not intended to be installed and sets `TARGET_BUILD_DIR` to `$(TARGET_TEMP_DIR)/UninstalledProducts`.

**Warning:** In most cases, setting `DSTROOT` to `/` is not appropriate because any build problems may damage your system. It's safer to set `DSTROOT` to a temporary location, such as `/tmp/MyRoot`, and copy the product to its final destination only after it has been successfully built.

When you use `xcodebuild` with the `install` action, `xcodebuild` automatically sets `DEPLOYMENT_POSTPROCESSING` to `YES`. In the Xcode application, you must turn on Deployment Postprocessing yourself. When Deployment Postprocessing is active, the following operations occur:

- If Strip Linked Product (`STRIP_INSTALLED_PRODUCT`) and Use Separate Strip (`SEPARATE_STRIP`) are turned on, the build system strips the binary using the `strip` tool.

  You can use the Additional Strip Flags (`STRIPFLAGS`) build setting to specify which flags to pass to `strip`. By default, the build system passes the `-S` flag for libraries and frameworks, and the `-s` flag for applications and tools. If, for example, your application loads plugins that access symbols directly from the application, you may want it to be stripped with `-S`. To do so, set Additional Strip Flags in the target to `-S`. Stripping is not done for noninstallation builds.

  To specify whether executable files copied by a Copy File build phase are stripped, use the Strip Debug Symbols From Binary (`COPY_PHASE_STRIP`) build setting. By default, this build setting is turned on.

  For more information on the build settings described here, see *Xcode Build Setting Reference*. To learn more about controlling exported symbols, see "Reducing the Number of Exported Symbols" in *Xcode Build System Guide*. To learn more about using dead-code stripping in Xcode, see "Dead-Code Stripping" in *Xcode Build System Guide*.

- The Copy Files and Run Script build phases that are marked, respectively, "Copy only when installing" and "Run script only when installing," are run. To learn more about the Copy Files and Run Script build phases, see "Copy Files Build Phase" in *Xcode Build System Guide* and "Run Script Build Phase" in *Xcode Build System Guide*.
- Permissions are set for the product. The owner, group, and mode of the produced files are set according to the value of the Install Permissions (`INSTALL_MODE_FLAG`) build setting.

# Building for Debugging

Before you can take advantage of the source-level debugger, the compiler must collect information for the debugger. To generate debugging symbols for a product, use the Debug build configuration. This configuration turns on build settings that prepare a product for debugging. For more information about building see Debugging Preferences.

# Building Universal Binaries

Xcode can create **universal binaries**, which are executable files that can contain code and data for more than one architecture. You can create a single binary file that runs on both PowerPC-based and Intel-based Macintosh computers. The Architectures (`ARCHS`) build setting lets you specify which architectures Xcode builds for.

Xcode compiles a target's source files for each architecture individually and creates a single binary file (a universal binary) from these input files. For more information on building universal binaries, see *Universal Binary Programming Guidelines, Second Edition*.

To define build settings for particular architectures, you can use conditional build settings. See "Conditional Build Settings" in *Xcode Build System Guide* to learn more about this facility.

# Building for Multiple Releases of an Operating System

You can develop software that can be deployed on, and take advantage of features from, different versions of iOS or Mac OS X, including versions different from the one you are developing for. This capability is known as **cross-development**.

In order to take advantage of cross-development, you must install the Mac OS X SDKs for the OS versions you plan on targeting. Then, in your projects, specify which SDK to use when building your products. You can also specify the earliest Mac OS X release on which the software must run.

In some cases, Apple distributes an SDK for an upcoming version of the operating system as a seed, allowing you to prepare your application to work with future versions of the Mac OS X before they have been released to the general public.

**Important:** Cross-development in Xcode requires native targets.

To set up your Xcode project to target multiple Mac OS X releases, take the following steps:

1. **Choose an SDK.** Select your project in the Groups & Files list and choose File > Get Info. In the General pane of the Project Info window, choose the SDK from the Base SDK for All Configurations 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 the OS, and links against the stub libraries in that SDK. This allows you to build products on your development computer that can be run on the OS release targeted by the SDK. Your software can use features available in system versions up to and including the one you select.

You can also specify the Base SDK through the Base SDK (`SDKROOT`) build setting.

2. **Choose a deployment version Mac OS X.** 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 the OS corresponding to the SDK version.

   To set the deployment version for a target:

   a. Select the target in the Groups & Files list and open an Info window. Click Build to open the Build pane.

   b. Find the Deployment Target (`MACOSX_DEPLOYMENT_TARGET`) or iPhone OS Deployment Target (`IPHONEOS_DEPLOYMENT_TARGET`) build setting (depending on which platform you're developing for) and choose a deployment operating system from the pop-up menu in the Value column.

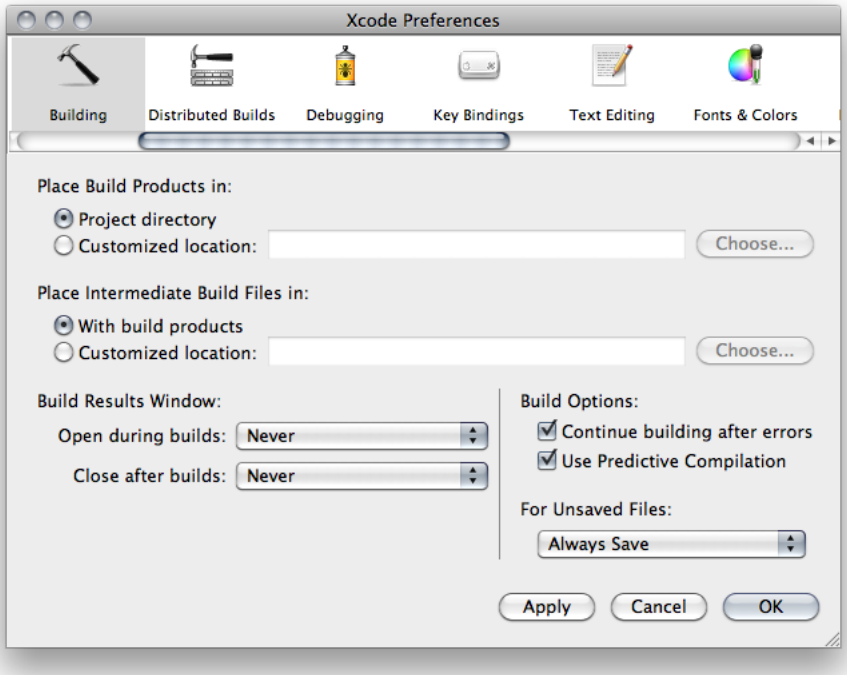
3. **For each target, supply a prefix file that takes into account the selected SDK.** To use an umbrella framework header from an SDK as your prefix file, add the appropriate `#include <Framework/Framework.h>` directive to your target's prefix file instead of setting a Prefix Header path to the umbrella framework header directly.

There is a lot more to successfully developing software for multiple versions of the Mac OS. For more information see *SDK Compatibility Guide*.

# Building Preferences

The Building pane of Xcode preferences allows you to set default project build locations and the behavior of the Build Results window and the Errors and Warnings smart group. Figure 8-10 shows the Building preferences pane.

**Figure 8-10** Building preferences pane




Here is what the pane contains:

- **Place Build Products In.** Specifies the location at which Xcode places the output generated when you build a target—such as an application or tool—for all new projects that you create.
  - **Project directory:** Places the product that is generated in the build folder in your project directory.
  - **Customized location:** Places build products in the folder specified in the associated text field. Use this option to specify a different folder for build products. 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 override this setting for individual projects. See "Build Locations" (page 63) for details.
- **Place Intermediate Files In.** Specifies 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.
  - **With build products.** 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. This is the default value.
  - **Customized location.** 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 override this setting for individual projects. See "Build Locations" (page 63) for details.
- **Build Results Window.** These pop-up menus specify when Xcode opens the Build Results window or pane:

- **Open during builds.** Specifies when (or whether) Xcode automatically opens the Build Results window or pane when you build a target.
- **Close after builds.** Specifies when (or whether) Xcode automatically closes the Build Results window or pane after a build completes.

To learn more about the Build Results window, see "Viewing Build Results" (page 78).

- **Build Options.** These settings let you specify 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.** Activates predictive compilation. Predictive compilation shortens build time by beginning to compile source files in the background, even before you initiate a build. See "Predictive Compilation" in *Xcode Build System Guide* for details.

- **For Unsaved Files.** This menu determines what Xcode does with files that have unsaved changes when you start a build. You can specify whether to ask before saving, automatically save, never save, or cancel the build.

# Analyzing Code

As you write your code, you may inadvertently leave in flaws that, if not discovered early, may produce costly, hard-to-find bugs. A tool that helps you identify such problems early would not only save you time and effort, but would also help you become a better programmer. Such a tool exists in the Xcode static analyzer.

**Software requirements:** The Xcode static analyzer is available in Xcode 3.2 and later.

This chapter describes how to use the static analyzer to identify and fix code flaws.

## Static Analysis Overview

The source code you write might contain flaws that may manifest themselves as bugs during testing or deployment of your product. Static analysis is a method for finding these potential bugs without running the corresponding executable; that is, before your product goes through testing and, more importantly, before it's released to customers.

Compilers, such as GCC, perform static analysis as part of the compilation process. They are great at ensuring adherence to a programming language's syntax and detecting some logic and data-type–usage errors. Many software bugs, however, occur only when a program takes a specific set of branches and loops. While compilers analyze the control flow of a program during compilation, they cannot find such problems because of practical limits on compilation time. Also, compilers typically lack domain-specific knowledge about correct API usage and the use of advanced memory-management techniques.

The source code you write must follow certain rules to ensure that your program runs correctly and uses resources appropriately. For example, in C-based languages you must initialize variables before using them; you must also deallocate the memory you allocate. Violations to these rules are known as "code flaws." Static analysis tools detect flaws in source code and describe their causes.

The Xcode **static analyzer** goes through your code to ensure that it follows logic and API-usage policies, including Cocoa memory-management rules. If the analyzer detects flaws in your code, it emits messages providing clear descriptions of their causes. In addition, the analyzer produces a diagnosis of the sequence of steps that would cause the bug to appear; these steps are known as bug's "flow path." Because you find these problems without having to run your binary, you get a chance at solving them at the time you're writing your code, when the information you need to address them is fresh in your mind. Early discovery also reduces the effort of solving bugs in your product because most problems don't reach the testing or deployment stage of your developing process, where it may require more effort to determine their cause.

The static analyzer checks for code flaws in these areas:

- **Logic.** Logic flaws include accessing uninitialized variables and dereferencing null pointers.
- **Memory management.** Memory management flaws include leaking allocated memory.
- **Dead stores.** Dead stores identify unused variables.

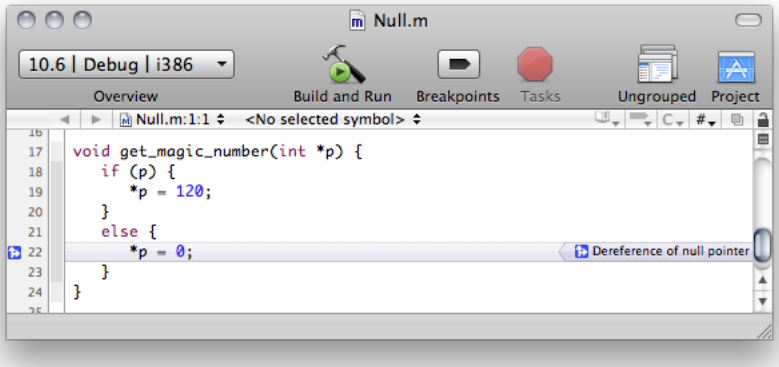
- **API usage.** API-usage flaws involve not following policies imposed by the frameworks and libraries your product uses.

For example, examine the following code listing:

```
#import <Foundation/Foundation.h>

void get_magic_number(int *p);

int main (int argc, const char * argv[]) {
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

    int *i = malloc(sizeof(int));
    get_magic_number(i);
    printf("Magic number: %i\n", *i);
    free(i);

    [pool drain];
    return 0;
}

void get_magic_number(int *p) {
    if (p) {
        *p = 120;
    }
    else {
        *p = 0;
    }
}
```

This program compiles without errors and runs cleanly when `p` is not null. However, when `p` is null, the program crashes when it dereferences a null pointer. Although the source code adheres to Objective-C syntax and correctness rules, it contains a logic error uncaught by the compiler. The static analyzer, however, detects the logic error and emits a descriptive message, as shown in Figure 9-1.

**Figure 9-1** Dereferencing a null pointer



The Xcode static analyzer is a great complement to other techniques for finding bugs in software, such as unit testing. You should always use these in conjunction with the static analyzer to ensure your code contains as few bugs as possible before shipping the final product to your customers.

The following are three important issues with static analysis:

- **Static analysis doesn't find every code flaw.** Just like most software, static analysis tools go through constant improvement. Therefore, it may not find all flaws in your code. You shouldn't rely on static analysis alone to ensure that your code is free of problems.
- **Static analysis produces false positives.** False positives are unlikely problems in source code that the analyzer identifies as problems. Source-code annotations help reduce false positives.
- **Static analysis increases build time.** Static analysis takes more time than compilation because of the deep code analyses and policy adherence checks performed. Using a build configuration tailored for static analysis lets you perform static analysis at times when fast build times are not critical.

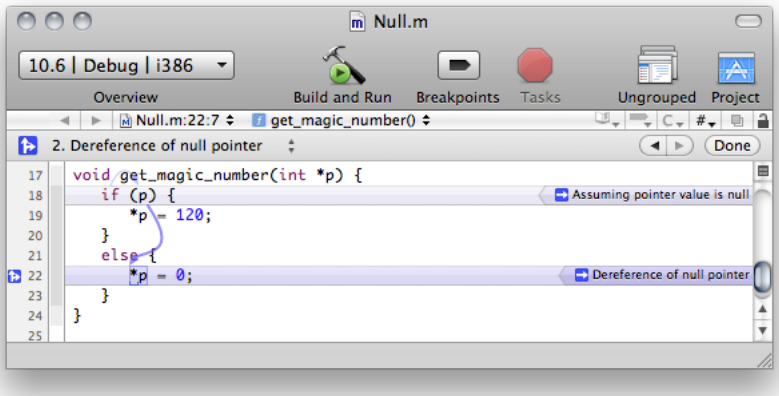
The static analyzer is based on the Clang static analysis engine. For more information, visit http://clang-analyzer.llvm.org.

# Static Analysis Workflow

To perform static analysis on the current project, choose Build > Build and Analyze.

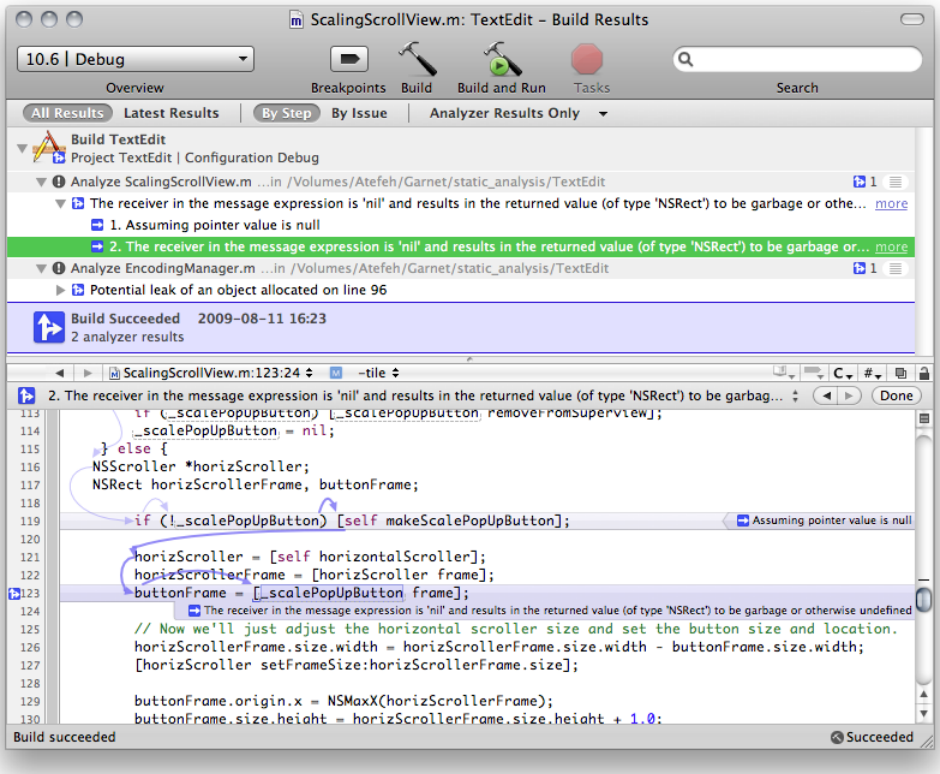
If you routinely analyze your code, each analysis should return very few errors, which you should be able to correct easily. When you perform static analysis for the first time on a large project, you may encounter many analyzer messages.

You should perform static analysis on your code frequently, for example after making a few changes to a source file. Xcode emits analyzer messages in the same window that contains your source file.



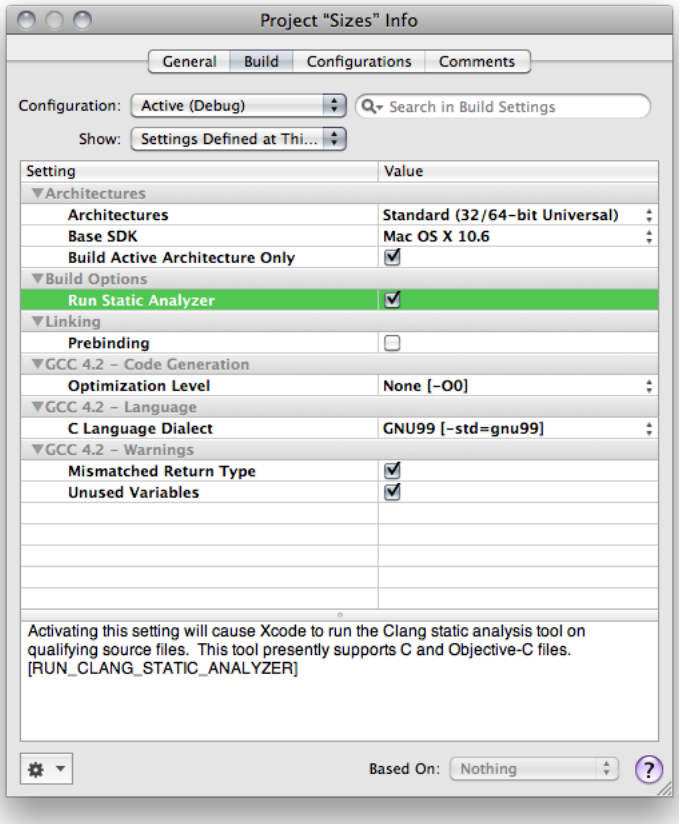

You can fix the code flaws without opening additional windows. After making the corrections, build and analyze your code again to confirm that you fixed the problem.

When you build and analyze a large project, the analyzer may emit several analyzer messages. To view all the analyzer messages for a project, use the Build Results window. In this window, you can navigate the analyzer messages by build step (source file) or by analyzer issue. This window also contains an editor pane in which you can fix each analyzer each.




In deciding how often to perform static analysis consider the size of your project, the scope and number of changes you're making, and whether you work on a team. For example, if you are working on a team, you may want to analyze your code before committing your work to the team's SCM repository.

You can also ensure that Xcode analyzes your code every time you build your product by turning on the Run Static Analyzer build setting.




Because code assertions are active in the Debug build configuration, you should analyze your code in this configuration or a configuration copied from it. Assertions indicate to the analyzer the assumptions you've made about your code. These assertions help suppress extraneous or undesired analyzer messages. For more information, see "Suppressing Static Analyzer Messages" (page 99).

When you make static analysis a regular part of your development process, you may consider creating a specialized Analyze build configuration for static analysis. Although you can turn on the Run Static Analyzer build setting in your Debug configuration, this can result in slower build times, which is contrary to the quick-turnaround goal for the Debug configuration. In the Analyze build configuration, in addition to turning on the Run Static Analyzer build setting, you should turn off the Build Active Architecture Only build setting so that the analyzer finds architecture-dependent problems (see "API-Usage Checks" (page 99) for more information). This model lets you easily switch between performing classic regular debugging tasks or static-analysis tasks.

To create the Analyze build configuration:

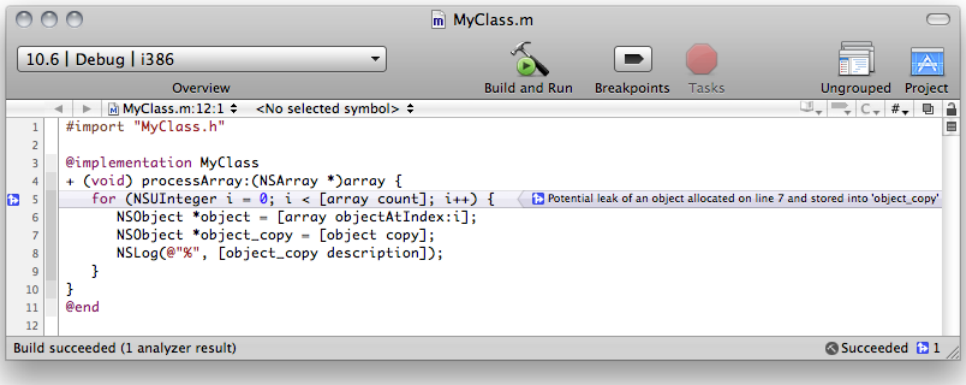
1. Open the Project Info window and click Configurations.
2. In the configuration list, select the Debug configuration.
3. Click Duplicate and name the new configuration "Analyze".
4. In the Build pane, choose Analyze from the Configuration pop-up menu.

# Interpreting Static Analyzer Messages

When you execute the Build and Analyze command, the static analyzer goes through your project source files in search of code flaws (violations of programming rules and policies). Xcode displays messages for each violation in the current source file in the text editor that contains it. You can also view code flaws for the entire build in the Build Results window (see "Viewing Static Analyzer Results" (page 96) for details.

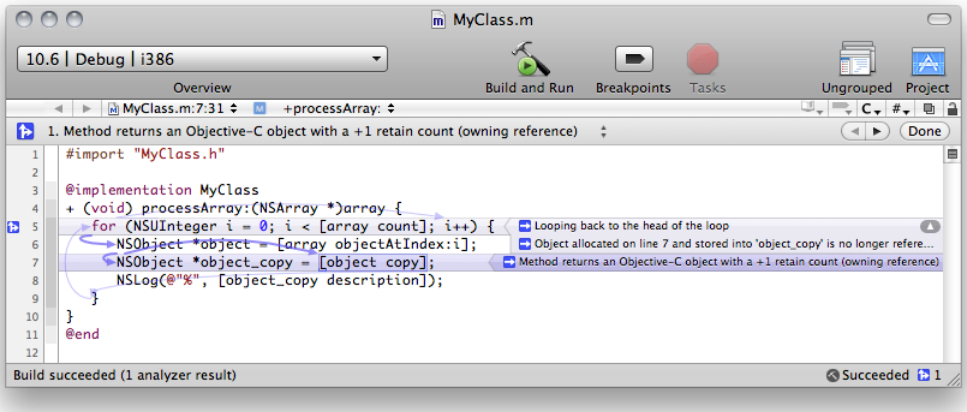
At first, the static analyzer shows a single message per code flaw in the editor, as Figure 9-2 shows. The message appears in the line at which the analyzer determines that a violation occurs. In this case, the violation is a memory leak at the head of the loop.

**Figure 9-2** Memory-management violation




When you click the message, Xcode identifies the flaw's flow path using blue arrows, as Figure 9-3 shows. Clicking the flaw message also displays the **analyzer toolbar** in the editor. This toolbar contains a pop-up menu with the analyzer messages for each step of the flow path. Choosing an item from this menu highlights the step in the editor. You can also move between steps using the analyzer toolbar's Left and Right buttons. Clicking the Done button dismisses the toolbar.

**Figure 9-3** Flow path of memory-management violation



For this code flaw, the analyzer took the flow path described in Table 9-1.

**Table 9-1** The flow path of a code flaw

| Line # | Statement/expression | Analyzer message |
|---|---|---|
| 7 | `NSObject *object_copy = [object copy]` | 1. Method returns an Objective-C object with a +1 retain count (owning reference). |
| 5 | `for` | 2. Looping back to the head of the loop. |
| 5 | `i++` | 3. Object allocated on line 8 and stored into 'object_copy' is no longer referenced after this point and has a retain count of +1 (object leaked). |

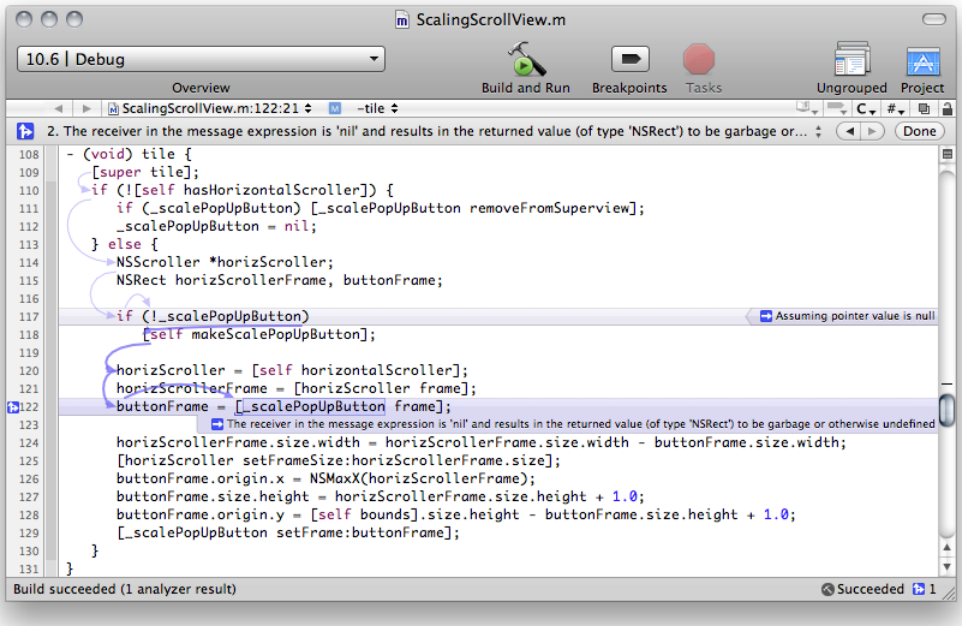
The flow path and the analyzer messages at each step of the path provide a detailed description of the events that unearthed the problem. In this case, after allocating an object inside the loop owned by the current scope, the sole reference to the object is lost after going back to the head of the loop.

As you highlight each step of the flow path, Xcode displays arrows that identify the statements the analyzer performed before arriving at the step. For step 1 of the flow path (creating a copy of `object` in line 8), the analyzer first executed the `for` statement in line 6, and the assignment to `object_copy` in line 7.

To fix this violation all that's needed is to add `[object_copy release];` to the bottom of the loop.

The static analyzer can also discover logic errors by by following possible code paths and making assumptions using information in the code. For example, in Figure 9-4, the static analyzer discovers a violation by taking the two possible code paths based on whether a variable is null: When it follows the code path taken when `_scalePopUpButton` is null, it finds that evaluating the expression `[_scalePopUpButton frame]` results in a message to a null object, which, in this case, returns garbage.

**Figure 9-4** Logic violation

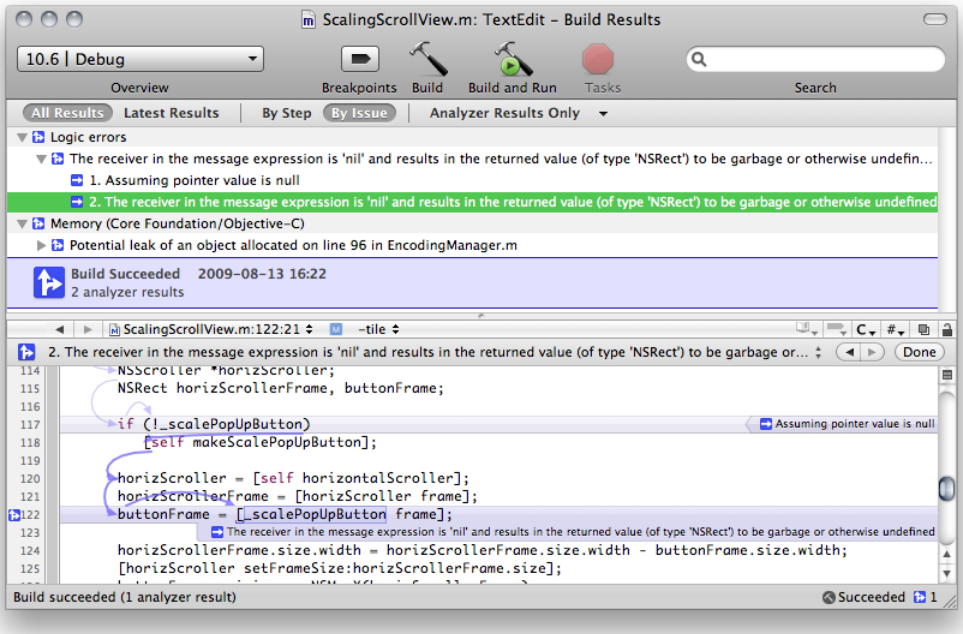



# Viewing Static Analyzer Results

The Build Results window displays the results of a build, including static analysis results. When you're performing static analysis, it may be useful to display only analyzer messages in the window.

To display the Build Results window, choose Build > Build Results. To show only analyzer results, choose Analyzer Results Only from the pop-up menu in the window's toolbar. To group analyzer results by violation type, click the By Issue button in the toolbar. Figure 9-5 shows the Build Results window displaying two code flaws in a project.

**Figure 9-5** Build Results window showing analyzer results grouped by violation type




You can correct code flaws in the window's editor pane. For details about working with this pane, see "Interpreting Static Analyzer Messages" (page 94).

# Specialized Static Analyzer Checks

In addition to logic checks, the static analyzer checks that your code follow the Cocoa and Core Foundation memory-management rules and that is uses their API correctly. This section examines how the analyzer detects memory-management flaws in your code—whether it runs under garbage collection—and hard-to-detect API-usage flaws.

## Memory-Management Checks

In Cocoa there are two memory-management models, one using reference counting and another using automatic garbage collection. In the reference-counting model, Cocoa—through API naming conventions and additional method calls by client code—maintains a count of entities that claim to own an object.

In the Cocoa reference-counting model, when your code obtains an object through a method whose name starts with `alloc` or `new`, or contains `copy`, your code inherently has ownership of the object. When your code receives an object from a method that doesn't return an object owned by the caller, you claim ownership of the object by invoking its `retain` method. At some point, the owner of the object (or a proxy for the owner) must relinquish ownership by invoking its `release` method. Otherwise, your code leaks the object and the memory it uses. A number of such leaks can cause your application and, potentially, the operation system to become memory starved. The static analyzer checks that your code releases the objects that it owns to avoid memory leaks.

**Note:** The static analyzer's ability to detect memory leaks does not preclude you from performing leak checking on the running applications, using tools such as Instruments.

Because Cocoa itself follows the Cocoa API conventions for memory management, the static analyzer knows whether the objects the framework returns are owned by your (the calling) code. To ensure that the analyzer correctly attributes ownership to the objects your methods return, your method names should follow the Cocoa API naming conventions.

In cases where it would be inconvenient to rename you methods, you can use source-code annotations (in the form of macros) that specify that the object a method returns is owned by the caller. These macros are `NS_RETURNS_RETAINED` and `CF_RETURNS_RETAINED` for Cocoa-based and Core Foundation–based code, respectively. Listing 9-1 shows how to use the `NS_RETURNS_RETAINED` macro in the declaration of a method that returns a caller-owned object but whose name doesn't follow the Cocoa naming conventions for methods that return caller-owned objects, described earlier.

**Listing 9-1** Using the `NS_RETURNS_RETAINED` macro to attribute object ownership

```
- (id) obtainAnObject:(NSString *)objectID        NS_RETURNS_RETAINED;
```

These macros are available in Mac OS X v10.6 and later. In earlier releases of Mac OS X and in iOS, you can define the macros yourself. _listing_ shows the code you add to your macro definitions to define the `NS_RETURNS_RETAINED` macro.

```
#ifndef __has_feature
#define __has_feature(x) 0     // Compatibility with non-clang compilers.
#endif

#ifndef NS_RETURNS_RETAINED
#if __has_feature(attribute_ns_returns_retained)
#define NS_RETURNS_RETAINED __attribute__((ns_returns_retained))
#else
#define NS_RETURNS_RETAINED
#endif
#endif
```

For more information about these macros, visit http://clang-analyzer.llvm.org/annotations. To learn more about reference-count–based memory management, see the following documents:

- *Memory Management Programming Guide*
- *Memory Management Programming Guide for Core Foundation*

When you use garbage collection in Cocoa, you don't need to worry about object ownership because the lifecycle of all Cocoa objects is managed by the garbage collector. That is, you don't need to call `retain` on objects your code needs to own, and call `release` on them to relinquish that ownership. However, if your code use Core Foundation objects, which are not automatically garbage collected, you need to make them collectable at creation time.

To make Core Foundation objects garbage collectable, call the `CFMakeCollectable` function immediately after creating them, as shown in _listing_.

```
CFStringRef myCFString =
    CFStringCreate...(...);                    // Incorrect: Causes memory leak
                                                  under GC.
```

```
CFStringRef myCFString =
    CFMakeCollectable(CFStringCreate...(...));  // Correct: Object is garbage collected
                                                   under GC.
```

For more information about garbage collection, see *Garbage Collection Programming Guide*.

## API-Usage Checks

The static analyzer verifies that your code follows Cocoa and Core Foundation programming policies and works correctly for the target architecture. Because the analyzer has deep knowledge of the Cocoa and Core Foundation API, it checks that your code doesn't contain flaws caused by incorrect data-type–size assumptions. For example, the code in Listing 9-2—when compiled for a 32-bit environment—reads 4 bytes after `&i` to return the `CFNumberRef` object. However, in a 64-bit environment the same function reads 8 bytes after `&i` to return the `CFNumberRef` object, which results in the returned object containing garbage.

**Listing 9-2** Analyzer message for an API-usage flaw

```
CFNumberRef sizes() {
    unsigned i = 10;
    CFNumberRef x =
        CFNumberCreate(0, kCFNumberLongType, &i);   => A 32-bit integer is used to
            initialize a CFNumber object that represents a 64-bit integer.
            32 bits of the CFNumber value will be garbage.
    return x;
}
```

One way to correct this flaw is to change the type of `i` to `unsigned long`.

The static analyzer performs many other API-usage checks on your code. Visit http://clang-analyzer.llvm.org to learn more.

# Suppressing Static Analyzer Messages

Although the static analyzer is a great tool for finding flaws in your code, it may produce unwanted warnings (or false positives), which are problems the analyzer identifies because it lacks information that would prevent it from flagging the code as flawed. This information includes assumptions you've made but have not codified.

To suppress false positives, you must add information to your code that makes clear your assumptions. To add this information you use assertions, attributes, or pragma directives.

The static analyzer follows the paths it thinks your code may follow at runtime. However, some of its assumed paths may not be possible. For example, the code in Listing 9-3 assumes that the loop is executed at least once.

**Listing 9-3** Analyzer message from a false flow path

```
void loop_at_least_once() {
    char *p = NULL;
    for (unsigned i = 0, n = iterations(); i < n; i++) {
        p = get_buffer(i);
    }
```

```
    *p = 1;                                 => Dereference of null pointer
}
```

Because the analyzer analyses one method as a single entity (it doesn't analyze called methods), it has limited information about the value of `n`; therefore, it has to consider the case where `n = 0`. To suppress the "Dereference of a null pointer" analyzer message, the code must contain an assertion that `p` cannot be null after the loop, as shown in Listing 9-4.

**Listing 9-4** Suppressing a false analyzer flow path

```
void loop_at_least_once() {
    char *p = NULL;
    for (unsigned i = 0, n = iterations(); i < n; i++) {
        p = get_buffer(i);
    }
    assert(p != NULL);
    *p = 1;
}
```

**Note:** If you use custom assertion handlers, you must let the analyzer know about them using the `noreturn` and `analyzer_noreturn` attributes. For details, visit http://clang-analyzer.llvm.org/annotations.

Dead stores are a category of dead-code bugs: A value stored in a variable is never read. To have the analyzer ignore dead stores, use pragma directives or API attributes. Listing 9-5, Listing 9-6, and Listing 9-7 show a dead store, and how to use the `unused` API attribute to suppress the "Value stored to 'x' during its initialization is never read" message.

**Listing 9-5** A dead-store message

```
int unused(int z) {
    int x = foo();                  => Value stored to 'x' during its
initialization is never read
    int y = 6;
    return y * z;
}
```

**Listing 9-6** Suppressing a dead-store message with the `#pragma (unused)` directive

```
int unused(int z) {
    int x = foo();
    int y = 6;
    return y * z;
    #pragma unused(x)
}
```

**Listing 9-7** Suppressing a dead-store message with the `__attribute__((unused))` API attribute

```
int unused(int z) {
    int x __attribute__((unused));
    x = foo();
    int y = 6;
    return y * z;
}
```

# Defining Executable Environments

An **executable environment** defines how a product is executed when you run it from Xcode. The executable environment tells Xcode which program to launch when you run the product, 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. You can also create your own executable environments for testing products such as plug-ins or frameworks for which you don't have the corresponding Xcode projects. You can 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 Environment Overview

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 create executable environments; in most cases, Xcode does this for you. 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 the product.

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 executable environment. This custom executable environment specifies the program to launch when you run the product, such as the application that uses your plug-in.

Even if your target creates a product that can run on its own, you may also want to customize the executable environment associated with it, in order to pass arguments to the executable or test it with environment variables.

Executable environments defined for a project are placed in the Executables group in the Groups & Files list in the project window. Executable environments that you define are stored in your user file for the project—that is, in the `.pbxuser` file in the project package (see "The Project Directory" (page 18) for details). As a result, each developer working on a project defines her or his own executable environments. When you run a product in Xcode, Xcode launches the program specified by the active executable, as described in the "Setting the Active Executable" (page 102).

# Setting the Active Executable

The **active executable** is the executable environment that Xcode uses when you run a product. 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 target's executable the active executable. 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 run.

The active executable is indicated by the blue (selected) button in the detail view. To change the active executable, do either of the following:

- Choose the desired executable environment from the Set Active Executable submenu of the Project menu.
- Select Executables in the Groups & Files list and select the desired executable environment in the detail view.

# Creating Custom Executable Environments

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 the Executables group that points to the target's product, and it knows to use that executable environment when you run the target.

Sometimes, though, you have a product that can't run by itself: for example, 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 specific command-line arguments. 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.

You can also use custom executable environments to debug programs in Xcode, even if you do not have an Xcode project for the program. For example, you may have a program that you did not build in Xcode, or a program that was built by another person, to which you do not have the source. To run this program in the debugger, you simply create an empty project and add one or more custom executable environments that are configured to launch your program.

To create a custom executable environment, choose Project > New Custom Executable.

Xcode displays a dialog in which you can specify characteristics for your executable environment:

- **Executable Name field.** Enter the name used to identify the executable environment in Xcode.
- **Executable Path field.** Enter the path to the executable to launch, or click Choose and navigate to the executable file or application bundle.
- **The Add To Project pop-up menu.** Choose (among the currently opened projects) the project to which Xcode adds the custom executable.

When you click Finish, Xcode adds the new executable environment to the chosen project. You can change the program that Xcode launches when using this executable environment—along with other executable settings—in the Executable Info window, as described in "Configuring Executable Environments" (page 103).

# Configuring Executable Environments

Executable environments give you control over how your product is run when you launch it from Xcode.

You configure executable environments using the Executable Info window. To open the Executable Info window:

1. Select the executable environment in the Groups & Files list or in the detail view.
2. Open the Executable Info window by choosing File > Get Info.

**Alternative:** You can open the Executable Info window by double-clicking the executable environment in the Groups & Files list or in the detail view.

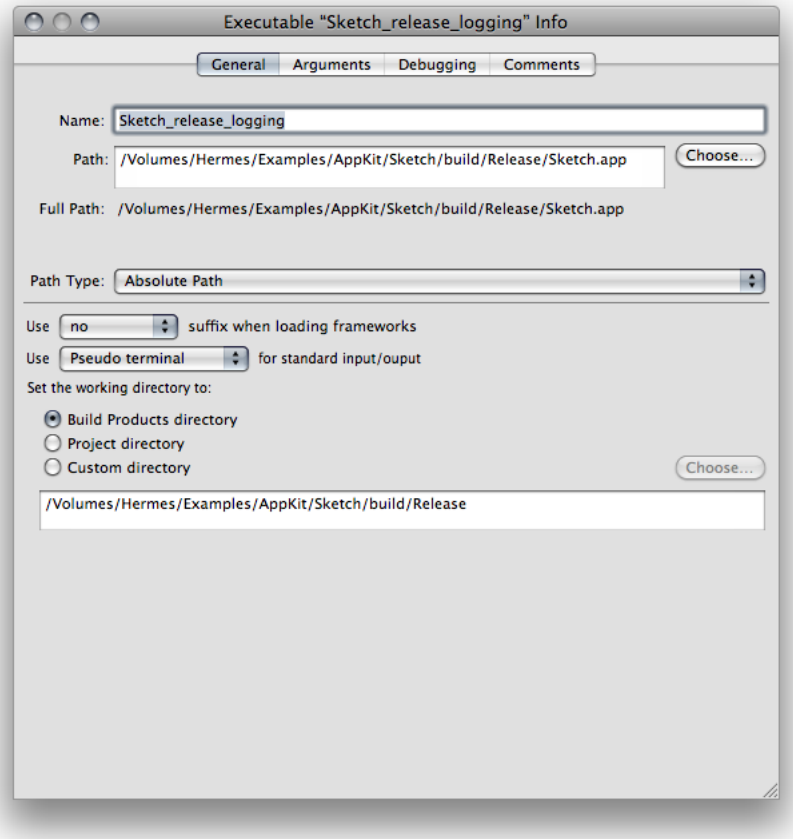
The following sections describe the settings you can configure in the Executable Info window.

- View and edit basic information about the executable environment, for example, you can view or edit the executable you are using or the working directory.
- Specify arguments to pass to the executable on launch, as well as any environment variables for Xcode to set before launching the executable.
- Specify which debugger to use when debugging the executable, as well as a number of other debugging options.
- Associate notes or other arbitrary text with the executable.

## Executable-Environment General Settings

The General pane of the Executable Info window (Figure 10-1) lets you configure essential aspects of an executable environment.

**Figure 10-1** General pane of the Executable Info window




These are the executable-environment aspects the General pane allows you to configure:

- **Name field.** The name of the executable environment.
- **Path field.** The path to the binary the executable environment runs.
- **Path Type pop-up menu.** Indicates whether *path* is an absolute path or a relative path (specifies the directory to which *Path* is relative).
- **Suffix pop-up menu.** Specifies whether to load normal, debugging, or profiling builds of the frameworks the product uses.
- **Standard input/output pop-up menu.** Specifies the device the product uses for standard input and output.
- **Set the working directory to.** Specifies the product's working directory when running. See "Build Locations" (page 63) for more information.

**Executable-environment paths:** Executable-environment settings such as *path* and *working directory* refer to specific locations in your file system. If you change the project's build directory, you may break the project's executable environments. To create more flexible executable environments, you can use build settings to specify executable-environment settings in relation to a project's base locations. For example, if you create an executable environment and specify the executable path as `$(CONFIGURATION_BUILD_DIR)/Sketch.app`, you can later change the location of the project's build directory without breaking this executable environment. See *Xcode Build Setting Reference* for more on build settings that identify a build and product locations.

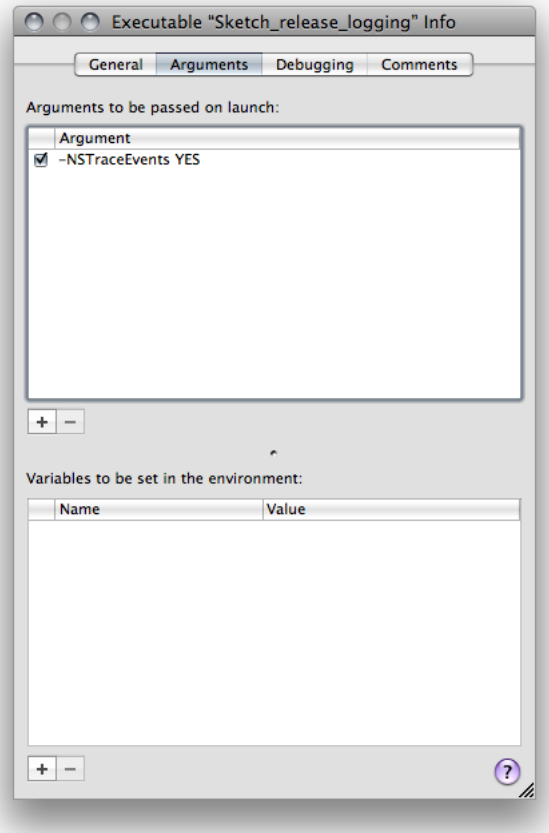
## Executable-Environment Arguments

If your product takes command-line arguments, you can define those arguments in an executable environment. Xcode passes those arguments to the product's binary when you run the product.

To test your product 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.

You specify arguments to pass to the binary, as well as environment variables that the executable environment sets before launching the binary, in the Arguments pane of the Executable Info window (Figure 10-2).

**Figure 10-2** Arguments pane of the Executable Info window



These are the executable-environment aspects the Arguments pane allows you to configure:

**"Arguments to be passed on launch" table.** Defines command-line arguments. Individual arguments can be active or inactive, facilitating the testing of particular combinations of arguments. To reorder the arguments list, drag the argument line to its new location in the list.

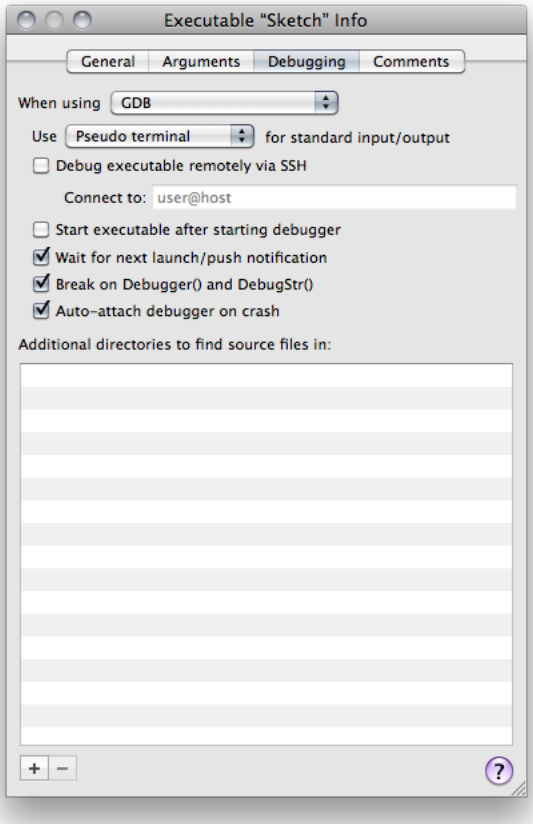
**"Variables to be set in the environment" table.** Defines environment variables for the running binary. Individual variables can be active or inactive. You can access the active variables during a run with `getenv`.

To learn more about the directories used in the build process, see "Build Locations" (page 63). For more information on these build settings, see *Xcode Build Setting Reference*.

## Executable-Environment Debugging Information

Executable environments specify a set of debugging-specific items that specify which debugger to use, as well as how Xcode communicates with the debugger. (You debug a product when you start it with breakpoints activated.) You configure these items in the Debugging pane of the Executable Info window, shown in Figure 10-3.

**Figure 10-3** Debugging pane of the Executable Info window




These are the executable-environment aspects the Arguments pane allows you to configure:

- **"When using" pop-up menu.** Specifies the debugger to use when running with breakpoints.
- **"Standard input/output" pop-up menu.** Specifies the device the product uses for debug input and output.
- **Debug executable remotely via SSH.** Activates remote debugging using SSH. See Debugging Programs Remotely for more information.
    - **Connect to:** Specifies the host computer on which the binary runs.
- **Start executable after starting debugger.** Specifies whether the executable environment starts the binary immediately after loading it in the debugger. If so, Xcode loads the binary in the debugger but does not start it until you restart. This feature lets you perform debugging operations—such as setting breakpoints—before the binary runs. You may also use this option to attach to a running program.
- **Wait for next launch/push notification.** Specifies whether to attach to the binary when it launches. This is particularly useful when debugging push notifications. For more information, see *Local and Push Notification Programming Guide*.
- **Break on Debugger() and DebugStr().** Tells Xcode to set the `USERBREAK` environment variable, which suspends execution of the binary on calls to the Core Services framework debugging functions `Debugger` and `DebugStr`.
- **Auto-attach debugger on crash.** Tells Xcode to try to attach to the binary when it crashes.
- **Additional directories to find source files in.** Lists additional directories containing source files corresponding to the symbol information for the binary.

  If you have the source code to the application, you can make that source code available to the debugger so you can see the source for variables and set breakpoints. This operation makes your source code available to the debugger, but does not give you any of the source code navigation features of Xcode or make the source code available to you to set breakpoints before the debugging sessions starts. For this type of access, you can add the source code directly to your project, as described in "Managing Files and Folders in a Project" (page 28).

# Running Programs

After you build a product, you can test it from Xcode. To run a product, perform these steps:

1. **Choose the appropriate build configuration.** Depending on how you want to test the product, you run a release or debug configuration of it. See Build Configurations for details.
2. **Build the product.** In the build process you may find problems that prevent Xcode from creating a binary to run. Xcode must successfully build the product before you can run it. See "Building with Xcode" (page 75) for more information.
3. **Run the product.** To run the product, do one of the following:
   - Choose Run > Run – Breakpoints Off.
   - Choose Run > Debug – Breakpoints On.
   - Choose Run > Run with Performance Tool.

   If none of these commands are available, try associating an executable with the selected target. See "Defining Executable Environments" (page 101).

In addition, after halting a program, you can restart it using the last run configuration by choosing Run > Go.

## Viewing Console Output

Many programs print messages to `stdout`, as well as logging debugging messages to the console or `stderr`. When you run your program in Xcode, you can see this output in the console window. In addition, if you are creating a command-line program that takes input from `stdin`, you can use the console to interact with your program. To open console, choose Run > Console.

## Halting a Program

In some unique cases, you may want to unconditionally stop your program's execution with a trap. Such an operation in programs launched from Xcode would cause the debugger to appear, pointing to the code line that caused the trap. In programs launched directly, the CrashReporter application would log a crash with stack traces for the program's threads.

Listing 11-1 shows the assembly-language instructions that stop a program.

**Listing 11-1** Halting a program with a trap instruction

```
asm {trap}            ; Stops a program running on PPC32 or PPC64.
```

```
__asm {int 3}          ; Stops a program running on IA-32.
```

# Document Revision History

This table describes the changes to *Xcode Project Management Guide*.

| Date | Notes |
| --- | --- |
| 2010-07-02 | Made minor corrections |
| 2010-05-27 | Updated information about the Overview toolbar menu. |
| 2010-03-24 | Made minor corrections and updated index. |
| 2009-10-19 | Made cosmetic changes. |
| 2009-09-09 | Added information about static analysis and single-file search. |
| | Added "Analyzing Code" (page 89). |
| | Added "Searching in a File" (page 37) and "Shortcuts for Finding Text and Symbol Definitions" (page 46). |
| | Added "Renaming a Project" (page 30). |
| | Updated "Viewing Build Results" (page 78). |
| 2009-05-27 | Added index. |
| 2009-01-06 | Made minor content changes. |
| 2008-11-19 | Made minor content addition and corrections. |
| | Added "Building Preferences" (page 86) (moved from *Xcode Workspace Guide*). |
| | Added instructions on negating build settings in files in "Per-File Compiler Flags" (page 70). |
| | Added cross-reference to software-product information in "Choosing a Project Template" (page 21). |
| 2008-10-15 | Fixed typos and incorporated feedback. |
| 2008-09-09 | Minor content reorganization. |
| 2008-07-08 | New document that provides practical descriptions of the major development tasks developers perform with Xcode. |
| | The content of this document was previously published in *Xcode User Guide*. |

Document Revision History

# Index

## F



## H



## I



## L



## M



## O



## P



## R



## S

## T



## U



## W



## X