Project Builder and Interface Builder

- Macintosh IDEs
- Using Project/Interface Builder
- Setting up CVS for Project Builder
- Configuring optimization for Project Builder
- Static code checking for Project Builder
The Macintosh has always supported application development with some very good development tools, Project Builder and Interface Builder among them. This chapter provides an overview of the Project Builder environment, its use, and covers common scenarios you will encounter when developing programs with Project Builder. Once you complete this chapter, you will be able to work the levers of Project/Interface Builder and will be able to get around in the environment.

3.1 Introduction

Traditionally, Macintosh development tools are centered on an Integrated Development Environment (IDE). An IDE is commonly composed of an integrated editor, compiler, linker, and debugger, all within one program. To develop a new application, you launch the IDE and create a new project file. The project file acts as a repository for all files that make up the project, including source files, libraries, and any support files. You write your program using the integrated editor, build the program by selecting a build command, and execute the program with a run command. Typically commands are accessible from menu items, and customization takes place through standard dialog boxes. Debugging a program is as simple as building the program in debugging mode and stepping through the program within the IDE. When encountering an error, you simply edit the code (in place), rebuild, and continue debugging the program.

The strength of this approach is that all tools and commands are accessible through a consistent user interface. Also, you can easily access hard-to-remember commands and options from menus and dialog boxes.

UNIX, on the other hand, has always offered a more segregated development environment. To create a new project, you first create a makefile, specifying what files compose the project, as well as the build tools, their options, and any numbers of build commands. You write the program using your favorite editor, and build and run the program from a shell. To debug the program, you run it within a command-line debugger (gdb), run it within a debugger in emacs, or use print statements.
3.1.1 Macintosh Programmer’s Workbench

One of the first development environments for the Macintosh was Macintosh Programmer’s Workbench (MPW), from Apple. MPW is an advanced environment for developing applications for 68k and PowerPC machines running Mac OS. It contains all the tools you would expect from an advanced development environment, including an editor, compiler, build tools, debugger, and shell. MPW was separated from other Macintosh development environments of its time by the blending of a command-line environment with elements of an IDE. To perform tasks and development activities, you entered commands into worksheets, much as you would under a UNIX shell, although the command language was specific to MPW. Figure 3.1 shows an MPW worksheet with some basic commands.

![Figure 3.1 MPW provides a command-line interface for entering commands and executing development tasks.](image)

MPW stood somewhere between a pure command-line development environment and an IDE. It required a bit of a learning curve compared to other IDEs, but it was far more powerful and extendable; it also was a favorite among advanced developers and those who preferred a command-line interface for application development. If you are interested, MPW is freely available from Apple at http://developer.apple.com/tools/mpw-tools.

3.1.2 THINK Pascal and THINK C

Another popular series of development tools for the Macintosh included the THINK Pascal and THINK C development environments from Symantec. Both
THINK Pascal and C were very good IDE-based development tools used by most Macintosh developers. The THINK Pascal debugger was one of the best parts of the program, foreshadowing many of the features that appeared in later Macintosh debuggers. In addition to their development tools and productive user interface, both environments supplied all the necessary software infrastructure for building Macintosh 68k applications. Later incarnations of THINK C included a C++ compiler and application framework called Think Class Library (TCL).1

You can still get a free copy of THINK Pascal from ftp://ftp.symantec.com//public/english_us_canada/products. Like MPW, it runs under Classic mode and is mainly of interest for its historical value or support of legacy applications.

### 3.1.3 CodeWarrior

Throughout the late 1980s to mid-1990s, the THINK tools were very popular. About this time, a company called Metrowerks began producing development tools for the Macintosh under the name CodeWarrior. The CodeWarrior environment was similar to the THINK tools; it included an editor, compilers, debugger, as well as an application framework called PowerPlant. At that time, the main features that distinguished CodeWarrior from other environments were its productive user interface, the quality of its compilers, and how it supported different compilers within a single development environment. In addition, Metrowerks was first to release a PowerPC (PPC) compiler when Apple transitioned its product line from the 68k to the PPC architecture.

THINK Pascal and THINK C were two separate products with two separate, yet similar, interfaces. In contrast, CodeWarrior offered a single development environment that supported different compilers, all within one product. Over time, Symantec lost market share to Metrowerks as the development environment of choice for building Macintosh applications. Metrowerks’ CodeWarrior is alive and well and is still one of the best commercial development tools for developing Macintosh programs (http://www.metrowerks.com).

### 3.1.4 Project Builder and Interface Builder

During this time, several attempts were made to bring UNIX tools to the Macintosh, although they were never mainstream efforts whose goal was to compete

---

1 THINK C was never a true C++ compiler. It lacked many C++ features such as constructors and method and operator overloading. Symantec C++ was the first true C++ compiler from THINK/Symantec, but it was too little too late, because most developers had already switched to Metrowerks compilers.
with commercial products or offer the broad feature set of MPW, THINK Pascal and C, and CodeWarrior. Before Mac OS X, you could find usable implementations of UNIX tools including Perl, gcc, bison, flex, and sed for the Macintosh. However, many of these tools never integrated well with the platform. With the introduction of Mac OS X, you now have access to a broad range of UNIX-based development tools from Apple, as well as third-party and open source developers. These tools do integrate well with the Mac OS X environment and provide a solid development foundation for building applications under Mac OS X.

Apple provides two main development tools for building applications under Mac OS X: Project Builder and Interface Builder.

Project Builder
Apple’s Project Builder is a freely available IDE that contains an editing, build, and run environment for developing Mac OS X applications. With Project Builder, you can build all types of Mac OS X applications, including Carbon and Cocoa applications, bundles, frameworks, kernel extensions, Java applications and applets, plug-ins, and tools (don’t worry if you do not know what some of these terms mean; they are all covered later in the chapter).

Project Builder is in the tradition of an IDE in the sense that all development tools and commands are aggregated under a single program. However, it does not include the main development tools (compiler, linker, assembler, version control, etc.) as part of the program. Instead, it uses UNIX development tools such as gcc, g++, and gdb. In a sense, Project Builder is evolutionary: it continues the line of IDE-based development environments for the Macintosh, but breaks with tradition by using external UNIX-based build tools for implementing build and development tasks. It strikes a nice balance by providing a modern interface for application development while leveraging the strengths of the UNIX tools set.

Interface Builder
Apple’s Interface Builder is used to design the user interface component of your program. Using Interface Builder, you design your application’s user interface components including menus, windows, icons, and dialog boxes. In addition, you can use Interface Builder to create your program’s code framework, which you fill in using Project Builder.
3.2 Creating an application with Project Builder

Before jumping into the details of Project Builder, let’s begin by looking at how simple it is to create an application. As you will see, Project Builder enables you to get a basic application shell running in no time. The example application you will build is a Cocoa program that displays an image in a window. You will learn all about Cocoa in chapters 5 and 6, but for now think of it as a collection of object-oriented libraries, or frameworks, for constructing GUI- and non GUI-based applications for Mac OS X.

Throughout the book, you will develop many applications. For consistency, you’ll store all the projects under a directory called projects, located in your home directory. At this point, create a new folder in your home directory and name it projects. Now, follow these steps:

1. Move to the Developer/Applications folder and launch Project Builder. (Better yet, drag the Project Builder icon to the Dock so you can get at it easily.)
2. Choose File→New Project (Shift-Command-N), select Cocoa Application (Nib Based) from the New Project list (see figure 3.2), and click the Next button.

---

For Cocoa-based applications, a Nib file holds your application’s interface objects (windows, menus, and so on) as well as the objects’ attributes and runtime relationship to other objects.

---
3 Set the location to your projects folder and the project name to DisplayCat, and click the Finish button. Project Builder will create a new project and display its main window.

Before making any changes, build and run the program by selecting Build → Build and Run (Command-R). As you can see in figure 3.3, with no coding you have a working application complete with a window and menu—all for free. Press Command-Q to quit the program.

Next, let’s add a picture to the project:

1 Select the Resource group, located in the Contents pane (on the left side of the main window under Groups & Files), and select Project → New Group. Call the new group Images (see figure 3.4).

2 Drag the file cat.tiff from DisplayCat/Images (located on the source code distribution disk) to the DisplayCat folder within your project directory.
Select the Images group and choose Project→Add Files. Select the cat.tiff file and click the Open button. Click the Add button in the next dialog to add the image file to the project. Doing so adds a picture of a cat to the project so it can be displayed on the program main window.

The next step is to add the cat picture to the main application window of the project:

1. If necessary, expand the Resource group (in the Contents pane) and double-click on the MainMenu.nib file. Doing so launches Interface Builder and loads the program's MainMenu.nib file. You'll use Interface Builder to design your application’s user interface, including menus, windows, icons, and dialog boxes. You will see four open windows within Interface Builder, as shown in figure 3.5. The first window (titled Window) is where you place the application’s picture. To its right is the Palette window, which holds Application Kit interface components. The window at the
bottom of the screen, called MainMenu.nib, holds the definition for the application menu, class instances, and images and sounds for the application. It also contains more complex information that is described in detail in chapters 6 and 7. Above this window is the application’s main menu.

2 Click on the Cocoa-Other icon, located in the toolbar of the palette window (third icon from the left), and select and drag the NSImageView object to the main window.

3 Move the NSImageView object toward the upper left in the window until you see the Aqua guides. The Aqua guides become active when you drag an interface object within a window. They provide on-screen feedback for placing an interface element in the correct location as specified in Apple’s Aqua Human Interface Guidelines. By following the Aqua guides, you can be sure you place interface elements such as buttons and text fields correctly within the window, adhering to Apple’s interface guidelines.

4 Using the Aqua guides for placement, resize the object until it fills the entire window (see figure 3.6).
5. Click the Image tab on the MainMenu.nib window and drag the image of the cat to the NSImageView object on the main window (see figure 3.7).
6. Save your work and switch back to Project Builder.
7. Build and run the project.
There you have it. In a few simple steps, you have a fully functioning application, complete with an application menu and window. As this example demonstrates, Project Builder gives you all the tools and infrastructure you need to construct applications with little effort. In fact, for this example you did not even write a line of code!

3.3 Project Builder in depth

As you saw in the previous section, creating the core infrastructure for an application is simple and straightforward with Project Builder. With just a few clicks, you were able to get a basic application running in no time. Next, let’s look at Project Builder in detail, discussing its features and operation.

3.3.1 Targets and build styles

Before describing Project Builder’s interface, let’s define some terms and concepts. First, you need to understand targets and build styles. A target collects project components that make up a project (source files, header files, and libraries), defines the basic instructions and attributes that specify how Project Builder builds a project, and holds information a program uses at runtime, such as command-line arguments and environment variables. Think of a target as a way of encapsulating the items and attributes that form a program. Build styles, on the other hand, override certain aspects of a target’s build instructions to create specialized versions of the program.

Let’s illustrate targets and build styles through an example. Imagine you wish to develop a program called AgentServer. The program reads XML-formatted messages from agents, performs some action, and returns a result to the agent. One of the primary requirements of the program is performance: it must service agent request in a timely and predictable manner. To test this requirement, you write several test agents that simulate various agent behaviors and create different versions of your program; each build has different compiler optimizations. The goal is to test these agents with each version of the program and see how they perform.

Let’s take a high-level look at how you can use targets and build styles for this problem:

1. Create a new project for the program. When you create the new project, Project Builder also creates a new default target with the same name as the project. This target automatically holds all files that compose the program, as well as default build settings.
Create several new build styles—one for each type of optimization setting you wish to test.

To get performance statistics, you also need to add profiling code, either by writing your own routines or by using the \(-pg\) compiler flag. The \(-pg\) flag adds profiling code that produces an execution profile of your program. GNU \texttt{gprof}, a profiling tool that comes with Mac OS X, uses this information to display performance statistics for your program. (This program, as well as the other Mac OS X developer tools, is discussed in chapter 4.)

Testing the different versions of the server is as easy as selecting a build style, building the program, running the program and the test agents, and collecting statistics. You repeat this process for each build style. Once you’ve finished, you can compare the effect of the different optimization settings on the program’s performance. As you can see from this example, build styles are a simple and intuitive way to apply different build settings to a target.

Let’s take this example a step further. Imagine your server uses a third-party XML parser to decode the XML-based strings sent by the agents. Also imagine you’ve wrapped the parser with custom code that encapsulates its behavior, so swapping in a different parser will not change any client code. After repeated testing, you find the parser is the performance bottleneck. At this point, you would like to swap in some other parsers (maybe one you wrote) to see if you can increase performance. This is a perfect application for using multiple targets within a project. Without targets, you would need to create several new projects, one for each parser. Using targets, you create a new target for each parser you wish to test, add the appropriate files to the target, and switch between each target for testing—all within one project.

To sum up, targets collect files and build settings for a program. If different versions of your program contain different files, you should express each version with a target. Build styles enable you to override the default build setting for the active target so you can perform particular types of builds.

### 3.3.2 Project Builder’s UNIX tools

Another concept to understand is that Project Builder uses UNIX command-line tools for performing builds, managing source code, and debugging applications. Thus you can leverage your current knowledge of UNIX development tools when using Project Builder. For example, Project Builder contains all the hooks for adding stricter static checking to your compiles. So, you can use the same compiler flags you use from the command line within Project Builder.
3.3.3 Project Builder’s interface

Now, let’s take a closer look at each of the components of the main project window. Figure 3.8 shows the main Project Builder window.

The toolbar

At the top of the window is the toolbar, which contains a series of icons representing common Project Builder commands (see figure 3.9).

The icons on the left side of the toolbar execute various build commands. Beginning at far left, the buttons are as follows:

- **Build Active Target**—(Command-B) Builds the active target by executing the build command.
- **Build and Debug Active Target**—(Command-Y) Executes a build all command and runs the program under the debugger.
- **Build and Run Active Target**—(Command-R) Similar to Build Active Target; but once Project Builder successfully builds the project, this button runs the program.
CHAPTER 3  
Project Builder and Interface Builder

Clean Active Target—(Shift-Command-K) Invokes the clean command, deleting any intermediate files from the project folder (including object and executable files). This command is like the UNIX make clean command.

Collectively, these icons enable easy access to the most frequently used build commands.

The Active Target menu enables you to move between all active targets in your project. A target encapsulates the items that compose a version of the program and the general attributes that define how Project Builder builds these components.

The buttons on the right end of the toolbar run debugging commands. Project Builder enables these icons once a program is running under the debugger. Beginning from the left, they’re as follows:

- **Restart**—Restarts, or reloads, the program in the debugger (does not continue from where you were, but from the beginning of the program).
- **Pause**—(Option-Command-C) Suspends execution of the current program.
- **Continue**—Resumes program execution.

The next three commands control how execution continues from a function or method call:

![Figure 3.9 The toolbar contains shortcuts to common build and run commands.](image-url)
- **Step Over**—(Shift-Command-O) Executes the current function or method, but does not single-step into the function code.
- **Step Into**—(Shift-Command-I) Jumps to the current function or method and single-steps through the code.
- **Step Out**—(Shift-Command-T) Immediately returns to the calling function after executing the rest of the function.

**The Contents pane**

The next component of the project window is the Contents Pane, shown in figure 3.10. The Contents pane provides various views of project items, as well as easy access to the individual items that compose the target:

- **Files view**—Accessible by clicking on the Files tab. Lists all files and components that make up the project. These include source and header files, libraries, frameworks for the application environment (Carbon, Cocoa, Java), resource files, and the project product, or executable application. Where applicable, clicking or double-clicking on a file will open it for editing or viewing. For example, clicking on a source file or a header file will display it in the Editor pane. Double-clicking on a .nib file will open the file in Interface Builder.
- **Classes view**—Enables easy access and browsing of application and framework class files (see figure 3.11). The Class pane (upper pane) displays a filtered
view of all classes in the active target. You can filter the files that Project Builder displays by selecting various filtering options from the pop-up menu at the bottom of the pane. You can also change the display options by clicking the Options button, also located at the bottom of the pane. (You have not created any new classes in the DisplayCat project, so certain filtration options may not show anything.) Double-clicking on a class’s book icon displays documentation for the class. Clicking on a class displays its members in the Members pane (lower pane). The Members pane lists all members for the selected class. Clicking on a class member loads the class’s implementation file into the editor. Double-clicking on a class member loads the class’s interface file into the editor in an external window.

Figure 3.11  The Classes view lets you browse the interface and implementation class files for both application and framework classes.
Project Builder in depth

73

Bookmarks view—Holds various pieces of information related to the project including web sites, project notes, code snippets, and documentation files. This is a nice way to store and access project-related information from within Project Builder.

Target view—Holds two display panes: Targets and Build Styles (see figure 3.12). The Targets pane displays all targets for the project. Clicking on a target displays the settings for the selected target. From here, you can view and edit all setting for the target. The Build Styles view lists all build styles for the selected target. As in the Targets pane, selecting a build style enables you to view and edit its settings. You create new targets and build styles by selecting Project→New Target or Project→New Build Style, or by holding down the Control key, clicking on the appropriate pane, and selecting New Target or New Build Style from the contextual menu.
CHAPTER 3  
Project Builder and Interface Builder

- **Executable view**—Displays the different executable programs that your program contains. For example, imagine you have several targets in your project. The Executable view will have an executable program entry for each corresponding target. (This feature has been move to the Targets tab under Mac OS X 10.2 [Jaguar].)
- **Breakpoints view**—Lists breakpoints from the current debugging session.

**The Action pane**
You use the Action pane to perform actions related to the current project, as well as view the results of the action. The panel contains four tabs, each enabling different functionality (if you have changed the settings in Preferences→Task Templates→Basic Settings from One Window to Some Windows or Many Windows, you may not see the Action pane or these four tabs; instead, when you select the Find item or press a build/run button, a dedicated find, build, run, or debug window will open):

- Click on the **Find tab** (see figure 3.13) to search any combination of project or framework files for a specified token. Find offers many options, including filtering the files that are searched and choosing a specific search type (such as textual and regular expression searches). In addition to finding text, you can choose to perform local or global search and replace operations.
- Project Builder automatically selects the **Build tab** when you build the project. The Build pane displays the progress of a build operation. You can control the level of detail displayed during a build by selecting Project Builder→Preferences, clicking the Building icon in the toolbar, and selecting the appropriate setting from the Build Log Detail Level menu.
- The **Run pane** displays the output a program sends to the stdout and stderr streams. This can be useful even in GUI applications for displaying debugging or logging messages.
- Project Builder activates the **Debug pane** when you run your program under the debugger. From here, you can view the contents of variables or data members, step into code, and view and traverse the call stack, all on a thread-by-thread basis. When debugging, you can also view the current contents of the console and standard IO (StdIO) buffers. Because Project Builder uses gdb as its debugger, all gdb commands are also available. To directly enter gdb commands, set a breakpoint in your program and start the debugger by clicking the Build and Debug icon. Once execution stops at the breakpoint, click on the Console tab and enter your gdb commands at the prompt.
The Editor pane

The Editor pane (see figure 3.14) is located at the bottom-right of the project window. It provides an area for displaying and editing source code, viewing documentation, and viewing and editing target and build settings. For example, when you select a source or header file from the Contents pane, Project Builder displays the file in the Editor pane. When you select Help → Project Builder Help, documentation files are displayed in the Editor pane; when you select a target or build style, the selected target or build style’s current values are displayed, enabling you to view or edit the values.

At the top of the Editor pane is a toolbar that changes based on the type of information displayed in the pane. Figure 3.15 shows the toolbar Project Builder displays for a source file:
CHAPTER 3
Project Builder and Interface Builder

Figure 3.14 You edit and view project elements in the Editor pane. Elements include source code, header files, target and build setting, and documentation files.

Figure 3.15 The Editor pane contains a toolbar that changes based on the type of information displayed in the pane.
The Go Back and Go Forward icons at the left end of the toolbar enable you to cycle forward and backward between currently loaded views.

The Current View item displays the currently loaded entry. Clicking on this item displays a menu of all loaded views (see figure 3.16). From here, you can select different views to display in the pane.

The Current Location item shows the cursor position within the current file. For example, if a source file is loaded and the cursor is on or within a method, Project Builder displays the method name. Like Current View, clicking on this item will show a menu that lists the file's functions or methods (see figure 3.17).

Project Builder enables the Check Syntax icon when a source file is loaded. Clicking on this button will check the syntax of the file based on its language.

The Display New Counterpart Syntax icon toggles between interface and implementation files. If an implementation file is loaded, clicking this button displays its interface file; if an interface file is loaded, clicking this button displays its implementation file.
The Split Editor button splits the Editor view into panes. Clicking on the Close Split icon closes the current pane.

The status bar

The status bar is located at the bottom of the project window. It displays information about the status of Project Builder operations, including the stages and results of a project build, the result of a find operation, and other tasks.

3.3.4 Project Builder scenarios

Now that you understand some of the basics of the Project Builder interface, let’s take a closer look at several scenarios that consistently come up when developing programs under Project Builder. Remember, programming, like most aspects of computing, is learned through practice, not just by reading or studying theory. Theory may be able to get you from linear to logarithmic, or exponential to linear time, but it cannot teach you to write good code—you accomplish this through practice. Consequently, be sure you try these examples as you read. Enough talk; let’s get down to work.

Creating a project

The first step in developing a program under Project Builder is to create a project. Project Builder enables you to create many kinds of programs for Mac OS X, including applications written in Carbon, Cocoa, Java, frameworks, bundles, and kernel extensions. Creating a project is similar to writing a makefile, but provides a friendlier way of controlling how a project is built and what is included in the build.

To create a new project, launch Project Builder and select File→New Project (Shift-Command-N). Project Builder opens a window that displays a list of the available project types (see figure 3.18).

Let’s take a closer look at the different project types. Currently, you can choose from seven categories plus the Empty Project option:

- Empty Project—A project with no added libraries, frameworks, or other software infrastructure files.
- Application—There are nine application types:
  - AppleScript Application—An AppleScript Studio application, which is a Cocoa-based application that contains hooks for AppleScript. This type of project is useful for putting a Cocoa-based GUI on an AppleScript-based program.
  - AppleScript Document-based Application—The same as an AppleScript Application, but includes support for the NDDocument architecture.
- **AppleScript Droplet**—An AppleScript application in which files are processed by dragging them to the application icon.

- **Carbon Application**—A Carbon application that includes all necessary support files and frameworks for developing both Mac OS and Mac OS X Carbon applications; it uses Resource Manager files to store application resources (.r or .rsrc files).

- **Carbon Application (Nib Based)**—The same as Carbon Application, but uses Nib-based resources (.nib).

- **Cocoa Application**—A Cocoa application using Objective-C as its development language.
• **Cocoa Document-based Application**—Same as Cocoa Application, but adds support for the NDDocument architecture.

• **Cocoa-Java Application**—A Cocoa application using Java as its development language.

• **Cocoa-Java Document-based Application**—Same as Cocoa-Java Application, but adds support for the NDDocument architecture.

**Bundle**—There are three bundle options:

• **Carbon Bundle**—A bundle linked to Carbon.

• **CFPlugin Bundle**—A bundle linked to the Core Foundation framework.

• **Cocoa Bundle**—A bundle linked to Cocoa.

**Framework**—There are two framework types:

• **Carbon Framework**—A framework linked to Carbon.

• **Cocoa Framework**—A framework linked to Cocoa.

**Kernel Extension**—There are two kernel extension options:

• **Generic Kernel Extension**—A kernel extension project.

• **I0Kit Driver**—An I/O Kit project, used for developing kernel drivers.

**Pure Java**—There are five Pure Java types to choose from:

• **Java AWT Applet**—A project for developing AWT-based (Abstract Window Toolkit) Java applets. AWT is superceded by Swing, which is more advanced and simpler to use. It is used primarily for compatibility with Mac OS 9.

• **Java AWT Application**—A project for developing AWT-based (Abstract Window Toolkit) Java applications. AWT is superceded by Swing, which is more advanced and simpler to use. It is used primarily for compatibility with Mac OS 9.

• **Java Swing Applet**—A project for developing Swing-based Java applets. Swing supercedes AWT and provides a more advanced and simpler to use interface toolkit.

• **Java Swing Application**—A project for developing Swing-based Java applications. Swing supercedes AWT and provides a more advanced and simpler to use interface toolkit.

• **Java Tool**—A project for developing Java applications or libraries that do not require a GUI.
Standard Apple Plug-ins—There are three options for standard Apple plug-ins:

- **IBPalette**—A project for developing an Interface Builder palette, which contains components available for developers to add to applications (including menus, text fields, and other interface components).
- **PreferencePane**—A project for developing a Preference Pane, which resides in the System Preference application and is used to set system-wide parameters such as screen saver settings, network settings, and energy settings.
- **Screen Saver**—A project for developing screen saver modules.

Tool—There are five command-line tool types:

- **C++ Tool**—A project for developing C++ applications. This option is used for building C++ command-line tools.
- **CoreFoundation Tool**—A project for developing a tool that is linked to the Core Foundation framework.
- **CoreServices Tool**—A project for developing a tool that is linked to the Core Services framework.
- **Foundation Tool**—A project for developing a tool that is linked to the Foundation framework.
- **Standard Tool**—A project for developing C applications. This option is used for building C command-line tools.

To create a new project, select the appropriate project type from the list and click the Next button. Enter the name of the project and the location where the project folder will be stored. You can select a location by clicking the Choose button and choosing the location from the directory sheet. Once you have filled in this information, click the Finish button. Project Builder will create a new project based on a template and store it in the specified location.

Adding files to a project

A common operation during program development is to add files to a project. Project Builder supports this process through the New File command.

Imagine you have already created a new Cocoa project, written in Objective-C, and you wish to add a new source file to the project. Follow these steps:

1. Select File→New File or press Command-N to open the New File window shown in figure 3.19.
2 The New File window lists the types of files you can add to your project, classified by project category. Let’s say you wish to add a new Objective-C class to a Cocoa project. To do this, select Objective-C Class from the list and click the Next button to open the window shown in figure 3.20.

![Figure 3.19](image1.png)

Figure 3.19
The New File window displays the types of files you can create for various languages and projects.

![Figure 3.20](image2.png)

Figure 3.20
When creating a new class file, Project Builder displays the New Objective-C class window. Use this window to specify the name of the class and its location.
Enter the name of the class. Make sure the Also Create checkbox is checked so a corresponding header file is created.

Select the location where the generated class files will be stored (typically the project folder). You can select a location by clicking the Choose button and choosing the location from the directory sheet, or you can type it in by hand. The Add To Project menu lets you select the project to which the new files are added (typically the current project).

Click the Finish button. Project Builder creates new files for the class’s interface and implementation, stores them in the specified location, and adds them to your project.

The new files are accessible from the Contents pane. By default, the files are located outside the listed folders. Usually, you will move the new class files to the Classes folder by highlighting them and dragging them to the folder.

Using CVS
As you already know, Project Builder uses UNIX tools to perform many of its tasks. For version control, Project Builder uses CVS (Concurrent Versions System). The CVS revision control program stores a file’s change history and supports commands for easy access to past versions of the file. CVS is built on top of a version control system called Revision Control System (RCS), and uses RCS commands behind the scene to perform its actions. (The RCS program dates back to the early 1980s and was written by Walter F. Tichy while at Purdue University.) Though the underpinnings of CVS and RCS are similar, the nomenclature, intended audience, and command set are very different.

Both RCS and CVS are excellent choices for a version control system and are available under Mac OS X. The system you use really depends on the organization of your project. Project Builder supports CVS from its interface. Unfortunately, it does not currently support RCS.

Setting up CVS for use with Project Builder is simple: you set up a CVS repository on one of your disks, set your CVS environment variables, and check in/out the project. Once these steps are complete, you can open the project under Project Builder and get full access to the CVS command set and repository. To make this clear, let’s go through each step in more detail.

Before using CVS, you need to configure a few things, including the CVS repository and the client environment. The first step is to set up the CVS repository and environment variables:
1 Open the Terminal application (located in /Applications/Utilities) and create a directory to hold the CVS repository for your projects. The repository is a central location that holds all files stored under version control. Place this directory on a disk partition that is accessible to all users of the version control system and that is large enough to handle the anticipated file storage requirements. Try to be overly conservative when estimating your disk requirement. For this example, place the repository in your home directory under the name cvs-repository.

2 Set the CVS environment variable CVSROOT to the location of the directory holding the repository (the directory you just created). Doing so enables CVS commands to locate files under version control. The following command sets the CVSROOT environment variable to the correct location (for the tcsh shell).

```bash
% setenv CVSROOT /Users/omalley/cvs-repository
```

3 The CVSEDITOR environment variable specifies the editor you will use to enter file revision descriptions. Set it to your favorite editor, such as emacs or vi. Under Project Builder, this variable is not used; instead, Project Builder opens a Sheet dialog in which you enter revision commands. If you will ever use CVS from the Terminal, it makes sense to set the variable anyway. Because I’m an emacs user, I set this variable to emacs:

```bash
% setenv CVSEDITOR emacs
```

4 Run the CVS initialization command to create the CVS administrative files in the repository:

```bash
% cvs init
```

You only need to run the cvs init command once, before anyone on the system uses the new repository. If for some reason you set up another repository in a different location, say for personal files, just change the CVSROOT environment variable to the location of the new repository and run the CVS initialization command.

For convenience, add the environment variables to your startup file. Doing so prevents you from entering them each time you open a shell. For the tcsh shell, add them to your .cshrc file:

```bash
% setenv CVSROOT /Users/omalley/cvs-repository
% setenv CVSEDITOR emacs
```

---

3 I typically do not set my CVSROOT in .cshrc. Instead, I set the repository location manually using aliases:

```bash
alias cvs-proj1 'setenv CVSROOT /Users/omalley/proj1'
alias cvs-proj2 'setenv CVSROOT /Users/omalley/proj2'
```

This approach enables me to switch between multiple project repositories.
% setenv CVSROOT /Users/omalley/cvs-repository
% setenv CVSEDITOR emacs

Now, the environment is set up and ready to go. The next step is to place a project under version control and access it from Project Builder. You use the `cvs import` command the first time you place a module under version control. The import command takes all files in the working directory (and subdirectories) and adds them to the repository specified by `CVSROOT`:

```
cvs import [-options] [repository] [vendortag] [releasetag]
```

The `options` argument specifies options applied by the import command. (See the CVS man page or documentation for a description of the options.) The `repository` argument specifies the directory where CVS will store the project files within the repository. The vendor and release tags specify vendor and release information.

Let’s import a project into CVS for use under Project Builder:

1. Create a new Cocoa project called `CocoaExample` and store it in your project directory. Build and close the project.
2. Open the Terminal application and change directory (`cd`) to the directory that holds the project:
   ```bash
   cd ~/projects/CocoaExample
   ```
3. Check in the project with the `import` command. The `–m` option is used to enter your revision control message from the command line. If you remove it, CVS will open the editor specified in the `CVSEDITOR` variable. When CVS imports or checks in files, it displays a file status symbol, expressed as the single, leftmost character (Table 3.1 lists the symbols seen at the beginning of the lines and their descriptions). The `import` command is as follows:

```
% cvs import –m "Initial revision." projects/CocoaExample book start
```

No conflicts created by this import.
Change directory to the parent of CocoaExample (cd ~/projects) and remove the CocoaExample directory (rm -rf CocoaExample). Change directory to the parent of the project directory and check out the version (do not remove or mess with the CVS directory; CVS uses it to resolve differences between local versions of files and those stored under version control):

```bash
% cd ~/projects; rm -rf CocoaExample; cd ..
cvs checkout: Updating projects/CocoaExample
U projects/CocoaExample/main.m
U projects/CocoaExample/CocoaExample.pbproj
U projects/CocoaExample/CocoaExample.pbproj/omalley.pbxuser
U projects/CocoaExample/CocoaExample.pbproj/project.pbxproj
U projects/CocoaExample/CocoaExample/English.lproj
U projects/CocoaExample/CocoaExample/English.lproj/InfoPlist.strings
U projects/CocoaExample/CocoaExample/English.lproj/MainMenu.nib
U projects/CocoaExample/CocoaExample/English.lproj/classes.nib
U projects/CocoaExample/CocoaExample/English.lproj/info.nib
U projects/CocoaExample/CocoaExample/English.lproj/MainMenu.nib/objects.nib
```

Within the CocoaExample directory, you will see a directory called CVS:

```bash
% cd projects/CocoaExample; ls
CVS  CocoaExample.pbproj  English.lproj  build  main.m
```
Launch Project Builder and open the CocoaExample project. You should see a CVS icon at the top left of the Contents pane; it indicates that Project Builder recognizes the project is under version control. Project Builder will also enable the CVS menu when you save changes to a source file (see figure 3.21).

You can use these menu items to interact with CVS and perform operations on the repository. Not all of the CVS commands are available from the CVS menu, but most of the basic ones are there. Typically, you will interact with the CVS repository from the command line and from within Project Builder. If you are a purist, you can still use CVS from the command line only.

Creating new targets and build styles

As you have already seen, a target collects all necessary components that make up a project, as well as instructions for building the project. Components include source files, header files, and libraries. On the other hand, build styles enable you to customize the build options of a specified target. Let’s look more at the relationship between targets and build styles through an example.

Imagine you are developing an application that implements two Fibonacci number generators, recursive and loop-based, and you wish to see how the different compiler optimizations affect performance. By using targets and build styles, you can compare two implementations without changing any client code in the main function. Follow these steps:

1. Launch Project Builder and create a new project using the C++ Tool template.
2. Create and add two files to the project, one for each implementation. Call the files FibonacciRecursive.cpp and FibonacciLoop.cpp. Implement each algorithm in its corresponding file and add code to main to call the function. Listing 3.1 shows the code for the different implementation of the Fibonacci program and the program’s main function.
CHAPTER 3
Project Builder and Interface Builder

Listing 3.1 Two implementations of the Fibonacci program and the main function

/*
Loop (iterative) implementation of the Fibonacci number series generator.
*/
long
Fibonacci(long n)
{
    if (n == 0)
        return 0;
    long x = 1;
    long y = 0;
    long z = 0;
    for (long i=1; i<n; i++) {
        z = x;
        x += y;
        y = z;
    }
    return x;
}

/*
Recursive implementation of the Fibonacci number series generator.
*/
long
Fibonacci(long n)
{
    if (n == 0)
        return 0;
    if ( (n == 1) || (n == 2) )
        return 1;
    return Fibonacci(n-1) + Fibonacci(n-2);
}

using namespace std;
#include <iostream>
#include "FibonacciRecursive.h"
#include "FibonacciLoop.h"
const int N_FIBONACCI_NUMS = 10;
int
main()
{
    long fibonacciResult[N_FIBONACCI_NUMS];
    for(int i=0; i<N_FIBONACCI_NUMS; i++) {
        fibonacciResult[i] = Fibonacci(i);
    }
cout << "Computed Fibonacci values:" << endl;
for(int i=0; i<N_FIBONACCI_NUMS; i++) {
    cout << "fib(" << i << ") = 
        " << fibonacciResult[i] << endl;
} return 0;

Now that the code is in place, you need to create two targets for testing. To create a new target, select Project→New Target or click on the Targets tab from the Contents pane, Control-click (right-click) on the upper pane, and select New Target. Select Tool and click on the Next button; name the new target Recursive.

Create another target, called Loop.

Add the appropriate source code files to its target (see figure 3.22). Activate the recursive target by clicking the radio button to its left. Click on the Files tab in the Contents pane, and click the checkbox to the left of the files main.c, FibonacciRecursive.cpp, FibonacciRecursive.h, and libstdc++.a (located under the External Frameworks and Libraries).

Activate the loop target and do the same, this time selecting main.c, FibonacciLoop.cpp, FibonacciLoop.h, and libstdc++.a.

To test each implementation, select its target and build and run the program.

As you can see, targets are a convenient way to build versions of a program that contain different files.

Next, let’s look at creating and applying build styles to the project. Build styles enable you to override aspects of a target’s build settings. By default, two build styles are defined: Development and Deployment. Each style overrides the build settings of the active target. Let’s create a few build styles for testing the effect of different compiler optimizations.

To create a new build style, follow these steps:

1. Select Project→New Build Style or click on the Targets tab from the Contents pane, Control-click (right-click) on the lower pane, and select New Build Style from the menu.
2. Enter the name of the build style. Within the Fibonacci project, create four build styles: OptimizationNone, Optimization1, Optimization2, and OptimizationSize.
To apply a build style to a target, click on the Targets tab from the Contents pane, select a target from the list, and select the appropriate build style.

Now, when you build the selected target, Project Builder will apply the selected build settings. As you might expect, build styles are a simple and straightforward way to quickly apply a variety of build settings to a target.

**Project Builder preferences**

The Preferences dialog, available from the Project Builder Preference menu, enables you to set application-wide preferences (see figure 3.23). Editor settings are updated through the Text Editing, Syntax Coloring, and Indentation items.

Using these items, you can set up the Project Builder editor to mimic basic aspects of the *emacs* language modes, as well as customization options including...
showing matching braces, tab settings, text syntax coloring options, and syntax-aware indentation settings. These options do not give you the customization available from *emacs*, but do provide lots of functionality for very little effort. (The editor does support a subset of common *emacs* key bindings such as Control-A and Control-E for moving the insertion point to the beginning and end of line.)

Many more customizations are supported through the Preferences dialog, including tailoring class and function navigation, modifying build option behavior, and changing interface and behavioral elements of the debugging environment. The best way to get a feel for them is to open the dialog box and try them.

**Setting build and link options**

In UNIX-style development, you specify your programs build settings in a makefile. Project Builder also enables you to add and remove build settings, but through its interface. Let’s look at some of the build options for controlling compiler and link settings. As previously mentioned, each target defines its own compiler and link settings, which you can override using a build style (see figure 3.24).

Each time you issue a build command, Project Builder writes the status of the build to the Build pane. You can set the level of detail that it displays through the Preferences dialog under the Build item. There are three options to choose from: minimal, standard, and detailed logs. Listing 3.2 shows the information Project Builder displays for each type of build.
Minimal Log:
/usr/bin/jam -d0 JAMBASE=/Developer/Makefiles/pbx_jamfiles/
    ProjectBuilderJambase
    JAMFILE=- build ACTION=build TARGETNAME=Recursive NATIVE_ARCH=ppc
    BUILD_STYLE=Development
CPP_HEADERMAP_FILE=/Users/omalley/projects/TargetBuildExample/
    build/intermediates/Recursive.build/Headermaps/Recursive.hmap
DSTROOT=/ OBJROOT=/Users/omalley/projects/TargetBuildExample/
    build/intermediates SRCROOT=/Users/omalley/projects/TargetBuildExample
SYMROOT=/Users/omalley/projects/TargetBuildExample/build
Completed phase <CopyHeaders> for Recursive
Completed phase <DeriveAndCompileSources> for Recursive
Completed phase <LinkWithFrameworksAndLibraries> for Recursive

Figure 3.24 Within a target, you add compiler options under the Build Settings tab.

Listing 3.2 Output from the three types of build options
Completed phase <RezResourceManagerFiles> for Recursive

Standard Logs:
/usr/bin/jam -d1
JAMBASE=/Developer/Makefiles/pbx_jamfiles/ProjectBuilderJambase
JAMFILE= build ACTION=build TARGETNAME=Recursive NATIVE_ARCH=ppc
BUILD_STYLE=Development
CPP_HEADERMAP_FILE=/Users/omalley/projects/TargetBuildExample/build/intermediates/Recursive.build/Headermaps/Recursive.hmap
DSTROOT= OBJROOT=/Users/omalley/projects/TargetBuildExample/build/intermediates SRCROOT=/Users/omalley/projects/
TargetBuildExample
SYMROOT=/Users/omalley/projects/TargetBuildExample/build
...updating 11 target(s)...

BuildPhase Recursive
Completed phase <CopyHeaders> for Recursive
Mkdir /Users/omalley/projects/TargetBuildExample/build/
intermediates/Recursive.build/Objects/ppc
CompileCplusplus /Users/omalley/projects/
TargetBuildExample/build/
intermediates/Recursive.build/Objects/ppc/main.o
CompileCplusplus /Users/omalley/projects/
TargetBuildExample/build/
intermediates/Recursive.build/Objects/ppc/FibonacciRecursize.o
BuildPhase Recursive
Completed phase <DeriveAndCompileSources> for Recursive
MasterObjectFile.Combine /Users/omalley/projects/
TargetBuildExample/build/
intermediates/Recursive.build/master.o
StandaloneExecutable /Users/omalley/projects/TargetBuildExample/build/
Recursive
BuildPhase Recursive
Completed phase <linkWithFrameworksAndLibraries> for Recursive
BuildPhase Recursive
Completed phase <RezResourceManagerFiles> for Recursive
...updated 11 target(s)...

Detailed Logs:
/usr/bin/jam -d2 JAMBASE=/Developer/Makefiles/pbx_jamfiles/
ProjectBuilderJambase
JAMFILE= build ACTION=build TARGETNAME=Recursive
NATIVE_ARCH=ppc BUILD_STYLE=Development
CPP_HEADERMAP_FILE=/Users/omalley/projects/TargetBuildExample/build/intermediates/Recursive.build/Headermaps/Recursive.hmap
DSTROOT= OBJROOT=/Users/omalley/projects/TargetBuildExample/build/intermediates SRCROOT=/Users/omalley/projects/
TargetBuildExample
SYMROOT=/Users/omalley/projects/TargetBuildExample/build
...updating 11 target(s)...

BuildPhase Recursive
    echo Completed phase "<CopyHeaders>" for "Recursive"
Completed phase <CopyHeaders> for Recursive

```
/bin/mkdir -p */Users/omalley/projects/
TargetBuildExample/build/
intermediates/Recursive.build/Objects/ppc

CompileCplusplus /Users/omalley/projects/TargetBuildExample/build/
intermediates/Recursive.build/Objects/ppc/main.o

BuildPhase Recursive
```
Completed phase <DeriveAndCompileSources> for Recursive
ClearFileList /Users/omalley/projects/TargetBuildExample/build/
intermediates/Recursive.build/Objects/LinkFileListPrelink

/bin/rm -rf "/Users/omalley/projects/TargetBuildExample/build/
intermediates/Recursive.build/Objects/LinkFileListPrelink"

AppendToFileList /Users/omalley/projects/TargetBuildExample/build/
intermediates/
Recursive.build/Objects/LinkFileListPrelink

for file_reference in
"/Users/omalley/projects/TargetBuildExample/build/
Recursive.build/Objects/ppc/main.o"
"/Users/omalley/projects/TargetBuildExample/build/
Recursive.build/Objects/ppc/FibonacciRecursive.o"
do
  echo "$file_reference" >>
"/Users/omalley/projects/TargetBuildExample/build/
Recursive.build/Objects/LinkFileListPrelink"
done

MasterObjectFile.Combine
/Users/omalley/projects/TargetBuildExample/build/intermediates/
Recursive.build/master.o

/usr/bin/cc -arch ppc -keep_private_externs -nostdlib
-filelist "/Users/omalley/projects/TargetBuildExample/build/
intermediates/
Recursive.build/Objects/LinkFileListPrelink" -r -o
"/Users/omalley/projects/TargetBuildExample/build/
Recursive.build/master.o"

ClearFileList /Users/omalley/projects/TargetBuildExample/build/
intermediates/
Recursive.build/Objects/LinkFileList

/bin/rm -rf "/Users/omalley/projects/TargetBuildExample/build/
intermediates/Recursive.build/Objects/LinkFileList"

AppendToFileList /Users/omalley/projects/TargetBuildExample/build/
intermediates/
Recursive.build/Objects/LinkFileList

for file_reference in
"/Users/omalley/projects/TargetBuildExample/build/
Recursive.build/master.o"
do
  echo "$file_reference" >>
"/Users/omalley/projects/TargetBuildExample/build/
Recursive.build/Objects/LinkFileList"
done

StandaloneExecutable
The minimal setting only shows the basics of the build: the commands run and any errors and warnings. The standard setting provides more information about each step in the build process. The detailed setting shows the commands run, their command line, and warnings and errors. Note the use of the Jam program (a make replacement) for managing the build process; it also uses different debugging values (-d0, -d1, -d2).

Now, let’s look at the Compiler Settings section in the Editor pane. The Code Generation area is used to set the desired optimization level for the build. The optimization levels in the menu correspond to the standard gcc optimization levels (see table 3.2).

Table 3.2  The compiler optimization levels available under gcc/g++

<table>
<thead>
<tr>
<th>Menu item</th>
<th>gcc option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>None (less optimized, more debuggable)</td>
<td>-O</td>
<td>Does not optimize</td>
</tr>
<tr>
<td>Level 1 -O1</td>
<td>-O0</td>
<td>Optimizes</td>
</tr>
</tbody>
</table>
(See the gcc documentation’s section “Options That Control Optimization” for more specific information about optimizations settings.)

The next option is the Generate Profiling Code checkbox. Enabling this box adds -pg to your build options, which adds code to support program performance analysis with gprof. The gprof program is used to display a performance execution profile for your program.

Enabling the Generate Debugging Symbols checkbox adds the -g option to the build, which adds symbolic information to the object files, enabling gdb to provide you with more information while debugging. The Other C Compiler Flags text field is used to adding additional compiler flags to the build.

Link options are set in the Linker Settings portion of the Build Settings section. This section enables you to customize elements of the link phase of the build. Project Builder uses ld, the Mach object file link editor, to perform link operations; libtool to create static and dynamic libraries; and dyld to load an application’s dynamic link libraries into its address space.

**Development from the command line under Project Builder**

In addition to building your programs from within Project Builder, you can choose to build them from the command line using the `pbxbuild` command. To use the command, open a shell and change to the directory that contains your project. The `pbxbuild` program has several command-line options:

```
pbxbuild [-activetarget | -alltargets | -target <targetname> ]
[-buildstyle <typename> ] [ clean | install ]
[ <variable>=<value> ]
```

To build Project Builder active target, use the -activetarget option; to build all targets in the project, use -alltargets; or to build a specific target, use the -target option, followed by the target name. You apply build styles by specifying the build style option followed by the name of the style. Both target and build style names
are case sensitive. In addition, pbxbuild supports make-like options such as clean and install, which build and install the program in the target directory.

The makefile in listing 3.3 is a simple example of how to use pbxbuild and its options.

```
#---------------------------------------------
# $Id$
#---------------------------------------------

BUILD_TOOL = pbxbuild
# Uncomment and edit <stylename> to a Project Builder
# build style name.
BUILD_STYLE = # -buildstyle <stylename>
#
# Build targets
# ---------------------------------------------
all:
    $(BUILD_TOOL) -alltargets $(BUILD_STYLE)
active:
    $(BUILD_TOOL) -activetarget $(BUILD_STYLE)
# Edit <targetname> to target to build.
target:
    $(BUILD_TOOL) -target <targetname> $(BUILD_STYLE)
#
# Build action (can specify more than one):
#  export install clean installsrc
# ---------------------------------------------
clean:
    $(BUILD_TOOL) clean
# Must specify SRCROOT in environment.
export:
    $(BUILD_TOOL) SRCROOT=. export
install:
    $(BUILD_TOOL) install
list:
    $(BUILD_TOOL) -list
```

You can run this makefile from the command line or from within an editor like emacs. The advantage of running it within an editor is that you can use the editor’s next/previous message command to jump to the source line of an error or warning. (Within emacs, use Control-x-~ to get the next compiler error or warning.)
Adding static checking to builds

Static code analysis refers to techniques and methods applied before a program is run that highlight potential problems, anomalies, or errors in source code. In the software engineering literature, as well as in practice, static code analysis means different things to different people, and consists of many techniques and methods. These include peer and formal review sessions, formal methods, software metrics, and methods that focus on detecting language-based and programming problems. In spite of the varying methods, the goal is the same—to examine a program’s source code using some measurable procedure with the goal of detecting and removing prospective errors.

Historically, developers of early C compilers made a clear separation between static analysis and compilation. In the spirit of UNIX design, a program should do one thing and do it well. In this spirit, compiler writers designed their compilers to be as small and fast as possible, leaving static analysis to another program, called lint. Some feel this approach was a mistake. As Peter van der Linden points out, many programmers do not use lint for semantic analysis, so we get faster compilation, but at the cost of allowing many detectable bugs to get past the compiler.4 Today, most compiler vendors implement stricter semantic checking in their compilers. For example, gcc and g++ provide a wide range of options for detecting semantic errors in source code, and Sun’s CC compiler contains options that are even more advanced.

One of the easiest and more productive static code analysis techniques is to use your compiler’s warning flags to detect programming errors. During the development process, your first line of defense is compiler options. By intelligently using compiler options, you can use the compiler to alert you to potential problems in your source code early in the development process.

To use Project Builder for semantic code analysis of C and C++ code, you need to understand gcc’s compiler flags. By enabling these flags, you tell the compiler to perform stricter semantic checking when processing source code. The gcc manual groups warnings into the following categories:

- Warning options
- C language options
- C++ language options

---

Warning messages tell the compiler to check for language or programming constructs that are potentially dangerous or may lead to errors or unexpected results. This is one of the most useful sets of options supplied by the compiler. Both C and C++ language options define a set of options that detect and verify conformance with various dialects of C, C++, and Objective C. These options are useful if you wish to check your code for conformance to a particular language standard.

For example, the –Wall option collects many useful compiler flags under a single switch, and the –W option adds even more checking. Including these two options in your build is a great way to perform basic semantic code analysis.

To set compiler warning flags within Project Builder, follow these steps:

1. Select a target and click on the Build Settings tab of the Edit pane (changed in Mac OS X 10.2 to a hierarchical list of settings with subpanes in Expert view).
2. In the list of build settings, double-click on the value section of the WARNING_CFLAGS record (see figure 3.25). Use this field to add any additional warning compiler flags to the build. Or, under the Compiler Setting section of the Edit pane, use the Other C Compiler Flags text field to add compiler flags.

### 3.4 Creating an application with Interface Builder

The cornerstone of developing Mac OS X programs using the Apple development tools is Project Builder. You use the Project Builder environment to write your program’s source code and build, run, and debug your program. However, for developing GUI-based applications under Mac OS X, this is only half the story. In addition to implementing the program’s logic, you also need to create its user interface. Enter Interface Builder.

You use Interface Builder to design the user interface component of your program. The relationship between Project Builder and Interface Builder is similar to that of Project Builder and the UNIX-based development tools. As you know, Project Builder uses the services of the UNIX-based development tools to perform common development tasks. For creating user interfaces, it uses Interface Builder. With Interface Builder, you design application menus, windows, icons, and dialog boxes that provide your application with its GUI.

The best way to understand the components of Interface Builder and its interaction with Project Builder is to see it in action. If you have not already done so, go through section 3.3 to get a feel for how Project Builder works.
3.4.1 Interface Builder scenarios

The following sections describe typical situations you will encounter when constructing your program's GUI with Interface Builder. These topics will give you a taste for some of Interface Builder’s most useful features.

Nib files

Under Mac OS X, you construct a Cocoa application's user interface using Interface Builder and store this information in one or more Nib files. Nib files come from the days of NeXT computer and stand for NeXT Interface Builder. Generally, a Nib file holds application interface components.

---

For example, the Nib file for a Cocoa program not only contains its user interface components (menus, windows, and so on), but also encodes and stores information about each object and the relationship between these objects within the object hierarchy. The runtime system decodes this information when the program is loaded.

Let’s look at the contents of a Nib file using a command-line program called nibtool. The nibtool program lets you display different information from a Nib file through its command-line options. For example, the –c option displays the local classes in a Nib file, –j outputs the setting for the objects, and –x prints connections between the objects. To experiment with this program, open a shell and change to a directory that holds a Nib file (usually under a project’s English.lproj directory). Listing 3.4 shows the condensed output of a nibtool command.

Listing 3.4 Information from a Nib file, displayed with nibtool

```bash
% nibtool -c MainMenu.nib
/* Classes */
Classes = {
  IBClasses = {
    {CLASS = FirstResponder; LANGUAGE = ObjC;
      SUPERCLASS = NSObject; },
    { ACTIONS = {clearMe = id; clickMe = id;
                myMenuAction = id; },
      CLASS = MyClass;
      LANGUAGE = ObjC;
      OUTLETS = {textItem = id; },
      SUPERCLASS = NSObject;
    }
  },
  IBVersion = 1;
}; /* End Classes */
%
% nibtool -j MainMenu.nib
Objects = {
  "Object 1" = {
    Class = "NSCustomObject";
    CustomClass = "NSApplication";
    Name = "File's Owner";
    className = "NSApplication";
  },
  "Object 2" = {
    Class = "NSView";
    autoresizingMask = "0";
    frameRect = "{{1, 9}, (404, 148)}";
    groupedIBObjectID = "<null>";
  }
};
```
Creating and editing menus, windows, and other interface objects

The usual way to use Interface Builder is in conjunction with Project Builder. Typically, you create a new project within Project Builder and edit its user interface using Interface Builder. From within Project Builder, you double-click on the application’s main Nib file, located in the Resource folder, to launch Interface Builder and load the Nib file. At this point, you can edit existing interface components or create addition interface elements.

When you open an application’s Nib file in Interface Builder, you will see a window that holds the application’s menu (the menu displayed at the top of the screen when the application is running). You can change the text of an existing menu item by double-clicking on its name and editing the text. To add a new menu item, click on the Cocoa Menus item in the Palette toolbar (see figure 3.26), select the item you wish to add, and drag it to its location within the menu window. Dragging it over a menu item opens the menu so you can place the item in the menu. You can add a single menu item by selecting the Item menu item, or choose a predefined menu item from the palette that already contains the menu item.

To delete an item, select it and press the Delete key. Make sure you read the Mac OS X User Interface guidelines to ensure that your application’s menus are stylistically correct.

Within Interface Builder, windows and other interface components are easy to construct, customize, and add to your program. For example, to add a new window to your program, simply select the Cocoa Windows item from the palette toolbar and drag it outside of the palette. Doing so creates a new window and adds it to your application instance. Creating other components is just as simple. Even
better, you can connect components entirely in Interface Builder if all you need is to have one component respond to another; you need code only to add functionality. You will learn more about creating interface components in chapter 6.

**Linking interface components to code**

Once you have defined your application’s user interface in Interface Builder, you need to add code to handle the user interaction with the interface. You do so in Interface Builder as follows:

1. Lay out your interface components.
2. Create a new class for an interface component.
3. Create the files for the class.
4. Create an instance of the class you just created.
5. Make a connection between the instance and the interface component.
6. Add implementation code to the skeleton classes with Project Builder.

Let’s tackle each of these steps through an example:

1. Launch Project Builder. Create a new Cocoa project by selecting File→New Project and choosing Cocoa Application from the project list.
2. Click the Next button, save the project as InterfaceBuilderExample, and click the Finish button.
Let’s add a few simple interface elements to the main window. Select Cocoa Views from the Palette window and drag a button and text field to the window. Place them anywhere you like and resize the window for the new controls. Double-click the button and rename it Click Me (see figure 3.27).

Create a class to implement the actions associated with the interface items. To do so, click the Classes tab in the MainMenu.nib window, select NSObject from the class browser (far-left window), and press Return. Call the class MyObject.

To add instance variables to the class, select MyObject from the class list and pressing Shift-Command-I to bring up the Class Info window. In this window, you add instance variables to the class—in this case, one per interface item. In Cocoa applications, instance variables are called outlets and instance methods are called actions. For this example, create one outlet to hold the contents of the text field and one action to respond when the user clicks the Click Me button.

In the MyObject Class Info window, click the Add button and name the outlet textItem. Click the Actions tab, click Add, and name the action clickMe. This action responds to clicks on the Click Me button.

Create the class’s source files. Make sure MyObject is selected in the Class list, select Classes→Create Files For MyObject, and click the Choose button to save the files. Interface Builder creates the interface and implementation files for the class and merges them into the Project Builder project.

Create an instance of the class. Select MyObject from the Class list and select Classes→Instantiate MyClass, which creates a new icon in the instances pane representing the instantiation of the class MyObject (see figure 3.28).

Now comes the important step: forming relationships between the class instance and its corresponding interface components. You are graphically telling the system what you usually do in code. Make sure Interface Builder is displaying the application window that contains the text field and Click Me button. To form a relationship for an outlet, click on the
MyObject instance while holding down the Control key and drag to the appropriate interface control. For example, to form a relationship between the instance and the text field, Control-click the MyObject instance and drag to the text field. Choose textItem from the outlets list and click the Connect button to form the connection (see figure 3.29).

11 Repeat for the Click Me button, but this time, Control-click and drag from the Click Me button to the MyObject instance. Select clickMe from the actions list (make sure the target is selected) and click the Connect button. By changing the direction of the Control-drag, you specify that the button is sending a clickMe message to MyObject.
Save the file and return to Project Builder. Locate the MyObject header file and click on its icon. Notice that Interface Builder added the instance variable’s `textItem` to the file. Now, all that remains is to add code to the `clickMe` method to place a text string into the text field. Open the implementation file (MyObject.m) and add the following code to the `sender` method.6

```swift
- (IBAction)clickMe:(id)sender
{
    // Place a static string in the text field.
    [textItem setStringValue: @"Hello World!"];
}
```

Build and run the project (Command-R). When the program displays the main window, click the Click Me button; the result appears in figure 3.30.

![Figure 3.30](image)
The final window for the sample program, after clicking the Click Me button

This is a very basic example, but it shows some of the fundamentals you will use when building programs that are more complex.

**Testing an interface**

During the development of your program’s user interface, things can change quite a bit as you discover more about what functionality you want. It’s useful to test the look and feel of the interface as you are laying it out, without recompiling the entire project. For example, it’s convenient to construct your application’s interface and play with it as you go until you are satisfied it’s correct. Interface Builder provides this functionality through the Test Interface feature. The Test Interface feature displays the application’s user interface, enabling you to test it without invoking Project Builder.

---

6 See [http://www2.latech.edu/~acm/HelloWorld.shtml](http://www2.latech.edu/~acm/HelloWorld.shtml) for a collection of Hello World examples in various programming languages.
To use this feature, construct your user interface and select File→Test Interface or press Command-R from within Interface Builder. Interface Builder displays your application’s interface, enabling you to use it and see if it's what you want. To exit the interface test, select Quit from the application menu (Command-Q).

3.5 Summary

This chapter has taken you through some of the basic features of Project Builder, Apple’s main IDE for building Mac OS X applications; and Interface Builder, the application used to create your program’s user interface. You've seen how to use these programs to create a simple Cocoa application and walked through some common scenarios that come daily when developing programs with Project Builder. You've also learned that Project Builder continues the development of IDE-based development environments for the Macintosh, but breaks with the past by using external UNIX-based development tools such as gcc, g++, gdb, RCS and CVS for implementing build and version control commands.

Armed with this knowledge, you are well on your way to creating your own Mac OS X applications with Project Builder and Interface Builder. In chapter 4, I will move on to discuss the details of the different development options available under Mac OS X. In chapters 5–7, I show how to write more advanced, fully functioning applications using Cocoa and AppleScript.