Chapter
3

Read this chapter if you know the basics of CM3-IDE and would like to learn how you can build packages and share them with others in your team.

3. Building and Sharing Packages

This chapter covers the basics of building and sharing packages in CM3-IDE. It also describes how CM3-IDE facilitates the building of large, multi-developer projects.

Chapter Organization

Each section of this chapter explores of a particular aspect of building and sharing packages with CM3-IDE.

Building Packages on page 48 describes how to build a package by invoking CM3-IDE's builder.

Directory Structure of a Package on page 49 illustrates the directory structure of a basic CM3-IDE package.

CM3-IDE Makefiles on page 50 defines the basic syntax for CM3-IDE makefiles.

Managing Multiple Packages on page 53 shows how to divide your projects into multiple packages.

Shipping Packages on page 54 describes how to ship a package to make it available for importing.

Sharing Packages on page 57 explains how to share packages in a multi-developer team with CM3-IDE.

Builder Options on page 62 lists the command-line options available for CM3-IDE's builder, `cm3`.

3.1 Building Packages

Along with navigational links, many of CM3-IDE's screens include associated actions. For example, a package summary page may allow a "build" action to bring the package up-to-date, or a module summary may allow an "edit" action to edit the source file for that module.

Most CM3-IDE pages include buttons for valid actions. For example, the Build Build button denotes the "build" action on package pages.

Step Starting from the CM3-IDE start screen, click on the P Packages icon. Following any of the links under P "proj" packages—your private directory—will lead you to a package summary page. If you have followed **Learning the Basics** on page 5 properly, you will see at least two links: `hello` and `MyPackage`.

Summary pages of your packages or their components always include a Build button.

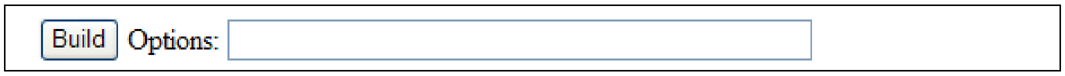


Figure 25. CM3-IDE's Build Button

Clicking on the Build button will start CM3-IDE's builder, and display the build results on the screen. If there are any errors, CM3-IDE displays hypertext links to errors in your source files.

CM3-IDE's builder is called "`cm3`", short for *Critical Mass Modula-3*. Indeed, `cm3` is a stand-alone compiler/builder for the Modula-3 language that is integrated within the CM3-IDE environment.

At the start of a build, `cm3` first looks for a makefile for the current package. (Makefiles in CM3-IDE are named "`m3makefile`".) If it can't find a makefile, it attempts to build a program from the files in your package directory. While you don't always need to create a makefile, it is a good idea to create one for clarity.

CM3-IDE's makefiles are discussed in more detail later in this chapter. You can find more information about customizing the behavior of the Build button in **Customizing CM3-IDE** on page 65, or in the `cm3` configuration file (`cm3.cfg`) in your CM3-IDE installation.

CM3-IDE's Builder

CM3-IDE's combined builder and compiler, `cm3`, has been designed specifically for the creation of robust and distributed programs.

When you click CM3-IDE's Build button, CM3-IDE invokes `cm3` to build your program. You may also invoke `cm3` from the command-line by issuing the command `cm3` from your command-line shell. See **Builder Options** on page 62 for more information on running `cm3` from the command-line.

3.2 Directory Structure of a Package

Each package in CM3-IDE resides in a directory, with sources in a source subdirectory, and generated files in a derived subdirectory.

Names of CM3-IDE's derived directories:
ALPHA_OSF
HPPA
IRIX5
IBMR2
LINUXELF
NT386
SOLsun
SOLgnu
SPARC

The source directory for a package is named "`src`"; its contents are the same on all platforms. In contrast, the name and contents of the derived directory for a package varies from one platform to another.

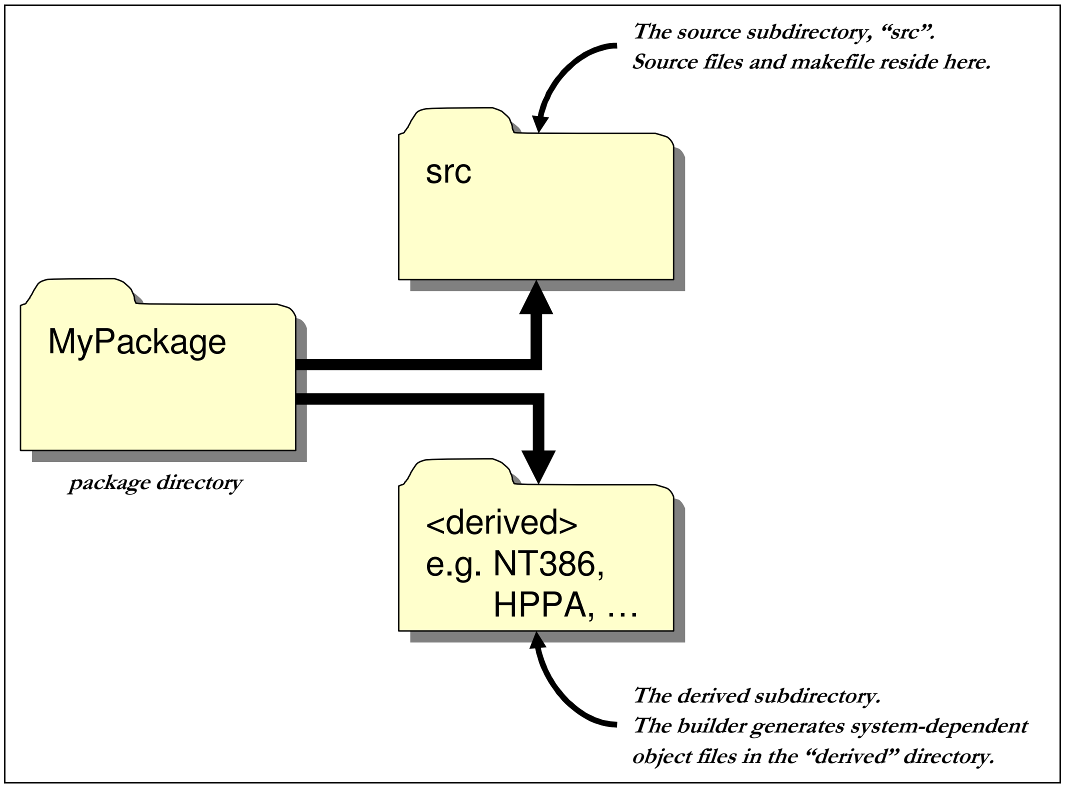

The source subdirectory, "src".
Source files and makefile reside here.
src
MyPackage
package directory
<derived>
e.g. NT386,
HPPA, …
The derived subdirectory.
The builder generates system-dependent object files in the "derived" directory.

Figure 26. CM3-IDE Package Directory Structure

The name of the derived directory denotes the platform where CM3-IDE built the system, for example, `NT386` is the platform name for Win32 running on Intel x86 processors, and `HPPA` is the name for HP/UX running on HP Precision Architecture series.

The default names for derived directories are:

`ALPHA_OSF`	Digital Unix (OSF/1) on DEC Alpha
`HPPA`	HP/UX on HP Precision Architecture
`IRIX5`	SGI Irix on SGI/MIPS
`IBMR2`	AIX on IBM RS/6000
`LINUXELF`	Linux/ELF on Intel x86
`NT386`	Win32 (Win95 or NT) on Intel x86
`SOLsun`	Solaris 2 on SPARC (Sun C compiler)
`SOLgnu`	Solaris 2 on SPARC (GNU C compiler)
`SPARC`	SunOS 4 on SPARC

The separation of source and derived files is useful when building larger programs, because it:

- isolates source files for backup, revision control, and searching
- enables sharing the same source tree across operating systems and architectures, without confusing object files from different platforms.

This arrangement, combined with CM3-IDE's multi-platform libraries, simplifies the management of large, multi-platform programs.

3.3 CM3-IDE makefiles

If you have worked through the tutorials in **Learning the Basics** on page 5, you've already seen and used a basic makefile. This section describes makefiles in more detail.

A CM3-IDE makefile (named `m3makefile`) is a small script which tells how to build a package. Most instructions in a makefile are calls to pre-defined functions of CM3-IDE's builder, such as "`import`", or "`program`". Together with the builder, these predefined functions replace the need for a "`make`" utility. Function calls replace declarations in the makefile, and the builder takes care of the dependencies between modules.

Working with Makefiles

To view a makefile, navigate to the Package Summary page, and click the makefile name under "Quake sources".

To edit a makefile from any package, click the button labeled "Edit m3makefile". CM3-IDE will start your text editor and open the makefile for the current package.

Here is an example of a simple CM3-IDE makefile:

```
% m3makefile for SimplePkg
import("libm3")
import("ui")
module("EditWindow")
implementation("Editor")
program("editor")
```

A CM3-IDE makefile is a script in a simple programming language called *Quake*, used by both CM3-IDE and `cm3`. You can find more information about Quake in the CM3-IDE on-line help under `/help/cm3/quake.html`. Nonetheless, coding simple makefiles will not require much knowledge beyond what is described here.

3.3.1 Basic Makefile Commands

The following are the commands used most often in CM3-IDE makefiles.

Most makefiles start with one or more import commands:

```
import( "package-name" )
```

The import command specifies a package to be imported in the build process. Any package that builds a library may be imported.

Most programs import the standard Modula-3 library, called `libm3`, via the command:

```
import( "libm3" )
```

Declaring Sources. The following commands declare the source files in your package that are to be included in the build:

```
interface( “X” )
```

Declares that the file `X.i3` contains an interface. All interface files that you want to include in your build must appear in interface commands. Don’t forget to leave off the “`.i3`” extensions.

```
implementation( “X” )
```

Declares that the file `X.m3` contains a module. All module files that you want to include in your build must appear in an implementation command.

```
module( “X” )
```

Declares `X.i3` and `X.m3` with one command. The `module` command is really a short-hand for doing both an `interface` and an `implementation`. This command is used most often, as many CM3-IDE modules consist of a single interface and implementation.

```
generic_interface( “X” )
generic_implementation( “X” )
generic_module( “X” )
```

Similar to their non-generic counterparts, the `generic_interface`, and `generic_implementation` commands declare the generic interface and implementation files to be included in the build. The command `generic_module` is a shorthand for calling `generic_interface` and `generic_implementation`. Generic interface and module files use the “`.ig`” and “`.mg`” extensions. See **Generics: Reusable Data Structures and Algorithms** on page 97 to learn more about generics.

Making sources visible to others. The above calls declare their arguments to be built, that is, but visible only within the current package. To declare an interface so that other team members can access it, you use the capitalized version of the same command, for example:

```
Interface(“X”)
```

and

```
Module (“X”).
```

Programs and Libraries. To tell the builder to build an executable, include the `program` command at the end of your makefile:

```
program( “executable-name” )
```

A makefile may have only one `program` command. It specifies what to name the program executable. Use the capitalized version of this command, `Program`, to make the program available to other developers.

If the makefile is describing a collection of interfaces and modules that are designed to be a library to be used by other packages, use `library` instead of `program`:

```
library ( “library-name” )
```

3.3.2 Additional Makefile Commands

Many standard packages in your CM3-IDE distribution define new makefile commands. Importing these packages makes the new commands available. For example, the standard library `libm3` includes a makefile command `bundle` which will bundle a file in your source directory so that it’s available at run-time.

For reference information about makefile commands and their syntax, see CM3-IDE’s on-line help under `/help/cm3/cm3.help`.

3.4 Managing Multiple Packages

Building a large and complex project as one package is certainly possible but probably not wise. Even if you are programming on a project by yourself, you may want to divide your project into more manageable pieces.

Suppose you are building an editor library and a number of editor programs which, using the editor library, support editing of various file format.s

A natural division of your code would be to put the core editor functionality in one library package (called `libedit` in this example), and put each editor incarnation (called `html-editor` in this example) in its own program package. To create an editor program, you import the `libedit` library and add some additional formatting code and a main module. Building such a package results in an editor program.

The makefile for the `libedit` package would look like:

```
% makefile for edit library
Module(“Edit”)
…other makefile statements…
Interface(“Format”)
Library(“libedit”)
```

The commands `Module`, `Interface`, and `Library` are capitalized to denote that the corresponding interfaces, and the library should be made available to other packages.

The makefile for the `html-editor` package would look like:

```
% makefile for html-editor
import("libm3")
import("libedit")
module("HTMLFormat")
implementation("HTMLEditor")
program("html-edit")
```

How do you make the `libedit` library available to `html-editor`'s build? The simplest way to make the functionality of a package available to other packages is the `ship` command. To make the library package `libedit` available for reuse in the program package `html-editor`, you need to ship the `libedit` library, first.

After you've successfully built the `libedit` package, you may ship it by clicking on the Ship Ship button on its package summary page.

After shipping the `libedit` library, you can build the `html-editor` package, which depends on the `libedit` package.

3.5 Shipping Packages

Shipping a package makes the contents of the package available to other packages in your system. The Ship Ship button, located above the Build button on any package page ships the current package.

Shipping does not modify the private copy of your package. It simply copies the essential parts into a public version of the package.

You may continue working on your private copy, and ship another version at your convenience.

Once shipped, CM3-IDE will keep track of two copies of your package:

- a private copy, which you just built. You can continue to change and build this copy without affecting other packages. In the default settings, this copy would resides under `/proj` in CM3-IDE's namespace.
- a public copy of your package which is available to other packages. In the default settings, this copy would reside under `/public` in CM3-IDE's namespace.

Shipping Packages

Shipping a package makes its contents available to other packages in your system. Once shipped, a package can be imported into other packages, and it's listed with the public packages.

To ship a package from CM3-IDE, you can click the Ship button. CM3-IDE copies the contents of the package to the public package root. From the command-line shell, you use the command "`cm3 -ship`" to ship packages.

3.6 Package Roots

So far, we have discussed two kinds of packages:

- your private packages, listed under "`proj`" packages in the CM3-IDE P Packages page. The two you created in **Learning the Basics** on page 5, `hello` and `MyPackage`, are examples of private packages. Their respective URLs are `/proj/hello` and `/proj/MyPackage`.
- public packages, listed under "`public`" packages in the CM3-IDE P Packages page. The standard library package, `libm3` which you have imported into your own packages is an example of a public package. Its URL is `/public/libm3`.

In CM3-IDE, packages are kept in directories called *package roots*. The package roots `proj` and `public` are examples. CM3-IDE is pre-configured to use `proj` for your private packages, and `public` for the public packages. Indeed, you can create new package roots to organize your projects, or coordinate sharing with others. For example, you may use a package root `graphics` to contain all the graphics-related packages in your development group.

Within CM3-IDE, the name of a package root resides at the top level of CM3-IDE's namespace. For example, the package root `graphics` would map to `/graphics`, and the `html-editor` package contained in the `graphics` root would map to `/graphics/html-editor`.

On your filesystem, a package root is represented as a directory that contains zero or more packages. CM3-IDE supports building or browsing packages in multiple package roots. You can add new package roots by using CM3-IDE's configuration screen. (See **Customizing CM3-IDE** on page 65.)

3.6.1 Example: Creating a New Package Root

In this example, we create a new package root and configure CM3-IDE to add the new package root to its database.

Step **Create a directory for a package root.** To create and configure CM3-IDE to use a package root, you must first create a directory for the package root. To do so, create a directory on your filesystem, such as:

```
D:\users\harry\graphics   (Win32)
/usr/harry/graphics       (Unix)
```

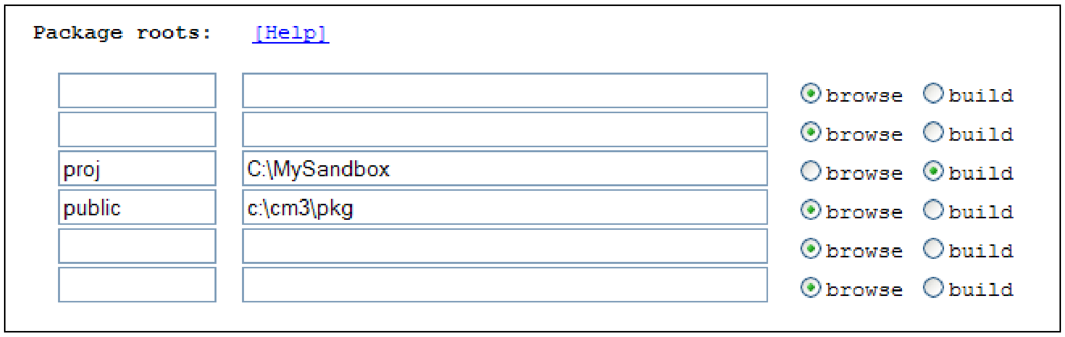
Step **Navigate to the package roots settings.** From the start page, click on the Configuration icon to visit CM3-IDE's configuration screen. The second section of CM3-IDE Configuration is labeled "Package Roots." The Package Roots section of the configuration page is where you list the package roots in your system.

The package roots specified in this section are scanned periodically by CM3-IDE. Each root is either available for building or browsing. When you are sharing packages with others, you should use the "browse" option, so that you don't accidentally alter their code. You configure your own package roots to allow building.

By default, your CM3-IDE installation comes with two pre-defined roots:

- Your private packages, listed under "`proj`" packages in CM3-IDE, reside in the directory (`$HOME/proj` on Unix, or `%HOME%\proj` on Win32).
- The public packages, listed under "`public`" packages in CM3-IDE, reside in the `pkg` subdirectory of your CM3-IDE installation.

When you ship a package, the contents of your package are copied to the public packages, making them available to other programmers in a controlled fashion.



Package roots: [Help]

		[✓ browse] build
		[✓ browse] build
proj	C:\MySandbox	browse [✓ build]
public	c:\cm3\pkg	[✓ browse] build
		[✓ browse] build
		[✓ browse] build

Figure 27. Package Roots Section of the Configuration Screen

Step **Back to the example.** Next, choose one of the blank rows to specify your new package root. You need to specify three pieces of information about a root: its name in

CM3-IDE's namespace, its filesystem path, and whether it is to be used for building or browsing:

Step

- **Choose the name**.

  Specify a short name for the new package root in the left-most field. You will use this short name to refer to this package root in CM3-IDE. Choose a short and descriptive name like "`graphics`" which will be known as `/graphics` within CM3-IDE. If the new root name collides with existing root names, CM3-IDE will substitute something less useful, like "`Root001`".

Step

- **Specify the path to the package root**.

  Specify the absolute path to the directory where this package root resides. CM3-IDE will periodically scan for packages from the path you specify. Examples are:

`D:\USERS\HARRY\GRAPHICS`	(Win32)
`/usr/harry/graphics`	(Unix)

Step

- **Enable either building or browsing for this package root**.

  Check the option "build" to make packages in this root available for editing and building.

  If you are configuring a new root to browse packages belonging to other developers, make sure to check the "browse" option so that you don't corrupt their packages inadvertently. (The public package root is an example of a browse-only package root, while your private package root, `proj`, allows building and editing of packages.)

Step When you finish entering the information about the new root, click on Save and apply changes Save and Apply Changes.

3.7 Sharing Packages

This section describes how to share a package with other developers and how to access its functionality.

Imagine that you are working on a large project as part of a team of programmers. You've been assigned to write a library that will be used by other programmers. To test and run their code, they need a stable copy of your library available to them at all times. What if you're still working on your code? How do you make sure they are using the most current version? How do you test and change your revised version of a library while simultaneously allowing others to continue their work based on a stable release of your package?

The answer in CM3-IDE, as you might have guessed, is shipping. In CM3-IDE, you ship your packages to copy them to the public package root, where they are available to others in your team. Each CM3-IDE installation has one public package root. The code you ship becomes available for browsing and importing by others, but they may not edit or compile it.

In contrast, you, who shipped the package in the first place, are considered to be the owner of this package; you are responsible for its upkeep. The buildable sources for your packages remain in your private directory; they are in your complete control. In fact, shipping a package does not affect its contents in any way.

You may continue to work on your package after shipping it. Whenever you are comfortable with your changes, you may ship the package again, making the new changes available to others. CM3-IDE will overwrite the old shipped files in the public package root with the new ones.

3.7.1 Example: Adam's and Eve's Joint Project

In the following example, we examine a project that involves multiple programmers and multiple packages. We'll review some of the concepts discussed earlier in the context of a multi-developer project.

Imagine two developers Adam and Eve working on a shared project Garden. As smart developers, they have decided to use CM3-IDE for their Garden development. As organized developers, they first set up their environment to make sharing easy.

Creating directories for package roots. The first step is to create a directory that contains both of their package roots (presumably with group write permissions):

```
E:\GARDEN\         (Win32)
/proj/garden/      (Unix)
```

Next, they create package roots for themselves:

```
E:\GARDEN\EVE      (Win32)     /proj/garden/eve   (Unix)
E:\GARDEN\ADAM     (Win32)     /proj/garden/adam  (Unix)
```

Packages in Eve's directory are her responsibility. Adam may be able to browse the code in Eve's packages, but he should not be able to modify the package contents. If Adam needs a change in one of Eve's packages, he should ask Eve to make the change.

Configuring package roots. Next, Adam and Eve use the configuration page of CM3-IDE to set up their package roots.

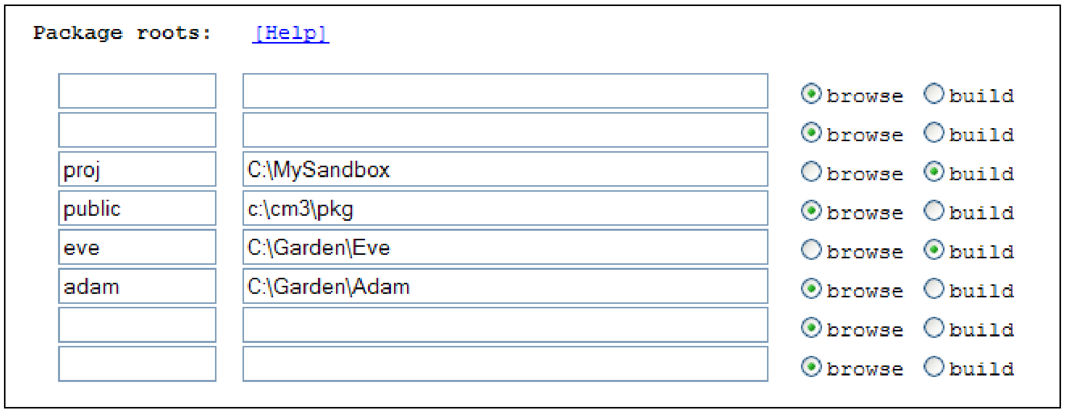
Eve types the package root name "`eve`", with the path set to the full path to the directory she just created for her packages:

```
E:\GARDEN\EVE (Win32)
/proj/garden/eve (Unix)
```

Eve checks the "build" option for her own package root, so that she can build new packages in her newly created package root.

Eve wants to be able to look at Adam's packages but she wants to make sure she doesn't accidentally do anything to them. So Eve includes Adam's package root, but she is careful not to check the "build" option. The only other choice is the "browse" option.

Eve is ready to leave the Configuration page. The package root section of her configuration looks like:



Package roots: [Help]

		[✓ browse]	build
		[✓ browse]	build
proj	C:\MySandbox	browse	[✓ build]
public	c:\cm3\pkg	[✓ browse]	build
eve	C:\Garden\Eve	browse	[✓ build]
adam	C:\Garden\Adam	[✓ browse]	build
		[✓ browse]	build
		[✓ browse]	build

Figure 28. The "Package Roots" section of Eve's Configuration Page

Saving changes to the configuration. Before leaving the Configuration page, Eve clicks the Save and Apply button.

Adam sets up his package roots in a similar fashion, but in Adam's configuration, Adam's package root is available for building and Eve's is only available for browsing.

Note that Adam and Eve could have chosen different names for their roots, but then communicating would have been harder.

Assigning project responsibilities. Now that they have organized their development environment, Adam and Eve meet to decide how to break up the work:

- Adam agrees to work on the end application, `pie`. The package `pie` will reside in Adam's package root, and will be available in Adam's and Eve's CM3-IDE as `/adam/pie`.

- Eve agrees to work on the core library, named `apple`. The package `apple` will reside in Eve's directory, and is available as `/eve/apple`.
- Adam's package, `pie`, will need Eve's package, `apple`.

Adam and Eve are the owners of their respective packages. No one else is allowed to modify sources in someone else's package root. Working in a team however, each allows the other to browse the current state of their Garden-related packages. (CM3-IDE does not enforce this policy; you must use your filesystem to do that.)

Getting ready to code. Adam and Eve spend some time discussing the design of the application until they agree on an initial set of interfaces that Eve's `apple` needs to support. The separation of interface from implementation in CM3-IDE is key to allowing Adam and Eve to work independently:

- Eve starts the first implementation of the `apple` interfaces.
- Adam starts the design and implementation of `pie`, assuming the agreed upon `apple` interfaces.

Shipping the first release. When Eve is satisfied with `apple`, she ships it. Now, there are two copies of the `apple` package, Eve's private copy (`/eve/apple`) and the public copy (`/public/apple`). Once Eve ships her package:

- Adam can import `apple` in the makefile for `pie` and build it.
- Eve may continue working on `apple`, shipping it whenever she finds a proper checkpoint where the code is stable.

Testing before a release. If `apple` becomes too complex, Eve may need to "unit test" its functionality before shipping so that Adam's `pie` is not affected by new bugs.

No problem: Eve can create her own package `juice` just for the purpose of testing the quality of `apple`. As a savvy developer, Eve also advertises `juice` as a sample program that uses `apple`. This is convenient for Adam since he can see the `juice` package as `/eve/juice` in his CM3-IDE's namespace even if Eve doesn't ship it. Moreover, because he configured `juice` for browsing only in his CM3-IDE configuration, Adam can't corrupt `juice` by accident.

Eve is left with one problem: `juice` is supposed to test `apple` before `apple` is shipped, but since `juice` needs to import `apple`, `apple` needs to be shipped before `juice` can test it.

How does Eve overcome this problem? Simple: she needs to force `juice` to use her private version of `apple` (`/eve/apple`) instead of the publicly available version (`/public/apple`).

Overriding a build. When building `juice`, Eve will need to tell CM3-IDE not to look in the public package root for the shipped `apple`. Instead, she must specify where to get `apple`; this is called overriding.

To override a build, Eve must perform two steps:

First, she must create an overrides file (named "`m3overrides`") in `juice`'s source directory that includes the names of the overridden packages and where to find them. In this example, Eve puts:

```
override("apple", "C:\\GARDEN\\EVE")      (Win32)
override("apple", "/proj/garden/eve")     (Unix)
```

in the `m3overrides` file.

Then, when building `juice`, Eve types "`-override`" as the build option to tell the builder that it should use the overrides file.

Finally, Eve compiles her `juice` package, tests `apple`, and when she is happy with the quality of `apple`, she ships it. Next time Adam builds `pie`, the builder will notice the fresh `apple` and will use it.

How Overrides Work. An overrides file contains a set of override commands to specify new paths for the builder to find packages. Each line of the override list contains the name of the package to replace and the path to the package root containing the new package, i.e.:

```
override(package,replacement-package-root)
```

where `package` and `replacement-package-root` are strings.

Note that you must escape the backslash character on Win32 by typing "\\" as your path delimiter.

When the `-override` option is specified, `cm3` looks for a file named `m3overrides` in the package's source directory. If the file `m3overrides` exists, it is evaluated prior to evaluating `m3makefile`. (Both `m3overrides` and `m3makefile` are Quake scripts.)

CM3-IDE allows you to leave permanent overrides in your makefiles but it isn't a good practice. By keeping all override calls in an `m3overrides` file and not in a makefile, you can readily switch between building packages based on private and public versions of imported packages without editing files.

The overrides in effect when a package was built are automatically carried forward into importers of the package, so there is no need to restate the complete set of overrides in every package, only of those packages that are directly imported into the current package. CM3-IDE's builder will warn you if you overspecify your override options.

Shipping and the -override Option. Shipping a package that is built with overrides makes little sense, as it depends on packages that are not available in the public package root. CM3-IDE's builder will refuse to ship a package that was built using overrides. This safety check helps ensure that packages shipped to the public package root stay consistent.

3.8 Builder Options

CM3-IDE's builder options are listed here. You can find this information on-line by specifying "`-help`" as your build option, or typing "`cm3 -help`" at a shell command-line.

Modes.

`-build`	compile and link
`-ship`	install package
`-clean`	delete derived files
`-find`	locate source files

(default: `-build`)

Compiler Options.

`-g`	produce symbol table information for the debugger
`-O`	optimize code
`-A`	disable code generation for assertions
`-once`	don't recompile to improve opaque object code
`-w0..-w3`	limit compiler warning messages
`-Z`	generate coverage analysis code

(default: `-g -w1`)

Program and Library Options.

`-c`	compile only, produce no program or library
`-a lib`	build library lib
`-o pgm`	build program pgm
`-skiplink`	skip the final link step

(default: `-o prog`)

Messages.

`-silent`	produce no diagnostic output
`-why`	explain why code is being recompiled
`-commands`	list system commands as they are performed
`-verbose`	list internal steps as they are performed
`-debug`	dump internal debugging information

(default: `-why`)

Information and Help.

`-help`	print this help message
`-?`	print this help message
`-version`	print the version number header
`-config`	print the version number header

Miscellaneous.

`-keep`	preserve intermediate and temporary files
`-times`	produce a dump of elapsed times
`-override`	include the ".m3overrides" file
`-x`	include the ".m3overrides" file
`-Dnm`	define the quake variable nm with the value TRUE
`-Dnm=val`	define the quake variable nm with the value val
`-console`	produce a Windows CONSOLE subsystem program
`-gui`	produce a Windows GUI subsystem program
`-windows`	produce a Windows GUI subsystem program

CM3-IDE's makefiles are discussed in more detail later in **CM3-IDE Makefiles** on page 50. For information about customizing the behavior of the Build button, see **Customizing CM3-IDE** on page 65, or review the configuration file `cm3.cfg` in your installation.

3.9 Summary

A *package* is the unit of building and shipping in CM3-IDE. Each package is represented by a directory on the filesystem. Packages typically have two subdirectories, a *source directory* (named "`src`") and a *derived directory* whose name varies per platform. Each package may have a *makefile* in its source directory which is a set of instructions for compiling sources in the package.

To *build* a package, use the Build button on the summary page of the package.

To make a package available for importing by other packages, *ship* it, by clicking the Ship button on the summary page of the package.

CM3-IDE builder. The stand-alone program `cm3` is CM3-IDE's builder. You may start the `cm3` builder by either clicking the Build button from any package page, or by invoking `cm3` from the command line. See **Builder Options** on page 62.

Package roots. You can organize packages for your various projects into package roots. A package root is a directory that can contain zero or more packages. See **Package Roots** on page 55.

The public package root is CM3-IDE's shared repository for shipped packages. Packages in the public package root can be browsed and imported by anyone but they cannot be edited or compiled in-place.

Overriding. The `-override` option can be used to tell `cm3` to look for a file named `m3overrides` in the source directory and, if it exists, evaluate it immediately before evaluating `m3makefile`. See **How Overrides Work** on page 61.