The consequence of programming in a world of live objects rather than with files and a text editor is that you have to do something explicit to export your program from your Pharo image. The old way of doing this is by creating a *fileout* or a *change set*, which are essentially encoded text files that can be imported into another system. The preferred way of sharing code in Pharo is to save packages and share them using a versioned repository on a server. Up to Pharo 5.0, this is done using a tool called Monticello, and is a much more powerful and effective way to work, especially when working in a team. In the future Pharo will officially offer support for Git and hosting on servers such as GitHub, BitBucket and more.

We gave you a quick overview of Monticello, Pharo’s packaging tool, in Chapter: A First Application. However, Monticello has many more features than were discussed there. Because Monticello manages *packages*, before telling you more about Monticello, it’s important that we first explain exactly what a package is.

### 1.1 Packages: groups of classes and methods

We have pointed out earlier (in Chapter: A First Application) packages are more or less a group of classes and methods. Now we will see exactly what the relationship is. The package system is a simple, lightweight way of organizing Pharo source code that exploits a simple naming convention.

Let’s explain this using an example. Suppose that you are developing a framework to facilitate the use of relational databases from Pharo. You have decided to call your framework *PharoLink*, and have created a couple of classes such as *OracleConnection*, *MySQLConnection*, *PostgresConnection* but also
Sharing code and source control

DBTable, DBRow, DBQuery, and so on. These classes are placed inside a package called PharoLink. However, not all of your code will reside in these classes. For example, you may also have a series of methods to convert objects into an SQL-friendly format: Object>>#asSQL, String>>#asSQL, or Date>>#asSQL.

These methods belong in the same package as the classes. But clearly the whole of class Object does not belong in your package! So you need a way of putting certain methods in a package, even though the rest of the class is in another package.

The way that you do this is by placing those methods in a protocol (of Object, String, Date, and so on) named *PharoLink (note the initial asterisk). The *PharoLink protocols added to the package named PharoLink. To be precise, the rules for what goes in a package are as follows.

A package named Foo contains:
1. All class definitions of classes in the package Foo.
2. All methods in any class in protocols named *Foo or *foo (When performing this comparison, the case of the letters in the names is ignored.), or whose name starts with *Foo- or *foo-.
3. All methods in classes in the package Foo except for those methods in protocols whose names start with *: Because classes of package Foo can be also extended by other packages.

A consequence of these rules is that each class definition and each method belongs to exactly one package.

Accessing packages

The class RPackage represents packages. The class RPackageOrganizer implements a singleton that holds all the Pharo packages. The following expressions are examples of the queries that you can perform.

```plaintext
(RPackageOrganizer default packageNamed: 'AST-Core') definedClassNames
(RPackageOrganizer default packageNamed: 'AST-Core') extendedClasses
```

Basic Monticello

When you open the Monticello browser, you will see two list panes and a row of buttons, as shown in Figure 1.1. The left-hand pane lists all of the packages that have been loaded into the image that you are running; the particular version of the package is shown in parentheses after the name.

The right-hand pane lists all of the source code repositories that Monticello knows about, usually because it has loaded code from them. If you select a
1.1 Packages: groups of classes and methods

Figure 1.1: The Monticello browser.

package in the left pane, the right pane is filtered to show only those reposi-
tories that contain versions of the selected package.

The package-cache

One of the repositories is a directory named package-cache, which is a sub-
directory of the directory in which your image is running. When you load code
from or write code to a remote repository, a copy is also saved in the package
cache. This can be useful if the network is not available and you need to access
a package. Also, if you are given a Monticello (.mcz) file directly, for example
as an email attachment, the most convenient way to access it is to place it in
the package-cache directory.

Adding repositories

To add a new repository to the list, click the +Repository, and choose the
kind of repository from the pop-up menu. Let’s add an HTTP repository.

To do  Open Monticello, click on +Repository, and select HTTP. Edit the
dialog to read:

MCHttpRepository
  location: 'http://squeaksource.com/PharoByExample'
  user: ''
  password: ''

Click on Open to open a repository browser on this repository. You should
see something like Figure 1.2. On the left is a list of all of the packages in the
repository; if you select one, then the pane on the right will show all of the
versions of the selected package in this repository.
Browsing versions

If you select one of the versions, you can Browse it (without loading it into your image), Load it, or look at the Changes that will be made to your image by loading the selected version. You can also make a Copy of a version of a package, which you can then write to another repository.

As you can see, the names of versions contain the name of the package, the name of the author of the version, and a version number. The version name is also the name of the file in the repository. Never change these names; correct operation of Monticello depends on them! Monticello version files are just zip archives, and if you are curious you can unpack them with a zip tool, but the best way to look at their contents is using Monticello itself.

Creating a package

To create a package with Monticello, you have to do two things: write some code, and tell Monticello about it.

To do  Create a package called PBE, and put a couple of classes in it, as shown in Figure 1.3. Also, create a method in an existing class, such as Object, and put it in the same package as your classes – see Figure 1.4.

Committing a package

When you add a package via the class browser, Monticello automatically add the package to its list. PBE is now in list of packages; the package entry will be marked with an asterisk to show that the version in the image has not yet been written to any repository (It is said to be dirty).
**Figure 1.3:** Two classes and one extension in the PBE package.

**Figure 1.4:** An extension method that is also be in the PBE package.
Initially, the only repository associated with this package will be your package cache, as shown in Figure 1.5. That’s OK: you can still save the code, which will cause it to be written to the package cache. Just click Save and you will be invited to provide a log message for the version of the package that you are about to save, as shown in Figure 1.6; when you accept the message, Monticello will save your package. To indicate this, the asterisk decorating the name in Monticello’s package pane will be removed, and the version number added.

**Introducing a change**

If you then make a change to the package – say by adding a method to one of the classes – the asterisk will re-appear, showing that you have unsaved changes. If you open a repository browser on the package cache, you can select the saved version, and use Changes and the other buttons. You can of course save the new version to the repository too; once you Refresh the repository view, it should look like Figure 1.7.

To save the new package to a repository other than the package cache, you need to first make sure that Monticello knows about the repository, adding it if necessary. Then you can use the Copy in the package-cache repository browser, and select the repository to which the package should be copied. You can also associate the desired repository with the package by right-clicking on the repository and selecting add to package... and select the corresponding package. Once the package knows about a repository, you can save a new version by selecting the repository and the package in the Monticello Browser, and clicking Save. Of course, you must have permission to write to a repository. The PharoByExample repository on SqueakSource is world readable but not world writable, so if you try and save there, you will see an error message.

However, you can create your own repository and save your code there. Nowadays the favorite code repositories are http://www.smalltalkhub.org and http://ss3.gemstone.com. This is especially useful as a mechanism to share...
Figure 1.6: Providing a log message for a new version of a package.

Figure 1.7: Two versions of our package are now in the package cache.
Sharing code and source control

Figure 1.8: The versions browser showing two versions of the ObjTest>>testIVOffset method.

your code with others and to make sure that you can use automated build systems such as Jenkins or Travis.

If you do try and save to a repository where you don’t have write permission, a version will nevertheless be written to the package-cache. So you can recover by editing the repository information (right-click in the Monticello Browser) or choosing a different repository, and then using Copy from the package-cache browser.

1.2 Source control

Versions of a method

When you save a new version of a method, the old one is not lost. Pharo keeps all of the old versions (assuming that you are using the same image and that you saved it), and allows you to compare different versions and to go back (revert) to an old version.

The browse > versions (v) menu item gives access to the successive modifications made to the selected method. In Figure 1.8 we can see two versions of a method.
The top pane displays one line for each version of the method, listing the name of the programmer who wrote it, the date and time at which it was saved, the names of the class and the method, and the protocol in which it was defined. The current (active) version is at the top of the list.

The existence of the versions browser means that you never have to worry about preserving code that you think might no longer be needed. If you find that you do need it, you can always revert to the old version, or copy the needed code fragment out of the old version and paste it into a another method. Get into the habit of using versions; commenting out code that is no longer needed is a bad practice because it makes the current code harder to read. Pharoers rate code readability extremely highly.

**Hint:** What if you delete a method entirely, and then decide that you want it back? You can find the deletion in a change set, where you can ask to see versions by right-clicking. The change set browser is described in section 1.2.

### Change sets and the changesorter

Whenever you are working in Pharo, any changes that you make to methods and classes are recorded in a *change set*. This includes creating new classes, re-naming classes, changing categories, adding methods to existing classes – just about everything of significance. However, arbitrary Do its are not included.

At any time, many change sets exist, but only one of them – ChangeSet current – is collecting the changes that are being made to the image. You can see which change set is current and can examine all of the change sets using the change sorter, available by selecting **World > Tools... > Change Sorter**.

Figure 1.9 shows the dual change sorter browser. The title bar shows which change set is current, and this change set is selected when the change sorter opens.

Other change sets can be selected in the top-left pane; the right-click menu allows you to make a different change set current, or to create a new change set. The next pane lists all of the classes affected by the selected change set (with their categories). Selecting one of the classes displays the names of its methods that are also in the change set (not all of the methods in the class) in the left central pane, and selecting a method name displays the method definition in the bottom pane.

The change sorter also lets you delete classes and methods from the change set using the right-click menu on the corresponding items.

The change sorter allows you to simultaneously view two change sets, one on the left hand side and the other on the right. This layout supports the change sorter’s main feature, which is the ability to move or copy changes from one change set to another, as shown by the right-click menu in Figure 1.9. It is also possible to copy individual methods from one side to the other.
Sharing code and source control

You may be wondering why you should care about the composition of a change set. The answer is that change sets provide a simple mechanism for exporting code from Pharo to the file system, from where it can be imported into another Pharo image. Change set export is known as Filing-out, and can be accomplished using the right-click menu on any change set, class or method in either browser. Repeated file outs create new versions of the file, but change sets are not a versioning tool like Monticello: they do not keep track of dependencies.

Before the advent of Monticello, change sets were the main means for exchanging code between Pharoers. They have the advantage of simplicity (the file out is just a text file, although we don’t recommend that you try to edit them with a text editor), and a degree of portability.

The main drawback of change sets, compared to Monticello packages, is that they do not support the notion of dependencies. In spite of these shortcomings, change sets still have their uses. In particular, you may find change sets on the Internet that you want to look at and perhaps use. So, having filed out a change set using the change sorter, we will now tell you how to file one in. This requires the use of another tool, the file list browser. Note that dropping a changeset on Pharo itself works brings a pop up to ask you whether you want to load or browse the code contained in a changeset.
1.3 The File List Browser

The file list browser is in fact a general-purpose tool for browsing the file system (as well as FTP servers) from Pharo. You can open it from the World > Tools... > File Browser menu. What you see of course depends on the contents of your local file system, but a typical view is shown in Figure 1.10.

When you first open a file list browser it will be focused on the current directory, that is, the one from which you started Pharo. The title bar shows the directory name. The larger pane on the left-hand side can be used to navigate the file system in the conventional way. When a directory is selected, the files that it contains (but not the directories) are displayed on the right. This list of files can be filtered by entering a Unix-style pattern in the small box at the top-left of the window. Initially, this pattern is *, which matches all file names, but you can type a different string there and accept it, changing the pattern. (Note that a * is implicitly prepended and appended to the pattern that you type.) The sort order of the files can be changed using the name, date and size buttons. The rest of the buttons depend on the name of the file selected in the browser. In Figure 1.10, the file name has the suffix .st, so the browser assumes that it is pharo code, and provides the possible actions as buttons.

Because the choice of buttons to display depends on the file’s name, and not on its contents, sometimes the button that you want won’t be on the screen.
However, the full set of options is always available from the right-click menu, so you can easily work around this problem.

The code button is perhaps the most useful for working with change sets; it opens a browser on the contents of the change set file or even an MC package; an example is shown in Figure 1.11. The file contents browser is similar to the browser except that the code is not loaded in memory.

1.4 **In Pharo, you can’t lose code**

It is quite possible to crash Pharo: as an experimental system, Pharo lets you change anything, including things that are vital to make Pharo work!

The good news is that you will never lose any work, even if you crash and go back to the last saved version of your image, which might be hours old. This is because all of the code that you executed is saved in the `.changes` file. This excludes Playground snippets, those are stored inside `/pharo-local/playcache` and they can be searched using Spotter (the results are displayed under Playground-cached pages section).

So here are the instructions on how to get your code back. There is no need to read this until you need it. However, when you do need it, you’ll find it here waiting for you.

In the worst case, you can use a text editor on the `.changes` file, but since it is many megabytes in size, this can be slow and is not recommended. Pharo
1.5 Chapter summary

offers you better ways.

How to get your code back

Restart Pharo from the most recent snapshot, and select World > Tools... >
Recover lost changes.

Smalltalk recover: 10000.
ChangeList browseRecentLog.
ChangeList browseRecent: 2000.

This will give you the opportunity to decide how far back in history you
wish to browse. Normally, it’s sufficient to browse changes as far back as
the last snapshot. (You can get much the same effect by editing ChangeList
browseRecent: 2000 so that the number 2000 becomes something else, using
trial and error.)

Once you have a recent changes browser, showing, say, changes back as far as
your last snapshot, you will have a list of everything that you have done to
Pharo during that time. You can delete items from this list using the right-
click menu. When you are satisfied, you can file-in what is left, thus incorpo-
rating the changes into your new image. It’s a good idea to start a new change
set, using the ordinary change set browser, before you do the file in, so that all
of your recovered code will be in a new change set. You can then file out this
change set.

One useful thing to do in the recent changes browser is to remove doIts. Usu-
ally, you won’t want to file in (and thus re-execute) doIts. However, there is an
exception. Creating a class shows up as a doIt.

Before you can file in the methods for a class, the class must exist. So, if you have
created any new classes, first file-in the class creation doIts, then remove
doIts and file in the methods.

When I am finished with the recovery, I like to file out my new change set, quit
Pharo without saving the image, restart, and make sure that the new change
set files back in cleanly.

1.5 Chapter summary

- Monticello is a tool for exporting, importing, versioning and sharing
  packages of classes and methods.
- A Monticello package consists of classes, and related methods in other
  packages.
- Change sets are automatically generated logs of all changes to the source
code of your image. They have largely been superseded by Monticello
  as a means to store and exchange versions of your source code, but are
still useful, especially for recovering from catastrophic failures, however rare these may be.

- The file list browser is a tool for browsing the file system. It also allows you to file in source code from the file system.

- In case your image crashes before you could save it or backup your source code with Monticello, you can always recover your most recent changes using a change list browser. You can then select the changes you want to replay and file them into the most recent copy of your image.