Chapter 1

Versioning your code with Monticello

with the participation of:
Oscar Nierstrasz *(oscar.nierstrasz@acm.org)*

A versioning system helps you to store and log multiple versions of your code. In addition it may help you to manage concurrent accesses to a common source code repository. It keeps track of all changes to a set of documents and enables several developers to collaborate. As soon as the size of your software increases beyond a few classes, you probably need a versioning system.

Many different versioning systems are available. CVS[1] and Subversion[2] are probably the most popular. In principle you could use them to manage the development of Pharo software projects, but such a practice would disconnect the versioning system from the Pharo environment. In addition, CVS-like tools only version plain text files and not individual packages, classes or methods. We would therefore lack the ability to track changes at the appropriate level of granularity. If the versioning tools know that you store classes and methods instead of plain text, they can do a better job of supporting the development process.

Monticello is a versioning system for Pharo in which classes and methods, rather than lines of text, are the units of change. *SqueakSource* is a central online repository in which you can store versions of your applications using Monticello. SqueakSource is the equivalent of GForge, and Monticello the equivalent of CVS.

[1] http://www.nongnu.org/cvs
[2] http://subversion.tigris.org

In this chapter, you will learn how to use use Monticello and SqueakSource to manage your software. We have already met Monticello briefly in earlier chapters[3]. This chapter delves into the details of Monticello and describes some additional features that are useful for versioning large applications.

1.1 Basic usage

We will start by reviewing the basics of creating a package and committing changes, and then we will see how to update and merge changes.

Running example — perfect numbers

We will use a small running example of perfect numbers[4] in this chapter to illustrate the features of Monticello. We will start our project by defining some simple tests.

Define a subclass of TestCase *called* PerfectTest *in the category* Perfect*, and define the following test methods in the protocol* running*:*

```
PerfectTest»testPerfect
   self assert: 6 isPerfect.
   self assert: 7 isPerfect not.
   self assert: 28 isPerfect.
```

Of course these tests will fail as we have not yet implemented the isPerfect method for integers. We would like to put this code under the control of Monticello as we revise and extend it.

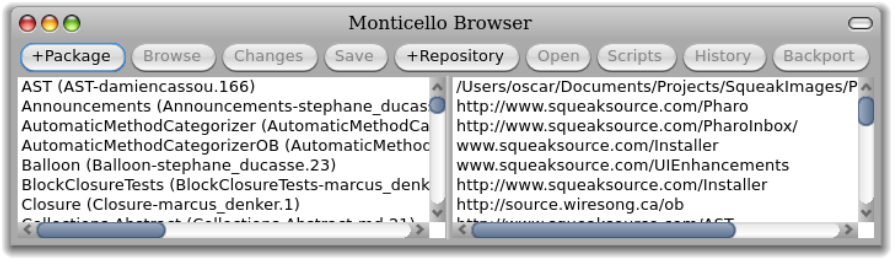
Launching Monticello

Monticello is included in the standard Pharo distribution. We will assume that Monticello is already installed in your image. Monticello Browser can be selected from the *World* menu.

In Figure **??** we see that the Monticello Browser consists of two list panes and one button pane. The left pane lists installed packages and the right panes shows known repositories. Various operations may be performed via the button pane and the menus of the two list panes.

[3]"A first application" and "The Pharo programming environment"

[4]Perfect numbers were discovered by Euclid. A perfect number is a positive integer that is the sum of its proper divisors. $6 = 1 + 2 + 3$ is the first perfect number.



AST (AST-damiencassou.166)
Announcements (Announcements-stephane_ducas
AutomaticMethodCategorizer (AutomaticMethodCa
AutomaticMethodCategorizerOB (AutomaticMethoc
Balloon (Balloon-stephane_ducasse.23)
BlockClosureTests (BlockClosureTests-marcus_denk
Closure (Closure-marcus_denker.1)

/Users/oscar/Documents/Projects/SqueakImages/P
http://www.squeaksource.com/Pharo
http://www.squeaksource.com/PharoInbox/
www.squeaksource.com/Installer
www.squeaksource.com/UIEnhancements
http://www.squeaksource.com/Installer
http://source.wiresong.ca/ob

Figure 1.1: The Monticello Browser.

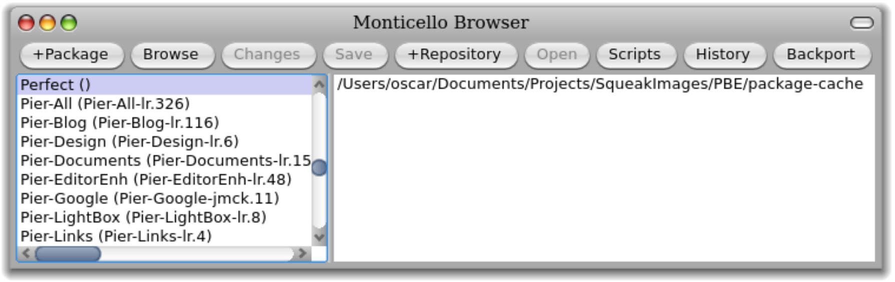
Creating a package

Monticello manages versions of *packages*. A package is essentially a named set of classes and methods. In fact, a package is an object — an instance of PackageInfo — that knows how to identify the classes and methods that belong to it.

We would like to version our PerfectTest class. The right way to do this is to define a package — called Perfect — containing PerfectTest and all the related classes and methods we will introduce later. For the moment, no such package exists. We only have a *category* called (not coincidentally) Perfect. This is perfect, since Monticello will map categories to packages for us.

Press the +Package *in the Monticello browser and enter* Perfect.

Voilà! You have just created the *Perfect* Monticello package.



Perfect ()
Pier-All (Pier-All-lr.326)
Pier-Blog (Pier-Blog-lr.116)
Pier-Design (Pier-Design-lr.6)
Pier-Documents (Pier-Documents-lr.15
Pier-EditorEnh (Pier-EditorEnh-lr.48)
Pier-Google (Pier-Google-jmck.11)
Pier-LightBox (Pier-LightBox-lr.8)
Pier-Links (Pier-Links-lr.4)

/Users/oscar/Documents/Projects/SqueakImages/PBE/package-cache

Figure 1.2: Creating the Perfect package.

Monticello packages follow a number of important naming conventions for class and method categories. Our new package named *Perfect* contains:

- All classes in the category *Perfect*, or in categories whose names start with *Perfect-*. For now this includes only our PerfectTest class.

- All methods belonging to *any* class (in any category) that are defined in a protocol named **perfect* or **Perfect*, or in protocols whose names start with **perfect-* or **Perfect-*. Such methods are known as *extensions*. We don't have any yet, but we will define some very soon.

- All methods belonging to any classes in the category *Perfect*, or in categories whose names begin with *Perfect-*, *except* those in protocols whose names start with * (*i.e.*, those belonging to *other* packages). This includes our testPerfect method, since it belongs to the protocol running.

Committing changes

Note in Figure **??** that the $\boxed{\text{Save}}$ button is disabled (greyed out).

Before we save our Perfect package, we need to specify *where* we want to save it. A *repository* is a package container, which may either be local to your machine or remote (accessed over the network). Various protocols may be used to establish a connection between your Pharo image and a repository. As we will see later (Section **??**), Monticello supports a large choice of repositories, though the most commonly used is HTTP, since this is the one used by SqueakSource.

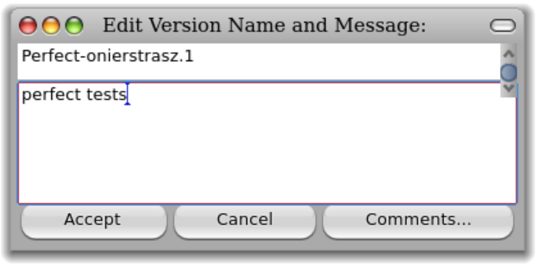
At least one repository, called package−cache, is set up by default, and is shown as the first entry in the list of repositories on the right-hand side of your Monticello browser (see Figure **??**). The package-cache is created automatically in the local directory where your Pharo image is located. It will contain a copy of all the packages you download from remote repositories. By default, copies of your packages are also saved in the package-cache when you save them to a remote server.

Each package knows which repositories it can be saved to. To add a new repository to the selected package, press the $\boxed{\text{+Repository}}$ button. This will offer a number of choices of different kinds of repository, including HTTP. For the rest of the chapter we will work with the package−cache repository, as this is all we need to explore the features of Monticello.

Select the directory repository named package cache, *press* $\boxed{\text{Save}}$, *enter an appropriate log message, and* $\boxed{\text{Accept}}$ *to save the changes.*

The Perfect package is now saved in package−cache, which is nothing more than a directory contained in the same directory as your Pharo image. Note, however, that if you use any other kind or repository (*e.g.*, HTTP, FTP, another local directory), a copy of your package will also be saved in the package-cache.

*Use your favorite file browser (*e.g.*, Windows Explorer, Finder or XTerm) to*



Perfect-onierstrasz.1

perfect tests

Figure 1.3: You may set a new version name and a commit message when you save a version of a package.

confirm that a file Perfect−XX.1.mcz *was created in your package cache.* XX *corresponds to your name or initials.*[5]

A *version* is an immutable snapshot of a package that has been written to a repository. Each version has a unique version number to identify it in a repository. Be aware, however, that this number is *not* globally unique — in another repository you might have the same file identifier for a *different snapshot*. For example, Perfect−onierstrasz.1.mcz in another repository might be the *final*, deployed version of our project! When saving a version into a repository, the next available number is automatically assigned to the version, but you can change this number if you wish. Note that version branches do not interfere with the numbering scheme (as with CVS or Subversion). As we shall see later, versions are by default ordered by their version number when viewing a repository.

Class extensions

Let's implement the methods that will make our tests green.

Define the following two methods in the class Integer, *and put each method in a protocol called* *perfect. *Also add the new boundary tests. Check that the tests are now green.*

```
Integer»isPerfect
   ↑ self > 1 and: [self divisors sum = self]

Integer»divisors
   ↑ (1 to: self − 1 ) select: [ :each | (self rem: each) = 0 ]

PerfectTest»testPerfectBoundary
```

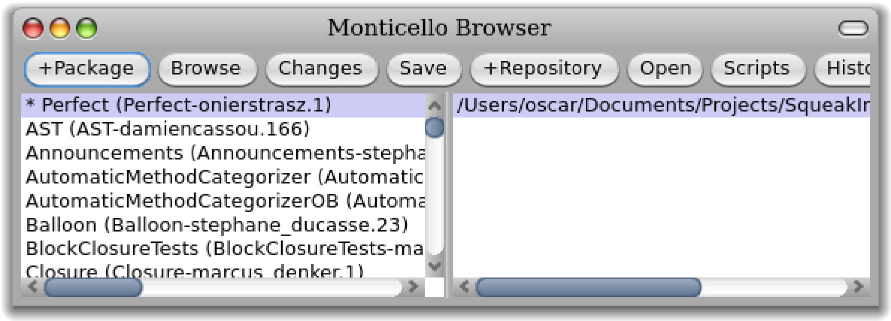
[5]In the past, the convention was for developers to log their changes using only their initials. Now, with many developers sharing identical initials, the convention is to use an identifier based on the full name, such as "apblack" or "AndrewBlack".

```
self assert: 0 isPerfect not.
self assert: 1 isPerfect not.
```

Although the methods on Integer do not belong to the *Perfect* category, they *do* belong to the Perfect package since they are in a protocol whose name starts with * and matches the package name. Such methods are known as *class extensions*, since they extend existing classes. These methods will be available *only* to someone who loads the Perfect package.

"Clean" and "Dirty" packages

Modifying the code in a package with any of the development tools makes that package *dirty*. This means that the version of the package in the image is different from the version that has been saved or loaded.



```
* Perfect (Perfect-onierstrasz.1)
AST (AST-damiencassou.166)
Announcements (Announcements-stepha
AutomaticMethodCategorizer (Automatic
AutomaticMethodCategorizerOB (Automa
Balloon (Balloon-stephane_ducasse.23)
BlockClosureTests (BlockClosureTests-ma
Closure (Closure-marcus_denker.1)
```

```
/Users/oscar/Documents/Projects/SqueakIr
```

Figure 1.4: Modifying our Perfect package will "dirty" it.

In the Monticello browser, a dirty package can be recognized by an asterix (*) preceding its name. This indicates which packages have uncommitted changes, and therefore need to be saved into a repository if those changes are not to be lost. Saving a dirty package cleans it.

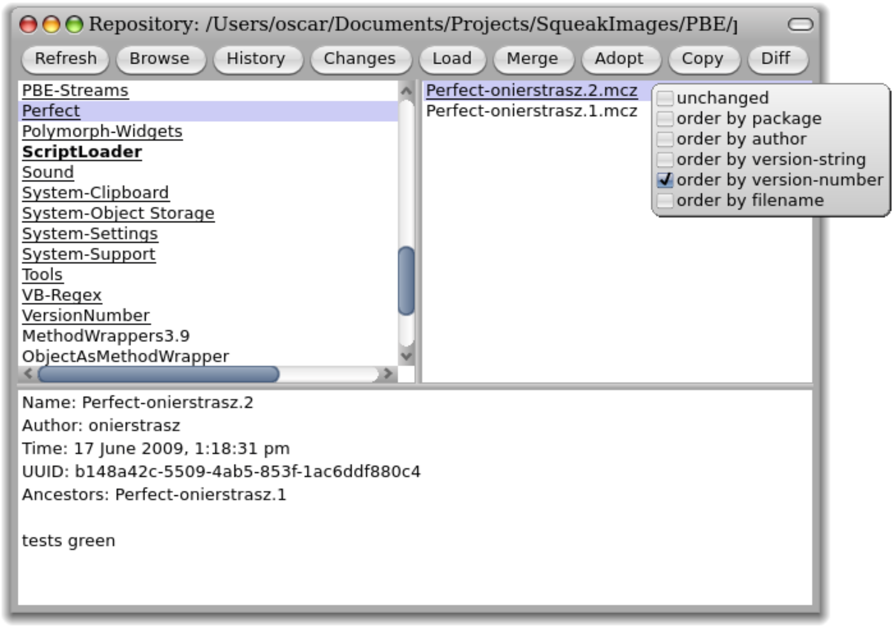
Try the Browse, History *and* Changes *buttons to see what they do*[6]. Save *the changes to the* Perfect *package. Confirm that the package is now "clean" again.*

The Repository inspector

The contents of a repository can be explored using a repository inspector, which is launched using the Open button of Monticello (cf Figure **??**).

Select the package−cache *repository and open it. You should see something like Figure* **??**.

[6]At the time of this writing, the Scripts button does not work.



[✓ order by version-number]

```
Name: Perfect-onierstrasz.2
Author: onierstrasz
Time: 17 June 2009, 1:18:31 pm
UUID: b148a42c-5509-4ab5-853f-1ac6ddf880c4
Ancestors: Perfect-onierstrasz.1

tests green
```

Figure 1.5: A repository inspector.

All the packages in the repository are listed on the left-hand side of the inspector:

- an underlined package name means that this package is installed in the image;
- a **bold underlined** name means that the package is installed, but that there is a more recent version in the repository;
- a name in a normal typeface means that the package is not installed in the image.

Once a package is selected, the right-hand pane lists the versions of the selected package:

- an underlined version name means that this version is installed in the image;
- a **bold** version name means that this version is not an ancestor of the installed version. This may mean that it is a newer version, or that it belongs to a different branch from the installed version;
- a version name displayed with a normal typeface shows an older version than the installed current one.

Action-clicking the right-hand side of the inspector opens a menu with different sorting options. The unchanged entry in the menu discards any particular sorting. It uses the order given by the repository.

Loading, unloading and updating packages

At present we have two versions of the Perfect package stored safely in our package−cache repository. We will now see how to unload this package, load an earlier version, and finally update it.

Select the Perfect *package and its repository in the Monticello browser. Action-click on the package name and select* unload package.

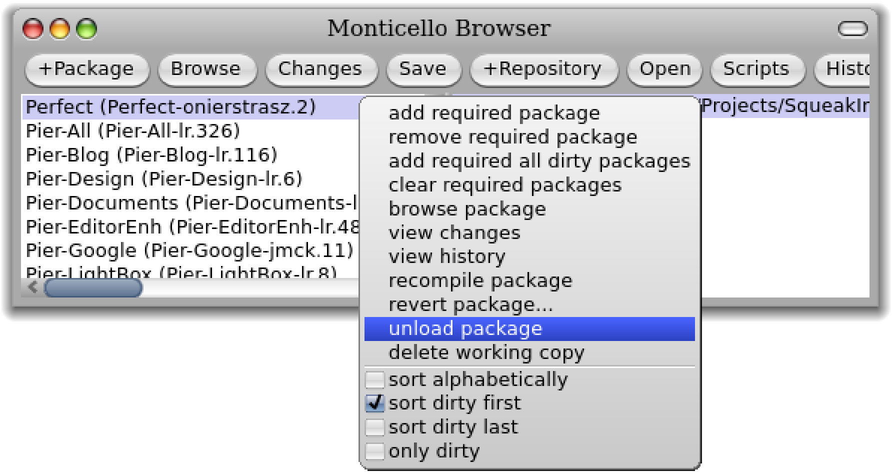


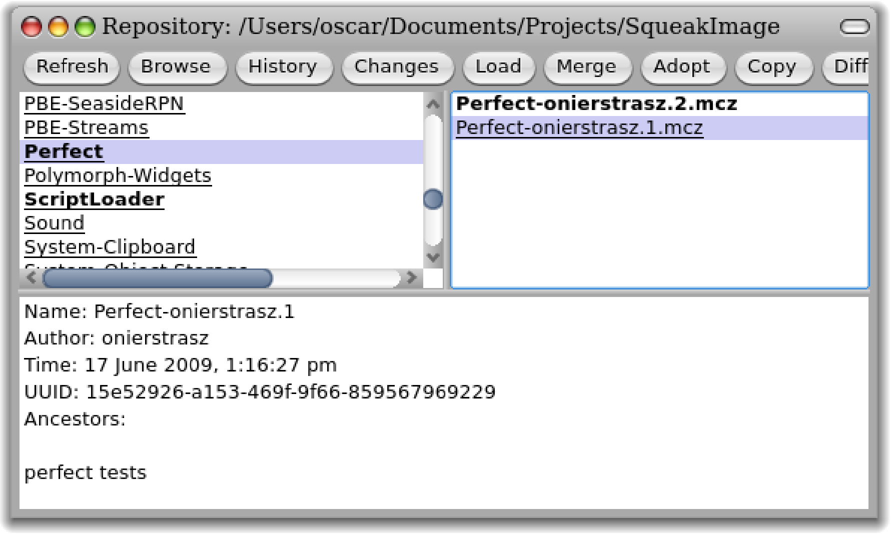
Figure 1.6: Unloading a package.

You should now be able to confirm that the Perfect package has vanished from your image!

In the Monticello browser, select the package−cache *in the repository pane, without selecting anything in the package pane, and* Open *the repository inspector. Scroll down and select the* Perfect *package. It should be displayed in a normal typeface, indicated that it is not installed. Now select version 1 of the package and* Load *it.*

You should now be able to verify that the only the original (red) tests are loaded.

Select the second version of the Perfect *package in the repository inspector and* Load *it. You have now* updated *the package to the latest version.*



```
Name: Perfect-onierstrasz.1
Author: onierstrasz
Time: 17 June 2009, 1:16:27 pm
UUID: 15e52926-a153-469f-9f66-859567969229
Ancestors:

perfect tests
```

Figure 1.7: Loading an earlier version.

Now the tests should be green again.

Branching

A *branch* is a line of development versions that exists independently of another line, yet still shares a common ancestor version if you look far enough back in time.

You may create new version branch when saving your package. Branching is useful when you want to have a new parallel development. For example, suppose your job is to maintain a software in your company. One day a different division asks you for the same software, but with a few parts tweaked for them, since they do things slightly differently. The way to deal with this situation is to create a second branch of your program that incorporate the tweaks, while leaving the first branch unmodified.

From the repository inspector, select version 1 of the Perfect *package and* Load *it. Version 2 should again be displayed in bold, indicating that it no longer loaded (since it is not an ancestor of version 1). Now implement the following two* Integer *methods and place them in the* *perfect *protocol, and also modify the existing* PerfectTest *test method as follows:*

```
Integer»isPerfect
   self < 2 ifTrue: [ ↑ false ].
   ↑ self divisors sum = self
```

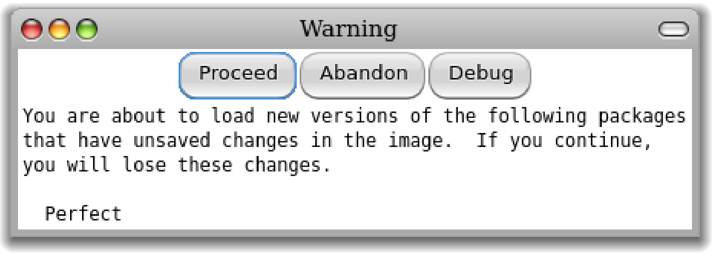
```
Integer»divisors
  ↑ (1 to: self − 1 ) select: [ :each | (self \\ each) = 0]

PerfectTest»testPerfect
  self assert: 2 isPerfect not.
  self assert: 6 isPerfect.
  self assert: 7 isPerfect not.
  self assert: 28 isPerfect.
```

Once again the tests should be green, though our implementation of perfect numbers is slightly different.

Attempt to load version 2 of the Perfect *package.*

Now you should get a warning that you have unsaved changes.



You are about to load new versions of the following packages that have unsaved changes in the image. If you continue, you will lose these changes.

Perfect

Figure 1.8: Unsaved changes warning.

Select Abandon *to avoid overwriting your new methods. Now* Save *your changes. You will get another warning that there may be newer versions. Select* Yes*, enter your log message, and* Accept *the new version.*

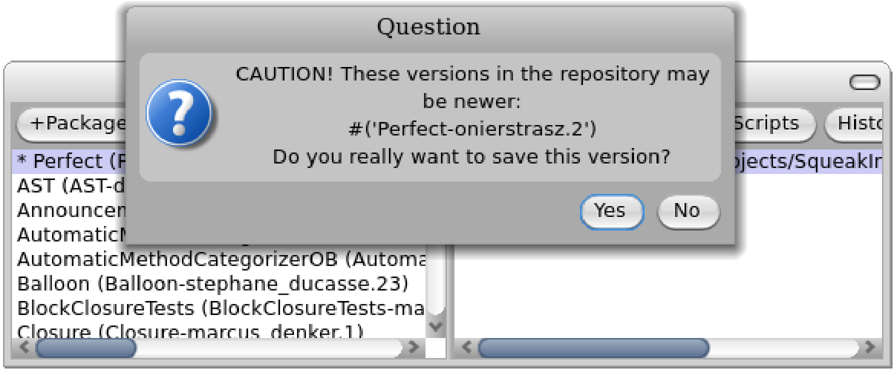
Congratulations! You have now created a new branch of the Perfect package.

If you still have the repository inspector open, Refresh *it to see the new version (Figure* **??***).*

Merging

This section describes the conventional merging facility of Monticello. To use it, make sure that the preference useNewDiffToolsForMC is disabled. You can either use the Preference Browser, or you can do this programmatically by evaluating the following expression:

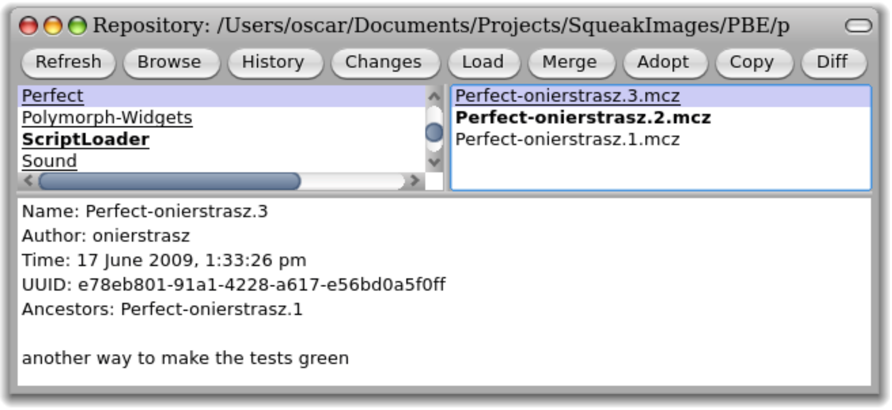
```
Preferences disable: #useNewDiffToolsForMC
```


Question

CAUTION! These versions in the repository may be newer:
#('Perfect-onierstrasz.2')
Do you really want to save this version?

Yes
No
+Package
Scripts
Histo
* Perfect (P
AST (AST-d
Announcen
Automatic
AutomaticMethodCategorizerOB (Automa
Balloon (Balloon-stephane_ducasse.23)
BlockClosureTests (BlockClosureTests-ma
Closure (Closure-marcus_denker.1)
ojects/SqueakIr

Figure 1.9: Newer versions warning.


Repository: /Users/oscar/Documents/Projects/SqueakImages/PBE/p

Name: Perfect-onierstrasz.3
Author: onierstrasz
Time: 17 June 2009, 1:33:26 pm
UUID: e78eb801-91a1-4228-a617-e56bd0a5f0ff
Ancestors: Perfect-onierstrasz.1

another way to make the tests green

Figure 1.10: Versions 2 and 3 are separate branches of version 1.

You can merge one version of a package with another using the Merge button in the Monticello browser. Typically you will want to do this when (i) you discover that you have been working on a out-of-date version, or (ii) branches that were previously independent have to be re-integrated. Both scenarios are common when multiple developers are working on the same package.

Consider the current situation with our Perfect package, as illustrated at the left of Figure **??**. We have published a new version 3 that is based on version 1. Since version 2 is also based on version 1, versions 2 and 3 constitute independent branches.

At this point we realize that there are changes in version 2 that we would like to merge with our changes from version 3. Since we have version 3

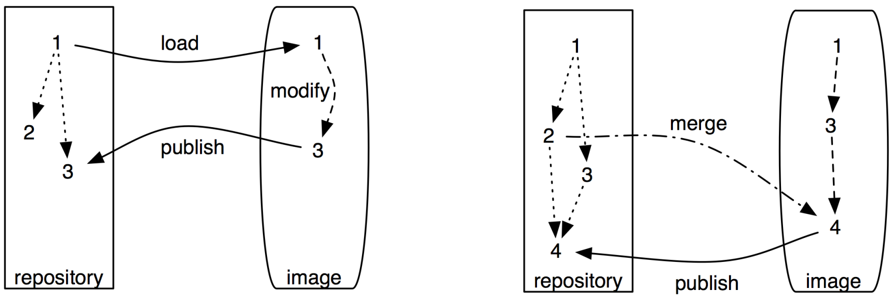
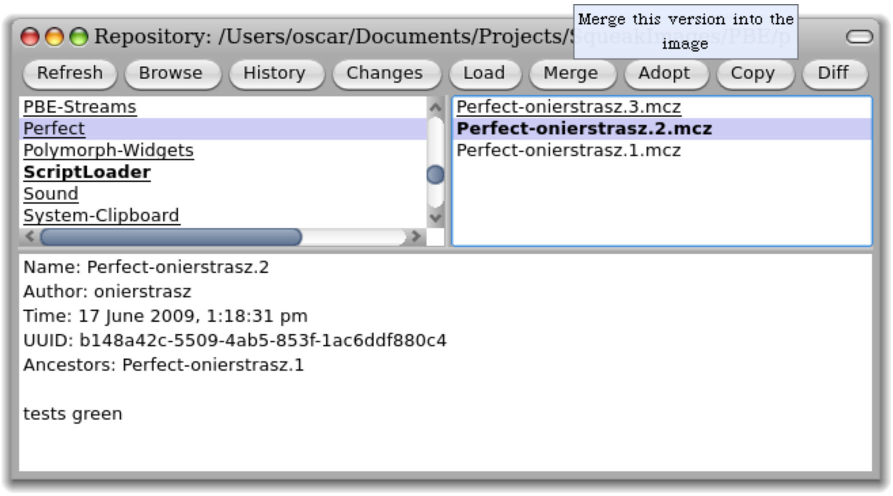


Figure 1.11: Branching (left) and merging (right).

currently loaded, we would like to merge in changes from version 2, and publish a new, merged version 4, as illustrated at the right of Figure ??.


Repository: /Users/oscar/Documents/Projects/S
Merge this version into the image
Refresh
Browse
History
Changes
Load
Merge
Adopt
Copy
Diff
PBE-Streams
Perfect
Polymorph-Widgets
ScriptLoader
Sound
System-Clipboard
Perfect-onierstrasz.3.mcz
Perfect-onierstrasz.2.mcz
Perfect-onierstrasz.1.mcz

Name: Perfect-onierstrasz.2
Author: onierstrasz
Time: 17 June 2009, 1:18:31 pm
UUID: b148a42c-5509-4ab5-853f-1ac6ddf880c4
Ancestors: Perfect-onierstrasz.1

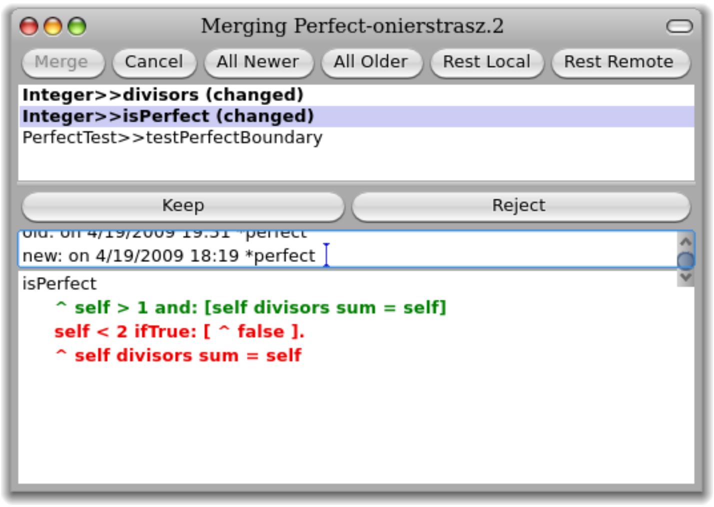
tests green

Figure 1.12: Select a separate branch (in bold) to be merged.

Select version 2 in the repository browser, as shown in Figure **??**, *and click the* Merge *button.*

The merge tool is a tool that allows for fine-grained package version merging. Elements contained in the package to-be-merged are listed in the upper text pane. The lower text pane shows the definition of a selected element.

In Figure **??** we see the three differences between versions 2 and 3 of the Perfect package. The method PerfectTest»testPerfectBoundary is new, and the two indicated methods of Integer have been changed. In the lower pane we



Integer>>divisors (changed)
Integer>>isPerfect (changed)
PerfectTest>>testPerfectBoundary

```
old: on 4/19/2009 19:31 *perfect
new: on 4/19/2009 18:19 *perfect
isPerfect
    ^ self > 1 and: [self divisors sum = self]
    self < 2 ifTrue: [ ^ false ].
    ^ self divisors sum = self
```

Figure 1.13: Version 2 of the Perfect package being merged with the current version 3.

see the old and new versions of the source code of Integer»isPerfect. New code is displayed in red, removed code is barred and displayed in blue, and unchanged code is shown in black.

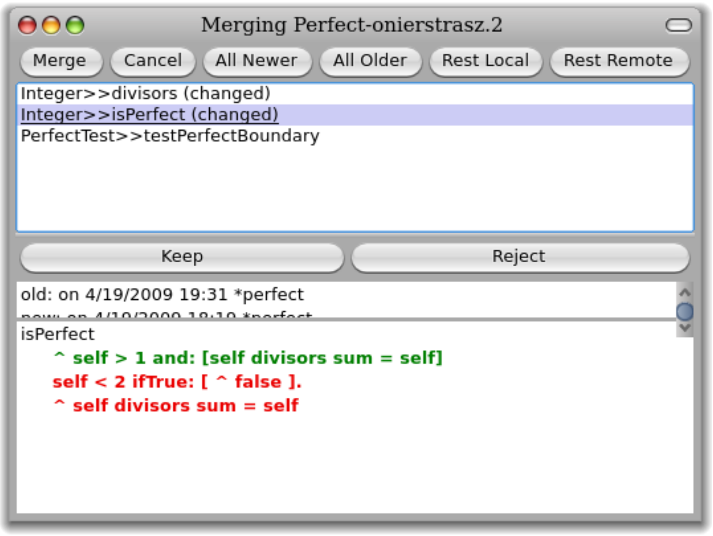
A method or a class is in conflict if its definition has been altered. Figure **??** shows 2 conflicting methods in the class Integer: isPerfect and divisors. A conflicting package element is indicated by being underlined, ~~barred~~, or **bold**. The full set of typeface conventions is as follows:

Plain=No Conflict. A plain typeface indicates the definition is non-conflicting. For example, the method PerfectTest»testPerfectBoundary does not conflict with an existing method, and can be installed.

Bold=A method is conflicting. A decision needs to be taken to keep the proposed change or reject it. The proposed method Integer»>isPerfect is in conflict with an existing definition in the image. The conflict can be resolved by clicking $\boxed{\text{Keep}}$ or $\boxed{\text{Reject}}$.

Underlined=Repository replace current. An underlined element will be kept and replace the current element in the image. In Figure **??** we see that Integer»isPerfect from version 2 has been kept.

Barred=Repository version rejected. A ~~barred~~ element has been rejected, and the local definition will not be replaced. In Figure **??** Integer»divisors from version 2 is rejected, so the definition from version 3 will remain.



```
Integer>>divisors (changed)
Integer>>isPerfect (changed)
PerfectTest>>testPerfectBoundary
```

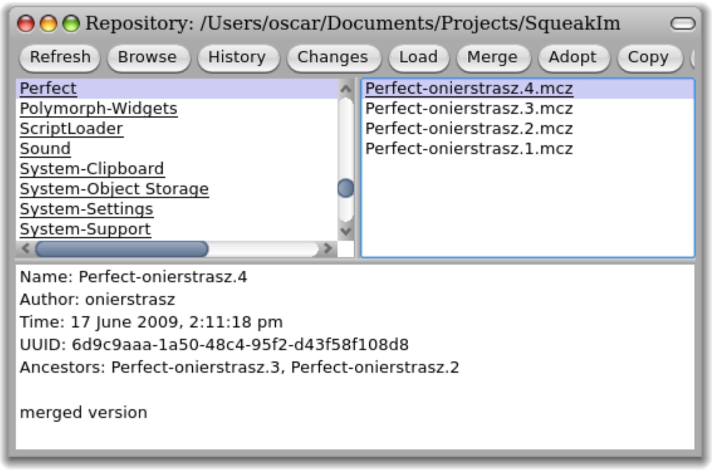
```
old: on 4/19/2009 19:31 *perfect
```

```
isPerfect
	^ self > 1 and: [self divisors sum = self]
	self < 2 ifTrue: [ ^ false ].
	^ self divisors sum = self
```

Figure 1.14: Keeping and rejecting changes.

Note that the merge tool offers buttons to select all newer or all older changes, or to select all local or all remote changes that are still in conflict.

Keep Integer»>isPerfect *and reject* Integer»divisors, *and click the* Merge *button. Confirm that the tests are all green. Commit the new merged version of* Perfect *as version 4.*



```
Perfect
Polymorph-Widgets
ScriptLoader
Sound
System-Clipboard
System-Object Storage
System-Settings
System-Support
```

```
Perfect-onierstrasz.4.mcz
Perfect-onierstrasz.3.mcz
Perfect-onierstrasz.2.mcz
Perfect-onierstrasz.1.mcz
```

```
Name: Perfect-onierstrasz.4
Author: onierstrasz
Time: 17 June 2009, 2:11:18 pm
UUID: 6d9c9aaa-1a50-48c4-95f2-d43f58f108d8
Ancestors: Perfect-onierstrasz.3, Perfect-onierstrasz.2

merged version
```

Figure 1.15: All older versions are now ancestors of merged version 4.

If you now refresh the repository inspector, you will see that there are no more versions shown in bold, *i.e.*, all versions are ancestors of the currently loaded version 4 (Figure **??**).

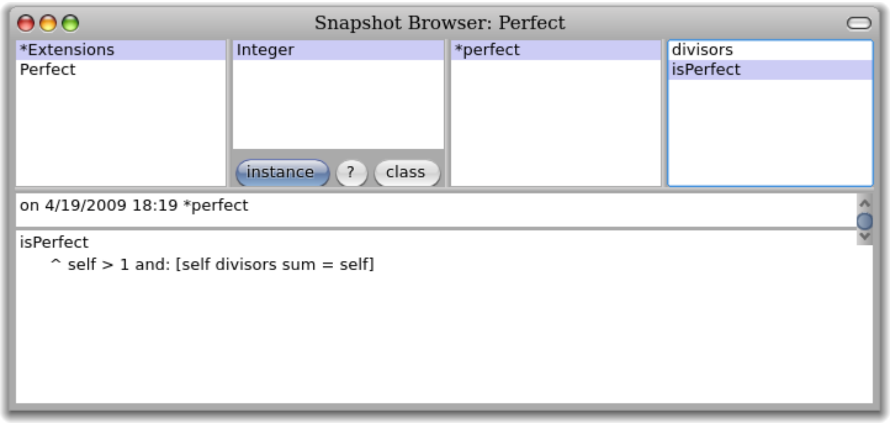
1.2 Exploring Monticello repositories

Monticello has many other useful features. As we can see in Figure **??**, the Monticello browser window has nine buttons. We have already used four of them — +Package, Save, +Repository and Open. We will now look at Browse, Changes and History, which are used to explore the state and history of repositories

Browse

The Browse button opens a "snapshot browser" to display the contents of a package. The advantage of the snapshot browser over the browser is its ability to display class extensions.

Select the Perfect *package and click the* Browse *button.*



```
on 4/19/2009 18:19 *perfect
```

```
isPerfect
    ^ self > 1 and: [self divisors sum = self]
```

Figure 1.16: The snapshot browser reveals that the Perfect package extends the class Integer with 2 methods.

For example, Figure **??** shows the class extensions defined in the *Perfect* package. Note that code cannot be edited here, though by action-clicking, if your environment has been set up accordingly) on a class or a method name you can open a regular browser.

It is a good practice to always browse the code of your package before publishing it, to ensure that it really contains what you think it does.

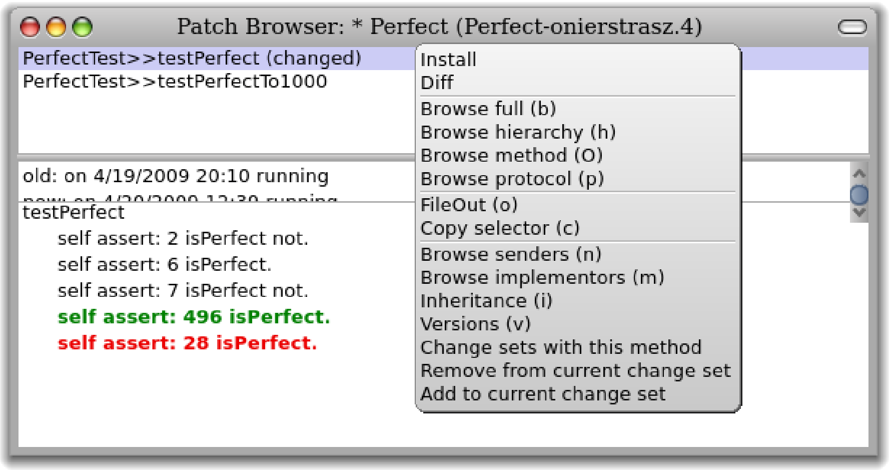
Changes

The Changes button computes the difference between the code in the image and the most recent version of the package in the repository.

Make the following changes to PerfectTest, *and then click the* Changes *button in the Monticello browser.*

```
PerfectTest»testPerfect
   self assert: 2 isPerfect not.
   self assert: 6 isPerfect.
   self assert: 7 isPerfect not.
   self assert: 496 isPerfect.

PerfectTest»testPerfectTo1000
   self assert: ((1 to: 1000) select: [:each | each isPerfect]) = #(6 28 496)
```



```
PerfectTest>>testPerfect (changed)
PerfectTest>>testPerfectTo1000

old: on 4/19/2009 20:10 running
testPerfect
    self assert: 2 isPerfect not.
    self assert: 6 isPerfect.
    self assert: 7 isPerfect not.
    self assert: 496 isPerfect.
    self assert: 28 isPerfect.
```

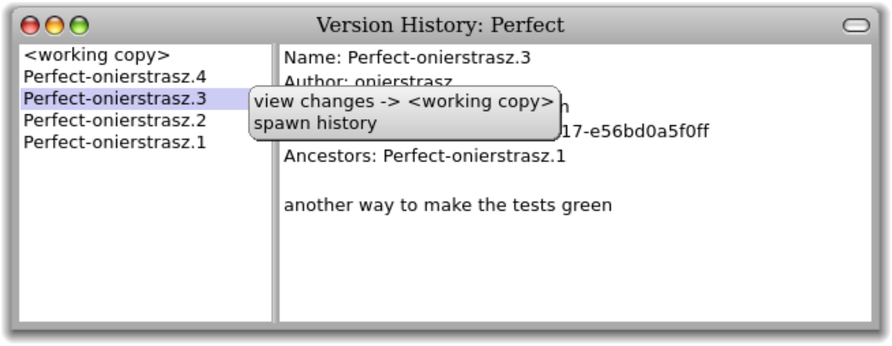
Figure 1.17: The patch browser shows the difference between the code in the image and the most recently committed version.

Figure ?? shows that the Perfect package has been locally modified with one changed method and one new method. As usual, action-clicking on a change offers you a choice of contextual operations.

History

The History button opens a version history viewer that displays the comments committed along with each version of the selected package (see Figure **??**). The versions of the package, in this case Perfect, are listed on the left, while information about the selected version is displayed on the right.

Select the Perfect *package and click the* History *button.*



```
<working copy>
Perfect-onierstrasz.4
Perfect-onierstrasz.3
Perfect-onierstrasz.2
Perfect-onierstrasz.1
```

```
Name: Perfect-onierstrasz.3
Author: onierstrasz
17-e56bd0a5f0ff
Ancestors: Perfect-onierstrasz.1

another way to make the tests green
```

Figure 1.18: The version history viewer provides information about the various versions of a package.

By action-clicking on a particular version, you can explore the changes with respect to the current working copy of the package loaded in the image, or spawn a new history browser relative to the selected version.

1.3 Advanced topics

Now we will have a look at several advanced topics, including backporting, managing dependencies, and class initialization.

Backporting

Sometimes we want to port changes from one branch to another, without actually being forced to merge those branches. Backporting is a process of applying selected changes from one version of a package to an ancestor so that these changes can be merged into later branches. This is especially useful when corrections to software defects must be merged into multiple branches.

The process is illustrated in Figure **??**. Suppose that the main branch of our software system consists of versions A and B, maintained by Manny. A contributor, Conny, has developed a separate experimental branch, C, with changes X and Y. Change X fixes a nasty problem in versions A and B, so Manny asks Conny to prepare a backported branch D containing *only* change X. Now Manny can merge B and D to produce a new version E that fixes the defect. Conny can continue to further develop her independent branch C.

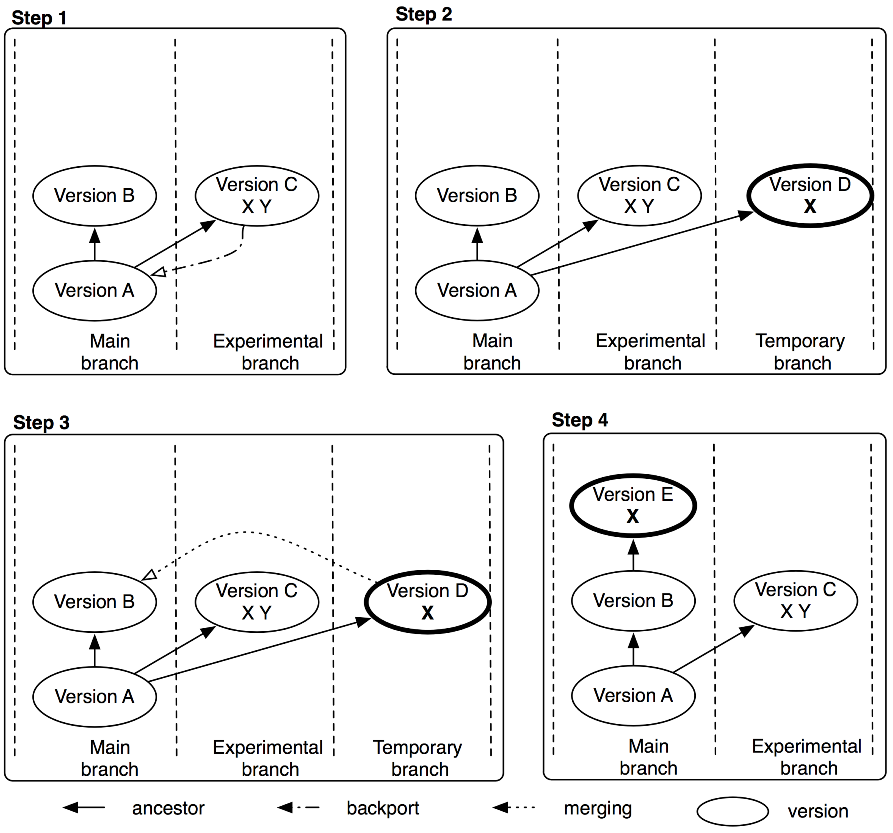


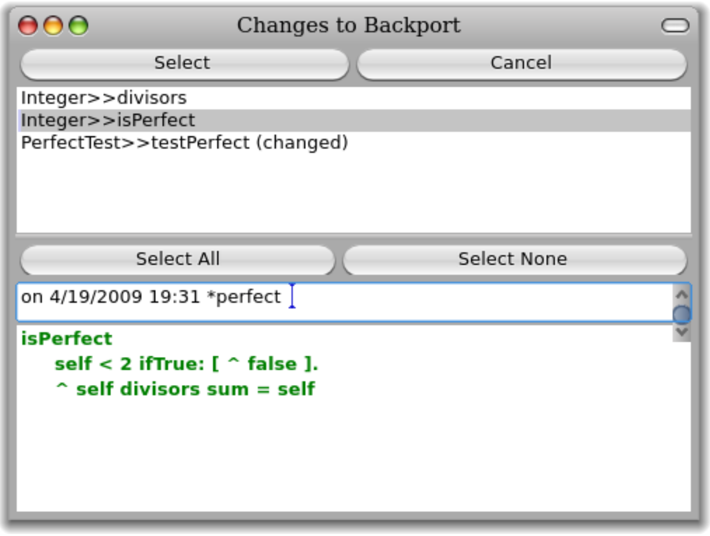
Figure 1.19: Change X is backported from version C to version A, producing a new branch D. D can then be merged into B, without affecting C.

The system records the fact that this new version was backported from a later version, and will make use of that information when merging.

To use Backport, you must have just saved your package — if your package is marked with the modified *, Backport is disabled. When you press Backport, you will first be asked to pick the ancestor version you want to backport to. You will then be presented with a multi-select list of all the changes between that ancestor and the current version. Choose only the changes you want to backport, and then press Select.

Let us see how this works in practice. Recall that we earlier rejected the implementation of isPerfect when we merged versions 2 and 3 of the Perfect package. Now we will recover that change as a backport to version 1. (Versions 1, 2 and 3 play the roles of versions A, B and C, respectively, in Figure **??**.)

Unload the Perfect *package. Now open a repository inspector on your* package−cache *and load* Perfect *version 3. In the Monticello browser select* Perfect *and click on the* Backport *button. Select version 1 as the ancestor. You should be able to browse the changes between version 3 and 1, as shown in Figure* **??**. *Now select the* Integer»isPerfect *method and click on* Select.



```
Integer>>divisors
Integer>>isPerfect
PerfectTest>>testPerfect (changed)
```

```
on 4/19/2009 19:31 *perfect
```

```
isPerfect
    self < 2 ifTrue: [ ^ false ].
    ^ self divisors sum = self
```

Figure 1.20: Backporting changes from version 3 to version 1 of the Perfect package.

Congratulations! You have now backported the isPerfect method from version 3 to version 1 of Perfect. Any changes you didn't select were reverted; that is, your image will now contain only the code from the ancestor version 1, plus the changes that you chose. In the Monticello browser you should see that the currently loaded version of Perfect is now version 1 (not version 3). If you click on Changes, you will see that the only change is the isPerfect method. You can now save this backported version, merge it into something else, or whatever you like.

Dependencies

Most applications cannot live on their own and typically require the presence of other packages in order to work properly. For example, let us have a look at Pier[7], a meta-described content management system. Pier is a large piece of software with many facets (tools, documentations, blog, catch strategies, security, ...). Each facet is implemented by a separate package. Most Pier packages cannot be used in isolation since they refer to methods and classes defined in other packages. Monticello provides a dependency mechanism for declaring the *required packages* of a given package to ensure that it will be correctly loaded.

Essentially, the dependency mechanism ensures that all required packages of a package are loaded before the package is loaded itself. Since required packages may themselves require other packages, the process is applied recursively to a tree of dependencies, ensuring that the leaves of the tree are loaded before any branches that depend on them. Whenever new versions of required packages are checked in, then new versions of the packages that depend on them will automatically depend on the new versions.

> *Dependencies cannot be expressed across repositories.* All requiring and required packages must live in the same repository.

Figure **??** illustrates how this works in Pier. Package Pier−All is an *empty package* that acts as a kind of umbrella. It requires Pier−Blog, Pier−Caching and all the other Pier packages.

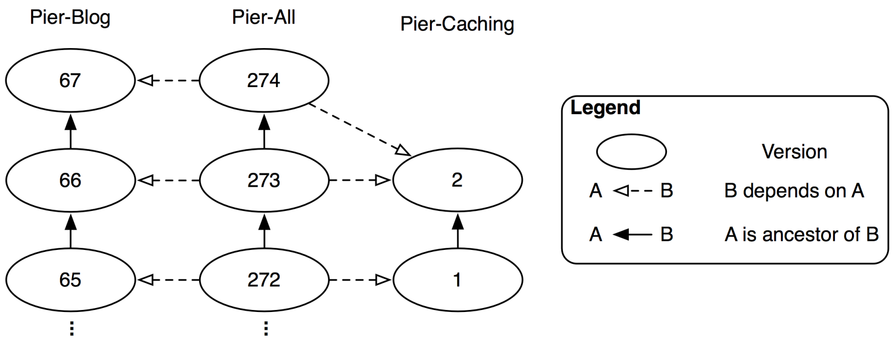


Figure 1.21: Dependencies in Pier.

Because of these dependencies, installing Pier−All causes all the other Pier packages to be installed. Furthermore, when developing, the only package

[7] http://source.lukas-renggli.ch/pier

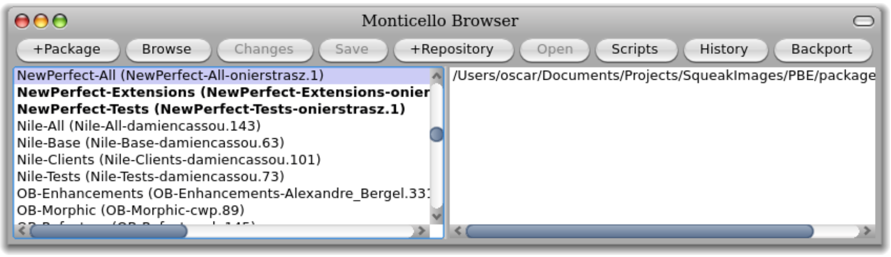
that needs to be saved is Pier−All; all dependent dirty packages are saved automatically.

Let us see how this works in practice. Our Perfect package currently bundles the tests together with the implementation. Suppose we would like instead to separate these into separate packages, so that the implementation can be loaded without the tests. By default, however, we would like to load everything.

Take the following steps:

- *Load version 4 of the* Perfect *package from the package cache*
- *Create a new package in the browser called* NewPerfect−Tests *and drag the class* PerfectTest *to this package*
- *Rename the* *perfect *protocol of the* Integer *class to* *newperfect−extensions *(action-click to rename it)*
- *In the Monticello browser, add the packages* NewPerfect−All *and* NewPerfect−Extensions.
- *Add* NewPerfect−Extensions *and* NewPerfect−Tests *as required packages to* NewPerfect−All *(action-click on* NewPerfect−All*)*
- *Save package* NewPerfect−All *in the package-cache repository. Note that Monticello prompts for comments to save the required packages too.*
- *Check that all three packages have been saved in the package cache.*
- *Monticello thinks that* Perfect *is still loaded. Unload it and then load* NewPerfect−All *from the repository inspector. This will cause* NewPerfect−Extensions *and* NewPerfect−Tests *to be loaded as well as required packages.*
- *Check that all tests run.*

Note that when NewPerfect−All is selected in the Monticello browser, the dependent packages are displayed in bold (see Figure **??**).



NewPerfect-All (NewPerfect-All-onierstrasz.1)
NewPerfect-Extensions (NewPerfect-Extensions-onier
NewPerfect-Tests (NewPerfect-Tests-onierstrasz.1)
Nile-All (Nile-All-damiencassou.143)
Nile-Base (Nile-Base-damiencassou.63)
Nile-Clients (Nile-Clients-damiencassou.101)
Nile-Tests (Nile-Tests-damiencassou.73)
OB-Enhancements (OB-Enhancements-Alexandre_Bergel.33
OB-Morphic (OB-Morphic-cwp.89)

/Users/oscar/Documents/Projects/SqueakImages/PBE/package

Figure 1.22: NewPerfect−All requires NewPerfect−Extensions and NewPerfect−Tests.

If you further develop the Perfect package, you should only load or save NewPerfect−All, not its required packages.

Here is the reason why:

- If you load NewPerfect−All from a repository (package-cache, or anywhere else), this will cause NewPerfect−Extensions and NewPerfect−Tests to be loaded from the same repository.
- If you modify the PerfectTest class, this will cause the NewPerfect−Tests and NewPerfect−All packages to both become dirty (but not NewPerfect−Extensions).
- To commit the change, you should save NewPerfect−All. This will commit a new version of NewPerfect−All which then requires the new version of NewPerfect−Tests. (It will also depend on the existing, unmodified version of NewPerfect−Extensions.) Loading the latest version of NewPerfect−All will also load the latest version of the required packages.
- If instead you save NewPerfect−Tests, this will *not* cause NewPerfect−All to be saved. This is bad because you effectively break the dependency. If you then load the latest version of NewPerfect−All you will not get the latest versions of the required packages. Don't do it!

Do not name your top level package with a suffix (*e.g.*, Perfect) that could match your subpackages. Do not define Perfect as a required package of Perfect−Extensions or PerfectTest. You would get in trouble since Monticello would save all the classes for three packages while you only want two packages and an empty one at the top level.

Class initialization

When Monticello loads a package into the image, any class that defines an initialize method on the class side will be sent the initialize message. The message is sent *only* to classes that define this method on the class side. A class that does not define this method will not be initialized, even if initialize is defined by one of its superclasses. NB: the initialize method is not invoked when you merely reload a package!

Class initialization can be used to perform any number of checks or special actions. A particularly useful application is to add new instance variables to a class.

Class extensions are strictly limited to adding new methods to a class. Sometimes, however, extension methods may need new instance variables to exist.

Suppose, for example, that we want to extend the TestCase class of SUnit with methods to keep track of the history of the last time the test was red. We would need to store that information somewhere, but unfortunately we cannot define instance variables as part of our extension.

A solution would be to define an initialize method on the class side of one of the classes:

```
TestCaseExtension class>>initialize
   (TestCase instVarNames includes: 'lastRedRun')
      ifFalse: [TestCase addInstVarName: 'lastRedRun']
```

When our package is loaded, this code will be evaluated and the instance variable will be added, if it does not already exist.

1.4 Getting a change set from two versions

A Monticello version is the snapshot of one or more packages. A version contains the complete set of class and method definitions that constitute the underlying packages. Sometimes, it is useful to have a "patch" from two versions. A patch is the set of all necessary side effect in the system to go from one version A to another version B.

Change set is a Pharo built-in mechanism to define system patches. A change set is composed of global side effects on the system. New change set may be created and edited from the *Change Sorter*. This tool is available from the World ▷ Tools entry.

The difference between two Monticello versions may be easily captured by creating a new change set before loading a second version of a package. As an illustration, we will capture the differences between version 1 and 2 of the *Perfect* package:

1. Load version 1 of *Perfect* from the Monticello browser
2. Open a change sorter and create a new change set. Let's name it DiffPerfect
3. Load version 2
4. In the change sorter, you should now see the difference between version 1 and 2. The change set may be saved on the filesystem by action-clicking on it and selecting file out. A DiffPerfect.X.cs file is now located next to your Pharo image.

1.5 Kinds of repositories

Several kinds of repositories are supported by Monticello, each with different characteristics and uses. Repositories can be read-only, write-only or read-write. Access rights may be defined globally or can be tied to a particular user (as in SqueakSource, for example).

HTTP. HTTP repositories are probably the most popular kind of repository since this is the kind supported by SqueakSource.

The nice thing about HTTP repositories is that it's easy to link directly to specific versions from web sites. With a little configuration work on the HTTP server, HTTP repositories can be made browsable by ordinary web browsers, WebDAV clients, and so on.

HTTP repositories may be used with an HTTP server other than SqueakSource . For example, a simple configuration[8] turns Apache into a Monticello repository with restricted access rights:

```
"My apache2 install worked as a Monticello repository right out of the box on my
RedHat 7.2 server.  For posterity's sake, here's all I had to add to my apache2 config:"
Alias /monticello/ /var/monticello/
<Directory /var/monticello>
  DAV on
  Options indexes
  Order allow,deny
  Allow from all
  AllowOverride None
  # Limit write permission to list of valid users.
  <LimitExcept GET PROPFIND OPTIONS REPORT>
    AuthName "Authorization Realm"
    AuthUserFile /etc/monticello-auth
    AuthType Basic
    Require valid-user
  </LimitExcept>
</Directory>
"This gives a world-readable, authorized-user-writable Monticello repository in
/var/monticello.  I created /etc/monticello-auth with htpasswd and off I went.
I love Monticello and look forward to future improvements."
```

FTP. This is similar to an HTTP repository, except that it uses an FTP server instead. An FTP server may also offer restricted access right and different FTP clients may be used to browse such Monticello repository.

[8] http://www.visoracle.com/squeak/faq/monticello-1.html

GOODS. This repository type stores versions in a GOODS object database. GOODS is a fully distributed object-oriented database management system that uses an active client model[9]. It's a read-write repository, so it makes a good "working" repository where versions can be saved and retreived. Because of the transaction support, journaling and replication capabilities offered by GOODS, it is suitable for large repositories used by many clients.

Directory. A directory repository stores versions in a directory in the local file system. Since it requires very little work to set up, it's handy for private projects; since it requires no network connection, it's the only option for disconnected development. The `package−cache` we have been using in the exercises for this chapter is an example of this kind of repository. Versions in a directory repository may be copied to a public or shared repository at a later time. SqueakSource supports this feature by allowing package versions (.mcz files) to be imported for a given project. Simply log in to SqueakSource, navigate to the project, and click on the Import Versions link.

Directory with Subdirectories. A "directory with subdirectories" is very similar to "directory" except that it looks in subdirectories to retrieve list of available packages. Instead of having a flat directory that contains all package versions, such as repository may be hierarchically structured with subdirectories.

SMTP. SMTP repositories are useful for sending versions by mail. When creating an SMTP repository, you specify a destination email address. This could be the address of another developer — the package's maintainer, for example — or a mailing list such as pharo-project. Any versions saved in such a repository will be emailed to this address. SMTP repositories are write-only.

Programmatically adding repositories For particular purposes, it may be necessary to programmatically add new repositories. This happens when managing configuration and large set of distributed monticello packages or simply customizing the entries available in the Monticello browser. For example, the following code snippet programmatically adds new directory repositories

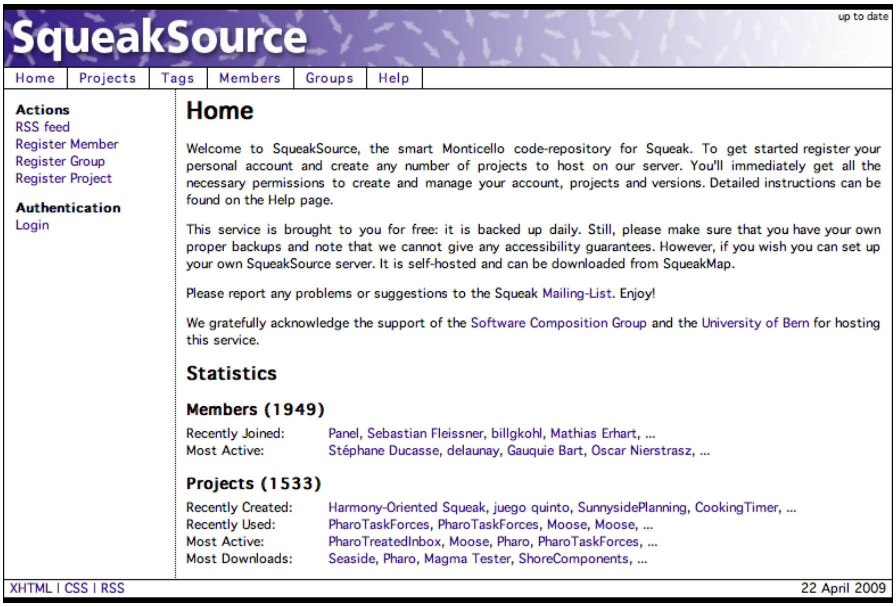
```
{'/path/to/repositories/project−1/'.
'/path/to/repositories/project−2/'.
'/path/to/repositories/project−3/'. } do:
[ :path |
   repo := MCDirectoryRepository new directory:
```

[9] http://www.garret.ru/goods.html

```
    (FileDirectory on: path).
  MCRepositoryGroup default addRepository: repo ].
```

Using SqueakSource

SqueakSource is a online repository that you can use to store your Monticello packages. An instance is running and accessible from http://www.squeaksource.com. At the time this chapter is being written, over 1500 projects are registered on SqueakSource and nearly 2000 people have an account. Figure **??** shows the main web page.



Actions
RSS feed
Register Member
Register Group
Register Project

Authentication
Login

Home

Welcome to SqueakSource, the smart Monticello code-repository for Squeak. To get started register your personal account and create any number of projects to host on our server. You'll immediately get all the necessary permissions to create and manage your account, projects and versions. Detailed instructions can be found on the Help page.

This service is brought to you for free: it is backed up daily. Still, please make sure that you have your own proper backups and note that we cannot give any accessibility guarantees. However, if you wish you can set up your own SqueakSource server. It is self-hosted and can be downloaded from SqueakMap.

Please report any problems or suggestions to the Squeak Mailing-List. Enjoy!

We gratefully acknowledge the support of the Software Composition Group and the University of Bern for hosting this service.

Statistics

Members (1949)

Recently Joined:	Panel, Sebastian Fleissner, billgkohl, Mathias Erhart, ...
Most Active:	Stéphane Ducasse, delaunay, Gauquie Bart, Oscar Nierstrasz, ...

Projects (1533)

Recently Created:	Harmony-Oriented Squeak, juego quinto, SunnysidePlanning, CookingTimer, ...
Recently Used:	PharoTaskForces, PharoTaskForces, Moose, Moose, ...
Most Active:	PharoTreatedInbox, Moose, Pharo, PharoTaskForces, ...
Most Downloads:	Seaside, Pharo, Magma Tester, ShoreComponents, ...

XHTML | CSS | RSS 22 April 2009

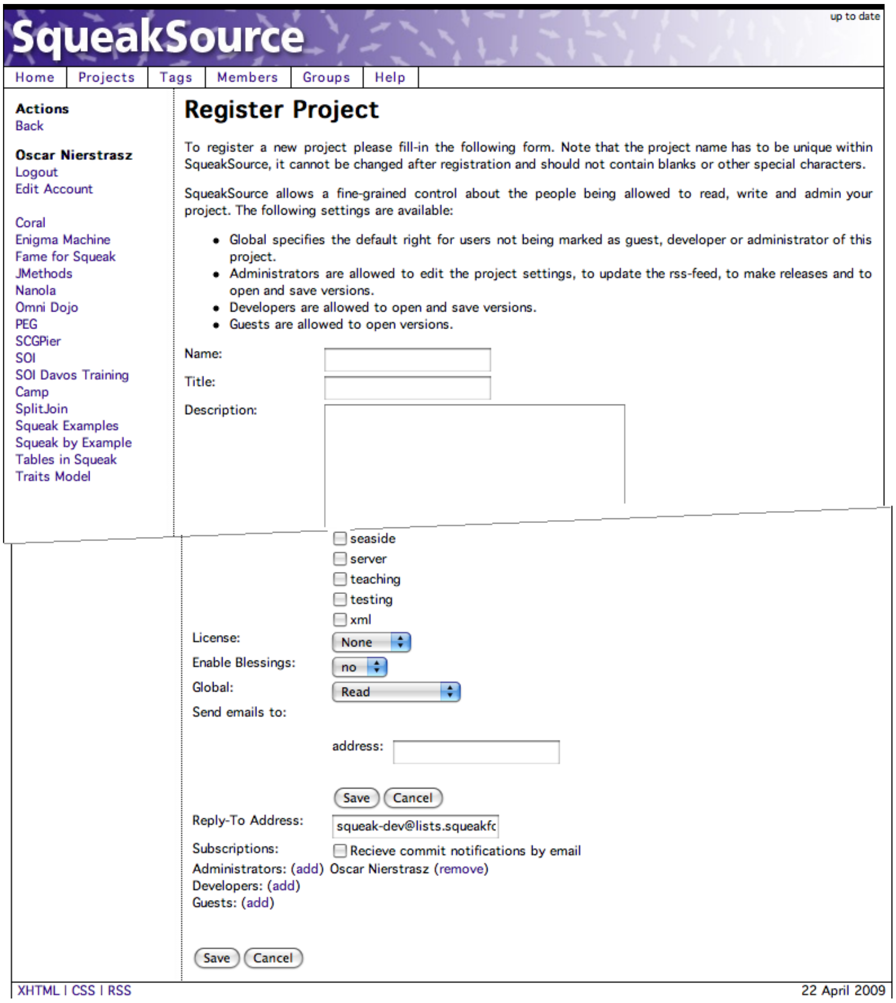
Figure 1.23: SqueakSource, the online Monticello code repository.

Use a web browser to visit the http://PharoByExample.org project at http://www.squeaksource.com/PharoByExample.html. This project contains the Lights Out project from the first volume of this book. In the registration section on that web page you should see this repository expression:

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

Add this repository to Monticello by clicking +Repository*, and then selecting* HTTP*. Fill out the template with the URL corresponding to the Lights Out project — you can copy the above repository expression from the web page and paste it into the template. Since you are not going to commit new versions of this package, you do not need to fill in the user and password.* Open *the repository, select the latest version and click* Load*.*

Pressing the Register Member link on the SqueakSource home page will probably be your first step if you do not have a SqueakSource account. Once you are a member, Register Project allows you to create a new project.



Register Project

To register a new project please fill-in the following form. Note that the project name has to be unique within SqueakSource, it cannot be changed after registration and should not contain blanks or other special characters.

SqueakSource allows a fine-grained control about the people being allowed to read, write and admin your project. The following settings are available:

- Global specifies the default right for users not being marked as guest, developer or administrator of this project.
- Administrators are allowed to edit the project settings, to update the rss-feed, to make releases and to open and save versions.
- Developers are allowed to open and save versions.
- Guests are allowed to open versions.

Name:
Title:
Description:

☐ seaside
☐ server
☐ teaching
☐ testing
☐ xml

License: None
Enable Blessings: no
Global: Read
Send emails to:
address:
Save Cancel
Reply-To Address: squeak-dev@lists.squeakfc
Subscriptions: ☐ Recieve commit notifications by email
Administrators: (add) Oscar Nierstrasz (remove)
Developers: (add)
Guests: (add)

Save Cancel

XHTML | CSS | RSS 22 April 2009

Figure 1.24: Repositories under SqueakSource are highly configurable.

Monticello offers a large range of options (cf. Figure **??**) to configure a project repository: tags may be assigned, a license may be chosen, access for people who are not part of the project may be restricted (read/write, read, no access), emails may be sent upon commits, mailing list may be managed, and users may be defined to be members of the project (as administrator, developer, or guest).

1.6 The .mcz file format

Versions are stored in repositories as binary files. These files are commonly call "mcz files" as they carry the extension .mcz. This stands for "Monticello zip" since an mcz file is simply a zipped file containing the source code and other meta-data.

An mcz file can be dragged and dropped onto an open image file, just like a change set. Pharo will then prompt you to ask if you want to load the package it contains. Monticello will not know which repository the package came from, however, so do not use this technique for development.

You may try to unzip such a file, for example to view the source code directly, but normally end users should not need to unzip these files themselves. If you unzip it, you will find the following members of the mcz file.

File contents Mcz files are actually ZIP archives that follow certain conventions. Conceptually a version contains four things:

- *Package*. A version is related to a particular package. Each mcz file contains a file called "package" that contains information about the package's name.
- *VersionInfo*. This is the meta-data about the snapshot. It contains the author initials, date and time the snapshot was taken, and the ancestry of the snapshot. Each mcz file contains a member called "version" which contains this information.

  A version doesn't contain a full history of the source code. It's a snapshot of the code at a single point in time, with a UUID identifying that snapshot, and a record of the UUIDs of all the previous snapshots it's descended from.

- *Snapshot*. A Snapshot is a record of the state of the package at a particular time. Each mcz file contains a directory named "snapshot/". All the members in this directory contain definitions of program elements, which when combined form the Snapshot. Current versions of Monticello only create one member in this directory, called "source.st".

- *Dependencies*. A version may depend on specific version of other packages. An mcz file may contain a "dependencies/" directory with a member for each dependency. These members will be named after each package the Monticello package depends upon. For example, a Pier−All mcz file will contains files named Pier−Blog and Pier−Caching in its dependencies directory.

Source code encoding The member named "snapshot/source.st" contains a standard fileout of the code that belongs to the package.

Metadata encoding The other members of the zip archive are encoded using S-expressions. Conceptually, the expressions represent nestable dictionaries. Each pair of elements in a list represent a key and value. For example, the following is an excerpt of a "version" file of a package named AA:

```
(name 'AA−ab.3' message 'empty log message' date '10 January 2008' time '10
:31:06 am' author 'ab' ancestors ((name 'AA−ab.2' message...)))
```

It basically says that the version AA−ab.3 has an empty log message, was created on January 10, 2008, by ab, and has an ancestor named AA−ab.2, ...

1.7 Chapter Summary

This chapter has presented the functionality of Monticello in detail. The following points were covered:

- Monticello are mapped to Smalltalk categories and method protocols. If you add a package called Foo to Monticello, it will include all classes in categories called Foo or starting with Foo−. It will also include all methods in those categories, except those in protocols starting with *. Finally it will include all *class extension* methods in protocols called *foo or starting with *foo− anywhere else in the system.

- When you modify any methods or classes in a package, it will be marked as "dirty" in Monticello, and can be saved to a repository.

- There are many kinds of repositories, the most popular being HTTP repositories, such as those hosted by SqueakSource.

- Saved packages are caches locally in a directory called package−cache.
- The Monticello repository inspector can be used to browse a repository. You can select which versions of packages to load or unload.
- You can create a new *branch* of a package by basing a new version on another version which is earlier than the latest version. The repository inspector keeps track of the ancestry of packages and can tell you which versions belong to separate branches.
- Branches can be *merged*. Monticello offers a fine degree of control over the resolution of conflicts between merged versions. The merged version will have as its ancestor the two versions it was merged from.
- Alternatively, selected changes of a branch can be *backported* to an arbitrary earlier version. This will create a new version that can be merged with any other version that needs those changes. The original backported branch remains independent in this case.
- Monticello can keep track of dependencies between packages. When a package with dependencies to required packages is saved, a new version of that package is created, which then depends on the latest versions of all the required packages.
- If classes in your packages have class-side initialize methods, then initialize will be sent to those classes when your package is loaded. This mechanism can be used to perform various checks or start-up actions. A particularly useful application is to add new instance variables to classes for which you are defining extension methods.
- Monticello stores package versions in a special zipped file with the file extension .mcz. The mcz file contains a snapshot of the complete source code of that version of your package, as well as files containing other important metadata, such a package dependencies.
- You can drag and drop an mcz file onto your image as a quick way to load it.