Chapter 1

Versioning Your Code with Monticello

Co-written with
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 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], Subversion[2], and Git[3] 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.

There are multiple repositories to store your projects. SmalltalkHub[4] and Squeaksource 3[5] are the two main and free-to-use repositories. They are versioning systems for Pharo in which classes and methods, rather than lines of

[1] http://www.nongnu.org/cvs
[2] http://subversion.tigris.org
[3] http://git-scm.com/
[4] http://smalltalkhub.com/
[5] http://ss3.gemstone.com/

text, are the units of change. In this chapter we will use SmalltalkHub, but Squeaksource 3 can be use similarly. *SmalltalkHub* is a central online repository in which you can store versions of your applications using Monticello. SmalltalkHub is the equivalent of SourceForge, and Monticello the equivalent of CVS.

In this chapter, you will learn how to use use Monticello and SmalltalkHub to manage your software. We have already been acquainted with Monticello briefly in earlier chapters[6]. 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[7] 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 package* 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. Monticello Browser can be selected from the *World* menu. In Figure 1.1, we see that the Monticello Browser consists of two list panes and one button pane. The left pane

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

[7] 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.

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.

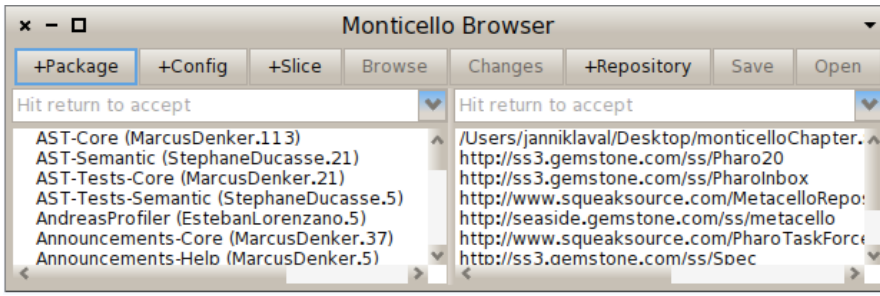

Figure 1.1: The Monticello Browser.

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.

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

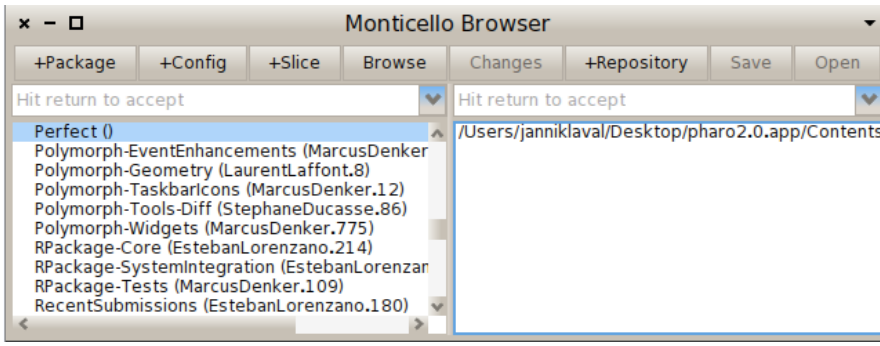

Figure 1.2: Creating the Perfect package.

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 1.2 that the Save button is disabled (greyed out).

Before we save our Perfect package, we need to specify to *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 1.5), Monticello supports a large choice of repositories, though the most commonly used is HTTP, since this is the one used by SmalltalkHub.

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 1.1). 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 +Repository button. This will offer a number of choices of different kind 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* Save, *enter an appropriate log message, and* Accept *to save the changes.*

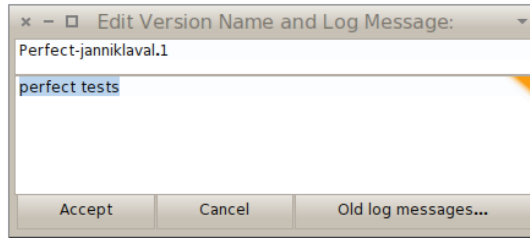

Figure 1.3: You may set a new version name and a commit message when you save a version of a package.

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 confirm that a file* Perfect−XX.1.mcz *was created in your package cache.* XX *corresponds to your name or initials.*[8]

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.*

[8] 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".

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

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

PerfectTest»testPerfectBoundary
   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.

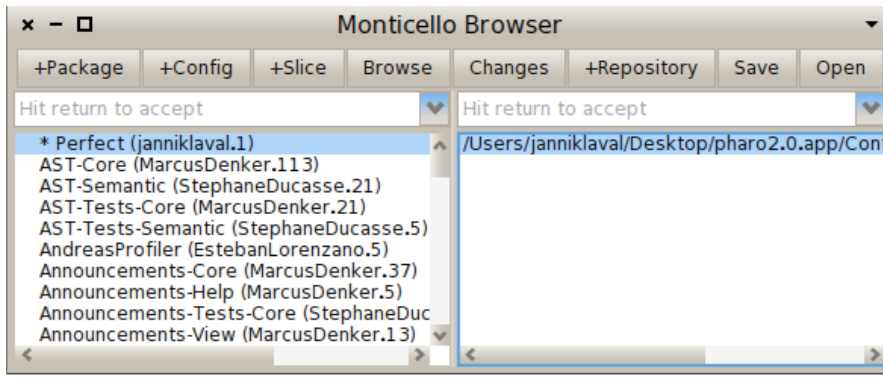

Figure 1.4: Modifying our Perfect package will "dirty" it.

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 *and* Changes *buttons to see what they do.* 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 1.5).

Select the package−cache *repository and open it. You should see something like Figure 1.5.*

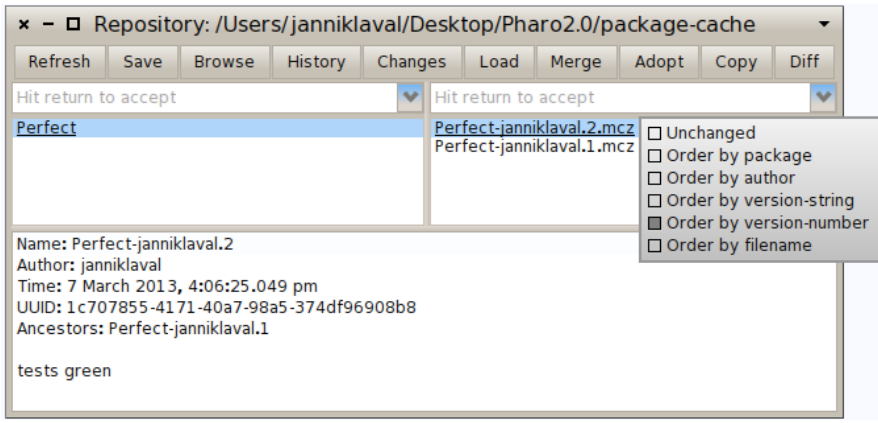

Figure 1.5: A repository inspector.

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.

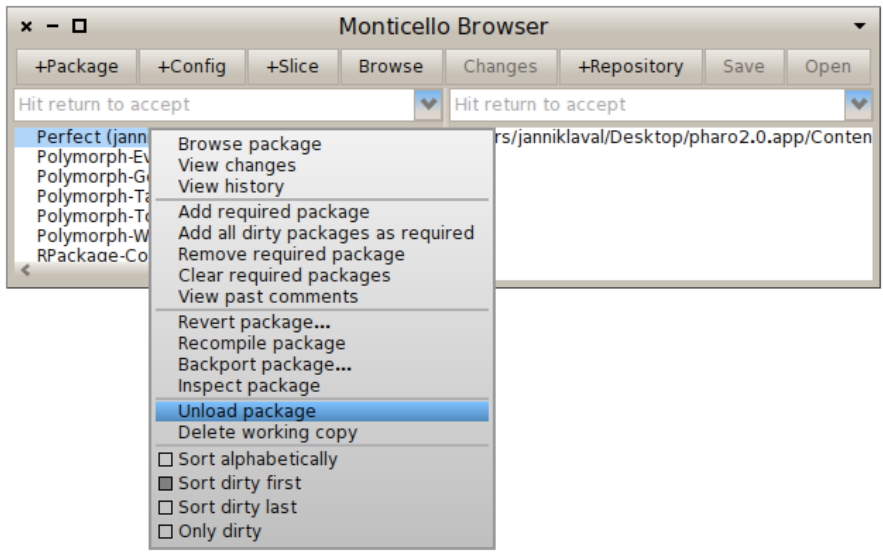

Figure 1.6: Unloading a package.

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.*

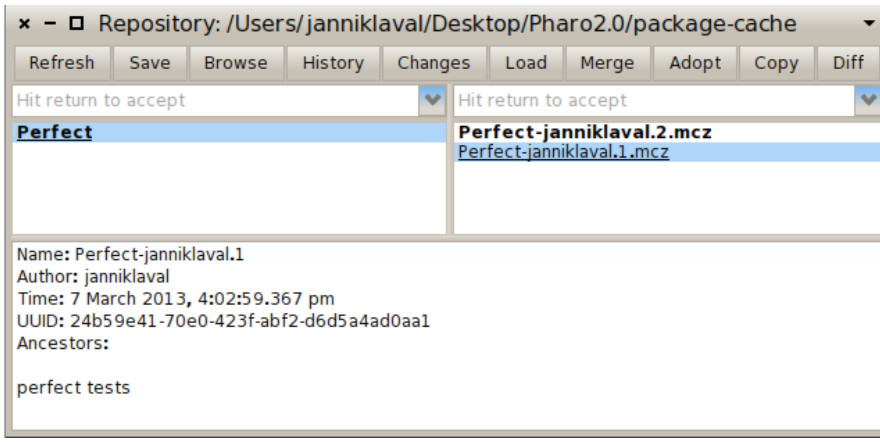

Figure 1.7: Loading an earlier version.

You should now be able to verify that 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.*

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 a 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 doing software maintenance 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.

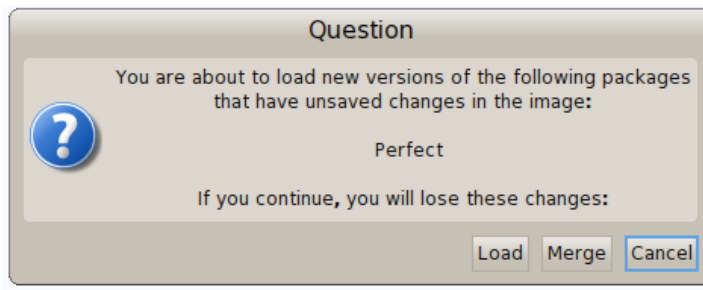

Figure 1.8: Unsaved changes warning.

Select Cancel *to avoid overwriting your new methods. Now* Save *your changes. 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 1.9).*

Merging

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)

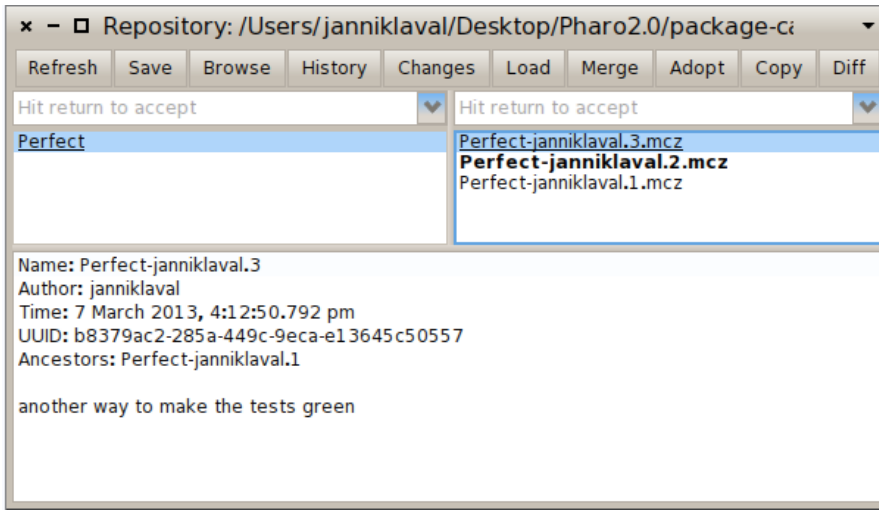

Figure 1.9: Versions 2 and 3 are separate branches of version 1.

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 1.10. 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.

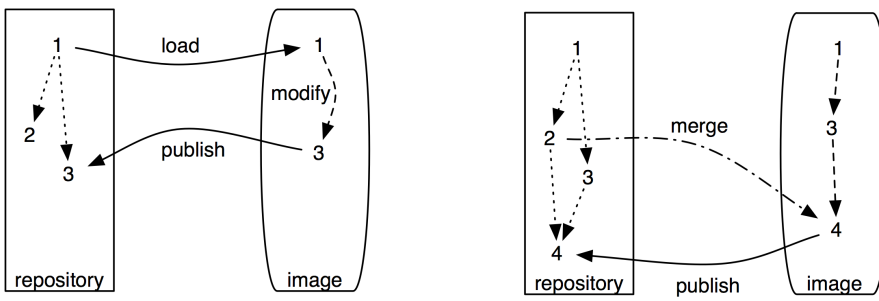

Figure 1.10: Branching (left) and merging (right).

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 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 1.10.

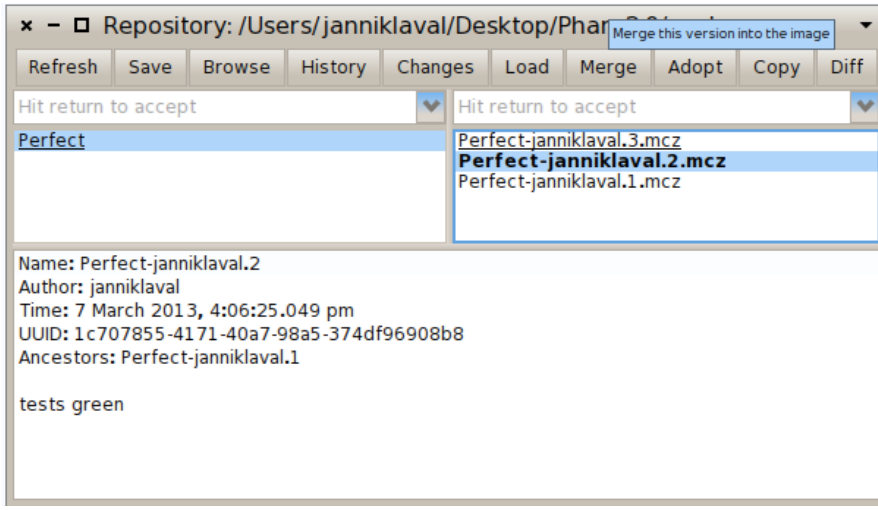

Figure 1.11: Select a separate branch (in bold) to be merged.

Select version 2 in the repository browser, as shown in Figure 1.11, 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.

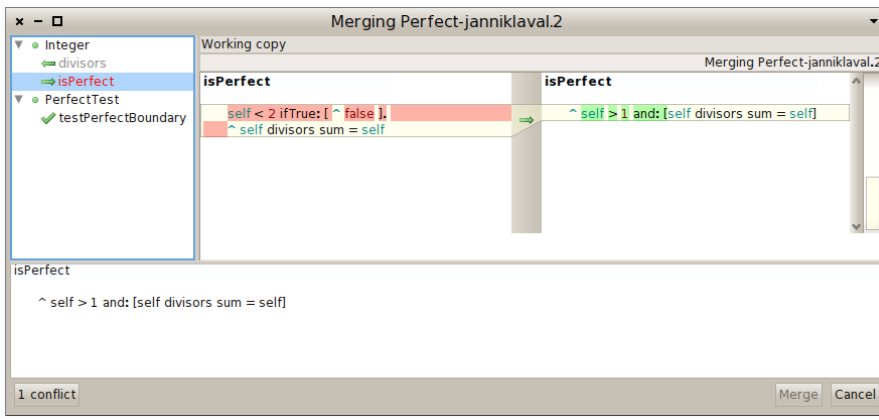

Figure 1.12: Version 2 of the Perfect package being merged with the current version 3.

In Figure 1.12 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 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 1.12 shows two 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.

Red=A method is conflicting. A decision needs to be made 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 right clicking on the method and Keep current version or Use incoming version.

Right arrow=Repository replace current. An element with a right arrow will be used and replace the current element in the image. In Figure 1.12 we see that Integer»isPerfect from version 2 has been used.

Left arrow=Repository version rejected. An element with left arrow has been rejected, and the local definition will not be replaced. In Figure 1.12 Integer»divisors from version 2 is rejected, so the definition from version 3 will remain.

 *Use incoming version of* Integer»>isPerfect *and keep current version of* Integer» divisors, *and click the* Merge *button. Confirm that the tests are all green. Commit the new merged version of* Perfect *as 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 1.13).

1.2 Exploring Monticello repositories

Monticello has many other useful features. As we can see in Figure 1.1, the Monticello browser window has eight buttons. We have already used four of them — +Package, Save, +Repository and Open. We will now look at Browse and Changes which are used to explore the state and history of repositories

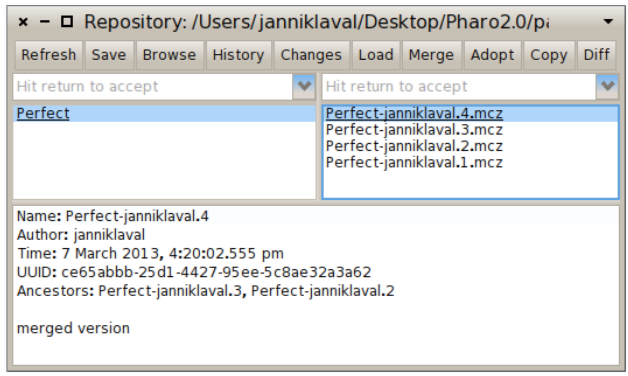

Figure 1.13: All older versions are now ancestors of merged version 4.

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.*

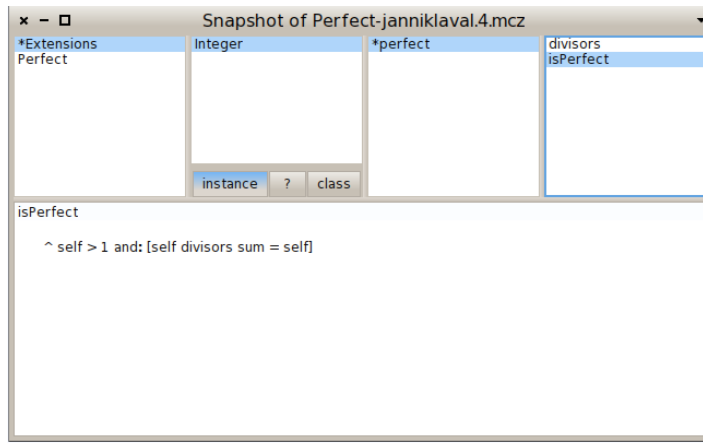

Figure 1.14: The snapshot browser reveals that the Perfect package extends the class Integer with 2 methods.

For example, Figure 1.14 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 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)
```

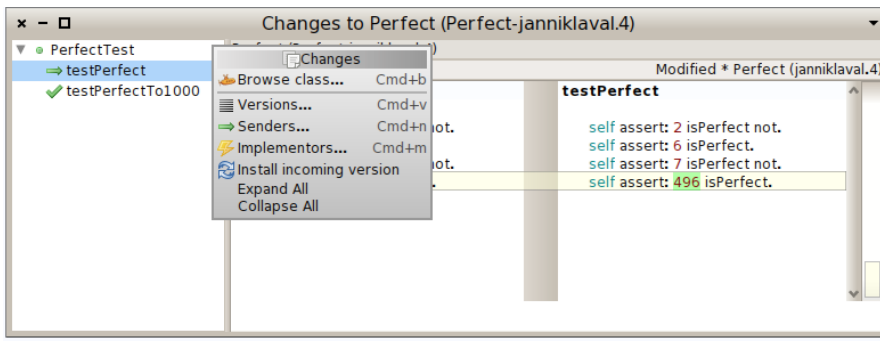

Figure 1.15: The patch browser shows the difference between the code in the image and the most recently committed version.

Figure 1.15 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.

1.3 Advanced topics

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

History

By action-clicking on a package, you can select the item History. It opens a version history viewer that displays the comments committed along with each version of the selected package (see Figure 1.16). 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, right click and select the* History *item.*

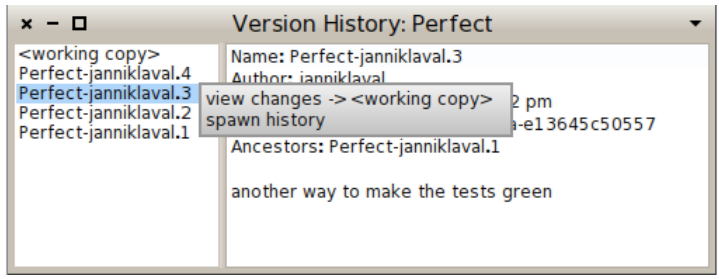

Figure 1.16: The version history viewer provides information about the various versions of a package.

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.

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[9], a meta-described content management system. Pier is a large piece of software with many facets (tools, documentations, blog, catch strategies, security, etc). 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

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

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 1.17 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.

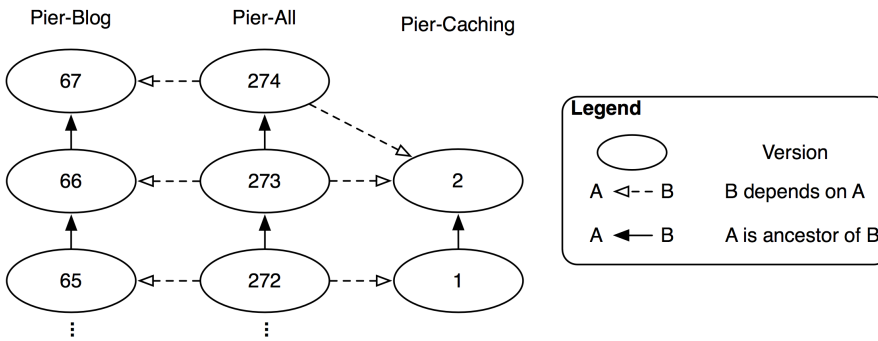

Figure 1.17: Dependencies in Pier.

Because of these dependencies, installing Pier−All causes all the other Pier packages to be installed. Furthermore, when developing, the only package 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 1.18).

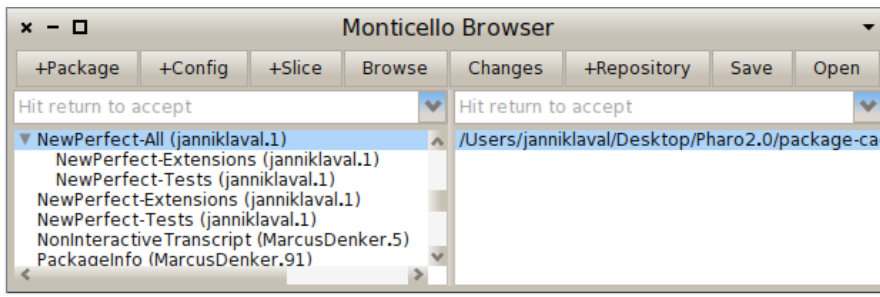

Figure 1.18: NewPerfect−All requires NewPerfect−Extensions and NewPerfect−Tests.

> 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 run into in trouble as Monticello would save all the classes for three packages, though you only want two packages and an empty one at the top level.

To build more flexible dependencies between packages, we recommend using a Metacello configuration (see Chapter **??**). The +Config button creates a kind of configuration structure. The only thing to do is to add the dependencies.

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. Note that if you change a class that is not in your package, the other package will become dirty. In the previous example, the package SUnit contains TestCase. After installing TestCaseExtension, the package SUnit will become dirty.

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 effects 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. A new change sets 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 SmalltalkHub, for example).

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

The nice thing about HTTP repositories is that it is 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 SmalltalkHub. For example, a simple configuration[10] 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 a Monticello repository.

[10] 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[11]. 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. Because 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. SmalltalkHub supports this feature by allowing package versions (.mcz files) to be imported for a given project. Simply log in to SmalltalkHub, 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 lists 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 that 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 configurations 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:
```

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

```
    (path asFileReference).
  MCRepositoryGroup default addRepository: repo ].
```

Using SmalltalkHub

SmalltalkHub is a online repository that you can use to store your Monticello packages. An instance is running and accessible from http://smalltalkhub.com/.

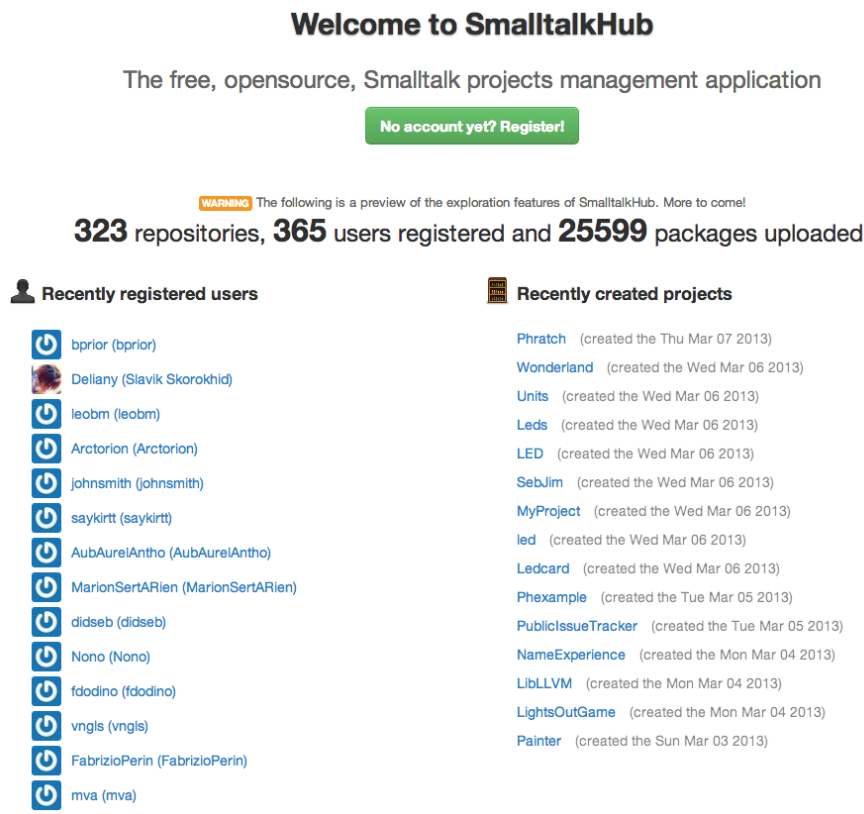

Figure 1.19: SmalltalkHub, the online Monticello code repository.

 *Use a web browser to visit the main page http://smalltalkhub.com/. When you select a project, you should see this kind of* repository expression:

```
MCHttpRepository
   location: 'http://smalltalkhub.com/mc/PharoExtras/Phexample/main'
   user: ''
   password: ''
```

Add this repository to Monticello by clicking +Repository*, and then selecting* HTTP*. Fill out the template with the URL corresponding to the 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 of Phexample and click* Load*.*

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

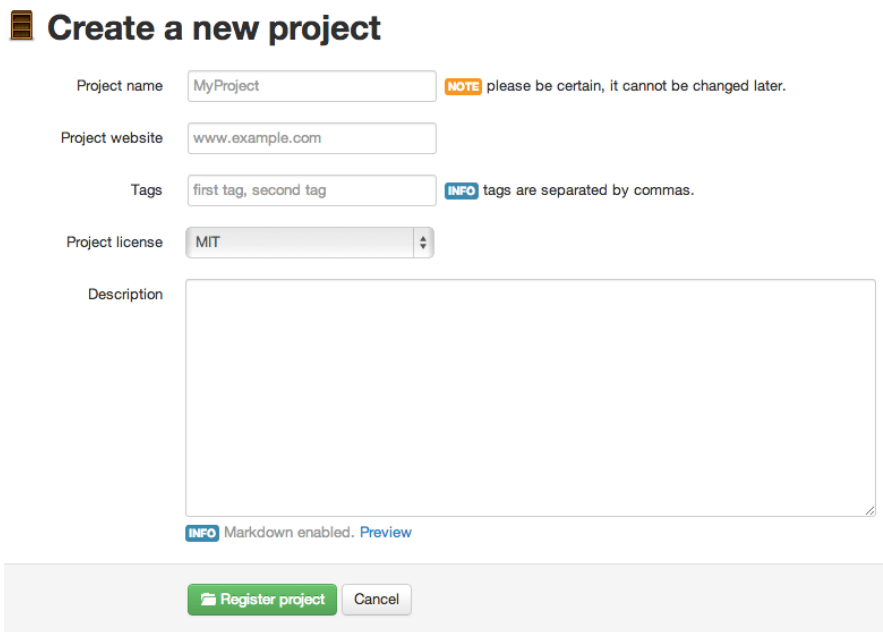

Figure 1.20: Repositories under SmalltalkHub are configurable.

SmalltalkHub offers options (cf. Figure 1.20) to configure a project repository: tags may be assigned, a license may be chosen with access for people who are not part of the project may be restricted (private, public), and users may be defined to be members of the project. You also can create a team that shares projects.

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 contain 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 SmalltalkHub.
- 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 from which it merged.

- 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.