

Chapter 1

Managing projects with Metacello

with the participation of:

Dale Henrichs (dale.henrichs@gemstone.com)

Mariano Martinez Peck (marianopeck@gmail.com)

Have you ever had this problem when trying to load a project: you get an error because a package that you were not even aware of is missing? Or worse—it is present, but you have the wrong version? This situation can easily occur, even though the project loads fine for its developers, when the developers are working in a context that is different from yours.

The solution is for the project developers to use a *package management system* to explicitly manage the dependencies between the packages that make up a project. This chapter shows you how to use Metacello, Pharo’s package management system, and the benefits of using it.

1.1 Introduction

We say that Metacello is a *package management system* for Monticello. But what, exactly, does that mean? A package management system is a collection of tools that automate the process of installing, upgrading, configuring, and removing *sets* of software packages. Metacello groups packages to simplify things for the user, and manages dependencies, *i.e.*, which versions of what components should be loaded to make sure that the whole set of packages is coherent.

A package management system provides a consistent way to install pack-

ages. Indeed, package management systems are sometimes incorrectly referred to as installers. This can lead to confusion, because a package management system does a lot more than install software. You may have used package management systems in other contexts: examples include Envy (in VisualAge Smalltalk), Maven (in Java), and apt-get/aptitude (in Debian and Ubuntu).

One of the key features of a package management system is that it should *correctly load any package*: you should never need to manually install anything. To make this possible, each dependency, and the dependencies of the dependencies, and so on, must be specified in the description of the package, with enough information to allow the package management tools to load them in the correct order.

As an example of the power of Metacello, you can take a PharoCore image, and load *any* package of *any* project without any problems with dependencies. Of course, Metacello does not do magic: this only works as long as the package developers have properly defined the dependencies.

1.2 One tool for each job

Pharo provides three tools for managing software packages; they are closely related, but each has its own purpose. The tools are Monticello, which manages versions of source code, Gofer, which is a scripting interface for Monticello, and Metacello, which is a package management system.

Monticello: source code versioning. Source code versioning is the process of assigning unique versions to particular software states. It is also called revision control. In particular, source code versioning incrementally keeps track of different versions, also known as revisions, of “pieces of software”. In object-oriented programming, these “pieces of software” are methods, classes or packages. A source code versioning system lets you commit a new version, update to a new version committed by someone else, merge changes, look at the differences between versions, revert to an older version, etc.

Pharo uses the Monticello source code versioning system, which manages Monticello packages. Monticello lets us do all of the above operations on individual packages, but Monticello does not provide a good way to easily specify dependencies *between* packages, identify stable versions of a package, or group packages into meaningful units. Chapter ?? describes it.

Gofer: Monticello’s scripting interface. Gofer is a small tool that sits on top of Monticello: it is used to load, update, merge, difference, revert, commit, recompile and unload groups of Monticello packages. Gofer

also makes sure that these operations are performed as cleanly as possible. For more information, see Chapter ??.

Metacello: package management. Metacello introduces the notion of a project as a set of related Monticello packages, and is used to manage projects, their dependencies, and their metadata. Metacello also manages dependencies between packages.

1.3 Metacello features

Metacello is consistent with the important features of Monticello. It is based on the following ideas.

Declarative project descriptions. A Metacello project has named versions consisting of lists of Monticello package *versions*. Dependencies are explicitly expressed in terms of named versions of required projects. A *required project* is a reference to another Metacello project. Collectively, all of these descriptions are called the project metadata.

Project metadata are versioned. Metacello project metadata is represented as instance methods in a class. Managing such metadata as code brings a lot of power in comparison to XML, which is what most package management systems use. Metacello project metadata can themselves be stored as a Monticello package, and are thus subject to version control. As a result, concurrent updates to the project metadata can be managed easily: parallel versions of the metadata can be merged just like parallel versions of the code base itself.

Metacello has the following features:

Cross-platform: Metacello runs on all platforms that support Monticello, which currently means Pharo, Squeak and GLASS.

Conditional package loading: to enable projects to run on multiple platforms, Metacello supports conditional loading of platform-specific Monticello packages.

Configurations: Metacello actually manages not projects but *configurations of projects*. This is because large projects frequently have multiple variants: there might be a configuration for Pharo and another configuration for Squeak, which might have different prerequisites. Moreover, there might be stable, released configurations as well as experimental configurations that have cool new features but also more bugs.

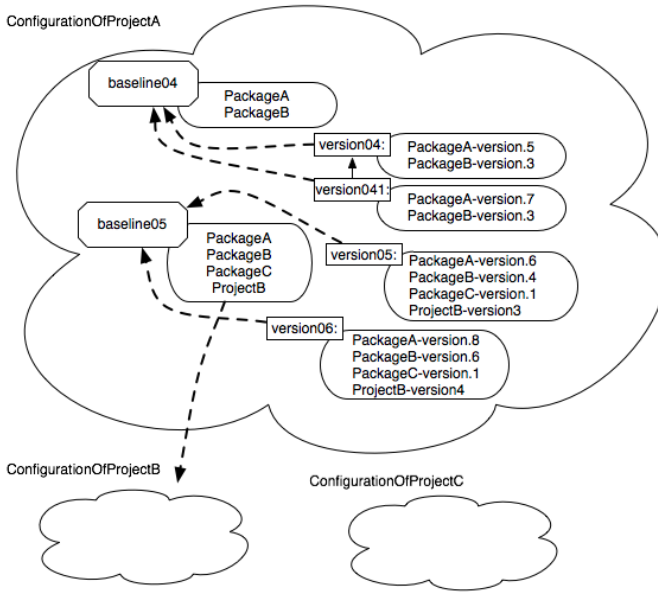


Figure 1.1: Configurations: groups of baselines and versions with dependencies.

Metacello supports the definition of two kinds of entities (represented as methods): *baselines* and *versions*.

Baselines. A baseline is a skeleton identifying abstractly the packages and subprojects that compose a configuration. Structural dependencies as well as code repositories are defined at the baseline level. The baseline can also specify the level of stability of the required components.

Versions. A version is a set of package *versions* that should be loaded. Often a version simply enriches a baseline with specific version information.

Let us explain this using Figure 1.1. ConfigurationOfProjectA contains several baselines (here two baselines 0.4 and 0.5) and versions (here 0.4, 0.41, 0.5, and 0.6). Baseline 0.4 defines that it is composed of two packages (PackageA and PackageB). Baseline 0.5 is composed of 3 packages (PackageA, PackageB, and PackageC) and it also depends on Project B. A version uses the description held in a baseline and refines it using specific package version. A version may also refine another one.

A client (another configuration or you) can load a given version of ProjectA: the loader uses the structural information defined in a baseline and the

specific version information to load the adequate packages. All these points are explained in detail in the rest of this chapter.

1.4 A Simple Case Study

In this example we start with a really simple configuration expressed only with versions and then we gradually add baselines. In normal life, it is better to directly start with a baseline and version.

Let's start using Metacello to manage a software project called *CoolBrowser*. The first step is to create a Metacello configuration for the project by simply copying the class `MetacelloConfigTemplate` and naming it `ConfigurationOfCoolBrowser`. A configuration is a class that describes the currently available configurations of a project (set of baselines and versions), *i.e.*, what we previously called metadata. A configuration represents different versions of projects so that you can load a project in different environment or in different versions of Pharo. By convention, the name of a Metacello configuration is constructed by prefixing the name of the project with 'ConfigurationOf'. To do this, find the class `MetacelloConfigTemplate` in the system browser, right click on the class name, and select the option `copy`.

This is the class definition:

```
Object subclass: #ConfigurationOfCoolBrowser
  instanceVariableNames: 'project'
  classVariableNames: 'LastVersionLoad'
  poolDictionaries: ''
  category: 'Metacello-MC-Model'
```

You will notice that `ConfigurationOfCoolBrowser` has some instance- and class-side methods; we will explain later how they are used. Notice also that this class inherits from `Object`. This is deliberate: it's important that Metacello configurations can be loaded without any prerequisites, including Metacello itself, so Metacello configurations cannot rely on a common superclass.

Now imagine that the project `CoolBrowser` has several versions, for example, 1.0, 1.0.1, 1.4, and 1.67. With Metacello, you create configuration methods, instance-side methods that describe the contents of each version of the project. Method names for version methods are unimportant as long as the method is annotated with the `<version: >` pragma (method tagging or annotations for Pharo), as shown below. However, there is a convention that version methods are named `versionXXX;`, where XXX is the version number with illegal characters (like `'.'`) removed.

Suppose for the moment that the project `CoolBrowser` contains two packages: `CoolBrowser-Core` and `CoolBrowser-Tests` (see Figure 1.2). A configuration method (here a version) method might look like the following one.



Figure 1.2: Simple version.

```

ConfigurationOfCoolBrowser>>version01: spec
<version: '0.1'>

spec for: #common do: [
  spec repository: 'http://www.example.com/CoolBrowser'.
  spec
    package: 'CoolBrowser-Core' with: 'CoolBrowser-Core-BobJones.10';
    package: 'CoolBrowser-Tests' with: 'CoolBrowser-Tests-JohnLewis.3' ]

```

The method `version01: spec` builds a description of version 0.1 of the project in the object `spec`. The common code for version 0.1 (specified using the message `for: #common do:`) consists of particular versions of the packages named 'CoolBrowser-Core' and 'CoolBrowser-Tests'. These are specified with the message `package: packageName with: versionName`. These versions are available in the Monticello repository 'http://www.example.com/CoolBrowser', which is specified using the message `repository:`.

Now let us look at more details.

- Immediately after the method selector you see the pragma definition: `<version: '0.1'>`. The pragma `version:` indicates that the version created in this method should be associated with version 0.1 of the CoolBrowser project. That's why we said that the name of the method is not that important. Metacello uses the pragma, not the method name, to identify the version being defined.
- The argument of the method, `spec`, is the only variable in the method and it is used as the receiver of four different messages: `for:do:`, `package:with:`, and `repository:`.
- Each time a block is passed as argument of the messages (`for:do:`, `package:with:...`) is executed a new object is pushed on a stack and the messages within the block are sent to the object on the top of the stack.
- The symbol `#common` indicates that this project version is common to all platforms. In addition to `#common`, there are pre-defined attributes for each of the platforms on which Metacello runs (`#pharo`, `#squeak`,

#gemstone, #squeakCommon, #pharo1.2.x, #pharo1.3.x, etc.). In Pharo, the method `metacelloPlatformAttributes` defines the tags that you can use.

About passwords. Sometimes, a Monticello repository requires a username and password. In such case, instead of `repository:`, you can use the message `repository:username:password:`.

```
spec repository: 'http://www.example.com/private' username: 'foo' password: 'bar'
```

Specification objects. A spec object is an object representing all the information about a given version. A version is just a number while the specification is the object. You can access (normally this is not needed) the specification using the `spec` message.

```
(ConfigurationOfCoolBrowser project version: '0.1') spec
```

This answers an object (of class `MetacelloMCVersionSpec`) that contains exactly the information of the method that defines version '0.1'.

Creating a new version. Let us assume that version 0.2 of our project consists of the package versions 'CoolBrowser-Core-BobJones.15' and 'CoolBrowser-Tests-JohnLewis.8' and a new package 'CoolBrowser-Addons' with version 'CoolBrowser-Addons-JohnLewis.3'. We specify this new configuration by creating the following method named `version02:`.

```
ConfigurationOfCoolBrowser>>version02: spec
<version: '0.2'>

spec for: #common do: [
    spec repository: 'http://www.example.com/CoolBrowser'.
    spec
        package: 'CoolBrowser-Core' with: 'CoolBrowser-Core-BobJones.15';
        package: 'CoolBrowser-Tests' with: 'CoolBrowser-Tests-JohnLewis.8';
        package: 'CoolBrowser-Addons' with: 'CoolBrowser-Addons-JohnLewis.3']
```

Naming your Configuration. Previously, we learned the convention to name configuration classes. In our example, `ConfigurationOfCoolBrowser`. There is also a convention to create a Monticello package with the same name as the configuration class, and to put the class in that package. So in this example you will create a package `ConfigurationOfCoolBrowser` containing exactly one class, `ConfigurationOfCoolBrowser`.

By making the package name and the configuration class name the same, and by starting them with the string `ConfigurationOf`, we make it easy to scan



Figure 1.3: Two versions of a project.

through a repository listing the available projects. It is also very convenient to have the configurations stored in their own Monticello repository.

As a general practice, we suggest that initially you save the Configuration package in your working project. When you decide it is ready for release, you can copy it into the MetacelloRepository. A process for publishing configurations in specific distribution repositories (like the previously mentioned) is currently being defined.

1.5 Loading a Metacello Configuration

Of course, the point of specifying project configurations in Metacello is to be able to load exactly that configuration into your image, and thus to be sure that you have a coherent set of package versions. To load versions, you send the message `load` to a version. Here are some examples for loading versions of the CoolBrowser:

```
(ConfigurationOfCoolBrowser project version: '0.1') load.
(ConfigurationOfCoolBrowser project version: '0.2') load.
```

Note that in addition, if you print the result of each expression, you get a list of packages *in load order*: Metacello manages not only which packages are loaded, but also the order. It can be handy to debug configurations.

Selective Loading. By default, the `load` message loads all the packages associated with the version (as we will see later, we can change that by defining a particular group called 'default'). If you want to load a subset of the packages in a project, you should list the names of the packages that you are interested in as an argument to the `load: method`:


```
((ConfigurationOfCoolBrowser project version: '0.2') load:
  { 'CoolBrowser-Core' .
    'CoolBrowser-Addons' }).
```

Debugging Configuration. If you want to simulate the loading of a configuration, without actually loading it, you should use `record:` instead of `load:`. Then to get the result of the simulation, you should send it the message `loadDirective` as follows:

```
((ConfigurationOfCoolBrowser project version: '0.2') record:
  { 'CoolBrowser-Core' .
    'CoolBrowser-Addons' }) loadDirective.
```

Apart from `load:` and `record:`, there is also another useful method which is `fetch`. As explained, `record:` simply records which Monticello files should be downloaded and in which order. `fetch:` accesses and downloads all the needed Monticello files. Just for the record, in the implementation `load:` first does a `fetch:` and then a `doLoad`.

1.6 Managing Dependencies between Packages

A project is generally composed of several packages, which often have dependencies to other packages. It is also likely that a certain package depends on a specific version of another package. Handling dependencies correctly is really important and is one of the major benefits of Metacello. There are two types of dependencies:

Internal dependencies. There are several packages inside a project; some of them depend on other packages of the same project.

Dependencies between projects. it is common for a project to depend on another project, or on some packages from another project. For example, *Pier* (a meta-described content management system) depends on *Magritte* (a metadata modeling framework) and *Seaside* (a framework for web application development).

Internal Dependencies. Let us focus on internal dependencies for now: imagine that the packages *CoolBrowser-Tests* and *CoolBrowser-Addons* both depend on *CoolBrowser-Core* as described in Figure 1.4. The specifications for versions 0.1 and 0.2 did not capture this dependency. Here is a new configuration that does:



Figure 1.4: Version 0.3 expresses internal dependencies between packages in the same project.

```

ConfigurationOfCoolBrowser>>version03: spec
<version: '0.3'>

spec for: #common do: [
  spec repository: 'http://www.example.com/CoolBrowser'.
  spec
    package: 'CoolBrowser-Core' with: 'CoolBrowser-Core-BobJones.15';
    package: 'CoolBrowser-Tests' with: [
      spec
        file: 'CoolBrowser-Tests-JohnLewis.8';
        requires: 'CoolBrowser-Core' ];
    package: 'CoolBrowser-Addons' with: [
      spec
        file: 'CoolBrowser-Addons-JohnLewis.3';
        requires: 'CoolBrowser-Core' ]].

```

In version03: we've added dependency information using the `requires:` directive.

We have also introduced the `file:` message Stéf ▶ *dale could we rename this file: into package:?* ◀, which refers to a specific version of the package. Both `CoolBrowser-Tests` and `CoolBrowser-Addons` require `CoolBrowser-Core`, which must be loaded before they are loaded. Notice that we did not specify the exact version of `CoolBrowser-Core` on which they depend. This can cause problems — but don't worry, we'll address this deficiency soon!.

With this version we are mixing structural information (required packages and repository) with version information (the exact number version). We can expect that, over time, the version information will change frequently

while the structural information will remain more or less the same. To capture this, Metacello introduces the concept of *Baselines*.

1.7 Baselines

A baseline represents the skeleton or architecture of a project in terms of the structural dependencies between packages or projects. A baseline defines the structure of a project using just package names. When the structure changes, the baseline should be updated. In the absence of structural changes, the changes are limited to picking specific versions of the packages in the baseline.

Now, let's continue with our example. First we modify it to use baselines: we create one method for our baseline. Note that the method name and the version tag can take any form. Still, for readability purposes, we use baseline in both of them. It is the argument of the blessing: message that is mandatory and defines a baseline.

```
ConfigurationOfCoolBrowser>>baseline04: spec           "convention"
    <version: '0.4-baseline'>                          "convention"

    spec for: #common do: [
        spec blessing: #baseline.  "mandatory to declare a baseline"
        spec repository: 'http://www.example.com/CoolBrowser'.
        spec
            package: 'CoolBrowser-Core';
            package: 'CoolBrowser-Tests' with: [ spec requires: 'CoolBrowser-Core'];
            package: 'CoolBrowser-Addons' with: [ spec requires: 'CoolBrowser-Core']]
```

The method baseline04: defines the structure of '0.4-baseline', which may be used by several versions. For example, the version '0.4' defined below uses it, as shown in Figure 1.5. The baseline specifies a repository, the packages, and the dependencies between those packages, but it does not specify the specific versions of the packages.

To define a version in terms of a baseline, we use the pragma <version:imports:>, as follows:

```
ConfigurationOfCoolBrowser>>version04: spec
    <version: '0.4' imports: #('0.4-baseline')>

    spec for: #common do: [
        spec
            package: 'CoolBrowser-Core' with: 'CoolBrowser-Core-BobJones.15';
            package: 'CoolBrowser-Tests' with: 'CoolBrowser-Tests-JohnLewis.8';
            package: 'CoolBrowser-Addons' with: 'CoolBrowser-Addons-JohnLewis.3'
    ].
```

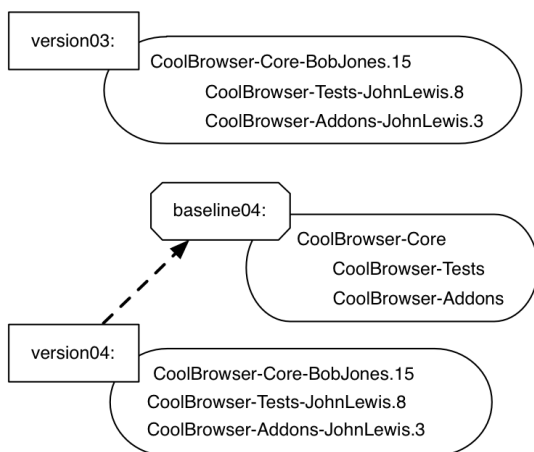


Figure 1.5: Version 0.4 now imports a baseline that expresses the dependencies between packages.

In the method `version04:`, we specify the specific versions of the packages. The pragma `version:imports:` specifies the list of versions that this version (version '0.4') is based upon. Once a specific version is specified, it is loaded in the same way as before, regardless of the fact that it uses a baseline.

```
(ConfigurationOfCoolBrowser project version: '0.4') load.
```

Loading Baselines

Even though version '0.4-baseline' does not contain explicit package version information, you can still load it!

```
(ConfigurationOfCoolBrowser project version: '0.4-baseline') load.
```

When the loader encounters a package without version information, it attempts to load the most recent version of the package from the repository.

Sometimes, especially when several developers are working on a project, it may be useful to load a *baseline* version to access the most recent work of all of the developers. In such a case, the baseline version is really the “bleeding edge” version.

Declaring a new version. Now suppose that we want to create a new version of our project, version '0.5', that has the same structure as version '0.4',

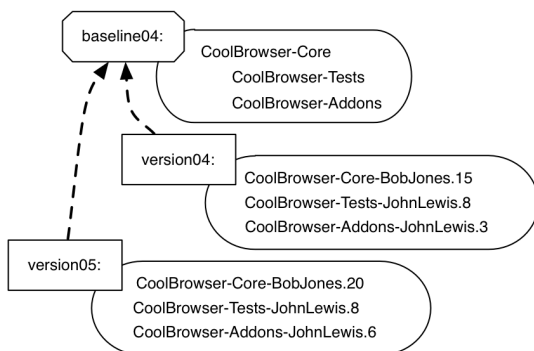


Figure 1.6: A second version (0.5) imports the same baseline as version 0.4.

but contains different versions of the packages. We can capture this intent by importing the same baseline; this relationship is depicted in Figure 1.6.

```

ConfigurationOfCoolBrowser>>version05: spec
  <version: '0.5' imports: #'(0.4-baseline')>

  spec for: #common do: [
    spec
      package: 'CoolBrowser-Core' with: 'CoolBrowser-Core-BobJones.20';
      package: 'CoolBrowser-Tests' with: 'CoolBrowser-Tests-JohnLewis.8';
      package: 'CoolBrowser-Addons' with: 'CoolBrowser-Addons-JohnLewis.6' ].
  
```

Creating a baseline for a big project will often require some time and effort, since it must capture all the dependencies of all the packages, as well as some other things that we will look at later. However, once the baseline is defined, creating new versions of the project is greatly simplified and takes very little time.

1.8 Groups

Suppose that now that the CoolBrowser project grows, a developer writes some tests for CoolBrowser-Addons. These constitute a new package named 'CoolBrowser-AddonsTests', which naturally depends on 'CoolBrowser-Addons' and 'CoolBrowser-Tests', as shown in Figure 1.7.

We may want to load projects with or without tests. In addition, it would be convenient to be able to load all of the tests with a simple expression like:

```
(ConfigurationOfCoolBrowser project version: '0.6') load: 'Tests'.
```

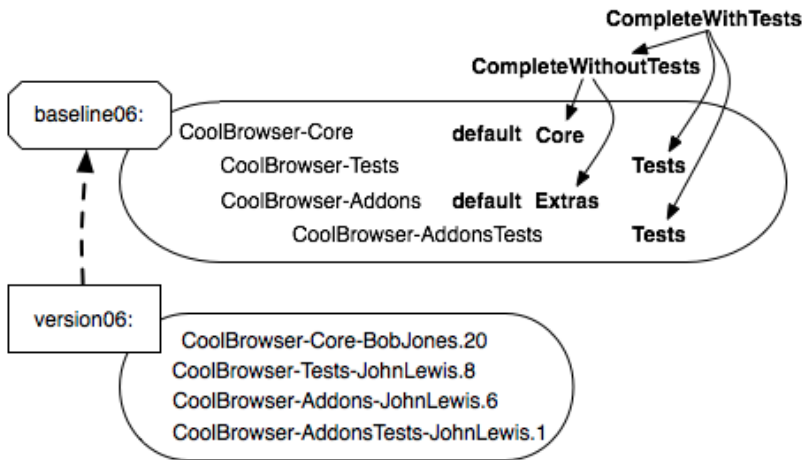


Figure 1.7: A baseline with six groups: default, Core, Extras, Tests, Complete-WithoutTests and CompleteWithTests.

instead of having to explicitly list all of the test packages, like this:

```
(ConfigurationOfCoolBrowser project version: '0.6')
load: #('CoolBrowser-Tests' 'CoolBrowser-AddonsTests').
```

Metacello provides the notion of *group*. A group is a collection of items; each item may be a package, a project, or even another group.

Groups are useful because they let you name sets of items for various purposes. Maybe you want to offer the user the possibility of installing just the core, or the core with add-ons and development features: you can make this easy by defining appropriate groups. Let's go back to our example, and look at how we might define a new baseline, '0.6-baseline' that defines 6 groups, as shown in Figure 1.7. In this example, we create a group called 'Tests' that comprises 'CoolBrowser-Tests' and 'CoolBrowser-AddonsTests'.

To define a group we use the method `group: groupName with: group elements`. The `with:` argument can be a package name, a project, another group, or a collection of those things. Here is the code corresponding to Figure 1.7.

```
ConfigurationOfCoolBrowser>>baseline06: spec
<version: '0.6-baseline'>
spec for: #common do: [
spec blessing: #baseline.
spec repository: 'http://www.example.com/CoolBrowser'.
spec
package: 'CoolBrowser-Core';
package: 'CoolBrowser-Tests' with: [ spec requires: 'CoolBrowser-Core' ];
```

```

package: 'CoolBrowser-Addons' with: [ spec requires: 'CoolBrowser-Core' ] ;
package: 'CoolBrowser-AddonsTests' with: [
    spec requires: #'('CoolBrowser-Addons' 'CoolBrowser-Tests' ) ].
spec
group: 'default' with: #'('CoolBrowser-Core' 'CoolBrowser-Addons');
group: 'Core' with: #'('CoolBrowser-Core');
group: 'Extras' with: #'('CoolBrowser-Addon');
group: 'Tests' with: #'('CoolBrowser-Tests' 'CoolBrowser-AddonsTests');
group: 'CompleteWithoutTests' with: #'('Core' 'Extras');
group: 'CompleteWithTests' with: #'('CompleteWithoutTests' 'Tests')
].

```

We are defining the groups in the baseline version, since a group is a structural component. Note that the default group will be used in the subsequent sections. Here the default group mentions that the two packages 'CoolBrowser-Core' and 'CoolBrowser-Addons' will be loaded when the method load is used.

Using this baseline, we can now define version 0.6 to be the same as version0.5, except for the addition of the new package CoolBrowser-AddonsTests.

```

ConfigurationOfCoolBrowser>>version06: spec
<version: '0.6' imports: #'('0.6-baseline')>

spec for: #common do: [
    spec
        package: 'CoolBrowser-Core' with: 'CoolBrowser-Core-BobJones.20';
        package: 'CoolBrowser-Tests' with: 'CoolBrowser-Tests-JohnLewis.8';
        package: 'CoolBrowser-Addons' with: 'CoolBrowser-Addons-JohnLewis.6' ;
        package: 'CoolBrowser-AddonsTests' with: 'CoolBrowser-AddonsTests-
        JohnLewis.1' ].

```

Examples. Once you have defined a group, you can use its name anywhere you would use the name of a project or package. The load: method takes as parameter the name of a package, a project, a group, or a collection of those items. So all of the following statements are possible:

```

(ConfigurationOfCoolBrowser project version: '0.6') load: 'CoolBrowser-Core'.
"Load a single package"

```

```

(ConfigurationOfCoolBrowser project version: '0.6') load: 'Core'.
"Load a single group"

```

```

(ConfigurationOfCoolBrowser project version: '0.6') load: 'CompleteWithTests'.
"Load a single group"

```

```

(ConfigurationOfCoolBrowser project version: '0.6')

```

```
load: #('CoolBrowser-Core' 'Tests').  
"Loads a package and a group"
```

```
(ConfigurationOfCoolBrowser project version: '0.6')  
load: #('CoolBrowser-Core' 'CoolBrowser-Addons' 'Tests').  
"Loads two packages and a group"
```

```
(ConfigurationOfCoolBrowser project version: '0.6')  
load: #('CoolBrowser-Core' 'CoolBrowser-Tests').  
"Loads two packages"
```

```
(ConfigurationOfCoolBrowser project version: '0.6') load: #('Core' 'Tests').  
"Loads two groups"
```

The groups 'default' and 'ALL'. The 'default' group is special one. The load message loads the members of the 'default' groups while the group 'ALL' will load all the packages. Moreover, by default, 'default' loads 'ALL'!

```
(ConfigurationOfCoolBrowser project version: '0.6') load.
```

loads just 'CoolBrowser-Core' and 'CoolBrowser-Addons'.

In the presence of a 'default' group, how do you load all the packages of a project? You use the predefined group 'ALL', as shown below:

```
(ConfigurationOfCoolBrowser project version: '0.6') load: 'ALL'.
```

1.9 Dependencies Between Projects

In the same way that a package can depend on other packages, a project can depend on other projects. For example, Pier, which is a content management system that uses meta-description, depends on Magritte and Seaside. A project can depend on the entirety of one or more other projects, on a group of packages from another project, or on just one or two packages from another project.

How we describe project dependencies depends on whether or not the other projects are described using Metacello.

Depending on a project *without* a Metacello description

Suppose that package A from Project X depends on package B from project Y, and that project Y has not been described using Metacello (this might be because there is only one package in project Y). In this case we can describe

the dependency as follows: **Stéf** ► *we should use the same example A and B sucks*◄

Mariano ► +1◄

"In a baseline method"

spec

package: 'PackageA' with: [spec requires: #'('PackageB')];

package: 'PackageB' with: [spec

repository: 'http://www.squeaksource.com/ProjectB'].

"In the version method"

package: 'PackageB' with: 'PackageB-JuanCarlos.80'.

This works, up to a point. The shortcoming of this approach is that because project B is not described by a Metacello configuration the dependencies of B are not managed. That is, any dependencies of package B will not be loaded. So, our recommendation is that in this case, you take the time to create a configuration for the project B.

Depending on a project *with* a Metacello configuration

Now let us look at the case where the projects on which we depend are described using Metacello. Let's introduce a new project called CoolToolSet that uses the packages from the CoolBrowser project. Its configuration class is called ConfigurationOfCoolToolSet. Suppose that there are two packages in CoolToolSet called CoolToolSet-Core and CoolToolSet-Tests. Of course, these packages depend on packages from CoolBrowser.

Version '0.1' of CoolToolSet is just a normal version that imports a baseline:

```
ConfigurationOfCoolToolSet>>version01: spec
```

```
<version: '0.1' imports: #'('0.1-baseline')>
```

```
spec for: #common do: [
```

```
spec
```

```
package: 'CoolToolSet-Core' with: 'CoolToolSet-Core-AlanJay.1';
```

```
package: 'CoolToolSet-Tests' with: 'CoolToolSet-Tests-AlanJay.1'.]
```

Here is the baseline, if the project you depend on followed the default convention (*i.e.*, class ConfigurationOfCoolBrowser in package ConfigurationOfCoolBrowser). By default you just need to specify the version (using versionString:) you want to load and the project repository (using repository:).

```
ConfigurationOfCoolToolSet >>baseline01: spec
```

```
<version: '0.1-baseline'>
```

```
spec for: #common do: [
```

```
spec repository: 'http://www.example.com/CoolToolSet'.
```

```
spec project: 'CoolBrowser ALL' with: [
```

```
spec repository: 'http://www.example.com/CoolBrowser' ]
spec
  package: 'CoolToolSet-Core' with: [ spec requires: 'CoolBrowser ALL' ];
  package: 'CoolToolSet-Tests' with: [ spec requires: 'CoolToolSet-Core' ]].
```

We've named the project reference 'CoolBrowser ALL' and in the specification for the 'CoolToolSet-Core' package, we've specified that 'CoolBrowser ALL' is required. The name of the project reference is arbitrary, you can select the name you want, although it is recommended to put a name that makes sense to that project reference. As explained later, the message `project:with:` allows one to specify the exact version of the project you want to load.

Now if the project you depend on does not follow the default convention you will have to provide more information to identify the configuration. Let's also assume that the class `ConfigurationOfCoolBrowser` is stored in a Monticello package called `CoolBrowser-Metacello` instead of the recommended `ConfigurationOfCoolBrowser`. This will help us to understand the role of each parameter.

Now we can now load `CoolToolSet` like this:

```
(ConfigurationOfCoolToolSet project version: '0.1') load.
```

For unconventional projects

Now if the project you depend on does not follow the default convention you will have to provide more information to identify the configuration. Let's also assume that the class `ConfigurationOfCoolBrowser` is stored in a Monticello package called `CoolBrowser-Metacello` instead of the recommended `ConfigurationOfCoolBrowser`. This will help us to understand the role of each parameter.

```
ConfigurationOfCoolToolSet >>baseline01: spec
  <version: '0.1-baseline'>
  spec for: #common do: [
    spec repository: 'http://www.example.com/CoolToolSet'.
    spec project: 'CoolBrowser ALL' with: [
      spec
        className: 'ConfigurationOfCoolBrowser';
        loads: #('ALL' );
        file: 'CoolBrowser-Metacello';
        repository: 'http://www.example.com/CoolBrowser' ].
    spec
      package: 'CoolToolSet-Core' with: [ spec requires: 'CoolBrowser ALL' ];
      package: 'CoolToolSet-Tests' with: [ spec requires: 'CoolToolSet-Core' ]].
```

We've named the project reference 'CoolBrowser ALL' and in the specification for the 'CoolToolSet-Core' package, we've specified that 'CoolBrowser ALL' is required. The name of the project reference is arbitrary, you can select the name you want, although it is recommended to put a name that makes sense to that project reference. What we did here in baseline0.1 was to create a *project reference* for the CoolBrowser project (see Figure 1.8).

- The message `className`: specifies the name of the class that contains the project metadata; in this case 'ConfigurationOfCoolBrowser'.
- The messages `file`: and `repository`: give Metacello the information that it might need to search for and load class 'ConfigurationOfCoolBrowser', if it is not present in the image. The argument of `file`: is the name of the Monticello package that contains the metadata class, and the argument of `repository`: is the URL of the Monticello repository that contains that package. If the Monticello repository is protected, then you should use the message: `repository:username:password:` instead. In this example, it is mandatory to specify the package name with `file`: because 'ConfigurationOfCoolBrowser' is not placed according to the convention (it is in a package called 'CoolBrowser-Metacello' rather than 'ConfigurationOfCoolBrowser'). If you follow the convention, then it is not necessary to specify it using `file`:.
- The message `loads`: specify which packages or groups to load. The parameter of `loads`: can be the same as `load`, *i.e.*, name of a package, or the name of a group, or a collection of these things. Notice that `loads`: is optional, you only need it if you want to load something different from the default.

Andrew ► So I could have said `loads: 'ALL'`; and that would have meant the same thing? If so, why didn't I? And why is this message `loads`: rather than `load`: like the others? ◀

Now we can now load CoolToolSet like this:

```
(ConfigurationOfCoolToolSet project version: '0.1') load.
```

Multiple Entries

Using 'ALL' will cause the entire CoolBrowser project to be loaded before 'CoolToolSet-Core'. If we wanted to specify dependencies on CoolBrowser's test package separately from those on the core package, we might define this baseline:

```
ConfigurationOfCoolToolSet >>baseline02: spec
  <version: '0.2-baseline'>
  spec for: #common do: [
```



Figure 1.8: Dependencies between configurations.

```
spec blessing: #baseline.
spec repository: 'http://www.example.com/CoolToolSet'.
spec project: 'CoolBrowser default' with: [
    spec
        className: 'ConfigurationOfCoolBrowser';
        loads: #('default' ); ``this is optional"
        file: 'CoolBrowser-Metacello';
        repository: 'http://www.example.com/CoolBrowser' ].
    project: 'CoolBrowser Tests' with: [
        spec
            className: 'ConfigurationOfCoolBrowser';
            loads: #('Tests' );
            file: 'CoolBrowser-Metacello';
            repository: 'http://www.example.com/CoolBrowser' ].
    spec
        package: 'CoolToolSet-Core' with: [ spec requires: 'CoolBrowser default' ];
        package: 'CoolToolSet-Tests' with: [
            spec requires: #('CoolToolSet-Core' 'CoolBrowser Tests' ) ].]
```

This baseline creates *two* project references: the reference named 'CoolBrowser default' loads the 'default' group and the reference named 'CoolBrowser Tests' loads the 'Tests' group of the configuration of CoolBrowser. We declared that 'CoolToolSet-Core' require 'CoolBrowser default' and 'CoolToolSet-Tests' require 'CoolToolSet-Core' and 'CoolBrowser Tests'.

Now it is possible to load just the core packages:

```
(ConfigurationOfCoolToolSet project version: '0.2') load: 'CoolToolSet-Core'.
```

or the core including tests:

```
(ConfigurationOfCoolToolSet project version: '0.2') load: 'CoolToolSet-Tests'.
```

As we did for internal dependencies, baseline02, and also in baseline01, do not specify the version of the project they depend on. Instead, we do this in the version method using the message :

```
ConfigurationOfCoolToolSet>>version02: spec
  <version: '0.2' imports: #'(0.2-baseline)'>
  spec for: #common do: [
    spec blessing: #beta.
    spec
      package: 'CoolToolSet-Core' with: 'CoolToolSet-Core-AlanJay.1';
      package: 'CoolToolSet-Tests' with: 'CoolToolSet-Tests-AlanJay.1';
      project: 'CoolBrowser default' with: '1.3';
      project: 'CoolBrowser Tests' with: '1.3'].
```

Loading specific packages

Now even if you do not specified references in the baseline, in the version you can also specify which packages of the project you depend on to load. Here in the ConfigurationOfSoup, we say that we want to load in the version 1.2 the packages 'XML-Parser' and 'XML-Tests-Parser'.

```
ConfigurationOfSoup>>version10: spec
  <version: '1.0' imports: #'(1.0-baseline)'>

  spec for: #pharo do: [
    spec
      project: 'XMLSupport'
      with: [spec
        loads: #'(XML-Parser' 'XML-Tests-Parser)';
        versionString: '1.2.0'].

    spec
      package: 'Soup-Core' with: 'Soup-Core-sd.11';
      package: 'Soup-Tests-Core' with: 'Soup-Tests-Core-sd.3';
      package: 'Soup-Help' with: 'Soup-Help-StephaneDucasse.2' ].
```

What you can also do is to use the loads: message in the project reference to specify which packages of the project you want to load. Such solution is nice because you factor the information in the project reference and you do not have to duplicate it in all the versions.

```
ConfigurationOfCoolToolSet>>version10: spec
  <version: '1.0' imports: #'(1.0-baseline)'>

  spec for: #pharo do: [
    spec project: 'XMLSupport' with: [
      spec className: 'ConfigurationOfXMLSupport';
```

```

versionString: #stable;
loads: #('XML-Parser' 'XML-Tests-Parser')
repository: 'http://www.squeaksource.com/XMLSupport' ].

spec
  package: 'Soup-Core' with: 'Soup-Core-sd.11';
  package: 'Soup-Tests-Core' with: 'Soup-Tests-Core-sd.3';
  package: 'Soup-Help' with: 'Soup-Help-StephaneDucasse.2' ].

```

Version in Baselines. Even if this is not recommended. Nothing prevents you from specifying versions from baselines. The same happens with project references. So, in addition to messages like `file:`, `className:`, `repository:`, etc., there is a message called `versionString:` which lets you specify the version of the project directly in the project reference. Example:

```

ConfigurationOfCoolToolSet >>baseline011: spec
  <version: '0.1.1-baseline'>
  spec for: #common do: [
    spec repository: 'http://www.example.com/CoolToolSet'.
    spec project: 'CoolBrowser ALL' with: [
      spec
        className: 'ConfigurationOfCoolBrowser';
        loads: #('ALL');
        versionString: '0.6' ;
        file: 'CoolBrowser-Metacello';
        repository: 'http://www.example.com/CoolBrowser' ].
    spec
      package: 'CoolToolSet-Core' with: [ spec requires: 'CoolBrowser ALL' ];
      package: 'CoolToolSet-Tests' with: [ spec requires: 'CoolToolSet-Core' ].

```

If we don't define a version for the references 'CoolBrowser default' and 'CoolBrowser Tests' in the version method, then the version specified in the baseline (using `versionString:`) is used. If there is no version specified in the baseline method, then Metacello loads the most recent version of the project.

In general this is not a good idea to add version information in a baseline. Now you can override the information into the version. Imagine that Kozen depends on Citezen. Citezen has two versions one stable and one under development. Now Kozen has a stable version and you want to build the development version and turn both development versions into stable ones. If you specify the baseline as follow.

```

ConfigurationOfKozen>>baseline103: spec
  <version: '1.3-baseline'>

  spec for: #pharo do: [
    spec blessing: #baseline.

```

```
spec repository: 'http://ss3.gemstone.com/ss/Kozen'.
spec package: 'Kozen'.
spec package: 'KoCizen' with: [spec requires: 'Citezen'].

spec group: 'default' with: #('Kozen').
spec group: 'full' with: #('Kozen' 'KoCizen').

spec project: 'Citezen' with: [
  spec className: 'ConfigurationOfCitezen';
  versionString: #stable;
  loads: #('NoWeb' );
  repository: 'http://ss3.gemstone.com/ss/citezen'.  ]].
```

In the version you

```
ConfigurationOfKozen>>version12: spec
<version: '1.2' imports: #('1.3-baseline')>

spec for: #pharo' do: [
  spec description: 'Honoring hooks (defaultTitle -> title)'.
  spec blessing: #release.
  spec author: 'StephaneDucasse'.
  spec timestamp: '12/03/2012 21:45'.
  spec project: 'Citezen' with: #development'.
  spec
    package: 'Kozen' with: 'Kozen-StephaneDucasse.32';
    package: 'KoCizen' with: 'KoCizen-StephaneDucasse.6'.]
```

Reusing information. As you can see, in baseline02: information is duplicated in the two project references. To remove that duplication, we can use the project.copyFrom:with: method. For example:

```
ConfigurationOfCoolToolSet >>baseline02: spec
<version: '0.2-baseline'>
spec for: #common do: [
  spec blessing: #baseline.
  spec repository: 'http://www.example.com/CoolToolSet'.
  spec project: 'CoolBrowser default' with: [
    spec
      className: 'ConfigurationOfCoolBrowser';
      loads: #('default');
      repository: 'http://www.example.com/CoolBrowser';
      file: 'CoolBrowser-Metacello']
    project: 'CoolBrowser Tests'
    copyFrom: 'CoolBrowser default'
    with: [ spec loads: #('Tests').].
  spec
    package: 'CoolToolSet-Core' with: [ spec requires: 'CoolBrowser default' ];
```

```
package: 'CoolToolSet-Tests' with: [
  spec requires: #('CoolToolSet-Core' 'CoolBrowser Tests') ].].
```

1.10 Executing code before and after installation

Occasionally, you may find that you need to execute some code either before or after a package or project is loaded. For example, if you are installing a System Browser it would be a good idea to register it as default after it is loaded. Or maybe you want to open some workspaces after the installation.

Metacello provides this feature by means of the messages `preLoadDolt:` and `postLoadDolt:`. The arguments to these messages are selectors of methods defined on the configuration class as shown below. For the moment, these pre- and post-scripts can be defined for a single package or for an entire project.

Continuing with our example:


```
ConfigurationOfCoolBrowser>>version08: spec
<version: '0.8' imports: #('0.7-baseline')>

spec for: #common do: [
  spec
    package: 'CoolBrowser-Core' with: [
      spec
        file: 'CoolBrowser-Core-BobJones.20';
        preLoadDolt: #preloadForCore;
        postLoadDolt: #postloadForCore:package: ];
    ....
    package: 'CoolBrowser-AddonsTests' with: 'CoolBrowser-AddonsTests-
    JohnLewis.1' ].
```

```
ConfigurationOfCoolBrowser>>preloadForCore
Transcript show: 'This is the preload script. Sorry I had no better idea'.
```

```
ConfigurationOfCoolBrowser>>postloadForCore: loader package: packageSpec
Transcript cr;
  show: '#postloadForCore executed, Loader: ', loader printString,
  ' spec: ', packageSpec printString.

Smalltalk at: #SystemBrowser ifPresent: [:cl | cl default: (Smalltalk classNamed:
#CoolBrowser)].
```

As you can notice there, both methods, `preLoadDolt:` and `postLoadDolt:` receive a selector that will be performed before or after the load. You can also note that the method `postloadForCore:package:` takes two parameters. The pre/post load methods may take 0, 1 or 2 arguments. The *loader*  *should*

explain that ◀ is the first optional argument and the loaded packageSpec is the second optional argument. Depending on your needs you can choose which of those arguments do you want.

These pre and post load scripts can be used not only in version methods but also in baselines. If a script depends on a version, then you can put it there. If it is likely not to change among different versions, you can put it in the baseline method exactly in the same way.

As we said before, these pre and post it can be at package level, but also at project level. For example, we can have the following configuration:

```
ConfigurationOfCoolBrowser>>version08: spec
  <version: '0.8' imports: #('0.7-baseline')>

  spec for: #common do: [
    spec blessing: #release.

    spec preLoadDolt: #preLoadForCoolBrowser.
    spec postLoadDolt: #postLoadForCoolBrowser.

    spec
      package: 'CoolBrowser-Core' with: [
        spec
          file: 'CoolBrowser-Core-BobJones.20';
          preLoadDolt: #preloadForCore;
          postLoadDolt: #postloadForCore;package: ];
        package: 'CoolBrowser-Tests' with: 'CoolBrowser-Tests-JohnLewis.8';
        package: 'CoolBrowser-Addons' with: 'CoolBrowser-Addons-JohnLewis.6';
      ];
      package: 'CoolBrowser-AddonsTests' with: 'CoolBrowser-AddonsTests-JohnLewis.1'].
```

In this example, we added pre and post load scripts at project level. Again, the selectors can receive 0, 1 or 2 arguments.

1.11 Platform specific package

Suppose that we want to have different packages loaded depending on the platform the configuration is loaded in. In the context of our example our Cool Browser we can have a package called CoolBrowser-Platform. There we can define abstract classes, APIs, etc. And then, we can have the following packages: CoolBrowser-PlatformPharo, CoolBrowser-PlatformGemstone, etc.

Metacello automatically loads the package of the used platform. But to do that, we need to give Metacello platform specific information using the method `for:do:` as shown in the following example.

```

ConfigurationOfCoolBrowser>>version09: spec
<version: '0.9' imports: #'(0.9-baseline)'>

spec for: #common do: [
    ...
    spec
    ...
    package: 'CoolBrowser-AddonsTests' with: 'CoolBrowser-AddonsTests-
JohnLewis.1' ].

spec for: #gemstone do: [
    spec package: 'CoolBrowser-Platform' with: 'CoolBrowser-PlatformGemstone-
BobJones.4' ].
spec for: #pharo do: [
    spec package: 'CoolBrowser-Platform' with: 'CoolBrowser-PlatformPharo-
JohnLewis.7' ].

```

You see that the version can handle different platform.

```

ConfigurationOfCoolBrowser>>baseline09: spec
<version: '0.9-baseline'>

spec for: #common do: [
    spec blessing: #baseline.
    spec repository: 'http://www.example.com/CoolBrowser'.

    spec
        package: 'CoolBrowser-Core';
        package: 'CoolBrowser-Tests' with: [ spec requires: 'CoolBrowser-Core' ];
        package: 'CoolBrowser-Addons' with: [ spec requires: 'CoolBrowser-Core'
];
    package: 'CoolBrowser-AddonsTests' with: [
        spec requires: #'('CoolBrowser-Addons' 'CoolBrowser-Tests' ) ].
    spec
        group: 'default' with: #'('CoolBrowser-Core' 'CoolBrowser-Addons' );
        group: 'Core' with: #'('CoolBrowser-Core' 'CoolBrowser-Platform' );
        group: 'Extras' with: #'('CoolBrowser-Addon');
        group: 'Tests' with: #'('CoolBrowser-Tests' 'CoolBrowser-AddonsTests' );
        group: 'CompleteWithoutTests' with: #'('Core', 'Extras' );
        group: 'CompleteWithTests' with: #'('CompleteWithoutTests', 'Tests' )].

spec for: #gemstone do: [
    spec package: 'CoolBrowser-Platform' with: 'CoolBrowser-PlatformGemstone'
.].
spec for: #pharo do: [
    spec package: 'CoolBrowser-Platform' with: 'CoolBrowser-PlatformPharo' ].

```

Notice that we add the package CoolBrowser-Platform in the Core group. As

you can see, we can manage this package as any other and in a uniform way. Thus, we have a lot of flexibility. At runtime, when you load CoolBrowser, Metacello automatically detects in which dialect the load is happening and loads the specific package for that dialect. The `for:do:` is not only for dialects but also for specific versions of those dialects. For example, we can have:

```
ConfigurationOfCoolBrowser>>baseline09: spec
  <version: '0.9-baseline'>

  ...

  spec for: #gemstone do: [
    spec package: 'CoolBrowser-Platform' with: 'CoolBrowser-PlatformGemstone'
  ].
  spec for: '#pharo1.3.x' do: [
    spec package: 'CoolBrowser-Platform' with: 'CoolBrowser-PlatformPharo13'].
  spec for: '#pharo1.2.x' do: [
    spec package: 'CoolBrowser-Platform' with: 'CoolBrowser-PlatformPharo12'].
```

Finally, note that the method `for:do:` is not only used to specify a platform specific package, but also for anything that has to do with different dialects. You can put whatever you want from the configuration inside that block. So, for example, you can define, change and customize groups, packages, repositories, etc, for each dialect dialect. For example, you can do this:

```
ConfigurationOfCoolBrowser>>baseline010: spec
  <version: '0.10-baseline'>

  spec for: #common do: [
    spec blessing: #baseline.].

  spec for: #pharo do: [
    spec repository: 'http://www.pharo.com/CoolBrowser'.

    spec
      ...
    spec
      group: 'default' with: #('CoolBrowser-Core' 'CoolBrowser-Addons' );
      group: 'Core' with: #('CoolBrowser-Core' 'CoolBrowser-Platform' );
      group: 'Extras' with: #('CoolBrowser-Addon');
      group: 'Tests' with: #('CoolBrowser-Tests' 'CoolBrowser-AddonsTests' );
      group: 'CompleteWithoutTests' with: #('Core', 'Extras' );
      group: 'CompleteWithTests' with: #('CompleteWithoutTests', 'Tests' )].

  spec for: #gemstone do: [
    spec repository: 'http://www.gemstone.com/CoolBrowser'.

    spec
      package: 'CoolBrowser-Core';
```

```
package: 'CoolBrowser-Tests' with: [ spec requires: 'CoolBrowser-Core' ];
spec
group: 'default' with: #('CoolBrowser-Core' 'CoolBrowser-Addons' );
group: 'Core' with: #('CoolBrowser-Core' 'CoolBrowser-Platform' )].
```

In this example, for Pharo we use a different repository than for Gemstone. However, this is not mandatory, since both can have the same repository and differ in other things like versions, post and pre code executions, dependencies, etc.

In addition, the addons and tests are not available for Gemstone, and thus, those packages and groups are not included. So, as you can see, all what we have been doing inside the `for: #common: do:` can be done inside another `for:do:` for a specific dialect.

1.12 Symbolic Versions

In any large evolving application, it is difficult to know which version of a configuration to use with a specific version. `ConfigurationOfOmniBrowser` provides a good example of the problem: version 1.1.3 is used in the Pharo1.0 one-click image, version 1.1.3 cannot be loaded into Pharo1.2, version 1.1.5 is for Pharo1.1, version 1.2.3 is for Pharo1.2, and it cannot load in Pharo1.0, etc.

Metacello introduces *symbolic versions* to describe versions in terms of existing literal versions (like 1.1.3, 1.1.5, and 1.2.3). Symbolic versions are specified using the `symbolicVersion:` pragma. Here we defined the stable versions for `OmniBrowser` for each version of Pharo.

```
OmniBrowser>>stable: spec
<symbolicVersion: #stable>
spec for: #'pharo1.0.x' version: '1.1.3'.
spec for: #'pharo1.1.x' version: '1.1.5'.
spec for: #'pharo1.2.x' version: '1.2.3'.
```

Symbolic versions can be used anywhere that a literal version can be used. From a load expressions such as

```
(ConfigurationOfOmniBrowser project version: #stable) load
```

to a project reference in a baseline version:

```
baseline10: spec
<version: '1.0-baseline'>
spec for: #squeakCommon do: [
spec blessing: #baseline.
spec repository: 'http://seaside.gemstone.com/ss/GLASSClient'.
```

```
spec
  project: 'OmniBrowser' with: [
    spec
      className: 'OmniBrowser';
      versionString: #stable;
      repository: 'http://www.squeaksource.com/MetacelloRepository' ].
spec
  package: 'OB-SUnitGUI' with: [
    spec requires: #('OmniBrowser') ];
  package: 'GemTools-Client' with: [
    spec requires: #('OB-SUnitGUI') ];
  package: 'GemTools-Platform' with: [
    spec requires: #('GemTools-Client') ]].
```

Note that the `#stable` here override the bleeding edge loading behavior that you would get if you would be (fool enough) to load a baseline (remember loading a baseline loads bleeding edge versions). Here we make sure that the stable version of OmniBrowser for your platform will be loaded (and not the latest one). The next section is about the different symbolic versions.

Standard Symbolic Versions

A couple of standard symbolic versions are defined:

stable. A symbolic version that specifies the stable literal version for a particular platform and version of such a platform. The stable version is the version that should be used for loading. With the exception of the `bleedingEdge` version (which has a pre-defined default defined), you will need to edit your configuration to add the stable or development version information. *I want a certified version for the platform.*

development. A symbolic version that specifies the literal version to use under development (i.e., whose blessing is development). Typically a development version is used by developers for managing pre-release activities as the project transitions from `bleedingEdge` to `stable`. There are a number of `MetacelloToolBox` methods that take advantage of the development symbolic version. *I want a certified version for the platform but in development mode.*

Mariano ► *did we speak already about MetacelloToolBox?* ◀

bleedingEdge. A symbolic version that specifies the latest mcz files and project versions. By default the `bleedingEdge` symbolic version is defined as the latest *baseline* version available. The default specification for `bleedingEdge` is defined for all projects. The `bleedingEdge` version is primarily for developers who know what they are doing. There are no

guarantees that the bleedingEdge version will even load, let alone function correctly. *I want the latest published file.*

Stéf ► I have the impression that this is not clear. it would be good to have an example from something stable and may from moose? ◀ **Stéf** ► How do I say that default is load stable? should I say it? same question for the other ones like bleedingEdge ◀ **Mariano** ► default for what EXACTLY? The only 'default' we talked so far is just to now which packages to load by default, that is, when using load. But that is to an specific version. But this (bleedingEdge, stable, etc) is about which version to load. There is no default. I mean...what do you want to do it exactly? Lets say I have ConfigurationOfProject project version: xxx. Where is the default? what do you want to do? ConfigurationOfProject load ?? I dont understand.... ◀

When specifying a symbolic version with a symbolicVersion: pragma it is legal to use another symbolic version like the following definition for the symbolic version stable:

```
stable: spec
  <symbolicVersion: #stable>

  spec for: #gemstone version: '1.5'.
  spec for: #'squeak' version: '1.4'.
  spec for: #'pharo1.0.x' version: '1.5'.
  spec for: #'pharo1.1.x' version: '1.5'.
  spec for: #'pharo1.2.x' version: #development.
```

Or to use the special symbolic version notDefined: as in the following definition of the symbolic version development:

```
development: spec
  <symbolicVersion: #development>

  spec for: #common version: #notDefined.
  spec for: #'pharo1.1.x' version: '1.6'.
  spec for: #'pharo1.2.x' version: '1.6'.
```

Here this indicates that there is no version for the common tag. Using a symbolic version that resolves to notDefined will result in a MetacelloSymbolicVersionNotDefinedError being signaled.

Project Blessing and Loading

Packages or projects pass through several stages or steps during the software development process or life cycle such as for example, development, alpha, beta, release. Sometimes we want to refer also to the state of a project.

Stéf ► Not sure if the following is up to date ◀ **Mariano** ► from what I could see, it is. ◀ Blessings are taken into account by the load logic. The result of the following expression:

```
ConfigurationOfCoolBrowser project latestVersion.
```

is not always the last version. This is because `latestVersion` answers the latest version whose blessing is *not* `#development`, `#broken`, or `#blessing`. To find the latest `#development` version for example, you should execute this expression:

```
ConfigurationOfCoolBrowser project latestVersion: #development.
```

Nevertheless, you can get the very last version independently of blessing using the `lastVersion` method as illustrated below

```
ConfigurationOfCoolBrowser project lastVersion.
```

In general, the `#development` blessing should be used for any version that is unstable. Once a version has stabilized, a different blessing should be applied.

The following expression will load the latest version of all of the packages for the latest `#baseline` version:

```
(ConfigurationOfCoolBrowser project latestVersion: #baseline) load.
```

Since the latest `#baseline` version should reflect the most up-to-date project structure, executing the previous expression loads the absolute bleeding edge version of the project.

Hints.

Some patterns emerge when working with Metacello. Here is a good one: Create a baseline version and use the `#stable` version for all of the projects in the baseline. In the literal version, use the explicit version, so that you get an explicit repeatable specification for a set of projects that were known to work together.

Here is an example, the pharo 1.2.2-baseline would include specs that look like this:

```
spec
  project: 'OB Dev' with: [
    spec
      className: 'ConfigurationOfOmniBrowser';
      versionString: #stable;
      ...];
  project: 'ScriptManager' with: [
    spec
      className: 'ConfigurationOfScriptManager';
      versionString: #stable;
      ...];
```

```
project: 'Shout' with: [
spec
  className: 'ConfigurationOfShout';
  versionString: #stable;
  ...];
....].
```

Loading Pharo 1.2.2-baseline would cause the `#stable` version for each of those projects to be loaded ... but remember over time the `#stable` version will change and incompatibilities between packages can creep in. By using `#stable` versions you will be in better shape than using `#bleedingEdge` because the `#stable` version is known to work.

Pharo 1.2.2 (literal version) will have corresponding specs that look like this:

```
spec
project: 'OB Dev' with: '1.2.4';
project: 'ScriptManager' with: '1.2';
project: 'Shout' with: '1.2.2';
....].
```

So that you have driven a stake into the ground stating that these versions are known to work together (have passed tests as a unit). 5 years in the future, you will be able to load Pharo 1.2.2 and get exactly the same packages every time, whereas the `#stable` versions may have drifted over time.

If you are just bringing up a PharoCore1.2 image and would like to load the Pharo dev code, you should load the `#stable` version of Pharo (which may be 1.2.2 today and 1.2.3 tomorrow). If you want to duplicate the environment that someone is working in, you will ask them for the version of Pharo and load that explicit version to reproduce the bug or whatever.

How to Deal with Package Structure Changes?

Imagine that you want to develop an application on both Pharo13 and Pharo14, and that your application has only a package for one version: either because you changed your application or because the package was integrated into the base system.

The solution is to define the dependencies and use the symbolic tag as a marker, as follows:

```
spec for: #'pharo' do: [
spec package: 'that depends upon zinc' with: [
  "the common required packages for your package"
```



```
].  
  
spec for: #'pharo1.3.x' do: [  
  spec project: 'Zinc' with: [  
    spec  
      className: 'ConfigurationOfZinc';  
      versionString: #'stable';  
      repository: 'http://www.squeaksource.com/MetacelloRepository' ].  
  spec package: 'that depends upon zinc' with: [  
    spec requires: #('Zinc') ].  
].
```

If you use the stable version in your baseline there is no need to anything special in your version specification.

1.13 Load types

Metacello lets you specify the way packages are loaded through its “load types”. For the time of this writing, there are only two possible load types: *atomic* and *linear*.

Atomic loading is used where packages have been partitioned in such a way that they can’t be loaded individually. The definitions from each package are munged together into one giant load by the Monticello package loader. Class side initialize methods and pre/post code execution are performed for the whole set of packages, not individually.

If you use a linear load, then each package is loaded in order. Class side initialize methods and pre/post code execution are performed just before or after loading that specific package.

It is important to notice that managing dependences does not imply the order packages will be loaded. That a package *A* depends on package *B* doesn’t mean that *B* will be loaded before *A*. It just guarantees that if you want to load *A*, then *B* will be loaded too.

A problem with this happens also with methods override. If a package overrides a method from another package, and the order is not preserved, then this can be a problem because we are not sure the order they will load, and thus, we cannot be sure which version of the method will be finally loaded.

When using atomic loading the package order is lost and we have the mentioned problems. However, if we use the linear mode, then each package is loaded in order. Moreover, the methods override should be preserved too.

A possible problem with linear mode is the following: suppose project *A* depends on other two projects *B* and *C*. *B* depends on the project *D* version

1.1 and C depends on project D version 1.2 (the same project but another version). First question, which D version does A have at the end? By default (you can change this using the method operator: in the project method), Metacello will finally load version 1.2, *i.e.*, the latest one.

However, and here is the relation with load types, in atomic loading *only* 1.2 is loaded. In linear loading, *both* versions may (depending on the dependency order) be loaded, although 1.2 will be finally loaded. But this means that 1.1 may be loaded first and then 1.2. Sometimes this can be a problem because an older version of a package or project may not even load in the Pharo image we are using.

For all the mentioned reasons, the default mode is linear. Users should use atomic loading in particular cases and when they are completely sure.

Finally, if you want to explicitly set a load type, you have to do it in the project method. Example:

```
ConfigurationOfCoolToolSet >>project
```

```
↑ project ifNil: [ | constructor |
  "Bootstrap Metacello if it is not already loaded"
  self class ensureMetacello.
  "Construct Metacello project"
  constructor := (Smalltalk at: #MetacelloVersionConstructor) on: self.
  project := constructor project.
  project loadType: #linear. ""or #atomic""
  project ]
```

1.14 Conditional loading

When loading a project, usually the user wants to decide whether to load or not certain packages depending on a specific condition, for example, the existence of certain other packages in the image. Suppose you want to load Seaside in your image. Seaside has a tool that depends on OmniBrowser and it is used for managing instances of web servers. What can be done with this little tool can also be done by code. If you want to load such tool you need OmniBrowser. However, other users may not need such package. An alternative could be to provide different groups, one that includes such package and one that does not. The problem is that the final user should be aware of this and load different groups in different situations. With conditional loading you can, for example, load that Seaside tool only if OmniBrowser is present in the image. This will be done automatically by Metacello and there is no need to explicitly load a particular group.

Suppose that our CoolToolSet starts to provide much more features. We first split the core in two packages: 'CoolToolSet-Core' and 'CoolToolSet-CB'.

CoolBrowser can be present in one image but not in another one. We want to load the package 'CoolToolSet-CB' by default only and if CoolBrowser is present.

The mentioned conditionals are achieved in Metacello by using the *project attributes* we saw in the previous section. They are defined in the project method. **Stéf** ► *to me it looks really bad and I'm sure that we want to document that* ◀ **Mariano** ► *We should not put things under the carpet. The only way to improve is to understand the current limitations and think how to make them better.* ◀ Example:

```
ConfigurationOfCoolBrowser >>project
| |
↑ project ifNil: [ | constructor |
  "Bootstrap Metacello if it is not already loaded"
  self class ensureMetacello.
  "Construct Metacello project"
  constructor := (Smalltalk at: #MetacelloVersionConstructor) on: self.
  project := constructor project.
  projectAttributes := ((Smalltalk at: #CBNode ifAbsent: []) == nil
    ifTrue: [ #( #'CBNotPresent' ) ]
    ifFalse: [ #( #'CBPresent' ) ]).
  project projectAttributes: projectAttributes.
  project loadType: #linear.
  project ]
```

As you can see in the code, we check if CBNode class (a class from CoolBrowser) is present and depending on that we set an specific project attribute. This is flexible enough to let you define your own conditions and set the amount of project attributes you wish (you can define an array of attributes). Now the question is how to use these project attributes. In the following baseline we see an example:

```
ConfigurationOfCoolToolSet >>baseline02: spec
<version: '0.2-baseline'>

spec for: #common do: [
  spec blessing: #baseline.
  spec repository: 'http://www.example.com/CoolToolSet'.
  spec project: 'CoolBrowser default' with: [
    spec
      className: 'ConfigurationOfCoolBrowser';
      versionString: '1.0';
      loads: #( 'default' );
      file: 'CoolBrowser-Metacello';
      repository: 'http://www.example.com/CoolBrowser' ];
    project: 'CoolBrowser Tests'
      copyFrom: 'CoolBrowser default'
      with: [ spec loads: #( 'Tests' ) ].
  ]
spec
```

```

package: 'CoolToolSet-Core';
package: 'CoolToolSet-Tests' with: [
    spec requires: #('CoolToolSet-Core') ];
package: 'CoolToolSet-CB';

spec for: #CBPresent do: [
    spec
        group: 'default' with: #('CoolToolSet-CB')
        yourself ].

spec for: #CBNotPresent do: [
    spec
        package: 'CoolToolSet-CB' with: [ spec requires: 'CoolBrowser default'
];
        yourself ].
].

```

You can notice that the way to use project attributes is through the existing method `for:do:`. Inside that method you can do whatever you want: define groups, dependencies, etc. In our case, if `CoolBrowser` is present, then we just add `'CoolToolSet-CB'` to the default group. If it is not present, then `'CoolBrowser default'` is added to dependency to `'CoolToolSet-CB'`. In this case, we do not add it to the default group because we do not want that. If desired, the user should explicitly load that package also.

Again, notice that inside the `for:do:` you are free to do whatever you want.

1.15 Project version attributes

A configuration can have several optional attributes such as an author, a description, a blessing and a timestamp. Let's see an example with a new version 0.7 of our project.

```

ConfigurationOfCoolBrowser>>version07: spec
<version: '0.7' imports: #('0.7-baseline')>

spec for: #common do: [
    spec blessing: #release.
    spec description: 'In this release .....'
    spec author: 'JohnLewis'.
    spec timestamp: '10/12/2009 09:26'.
    spec
        package: 'CoolBrowser-Core' with: 'CoolBrowser-Core-BobJones.20';
        package: 'CoolBrowser-Tests' with: 'CoolBrowser-Tests-JohnLewis.8';
        package: 'CoolBrowser-Addons' with: 'CoolBrowser-Addons-JohnLewis.6
';

```

```
package: 'CoolBrowser-AddonsTests' with: 'CoolBrowser-AddonsTests-  
JohnLewis.1' ].
```

We will describe each attribute in detail:

Description: a textual description of the version. This may include a list of bug fixes or new features, changelog, etc.

Author: the name of the author who created the version. When using the OB-Metacello tools or MetacelloToolbox, the author field is automatically updated to reflect the current author as defined in the image.

Timestamp: the date and time when the version was completed. When using the OB-Metacello tools or MetacelloToolbox, the timestamp field is automatically updated to reflect the current date and time. Note that the timestamp must be a String.

To end this section, we show you can query this information. This illustrates that most of the information that you define in a Metacello version can then be queried. For example, you can evaluate the following expressions:

```
(ConfigurationOfCoolBrowser project version: '0.7') blessing.  
(ConfigurationOfCoolBrowser project version: '0.7') description.  
(ConfigurationOfCoolBrowser project version: '0.7') author.  
(ConfigurationOfCoolBrowser project version: '0.7') timestamp.
```

1.16 Conclusion

Metacello is an important part of Pharo. It will allow your project to scale. It allows you to control when you want to migrate to new version and for which packages. It is an important architectural backbone.

1.17 Metacello Memento

```

ConfigurationOfCoolToolSet>>baseline06: spec
  <version: '0.6-baseline'>
  spec for: #common do: [
    spec blessing: #baseline.
    spec repository: 'http://www.example.com/CoolToolSet'.

    "When we depend on other projects"
    spec project: 'CoolBrowser default' with: [
      spec
        className: 'ConfigurationOfCoolBrowser';
        versionString: #bleedingEdge;
        loads: #('default');
        file: 'CoolBrowser-Metacello';
        repository: 'http://www.example.com/CoolBrowser' ];
      project: 'CoolBrowser Tests'
      copyFrom: 'CoolBrowser default'
      with: [ spec loads: #('Tests').]

    "Our internal package dependencies"
    spec
      package: 'CoolToolSet-Core';
      package: 'CoolToolSet-Tests' with: [ spec requires: #('CoolToolSet-Core') ];
      package: 'CoolBrowser-Addons' with: [ spec requires: 'CoolBrowser-Core' ];
      package: 'CoolBrowser-AddonsTests' with: [
        spec requires: #('CoolBrowser-Addons' 'CoolBrowser-Tests' ) ].

    spec
      group: 'default' with: #('CoolBrowser-Core' 'CoolBrowser-Addons');
      group: 'Core' with: #('CoolBrowser-Core');
      group: 'Extras' with: #('CoolBrowser-Addon');
      group: 'Tests' with: #('CoolBrowser-Tests' 'CoolBrowser-AddonsTests');
      group: 'CompleteWithoutTests' with: #('Core' 'Extras');
      group: 'CompleteWithTests' with: #('CompleteWithoutTests' 'Tests')
    ].
  ].

```

```

ConfigurationOfCoolBrowser>>version07: spec
  <version: '0.7' imports: #('0.6-baseline')>
  spec for: #common do: [
    spec blessing: #release.
    spec description: 'In this release .....'.
    spec author: 'JohnLewis'.
    spec timestamp: '10/12/2009 09:26'.
    spec
      package: 'CoolBrowser-Core' with: 'CoolBrowser-Core-BobJones.20';
      package: 'CoolBrowser-Tests' with: 'CoolBrowser-Tests-JohnLewis.8';
      package: 'CoolBrowser-Addons' with: 'CoolBrowser-Addons-JohnLewis.6';
      package: 'CoolBrowser-AddonsTests' with: 'CoolBrowser-AddonsTests-JohnLewis.1']
  ].

```

Mariano ► what do you mean by "do not import baseline from other baselines" ? ◀ **Stéf** ► A baseline should not imports from another baseline: <version: '0.7' imports: #('0.6-baseline')> only in version: ◀

```
ConfigurationOfGemToolsExample>>development: spec      "note that the selector can be anything"
<symbolicVersion: #development>                        "#stable/#development/#bleedingEdge"
spec for: #common version: '1.0'.                      "'1.0' is the version of your development version"
"#common or your platform attributes: #gemstone, #pharo, or #pharo1.4"
```

```
ConfigurationOfGemToolsExample>>baseline10: spec
<version: '1.0-baseline'>
spec for: #common do: [
  spec blessing: #baseline.                            "required see above"
  spec repository: 'http://seaside.gemstone.com/ss/GLASSClient'.
spec
  project: 'FFI' with: [
    spec
      className: 'ConfigurationOfFFI';
      versionString: #bleedingEdge;                    "Optional. #stable/#development/#bleedingEdge/specific
version"
      repository: 'http://www.squeaksource.com/MetacelloRepository' ];
  project: 'OmniBrowser' with: [
    spec
      className: 'ConfigurationOfOmniBrowser';
      versionString: #stable;                          "Optional. #stable/#development/#bleedingEdge/specific
version"
      repository: 'http://www.squeaksource.com/MetacelloRepository' ];
  project: 'Shout' with: [
    spec
      className: 'ConfigurationOfShout';
      versionString: #stable;
      repository: 'http://www.squeaksource.com/MetacelloRepository' ];
  project: 'HelpSystem' with: [
    spec
      className: 'ConfigurationOfHelpSystem';
      versionString: #stable;
      repository: 'http://www.squeaksource.com/MetacelloRepository'].
spec
  package: 'OB-SUnitGUI' with: [spec requires: #('OmniBrowser')];
  package: 'GemTools-Client' with: [ spec requires: #('OmniBrowser' 'FFI' 'Shout' 'OB-SUnitGUI' ).];
  package: 'GemTools-Platform' with: [ spec requires: #('GemTools-Client' ). ];
  package: 'GemTools-Help' with: [
    spec requires: #('HelpSystem' 'GemTools-Client' ). ].
spec group: 'default' with: #('OB-SUnitGUI' 'GemTools-Client' 'GemTools-Platform' 'GemTools-Help').
```

```
ConfigurationOfGemToolsExample>>version10: spec
<version: '1.0' imports: #('1.0-baseline')>
spec for: #common do: [
  spec blessing: #development.
  spec description: 'initial development version'.
  spec author: 'dkh'.
  spec timestamp: '1/12/2011 12:29'.
spec
  project: 'FFI' with: '1.2';
  project: 'OmniBrowser' with: #stable;
  project: 'Shout' with: #stable;
  project: 'HelpSystem' with: #stable.
spec
  package: 'OB-SUnitGUI' with: 'OB-SUnitGUI-dkh.52';
  package: 'GemTools-Client' with: 'GemTools-Client-NorbertHartl.544';
  package: 'GemTools-Platform' with: 'GemTools-Platform.pharo10beta-dkh.5';
  package: 'GemTools-Help' with: 'GemTools-Help-DaleHenrichs.24'. ]
```

Loading. `load`, `load`: The `load` method loads the default group and if there is no default group defined, then all packages are loaded. The `load:` method takes as parameter the name of a package, a project, a group, or a collection of those items.

```
(ConfigurationOfCoolBrowser project version: '0.1') load.  
(ConfigurationOfCoolBrowser project version: '0.2') load: {'CBrowser-Core' . 'CBrowser-  
Addons'}.
```

Debugging. `record`, `record`: `loadDirectives` The message `record` does the record for the default group and if you want a specific group of items, you can use `record:`, just as it is for `load`.

```
((ConfigurationOfCoolBrowser project version: '0.2') record:  
 { 'CoolBrowser-Core' .  
   'CoolBrowser-Addons' }) loadDirective.
```

Proposed development process. Using metacello we suggest the following development steps.

Baseline	<i>"first we define a baseline"</i>
Version development	<i>"Then a version tagged as development"</i>
Validate the map	<i>"Once it is validated and the project status arrives to the desired status"</i>
Version release	<i>"We are ready to tag a version as release"</i>
Version development	<i>"Since development continue we create a new version"</i>
...	<i>"Tagged as development. It will be tagged as release and so on"</i>
Baseline	<i>"When architecture or structure changes, a new baseline will appear"</i>
Version development	<i>"and the story will continue"</i>
Version release	