

Chapter 1

Managing projects with Metacello

Have you ever had a problem when trying to load a nice project where you got an error because a package that you were not even aware of is missing or not correct? You've probably seen such a problem. The problem probably occurred because the project loaded fine for the developer but only because he has a different context than yours. The project developer did not use a *package management system* to explicitly manage the dependencies between his packages. In this chapter we will show you how to use Metacello, a package management system and the power that you can get using it.

1.1 Introduction

Metacello is a package *management* system for Monticello. But, exactly what is a *Package Management System*? It is a collection of tools to automate the process of installing, upgrading, configuring, and removing a set of software packages. It also groups packages to help eliminate user confusion and manages dependencies *i.e.*, which versions of what components should be loaded to make sure that the complete system is coherent.

A package management system provides a consistent method of installing packages. A package management system is sometimes incorrectly referred to as an installer. This can lead to confusion between them. Just for those who are familiar, package management systems for other technologies include Envy (in VisualAge Smalltalk), Maven in Java, apt-get/aptitude in Debian or Ubuntu, etc.

One of the key points of good package management is that *any package should be correctly loaded without needing to manually install anything other than what is specified in the package configuration*. Each dependency, and the dependencies of the dependencies must also be loaded in the correct order.

If it was not clear enough, the idea is that when using Metacello, you can take a PharoCore image, for example, and load *any* package of *any* project without any problems with dependencies. Of course, Metacello does not do magic so it is up to the developer to define the dependencies properly.

1.2 One tool for each job

To manage software, Pharo proposes several tools that are very closely related. In Pharo we have three tools: Monticello (which manages versions of source code), Gofer (which is a scripting API for Monticello) and Metacello (which is a package management system).

Monticello: Source code versioning. Source code versioning is the process of assigning either unique version names or unique version numbers to unique software states. At a fine-grained level, revision control incrementally keeps track of different versions of “pieces of software”. In object-oriented programming, these “pieces of software” are methods, classes or packages. A versioning system tool lets you commit a new version, update to a new one, merge, diff, revert, etc. Monticello is the source code versioning system used in Pharo and it manages Monticello packages. With Monticello we can do most of the above operations on packages but there is no way to easily specify dependencies, identify stable versions, or group packages into meaningful units. Monticello just manages package versions. Metacello manages package dependencies and the notion of projects.

Gofer: Monticello’s Scripting API. Gofer is a small tool on top of Monticello that loads, updates, merges, diffs, reverts, commits, recompiles and unloads groups of Monticello packages. Contrary to existing tools Gofer makes sure that these operations are performed as cleanly as possible. Gofer is a scripting API to Monticello (See Chapter ??).

Metacello: Package Management System. Metacello manages projects (sets of related Monticello packages) and their dependencies as well as project metadata. Metacello manages also dependencies between packages.

1.3 Metacello features

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

Declarative modeling. A Metacello project has named versions consisting of lists of explicit 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.

Distributed repositories. Metacello project metadata is represented as instance methods in a class therefore the Metacello project metadata is stored in a Monticello package. As a result, it is easy for distributed groups of developers to collaborate on ad-hoc projects.

Optimistic development. With Monticello-based packages, concurrent updates to the project metadata can be easily managed. Parallel versions of the metadata can be merged just like parallel versions of the code base itself.

Additionally, the following points are important considerations for Metacello:

- Cross-platform operations. Metacello must run on all platforms that support Monticello: currently Pharo, Squeak and GLASS.
- Conditional Monticello package loading. For projects that are expected to run on multiple platforms, it is essential that platform-specific Monticello packages can be conditionally loaded.

1.4 A Simple Case

Let's start using Metacello for managing a software project called CoolBrowser. The first step is to create a configuration for the project by simply copying the class `MetacelloConfigTemplate` and naming it `ConfigurationOfCoolBrowser` (by convention the class name for a Metacello configuration is composed by prefixing the name of the project with 'ConfigurationOf'). To do this, right click in the class `MetacelloConfigTemplate` 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 the `ConfigurationOfCoolBrowser` has some instance and class side methods. We will explain later how they are used. Notice that this class inherits from `Object`. Metacello configurations should be written such that they can be loaded without any prerequisites, including Metacello itself. So (at least for the time being) Metacello configurations cannot rely on a common superclass.

Now, imagine that the project “Cool Browser” has different versions, for example, 1.0, 1.0.1, 1.4, 1.67, etc. With Metacello you create an instance side method for each version of the project. Method names for version methods are unimportant as long as the method is annotated with the `<version:>` pragma as shown below.

By convention the version method is named `'versionXXX:'`, where XXX is the version number with illegal characters (like `'.'`) removed.

Suppose for the moment that our project “Cool Browser” has two packages: `CoolBrowser-Core` and `CoolBrowser-Tests` we name the method `ConfigurationOfCoolBrowser>>version01: spec` as shown below:

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

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

In this example, there are a lot of things we need to explain:

- Immediately after the method selector you see the pragma definition: `<version: '0.1'>`. The pragma 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 to identify the version being constructed.
- Looking a little closer you see that the argument to the method, `spec`, is the only variable in the method and it is used as the receiver to four different messages: `for:do:`, `package:with:`, `file:` and `repository:`.

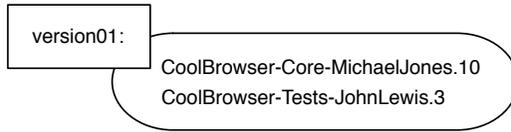


Figure 1.1: Simple version.

- Each time a block expression 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.
- In addition to `#common`, there are pre-defined attributes for each of the platforms upon which Metacello runs (`#pharo`, `#squeak`, `#gemstone` and `#squeakCommon`). Later in the chapter we will detail this feature.

The method `version01:` should be read as: Create version '0.1'. The common code for version '0.1' (specified using the message `for:do:`) consists of the packages named 'CoolBrowser-Core' (specified using the message `package:with:`) and 'CoolBrowser-Tests' whose files are named 'CoolBrowser-Core-MichaelJones.10' and 'CoolBrowser-Tests-JohnLewis.3' and whose repository is 'http://www.example.com/CoolBrowser' (specified using the message `repository:`).

Sometimes, a Monticello repository can be restricted and requires username and password. In such case the following message can be used:

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

We can access the specification created for version 0.1 by executing the following expression: (`ConfigurationOfCoolBrowser project version: '0.1'`) `spec`.

Creating a new version. Let us assume that the version 0.2 consists of the files 'CoolBrowser-Core-MichaelJones.15' and 'CoolBrowser-Tests-JohnLewis.8' and a new package 'CoolBrowser-Addons' with version 'CoolBrowser-Addons-JohnLewis.3'. Then, all you have to do is to create the following method named `version:`.

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

```
spec for: #common do: [
  spec repository: 'http://www.example.com/CoolBrowser'.
  spec
  package: 'CoolBrowser-Core' with: 'CoolBrowser-Core-MichaelJones.15';
```

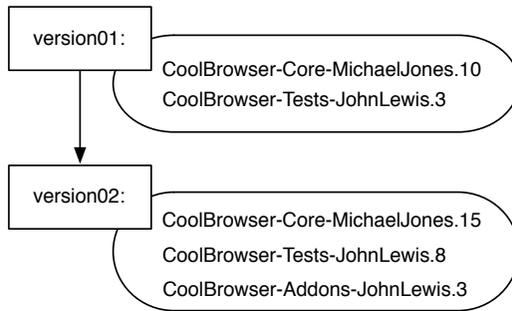


Figure 1.2: An second simple version.

```

package: 'CoolBrowser-Tests' with: 'CoolBrowser-Tests-JohnLewis.8' ;
package: 'CoolBrowser-Addons' with: 'CoolBrowser-Addons-JohnLewis.3'
  
```

1.5 Naming your configuration

In the previous section, we learned that we have to create a class for our configuration. It is not necessary to name this class with a particular name. Nevertheless there is a convention that we recommend you follow. The convention is to name the class `ConfigurationOfXXX` where `XXX` is your project. In our example, it is `ConfigurationOfCoolBrowser`.

There is a convention also to create a particular package with the same name as the configuration class and put the class there. In our case you will have the package `ConfigurationOfCoolBrowser` with only one class, `ConfigurationOfCoolBrowser`.

The package name and the class name match and by starting with `ConfigurationOfXXX` it is easier to scan through a repository listing the available projects. It is also very convenient to have the configurations grouped together rather than jumping around in the browser. That is why the repository <http://www.squeaksource.com/Pharo10MetacelloRepository>, <http://www.squeaksource.com/Pharo11MetacelloRepository> were created. They contain the configurations of several tools and applications and serve as a central repository.

Having all configurations in the same place has several advantages:

- Finding the configuration package is easier in the Monticello browser package list.
- Do not have any conflict with Monticello package naming (for exam-

ple, you can have the CoolBrowser package and this might conflict with the CoolBrowserConfiguration).

- When you have to manage multiple Configurations in the Package-Browser.
- Given that the name is slightly counter intuitive, it also has very few chances to collide with other names.

As a general practice, we suggest that you save the Configuration package in your working project and when you decide it is ready you can copy it into the MetacelloRepository.

Loading a Metacello configuration

Of course, the point of specifying packages in Metacello is to be able to load a coherent set of package versions. Here are a couple of examples for loading versions of the CoolBrowser project.

If you print the result of each expression, you will see the list of packages in load order. Metacello records not only which packages are loaded but also the order.

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

Note that in each case, all of the packages associated with the version are loaded – this is the default behavior. If you want to load a subset of the packages in a project, you should list the packages that you are interested in as an argument to the load: method as shown below:

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

1.6 Managing packages internal dependencies

A project is generally composed of several packages which often have dependencies on other packages. It is probable that a certain package depends on a specific version to behave correctly. Handling dependencies correctly is really important and this is what Metacello does for us.

There are two types of dependencies:

- Internal packages dependencies: Inside a certain project there are several packages and some of them depend on other packages in the same project.
- Dependencies between projects. It is common also that a project depends on another project or just on some packages of it. For example Pier (a meta-described cms) depends on Magritte (a meta-data modeling framework) and Seaside (a framework for web application development).

For now we will focus on the first case. In our example, imagine that the package CoolBrowser-Tests and CoolBrowser-Addons depends on CoolBrowser-Core. The new configuration '0.3' is defined as follows (See Figure 1.3):

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

spec for: #common do: [
  spec repository: 'http://www.example.com/CoolBrowser'.
  spec
    package: 'CoolBrowser-Core' with: 'CoolBrowser-Core-MichaelJones.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. Both CoolBrowser-Tests and CoolBrowser-Addons require CoolBrowser-Core to be loaded before they are loaded. Pay attention that since we did not specify the exact version number for the Cool-Browser package, we can have some problems (but do not worry, we will address this problem soon!).

With this version we are mixing structural information (required packages and repository) with the file version info. It is expected that over time the file version info will change from version to version while the structural information will remain relatively the same. To resolve this, Metacello introduces the concept of *Baselines*.

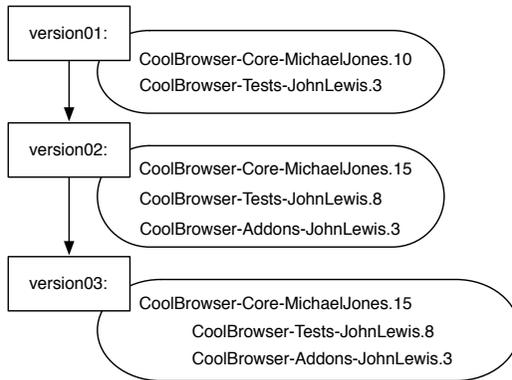


Figure 1.3: A version expressing requirements between packages.

1.7 Baselining

A baseline is a concept related to Software Configuration Management (SCM). From this point of view, a baseline is a well-defined, well-documented reference that serves as the foundation for other activities. Generally, a baseline may be a distributed work product, or conflicting work products that can be used as a logical basis for comparison.

In Metacello, a baseline represents the skeleton 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 package versions.

Now, let's continue with our example. First we modify it to use baselines: we create a method per baseline. **Stéf** ► *is the blessing: baseline important if so we should say it* ◀

```
ConfigurationOfCoolBrowser>>baseline04: spec
<version: '0.4-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' ]].
```

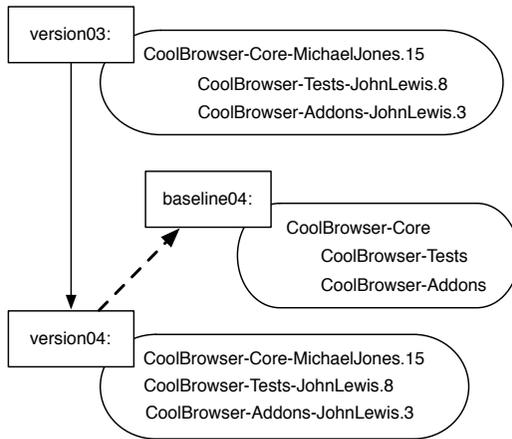


Figure 1.4: A version now imports a baseline that expresses dependencies between packages.

Baseline `baseline04:` will be used across several versions as for example the version '0.4' defined below (see Figure ??). In method `baseline04:` the structure of version '0.4–baseline' is specified. The baseline specifies a repository, the packages, but without version information, and the required packages (dependencies). We'll cover the blessing: method later.

To define the version, we use another 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-MichaelJones.15';
      package: 'CoolBrowser-Tests' with: 'CoolBrowser-Tests-JohnLewis.8';
      package: 'CoolBrowser-Addons' with: 'CoolBrowser-Addons-JohnLewis.3' ].
  
```

In the method `version04:` versions are specified. Note that the pragma `version:imports:` specifies the list of versions that this version (version '0.4') is based upon. In fact, if you print the spec for '0.4–baseline' and then print the spec for '0.4' you will see that '0.4' is a composition of both versions.

Using baseline the way to load this version is still the same:

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

Loading baselines. Even though version '0.4-baseline' does not have explicit package versions, you may load it. When the loader encounters a package name without version information it attempts to load the latest version of the package from the repository. Take into account that exactly the same happens if you define a package in a baseline but you don't specify a version for that package in a version method.

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

Sometimes when a number of developers are working on a project it may be useful to load a ctbaseline version so that you get the latest work from all of the project members.

New version. Now for example, we can have a new version '0.5' that has the same baseline (the same structural information), but different packages versions.

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

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

Note that version '0.5' uses the same baseline as version '0.4': '0.4-baseline' (see Figure 1.5).

After all these explanations you may have noticed that creating a baseline for a big project may require time. This is because you must know all the dependencies of all the packages and other things we will see later (this was a simple baseline). Once the baseline is defined, creating new versions of the project is very easy and takes very little time.

1.8 Groups

Suppose that now the CoolBrowser project is getting better and someone wrote tests for the addons. We have a new package 'CoolBrowser-AddonsTests'. This package depends on 'CoolBrowser-Addons' and 'CoolBrowser-Tests'. 

► add a figure ◀

Now we may want to load projects with or without tests. In addition,

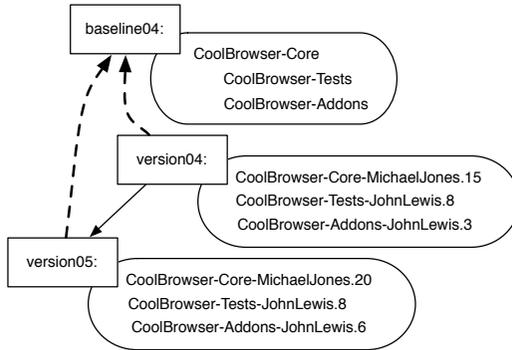


Figure 1.5: A second version imports again the baseline.

it would be convenient to be able to load all of the tests with a simple expression like the following:

```
(ConfigurationOfCoolBrowser project version: '1.0') load: 'Tests'.
```

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

```
(ConfigurationOfCoolBrowser project version: '1.0')
load: #('CoolBrowser-Tests' 'CoolBrowser-AddonsTests').
```

To solve this problem, Metacello offers the notion of group. A group is a list of items: packages, projects (as we will see in Section ??) or even other groups.

Groups are very useful because they let you customize different groups of items of different interests. Maybe you want to offer the user the possibility to install just the core, or with add-ons and development features. Let's go back to our example. Here we defined a new baseline '0.6-baseline' which defines 6 groups (see Figure 1.6).

To define a group we use the method `group: groupName with: group elements`. The parameter of `with:` can be a package name, a project, another group, or even an collection of those items. This way you can compose groups by using other groups.

```
ConfigurationOfCoolBrowser>>baseline06: spec
<version: '0.6-baseline'>

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

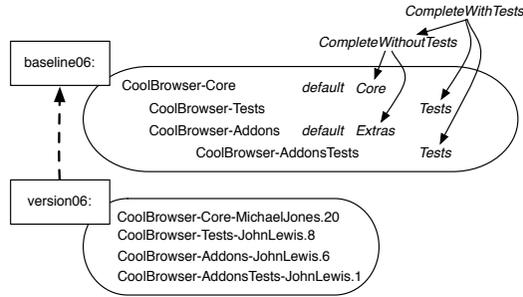


Figure 1.6: A baseline with groups: default, Core, Extras, Tests, Complete-WithoutTests and CompleteWithTests.

```

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' )
].

```

Note that we are defining the groups in the baseline version, since groups are a structural component. The package version is the same as version 0.5 in the previous example but with the new package CoolBrowser-AddonsTests.

```

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

```

```

spec for: #common do: [
  spec blessing: #development.
  spec
    package: 'CoolBrowser-Core' with: 'CoolBrowser-Core-MichaelJones.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 group, the idea is that you can use the name of a group anywhere that you would use the name of project or package. The load: method takes as parameter the name of a package, a project, a group or even an collection of those items. Any of the following statements are then possible:

(ConfigurationOfCoolBrowser project version: '1.0') load: 'CoolBrowser-Core'.
"Load a single package"

(ConfigurationOfCoolBrowser project version: '1.0') load: 'Core'.
"Load a single group"

(ConfigurationOfCoolBrowser project version: '1.0') load: 'CompleteWithTests'.
"Load a single group"

(ConfigurationOfCoolBrowser project version: '1.0') load: #('CoolBrowser-Core' 'Tests').
"Loads a package and a group"

(ConfigurationOfCoolBrowser project version: '1.0') load: #('CoolBrowser-Core' 'CoolBrowser-Addons' 'Tests').
"Loads two packages and a group"

(ConfigurationOfCoolBrowser project version: '1.0') load: #('CoolBrowser-Core' 'CoolBrowser-Tests').
"Loads two packages"

(ConfigurationOfCoolBrowser project version: '1.0') load: #('Core' 'Tests').
"Loads two groups"

Default group. The 'default' group is a special one and when a default group is defined, the load method loads the members of the 'default' group instead of all of the packages:

(ConfigurationOfCoolBrowser project version: '1.0') load.

In such a case, if you want to load all the packages of a project, you should use the predefined group named 'ALL' as shown below:

(ConfigurationOfCoolBrowser project version: '1.0') load: 'ALL'.

1.9 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-MichaelJones.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:

Blessing: in software development it is very common that 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, release candidate, etc. Sometimes we want to refer also to the state of a project.

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.

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 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 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.
```

1.10 Pre and post code execution

Occasionally, you find that you need to perform some code either after or before a package or project is loaded. For example, if we 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 offers such feature by means of the two methods `preLoadDolt:` and `postLoadDolt:`. The arguments passed to these methods are selectors of methods defined on the configuration class as shown below. For the

moment, these pre and post scripts can be assigned to a single package or a whole project.

Continuing with our example:

```
ConfigurationOfCoolBrowser>>version08: spec
<version: '0.8' 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: [
      spec
        file: 'CoolBrowser-Core-MichaelJones.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' ].
```

```
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* Stéf ▶ 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 description: 'In this release .....'.
    spec author: 'JohnLewis'.
    spec timestamp: '10/12/2009 09:26'.
    spec preLoadDolt: #preLoadForCoolBrowser.
    spec postLoadDolt: #postLoadForCoolBrowser.

    spec
      package: 'CoolBrowser-Core' with: [
        spec
          file: 'CoolBrowser-Core-MichaelJones.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 platform where we are loading the code. But in order to do that, we need to give Metacello some 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-Core' with: 'CoolBrowser-Core-MichaelJones.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' ].

```

```

spec for: #gemstone do: [
  spec package: 'CoolBrowser-Platform' with: 'CoolBrowser-PlatformGemstone-
MichaelJones.4' ].
spec for: #pharo do: [
  spec package: 'CoolBrowser-Platform' with: 'CoolBrowser-PlatformPharo-
JohnLewis.7' ].
spec for: #squeak do: [
  spec package: 'CoolBrowser-Platform' with: 'CoolBrowser-JohnLewis-dkh.3' ].

```

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' ].
spec for: #squeak do: [
  spec package: 'CoolBrowser-Platform' with: 'CoolBrowser-PlatformSqueak' ].

```

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 uni-

fied 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.

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 groups, packages, repositories, etc, that are dependent on a 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 Project Configuration Dependencies

Stéf ► *I would move that section before the property ones* ◀

In the same way a package can depend on other packages, a project can depend on other projects. For example, Pier which is a CMS using meta-description depends on Magritte and Seaside. A project can depend completely on one or more other projects, on a group of packages of a project, or even just on one or more packages of a project. Here we have basically two scenarios depending whether the other projects is described or not using a Metacello configurations.

Depending on project without configuration

A package A from a Project X depends on a package B from project Y and project Y does not have any Metacello configuration (typically when there is only one package in the project). In this case do the following: **Stéf** ► *we should use the same example A and B sucks* ◀

```
``In the 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'.
```

The problem here is that as the project B does not have a Metacello configuration, the dependencies of B are not managed. Thus, package B can have dependencies, but they 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 project with configuration

Package A depends on a package or project B, but project B has also a configuration. This is the common case that we want to present now.

To continue with our example, we introduce a new project called CoolToolSet which uses the packages from the CoolBrowser project. The configuration class is called ConfigurationOfCoolToolSet. We define two packages in CoolToolSet called CoolToolSet-Core and CoolToolSet-Tests. Of course, these

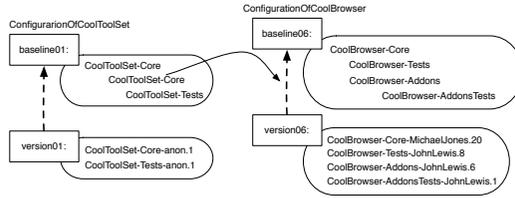


Figure 1.7: Dependencies between configurations.

packages depend on packages from CoolBrowser. Let's assume for a moment that the package that contains ConfigurationOfCoolBrowser class is called CoolBrowser-Metacello instead of also ConfigurationOfCoolBrowser (as we recommended). This will be better to understand each parameter.

Let's see the first version then:

```

ConfigurationOfCoolToolSet>>version01: spec
<version: '1.0' imports: #'(1.0-baseline)'>
spec for: #common do: [
  spec
    package: 'CoolToolSet-Core' with: 'CoolToolSet-Core-anon.1';
    package: 'CoolToolSet-Tests' with: 'CoolToolSet-Tests-anon.1'].
  
```

```

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';
      versionString: '1.0';
      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' ]].
  
```

What we did here in baseline0.1 was to create a project reference for the CoolBrowser project (see Figure 1.7). The className: specifies the name of the class that contains the project metadata. If the class is not present in the image, then we need to supply all the necessary information so that Metacello can search and load the configuration for the project.

The file: and repository: specifications give us the information needed to

load the project metadata from a repository in case the configuration class is not already present in the image. If the Monticello repository is protected, then you have to use the message: repository:username:password:.

Note that the values for the className: and file: attributes are the same: 'ConfigurationOfCoolBrowser'. As we've mentioned before, by convention the name of the configuration class and the configuration package should be the same, however, with Metacello it is not required that the two be the same.

Finally, the versionString: and loads: tell us which version of the project to load and which packages or groups (the parameter of load: can be the name of a package, or the name of a group or those predefined groups like 'ALL') to load from the project. **Stéf** ▶ *should I get a version 1.0 of the version/baseline of CoolBrowser. This is not clear and I need to know that.* ◀

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 is recommended to put a name that make sense to that project reference. **Stéf** ▶ *why all how can I say default* ◀
Stéf ▶ *How can I say that I should load a specific version of another configurations?* ◀

Now can now download CoolToolSet like this:

(ConfigurationOfCoolToolSet project version: '0.1') load.

Note that the entire CoolBrowser project is loaded before 'CoolToolSet-Core'.

Now as is often the case, it is useful to separate the test package from the core packages for a project. So, we can write for example, the following baseline:

```
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' with: [
spec
```

```

    className: 'ConfigurationOfCoolBrowser';
    versionString: '1.0';
    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' )].

```

Here we created 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 Cool Browser. We then made 'CoolToolSet-Core' require 'CoolBrowser default' and 'CoolToolSet-Tests' requires 'CoolToolSet-Core' and 'CoolBrowser Tests'.

Now it is possible to load just the core packages:

```
(ConfigurationOfCoolToolSet project version: '1.1') load: 'CoolToolSet-Core'.
```

or the core including tests:

```
(ConfigurationOfCoolToolSet project version: '1.1') load: 'CoolToolSet-Tests'.
```

As you can see, in baseline02: there is redundant information for each of the project references. To solve that situation, we can use the project:copyFrom:with: method to eliminate the need to specify the bulk of the project information twice. 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' with: [ spec requires: 'CoolBrowser default' ];
  package: 'CoolToolSet-Tests' with: [
    spec requires: #('CoolToolSet-Core' 'CoolBrowser Tests' )].

```

Not only in this baseline but also in baseline01 we did something that is not always useful: we put the version of the referenced projects in the baseline instead of in the version method. If you look at baseline01 or baseline02 you can see that we put `versionString: '1.0'`. But sometimes, this information (the version of the referenced project) remains the same among different versions of the dependent project. If the referenced project is very stable this technique works very well.

On the other hand, if the referenced project will be changing over time, then we recommend that you not specify the `#versionString` in the baseline method, but specify in the version method as follows:

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

If we don't define a version String at all for 'CoolBrowser default' and 'CoolBrowser Tests' in the version method, then the version specified in the baseline will be used. If there is no version specified in the baseline method, then Metacello loads the latest version of the project.

1.13 Symbolic Versions

In any large evolving application relying on other applications and libraries, it is difficult to know which version of a configuration to use with a specific versions. This is especially true for Pharo applications where some people should maintained applications developed for a given version, while others are working on the latest build.

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 1.2.3 is currently the latest development version aimed at Pharo1.2, and version 1.2.3 cannot be loaded into Pharo1.0.

Obviously version 1.1.3 should be used in Pharo1.0 and version 1.2.3 should be used in Pharo1.2. Now up until recently there is no way for a developer to communicate this information to his users using Metacello.

The latest version of Metacello introduces symbolic versions whose purpose is to provide a way 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 symbolic version pragma:

```
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 statement: (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'. ) ]].
```

New Platform Version Attributes

```
project
  | versionString projectAttributes |
  ↑ project
  ifNil: [
    "Bootstrap Metacello if it is not already loaded"
    self class ensureMetacello.
    "Construct Metacello project"
    project := (Smalltalk at: #MetacelloMCPProject) new.
    projectAttributes := #().
    Smalltalk
      at: #SystemVersion
      ifPresent: [ :cl |
        versionString := cl current version.
```

```

((versionString beginsWith: 'Pharo-1.1.1') or: [ versionString beginsWith: '
Pharo1.1.1' ])
  ifTrue: [ projectAttributes add: #'pharo1.1.x' ] ].
project projectAttributes: projectAttributes.
(Smalltalk at: #MetacelloVersionConstructor) on: self project: project.
project loadType: #linear ]

```

Standard Symbolic Versions

A couple of standard symbolic versions have already been defined:

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.

development. A symbolic version that specifies the literal version that us 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.

stable. A symbolic version specifies the stable literal version for a particular 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.

1.14 Script and Tool Support

With this release, I've added two new classes: MetacelloBaseConfiguration and MetacelloToolBox, which should make developing with Metacello and writing tools for Metacello a whole lot easier.

Development Support

The MetacelloBaseConfiguration class is aimed at eventually becoming the common superclass for all Metacello configurations, but I'm not quite

ready to go there, yet. For now, though, the class serves as the location for defining the common default symbolic versions (bleedingEdge at the present time) and as the place to find development support methods:

```
compareVersions -â Compare theâ #stable version to #development version.
createNewBaselineVersion -â Create a new baseline version based upon the #stable
  version as a model.
createNewDevelopmentVersion -â Create a new #development versionâ using the
  #stable version as model.
releaseDevelopmentVersion: -â Release #development version: set version blessing
  to #release, update the #development and #stable symbolic version methods and
  save the configuration.
saveConfiguration: -â Save the mcz file that contains the configuration to it's
  repository.
saveModifiedPackagesAndConfiguration: -â Save modified mcz files, update the
  #development version and then save the configuration.
updateToLatestPackageVersions: -â Update the #development version to match
  currently loaded mcz files.
validate -â Check the configuration for Errors, Critical Warnings, and Warnings.
I suggest that you take a look at these methods and copy the methods that you find
  useful to the class-side of your own configuration, so that you can simplify the
  management of your own configuration.
```

Metacello Toolbox API

Theâ MetacelloToolBox class is aimed at providing a common API for development scripts and Metacello tools. The development support methods were implemented using the Metacello Toolbox API and the OB-Metacello tools have been reimplemented to use the Metacello Toolbox API.â For an overview of the Metacello Toolbox API, you can look in theâ HelpBrowser at the 'Metacello>>API Documentation' section. The instance-side methods forâ MetacelloToolBox support the â programmatic editing of Metacello configurations from the creation of a new configuration classes to the creation and changing of literal and symbolic version methods. The instance-side methods are intended for the use of Tools developers and are covered in the ProfStef tutorial: 'Inside Metacello Toolbox API'. The class-side methods for MetacelloToolBox support a number of configuration management tasks. The target the initial release of the Metacello Toolbox API is to support the basic Metacello development cycle. In addition to the following section the Metacello development cycle is covered in the ProfStef tutorial: 'Metacello Development Cycle'.

Development Cycle Walk Through

In this section we'll take a walk through a typical development cycle and provide examples of how the Metacello Toolbox API can be used to support your development process:

Example Setup

Project Startup

Create Configuration and Initial Baseline

Create Initial Literal Version

Validation
 Save Initial Configuration
 Development Cycle
 Platform Testing
 Release
 Open New Version for Development
 Configuration Checkpoints
 Update Project Structure
 and Repeat...
 1.0 Example Setup

When you are developing your project and are building your configuration for the first time, you already have the packages that make up your project loaded and correctly running on your image. In this example, it is necessary to load a set of packages to simulate a image that will be used to build the first configuration of the project. We'll cheat here and use an existing configuration (ConfigurationOfGemTools) to download and install in our image all the packages and dependencies needed (just as we would have to do by hand if we were the maintainers of the project). So, don't pay much attention to this step and only focus on the fact that after evaluating it, you'll have loaded in your image all the packages needed to build the example configuration:

Gofer new

```
squeaksource: 'MetacelloRepository';
package: 'ConfigurationOfGemTools';
load.
```

((Smalltalk at: #ConfigurationOfGemTools) project version: '1.0-beta.8.3')

```
load: 'ALL'.
```

GemTools is expected to work in Squeak (Squeak3.10 and Squeak4.1) and Pharo (Pharo1.0 and Pharo1.1) and depends upon 4 other projects:

FFI

OmniBrowser

Shout

HelpSystem

GemTools itself is made up of 5 mcz files from the <http://seaside.gemstone.com/ss/GLASSClient> repository (note that I've made some minor structural changes to simplify the example):

OB-SUnitGUI – requires 'OmniBrowser'. in 'default' group.

GemTools-Client – requires 'OmniBrowser', 'FFI', 'Shout', and 'OB-SUnitGUI'. in ' default' group.

GemTools-Platform – requires 'GemTools-Client'. in 'default' group.

GemTools-Help – requires 'HelpSystem' and 'GemTools-Client'. in 'default' group.

2.0 Project Startup

Create Configuration and Initial Baseline

Create Initial Literal Version

Validation

Save Initial Configuration

2.1 Create Configuration and Initial Baseline

Use the toolbox API to create the initial baseline version by specifying the name, repository, projects, packages, dependencyMap and group composition:

```
MetacelloToolBox
```

```
createBaseline: '1.0-baseline'
```

```
for: 'GemToolsExample'
```

```
repository: 'http://seaside.gemstone.com/ss/GLASSClient'
```

```
requiredProjects: #('FFI' 'OmniBrowser' 'Shout' 'HelpSystem')
```

```
packages: #('OB-SUnitGUI' 'GemTools-Client' 'GemTools-Platform'
  'GemTools-Help' )
```

```
dependencies:
```

```
{('OB-SUnitGUI' -> #( 'OmniBrowser' )).
```

```
('GemTools-Client' -> #( 'OmniBrowser' 'FFI' 'Shout' 'OB-SUnitGUI' )).
```

```
('GemTools-Platform' -> #( 'GemTools-Client' )).
```

```
('GemTools-Help' -> #( 'HelpSystem' 'GemTools-Client' )). }
```

```
groups:
```

```
{('default' -> #( 'OB-SUnitGUI' 'GemTools-Client' 'GemTools-Platform'
  'GemTools-Help' )}). }
```

The #createBaseline:... method copies the class MetacelloConfigTemplate to a ConfigurationOfGemToolsExample and creates a #baseline10: method that looks like the following:

```
baseline10: spec
```

```
<version: '1.0-baseline'>
```

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

```
spec blessing: #'baseline'.
```

```
spec repository: 'http://seaside.gemstone.com/ss/GLASSClient'.
```

```
spec
```

```
project: 'FFI' with: [
```

```
spec
```

```
className: 'ConfigurationOfFFI';
```

```
versionString: #'bleedingEdge';
```

```
repository: 'http://www.squeaksource.com/MetacelloRepository' ];
```

```
project: 'OmniBrowser' with: [
```

```
spec
```

```
className: 'ConfigurationOfOmniBrowser';
```

```
versionString: #'stable';
```

```
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' ). ].

```

Note that for the 'FFI' project the versionString is #bleedingEdge, while the versionString for the other projects is #stable'. At the time of this writing the FFI project did not have a #stable symbolic version defined, so the default versionString is set to a #bleedingEdge. If a #stable symbolic version is defined for the project, the the default versionString is #stable. There are no special version dependencies for the GemTools project so the defaults will work just fine.

2.2 Create Initial Literal Version

Now we use the toolbox API to create the initial literal version of the project. This toolbox method bases the definition of the literal version on the baseline version that we created above and uses the currently loaded state of the image to define the project versions and mcZ file versions:

```

MetacelloToolBox
  createDevelopment: '1.0'
  for: 'GemToolsExample'
  importFromBaseline: '1.0-baseline'
  description: 'initial development version'.

```

The #createDevelopment:... method creates a #version10: method in your configuration that looks like this:

```

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'.].

Note that when the #stable symbolic version specifications were carried through into the literal version. If the version isn't #stable, then the currentVersion of the project is filled in, just as the current version of each mcz file is set for the packages. Note also that the blessing of the version '1.0' is set to #development. By setting the blessing of a newly created version to #development, you indicate that the version is under development and is subject to change without notice. The #createDevelopment:... method also creates a #development: method and specifies that version '1.0' is a #development symbolic version:

```
development: spec
  <symbolicVersion: #'development'>
  spec for: #'common' version: '1.0'.
```

2.3 Validation

Whenever you finish editing a configuration you should validate it to check for mistakes that may cause problems later on:

(MetacelloToolBox validateConfiguration: ConfigurationOfGemToolsExample) explore.

If the list comes back empty then you are clean. Otherwise you should address the validation issues that show up. Validation issues are divided into three categories:

Warning – issues that point out oddities in the definition of a version that do not affect behavior.

Critical Warning – issues that identify inconsistencies in the definition of a version that may result in unexpected behavior.

Error – issues that identify explicit problems in the definition of a version that will result in errors if an attempt is made to resolve the version.

Here's an example of a Critical Warning validation issue:

```
Critical Warning: No version specified for the project reference 'OCompletion'
in version '1.1'
```

```
{ noVersionSpecified }
```

```
[ ConfigurationOfOmniBrowser #validateVersionSpec: ]
```

The first and second line is the explanation, a human readable error message. The third line is the reasonCode, a symbol that represents the category of the issue.

You can check out the meanings of the various reasonCodes online or through the following toolbox call:

```
(MetacelloToolBox descriptionForValidationReasonCode: #noVersionSpecified)
inspect.
```

The fourth line lists the configurationClass, the configuration that spawned the issue (there is a different toolbox method for running a recursive configuration validation) and the callSite, which is the name of the validation method that generated the error (this is used mainly for debugging).

2.4 Save Initial Configuration

The first time you save your configuration, you have to decide where to keep your configuration. It makes sense to keep the configuration in your development repository. The first time that you save your configuration you need to use the MonticelloBrowser or an expression like the following:

```
Gofer new
  url: 'http://www.example.com/GemToolsRepository';
  package: 'ConfigurationOfGemToolsExample';
  commit: 'Initial version of configuration'.
```

3.0 Development Cycle

Platform Testing

Release

Open New Version for Development

Configuration Checkpoints

Update Project Structure

and Repeat...

3.1 Platform Testing

To finish the validation of your configuration, you need to do some test loads on your intended platforms. For GemTools I generally do a test load into a fresh image (each of the supported PharoCore and Squeak4.1) with the following load expression:

```
Gofer new
  url: 'http://www.example.com/GemToolsRepository';
  package: 'ConfigurationOfGemToolsExample';
  load.
```

((Smalltalk at: #ConfigurationOfGemToolsExample)

project version: #development) load.

and then run the unit tests. Note that for the GemTools unit tests you need to have

GemStone installed.

3.2 Release

Once you are satisfied that the configuration loads correctly on your target platforms, you can release the #development into production using the following expression:

```
MetacelloToolBox
  releaseDevelopmentVersionIn: ConfigurationOfGemToolsExample
  description: '- release version 1.0'.
```

The toolbox method #releaseDevelopmentVersionIn:description: does the following:

set the blessing of the #development â version to #release

sets the #development â version to #notDefined

sets the #stable â version to the literal version of the #development version (in this case '1.0')

saves the configuration mcz file to the correct repository

The #development: method ends up looking like this:

```
development: spec
```

```
<symbolicVersion: #'development'>
```

```
spec for: #'common' version: #'notDefined'.
```

The #stable: method ends up looking like this:

```
stable: spec
```

```
<symbolicVersion: #'stable'>
```

```
spec for: #'common' version: '1.0'.
```

Finally you can copy the configuration to the MetacelloRepository using the following expression:

```
MetacelloToolBox
  copyConfiguration: ConfigurationOfGemToolsExample
  to: 'http://www.squeaksource.com/MetacelloRepository'.
```

3.3 Open New Version for Development

```
MetacelloToolBox
  createNewDevelopmentVersionIn: ConfigurationOfGemToolsExample
  description: '- open 1.1 for development'.
```

3.4 Configuration Checkpoints

During the course of development it makes sense to save checkpoints of your development to your repository. To setup this example you should load a newer version of GemTools and get some new mcz files loaded to simulate development:

```
(ConfigurationOfGemTools project version: '1.0-beta.8.4')
  load: 'ALL'.
```

Now that you've simulated some development you can update the #development version of your project so that it references the new mcz files you've loaded

```
MetacelloToolBox
  updateToLatestPackageVersionsIn: ConfigurationOfGemToolsExample
  description: '- fixed Issue 1090'.
```

Then save the configuration to your repository:

```
MetacelloToolBox
  saveConfigurationPackageFor: 'GemToolsExample'
  description: '- fixed Issue 1090'.
```

Or do both steps with one toolbox method:

```
MetacelloToolBox
  saveModifiedPackagesAndConfigurationIn: ConfigurationOfGemToolsExample
  description: '- fixed Issue 1090'.
```

3.5 Update Project Structure

In the course of development it is sometimes necessary to add a new package or reference and addition project. In this case let's add a package to the project called 'GemTools-Overrides' ('GemTools-Overrides' is actually part of the GemTools project already and I just left it out of the previous example). To add a new package a project you need to:

```
create a new baseline version to reflect the new package and dependencies
update existing #development version to reference the new baseline version and
include the explicit version for the new package
```

You can do that manually by using a class browser to manually:

```
copy and edit the old baseline version to reflect the new structure
```

```
edit the existing #development version method
```

Or you can use the Metacello Toolbox API (not finished/tested yet):

```
| toolbox |
toolbox := MetacelloToolBox configurationNamed: 'GemToolsExample'.
```

toolbox

```
createVersionMethod: 'baseline11:' inCategory: 'baselines' forVersion: '1.1-baseline';
addSectionsFrom: '1.0-baseline'
  forBaseline: true
  updateProjects: false
  updatePackages: false
  versionSpecsDo: [ :attribute :versionSpec |
    attribute == #common
      ifTrue: [ versionSpec packages add: (toolbox createPackageSpec: 'GemTools-
        Overrides') ].
      true ];
commitMethod;
modifyVersionMethodForVersion: '1.1'
  versionSpecsDo: [ :attribute :versionSpec |
    attribute == #common
      ifTrue: [ versionSpec packages add: (toolbox createPackageSpec: 'GemTools-
        Overrides') ].
    false ];
commitMethod.
```
