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.

Packages are distributions of software and metadata such as the software's full name, description of its purpose, version number, author, changes of a version, among others, and a list of dependencies necessary for the software to run properly. Upon installation, metadata can be stored in a local package database, in code or any other scheme.

A package management system provides a consistent method of in-

stalling software. 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.

The software is often downloaded from a number of software repositories. It is also common to have a central repository where all the software is submitted.

1.2 One tool for each job

To manage software in this century we use several tools that are very closely related. As a principle, we have to know that we need *a tool for each job*. A tool cannot do everything. Each tool satisfies a limited amount of problems and it will probably delegate to another tool to do certain tasks. If you read the first Pharo By Example book you may noticed that this concept is quite similar to Object-Oriented Design: we don't want to have only one object that does everything and we rather to have multiple objects where each object has certain behavior and they collaborate together to do a specific task.

So we have Metacello, Monticello and Gofer.

Monticello: Source code versioning. Source code versioning is the process of assigning either unique version names or unique version numbers to unique states of software. At a fine-grained level, revision control is often used for keeping track of incrementally different versions of "pieces of software". In object oriented programming, these "pieces of software" that are versioned can be methods, classes or even packages. A Version System tool lets you commit a new version, update to a new one, merge, diff, revert, etc. In our case, the source code versioning system we use is Monticello and the "piece of software" are the Monticello packages. With Monticello we can do most of these functions but there is no way to specify dependencies, identify stable versions, or group packages into meaningful units. The last ones are functions for Metacello. Monticello just manages the packages versions.

Gofer: Scripting Monticello 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 clean as possible. Gofer acts a scripting API to Monticello.

Metacello: Package set management. We have already explained what a package management system and we said Metacello was exactly that.

Stéf ►*do we have a comparison apt ... one guy snet that in the mailing list*◄

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.

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 a 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

```
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:.

  With the evaluation of each block expression, 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.

This method should be read as: Create version '0.1'. The common code for version '0.1' (for:do:) consists of the packages named 'CoolBrowser-Core' (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/CoolBrow (repository:).

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

Let us assume that the version 0.2 consist 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:

```
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';
      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. So, for example, in our case you will have the package ConfigurationOfCoolBrowser with only one class, ConfigurationOfCoolBrowser.

The package name and the class name match and by beginning 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/MetacelloRepository* was created. It contains the configurations of several tools and applications and serves as a central repository.

Having all configurations in the same place has several advantages:

- Finding the configuration package is easier in the Monticello package list.
- Do not have any conflict with Monticello package naming (for example, you can have the CoolBrowser package and this might conflict with the CoolBrowserConfiguration).
- When you have to manage multiple Configurations in the PackageBrowser.
- 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 examples for loading versions of the CoolBrowser.

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 internal dependencies of packages

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

There are two types of dependencies:

- Internal dependencies of packages. Inside a certain project there are several packages and each of them depends on others of the same project.
- Dependencies between projects. It is common also that a project depends completely on another project or just on some packages of it.

Now we will focus in the first case and later we will also explain you the second one. In our example, imagine that 'CoolBrowser-Tests' and 'CoolBrowser-Addons' depends on 'CoolBrowser-Core'. Let's see how we should write this information in our configuration:

```
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. Now do not specify the exact number version of the Cool-Browser package files to be loaded. This can be a problem and we will address this problem soon.

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

With this version we are mixing structural information (required packages and repository) with the dynamic 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 static. To resolve this, we introduce a new conecpt: *Baselines*.

1.7 Baselining

The baseline is a concept related to Software Configuration Managment (SCM). From such 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 the Metacello world a baseline is a bit more related to static information. It is like an snapshot of the static information of a project at certain moment. The baseline should define the structure of a project using just names of packages. When the structure changes, the baseline should be updated. In the absence of structural changes, the changes in package versions is what is interesting.

Now, let's continue with our example. We will modify it to use baselines. With the baseline it happens exactly the same as the versions: we need to create a method per baseline but what is really important is the pragma used inside that method.

```
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' ].
```

```
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' ] ].
```

Here we are creating a baseline version specification which is expected to be used across several versions and the version specification which is restricted to the file versions.

In method #baseline04: the structure of version '0.4-baseline' is specified. The repository is listed, the packages are listed and the required packages (dependencies) are defined. We'll cover the #blessing: later.

In the method version04: the file versions are specified. You will note that the pragma adds an #imports: component that 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 can see that '0.4' is a composition of both versions.

The way to load this version is still the same:

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

Finally, even though version '0.4-baseline' does not have explicit package versions, you may load the version. When the loader encounters a package name without version information it will attempt 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 #baseline version so that you get the latest work from all of the project members.

Now for example, I can have a new version 0.5 that has the same baseline (the same static information), but different versions. For example:

```
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 5 uses the same baseline of version 4: '0.4-baseline'.

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 Cool Browser is even better and someone wrote tests for the addons. We have a new package called 'CoolBrowser-AddonsTests'. This package depends on 'CoolBrowser-Addons' and 'CoolBrowser-Tests'.

With two Test packages 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: { 'Example-Tests'. '
    Example-AddOnTests'. }.
```

Metacello has a very good and useful feature called "groups". It is a very simple concept and it just defined as a group of items. These items can be: packages, projects (as we will see later) or even other groups.

Groups are very useful because they let you customize different group of items for different interest. Maybe a user wants to install just the runtime or core, other not only the core but also the development tools. Another person may want or not to include the addons or tests, etc. Thus, I can create different groups to satisfy all these different needs.

So...let's go to the example:

```
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' )
    ].
```

The version is the same as the previous example but with a new package:

```
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' ].
```

As you can notice, the method used is *#group:with:*. The parameter of *with:* can receive the name of a package, a project, another group, or even an collection of those items. So you can compose groups by other groups.

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 loading all of the packages:

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

If you want to load all of the packages in a project, then the pseudo group 'ALL' may be used as follows:

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

If we define groups the idea then is that you are able to load a group. The #load: method receives as parameter the same as we explained for the #group:with: . So, any of the following statements are correct:

```
"Load a single package"
(ConfigurationOfCoolBrowser project version: '1.0') load: 'CoolBrowser-Core'.
"Load a single group"
(ConfigurationOfCoolBrowser project version: '1.0') load: 'Core'.
"Load a single group"
(ConfigurationOfCoolBrowser project version: '1.0') load: 'CompleteWithTests'.
"Loads a package and a group"
(ConfigurationOfCoolBrowser project version: '1.0') load: #('CoolBrowser-Core
      ' 'Tests').
"Loads two packages and a group"
```

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

1.9 Project version attributes

First, let us tell you something. Metacello not only includes all what we have been learning and what we will be still learning along this chapter, but also a complete set of tools with UI (user interface). These tools are based on OmniBrowser (OB), and thus, they are called OB-Metacello. This tool lets you save packages, spawn new versions, update package methods, load latest packages, save projects, update projects, among others. Unfortunately, this topic is not covered in this chapter. That's all you need to know for the moment.

Now...regarding the title of this section, there are attributes that are very useful when creating a new version of a project. For instances, we can have an author, a description, a blessing and a timestamp. Maybe in the future there can be even more. Let's see an example in a new version 7.

```
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 see each attribute in particular:

- Author: it is the name of the author who did that version. When using the OB-Metacello tools the author field is automatically updated to reflect the current author as defined in the image.
- Description: this a description of a version. This may include bug fixes, new features, changelog, etc.
- Timestamp: it is just the date and time when the version was done. When using the OB-Metacello tools the timestamp field is automatically updated to reflect the current DateAndTime. Finally, note that the timestamp must be a String.
- Blessing: in software development is very common that packages or projects pass through several stages or steps during the software development process or life cycle. For example, development, alpha, beta, release, release candidate, etc. Sometimes we want to refer also to the "state" of a project, or sometimes we just want to use any other tag that may be useful. Blessing attribute can be used for all these cases.

  Blessings can be also used as a filter. For example, you will notice that 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 would execute this expression:

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

  Nevertheless, you can get the very last version independent of blessing by executing this expression:

```
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 should load the absolute bleeding edge of the project.

To finalize this section, we will show you can ask for this information. Know that most of the information that you define in a Metacello configuration can then be queried. For these cases you can evaluate, for example, the following expressions:

```
(ConfigurationOfCoolBrowser project version: '0.7') author.
(ConfigurationOfCoolBrowser project version: '0.6') blessing.
(ConfigurationOfCoolBrowser project version: '0.7') blessing.
(ConfigurationOfCoolBrowser project version: '0.7') description.
(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 load some workspaces after the installation. In summary, there are a lot of situations where we need this kind of stuff. Fortunately Metacello provide us with such features. They are called #preLoadDoIt: and #postLoadDoIt: . 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';
          preLoadDoIt: #preloadForCore;
          postLoadDoIt: #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, #preLoadDoIt: and #postLoadDoIt: receives a selector that will be performed before or after the load. You can also note that the method #postloadForCore:package has two parameters. The pre/post load methods may take 0, 1 or 2 arguments. The *loader* (don't worry, we will explain later what it is) 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.

Another important thing that you should know is that these pre and post load sripts can be used not only in version methods but also in baselines. If the script depends on the version, then you can put it there. If it is likely it will not 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, I 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 preLoadDoIt: #preLoadForCoolBrowser.
    spec postLoadDoIt: #postLoadForCoolBrowser.
```

```
spec
   package: 'CoolBrowser-Core' with: [
      spec
         file: 'CoolBrowser-Core-MichaelJones.20';
         preLoadDoIt: #preloadForCore;
         postLoadDoIt: #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. We let to your imagination what those methods can do.

1.11 Platform specific package

Smalltalk has some differences to most common programming languages. A good package management system for Smalltalk, should take care about these differences and try to take advantage from them. And that's exactly what Metacello does. One of these differences is that there is no such Smalltalk as programming language, but there are different dialects. Each dialect is like a different implementation of Smalltalk, as you may know. So you have for example Pharo, Gemstone, Squeak, etc. This is good because each dialect tries to accomplish certain aims and you may find the smalltalk that better fits your needs. Of course, not everything is positive. There are differences between each dialect. And this make things complicated when you are developing software that has to run in different Smalltalk dialects. For such cases, it is very common to have a specific package that contains all the platform specific code. And then there will be one platform package per dialect.

For example, suppose that in our Cool Browser we have a package called CoolBrowser-Platform. There we can put some abstract classes, APIs, etc. And then, we can have the following packages: CoolBrowser-PlatformPharo, CoolBrowser-PlatformGemstone, etc. In this case we want our code from CoolBrowser to speak Òpolymorphicallyó to the objects of the platform.

Let me go a little off topic for a second. In most static or typed languages like Java for example, to achieve such a thing, we need not only to implement the same method in the different classes, but also

have a type in common. This type can be a super class (which can usually be abstract) or an interface. But in Smalltalk you get "polymorphism" just when two objects understand the same message. So, the package that we were talking about, CoolBrowser-Platform, may be even not necessary in Smalltalk. However, some people would rather create that package, and create classes and methods that work like an API. To do this, just describe what each method should do, and (most commonly) implement them all with *self subclassResponsibility*.

So...coming back to our example, the idea is that Metacello should automatically load the package of the platform where we are loading the code. But in order to do that, we need to give Metacello some information. See the following example:

```
ConfigurationOfCoolBrowser>>version09: spec
  <version: '0.9' imports: #('0.9-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' ].

  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'.].
```

```
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 unified way. Thus, we have a lot of flexibility. At runtime, when you load CoolBrowser, Metacello will automatically detect in which dialect that is happening and will load the specific package for that dialect.

Finally, note that the method #for:do: is not only used in this situation of a platform specific package, but also in anything that has to do with different dialects. You can put whatever you want from the configuration inside that method. So, for example, you can define groups, packages, repositories, etc, that are just depending on a dialect. For example, you can perfectly 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
      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 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 that for Gemstone. 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:#coommon:do:* can be inside another #for:do: for an specific dialect.

1.12 Project configurations references

We would need first to define what Project means. But doing this may require a complete book. So, let's define what a Project can be inside our world of software developing with Pharo and Metacello. A Project may include, among a lot of other things, pieces of software.

Those pieces of software can be represented as packages. Then we can think of a project like a set of packages. In our example, the project was the Cool Browser. Others projects may be Seaside, Refactoring Browser, Magma, or even big projects (complete Smalltalk dialects) like Pharo or Gemstone.

In the same way a package can depend on other packages, a project can depend on other projects. Actually, 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:

1. Without configuration reference: this is when a package A from a Project X depends on a package B from another project Y and the project Y has not any Metacello configuration. In such case, the way to deal with that is the following:

```
``In the baseline''
spec
   package: 'PackageA' with: [
      spec
         requires: #('PackageB')];
   package: 'PackageB' with: [
      spec repository: 'http://www.squeaksource.com/ProjectB' ].
```

```
``In the version''
package: 'PackageB' with: 'PackageB-JaunCarlos.80'.
```

   The problem here is that as the project B does not have a Metacello configurations, 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.

2. With configuration reference: when a package or project A depends on a package or project B, but project B has also a configuration. This is known as "Project configurations references", the purpose of this section.

To continue with our example, we introduce a new project called CoolToolSet which utilizes the packages from the CoolBrowser project. The configuration class is called ConfigurationOfCoolToolSet. We define two package in CoolToolSet called CoolToolSet-Core and CoolToolSet-Tests. Of couse, those 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 blessing: #beta.

    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 blessing: #baseline.
    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. 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.

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 keywords like 'ALL') to load from the project.

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.

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. 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 notice, 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 sometimes is not 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 differences versions of the dependent project. So, you have two possibilities: A) put a version as we

did but then when needed change it in the version method or; B) don't put the #versionString: '1.0' in the baseline and define that in the version method. We recommend B) solution.

For any of those cases, the version method looks the same way:

```
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 write the version in this way there is no difference between A) and B). The only difference may be if we don't define a version for 'CoolBrowser default' and 'CoolBrowser Tests' in the version method. In such case, in A) it will load the version specified in the baseline and in case B) it will load the last version (as always, when there is no specified version, Metacello downloads the last one).

1.13 Grouping projects

Usually, a project has more than one package and/or the package has dependencies upon other configurations. If such is the case it is common to create a separate configuration for all those packages or projects. With this, I can reuse dependency and version information among other projects.

Maybe the sole purpose of this configuration is to group packages or projects and define stable version for them so that they can be used in different projects.