

More Pharo by Example

with the participation of O. Nierstrasz, Dale, Mariano, Sven

Alexandre Bergel Damien Cassou Stéphane Ducasse Jannik Laval

Version of 2011-07-09

Contents

1	Preface	1
I	Frameworks	
2	Creating Browsers with OmniBrowser	5
2.1	Building a simple browser step by step	5
2.2	Enhancing the file browser with actions, definitions and icons.	9
2.3	Graph and Metagraph of a Browser	15
2.4	The OmniBrowser-based System Browser	19
2.5	Evaluation and Discussions	25
2.6	Chapter Summary	27
3	Glamour	29
3.1	Installation and first browser.	29
3.2	Tutorial: Implementing a code browser	32
4	Using LDAP	45
4.1	Présentation du protocole LDAP	45
4.2	Les principales classes	46
4.3	Présentation des classes principales	52
4.4	Conclusion	52
5	Mondrian	53
5.1	Introduction.	53
5.2	First visualization.	53

5.3	Visualizing the collection framework	54
5.4	Reshaping nodes	55
5.5	Multiple edges	57
5.6	Adding colors	59
5.7	More on colors	60
5.8	Popup view	61
5.9	Subviews	62
5.10	Forwarding events	64
5.11	Events	65
5.12	Interaction	66
5.13	Conclusion	67
6	DBXTalk	69
6.1	DBXTalk Driver Architecture	69
6.2	Installing DBXTalk	70

II Language

7	Handling exceptions	73
7.1	Ensuring execution	74
7.2	Handling non-local returns	75
7.3	Exception handlers	76
7.4	Error codes — don't do this!	77
7.5	Specifying which Exceptions will be Handled.	78
7.6	Signaling an exception	79
7.7	How breakpoints are Implemented	81
7.8	How handlers are found	82
7.9	Handling exceptions.	85
7.10	Resuming execution	87
7.11	Passing exceptions on	90
7.12	Resending exceptions	92
7.13	Comparing outer with pass	93
7.14	Catching sets of exceptions	93
7.15	How exceptions are implemented	94

7.16	Other kinds of Exception	97
7.17	When not to use Exceptions	99
7.18	Chapter Summary	100
8	Block and Dynamic Behavior of Smalltalk-Runtime	103
8.1	Basics	103
8.2	Variables and Blocks	104
8.3	Basics on Return	106
8.4	Returning from inside a block	106
8.5	Storing a block	108
8.6	may be to trash.	108
8.7	Lexical Closure.	109
8.8	Conclusion	111
9	Fun with Floats	115
9.1	Never test equality on floats	115
9.2	Dissecting a Float	116
9.3	With floats, printing is inexact	120
9.4	Float rounding is also inexact	121
9.5	Fun with Inexact representations	122
9.6	Conclusion	123
III	Web	
IV	Libraries	
10	Files with FileSystem	129
10.1	Getting started	129
10.2	Navigating the Filesystem.	130
10.3	Design	132
11	Announcements: an Object Dependency Framework	137
11.1	A word about Component Coupling	137
12	Sockets	141
12.1	Basic Concepts	141

12.2	A Simple TCP Client	143
12.3	A Simple TCP Server	146
12.4	SocketStream	151
12.5	A Basic Chat Application	152
12.6	Distributed Object Applications.	152
12.7	Chapter summary.	152
13	The Settings Framework	153
13.1	Settings in a Nutshell	153
13.2	The Settings Browser.	155
13.3	Declaring a setting	157
13.4	Organizing your settings	163
13.5	Providing more precise value domain	167
13.6	Launching a script	169
13.7	Setting styles management	171
13.8	Extending the <i>Settings Browser</i>	173
13.9	Conclusion	177
14	Regular Expressions in Pharo	179
14.1	Tutorial example—generating a site map	180
14.2	Regex syntax	187
14.3	Regex API	193
14.4	Implementation Notes by Vassili Bykov	199
14.5	Chapter Summary	199
V	Source Management	
15	Versioning your code with Monticello	203
15.1	Basic usage	204
15.2	Exploring Monticello repositories	217
15.3	Advanced topics	219
15.4	Getting a change set from two versions	225
15.5	Kinds of repositories.	226
15.6	The .mcz file format	230
15.7	Chapter Summary	231

16	Gofer: Scripting package loading	233
16.1	Preamble: Package management system.	233
16.2	What is Gofer?	236
16.3	Using Gofer	237
16.4	Gofer Actions	238
16.5	Conclusion	244
17	Managing projects with Metacello	245
17.1	Introduction	245
17.2	One tool for each job	246
17.3	Metacello features.	247
17.4	A Simple Case	247
17.5	Naming your configuration	250
17.6	Managing package internal dependencies	251
17.7	Baselining	253
17.8	Groups.	255
17.9	Project Configuration Dependencies	258
17.10	Pre and post code execution	262
17.11	Platform specific package	264
17.12	Symbolic Versions.	266
17.13	Load types	280
17.14	Conditional loading	281
17.15	Project version attributes	283
17.16	Conclusion	284

VI Tools

18	Optimizing Application	287
18.1	What does profiling mean?	287
18.2	A simple example.	288
18.3	Code profiling in Pharo.	289
18.4	Read and interpret the results	292
18.5	Illustrative Analysis	297
18.6	Counting messages	299

18.7	Memorized Fibonacci	299
18.8	SpaceTally for Memory Consumption per Class	301
18.9	Few advices	301
18.10	How MessageTally is implemented?	302
18.11	Chapter Summary	303

Chapter 1

Preface

Acknowledgments

Thanks to the following reviewers: Orla Greevy, Lukas Renggli.

Thanks to Vassili Bykov for permission to adapt his Regex documentation.

Part I

Frameworks

Chapter 2

Creating Browsers with OmniBrowser

with the participation of:
Alexandre Bergel (alexandre@bergel.eu)

In this chapter, we present OmniBrowser, a browser framework that supports the definition of browsers based on explicit metamodels. In the OmniBrowser framework, a browser is a graphical list-oriented tool to navigate and edit an arbitrary domain. The most common representative of this category of tools is the Smalltalk system browser, which is used to navigate and edit Smalltalk source code. In OmniBrowser, a browser is described by a domain model and a metagraph that specifies how the domain space may be navigated. Widgets, such as list menus and text panels, are used to display information gathered from a particular path in the metagraph. Although widgets are programmatically composed by the framework, OmniBrowser allows for interaction with the end user.

We will look at how to build new browsers from predefined parts and how to easily describe new tools. We will illustrate how to define sophisticated browsers for various domains with the help of three examples: a file browser, a remake of the ubiquitous Smalltalk system browser, and a coverage browser.

2.1 Building a simple browser step by step

To illustrate how to instantiate the OmniBrowser framework, we describe the implementation of a simple file browser supporting navigation of directories and files.

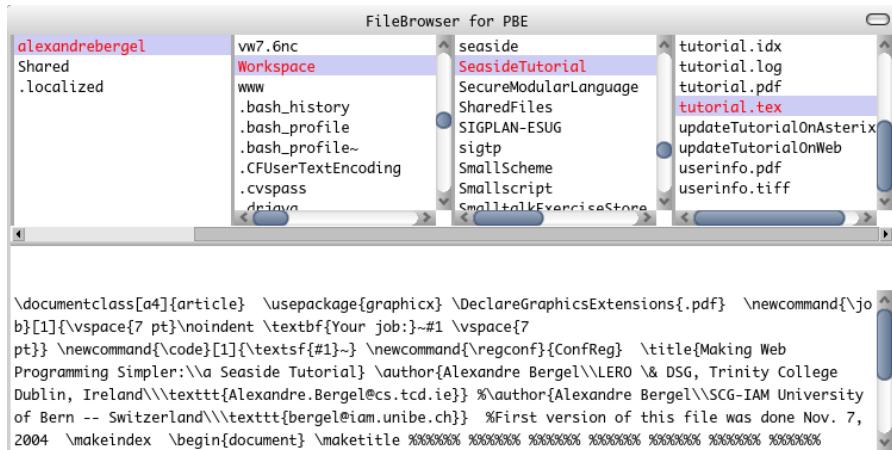


Figure 2.1: A minimal file browser based on OmniBrowser.

Figure 2.1 shows the file browser in action. The navigation columns in the case of a file browser are used to navigate through directories, where every column lists the contents of the directory selected in its left column, similar to the *Column View* of the Finder in the Mac OS-X operating system. Note that we can have an arbitrary number of panes navigating through the file system. The horizontal scrollbar lets the user browse the directory structure. A text panel below the columns displays additional properties of the currently selected directory or file and provides means to manipulate these properties.

The class describing a browser must inherit from the class OBBrowser. The class FileBrowser may be defined as follows:

```
OBBrowser subclass: #FileBrowser
instanceVariableNames: ""
classVariableNames: ""
poolDictionaries: ""
category: 'PBE-Omnibrowser'
```

The default browser title may be overridden by redefining the class method title:

```
FileBrowser class>title
↑ 'FileBrowser for PBE'
```

OmniBrowser uses a *metagraph* to model different navigation paths that a user may follow by clicking on browser items. To keep this example simple, we will assume that a file system contains only two kind of entities, files and

directories.

```
FileBrowser class»defaultMetaNode
    "returns the directory metanode that acts as the root metanode"

    | directory file |
    directory := OBMetaNode named: 'Directory'.
    file := OBMetaNode named: 'File'.
    directory
        childAt: #directories put: directory;
        childAt: #files put: file.
    ↑ directory
```

A metagraph is composed of metanodes, each of which represents one particular set of selected items. To model the navigation of a file system we thus need two metanodes in the metagraph, Directory and File. Within any directory of a file system, we can navigate to further files and other directories, hence we need two kinds of transitions out of a directory metanode, which will be labelled files and directories in the metagraph.

The next step is to create class nodes that will model browser items. For that purpose, we define two subclasses of OBNode:

```
OBNode subclass: #FileNode
    instanceVariableNames: 'path'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'PBE–Omnibrowser'

FileNode subclass: #DirectoryName
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'PBE–Omnibrowser'
```

Instances of FileNode andDirectoryName will represent files and directories of the domain being navigated. Nodes wrap objects of the browsed domain. First the class FileNode, a subclass of OBNode, has to be defined. Instances of this class will represent concrete files. A file node is identified by a full path name, stored in a variable. A directory is another entity in our model that contains directories and files. A directory can be simply modeled as a special kind of file. The only difference between a file and a directory node is that for a directory the path variable points to a directory, not to a file.

The path variable needs accessors:

```
FileNode»path
    ↑ path

FileNode»path: aString
```

```
path := aString
```

A node needs to answer to the message name and return the title used in the browser:

```
FileNode»name
  "return the name of a file"
  ↑ (self path subStrings: '/') last
```

When selected, a node may provide a character string used to fill the lower text pane. In our situation, clicking on a file node should display the content of the selected file. The method text has to be defined.

```
FileNode»text
  "return the first 1000 characters"
  ↑ ((FileStream readOnlyFileNamed: path) converter: Latin1TextConverter new;
    next: 1000) asString
```

Note that the method text provides a purely read-only view of the node. If we want to be able to edit the contents of a file, then we need to take a different approach, as we will see later on.

Each node is displayed in a column, and when selected, it provides the list of nodes used to fill the next column. Clicking on a directory must cause the list of contained files and folders to be displayed in the column located to the right. The transitions directories and files have to be defined as methods:

```
DirectoryNode»directories
| dir |
dir := FileDirectory on: path.
↑ dir directoryNames collect: [:each |
  DirectoryNode new path: (dir fullNameFor: each)]
```

```
DirectoryNode»files
| dir |
dir := FileDirectory on: path.
↑ dir fileNames collect: [:each |
  FileNode new path: (dir fullNameFor: each)]
```

The name directories and files are also the names of the transitions between the Directory and File meta nodes. When one of the two #directories and #files metaedges is traversed, the name of this metaedge is used as a message name sent to the metanode's node. The two methods are invoked when a directory node is selected.

The implementation of FileNode and DirectoryNode shows the two responsibilities of a node: rendering itself (implemented in the text method), and calculating the nodes reachable from a node (in the directories and files meth-

ods). As there is no further navigation leaving a file node, such a node does not have to define navigation methods such as directories or files.

The file browser is opened for a given directory, *e.g.*, the root directory of the file system. Thus the metagraph's root metanode represents a directory. The default root node is given by the class method `defaultRootNode`:

```
FileBrowser class»defaultRootNode
↑ DirectoryNode new path: '/'
```

Our file browser may now be opened by executing `FileBrowser open`. Currently, no much can be done. Only navigating through directories and displaying file contents. In the subsequent sections, we will see how commands and icons may be added.

2.2 Enhancing the file browser with actions, definitions and icons

Spawning browsers

A command defines how the user can interact with browsed items. All interactions triggered from a menu have to be defined as commands. We will define a browser command that opens a second file browser on a selected node. Figure 2.2 depicts the activation of the contextual menu.

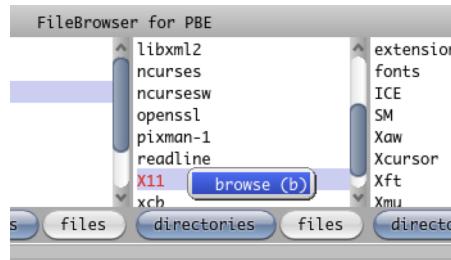


Figure 2.2: The `browse` command in the filer browser.

`BrowseCommand` has to be a subclass of `OBCommand` and must override at least 4 methods in order to be effective. Each command must have a menu item label.

```
BrowseCommand»label
↑ 'browse'
```

A command may or may not be active (*i.e.*, listed in the contextual menu) depending on some selection or the particular state of the selected item. In our case, we just need to redefine `isActive` to answer whether the node for which the command is triggered is selected. This implementation of `isActive` is very common and you will probably need it for most of your commands.

```
BrowseCommand»isActive
↑ (requestor isSelected: target)
```

When these methods are evaluated, the command already knows the column from which it gets triggered (stored in the instance variable `requestor`) and the target node for which the action has to be executed (stored in the instance variable `target`).

A command may be also invoked using a particular keystroke. The apple key (on Mac) or alt key (on MS Windows and Linux) has to be combined with the character returned by keystroke to activate the command.

```
BrowseCommand»keystroke
↑ $b
```

The shift key may be added to the keystroke by returning a capital letter instead of a minuscule one. Note that by defining a keystroke, the label is automatically appended with “(b)” in menus.

The method `execute` is evaluated when the command is active and triggered by either an appropriate keystroke or a menu selection.

```
BrowseCommand»execute
FileBrowser openOn: target.

FileBrowser class»openOn: aNode
↑ (self root: aNode selection: aNode) open
```

The method `openOn:` needs to be added to the class side of `FileBrowser` to open a browser on a given node.

The very last step is to link `FileBrowser` to `BrowseCommand`. This is done by defining an instance side method that returns the class command (and not an instance of it). This name of the method needs to begin with “cmd”, this is a naming convention similar to testing method names beginning with “test”. The instance side method needs to be defined:

```
FileBrowser»cmdBrowse
↑ BrowseCommand
```

The command is now fully defined. Open a new file browser and right click on an item.

Creating new files

Our browse command is very simple: there is no list to refresh; no interaction with the user; no automatic node selection. We will cover these important topics by adding a command for creating files. The browser column will have to be updated with a new entry (the file recently created), and the user will have to enter the name of the file to be created.

The NewFileCommand is define as follows:

```
OBCommand subclass: #NewFileCommand
instanceVariableNames: ""
classVariableNames: ""
poolDictionaries: ""
category: 'PBE–OmniBrowser'

NewFileCommand>keystroke
↑ $n

NewFileCommand>label
↑ 'new file'
```

The command should be activated when we right click either on a directory (to create the file in it), or on a column. This suggests a slightly more elaborate isActive method:

```
NewFileCommand>isActive
↑ ((requestor isSelected: target) and: [target isKindOf:DirectoryName])
or: [requestor selectedNode isNil]
```

The OmniBrowser framework provides a number of utility classes to ask for user input. These class are in the package OmniBrowser–Notifications. We will use OBTextRequest to ask for user input.

```
NewFileCommand>execute
| filename fullFilename stream nodeToSelect |
filename := OBTextRequest
    prompt: 'Please type the name of the file to create'
    template: "".
fullFilename := target path, FileDirectory pathNameDelimiterasString, filename.
stream := FileStream newFileNamed: fullFilename.
stream close.

nodeToSelect := target files detect: [:fileNode | fileNode name = filename].
requestor announce: (OBNodeCreated node: nodeToSelect).

self select: nodeToSelect with: requestor announcer.
```

When a new file is created, it is automatically selected. The last line of execute selects noteToSelect.

Modifying files

So far, we've only seen how to create empty files. To modify the content of a file, the browser needs to accept user input from the lower text pane and perform an action on it.

Let's subclass OBDefinition:

```
OBDefinition subclass: #FileDefinition
instanceVariableNames: 'fileNode'
classVariableNames: ''
poolDictionaries: ''
category: 'PBE–OmniBrowser'
```

The variable `fileNode` is intended to refer to the selected file node. This variable will have to be initialized when the definition is instantiated. We need accessors:

```
FileDefinition>fileNode
↑ fileNode

FileDefinition>fileNode: aFileNode
fileNode := aFileNode
```

As we saw previously, the content of the lower text pane is given by the text methods defined on `FileNode`. As we now use a definition, `FileNode>text` is now useless. You can safely remove it and define it on `FileDefinition` instead:

```
FileDefinition>text
"return the first 1000 characters"
↑ ((FileStream readOnlyFileNamed: fileNode path) converter: Latin1TextConverter
new;
next: 1000) asString
```

The message `text:` is sent to the OB definition when the user presses apple-s on Mac or alt-s on other platforms. When the content is “accepted”, we need to open the file, write the content provided by the user into it, and close the file:

```
FileDefinition>text: aText
"return the first 1000 characters"
(FileStream fileNamed: fileNode path)
nextPutAll: aText asString;
close.
↑ true
```

`FileDefinition` is currently not linked to the browser. To do so, define the method `definition` in `FileNode`:

```
FileNode»definition
↑ FileDefinition new fileNode: self
```

Each selected item can have its own definition.

Distinguishing folders from plain files

To visually distinguish files from directories when browsing a directory with our file browser, we will denote directories with a small icon. This section shows how to import new icons into Pharo, then how to associate icons to browsed items.

The first step is to integrate the icon itself into a Pharo image. In the class `OBMorphicIcons` you can see some pre-defined icons stored in methods such as `package`. To import an icon stored as an image (*e.g.*, as a GIF file), you can use this code:

```
| image stream |
image := ColorForm fromFileNamed: '/path/to/icon.gif'.
stream := WriteStream with: String new.
image storeOn: stream.
stream contents.
```

Inspect this whole code listing. In the inspector you see the definition of the color form for the icon. You can now install the content of this `ByteString` as a method in the method protocol `icons` of `OBMorphicIcons` in a method called `folder`. Make sure that you do not return the string, but the code within the string, so that if the method gets invoked a color form for the folder icon is returned. For example, the package icon is defined as:

```
OBMorphicIcons»package
↑(Form
extent: 14@14
depth: 32
fromArray: #( 4294960327 4289901233 ..... 4294960327)
offset: 0@0)
```

The list of numbers corresponds to the image bit encoding of the icon.

In the second step you can take this icon and display it in the columns for every directory. To achieve this, simply add a method `icon` to the class `DirectoryNode`:

```
DirectoryNode »icon
↑ #folder
```

If you do not have a .gif under hand, you may replace `#folder` by `#package`, since a package icon is available. The method `icon` is evaluated for every

element that is added to a column. If it answers a symbol, then the method of OBMorphicIcons with the same name is evaluated, answering the icon as a color form to be added on the left of the list element, *i.e.*, the directory name.

Another way to distinguish files from directories is to keep them separated in different tabs. Figure 2.2 illustrates this. This is done by simply adding a filter to the directory metanode when defining the metagraph:

```
FileBrowser class»defaultMetaNode
...
directory
    childAt: #directories put: directory;
    childAt: #files put: file;      "Change the period for a semi-column"
    addFilter: OBModalFilter new.  "Line to add"
...
...
```

The effect of adding a filter to a metanode is to let the user select the metaedge to follow. One button is created for each metaedge. Without any filter, all edges are used to compute of the next list as we previously saw.

Coloring and font selection

Item names in a menu may be colored and use font modifier such as italic and bold effect. This feature is useful for distinguishing some items from others or for emphasizing some items. Font particularization is achieved by returning instance of Text instead of a simple string when overriding OBNode >>name.

```
TreeNode»name
"return the name of a file"

↑ Text string: ((self path subStrings: '/') last) attribute: TextColor blue

DirectoryNode»name
"return the name of a directory"
↑ Text string: ((self path subStrings: '/') last) attribute: TextColor gray
```

Plain files are now listed in blue and directories in gray.

A number of emphasizes are available: bold, italic, underlined, and barred. The attribute corresponding to the emphasis is obtained by sending a message to the TextEmphasis class.

Attributes may be combined by being added to a text using addAttribute:. For example, the following name definition makes directory appears in gray and underlined:

```
DirectoryNode»name
↑ (Text fromString: ((self path subStrings: '/') last))
```

```
addAttribute: TextEmphasis underlined;
addAttribute: TextColor gray
```

2.3 Graph and Metagraph of a Browser

The remainder of this chapter reviews the notions introduced so far, but in a more formal fashion. We describe the internals of the framework, which should help the reader to understand when OmniBrowser is appropriate for a given task and what are its strengths and limitations.

The OmniBrowser framework is structured around (i) an explicit domain model and (ii) a metagraph, a state machine, that specifies the navigation and interaction with the domain model. The user interface is constructed by the framework, and uses a layout similar to the code browser, with two horizontal parts. The top part is a column-based section which supports navigation. The bottom half is a text pane.

Overview of the OmniBrowser framework

The major classes that make up the OmniBrowser framework are presented in Figure 2.3, and explained briefly in the rest of this section.

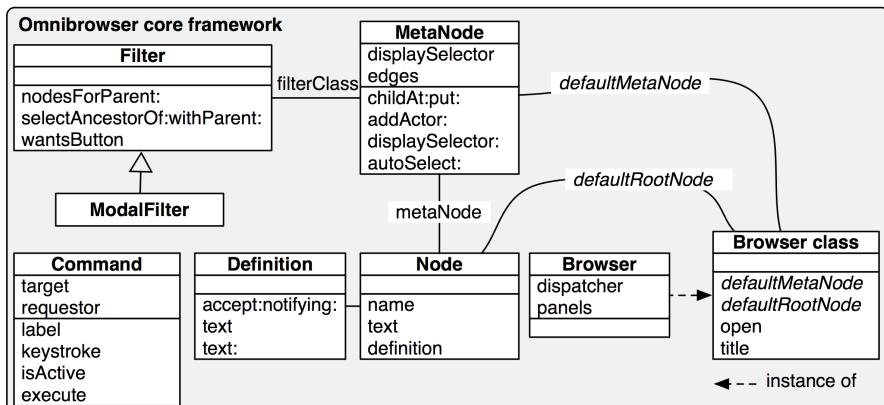


Figure 2.3: Core of the OmniBrowser framework.

Browser. A *browser* is a graphical tool to navigate and edit a domain space. This domain has to be described in terms of a directed cyclic graph (DCG). It is cyclic because, for example, file systems or structural meta models of

programming languages (*i.e.*, packages, classes, methods...) contain cycles, and we need to be able to model these. The domain graph has to have an entry point, its root. The path from this root to a particular node corresponds to a state of the browser defined by a particular combination of user actions (such as menu selections or button presses). The navigation of this domain graph is specified in a *metagraph*, a state machine describing the states and their possible transitions.

Node. A *node* is a wrapper for a domain object, and has two responsibilities: rendering the domain object, and returning domain nodes.

Metagraph. A browser's *metagraph* specifies the way a user traverses the graph of domain nodes. A metagraph is composed of metanodes and metaedges. A metanode identifies a state in which the browser may be. A metanode may reference a filter (described below) The metanode does not have the knowledge of the domain nodes, however each node is associated to a metanode. Transitions between metanodes are defined by metaedges. When a metaedge is traversed (*i.e.*, result of pressing a button or selecting an entry list), sibling nodes are created from a given node by invoking a method that has the name of the metaedge.

A *metanode* has the ability to be auto selected with the method `MetaNode»autoSelect: aMetaNode`. When a particular child for auto selection is designated, the first node produced by following its metaedge will be selected.

Command. A *command* enables interaction with and manipulation of the domain graph. Commands may be available through menus and buttons in the browser. They therefore have the ability to render themselves in a user interface and are responsible for handling exceptions that may be raised when they are executed.

Commands are defined in a non-invasive way: commands are added and removed without redefining any method of the core framework. This enables a smooth gathering of independently realized commands.

Filter. The metagraph describes a state machine. When the browser is in a state in which more than one transition is available, the user decides which transition to follow. To allow that to happen OmniBrowser displays the possible transitions to the user. From all the possible transitions, the OmniBrowser framework fetches all the nodes that represent the states the user could arrive at by following those transitions and list them in the next column. Note that the transition is not actually made yet, and the definition pane is still displaying the current definition. Once an item selected, the transition actually occurs, the definition pane is updated (and perhaps other panes

such as button bars), and OmniBrowser gathers the next round of possible transitions.

A *filter* provides a strategy for filtering out some of the nodes from the display. If a node is the starting point of several edges, a filter may be needed to filter out all but one edge to determine which path has to be taken in the metagraph.

Definition. While navigating in the domain space, information about the selected node is displayed in a dedicated textual panel. If the text is expected to be edited by the browser user, then a *definition* is needed to handle modification and commitment (*i.e.*, an *accept* in the Smalltalk terminology). A definition is produced by a node.

Core Behavior of the Framework

The core of the OmniBrowser framework is composed of 8 classes (Figure 2.3).

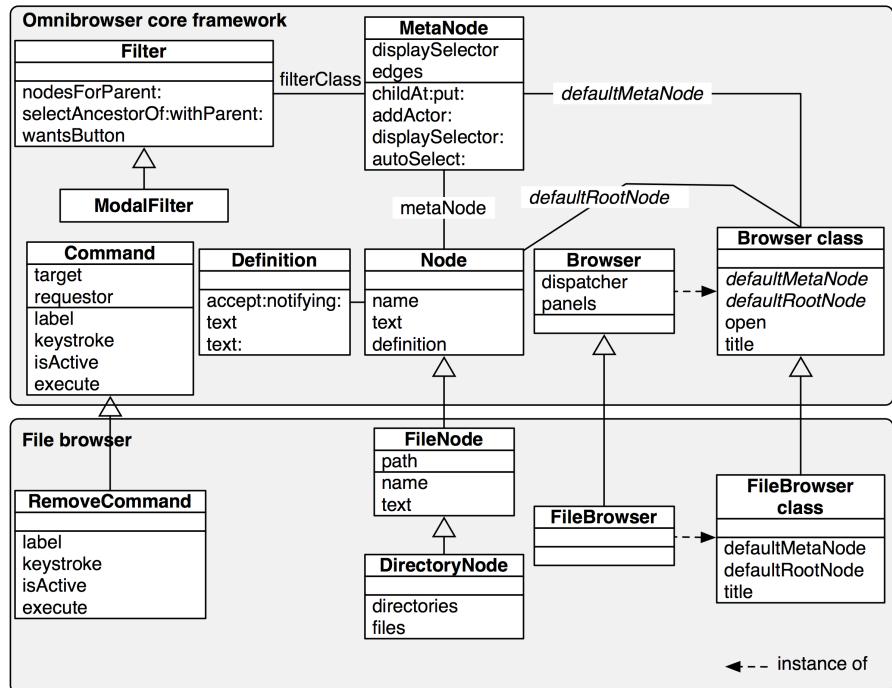


Figure 2.4: Core of the OmniBrowser framework and its extension for the file browser.

The metaclass of the class OBBrowser is OBBrowser class. It defines two abstract methods `defaultMetaNode` and `defaultRootNode`. These methods are abstract, they therefore need to be overridden in subclasses. These methods are called when a browser is instantiated. The methods `defaultMetaNode` and `defaultRootNode` return the root metanode and the root domain node, respectively. As we already saw, a browser is opened by sending the message `open` to an instance of the class OBBrowser.

The navigation graph is built with instances of the class OBMetanode. Transitions are built by sending the message `childAt: selector put: metanode` to a Metanode instance. This has the effect to create a metaedge named `selector` leading away the metanode receiver of the message and metanode.

At runtime, the graph traversal is triggered by user actions (*e.g.*, pressing a button or selecting a list entry) which send the metaedge's name to the node that is currently selected. The rendering of a node is performed by invoking on the domain node the selector stored in the variable `displaySelector` in the metanode.

The class OBCmd is instantiated by the framework and the set of commands for a browser is discovered (through the Smalltalk reflection API) when a browser is instantiated. All methods starting with the `cmd` prefix are considered to be commands. Each of this method should return the *class* of the command (and not an instance of it).

When the browser is in a state where several transitions are available, it displays the navigation possibilities to the user. From all the possible transitions, the OmniBrowser framework fetches all the nodes that represent the states the user could arrive at by following those transitions and lists them in the next column. Once a selection is made, the transition actually occurs, the pane definition is updated and the process repeats.

As explained before, a filter or modal filter can be used to select only a number of outgoing edges when not all of them need to be shown to the user. This is useful for instance to display the instance side, comments, or class side of a particular class in the classic standard system browser (cf. Section 2.4). Class OBFilter is responsible for filtering nodes in the graph. The method `nodesForParent:` computes a transition in the domain metagraph. This method returns a list of nodes obtained from a given node passed as argument. The class OBFilter is subclassed into OBModalFilter, a handy filter that represents transitions in the metagraph that can be traversed by using a radio button in the GUI.

Glueing Widgets with the Metagraph

From the programmer point of view, creating a new browser implies defining a domain model (set of nodes like `TreeNode` and `DirectoryNode`), a meta-

graph intended to steer the navigation and a set of commands to define interaction and actions with domain elements. The graphical user interface of a browser is automatically generated by the OmniBrowser framework. The GUI generated by OmniBrowser framework is contained in one window, and it is composed of 4 kinds of widgets (lists, radio buttons, menus and text panes).

Lists. Navigation in OmniBrowser framework is rendered with a set of lists and triggered by selecting one entry in a list. Lists displayed in a browser are ordered and are displayed from left to right. Traversing a new metanode, by selecting a node in a list A , triggers the construction of a set of nodes intended to fill a list B . List B follows list A .

The root of a metagraph corresponds to the left-most list. The number of lists displayed is equal to the depth of the metagraph. The depth of the system browser metagraph (as it will see later on in Figure 2.7) is 4, therefore the system browser has 4 lists (Figure 2.5). Because the metagraph of a file system contains a cycle (remember `directory childAt: #directories put: directory` in the file browser metagraph in Section 2.1), the number of lists in the browser increases for each directory selected in the right-most list. Therefore a horizontal scrollbar is used to keep the width of the browser constant, yet displaying a potentially infinite number of lists in the top half.

Radio buttons. A modal filter in the metagraph is represented in the GUI by a radio button. Each edge leading away from the filter is represented as a button in the radio button. Only one button can be selected at a time in the radio button, and the associated choice is used to determine the outgoing edges. For example, the second list in the system browser contains the three buttons `instance`, `?` and `class` as shown the transition from the environment to the three metanodes `class`, `class comment` and `metaclass` in Figure 2.5.

Menus. A menu can be displayed for each list widget of a browser. Typically such a menu displays a list of actions that can be evaluated by the user. These actions enable interaction with the domain model, however they do not allow further navigation in the metagraph.

2.4 The OmniBrowser-based System Browser

In this section we show how the framework is used to implement the traditional class system browser.

The Smalltalk System Browser

The system browser is probably the most important tool offered by the Pharo programming environment. It enables code navigation and code editing. Figure 2.5 shows the graphical user interface of this browser, and how it appears to the Smalltalk programmer.



Figure 2.5: OmniBrowser based Smalltalk system browser.

The system browser is mainly composed of four lists (upper part) and a panel (lower part). From left to right, the lists represent (i) categories, (ii) classes contained in the selected category, (iii) method categories defined in the selected class to which the -- all -- category is added, and (iv) the list of methods defined in the selected method category. On Figure 2.5, the class named `Class`, which belongs to the category `Kernel-Classes` is selected. `Class` has three methods categories, plus the -- all -- one. The method `templateForSubclassOf:category` contained in the instance creation method category is selected.

The lower part of the system browser contains a large textual panel displaying information about the current selection in the lists. Selecting a category triggers the display of a class template intended to be filled out to create a new class in the system. If a class is selected, then this panel shows the definition of this class. If a method is selected, then the definition of this method is displayed. The text contained in the panel can be edited. The effect of this is to create a new class, a new methods, or changing the definition of a class (e.g., adding a new variable, changing the superclass) or redefining a method.

In the upper part, the class list contains three buttons (titled `instance`, `?` and `class`) to let one switch between different “views” on a class: the class

definition, its comment and the definition of its metaclass. Just above the definition panel, there is a toolbar intended to open more specific browsers like a hierarchy browser or a variable access browser.

The -- all -- method category gets automatically selected when no other method category is selected. This is specified in the `OBMetagraphBuilder» populateClassName` method by invoking `autoSelect: aMetanode`.

System Browser Internals

The OmniBrowser-based implementation of the Pharo system browser is composed of 17 classes (2 classes for the browser, 3 classes for the definitions of classes, methods and organization, 10 classes defining nodes and 2 utility classes with abstractions to help link the browser and the system). Figure 2.6 shows the classes in OmniBrowser framework that need to be subclassed to produce the system browser. Note that the two utility classes are not represented on the picture.

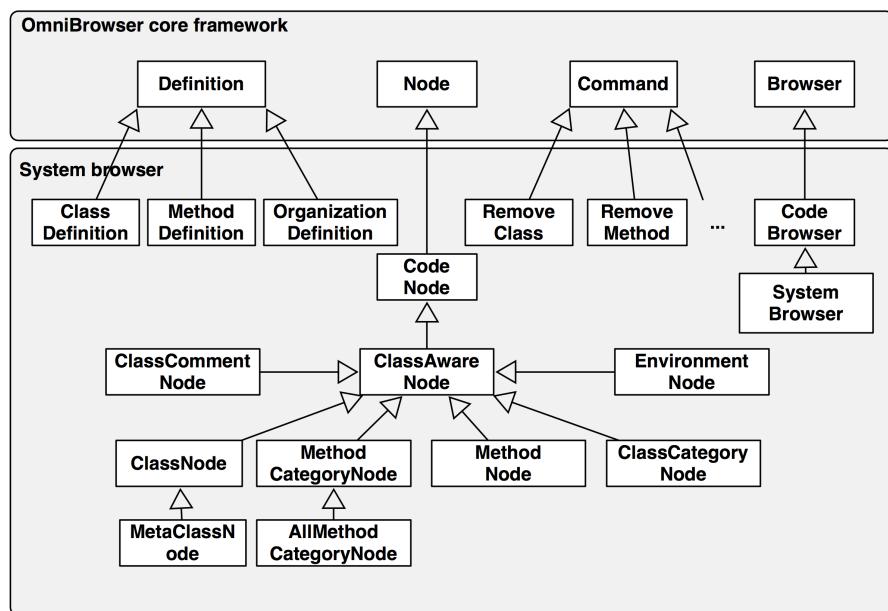


Figure 2.6: Extension of the OmniBrowser framework to define the system browser.

Compared to the default implementation of the Pharo System Browser this is less code and better factored. In addition other code-browsers can freely reuse these parts.

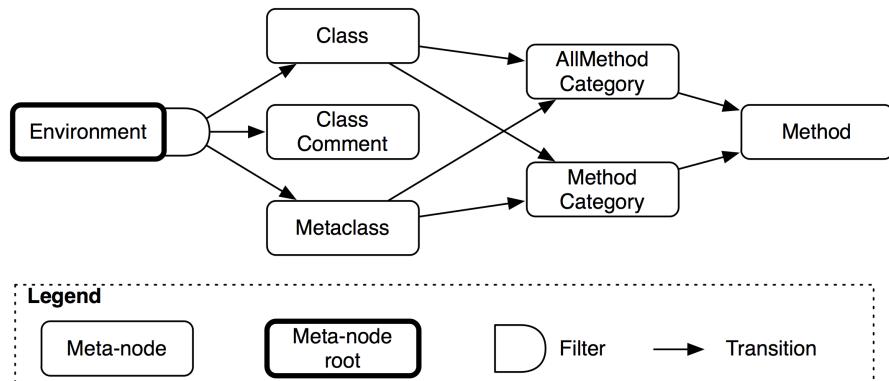


Figure 2.7: Metagraph of the system browser.

Figure 2.7 depicts the metagraph of the system browser. The metanode environment contains information about categories. The filter is used to select what has to be displayed from the selected class (*i.e.*, the class definition, its comment or the metaclass definition). A class and a metaclass have a list of method categories, including the `-- all --` method category that shows a list of all methods.

As in the file browser example, we implement a method `defaultMetaNode` on the class side of the browser class, *i.e.*, `OBSSystemBrowser`, returning the root metanode of the metagraph. This method reads:

```

OBSSystemBrowser class»defaultMetaNode
| env classCategory |
env := OBMetaNode named: 'Environment'.
classCategory := OBMetaNode named: 'ClassCategory'.
env childAt: #categories put: classCategory.
classCategory ancestrySelector: #isDescendantOfClassCat:.
self buildMetagraphOn: classCategory.
^env
  
```

There is a dedicated utility class called `OBMetagraphBuilder` to create the complex metagraph of the system browser. The method `defaultMetaNode` outsources most parts of the metagraph building to this class. `OBMetagraphBuilder` implements its functionality in several small methods, *i.e.*, for every metanode of the metagraph there is a method holding all code to create this metanode and the outgoing edges, hence it is easily possible to adapt the metagraph by providing a dedicated subclass overriding the appropriate methods to change the right metanodes.

The root node of the domain graph is answered by the method `defaultRootNode`. For the system browser, the root node is the environment

node:

```
OBSystemBrowser class»defaultRootNode
  ↑OBEnvironmentNode forImage
```

Filtering of nodes. In the metagraph we can also define several filters for a metanode, used to filter and otherwise manipulate the nodes represented by this metanode before they get displayed in columns. For the category metanode, for instance, there are two filters defined: a class sort filter and the modal filter used to select one of the three outgoing metaedges instance, comment or class. The modal filter was introduced earlier in the chapter.

Let's have a look at these two filters, starting with the class sort filter implemented in class OBClassSortFilter. Its responsibility is to sort and indent all classes of a category according to their position in a class hierarchy. If a category for instance contains two distinct class hierarchies, *e.g.*, class C inherits from B, and B and D inherit from A, and E has two subclasses F and G, then the class sort filter sorts and indents these classes as shown in Figure 2.8.

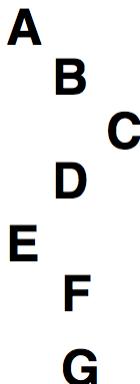


Figure 2.8: How OBClassSortFilter sorts and indents two distinct class hierarchies in one category.

When a metanode is asked for its children nodes (in method `childrenForNode: aNode`) it asks its associated filters to answer the nodes by invoking their `nodesFrom: aCollection forNode: aNode` method. In the case of the class sort filter, `aNode` refers to the category node and `aCollection` holds all class nodes this category node returns when the message `classes` is sent to it. The class sort filter can now sort the passed class nodes and indent them appropriately in the method `OBClassSortFilter»nodesFrom:forNode:..`

The other filter defined for a category metanode, `OBModalFilter`, has a different task: It selects one edge of the three outgoing edges from the category metanode, *i.e.*, instance, comment or class. The user of the system browser can select using the switch in the class column whether he wants to see the instance-, the class-side or the comment of the selected class. `OBModalFilter` remembers the selection of the user. Dependent on this selection, it answers the corresponding metaedge to be traversed, *e.g.*, the comment metaedge. This is done in the method `edgesFrom: aCollection forNode: aNode`. The metanode, *i.e.*, the category metanode, passes all available metaedges to this method, along with the currently selected class node, and the modal filter answers just the metaedge selected by the user. Other filters than a modal filter, such as the class sort filter, typically just return all edges passed to them.

There are two other important tasks performed by filters besides filtering edges and nodes: Manipulating the name of a node to be displayed and defining an icon shown along with a node in the column. The former is handled in the method `displayString: aString forParent: pNode child:`, the latter in `icon: aSymbol forNode: aNode`. Before a node's name gets displayed, all defined filters can manipulate the display of its name, *e.g.*, emphasize it in bold. Note that the filter also has access to the parent of a node to be displayed, not the current node alone. There are also filters enriching a node with an icon before display, the `OBInheritanceFilter` for instance adds arrow up, down icons to methods, if a method overrides a method with the same name from a super class or is overridden in subclasses.

A metanode can have arbitrarily many filters, resulting in a chain of filters. However, if several filters do the same kind of task, *e.g.*, adding an icon to a node, the last added filter providing this functionality will finally be responsible to define the icon which the node gets. Hence the order in which the filters get added to the metanode is relevant.

Widget notification. Widgets like menu lists and text panels interact with each other by triggering events and receiving notifications. Each browser has a dispatcher (referenced by the variable `dispatcher` in the class `Browser`) to conduct events passing between widgets of a browser. The vocabulary of events is the following one:

- `refresh` is emitted when a complete refresh of the browser is necessary. For instance, if a change happens in the system, this event is triggered to trigger a complete redraw.
- `nodeSelected` is emitted when a list entry is selected with a mouse click.
- `nodeDeleted` is emitted when a list entry has been removed, *e.g.*, by executing a remove command.

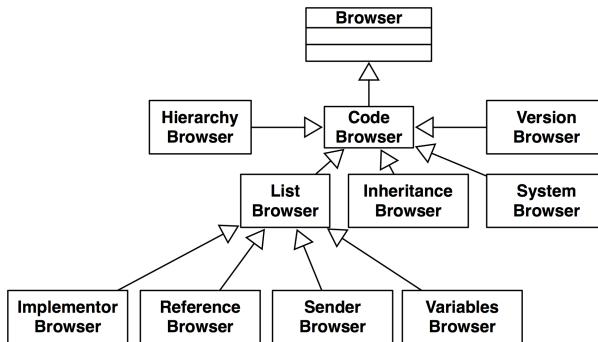


Figure 2.9: Some code browsers developed using OmniBrowser framework.

- `nodeChanged` is emitted when the node that is currently displayed changes. This typically occurs when a filter button related to the class is selected. For example, if a class is displayed, pressing the button `instance`, `class` or `comment` triggers this event.
- `okToChangeNode` is emitted to prevent losing some text edition while changing the content of a text panel if this was modified without being validated. This happens when a user writes the definition of a method, without accepting (*i.e.*, compiling) it, and then selects another method.

Each graphical widget composing a browser is a listener and can emit events. Creation and registration of widgets as listeners and event emitters is completely transparent to the end user.

State of the browser. Contrary to the original Pharo system browser where each widget state is contained in a dedicated variable, the state of a OmniBrowser framework-based browser is defined as a path in the metagraph starting from the root metanode. Each metanode taking part in this path is associated to a domain node. This preserves the synchronization between different graphical widgets of a browser.

2.5 Evaluation and Discussions

Several other browsers supporting new language constructs such as Traits have been developed using OmniBrowser, which demonstrate that the framework is mature and extensible. Figure 2.9 shows some browsers that are based on OmniBrowser. We now discuss the strengths and limitations of the framework.

Strengths

Ease of use. The fact that the browser navigation is explicitly defined in one place lets the programmer easily understand and control the tool navigation and user interaction. The programmer does not have the burden to explicitly create and glue together the UI widgets and their specific layout. To add additional custom widgets in a concrete browser, the developer can simply define a class implementing this widget and add an object of this class to the list of widgets used during the creation of the browser. This list is defined on the class-side of OBBrowser in the method panels. Still the programmer focuses on the key domain of the browser: its navigation and the interaction with the user.

Explicit state transitions. Maintaining coherence among different widgets and keeping them synchronized is a non-trivial issue that, while well supported by GUI frameworks, is often not well used. For instance, in the original Pharo browser, methods are scattered with checks for nil or 0 values. For instance, the method `classComment: aText notifying: aPluggableTextMorph`, which is called by the text pane (F widget) to assign a new comment to the selected class (B widget), is:

```
theClass := self selectedClassOrMetaClass.
theClass
ifNotNil: [ ... ]
```

The code above copes with the fact that when pressing on the class comment button, there is no guarantee that a class has been selected. In a good UI design, the comment class button should have been disabled, however there are still checks done whether a class is selected or not. Among the 438 accessible methods in the non OmniBrowser-based Pharo class Browser, 63 of them invoke `ifNil:` to test whether a list is selected or not and 62 of them send the message `ifNotNil:.` Those are not isolated Smalltalk examples. The code that describes some GUI present in the JHotDraw framework also contains the pattern checking for a nil value of variables that may reference graphical widgets.

Such a situation does not occur in OmniBrowser framework, as metagraphs are declaratively defined, and each metaedge describes an action the user can perform on a browser, states a browser can be in are explicit and fully described.

Separation of domain and navigation. The domain model and its navigation are fully separated: a metanode does not and cannot have a reference to the domain node currently selected and displayed. Therefore both can be reused independently.

Limitations

Hardcoded flow. As any framework, the OmniBrowser framework constrains the space of its own extension. The OmniBrowser framework does not support well the definition of navigation that does not follow the strict left to right list construction (the result of the selection creates a new pane to the right of the current one and the text pane is displayed). For example, building a browser such as Whiskers that displays multiple methods at the same time would require to deeply change the text pane state to keep the status of the currently edited methods.

2.6 Chapter Summary

This chapter presents the construction of a very simple browser based on the OmniBrowser framework. It also presents the design and implementation of a complex tool, the Smalltalk system browser.

- A browser is implemented by subclassing the class OBBrowser.
- All the navigation permitted by the browser for a given domain is defined with a metagraph.
- A browser's metagraph is returned by the defaultMetaNode class-side method.
- A metagraph is made of instances of OBMetaNode linked each other by sending the message childAt: aName put: aMetaNode. The symbol aName is then used as a message sent to a node to obtain the nodes to populate the following node.
- Domain nodes are implemented by subclassing OBNode.
- The root domain node is returning from a browser by sending the defaultRootNode class-side message.
- An action the user can perform is implemented with a command.
- A command must subclass OBCommand and the methods label, isActive, keystroke and execute are usually overridden.
- To enable a user to edit the content of the lower text pane, a definition needs to be defined by subclassing OBDefinition.
- A definition answers to text to returns a textual content and text: to accept a new content.

- A node needs to answer to the definition message to enables the user to modify its textual content.
- An icon may be added to a node by overriding OBNode>icon.
- Filters may be added to introduce panes in order to split the list of elements.

Acknowledgment. Colin Putney developed the original Omnibrowser. A number of contributions were made by the Pharo community. In particular, we gratefully acknowledge David Röthlisberger for his devotion and valued work on the Pharo tools.

Chapter 3

Glamour

Browsers are a crucial instrument to understand complex systems or models. Each problem domain is accompanied by an abundance of browsers that are created to help analyze and interpret the underlying elements. The issue with these browsers is that they are frequently rewritten from scratch, making them expensive to create and burdensome to maintain. While many frameworks exist to ease the development of user interfaces in general, they provide only limited support to simplifying the creation of browsers.

Glamour is a dedicated framework to describe the navigation flow of browsers. Thanks to its declarative language, Glamour allows to quickly define new browsers for their data.

In this chapter we will first detail the creation of some example browsers to have an overview of the Glamour framework. In a second part, we will describe Glamour in more details.

3.1 Installation and first browser

To install Glamour on your Pharo image execute the following code:

```
Gofer new
    squeaksource: 'Glamour';
    package: 'ConfigurationOfGlamour';
    load.
(Smalltalk at: #ConfigurationOfGlamour)
    perform: #loadDefault.
```

Now that Glamour is installed, we are going to build a first browser in order to dive into Glamour's declarative language. What about building an Apple's Finder-like file browser? This browser is built using the Miller

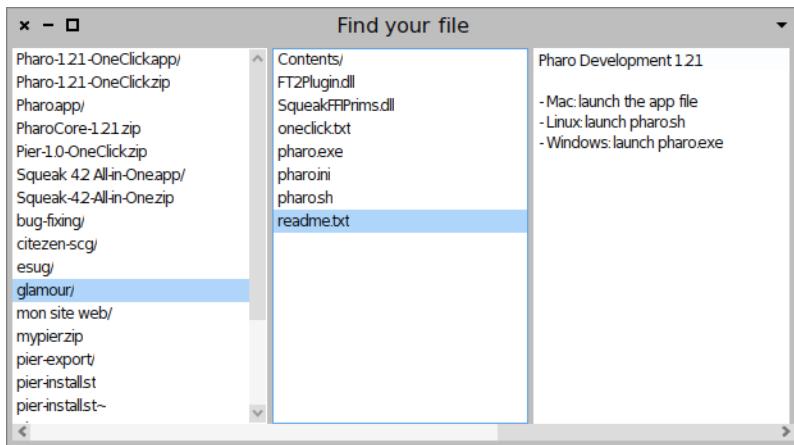


Figure 3.1: File finder as a Glamour implementation.

Columns browsing technique, displaying hierarchical elements in a series of columns. The principle of such a browser is that a column always reflects the content of the element selected in the previous column, the first column-content being chosen on opening.

In our case of implementing a file browser, we want to display a list of a particular directory's entries (each files and directories) in the first column and then, depending on the user selection, appending another column (see Figure 3.1):

- if the user selects a directory, the next column will display the entries of that particular directory;
- if the user selects a file, the next column will display the content of the file.

This may look complex at first. However, Glamour provides a very simple way of describing Miller Columns-based browsers. Glamour calls that kind of browsers finders, referring to the Apple's Finder found on Mac OS X. To create such a browser, we are going to use the `GLMFinder` class and then tell Glamour that we want elements to be in a list:

```
|browser|
browser := GLMFinder new.
browser list
    display: #children.
browser openOn: FSFilesystem onDisk root.
```

From this small piece of code you get a list of all entries (either files or directories) found at the root of your file system, each line representing either a file or a directory. If you click on a directory, you can see the entries of this directory in the next column. The filesystem navigation facilities are provided by the Filesystem framework, thoroughly discussed in Chapter 10.

This code has some problems however. Each line displays the full print string of the entry and this is probably not what you want. A typical user would expect only names of each entry. This can easily be done by customizing the list:

```
browser list
display: #children;
format: #basename.
```

This way, the message basename will be sent to each entry to get its name. This makes the files and directory much easier to read.

Another problem is that the code does not distinguish between files and directories. If you click on a file, you will get an error because the browser will send it the message children that it does not understand. To fix that, we just have to avoid displaying a list of contained entries if the selected element is a file:

```
browser list
when: #isDirectory;
display: #children;
format: #basename.
```

This works well but the user can't distinguish between a line representing a file or a directory. This can be fixed by, for example, adding a slash at the end of the file name if it is a directory:

```
browser list
when: #isDirectory;
display: #children;
format: #basenameWithIndicator.
```

The last thing we might want to do is to display the contents of the entry if it is a file. The following gives the final version of the file browser:

```
|browser|
browser := GLMFinder new
variableSizePanes;
title: 'Find your file';
yourself.
```

```
browser list
when: #isDirectory;
```

```

display: [:each | [each children sorted]
on: Exception
do: [Array new]];
format: #basenameWithIndicator.

browser text
when: #isFile;
display: [:entry | [entry readStream contents]
on: Exception
do: ['Can''t display the content of this file']].
browser openOn: FSFilesystem onDisk root.

```

This code extends the previous one with variable-sized panes, a title as well as directory entry sorting, access permission handling and file content reading. The resulting browser is presented in Figure 3.1.

This short introduction has just presented how to install Glamour and how to use it to create a simple file browser.

3.2 Tutorial: Implementing a code browser

The following will present a more complete example as well as detailed information about the framework.

Running example

In the following tutorial we will be creating a simple Smalltalk class navigator. Such navigators are used in many Smalltalk browsers and usually consist of four panes, which are abstractly depicted in figure 3.2.

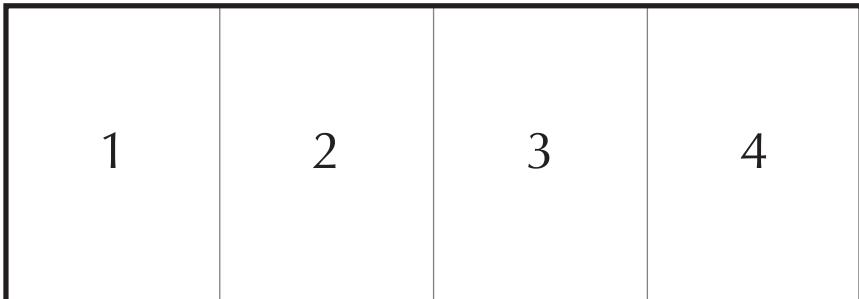


Figure 3.2: Wireframe representation of a Smalltalk class navigator.

The class navigator functions as follows: Pane 1 shows a list or a tree

of *packages* (containing classes) which make up the organizational structure of the environment. When a package is selected, pane 2 shows a list of all classes in the selected package. When a class is selected, pane 3 shows all *protocols* (a construct to group methods also known as method categories) and all methods of the class are shown on pane 4. When a protocol is selected in pane 3, only the subset of methods that belong to that protocol are displayed on pane 4.

Starting the Browser

We build the browser iteratively and gradually introduce new constructs of Glamour. To start with, we simply want to open a new browser on the list of packages. Because the example is going to involve more code than the previous file browser, we are going to implement the code browser in a dedicated class.

The first step is then to create the class with some initial methods:

```
Object subclass: #PBE2CodeNavigator
instanceVariableNames: 'browser'
classVariableNames: ""
poolDictionaries: ""
category: 'PBE2–CodeBrowser'
```

```
PBE2CodeNavigator class>>open
↑ self new open
```

```
PBE2CodeNavigator>>open
self buildBrowser.
browser openOn: self organizer.
```

```
PBE2CodeNavigator>>organizer
↑ RPackageOrganizer default
```

```
PBE2CodeNavigator>>buildBrowser
browser := GLMTabulator new.
```

Executing `PBE2CodeNavigator open` opens a new browser with the text “`a RPackageOrganizer`” and nothing else. We now extend our browser by adding a pane to display a list of packages.

```
PBE2CodeNavigator>>buildBrowser
browser := GLMTabulator new.
browser
column: #packages.
```

```
browser transmit to: #packages; andShow: [:a | self packagesIn: a].
```

```
PBE2CodeNavigator>>packagesIn: constructor
constructor list
display: [:organizer | organizer packageNames sorted];
format: #asString
```

In Glamour browsers are composed in terms of *panes* and the *flow of data* between them. In our browser we currently have only one pane displaying packages. The flow of data is specified by means of *transmissions*. These are triggered when certain changes occur, such as the change of the selection in a list. To exemplify this, we extend our browser with a list of classes for the currently selected package (see Figure 3.3).

```
PBE2CodeNavigator>>buildBrowser
browser := GLMTabulator new.
browser
column: #packages;
column: #classes.

browser transmit to: #packages; andShow: [:a | self packagesIn: a].
browser transmit from: #packages; to: #classes; andShow: [:a | self classesIn: a].
```

```
PBE2CodeNavigator>>classesIn: constructor
constructor list
display: [:packageName | (self organizer packageName: packageName)
definedClasses]
```

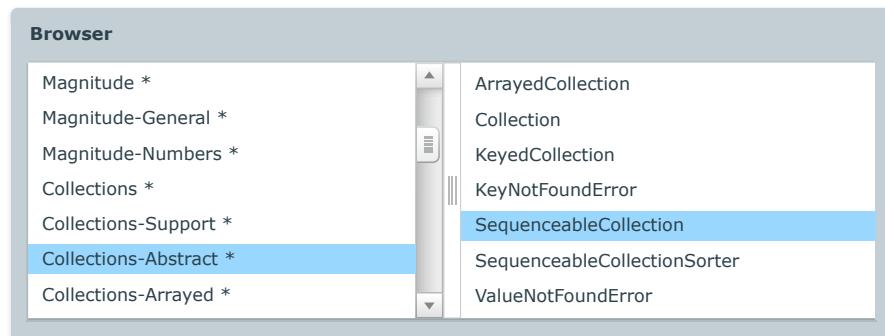


Figure 3.3: Two-pane browser. When a package is selected in the left pane, the contained classes are shown on the right pane.

The listing above shows almost all of the core language constructs of Glamour. Since we want to be able to reference the panes later, we give them the distinct names “*packages*” and “*classes*” and arrange them in columns using the `column:` keyword. Similarly, a `row:` keyword exists with which panes can be organized in rows.

The `transmit:`, `to:` and `from:` keywords create a *transmission*—a directed connection that defines the flow of information from one pane to another. In this case, we create a link from the *packages* pane to the *classes* pane. The `from:` signifies the *origin* of the transmission and `to:` the *destination*. If nothing more specific is stated, Glamour assumes that the origin refers to the *selection* of the specified pane. We show how to specify other aspects of the origin pane and how to use multiple origins below.

Finally, the `andShow:` specifies what to display on the destination pane when the connection is activated or *transmitted*. In our example, we want to show a list of the classes that are contained in the selected package.

`display:` simply stores the supplied block within the presentation. The blocks will only be evaluated later, when the presentation should be displayed on-screen. If no explicit `display` block is specified, Glamour will attempt to display the object in some generic way. In the case of list presentations, this means that the `displayString` message will be sent to the object to retrieve a standard string representation. `format:` can be used to change this default behavior.

Along with `display:`, it is possible to specify a `when:` condition to limit the applicability of the connection. By default, the only condition is that an item is in fact selected, *i.e.*, that the `display` variable argument is not `nil`.

Another Presentation

Up to now, we have been displaying the packages as a list. The packages in Smalltalk, however, are actually organized in a hierarchy and we have only been looking at the first level of this structure. To mend this, we replace the list by a tree presentation for packages:

```
PBE2CodeNavigator>>packagesIn: constructor
constructor tree
    display: [ :organizer | (self rootPackagesOn: organizer) asSet sorted ];
    children: [ :rootPackage :organizer | (self childrenOf: rootPackage on: organizer)
        sorted ];
    format: #asString

PBE2CodeNavigator>>classesIn: constructor
constructor list
    when: [:packageName | self organizer includesPackageName: packageName ]
    display: [:packageName | (self organizer packageName: packageName)
        definedClasses];

PBE2CodeNavigator>>childrenOf: rootPackage on: organizer
↑ organizer packageNames select: [ :name | name beginsWith: rootPackage , '-' ]

PBE2CodeNavigator>>rootPackagesOn: organizer
```

```
↑ organizer packageNames collect: [ :string | string readStream upTo: $- ]
```

The tree presentation uses a `children:` argument that takes a selector or a block which specifies how to retrieve the children of a given item in the tree. Since the children of each package are now selected by our tree presentation, we have to pass only the roots of the package hierarchy to the `display:` argument.

At this point, we can also add pane 3 that shows the method categories as shown in figure 3.2. The listing below introduces no new elements that we have not already discussed:

```
PBE2CodeNavigator>>buildBrowser
browser := GLMTabulator new.
browser
    column: #packages;
    column: #classes;
    column: #categories.

browser transmit to: #packages; andShow: [:a | self packagesIn: a].
browser transmit from: #packages; to: #classes; andShow: [:a | self classesIn: a].
browser transmit from: #classes; to: #categories; andShow: [:a | self categoriesIn: a].
```

```
PBE2CodeNavigator>>categoriesIn: constructor
constructor list
    display: [:class | class organization categories]
```

The browser resulting from the above changes is shown in figure 3.4.

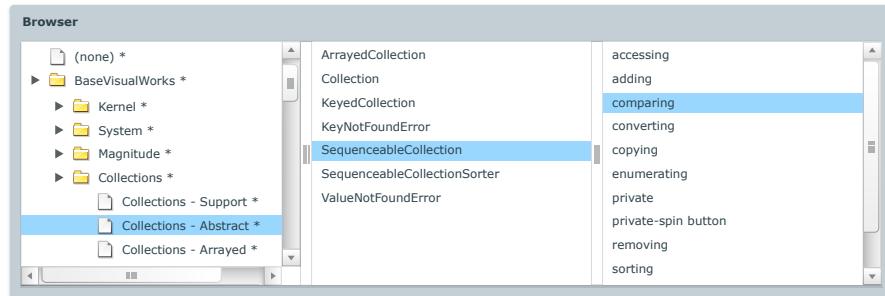


Figure 3.4: Improved class navigator including a tree to display the packages and a list of method categories for the selected class.

Multiple Origins

The mechanism to show the methods is slightly more complicated. When a method category is selected we want to show *only* the methods that belong

to that category. If no category is selected, we want to show *all* methods that belong to the current class.

This leads to our methods pane depending on the selection of two other panes, the class pane and the category pane. Multiple origins can be defined using multiple from: keywords as shown below.

```
PBE2CodeNavigator>>buildBrowser
browser := GLMTabulator new.
browser
    column: #packages;
    column: #classes;
    column: #categories;
    column: #methods.

browser transmit to: #packages; andShow: [:a | self packagesIn: a].
browser transmit from: #packages; to: #classes; andShow: [:a | self classesIn: a].
browser transmit from: #classes; to: #categories; andShow: [:a | self categoriesIn: a].
browser transmit from: #classes; from: #categories; to: #methods; andShow: [:a | self
    methodsIn: a].
```

```
PBE2CodeNavigator>>methodsIn: constructor
constructor list
    display: [:class :category | (class organization listAtCategoryNamed: category)
        sorted].
constructor list
    when: [:class :category | class notNil and: [category isNil]];
    display: [:class | class selectors sorted];
    allowNil
```

The listing shows a couple of properties we have not seen before. First, the multiple origins are reflected in the number of arguments of the blocks that are used in the display: and when: clauses. Secondly, we are using more than one presentation—Glamour shows all presentations whose conditions match in the order that they were defined when the corresponding transmission is fired.

In the first presentation, the condition matches when all arguments are defined (not nil), this is the default for all presentations. The second condition matches only when the category is undefined but the class is not. When a presentation must be displayed even in the presence of undefined origin, it is necessary to use allowNil as shown. We can therefore omit the category from the display block.

The completed class navigator is displayed in figure 3.5.

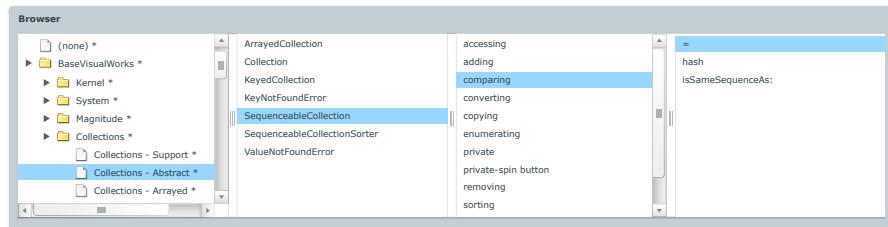


Figure 3.5: Complete code navigator. If no method category is selected, all methods of the class are displayed. Otherwise, only the methods that belong to that category are shown.

Ports

When we stated that transmissions connect panes this was not entirely correct. More precisely, transmissions are connected to properties of panes called *ports*. Such ports consist of a name and a value which accommodates a particular aspect of state of the pane or its contained presentations. If the port is not explicitly specified by the user, Glamour uses the *selection* port by default. As a result, the following two statements are equivalent:

```
browser transmit from: #packages; to: #classes; andShow: [:a | ...].  
browser transmit from: #packages port: #selection; to: #classes; andShow: [:a | ...].
```

Other ports exist and may be used depending on the presentation. For example, the list presentation also populates the *hover* port when the user hovers over an item over a list and a text presentation updates the *text* port to reflect its contents as a user types within it. For a full reference, see the documentation of the presentations being used.

Reusing Browsers

One of the strengths of Glamour lies in the ability to use browsers in place of primitive presentations such as lists and trees. This allows us to reuse browsers and nest them within each other.

In the next example we want to create a class *editor* as shown in figure 3.6. Panes 1 through 4 are equivalent to those described previously. Pane 5 shows the source code of the method that is currently selected in pane 4.

For the sake of the example, we will create this editor in a new class called PBE2CodeEditor. This editor will delegate the presentation of panes 1 through 4 to the previously implemented PBE2CodeNavigator. To achieve this, we first have to make the existing navigator return the constructed browser.

```
PBE2CodeNavigator>>buildBrowser
```

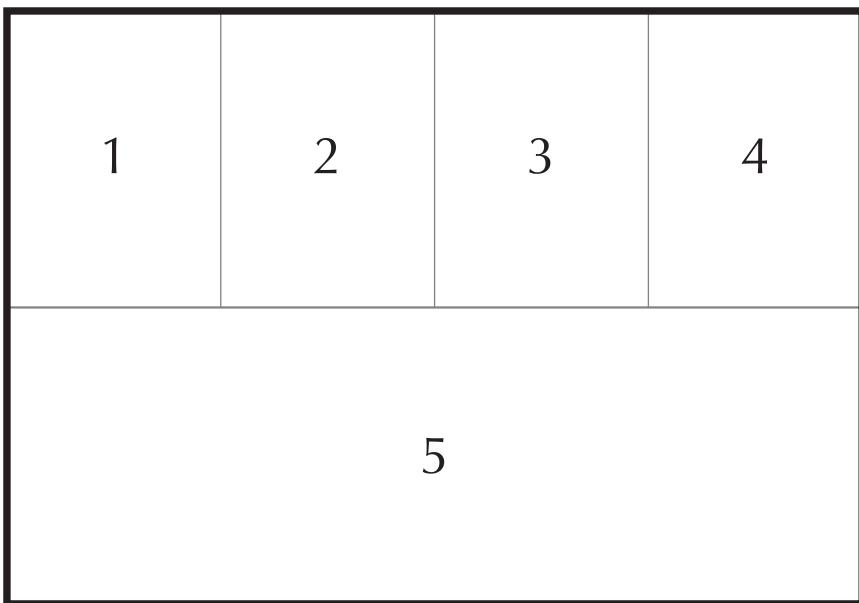


Figure 3.6: Wireframe representation of a Smalltalk class editor.

```
[...]  
↑ browswer
```

We can then reuse the navigator in the new editor browser as shown below.

```
Object subclass: #PBE2CodeEditor  
instanceVariableNames: 'browser'  
classVariableNames: ""  
poolDictionaries: ""  
category: 'PBE2-CodeBrowser'.
```

```
PBE2CodeEditor class>>open  
↑ self new open
```

```
PBE2CodeEditor>>open  
self buildBrowser.  
browser openOn: self organizer
```

```
PBE2CodeEditor>>organizer  
↑ RPackageOrganizer default
```

```
PBE2CodeEditor>>buildBrowser  
browser := GLMTabulator new.
```

```

browser
row: #navigator;
row: #source.

browser transmit to: #navigator; andShow: [:a | self navigatorIn: a].
```

```
PBE2CodeEditor->>navigatorIn: constructor
constructor custom: (PBE2CodeNavigator new buildBrowser)
```

The listing shows how the browser is used exactly like we would use a list or other type of presentation. In fact, browsers are a type of presentation.

When evaluating the code PBE2CodeEditor open, a new browser is opened that shows the navigator embedded in the top pane and an empty pane at the bottom. No source code will be displayed because we have not yet created any connections between the panes. To get to the source, we need both the name of the selected method as well as the class in which it is defined. Since this information is defined only within the navigator browser, we must first export it to the outside world by using the `sendToOutside:from:` message. For this we append the following lines to `codeNavigator`:

```
PBE2CodeNavigator->>buildBrowser
[...]
browser transmit from: #classes; toOutsidePort: #selectedClass.
browser transmit from: #methods; toOutsidePort: #selectedMethod.

↑ browser
```

This will send the selection within classes and methods to the *selectedClass* and *selectedMethod* ports of the containing pane. Alternatively, we could have added these lines to the `navigatorIn:` method in the code editor—it makes no difference to Glamour.

```
PBE2CodeEditor->>navigatorIn: constructor

"Alternative way of adding outside ports. There is no need to use this
code and the previous one simultaneously."

| navigator |
navigator := PBE2CodeNavigator new buildBrowser
    sendToOutside: #selectedClass from: #classes -> #selection;
    sendToOutside: #selectedMethod from: #methods -> #selection;
    yourself.

constructor custom: navigator
```

However, we consider it sensible to clearly define the interface on the side of the code *navigator* rather than within the code editor in order to promote the reuse of this interface as well.

Note that a message for achieving the reverse—importing a port from the outside pane and storing its value on one of the browser’s panes also exists and requires the use of `fromOutsidePort:` and `to::`:

We extend our code editor example as follows:

```
PBE2CodeEditor>>buildBrowser
browser := GLMTabulator new.
browser
    row: #navigator;
    row: #source.

browser transmit to: #navigator; andShow: [:a | self navigatorIn: a].
browser transmit
    from: #navigator port: #selectedClass;
    from: #navigator port: #selectedMethod;
    to: #source;
    andShow: [:a | self sourceIn: a].
```



```
PBE2CodeEditor>>sourceIn: constructor
constructor text
display: [:class :method | class sourceCodeAt: method]
```

We can now view the source code of any selected method and have created a modular browser by reusing the class navigator that we had already written earlier. The composed browser described by the listing is shown in figure 3.7.

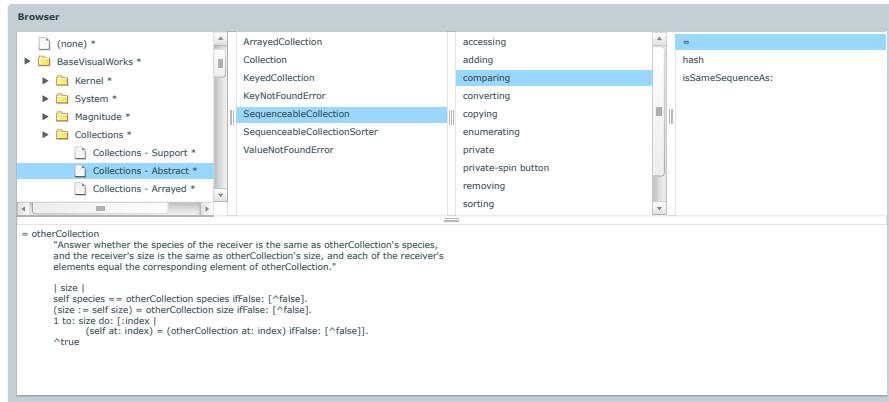


Figure 3.7: Composed browser that reuses the previously described class navigator to show the source of a selected method.

Actions

Browsers generally rely on *actions*—first-class behavioral objects that are executed when a keyboard shortcut is pressed or when an entry in a context menu is clicked. Glamour supports such actions through the `act:on:` message sent to a presentation:

```
PBE2CodeEditor>>sourceln: constructor
constructor text
  display: [:class :method | class sourceCodeAt: method ];
  act: [:presentation :class :method | class compile: presentation text] on: $s.
```

The argument passed to `on:` is a character that specifies the keyboard shortcut that should be used to trigger the action when the corresponding presentation has the focus. Whether the character needs to be combined with a meta-key—such as command, control or alt—is platform specific and need not be specified. The `act:` block provides the corresponding presentation as its first argument which can be used to poll its various properties such as the contained text or the current selection. The other arguments to the block are the incoming origins as defined by `from:` and are equivalent to the arguments of `display:` and `when:`.

Actions can also be displayed as context menus. For this purpose, Glamour provides the messages `act:on:entitled:` and `act:entitled:` where the last argument is a string that should be displayed as the entry in the menu. For example, the following snippet extends the above example to provide a context menu entry to “save” the current method back to the class:

```
...
act: [:presentation :class :method | class compile: presentation text]
on: $s.
entitled: 'Save'
```

Multiple Presentations

Frequently, developers wish to provide more than one presentation of a specific object. In our code browser for example, we may wish to show the classes not only as a list but as a visualization of their *system complexity* as well. Glamour includes support to display and interact with visualizations created using the *Mondrian visualization engine*. To add a second presentation, we simply define it in the `using:` block as well:

```
PBE2CodeNavigator>>classesIn: constructor
constructor list
  when: [:packageName | self organizer includesPackageName: packageName ];
  display: [:packageName | (self organizer packageNamed: packageName)
    definedClasses];
```

```

title: 'Class list'.

constructor mondrian
when: [:packageName | self organizer includesPackageNamed: packageName];
painting: [ :view :packageName |
    view nodes: (self organizer packageNamed: packageName)
        definedClasses.
    view edgesFrom: #superclass.
    view treeLayout];
title: 'Hierarchy'

```

Glamour distinguishes multiple presentations on the same pane with the help of a tab layout. The appearance of the Mondrian presentation as embedded in the code editor is shown in figure 3.8. The clause `title:` sets the name of the tab used to render the presentation.

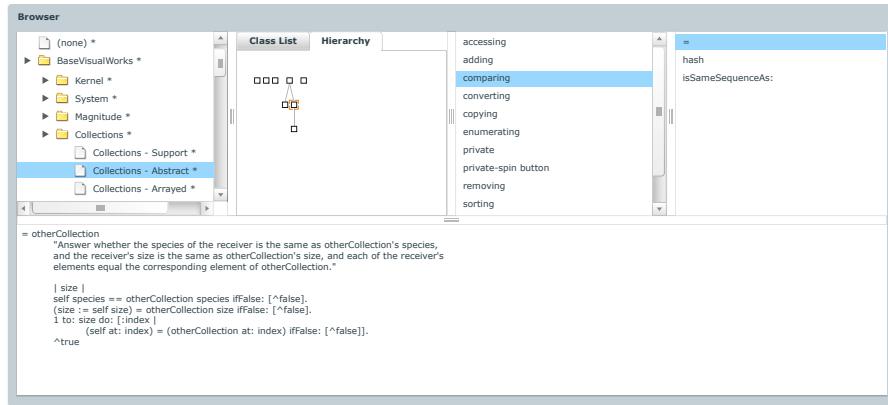


Figure 3.8: Code editor sporting a Mondrian presentation in addition to a simple class list.

Other Browsers

Up to now in the tutorial, we have only used the `GLMTabulator` which is named after its ability to generate custom layouts using the aforementioned `row:` and `column:` keywords. Additional browsers are provided or can be written by the user. Browser implementations can be subdivided into two categories: browsers that have *explicit panes*, i.e., they are declared explicitly by the user—and browsers that have *implicit panes*.

The `GLMTabulator` is an example of a browser that uses explicit panes. With implicit browsers, we do not declare the panes directly but the browser creates them and the connections between them internally. An example of such

a browser is the Finder, which has been discussed in Section 3.1. Since the panes are created for us, we need not use the `from:to:` keywords but can simply specify our presentations:

```
browser := GLMFinder new.

browser list
display: [:class | class subclasses].

browser openOn: Collection
```

The listing above creates a browser (shown in figure 3.9) and opens to show a list of subclasses of `Collection`. Upon selecting an item from the list, the browser expands to the right to show the subclasses of the selected item. This can continue indefinitely as long as something to select remains.

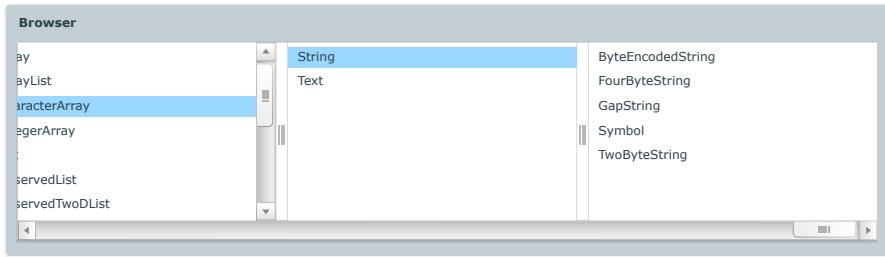


Figure 3.9: Subclass navigator using Miller Columns style browsing.

To discover other kinds of browsers, explore the hierarchy of the `GLMBrowser` class.

Tutorial Conclusion

This concludes our tutorial of Glamour. Please note that this tutorial is not meant to give an exhaustive overview of Glamour, but is merely intended to introduce the reader to the usage and to our intent for our approach. For a more extensive view of Glamour, its concepts and implementation, the Moose book¹ has a dedicated chapter dedicated.

¹<http://www.themoosebook.org/book>

Chapter 4

Using LDAP

with the participation of:

Olivier Auverlot (olivier.auverlot@inria.fr)

4.1 Présentation du protocole LDAP

Le terme LDAP (Lightweight Directory Access Protocol) désigne un protocole permettant l'accès en lecture et en écriture à des annuaires d'entreprises. Ces bases de données, optimisées pour la lecture, sont des référentiels d'informations sur les composants d'une organisation (individus, ressources, organisation fonctionnelle). Le plus souvent, les annuaires LDAP sont exploités pour l'identification des utilisateurs aux services de messagerie ou aux applications intranet. Dans le cadre du développement d'applications e-business, les annuaires LDAP sont des composants techniques essentiels et critiques : il est le plus souvent obligatoire d'interfacer un logiciel avec ces annuaires.

Pharo dispose de différents frameworks pour le développement web (Seaside, AIDAweb, etc.) ainsi que pour l'accès aux bases de données (DBXTalk). Quelle est la situation avec LDAP ? Peut-on créer des applications capables d'identifier un utilisateur ou de récupérer des informations dans un annuaire LDAP ?

Installation de LDAPPlayer

Pour communiquer avec un annuaire LDAP, Pharo dispose de la bibliothèque de classes LDAPPlayer créée par Ragnar Hojland Espinosa. Elle disponible via Squeaksource à l'adresse suivante: <http://www.squeaksource.org/>

LDAPPlayer/.

Pour l'installer, lancez Pharo et définissez un nouveau dépôt HTTP à l'aide de Monticello.

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

4.2 Les principales classes

La plupart des classes que nous allons découvrir sont stockées dans le paquet LDAP-Core. Une connexion à un annuaire LDAP instancie un objet LDAPConnection. Des requêtes vers l'annuaire retournent des objets LDAPResult. La classe LDAPAttrModifier fournit plusieurs méthodes de classe permettant de modifier les attributs d'une entrée de l'annuaire. La classe LDAPFilter a pour finalité de définir des filtres de recherche et de limiter ainsi le volume d'informations retourné au client.

Définir une connexion. Créons une nouvelle catégorie nommée 'AppLDAP' pour découvrir les fonctionnalités de LDAPPlayer. Notre objectif est de mettre en situation les principales fonctionnalités de la bibliothèque LDAPPlayer et de vous permettre de les réutiliser facilement dans une véritable application. Cette catégorie contient une classe unique nommée 'Ldap' qui renfermera l'ensemble de nos méthodes d'instances.

Pour définir la connexion, nous allons utiliser cinq variables d'instance qui représentent le nom du serveur LDAP (hostname), le port TCP de connexion (port), le compte utilisateur (bindDN), le mot de passe de l'utilisateur (password) et la base de la recherche (baseDN) qui sera utilisée pour construire un DN (Distinguished Name).

```
Object subclass: #Ldap
  instanceVariableNames: 'baseDN bindDN hostname password port'
  classVariableNames: ''
  poolDictionaries: ''
  category: 'Ldap'
```

```
Ldap>>baseDN
↑ baseDN
Ldap>>baseDN: anObject
  baseDN := anObject
Ldap>>bindDN
↑bindDN
Ldap>>bindDN: anObject
```

```

bindDN := anObject
Ldap>>hostname
  ↑hostname
Ldap>>hostname: anObject
  hostname := anObject
Ldap>>password
  ↑password
Ldap>>password: anObject
  password := anObject
Ldap>>port
  ↑ port
Ldap>>port: anObject
  port := anObject

```

Pour initialiser ces variables d'instances, créons une méthode initialize dans le protocole initialize-release.

```

Ldap>>initialize
super initialize.
self hostname: 'ldap.mondomaine.org'.
self port: 389.
self bindDN: 'cn=admin,dc=domaine,dc=org'.
self password: 'mysecretpassword'.
self baseDN: 'ou=people,dc=domaine,dc=org'.

```

Nous avons également besoin d'une variable d'instance permettant de sauvegarder un objet décrivant la connexion vers l'annuaire LDAP.

```

Object subclass: #Ldap
instanceVariableNames: 'connection baseDN bindDN hostname password port'
classVariableNames: ""
poolDictionaries: ""
category: 'Ldap'

```

Bien évidemment, nous avons besoin de deux accesseurs pour écrire et lire dans cette variable d'instance. Créons un protocole 'accessing' et définissons les deux méthodes suivantes :

```

Ldap>>connection
  ↑ connection
Ldap>>connection: anObject
  connection := anObject

```

Nous pouvons maintenant définir un protocole 'action' qui contiendra l'ensemble des méthodes d'instances ayant une interaction avec l'annuaire LDAP. Commençons par définir une méthode connect qui assure la connexion et initialise la variable d'instance connection.

| req |

```
self connection: (LDAPConnection to: self hostname port: self port).
req := connection bindAs: self bindDN credentials: self password.
req wait.
↑ self.
```

Pour travailler proprement, il nous faut également une méthode permettant de déconnecter notre application une fois que la connexion à l'annuaire LDAP n'est plus utile. Pour cela, définissez simplement une méthode 'disconnect' qui sera chargée de cette tâche.

```
Ldap>>disconnect
self connection disconnect.
```

Est-ce que cela fonctionne ? Un petit test rapide pour vérifier. Ouvrons le Workspace et exécutons le code suivant :

```
ann := Ldap new connect.
ann disconnect.
```

Si tout c'est bien passé, félicitations ! Notre application s'est connecté à votre annuaire LDAP. Passons à la suite et tentons de récupérer des informations.

Chercher des informations

Nous pouvons maintenant essayer de lire des données dans notre annuaire LDAP. Pour cela, nous allons définir une première méthode 'searchAll', très simple, qui récupère l'intégralité des entrées à partir du DN de base. Facile n'est-ce pas ? Par contre, ce n'est pas une méthode très recommandable si votre annuaire contient plusieurs milliers d'entrées. Lorsque l'on fait une recherche dans un LDAP, on travaille plutôt avec des filtres afin de limiter le volume d'information récupéré par l'application cliente. Définissons une méthode 'search' recherchant les entrées pour lesquelles un attribut à une certaine valeur.

```
Ldap>>searchAll
| req result |
req := self connection
newSearch: self baseDN
scope: (LDAPConnection wholeSubtree)
deref: (LDAPConnection derefNever)
filter: nil
attrs: nil
wantAttrsOnly: false.
↑req result
```

Dans le workspace, nous pouvons maintenant lire et afficher ces informations. A titre d'exemple, nous n'affichons que l'attribut cn :

```
| ann result |
ann := Ldap new connect.
result := ann searchAll.
result do: [ :each |
    Transcript show: (each attrAt: #cn); cr].
ann disconnect.
```

```
Ldap>>search: aValue attribute: aAttribute
| req result |
req := self connection
newSearch: self baseDN
scope: (LDAPConnection wholeSubtree)
deref: (LDAPConnection derefNever)
filter: (LDAPFilter with: aAttribute equalTo: aValue)
attrs: nil
wantAttrsOnly: false.
↑req result
```

Dans le Workspace, nous pouvons maintenant rechercher les entrées pour lesquelles le champ 'équipe' est égal à 'INFORMATIQUE'.

```
| ann result |
ann := Ldap new connect.
result := ann search: 'INFORMATIQUE' attribute: 'équipe'.
result do: [ :each |
    Transcript show: (each attrAt: #cn); cr].
ann disconnect.
```

Modifier des attributs

La modification des attributs consiste à ajouter une valeur à un attribut ou à supprimer un attribut. Tout d'abord, définissons une méthode pour ajouter un attribut à une entrée existante. Cette méthode reçoit trois paramètres qui sont l'identifiant unique d'une entrée dans l'annuaire (DN) construit à partir de l'UID (User ID) et du DN de base, le nom du paramètre modifié et la valeur affecté à ce paramètre.

```
Ldap>>addValueTo: aDN attribute: aAttribute with: aValue
| req ops |
ops := { LDAPAttrModifier addTo: aAttribute values: { aValue } }.
req := connection modify: (aDN, ',', self baseDN) with: ops. req wait.
```

Essayons maintenant cette méthode dans le Workspace en ajoutant une valeur à l'attribut 'givenName' de l'utilisateur 'dupont' :

```
| ann |
ann := Ldap new connect.
ann addValueTo: 'UID=dupont' attribute: 'givenName' with: 'Alfred'.
ann disconnect.
```

Nous allons ensuite créer une méthode permettant de changer la valeur d'un attribut. Cette méthode reçoit trois paramètres qui sont le DN de l'entrée devant être modifiée, le nom de l'attribut et la valeur affectée.

```
Ldap>>changeValueOf: aDN attribute: aAttribute with: aValue
| req ops |
ops := { LDAPAttrModifier set: aAttribute to: { aValue }}.
req := connection modify: (aDN, ',', self baseDN) with: ops.
req wait.
```

L'exemple suivant modifie l'attribut 'Statut' de l'utilisateur 'dupont'.

```
| ann |
ann := Ldap new connect.
ann changeValueOf: 'UID=dupont' attribute: 'Statut' with: 'programmer'.
ann disconnect.
```

Créons maintenant une méthode qui supprime un attribut dans le LDAP. Cette méthode reçoit un DN en paramètre afin de spécifier l'entrée concernée par la suppression de l'attribut.

```
Ldap>>deleteAttribute: aAttribute from: aDN
| req ops |
ops := { LDAPAttrModifier del: aAttribute}.
req := connection modify: (aDN, ',', self baseDN) with: ops.
req wait.
```

Nous pouvons maintenant supprimer l'entrée 'stage' pour l'UID 'dupont' :

```
| ann |
ann := Ldap new connect.
ann deleteAttribute: 'stage' from: 'UID=dupont'.
ann disconnect.
```

Il est également possible d'effacer un attribut en fonction de sa valeur. Ecrivons une méthode `deleteAttribute:value:from:` qui réalise cela :

```
Ldap>>deleteAttribute: aAttribute value: aValue from: aDN | req ops |
ops := { LDAPAttrModifier delFrom: aAttribute values: { aValue }}.
req := connection modify: (aDN, ',', self baseDN) with: ops.
req wait.
```

L'exemple suivant supprime l'attribut 'givenName' si sa valeur est égale à 'Alfred'. La recherche est basée ici sur un DN constitué d'un UID utilisateur et du DN de base.

```
| ann |
ann := Ldap new connect.
ann deleteAttribute: 'givenName' value: 'Alfred' from: 'UID=dupont'.
ann disconnect.
```

Créer une entrée dans l'annuaire

Si nous voulons ajouter un nouvel élément à notre annuaire, il nous faut créer une nouvelle entrée. Nous allons créer une méthode `createEntry:with:` qui reçoit en paramètre un UID ainsi qu'une collection contenant les informations devant être insérées dans la nouvelle entrée.

```
Ldap>>createEntry: aUID with: aCollection
| req |
req := self connection addEntry: aUID attrs: aCollection.
req wait.
```

Nous pouvons alors ajouter une nouvelle entrée de type 'inetOrgPerson' pour l'utilisateur 'dupont'. Les attributs 'cn' et 'sn' sont fixés à 'dupont' également.

```
| ann attrs |
attrs := Dictionary new
at: 'objectClass'
put: (OrderedCollection new add: 'inetOrgPerson'; yourself);
at: 'cn' put: 'dupont';
at: 'sn' put: 'dupont'; yourself.
ann := Ldap new connect.
ann createEntry: ('uid=dupont,' , ann baseDN) with: attrs.
ann disconnect.
```

Supprimer une entrée dans l'annuaire

Effacer une entrée dans l'annuaire est simple et rapide. Attention aux mauvaises manoeuvres ! Pour cela, nous pouvons définir une méthode `deleteEntry:` recevant en paramètre l'UID de l'entrée qui sera effacée.

```
Ldap>>deleteEntry: aUID
| req |
req := self connection delEntry: (aUID , ',' , self baseDN).
req wait.
```

Essayons notre méthode en supprimant dans l'annuaire, l'entrée précédemment créée pour l'utilisateur 'dupont' :

```
| ann |
ann := Ldap new connect.
ann deleteEntry: 'uid=dupont'.
ann disconnect.
```

4.3 Présentation des classes principales

4.4 Conclusion

Nous venons de découvrir les principales fonctionnalités de LDAPlayer et vous disposez maintenant des outils essentiels pour exploiter un annuaire LDAP avec Pharo. Vous pouvez interroger un annuaire, modifier ou effacer des entrées selon vos besoins.

Liens utiles <http://www.openldap.org/>, <http://en.wikipedia.org/wiki/LDAP> http://fr.wikipedia.org/wiki/Lightweight_Directory_Access_Protocol

Chapter 5

Mondrian

with the participation of:
Alexandre Bergel (alexandre@bergel.eu)

@@What the reader will learn why this is worth reading this chapter@@

5.1 Introduction

Mondrian is made to visualize data, and is commonly employed to operate on software source code. This chapter will keep this original intent by using small and concrete examples of software and numerical.

There are essentially two ways to work with Mondrian. By either using the easel which provides an interactive scripting engine or by directly interacting with a view renderer.

We will first use Mondrian in its easiest way, by using the easel. Note that this chapter assume you have basic Smalltalk knowledge.

To open an easel, you can either use the World menu or execute the expression

```
MOEasel open.
```

5.2 First visualization

In the easel you have just opened, enter the following and press the *generate* button

```
view nodes: (1 to: 20).
```

You should see a new window with about 20 small boxes lined up in the top left corner. You have just rendered the numerical set between 1 and 20.

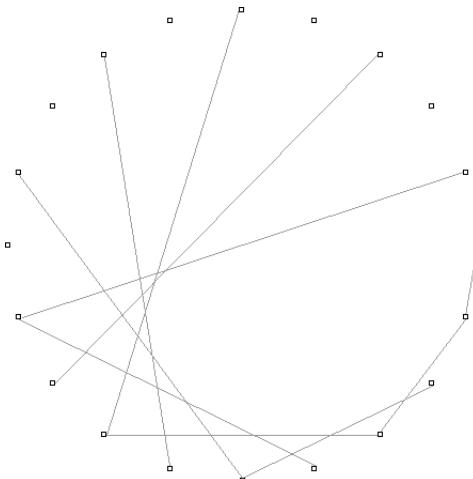
Edges may be easily added. Add a second line:

```
view nodes: (1 to: 20).
view edgesFrom: [ :v | v * 2 ].
```

Each value is connected to its next value. Nodes may be dragged by double clicking on it.

Nodes may be ordered in circule using the circle layout.

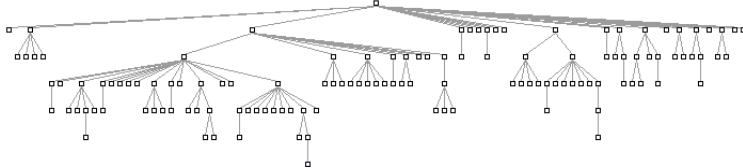
```
view nodes: (1 to: 20).
view edgesFrom: [ :v | v * 2 ].
view circleLayout.
```



5.3 Visualizing the collection framework

We will now visualize Smalltalk classes. In the remaining of this section, we will intensively use Smalltalk reflection to make compelling examples. Let's visualize the hierarchy of classes contained in the Collection framework:

```
view nodes: Collection withAllSubclasses.
view edgesFrom: #superclass.
view treeLayout.
```



We have used a tree layout to visualize Smalltalk class hierarchies. This fits well since Smalltalk is single-inheritance oriented.

`Collection` is the root class of the Smalltalk collection framework library. The message `withAllSubclasses` returns the list of its subclasses.

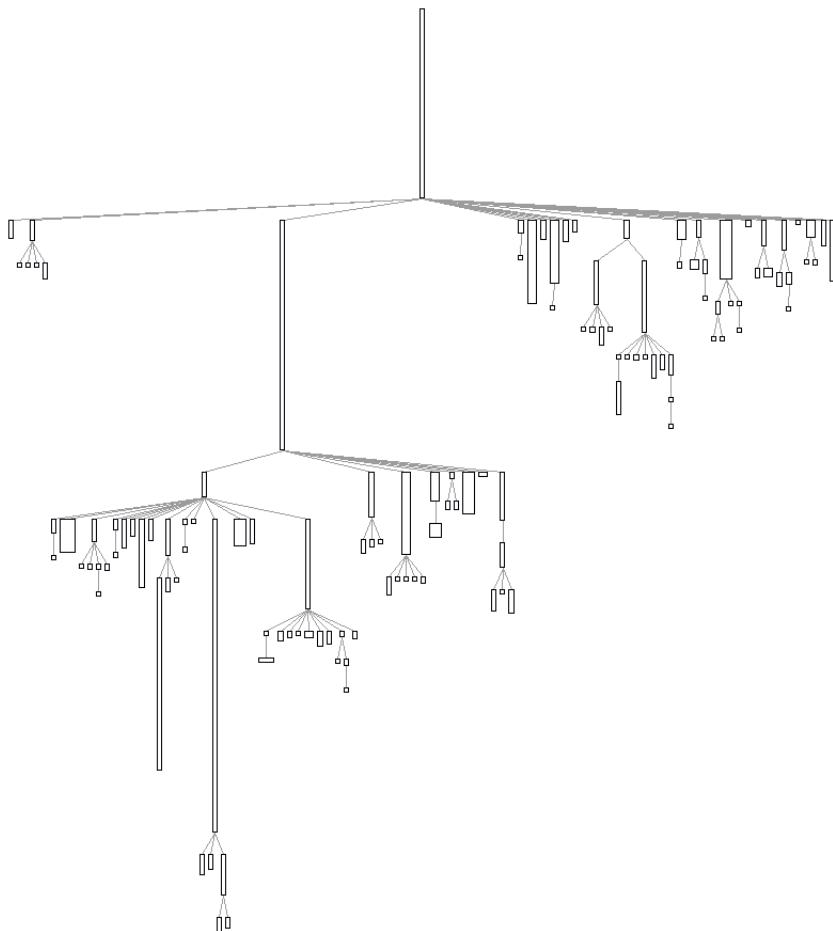
Classes are ordered vertically along their inheritance link. A superclass is above its subclasses.

5.4 Reshaping nodes

Mondrian visualizes graph of objects. Each graph element (i.e., node and edge) has a shape that represents the visual aspect of the element. As you have seen the default shape of a node is a five pixels wide square and the default shape of an edge is a thin straight gray line.

A number of dimensions defines the appearance of a shape: the width and the height of a rectangle; the size of a line dash, for example. We will reshape the node of our visualization to convey more information about the classes we are visualizing.

```
view shape rectangle
width: [ :each | each instVarNames size * 3];
height: [ :each | each methods size ].
view nodes: Collection withAllSubclasses.
view edgesFrom: #superclass.
view treeLayout.
```



Note that the expression

```
view edgesFrom: #superclass
```

is equivalent to

```
view edges: Collection withAllSubclasses from: #yourself to: #superclass.
```

itself equivalent to

```
view
edges: Collection withAllSubclasses
from: [ :each | each superclass ]
to: [ :each | each ].
```

Edges are created by computing the source and target node for each node provided as first argument. In Mondrian, the expression

```
[ :aClass | aClass superclass ].
```

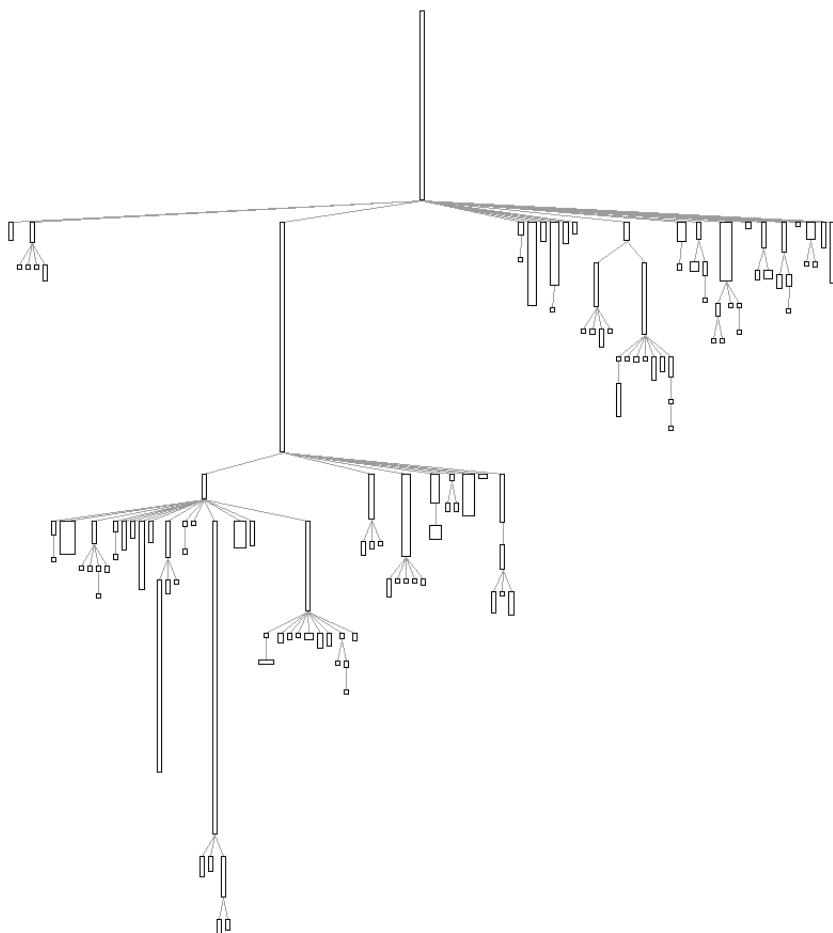
is equivalent to

```
#superclass
```

5.5 Multiple edges

Multiple edges may be defined. In the previous section, edges where departed from the superclass to the running node. Alternatively, edges may depart from the running nodes to all subclasses. For that very purpose, we will use `edges:from:toAll:`, a variant of `edges:from:to:`.

```
view shape rectangle
width: [ :each | each instVarNames size * 3];
height: [ :each | each methods size ].
view nodes: Collection withAllSubclasses.
view edges: Collection withAllSubclasses from: #yourself toAll: #subclasses.
view treeLayout.
```



With `edges:from:toAll:` and `edgesFrom:`, two edges cannot start from a unique node.

Time to time, you may need a finer grain for edge declaration. The following is commonly employed as in:

```

view nodes: (1 to: 5).
view shape arrowedLine.
view edges: { 1 -> 2 . 1 -> 5 . 4 -> 3 } from: #key to: #value.
view circleLayout

```

5.6 Adding colors

Node color is an important information support, as the width and the height of a node. Color should be easy to pick to represent particular condition

The keyword `if:fillColor:` enables one to assign a color for a particular condition. Consider we want to extend the previous example by coloring abstract classes in red.

```
view shape rectangle
width: [ :each | each instVarNames size * 3];
height: [ :each | each methods size ];
if: #isAbstractClass fillColor: Color red.
view nodes: Collection withAllSubclasses.

view edges: Collection withAllSubclasses from: #yourself toAll: #subclasses.

view treeLayout.
```

The method `isAbstractClass` is defined on Behavior and Metaclass. By sending the `isAbstractClass` to a class return a boolean value telling us whether the class is abstract or not. We recall that an abstract class in Smalltalk is a class that defines or inherits at least one abstract method (i.e., which contain 'self subclassResponsibility').

The message `if:fillColor:` may be sent to a shape to conditionally set a color

```
view shape rectangle
width: [ :each | each instVarNames size * 3];
height: [ :each | each methods size ];
if: #isAbstractClass fillColor: Color red.
view nodes: Collection withAllSubclasses.

view edgesFrom: #superclass.

view treeLayout.
```

All red nodes represent abstract class. Waving the mouse above a node make a text tooltip appear revealing its name.

Extended possibilities exist to define interaction. We will review them in a future section. For now, if you are interested in opening a system browser directly from a node, you define this interaction:

```
view interaction action: #browse.
view shape rectangle
width: [ :each | each instVarNames size * 3];
height: [ :each | each methods size ];
if: #isAbstractClass fillColor: Color red.
view nodes: Collection withAllSubclasses.
```

```
view edgesFrom: #superclass.
```

```
view treeLayout.
```

You can easily spot some red node that do not have subclasses. This indicates a design flow since an abstract must to have subclasses. It makes no sense for a class that is not supposed to be instantiated (since it is abstract) to not have a subclass.

The very same analyzes can be realized on your own classes.

```
view interaction action: #browse.
```

```
view shape rectangle
```

```
width: [ :each | each instVarNames size * 3];
```

```
height: [ :each | each methods size ];
```

```
if: #isAbstractClass fillColor: Color red.
```

```
view nodes: (PackageInfo named: 'Mondrian') classes.
```

```
view edgesFrom: #superclass.
```

```
view treeLayout.
```

A shape may contains more than one condition. Let's distinguish abstract classes from classes that define abstract methods.

```
view shape rectangle
```

```
width: [ :each | each instVarNames size * 3];
```

```
height: [ :each | each methods size ];
```

```
if: #isAbstractClass fillColor: Color lightRed;
```

```
if: [:cls | cls methods anySatisfy: #isAbstract ] fillColor: Color red.
```

```
view nodes: Collection withAllSubclasses.
```

```
view edgesFrom: #superclass.
```

```
view treeLayout.
```

All red nodes are still abstract classes. Light red indicates classes that do not define abstract methods; strong red indicates classes that define at least one abstract method.

5.7 More on colors

Color may be defined in many different ways. For example, you can directly send `fillColor:` to a shape.

```
view shape rectangle
  fillColor: Color lightBlue.
```

The list of available colors may be found in the class side of the class `Color`. The intensity of a node color may also indicate a quantitative value. For example, the intensity of node color is linear to the number of lines of code defined in the class.

```
view interaction action: #browse.
view shape rectangle
  width: [ :each | each instVarNames size * 3];
  height: [ :each | each methods size ];
  linearFillColor: #numberOfLinesOfCode within: Collection withAllSubclasses.
view nodes: Collection withAllSubclasses.

view edgesFrom: #superclass.

view treeLayout.
```

The visualization you now obtain put in relation for each class the number of methods, the number of instance variable and the number of lines of code. You can easily spot differences between classes, which might suggest you some a refactoring activity. This visualization is named 'System complexity', if you wish to know more about this visualization, you can refer to 'Polymetric Views—A Lightweight Visual Approach to Reverse Engineering' (Transactions on Software Engineering, 2003).

The message `linearFillColor:within:` takes as first argument a block function that return a numerical value. The second argument is a group of elements that is used to scale the intensity of each node. The block function is applied to each element of the group. The greatest value is set to black. The fading scales from 0 to this maximum.

5.8 Popup view

Consider the example that indicate abstract classes. Assume now that we would like to know what are the abstract methods defined. An easy way to do, is to extend the popup text with the list of abstract methods. Putting the mouse above a class does not only give its name, but also the list of abstract methods defined in the class.

```
view shape rectangle
  width: [ :each | each instVarNames size * 3];
  height: [ :each | each methods size ];
  if: #isAbstractClass fillColor: Color lightRed;
  if: [:cls | cls methods anySatisfy: #isAbstract ] fillColor: Color red.
```

```

view interaction popupText: [ :aClass |
| stream |
stream := WriteStream on: String new.
(aClass methods select: #isAbstract thenCollect: #selector)
do: [:sel | stream nextPutAll: sel; nextPut: $; cr].
aClass name printString, ' => ', stream contents ].
view nodes: Collection withAllSubclasses.
view edgesFrom: #superclass.
view treeLayout.

```

Another view may be opened instead of a text. Consider the following view definition:

```

view shape rectangle
width: [ :each | each instVarNames size * 3];
height: [ :each | each methods size ];
if: #isAbstractClass fillColor: Color lightRed;
if: [:cls | cls methods anySatisfy: #isAbstract ] fillColor: Color red.

view interaction popupView: [ :element :secondView |
secondView shape rectangle
if: #isAbstract fillColor: Color red;
size: 10.
secondView nodes: (element methods sortedAs: #isAbstract).
secondView gridLayout gapSize: 2
].
view nodes: Collection withAllSubclasses.
view edgesFrom: #superclass.
view treeLayout.

```

When the mouse enters a node, a new view is defined and displayed next to the node. The method `popupView:` receives a two-arguments block as parameters. The first parameter of the block is the element represented by the node located below the mouse. The second argument is a new view that will be opened.

In the example, we used `sortedAs:` to order the nodes representing methods. This method is defined on `Collection` and belongs to `Mondrian`. To see example usages of `sortedAs:`, browse its corresponding unit test:

```
ToolSet browse: MOViewRendererTest selector: #testSortedAs
```

5.9 Subviews

Each node may embed a view. This is achieved via the keywords `nodes:forEach:` and `node:forIt:`. Consider an alternative view for visualizing ab-

stract methods:

```

view shape rectangle
if: #isAbstractClass fillColor: Color lightRed.
view nodes: Collection withAllSubclasses forEach: [ :aClass |
view shape rectangle
if: #isAbstract fillColor: Color red.
view nodes: (aClass methods sortedAs: #isAbstract).
view gridLayout gapSize: 2.
].
view edgesFrom: #superclass.
view treeLayout.
```

Abstract classes are painted in light red and abstract method in intense red. Each node may embed any arbitrary view. Consider this script to visualize class hierarchies for each package of the Collection framework.

```

packages := PackageOrganizer default packageNames
select: [ :name | name beginsWith: 'Collections-' ]
thenCollect: [ :name | PackageInfo named: name ].

view nodes: packages forEach: [ :aPackage |
view nodes: (aPackage classes reject: #isTrait).
view edgesFrom: #superclass.
view treeLayout
].
```

We see an interesting issue here. A little more information is here shown. Not only we see the hierarchies defined in each packages, but we also see inter-package links. We can suppress relations between packages and indicate in blue classes for which their superclasses lives in a different package.

```

packages := PackageOrganizer default packageNames
select: [ :name | name beginsWith: 'Collections-' ]
thenCollect: [ :name | PackageInfo named: name ].

view nodes: packages forEach: [ :aPackage |
| classes |
classes := aPackage classes reject: #isTrait.
view shape rectangle
if: [ :cls | (classes includes: cls superclass) not ] fillColor: Color blue.
view nodes: classes.
classes := classes copy select: [ :cls | classes includes: cls superclass ].
view edges: classes from: #superclass to: #yourself.
view treeLayout
].
```

Package names can be included above each box.

```

packages := PackageOrganizer default packageNames
    select: [ :name | name beginsWith: 'Collections-' ]
    thenCollect: [ :name | PackageInfo named: name ].

view shape rectangle withoutBorder.
view nodes: packages forEach: [ :aPackage |
    view shape label text: #packageName.
    view node: aPackage.
    view node: aPackage forIt: [
        | classes |
        classes := aPackage classes reject: #isTrait.
        view shape rectangle
            if: [ :cls | (classes includes: cls superclass) not ] fillColor: Color blue.
        view nodes: classes.
        classes := classes copy select: [ :cls | classes includes: cls superclass ]..
        view edges: classes from: #superclass to: #yourself.
        view treeLayout
    ].
    view verticalLineLayout center
].

```

5.10 Forwarding events

Package of the visualization given in the previous section may be moved by a simple drag and drop. However, the invisible outer node must be grabbed. The name and the hierarchy box may be dragged on their own still. A better adjustment is to let the package name and the hierarchy box to forward the event to the outer invisible node (when executing the script below, declare packages as a temporary variable).

```

packages := PackageOrganizer default packageNames
    select: [ :name | name beginsWith: 'Collections-' ]
    thenCollect: [ :name | PackageInfo named: name ].

view shape rectangle withoutBorder.
view nodes: packages forEach: [ :aPackage |
    view shape label text: #packageName.
    view interaction forward.
    view node: aPackage.

    view interaction forward.
    view node: aPackage forIt: [
        | classes |
        classes := aPackage classes reject: #isTrait.
        view shape rectangle
            if: [ :cls | (classes includes: cls superclass) not ] fillColor: Color blue.
    ].

```

```

view nodes: classes.
classes := classes copy select: [ :cls | classes includes: cls superclass ].
view edges: classes from: #superclass to: #yourself.
view treeLayout
].
view verticalLineLayout center
].

```

A node may now be dragged from its name or the hierarchy box. It may be useful to selectively forward events: `forward:` is a variant of `forward`.

```

packages := PackageOrganizer default packageNames
    select: [ :name | name beginsWith: 'Collections-' ]
    thenCollect: [ :name | PackageInfo named: name ].

view shape rectangle withoutBorder.
view nodes: packages forEach: [ :aPackage |
    view shape label text: #packageName.
    view interaction forward: MOMouseDrag.
    view node: aPackage.

    view interaction forward: MOMouseDrag.
    view node: aPackage forIt: [
        | classes |
        classes := aPackage classes reject: #isTrait.
        view shape rectangle
            if: [ :cls | (classes includes: cls superclass) not ] fillColor: Color blue.
        view nodes: classes.
        classes := classes copy select: [ :cls | classes includes: cls superclass ].
        view edges: classes from: #superclass to: #yourself.
        view treeLayout
    ].
    view verticalLineLayout center
].

```

Note that several nodes may be selected by pressing the Ctrl or Cmd key.

5.11 Events

Each mouse movement, click and keyboard click correspond to a particular events in Mondrian. The root of the events is `MOEvent`. Event handler are defined directly on the interaction object.

```

view shape rectangle
width: [ :each | each instVarNames size * 3];
height: [ :each | each methods size ];
if: #isAbstractClass fillColor: Color lightRed;

```

```

if: [:cls | cls methods anySatisfy: #isAbstract ] fillColor: Color red.

view interaction on: MOMouseDouble do: [ :ann |
  (OBSSystemBrowser onClass: ann modelElement) open
].
view nodes: Collection withAllSubclasses.
view edgesFrom: #superclass.
view treeLayout.

```

The block handler accepts one argument, the event generated. The object that triggered the event is obtained by sending `modelElement` to the event object.

5.12 Interaction

Mondrian offers a number of contextual interaction mechanisms. The interaction object contains a number of keywords for that purpose. Consider the example:

```

view interaction
  highlightWhenOver: [:v | {v - 1 . v + 1. v + 4 . v - 4}].
view shape rectangle
  width: 40;
  height: 30;
  withText.
view nodes: (1 to: 16).
view gridLayout gapSize: 2.

```

We will extend our running example with an indication about the test coverage. Consider the following script:

```

view shape rectangle
  width: [ :each | each instVarNames size * 3];
  height: [ :each | each methods size ];
  linearFillColor: #numberOfLinesOfCode within: Collection withAllSubclasses.
view nodes: Collection withAllSubclasses.
view edgesFrom: #superclass.
view treeLayout.

view shape rectangle withoutBorder.
view node: 'compound' forIt: [
  view shape label.
  view node: 'Collection tests'.

view node: 'Collection tests' forIt: [
  | testClasses |
  testClasses := (PackageInfo named: 'CollectionsTests') classes reject: #isTrait.

```

```
view shape rectangle
width: [ :cls | (cls methods inject: 0 into: [ :sumLiterals :mtd | sumLiterals + mtd
    allLiterals size]) / 100 ];
height: [ :cls | cls numberOfLinesOfCode / 50 ].
view interaction
highlightWhenOver: [ :cls | ((cls methods inject: #()
    into: [:sum :el | sum , el allLiterals ]) select: [:v | v isKindOf: Association ]
    thenCollect: #value) asSet ].
view nodes: testClasses.
view edgesFrom: #superclass.
view treeLayout ].

view verticalLineLayout alignLeft
].
```

The script contains two parts. The first part is the ubiquitous system complexity of the collection framework. We have seen it a number of times already. The second part renders the tests contained in the CollectionsTests. The width of a class is the number of literals contained in it. The height is the number of lines of code. Since the collection tests makes a great use of traits to reuse code, these metrics have to be scaled down. When the mouse is put over a test unit, then all the classes of the collection framework referenced in this class are highlighted.

5.13 Conclusion

Mondrian is a perpetual evolving application. This chapter intents to cover the features frequently used. If there is a topic you wish to see discussed here, send an email to alexandre@bergel.eu.

We hope you haved enjoyed it!

Chapter 6

DBXTalk

DBXTalk is a database driver that allows interaction with major relational database engines such as Oracle and MSSQL, apart from those which are open source, like PostgreSQL and MySQL.

Moreover, this driver is integrated with GLORP, enabling a complete and open-source solution to relational database access. To do this, DBXTalk uses a library called OpenDBX.

6.1 DBXTalk Driver Architecture

The DBXTalk Driver relies on several components in order to connect to different relational databases:

- The OpenDBXDriver package talks to the OpenDBX library and handles the engines differences.
- OpenDBX is a C library which stands as an adapter between the different database engines and our Pharo image, and provides a common interface to interact with through FFI.
- We will need the corresponding client database library that OpenDBX will talk to.



6.2 Installing DBXTalk

This section will introduce the installation of the components needed to run DBXTalk in MS Windows, Unix and MacOS.

Part II

Language

Chapter 7

Handling exceptions

with the participation of:

Stéphane Ducasse (stephane.ducasse@inria.fr)

All applications have to deal with exceptional situations. Arithmetic errors may occur (such as division by zero), unexpected situations may arise (file not found), or resources may be exhausted (network down, disk full, etc.). The old-fashioned solution is to have operations that fail return a special *error code*; this means that client code must check the return value of each operation, and take special action to handle errors.

Modern programming languages, including Smalltalk, instead offer a dedicated exception-handling mechanism that greatly simplifies the way in which exceptional situations are signaled and handled. Before the development of the ANSI Smalltalk standard in 1996, several exception handling mechanisms existed, largely incompatible with each other. Pharo’s exception handling follows the ANSI standard, with some embellishments; we present it in this chapter from a user perspective.

The basic idea behind exception handling is that client code does not clutter the main logic flow with checks for error codes, but specifies instead an *exception handler* to “catch” exceptions. When something goes wrong, instead of returning an error code, the method that detects the exceptional situation interrupts the main flow of execution by *signaling* an exception. This does two things: it captures essential information about the context in which the exception occurred, and transfers control to the exception handler, written by the client, which decides what to do about it. The “essential information about the context” is saved in an *Exception* object; various classes of *Exception* are specified to cover the varied exceptional situations that may arise.

Pharo’s exception-handling mechanism is particularly expressive and flexible, covering a wide range of possibilities. Exception handlers can be

used to *ensure* that certain actions take place even if something goes wrong, or to take action only if something goes wrong. Like everything in Smalltalk, exceptions are objects, and respond to a variety of messages. When an exception is caught by a handler, there are many possible responses: the handler can specify an alternative action to perform; it can ask the exception object to *resume* the interrupted operation; it can *retry* the operation; it can *pass* the exception to another handler; or it can *reraise* a completely different exception.

With the help of a series of examples, we shall explore all of these possibilities, and we shall also take a brief look into the internal mechanics of exceptions and exception handlers. However, before we do that, we need to stop and think for a moment about the consequence of adding exceptions into a language: *we can no longer be sure that a message send will give us an answer*. In other words, once we have exceptions, any message send has the potential not to return to the sender: it may fail.

7.1 Ensuring execution

The `ensure:` message can be sent to a block to make sure that, even if the block fails (*e.g.*, raises an exception) the argument block will still be executed:

```
anyBlock ensure: ensuredBlock  "ensuredBlock will run even if anyBlock fails"
```

Consider the following example, which creates an image file from a screenshot taken by the user:

```
| writer |
writer := GIFReaderWriter on: (FileStream newFileNamed: 'Pharo.gif').
[ writer nextPutImage: (Form fromUser) ]
    ensure: [ writer close ]
```

This code ensures that the writer file handle will be closed, even if an error occurs in `Form fromUser` or while writing to the file.

Here is how it works in more detail. The `nextPutImage:` method of the class `GIFReaderWriter` converts a form (*i.e.*, an instance of the class `Form`, representing a bitmap image) into a GIF image. This method writes into a stream which has been opened on a file. The `nextPutImage:` method does not close the stream it is writing to, therefore we should be sure to close the stream even if a problem arises while writing. This is achieved by sending the message `ensure:` to the block that does the writing. In case `nextPutImage:` fails, control will flow into the block passed to `ensure:`. If it does *not* fail, the ensured block will still be executed. So, in either case, we can be sure that `writer` is closed.

Here is another use of `ensure:`, in class `Cursor`:

```

Cursor»showWhile: aBlock
  "While evaluating the argument, aBlock,
  make the receiver be the cursor shape."
  | oldcursor |
  oldcursor := Sensor currentCursor.
  self show.
  ↑aBlock ensure: [ oldcursor show ]

```

The argument [oldcursor show] is evaluated whether or not aBlock signals an exception. Note that the result of ensure: is the value of the receiver, not that of the argument.

```
[ 1 ] ensure: [ 0 ] → 1 "not 0"
```

7.2 Handling non-local returns

The message ifCurtailed: is typically used for “cleaning” actions. It is similar to ensure:, but instead of ensuring that its argument block is evaluated even if the receiver terminates abnormally, ifCurtailed: does so *only* if the receiver fails or returns.

In the following example, the receiver of ifCurtailed: performs an early return, so the following statement is never reached. In Smalltalk, this is referred to as a *non-local return*. Nevertheless the argument block will be executed.

```
[↑ 10] ifCurtailed: [Transcript show: 'We see this'].
Transcript show: 'But not this'.
```

In the following example, we can see clearly that the argument to ifCurtailed: is evaluated only when the receiver terminates abnormally.

```
[Error signal] ifCurtailed: [Transcript show: 'Abandoned'; cr].
Transcript show: 'Proceeded'; cr.
```

 Open a transcript and evaluate the code above in a workspace. When the debugger window opens, first try selecting Proceed and then Abandon. Note that the argument to ifCurtailed: is evaluated only when the receiver terminates abnormally. What happens when you select Debug?

Here are some examples of ifCurtailed: usage: the text of the Transcript show: describes the situation:

```
[↑ 10] ifCurtailed: [Transcript show: 'This is displayed'; cr]
```

```
[10] ifCurtailed: [Transcript show: 'This is not displayed'; cr]
```

```
[1 / 0] ifCurtailed: [Transcript show: 'This is displayed after selecting Abandon in the debugger'; cr]
```

Although in Pharo `ifCurtailed:` and `ensure:` are implemented using a marker primitive (described at the end of the chapter), in principle `ifCurtailed:` could be implemented using `ensure:` as follows:

```
ifCurtailed: curtailBlock
| result curtailed |
curtailed := true.
[ result := self value.
  curtailed := false
] ensure: [ curtailed ifTrue: [ curtailBlock value ] ].  
↑ result
```

In a similar fashion, `ensure:` could be implemented using `ifCurtailed:` as follows:

```
ensure: ensureBlock
| result |
result := self ifCurtailed: ensureBlock.
"If we reach this point, then the receiver has not been curtailed, so ensureBlock still needs to be evaluated"
ensureBlock value.  
↑ result
```

Both `ensure:` and `ifCurtailed:` are very useful for making sure that important “cleanup” code is executed, but are not by themselves sufficient for handling all exceptional situations. Now let’s look at a more general mechanism for handling exceptions.

7.3 Exception handlers

The general mechanism is provided by the message `on:do:`. It looks like this:

```
aBlock on: exceptionClass do: handlerAction
```

`aBlock` is the code that detects an abnormal situation and signals an exception; it is called the *protected block*. `handlerAction` is the block that is evaluated if an exception is signaled; it is called the *exception handler*. `exceptionClass` defines the class of exceptions that `handlerAction` will be asked to handle.

The beauty of this mechanism lies in the fact that the protected block can be written in a straightforward way, *without regard to any possible errors*. A

single exception handler is responsible for taking care of anything that may go wrong.

Consider the following example, where we want to copy the contents of one file to another. Although several file-related things could go wrong, with exception handling we simply write a straight-line method, and define a single exception handler for the whole transaction:

```
source := 'log.txt'.
destination := 'log-backup.txt'.
[ fromStream := FileDirectory default oldFileNamed: source.
  [ toStream := FileDirectory default newFileNamed: destination.
    [ toStream nextPutAll: fromStream contents ]
    ensure: [ toStream close ] ]
  ensure: [ fromStream close ] ]
on: FileStreamException
do: [ :ex | UIManager default inform: 'Copy failed -- ', ex description ].
```

If any exception concerning `FileStreams` is raised, the handler block (the block after `do:`) is executed with the exception object as its argument. Our handler code alerts the user that the copy has failed, and delegates to the exception object `ex` the task of providing details about the error. Note the two nested uses of `ensure:` to make sure that the two file streams are closed, whether or not an exception occurs.

It is important to understand that the block that is the receiver of the message `on:do:` defines the scope of the exception handler. This handler will be used only if the receiver (*i.e.*, the protected block) has not completed. Once completed, the exception handler will not be used. Moreover, a handler is associated exclusively with the kind of exception specified as the first argument to `on:do:`. Thus, in the previous example, only a `FileStreamException` (or a more specific variant thereof) can be handled.

7.4 Error codes — don't do this!

Without exceptions, one (bad) way to handle a method that may fail to produce an expected result is to introduce explicit error codes as possible return values. In fact, in languages like C, code is littered with checks for such error codes, which often obscure the main application logic. Error codes are also fragile in the face of evolution: if new error codes are added, then all clients must be adapted to take the new codes into account. By using exceptions instead of error codes, the programmer is freed from the task of explicitly checking each return value, and the program logic stays uncluttered. Moreover, because exceptions are classes, as new exceptional situations are discovered, they can be subclassed; old clients will still work, although they may provide less-specific exception handling than newer clients.

If Smalltalk did not provide exception-handling support, then the tiny example we saw in the previous section would be written something like this, using error codes:

```
"Pseudo-code -- luckily Smalltalk does not work like this. Without the
benefit of exception handling we must check error codes for each operation."
source := 'log.txt'.
destination := 'log-backup.txt'.
success := 1. "define two constants, our error codes"
failure := 0.
fromStream := FileDirectory default oldFileNamed: source.
fromStream ifNil: [
    UIManager default inform: 'Copy failed -- could not open', source.
    ↑ failure "terminate this block with error code".
    toStream := FileDirectory default newFileNamed: destination.
    toStream ifNil: [
        fromStream close.
        UIManager default inform: 'Copy failed -- could not open', destination.
        ↑ failure ].
    contents := fromStream contents.
    contents ifNil: [
        fromStream close.
        toStream close.
        UIManager default inform: 'Copy failed -- source file has no contents'.
        ↑ failure ].
    result := toStream nextPutAll: contents.
    result ifFalse: [
        fromStream close.
        toStream close.
        UIManager default inform: 'Copy failed -- could not write to ', destination.
        ↑ failure ].
    fromStream close.
    toStream close.
    ↑ success.
```

What a mess! Without exception handling, we must explicitly check the result of each operation before proceeding to the next. Not only must we check error codes at each point that something might go wrong, but we must also be prepared to cleanup any operations performed up to that point and abort the rest of the code.

7.5 Specifying which Exceptions will be Handled

In Smalltalk, exceptions are, of course, objects. In Pharo, an exception is an instance of an exception class which is part of a hierarchy of exception classes. For example, because the exceptions

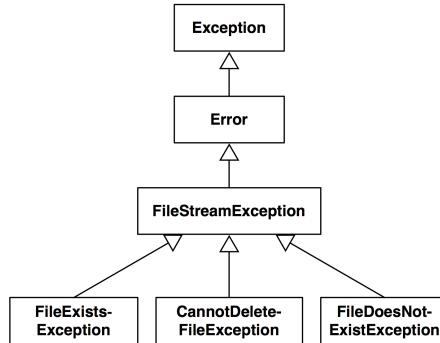


Figure 7.1: A small part of the Pharo exception hierarchy.

`FileDoesNotExistException`, `FileExistsException` and `CannotDeleteFileNotFoundException` are special kinds of `FileStreamException`, they are represented as subclasses of `FileStreamException`, as shown in Figure 7.1. This notion of “specialization” lets us associate an exception handler with a more or less general exceptional situation. So, we can write:

```
[ ... ] on: Error do: [ ... ]
[ ... ] on: FileStreamException do: [ ... ]
[ ... ] on: FileDoesNotExistException do: [ ... ]
```

The class `FileStreamException` adds information to class `Exception` to characterize the specific abnormal situation it describes. Specifically, `FileStreamException` defines the `fileName` instance variable, which contains the name of the file that signaled the exception. The root of the exception class hierarchy is `Exception`, which is a direct subclass of `Object`.

Two key messages are involved in exception handling: `on:do:`, which, as we have already seen, is sent to blocks to set an exception handler, and `signal`, which is sent to subclasses of `Exception` to signal that an exception has occurred.

7.6 Signaling an exception

To signal an exception¹, you only need to create an instance of the exception class, and to send it the message `signal`, or `signal:` with a textual description. The class `Exception` class provides a convenience method `signal`, which creates and signals an exception. So, here are two equivalent ways to signal a `ZeroDivide` exception:

¹Synonyms are to “raise” or to “throw” an exception. Since the vital message is called `signal`, we use that terminology exclusively in this chapter.

`ZeroDivide new signal.`

`ZeroDivide signal. "class-side convenience method does the same as above"`

You may wonder why it is necessary to create an instance of an exception in order to signal it, rather than having the exception class itself take on this responsibility. Creating an instance is important because it encapsulates information about the context in which the exception was signaled. We can therefore have many exception instances, each describing the context of a different exception.

When an exception is signaled, the exception handling mechanism searches in the execution stack for an exception handler associated with the class of the signaled exception. When a handler is encountered (*i.e.*, the message `on:do:` is on the stack), the implementation checks that the `exceptionClass` is a superclass of the signaled exception, and then executes the `handlerAction` with the exception as its sole argument. We will see shortly some of the ways in which the handler can use the exception object.

When signaling an exception, it is possible to provide information specific to the situation just encountered, as illustrated in the code below. For example, if the file to be opened does not exist, the name of the non-existent file can be recorded in the exception object:

```
StandardFileStream class»oldFileNamed: fileName
  "Open an existing file with the given name for reading and writing. If the name has no
   directory part, then default directory will be assumed. If the file does not exist, an
   exception will be signaled. If the file exists, its prior contents may be modified or
   replaced, but the file will not be truncated on close."
| fullName |
fullName := self fullName: fileName.
↑(self isAFileNamed: fullName)
  ifTrue: [self new open: fullName forWrite: true]
  ifFalse: ["File does not exist...
    (FileDoesNotExistException new fileName: fullName) signal]
```

The exception handler may make use of this information to recover from the abnormal situation. The argument `ex` in an exception handler `[:ex | ...]` will be an instance of `FileDoesNotExistException` or of one of its subclasses. Here the exception is queried for the filename of the missing file by sending it the message `fileName`.

```
| result |
result := [(StandardFileStream oldFileNamed: 'error42.log') contentsOfEntireFile]
  on: FileDoesNotExistException
  do: [:ex | ex fileName , ' not available'].
Transcript show: result; cr
```

Every exception has a default description that is used by the development tools to report exceptional situations in a clear and comprehensible manner. To make the description available, all exception objects respond to the message description. Moreover, the default description can be changed by sending the message messageText: *aDescription*, or by signaling the exception using signal: *aDescription*.

Another example of signaling occurs in the doesNotUnderstand: mechanism, a pillar of the reflective capabilities of Smalltalk. Whenever an object is sent a message that it does not understand, the VM will (eventually) send it the message doesNotUnderstand: with an argument representing the offending message. The default implementation of doesNotUnderstand:, defined in class Object, simply signals a MessageNotUnderstood exception, causing a debugger to be opened at that point in the execution.

The doesNotUnderstand: method illustrates the way in which exception-specific information, such as the receiver and the message that is not understood, can be stored in the exception, and thus made available to the debugger.

```
Object»doesNotUnderstand: aMessage
```

"Handle the fact that there was an attempt to send the given message to the receiver but the receiver does not understand this message (typically sent from the machine when a message is sent to the receiver and no method is defined for that selector)."

```
MessageNotUnderstood new
    message: aMessage;
    receiver: self;
    signal.
    ↑ aMessage sentTo: self.
```

That completes our description of how exceptions are used. The remainder of this chapter discusses how exceptions are implemented, and adds some details that are relevant only if you define your own exceptions.

7.7 How breakpoints are Implemented

As we discussed in the Debugger chapter of *Pharo By Example*, the usual way of setting a breakpoint within a Smalltalk method is to insert the message send self halt into the code. The method halt, implemented in Object, uses exceptions to open a debugger at the location of the breakpoint; it is defined as follows:

Object»halt

"This is the typical message to use for inserting breakpoints during debugging. It behaves like halt:, but does not call on halt: in order to avoid putting this message on the stack. Halt is especially useful when the breakpoint message is an arbitrary one."

Halt signal

Halt is a direct subclass of Exception. A Halt exception is *resumable*, which means that it is possible to continue execution after a Halt is signaled.

Halt overrides the defaultAction method, which specifies the action to perform if the exception is not caught (*i.e.*, there is no exception handler for Halt anywhere on the execution stack):

Halt»defaultAction

"No one has handled this error, but now give them a chance to decide how to debug it. If no one handles this then open debugger (see UnhandledError–defaultAction)"

UnhandledError signalForException: self

This code signals a new exception, UnhandledError, that conveys the idea that no handler is present. The defaultAction of UnhandledError is to open a debugger:

UnhandledError»defaultAction

"The current computation is terminated. The cause of the error should be logged or reported to the user. If the program is operating in an interactive debugging environment the computation should be suspended and the debugger activated."

↑ ToolSet debugError: exception.

A few messages later, the debugger opens:

StandardToolSet»debug: aProcess context: aContext label: aString contents: contents

fullView: aBool

↑ Debugger openOn: aProcess context: aContext label: aString contents: contents

fullView: aBool

7.8 How handlers are found

We will now take a look at how exception handlers are found and fetched from the execution stack when an exception is signaled. However, before we do this, we need to understand how the control flow of a program is internally represented in the virtual machine.

At each point in the execution of a program, the execution stack of the program is represented as a list of activation contexts. Each activation context represents a method invocation and contains all the information needed

for its execution, namely its receiver, its arguments, and its local variables. It also contains a reference to the context that triggered its creation, *i.e.*, the activation context associated with the method execution that sent the message that created this context. In Pharo, the class `MethodContext` models this information. The references between activation contexts link them into a chain: this chain of activation contexts is Smalltalk's execution stack.

Actually, there are two kinds of activation context in Pharo: `methodContext`s and `blockContexts`: the latter are used to represent the execution of blocks. They have a common superclass `ContextPart`. We will ignore this detail for now.

Suppose that we attempt to open a `FileStream` on a non-existent file from a `dolt`. A `FileDoesNotExistException` will be signaled, and the execution stack will contain `MethodContexts` for `dolt`, `oldFileName:`, and `signal`, as shown in Figure 7.2.

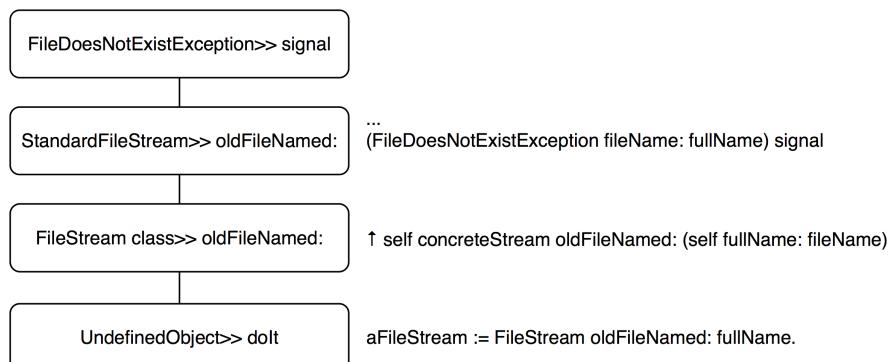


Figure 7.2: A Pharo execution stack.

Since everything is an object in Smalltalk, we would expect method contexts to be objects. However, some Smalltalk implementations use the native C execution stack of the virtual machine to avoid creating objects all the time. The current Pharo virtual machine does actually use full Smalltalk objects all the time; for speed, it recycles old method context objects rather than creating a new one for each message-send.

When we send `aBlock on: ExceptionClass do: actionHandler`, we intend to associate an exception handler (`actionHandler`) with a given class of exceptions (`ExceptionClass`) for the activation context of the protected block `aBlock`. This information is used to identify and execute `actionHandler` whenever an exception of an appropriate class is signaled; `actionHandler` can be found by traversing the stack starting from the top (the most recent message-send) and working down to the context that sent the `on:do:` message.

If there is no exception handler on the stack, the message `defaultAction` will

be sent either by ContextPart»handleSignal: or by UndefinedObject»handleSignal:. The latter is associated with the bottom of the stack, and is defined as follows:

UndefinedObject»handleSignal: exception

"When no more handler (on:do:) context is left in the sender chain, this gets called.

Return from signal with default action."

↑ exception resumeUnchecked: exception defaultAction

The message handleSignal: is sent by Exception»signal.

When an exception E is signaled, the system identifies and fetches the corresponding exception handler by searching down the stack as follows:

1. Look in the current activation context for a handler, and test if that handler canHandleSignal: E .
2. If no handler is found and the stack is not empty, go down the stack and return to step 1.
3. If no handler is found and the stack is empty, then send defaultAction to E . The default implementation in the Error class leads to the opening of a debugger.
4. If the handler is found, send it value: E .

Nested Exceptions. Exception handlers are outside of their own scope. This means that if an exception is signaled from within an exception handler—what we call a nested exception—a *separate* handler must be set to catch the nested exception.

Here is an example where one on:do: message is the receiver of another one; the second will catch errors signaled by the handler of the first:

```
result := [[ Error signal: 'error 1' ]
  on: Exception
  do: [ Error signal: 'error 2' ]]
  on: Exception
  do: [:ex | ex description ].
result   —→  'Error: error 2'
```

Without the second handler, the nested exception will not be caught, and the debugger will be invoked.

An alternative would be to specify the second handler within the first one:

```

result := [ Error signal: 'error 1' ]
on: Exception
do: [[ Error signal: 'error 2' ]
on: Exception
do: [:ex | ex description ]].
result    →  'Error: error 2'

```

7.9 Handling exceptions

When an exception is signaled, the handler has several choices about how to handle it. In particular, it may:

- (i) *abandon* the execution of the protected block, by simply specifying an alternative result;
- (ii) *return* an alternative result for the protected block, by sending `return aValue` to the exception object;
- (iii) *retry* the protected block, by sending `retry`, or try a different block by sending `retryUsing::`;
- (iv) *resume* the protected block at the failure point, by sending `resume` or `resume::`;
- (v) *pass* the caught exception to the enclosing handler, by sending `pass`;
- (vi) *resignal* a different exception, by sending `resignalAs:` to the exception.

We will briefly look at the first three possibilities, and then we will take a closer look at the remaining ones.

Abandon the protected block

The first possibility is to abandon the execution of the protected block, as follows:

```

answer := [ |result|
result := 6 * 7.
Error signal.
result "This part is never evaluated"
] on: Error
do: [ :ex | 3 + 4 ].
answer → 7

```

The handler takes over from the point where the error is signaled, and any code following in the original block is not evaluated.

Return a value with return:

A block returns the value of the last statement in the block, regardless of whether the block is protected or not. However, there are some situations where the result needs to be returned by the handler block. The message `return: aValue` sent to an exception has the effect of returning `aValue` as the value of the protected block:

```
result := [Error signal]
on: Error
do: [:ex | ex return: 3 + 4].
result → 7
```

The ANSI standard is not clear regarding the difference between using `do: [:ex | 100]` and `do: [:ex | ex return: 100]` to return a value. We suggest that you use `return:` since it is more intention-revealing, even if these two expressions are equivalent in Pharo.

A variant of `return:` is the message `return`, which returns `nil`.

Note that, in any case, control will *not* return to the protected block, but will be passed on up to the enclosing context.

```
6 * ([Error signal] on: Error do: [:ex | ex return: 3 + 4]) → 42
```

Retry a computation with retry and retryUsing:

Sometimes we may want to change the circumstances that led to the exception and retry the protected block. This is done by sending `retry` or `retryUsing:` to the exception object. It is important to be sure that the conditions that caused the exception have been changed before retrying the protected block, or else an infinite loop will result:

```
[Error signal] on: Error do: [:ex | ex retry] "will loop endlessly"
```

Here is a better example. The protected block is re-evaluated within a modified environment where `theMeaningOfLife` is properly initialized:

```
result := [ theMeaningOfLife * 7 ] "error -- theMeaningOfLife is nil"
on: Error
do: [:ex | theMeaningOfLife := 6. ex retry].
result → 42
```

The message `retryUsing: aNewBlock` enables the protected block to be replaced by `aNewBlock`. This new block is executed and is protected with the same handler as the original block.

```

x := 0.
result := [ x/x ]  "fails for x=0"
on: Error
do: [:ex |
  x := x + 1.
  ex retryUsing: [1/((x-1)*(x-2))]  "fails for x=1 and x=2"
].
result  →  (1/2)  "succeeds when x=3"

```

The following code loops endlessly:

```
[1 / 0] on: ArithmeticError do: [:ex | ex retryUsing: [ 1 / 0 ]]
```

whereas this will signal an Error:

```
[1 / 0] on: ArithmeticError do: [:ex | ex retryUsing: [ Error signal ]]
```

As another example, recall the file handling code we saw earlier, in which we printed a message to the Transcript when a file is not found. Instead, we could prompt for the file as follows:

```

[(StandardFileStream oldFileNamed: 'error42.log') contentsOfEntireFile]
on: FileDoesNotExistException
do: [:ex | ex retryUsing: [FileDialog modalFileSelector contentsOfEntireFile] ]

```

7.10 Resuming execution

A method that signals an exception that isResumable can be resumed at the place immediately following the signal. An exception handler may therefore perform some action, and then resume the execution flow. This behavior is achieved by sending resume: to the exception in the handler. The argument is the value to be used in place of the expression that signaled the exception. In the following example we signal and catch MyResumableTestError, which is defined in the Tests-Exceptions category:

```

result := [ | log |
  log := OrderedCollection new.
  log addLast: 1.
  log addLast: MyResumableTestError signal.
  log addLast: 2.
  log addLast: MyResumableTestError signal.
  log addLast: 3.
  log ]
  on: MyResumableTestError
  do: [ :ex | ex resume: 0 ].
result  →  an OrderedCollection(1 0 2 0 3)

```

Here we can clearly see that the value of MyResumableTestError signal is the value of the argument to the resume: message.

The message resume is equivalent to resume: nil.

The usefulness of resuming an exception is illustrated by the class Installer, which implements an automatic package loading mechanism. When installing packages, warnings may be signaled. Warnings should not be considered fatal errors, so the installer should simply ignore the warning and continue installing.

```
Installer»installQuietly: packageNameCollectionOrDetectBlock
  self package: packageNameCollectionOrDetectBlock.
  [ self install ] on: Warning do: [ :ex | ex resume ].
```

Another situation where resumption is useful is when you want to ask the user what to do. For example, suppose that we were to define a class ResumableLoader with the following method:

```
ResumableLoader»readOptionsFrom: aStream
  | option |
  [aStream atEnd]
    whileFalse: [option := self parseOption: aStream.
      "nil if invalid"
      option isNil
        ifTrue: [InvalidOption signal]
        ifFalse: [self addOption: option]].
  aStream close
```

If an invalid option is encountered, we signal an InvalidOption exception. The context that sends readOptionsFrom: can set up a suitable handler:

```
ResumableLoader»readConfiguration
  | stream |
  stream := self optionStream.
  [self readOptionsFrom: stream]
    on: InvalidOption
    do: [:ex | UIManager default confirm: 'Invalid option line. Continue loading?'
      ifTrue: [ex resume]
      ifFalse: [ex return]].
  stream close
```

Depending on user input, the handler in readConfiguration might return nil, or it might resume the exception, causing the signal message send in readOptionsFrom: to return and the parsing of the options stream to continue.

Note that InvalidOption must be resumable; it suffices to define it as a subclass of Exception.

Example: Deprecation

Deprecation offers a case study of a mechanism built using resumable exceptions. Deprecation is a software re-engineering pattern that allows us to mark a method as being “deprecated”, meaning that it may disappear in a future release and should not be used by new code. In Pharo, a method can be marked as deprecated as follows:

```
Utilities class>convertCRtoLF: fileName
    "Convert the given file to LF line endings. Put the result in a file with the extention '.lf'"

self deprecated: 'Use "FileStream convertCRtoLF: fileName" instead.'
on: '10 July 2009' in: #Pharo1.0 .
FileStream convertCRtoLF: fileName
```

When the message `convertCRtoLF:` is sent, if the `raiseDeprecatedWarnings` preference is true, then a pop-up window is displayed with a notification and the programmer may resume the application execution; this is shown in Figure 7.3.

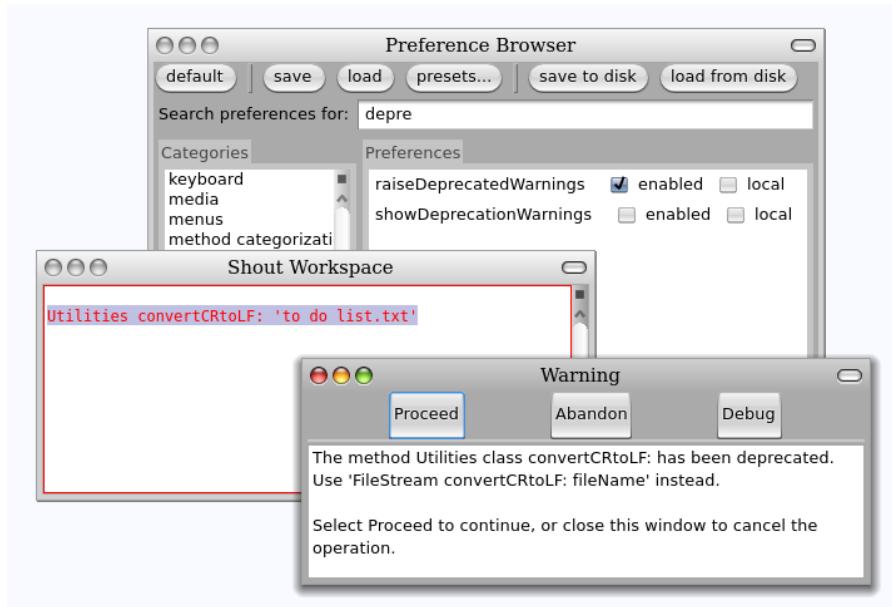


Figure 7.3: Sending a deprecated message.

Deprecation is implemented in Pharo in just a few steps. First, we define `Deprecation` as a subclass of `Warning`. It should have some instance variables to contain information about the deprecation: in Pharo these are `methodReference`, `explanationString`, `deprecationDate` and `versionString`; we therefore

need to define an instance-side initialization method for these variables, and a class-side instance creation method that sends the corresponding message.

When we define a new exception class, we should consider overriding `isResumable`, `description`, and `defaultAction`. In this case the inherited implementations of the first two methods are fine:

- `isResumable` is inherited from `Exception`, and answers true;
- `description` is inherited from `Exception`, and answers an adequate textual description.

However, it is necessary to override the implementation of `defaultAction`, because we want that to depend on some preferences. Here is Pharo's implementation:

```
Deprecation»defaultAction
Log ifNotNil: [:log| log add: self].
Preferences showDeprecationWarnings ifTrue:
    [Transcript nextPutAll: explanationString; cr; flush].
Preferences raiseDeprecatedWarnings ifTrue:
    [super defaultAction]
```

The first preference simply causes a warning message to be written on the Transcript. The second preference asks for an exception to be signaled, which is accomplished by super-sending `defaultAction`.

We also need to implement some convenience methods in `Object`, like this one:

```
Object»deprecated: anExplanationString on: date in: version
(Deprecation
method: thisContext sender method
explanation: anExplanationString
on: date
in: version) signal
```

7.11 Passing exceptions on

To illustrate the remaining possibilities for handling exceptions, we will look at how to implement a generalization of the `perform: method`. If we send `perform: aSymbol` to an object, this will cause the message named `aSymbol` to be sent to that object:

```
5 perform: #factorial → 120 "same as: 5 factorial"
```

Several variants of this method exist. For example:

```
1 perform: #+ withArguments: #(2) → 3 "same as: 1 + 2"
```

These `perform`-like methods are very useful for accessing an interface dynamically, since the messages to be sent can be determined at run-time.

One message that is missing is one that will send a cascade of unary messages to a given receiver. A simple and naive implementation is:

```
Object»performAll: selectorCollection
    selectorCollection do: [:each | self perform: each] "aborts on first error"
```

This method could be used as follows:

```
Morph new performAll: #( #activate #beTransparent #beUnsticky)
```

However, there is a complication. There might be a selector in the collection that the object does not understand (such as `#activate`). We would like to ignore such selectors and continue sending the remaining messages. The following implementation seems to be reasonable:

```
Object»performAll: selectorCollection
    selectorCollection do: [:each |
        [self perform: each]
        on: MessageNotUnderstood
        do: [:ex | ex return]] "also ignores internal errors"
```

On closer examination we notice another problem. This handler will not only catch and ignore messages not understood by the original receiver, but also any messages sent but not understood in methods for messages that *are* understood! This will hide programming errors in those methods, which is not our intent. To fix this, we need our handler to analyze the exception to see if it was indeed caused by the attempt to perform the current selector. Here is the correct implementation.

Method 7.1: `Object»performAll:`

```
Object»performAll: selectorCollection
    selectorCollection do: [:each |
        [self perform: each]
        on: MessageNotUnderstood
        do: [:ex | (ex receiver == self and: [ex message selector == each])
            ifTrue: [ex return]
            ifFalse: [ex pass]]] "pass internal errors on"
```

This has the effect of passing on `MessageNotUnderstood` errors to the surrounding context when they are not part of the list of messages we are performing. The `pass` message will pass the exception on to the next applicable handler in the execution stack.

If there is no next handler on the stack, the `defaultAction` message is sent to the exception instance. The `pass` action does not modify the sender chain in any way—but the handler that control is passed to may do so. Like the other messages discussed in this section, `pass` is special—it never returns to the sender.

The goal of this section has been to demonstrate the power of exceptions. It should be clear that while you can do almost anything with exceptions, the code that results is not always easy to understand. There is often a simpler way to get the same effect without exceptions; see method 7.2 on page 100 for a better way to implement `performAll`.

7.12 Resending exceptions

Now suppose that in our `performAll`: example we no longer want to ignore selectors not understood by the receiver, but instead we want to consider an occurrence of such a selector as an error. However, we want it to be signaled as an application-specific exception, let's say `InvalidAction`, rather than the generic `MessageNotUnderstood`. In other words, we want the ability to “resignal” a signaled exception as a different one.

It might seem that the solution would simply be to signal the new exception in the handler block. The handler block in our implementation of `performAll`: would be:

```
[:ex | (ex receiver == self and: [ex message selector == each])
  ifTrue: [InvalidAction signal]  "signals from the wrong context"
  ifFalse: [ex pass]]
```

A closer look reveals a subtle problem with this solution, however. Our original intent was to replace the occurrence of `MessageNotUnderstood` with `InvalidAction`. This replacement should have the same effect as if `InvalidAction` were signaled at the same place in the program as the original `MessageNotUnderstood` exception. Our solution signals `InvalidAction` in a different location. The difference in locations may well lead to a difference in the applicable handlers.

To solve this problem, resignaling an exception is a special action handled by the system. For this purpose, the system provides the message `resignalAs`:. The correct implementation of a handler block in our `performAll`: example would be:

```
[:ex | (ex receiver == self and: [ex message selector == each])
  ifTrue: [ex resignalAs: InvalidAction]  "resignals from original context"
  ifFalse: [ex pass]]
```

7.13 Comparing outer with pass

The method `outer` is very similar to `pass`. Sending `outer` to an exception also evaluates the enclosing handler action. The only difference is that if the outer handler resumes the exception, then control will be returned to the point where `outer` was sent, not the original point where the exception was signaled:

```
passResume := [[ Warning signal . 1 ]  "resume to here"
  on: Warning
  do: [ :ex | ex pass . 2 ]
    on: Warning
    do: [ :ex | ex resume ].
passResume  → 1  "resumes to original signal point"
```

```
outerResume := [[ Warning signal . 1 ]
  on: Warning
  do: [ :ex | ex outer . 2 ]]  "resume to here"
  on: Warning
  do: [ :ex | ex resume ].
outerResume  → 2  "resumes to where outer was sent"
```

7.14 Catching sets of exceptions

So far we have always used `on:do:` to catch just a single class of exception. The handler will only be invoked if the exception signaled is a sub-instance of the specified exception class. However, we can imagine situations where we might like to catch multiple classes of exceptions. This is easy to do:

```
result := [ Warning signal . 1/0 ]
  on: Warning, ZeroDivide
  do: [:ex | ex resume: 1 ].
result  → 1
```

If you are wondering how this works, just have a look at the implementation of `Exception class»`, `anotherException`,

`"Create an exception set."`

```
↑ExceptionSet new
  add: self;
  add: anotherException;
  yourself
```

The rest of the magic occurs in the class `ExceptionSet`, which has a surprisingly trivial implementation.

```
Object subclass: #ExceptionSet
instanceVariableNames: 'exceptions'
classVariableNames: ''
poolDictionaries: ''
category: 'Exceptions-Kernel'

ExceptionSet»initialize
super initialize.
exceptions := OrderedCollection new

ExceptionSet», anException
self add: anException.
↑self

ExceptionSet»add: anException
exceptions add: anException

ExceptionSet»handles: anException
exceptions do: [:ex | (ex handles: anException) ifTrue: [↑true]].
↑false
```

7.15 How exceptions are implemented

Let's have a look at how exceptions are implemented at the Virtual Machine level.

Storing Handlers. First we need to understand how the exception class and its associated handler are stored and how this information is found at run-time. Let's look at the definition of the central method `on:do:` defined on the class `BlockClosure`.

```
BlockClosure»on: exception do: handlerAction
"Evaluate the receiver in the scope of an exception handler."
| handlerActive |
<primitive: 199>
handlerActive := true.
↑self value
```

This code tells us two things: First, this method is implemented as a primitive, which means that a primitive operation of the virtual machine is executed when this method is invoked. VM primitives don't normally

return: successful execution of a primitive terminates the method that contains the <primitive: n> instruction, answering the result of the primitive. So, the Smalltalk code that follows the primitive serves two purposes: it documents what the primitive does, and is available to be executed if the primitive should fail. Here we see that `on:do:` simply sets the temporary variable `handlerActive` to true, and then evaluates the receiver (which is, of course, a block).

This is surprisingly simple, but somewhat puzzling. Where are the arguments of the `on:do:` method stored? Let's look at the definition of the class `MethodContext`, whose instances make up the execution stack:

```
ContextPart variableSubclass: #MethodContext
  instanceVariableNames: 'method closureOrNil receiver'
  classVariableNames: ''
  poolDictionaries: ''
  category: 'Kernel-Methods'
```

There is no instance variable here to store the exception class or the handler, nor is there any place in the superclass to store them. However, note that `MethodContext` is defined as a `variableSubclass`. This means that in addition to the named instance variables, objects of this class have some numbered slots. In fact, every `MethodContext` has a numbered slot for each argument of the method whose invocation it represents. There are also additional numbered slots for the temporary variables of the method.

To verify this, you can evaluate the following piece of code:

```
| exception handler |
[exception := thisContext sender at: 1.
handler := thisContext sender at: 2.
1 / 0]
on: Error
do: [:ex|].
{exception . handler} explore
```

The last line explores a 2-element array that contains the exception class and the exception handler.

Finding Handlers. Now that we know where the information is stored, let's have a look at how it is found at runtime.

We might think that the primitive 199 is complex to write. But it too is trivial, because primitive 199 *always* fails! Because the primitive always fails, the Smalltalk body of `on:do:` is always executed. However, the presence of the <primitive: 199> bytecode marks the executing context in a unique way.

The source code of the primitive is found in `Interpreter»primitiveMarkHandlerMethod` in the `VMMaker SqueakSource` project:

primitiveMarkHandlerMethod

"Primitive. Mark the method for exception handling. The primitive must fail after marking the context so that the regular code is run."

self inline: false.

↑self primitiveFail

So now we know that when the method `on:do:` is executed, the `MethodContext` that makes up the stack frame is tagged and the handler and exception class are stored there.

Now, if an exception is signaled further up the stack, the method `signal` can search the stack to find the appropriate handler:

Exception»signal

"Ask ContextHandlers in the sender chain to handle this signal."

The default is to execute and return my `defaultAction`."

signalContext := thisContext contextTag.

↑ thisContext nextHandlerContext handleSignal: self

ContextPart»nextHandlerContext

↑ self sender findNextHandlerContextStarting

The method `findNextHandlerContextStarting` is implemented as a primitive (number 197); its body describes what it does. It looks to see if the stack frame is a context created by the execution of the method `on:do:` (it just looks to see if the primitive number is 199). If this is the case it answers with that context.

ContextPart»findNextHandlerContextStarting

"Return the next handler marked context, returning nil if there is none. Search starts with self and proceeds up to nil."

| ctx |

<primitive: 197>

ctx := self.

[ctx isHandlerContext ifTrue: [↑ctx].

(ctx := ctx sender) == nil] whileFalse.

↑nil

MethodContext»isHandlerContext

"Is this context for method that is marked?"

↑method primitive = 199

Since the method context supplied by `findNextHandlerContextStarting` contains all the exception-handling information, it can be examined to see if

the exception class is suitable for handling the current exception. If so, the associated handler can be executed; if not, the look-up can continue further. This is all implemented in the handleSignal: method.

ContextPart»handleSignal: exception

"Sent to handler (on:do:) contexts only. If my exception class (first arg) handles exception then execute my handle block (second arg), otherwise forward this message to the next handler context. If none left, execute exception's defaultAction (see nil>>handleSignal:)."

```
| val |
(((self tempAt: 1) handles: exception) and: [self tempAt: 3]) ifFalse: [
    ↑ self nextHandlerContext handleSignal: exception].
```

exception privHandlerContext: self contextTag.
 self tempAt: 3 put: false. *"disable self while executing handle block"*
 val := [(self tempAt: 2) valueWithPossibleArgs: {exception}]
 ensure: [self tempAt: 3 put: true].
 self return: val. *"return from self if not otherwise directed in handle block"*

Notice how this method uses tempAt: 1 to access the exception class, and ask if it handles the exception. What about tempAt: 3? That is the temporary variable handlerActive of the on:do: method. Checking that handlerActive is true and then setting it to false ensures that a handler will not be asked to handle an exception that it signals itself. The return: message sent as the final action of handleSignal is responsible for “unwinding” the execution stack by removing the stack frames above self.

The full story is only slightly more complicated because there are actually two classes of objects that make up the stack, MethodContexts, which we have already discussed, and BlockContexts, which represent the execution of blocks. ContextPart is their common superclass.

So, to summarize, the signal method, with minimal assistance from the virtual machine, finds the context that correspond to an on:do: message with an appropriate exception class. Because the execution stack is made up of Context objects that may be manipulated just like any other object, the stack can be shortened at any time. This is a superb example of flexibility of Smalltalk.

7.16 Other kinds of Exception

The class Exception in Pharo has ten direct subclasses, as shown in Figure 7.4. The first thing that we notice from this figure is that the Exception hierarchy is a bit of a mess; you can expect to see some of the details change as Pharo is improved.

The second thing that we notice is that there are two large sub-hierarchies:

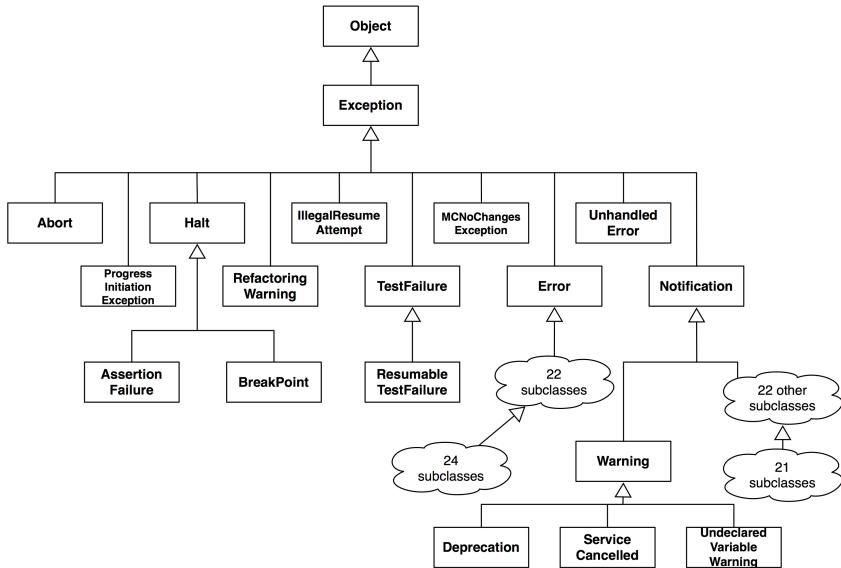


Figure 7.4: The whole Pharo exception hierarchy.

Error and Notification. Errors tell us that the program has fallen into some kind of abnormal situation. In contrast, Notifications tell us that an event has occurred, but without the assumption that it is abnormal. So, if a Notification is not handled, the program will continue to execute. An important subclass of Notification is Warning; warnings are used to notify other parts of the system, or the user, of abnormal but non-lethal behavior.

The property of being resumable is largely orthogonal to the location of an exception in the hierarchy. In general, Errors are not resumable, but 10 of its subclasses *are* resumable. For example, `MessageNotUnderstood` is a subclass of `Error`, but it is resumable. `TestFailures` are not resumable, but, as you would expect, `ResumableTestFailures` are.

Resumability is controlled by the private `Exception` method `isResumable`. For example:

```

Exception new isResumable   →  true
Error new isResumable     →  false
Notification new isResumable →  true
Halt new isResumable      →  true
MessageNotUnderstood new isResumable →  true

```

As it turns out, roughly 2/3 of all exceptions are resumable:

```

Exception allSubclasses size → 103
(Exception allSubclasses select: [:each | each new isResumable]) size → 66

```

If you declare a new subclass of exceptions, you should look in its protocol for the `isResumable` method, and override it as appropriate to the semantics of your exception.

In some situations, it will never make sense to resume an exception. In such a case you should signal a non-resumable subclass—either an existing one or one of your own creation. In other situations, it will always be OK to resume an exception, without the handler having to do anything. In fact, this gives us another way of characterizing a notification: a `Notification` is a resumable Exception that can be safely resumed without first modifying the state of the system. More often, it will be safe to resume an exception only if the state of the system is first modified in some way. So, if you signal a resumable exception, you should be very clear about what you expect an exception handler to do before it resumes the exception.

7.17 When not to use Exceptions

Just because Pharo has exception handling, you should not conclude that it is always appropriate to use it. Recall that in the introduction to this chapter, we said that exception handling is for *exceptional* situations. So, the first rule for using exceptions is *not* to use them for situations that *can reasonably be expected to occur* in a normal execution.

Of course, if you are writing a library, what is normal depends on the context in which your library is used. To make this concrete, let's look at `Dictionary` as an example: `aDictionary at: aKey` will signal an `Error` if `aKey` is not present. But you should not write a handler for this error! If the logic of your application is such that there is some possibility that the key will not be in the dictionary, then you should instead use `at: aKey ifAbsent: [remedial action]`. In fact, `Dictionary»at:` is implemented using `Dictionary»at:ifAbsent:.. aCollection detect: aPredicateBlock` is similar: if there is any possibility that the predicate might not be satisfied, you should use `aCollection detect: aPredicateBlock ifNone: [remedial action]`.

When you write methods that signal exceptions, you should consider whether you should also provide an alternative method that takes a remedial block as an additional argument, and evaluates it if the normal action cannot be completed. Although this technique can be used in any programming language that supports closures, because Smalltalk uses closures for *all* its control structures, it is a particularly natural one to use in Smalltalk.

Another way of avoiding exception handling is to test the precondition of the exception before sending the message that may signal it. For example, in method 7.1, we sent a message to an object using `perform:`, and handled the `MessageNotUnderstood` error that might ensue. A much simpler alternative is to check to see if the message is understood before executing the `perform:`

Method 7.2: *Object»performAll: revisited*

```
performAll: selectorCollection
    selectorCollection
        do: [:each | (self respondsTo: each)
            ifTrue: [self perform: each]]
```

The primary objection to method 7.2 is efficiency. The implementation of `respondsTo:` *s* has to lookup *s* in the target's method dictionary to find out if *s* will be understood. If the answer is yes, then `perform:` will look it up again. Moreover, the first lookup is implemented in Smalltalk, not in the virtual machine. If this code is in a performance-critical loop, this might be an issue. However, if the collection of messages comes from a user interaction, the speed of `performAll:` will not be a problem.

7.18 Chapter Summary

In this chapter we saw how to use exceptions to signal and handle abnormal situations arising in our code.

- Don't use exceptions as a control-flow mechanism. Reserve them for notifications and for *abnormal* situations. Consider providing methods that take blocks as arguments as an alternative to signaling exceptions.
- Use `protectedBlock ensure: actionBlock` to ensure that `actionBlock` will be performed even if `protectedBlock` terminates abnormally.
- Use `protectedBlock ifCurtailed: actionBlock` to ensure that `actionBlock` will be performed *only* if `protectedBlock` terminates abnormally.
- Exceptions are objects. Exception classes form a hierarchy with the class `Exception` at the root of the hierarchy.
- Use `protectedBlock on: ExceptionClass do: handlerBlock` to catch exceptions that are instances of `ExceptionClass` (or any of its subclasses). The `handlerBlock` should take an exception instance as its sole argument.
- Exceptions are signaled by sending one of the messages `signal` or `signal:.` `signal:` takes a descriptive string as its argument. The description of an exception can be obtained by sending it the message `description`.
- You can set a breakpoint in your code by inserting the message-send `self halt`. This signals a resumable Halt exception, which, by default, will open a debugger at the point where the breakpoint occurs.

- When an exception is signaled, the runtime system will search up the execution stack, looking for a handler for that specific class of exception. If none is found, the `defaultAction` for that exception will be performed (*i.e.*, in most cases the debugger will be opened).
- An exception handler may terminate the protected block by sending `return:` to the signaled exception; the value of the protected block will be the argument supplied to `return::`.
- An exception handler may retry a protected block by sending `retry` to the signaled exception. The handler remains in effect.
- An exception handler may specify a new block to try by sending `retryUsing:` to the signaled exception, with the new block as its argument. Here, too, the handler remains in effect.
- Notifications are subclass of `Exception` with the property that they can be safely resumed without the handler having to take any specific action.

Acknowledgments. We gratefully acknowledge Vassili Bykov for the raw material he provided. We also thank Paolo Bonzini, the main developer of GNU Smalltalk, for the Smalltalk implementations of `ensure:` and `ifCurtailed::`.

Chapter 8

Block and Dynamic Behavior of Smalltalk-Runtime

*with the participation of:
Stéphane Ducasse (stephane.ducasse@inria.fr)*

Blocks are a powerful and essential feature of Smalltalk. Without them it would be difficult to have a so small and compact syntax. The use of blocks in Smalltalk is the key to get conditional and loops not hardcoded in the language syntax but just simple messages.

In addition block are effective to improve the readability, reusability and efficiency of code. However, the dynamic runtime semantics of Smalltalk is often not well documented. Blocks in presence of return statements behave like an excepting mechanism.

In this chapter we will discuss some basic block behavior such as the notion of a static environment defined at block compiled time. But let us first recall some basics

8.1 Basics

What are a Block? A block is sustain piece of code. Historically, it's a Lambda expression, or a anonymous function. We can define a piece of code that is prepare to be call later. In Pharo the Block is define by square bracket. That block can be evaluate by sending the `#value` message to it.

`value value:`

`ifTrue:ifFalse:`

8.2 Variables and Blocks

In Smalltalk, private variables (such as self, instance variables, temporaries and arguments) are lexically scoped. These variables are bound in the context in which the block that contains them is defined, rather than the context in which the block is evaluated. We call the context (set of bindings) in which a block is defined, the block home context,

Define a class name BExp (for BlockExperience) and the following methods

```
BExp>>testScope
| t |
t := 42.
self testBlock: [Transcript show: t printString]

BExp>>testBlock: aBlock
| t |
t := nil.
aBlock value
```

Execute BExp new testScope

Evaluating the testScope message will print 42 in the Transcript. What you see is that the value of the temporary variable t defined in method testScope is the one used.

More blablablab here.

```
BExp>>testScope2
| t |
t := 42.
self testBlock: [t := 33.
Transcript show: t printString]

BExp>>testBlock: aBlock
| t |
t := nil.
aBlock value
```

This experience shows that a block is not only an anonymous method but one with an execution context or environment. In this environment temporary variables are bound with the values they hold when the block is defined. Naturally we can expect that method arguments are also bound and also self and instance variables of the class in which the method defining a block is. Let's illustrate these points now.

For method arguments

```
BExp>>testScopeArg: arg
```

```
"self new testScopeArg: 'foo'"  
  
self testScopeArgValue: [Transcript show: arg ; cr]  
  
BExp>>testScopeArgValue: aBlock  
| arg |  
arg := 'zork'.  
aBlock value
```

Now executing `self new testScopeArg: 'foo'` prints foo even if in the method `testScopeArgValue:` the temporary arg is redefined.

For binding of `self`, we can simply define a new class and a couple of method

Add the instance variable `x` to the class `BExp` and define the initialize method as follows:

```
BExp>>initialize  
x := 666.
```

Define another class

```
Object subclass: #BExp2  
instanceVariableNames: 'x'  
classVariableNames: ""  
poolDictionaries: ""  
category: 'BlockExperiment'
```

```
BExp2>>initialize  
x := 69.  
  
BExp2>>testScopeSelf: aBlock  
aBlock value
```

Then define the methods that will invoke the methods defined in `BExp2`.

```
BExp>>testScopeSelf  
"self new testScopeSelf"  
self testScopeSelf: [Transcript show: self printString ; show: x; cr]  
  
BExp>>testScopeSelf: aBlock  
BExp2 new testScopeSelf: aBlock
```

Now when we execute `BExp new testScopeSelf` and we see that a `BExp` gets printed, showing that a block captures `self`.

8.3 Basics on Return

A return statement allows one to return a different value than the receiver of the message. Return expressions behave

```
foo
#(1 2 3 4) do: [:each | Transcript show: each printString.
    each = 3
    ifTrue: [↑ 3]]
```

8.4 Returning from inside a block

It is not a really good idea to have return statement in a block that you pass or store into instance variables and we will explain why in this section.

In certain Smalltalk dialects, you can have expressions after a return statement in a block, in Pharo ↑ should be the last statement of a block body. You should get a compile error if you type and compile the following expression.

```
[ Transcript show: 'two'.
↑ self.
Transcript show: 'not printed' ]
```

Now to see that a return is really escaping the current execution you

```
testExplicitReturn
"self new testExplicitReturn a BExp"

Transcript show: 'one'.
0 isZero ifTrue: [ Transcript show: 'two'. ↑ self].
Transcript show: ' not printed'
-> one two
```

Simple block [:x :y| x*x. x+y] returns the value of the last statement to the method that sends it the message value

Continuation blocks [:x :y| ↑ x + y] returns the value to the method that activated its homeContext. As a block is always evaluated in its homeContext, it is possible to attempt to return from a method which has already returned using other return. This runtime error condition is trapped by the VM.

```
Object>>returnBlock
"self new returnBlock value -> error"

↑[↑self]
```

```
Object new returnBlock
-> Exception
```

```
|b|
b:= [:x| Transcript show: x. x].
b value: a. b value: b.
b:= [:x| Transcript show: x. ↑x].
b value: a. b value: b.
```

Continuation blocks cannot be executed several times!

```
Test>>testScope
|t|
t := 15.
self testBlock: [Transcript show: "--", t printString, "--".
↑35].
↑ 15
```

```
Test>>testBlock:aBlock
|t|
t := 50.
aBlock value.
self halt.
```

```
Test new testBlock
print: *15* and not halt.
return: 35
```

```
|val|
val := [:exit |
|goSoon|
goSoon := Dialog confirm: 'Exit now?'.
goSoon ifTrue: [exit value: 'Bye'].
Transcript show: 'Not exiting'.
'last value'] myValueWithExit.
Transcript show: val.
val
yes -> print Bye and return Bye
no -> print Not Exiting 2 and return 2
```

```
BlockClosure>>myValueWithExit
    self value: [:arg| ↑arg ].
    ↑ '2'
BlockClosure>>myValueWithExit
    ↑ self value: [:arg | ↑ arg]
```

8.5 Storing a block

8.6 may be to trash

VM represents the state of execution as Context objects for method Method-Context for block BlockContext

aContext contains a reference to the context from which it is invoked, the receiver arguments, temporaries in the Context

We call home context the context in which a block is defined

Arguments, temporaries, instance variables are lexically scoped in Smalltalk. These variables are bound in the context in which the block is defined and not in the context in which the block is evaluated.

8.7 Lexical Closure

Lexical closure is a concept introduced by SCHEME in 70s. Scheme uses lambda expression that basically a anonymous function (such the block). But using anonymous function imply to connect it to the current execution context. That why the lexical closure is important because it define when variables of block are bound to the execution context. The variable is depending of the scope where it's define. Let's illustrate that :

```
blockLocalTemp
| collection |
  collection := OrderedCollection new.
  1 to: 3 do: [ :index || temp |
    temp := index.
    collection add: [ temp ]].
  ↑collection collect: [:each | each value].
```

Let's comment the code, we create a loop the store the arg value, in a temporary variable created in the loop (then local) and change it in the loop. We store a block containing the simply temp read access in a collection. And after the loop, we evaluate each block and return the collection of value. If we evaluate this method that will return #(1 2 3). What's happen? At each loop we create a variable existing locally and bind it to a block. Then at the end evaluate block, we evaluate each block with this contextual *temp*.

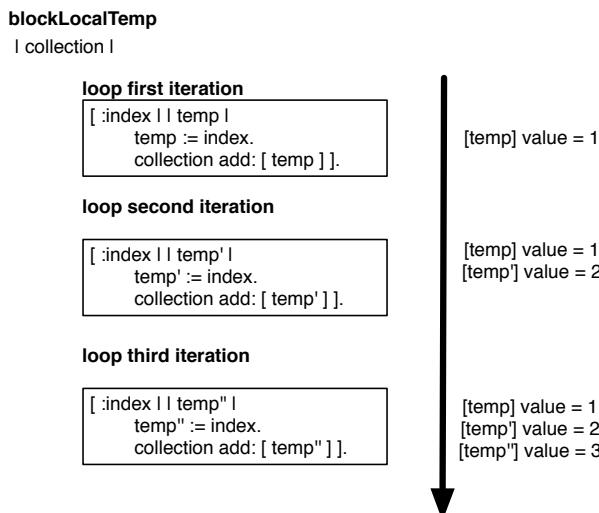


Figure 8.1: `blockLocalTemp` Execution

Now see another case :

```
blockOutsideTemp
| collection temp |
collection := OrderedCollection new.
1 to: 3 do: [ :index |
temp := index.
collection add: [ temp ]].
^collection collect: [:each | each value].
```

Same case except the *temp*, variable will be declare in the upper scope. Then what will happen ? Here the *temp* at each loop is the **same** shared variable bind. So when we collect the evaluation of the block at the end we will collect #(3 3 3).

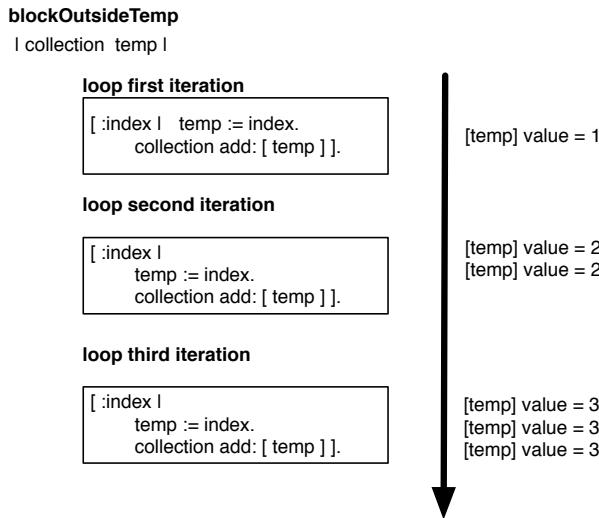


Figure 8.2: blockOutsideTemp Execution

When we look at the following Scheme expression and evaluate it you get 4. Indeed a binding is created which associates the variable `index` to the value 0. Then a lambda expression is defined and it returns the variable `index` (its value). Then within this context another expression is evaluated which starts with a `begin` statement: first the value of the variable `index` is set to 4. Second the lambda expression is evaluated. It returns then the value of the

```
(let* ((index 0)
      (y (lambda () index)))
(begin
```

```
(set index 4)(y))

(let ((index 0))
  (let ((y (lambda () index)))
    (begin
      (set index 4)(y)))))

((lambda (index)
  ((lambda () (begin
    (set index 4)index))))0)
```

What you see is that the lambda expression is sharing the binding (index 0) with expression (begin...) therefore when this binding is modified from the body of the begin expression, the lambda expression sees its impact and this is why it returns 4 and not 0 because.

8.8 Conclusion

I have a method that takes OBCommands and returns Actions:

```
actionsFrom: aCollectionOfOBCommandClasses on: aTarget for: aRequestor
| command |
↑ aCollectionOfOBCommandClasses collect: [ :each |
  command := each on: aTarget for: aRequestor.
  GLMAction new
    icon: command icon;
    title: command label;
    action: [:presentation | command execute ];
    yourself
]
```

These actions have a block that will be executed at a later time. The problem here was that the command in the action block was always pointing to the same command object, even at each point the command variable was populated correctly.

However, when the command is defined inside the block, everything works as expected.

```
actionsFrom: aCollectionOfOBCommandClasses on: aTarget for: aRequestor
| command |
↑ aCollectionOfOBCommandClasses collect: [ :each |
  command := each on: aTarget for: aRequestor.
  GLMAction new
    icon: command icon;
```

```
title: command label;
action: [:presentation | command execute ];
yourself
]
```

The semantics change in various ways. The trivial example that everyone knows is this:

```
factorial := [ :n |
  n > 1
    ifTrue: [ n * (factorial value: n - 1) ]
    ifFalse: [ 1 ]].
factorial value: 10.
```

Without closures you get an error, with closures you get the expected result.

Another significant change is the existence of local variables in blocks. Without closures blocks don't have local variables, with closures they do:

```
b := [ :p |
  | t |
  t ifNil: [ t := p ]].
{ b value: 1. b value: 2 }
```

In a non-closure image you get #(1 1) as result, because t is not a block local variable even if it looks like one. In a closure image you get #(1 2) because t is a block local variable.

```
testValueWithExitBreak
```

```
| val |
[ :break |
  1 to: 10 do: [ :i |
    val := i.
    i = 4 ifTrue: [break value].
  ]
] valueWithExit.
self assert: val = 4.
```

```
testValueWithExitContinue
```

```
| val last |
val := 0.
1 to: 10 do: [ :i |
  [ :continue |
    i = 4 ifTrue: [continue value].
    val := val + 1.
```

```
    last := i
] valueWithExit.
].
self assert: val = 9.
self assert: last = 10.

BlockClosure>>valueWithExit
    self value: [ ↑nil ]
```


Chapter 9

Fun with Floats

with the participation of:
Nicolas Cellier (nicolas.cellier.aka.nice@gmail.com)

Floats are inexact by nature and this can confuse programmers. In this chapter we present an introduction to this problem. The basic message is that Floats are what they are: inexact but fast numbers.

9.1 Never test equality on floats

The first basic principle is to never compare float equality. Let's take a simple case: the addition of two floats may not be equal to the float representing their sum. For example $0.1 + 0.2$ is not equal to 0.3 .

```
(0.1 + 0.2) = 0.3  
returns false
```

Hey, this is unexpected, you did not learn that in school, did you? This behavior is surprising indeed, but it's normal since floats are inexact numbers. What is important to understand is that the way floats are printed is also impacting our understanding. Some approaches prints a simpler representation of reality than others. In early versions of Pharo printing $0.1 + 0.2$ were printing 0.3 , now printing it returns 0.3000000000000004 . This change was guided by the idea that it is better not to lie to the user. Showing the inexactness of float is better than hiding because one day or another we can be deeply bitten by them.

```
(0.2 + 0.1) printString  
returns '0.3000000000000004'
```

```
0.3 printString
    returns '0.3'
```

We can see that we are in presence of two different numbers by looking at the hexadecimal values.

```
(0.1+0.2) hex
    returns '3FD3333333333334'
0.3 hex
    returns '3FD3333333333333'
```

The method `storeString` also conveys that we are in presence of two different numbers.

```
(0.1+0.2) storeString
    returns '0.30000000000000004'
0.3 storeString
    returns '0.3'
```

About `closeTo`: One way to know if two floats are probably close enough to look like the same number is to use the message `closeTo`:

```
(0.1 + 0.2) closeTo: 0.3
    returns true

0.3 closeTo: (0.1 + 0.2)
    returns true
```

About Scaled Decimals. There is a solution if you absolutely need exact floating point numbers, use Scaled Decimals. Scaled Decimals are exact numbers so they exhibit the behavior you expected.

```
0.1s2 + 0.2s2 = 0.3s2
    returns true
```

9.2 Dissecting a Float

To understand what operation is involved in above addition, we must know how Floats are represented internally in the computer: Pharo's Float format is a wide spread standard found on most computers - IEEE 754-1985 double precision on 64 bits (See http://en.wikipedia.org/wiki/IEEE_754-1985 for more details). In this format, a Float is represented in base 2 by this formula:

$$\text{sign} \cdot \text{mantissa} \cdot 2^{\text{exponent}}$$

- The sign is represented on 1 bit.
 - The exponent is represented on 11 bits.
 - The mantissa is a fractional number in base two, with a leading 1 before decimal point, and with 52 binary digits after fraction point.

For example, a series of 52 bits:

means the mantissa is:

which also represents the following fractions:

$$1 + \frac{0}{2} + \frac{1}{2^2} + \frac{1}{2^3} + \frac{0}{2^4} + \frac{0}{2^5} + \frac{1}{2^6} + \cdots + \frac{0}{2^{52}}$$

The mantissa value is thus between 1 (included) and 2 (excluded) for normal numbers.

Building a Float. Let us construct such a mantissa:

(#(0 2 3 6) detectSum: [:i | (2 raisedTo: i) reciprocal]) asFloat.
returns 1.390625

Now let us multiply by 2^3 to get a non null exponent:

(#(0 2 3 6) detectSum: [:i | (2 raisedTo: i) reciprocal]) asFloat timesTwoPower: 3.
returns 11.125

In Pharo, you can retrieve these informations:

11.125 sign.
returns 1

11.125 significand.
returns 1.390625

11.125 exponent.
returns 3

In Pharo, there is no message to handle the normalized mantissa directly. Instead it is possible to handle the mantissa as an Integer after a 52 bits shift to the left. There is one good reason for this: operating on Integer is easier because arithmetic is exact. The result includes the leading 1 and should thus be 53 bits long for a normal number (that's the float precision):

```
11.125 significandAsInteger highBit.
returns 53
```

```
Float precision.
returns 53
```

You can also retrieve the exact fraction corresponding to the internal representation of the Float:

```
11.125 asTrueFraction.
returns (89/8)
```

```
(#(0 2 3 6) detectSum: [:i | (2 raisedTo: i) reciprocal] * (2 raisedTo: 3).
returns (89/8)
```

Until there we retrieved the exact input we've injected into the Float. Are Float operations exact after all? Hem, no, we only played with fractions having a power of 2 as denominator and a few bits in numerator. If one of these conditions is not met, we won't find any exact Float representation of our numbers. In particular, it is not possible to represent $1/5$ with a finite number of binary digits. Consequently, a decimal fraction like 0.1 cannot be represented exactly with above representation.

```
(1/5) asFloat = (1/5).
returns false
```

```
(1/10) = 0.1
returns false
```

Let us see in detail how we could get the fractional bits of $1/10$ *i.e.*, $2r1/2r101$. For that, we must lay out the division:

1	101
10	0.00110011
100	
1000	
-101	
11	
110	
-101	
1	
10	
100	
1000	
-101	
11	
110	
-101	
1	

What we see is that we get a cycle: every 4 Euclidean divisions, we get a quotient 2r0011 and a remainder 1. That means that we need an infinite series of this bit pattern 0011 to represent $1/5$ in base 2. Let us see how Pharo dealt to convert $(1/5)$ to a Float:

```
(1/5) asFloat significandAsInteger printStringBase: 2.  
returns '11001100110011001100110011001100110011001100110011010'
```

```
(1/5) asFloat exponent.  
returns -3
```

That's the bit pattern we expected, except the last bits 001 have been rounded to upper 010. This is the default rounding mode of Float, round to nearest even. We now know why 0.2 is represented inexactly in machine. It's the same mantissa for 0.1, and its exponent is -4 .

So, when we entered $0.1 + 0.2$, we didn't get exactly $(1/10) + (1/5)$. Instead of that we got:

```
0.1 asTrueFraction + 0.2 asTrueFraction.  
returns (10808639105689191/36028797018963968)
```

But that's not all the story... Let us inspect the bit pattern of above fraction, and check the span of this bit pattern, that is the position of highest bit set to 1 (leftmost) and position of lowest bit set to 1 (rightmost):

```
10808639105689191 printStringBase: 2.  
returns '1001100110011001100110011001100110011001100110011001100111'
```

10808639105689191 highBit.
returns 54

10808639105689191 lowBit.
returns 1

The denominator is a power of 2 as we expect, but we need 54 bits of precision to store the numerator... Float only provides 53. There will be another rounding error to fit into Float representation:

(0.1 asTrueFraction + 0.2 asTrueFraction) asFloat = (0.1 asTrueFraction + 0.2 asTrueFraction).
returns false

To summarize what happened, including conversions of decimal representation to Float representation:

(1/10) asFloat	0.1	inexact (rounded to upper)
(1/5) asFloat	0.2	inexact (rounded to upper)
(0.1 + 0.2) asFloat	...	inexact (rounded to upper)

3 inexact operations occurred, and, bad luck, the 3 rounding operations were all to upper, thus they did cumulate rather than annihilate. On the other side, interpreting 0.3 is causing a single rounding error (3/10) asFloat. We now understand why we cannot expect $0.1 + 0.2 = 0.3$.

As an exercise, you could show why $1.3 * 1.3 \neq 1.69$.

9.3 With floats, printing is inexact

One of the biggest trap we learned with above example is that despite the fact that 0.1 is printed '0.1' as if it were exact, it's not. The name `absPrintExactlyOn:base:` used internally by `printString` is a bit confusing, it does not print exactly, but it prints the shortest decimal representation that will be rounded to the same `Float` when read back (Pharo always converts the decimal representation to the nearest `Float`).

Another message exists to print exactly, you need to use `printShowingDecimalPlaces:` instead. As every finite `Float` is represented internally as a `Fraction` with a denominator being a power of 2, every finite `Float` has a decimal representation with a finite number of decimal digits.

(just multiply numerator and denominator with adequate power of 5, and you'll get the digits). Here you go:

```
0.1 asTrueFraction denominator highBit.  
returns 56
```

This means that the fraction denominator is 2^{55} and that you need 55 decimal digits after the decimal point to really print internal representation of 0.1 exactly.

```
0.1 printShowingDecimalPlaces: 55.  
returns '0.100000000000000055511151231257827021181583404541015625'
```

And you can retrieve the digits with:

```
0.1 asTrueFraction numerator * (5 raisedTo: 55).  
returns 100000000000000055511151231257827021181583404541015625
```

You can just check our result with:

```
100000000000000055511151231257827021181583404541015625/(10 raisedTo: 55) =  
0.1 asTrueFraction  
returns true
```

You see that printing the exact representation of what is implemented in machine would be possible but would be cumbersome. Try printing $1.0e-100$ exactly if not convinced.

9.4 Float rounding is also inexact

While float equality is known to be evil, you have to pay attention to other aspects of floats. Let us illustrate that point with the following example.

```
2.8 truncateTo: 0.01  
returns 2.8000000000000003
```

```
2.8 roundTo: 0.01  
returns 2.8000000000000003
```

It is surprising but not false that `2.8 truncateTo: 0.01` does not return 2.8 but 2.8000000000000003. This is because `truncateTo:` and `roundTo:` perform several operations on floats: inexact operations on inexact numbers can lead to cumulative rounding errors as you saw above, and that's just what happens again.

Even if you perform the operations exactly and then round to nearest Float, the result is inexact because of the initial inexact representation of 2.8 and 0.01.

```
(2.8 asTrueFraction roundTo: 0.01 asTrueFraction) asFloat
returns 2.8000000000000003
```

Using 0.01s2 rather than 0.01 let this example appear to work:

```
2.80 truncateTo: 0.01s2
returns 2.80s2
```

```
2.80 roundTo: 0.01s2
returns 2.80s2
```

But it's just a case of luck, the fact that 2.8 is inexact is enough to cause other surprises as illustrated below:

```
2.8 truncateTo: 0.001s3.
returns 2.799s3
```

```
2.8 < 2.800s3.
returns true
```

Truncating in the Float world is absolutely unsafe. Though, using a ScaledDecimal for rounding is unlikely to cause such discrepancy, except when playing with last digits.

9.5 Fun with Inexact representations

To add a nail to the coffin, let's play a bit more with inexact representations. Let us try to see the difference between different numbers:

```
{
((2.8 asTrueFraction roundTo: 0.01 asTrueFraction) - (2.8 predecessor)) abs -> -1.
((2.8 asTrueFraction roundTo: 0.01 asTrueFraction) - (2.8)) abs -> 0.
((2.8 asTrueFraction roundTo: 0.01 asTrueFraction) - (2.8 successor)) abs -> 1.
} detectMin: [:e | e key]

returns the pair
0.0->1
```

The following expression returns 0.0->1, which means that: (2.8 asTrueFraction roundTo: 0.01 asTrueFraction) asFloat = (2.8 successor)

But remember that

```
(2.8 asTrueFraction roundTo: 0.01 asTrueFraction) ~= (2.8 successor)
```

It must be interpreted as the nearest Float to (2.8 asTrueFraction roundTo: 0.01 asTrueFraction) is (2.8 successor).

If you want to know how far it is, then get an idea with:

```
((2.8 asTrueFraction roundTo: 0.01 asTrueFraction) - (2.8 successor asTrueFraction))
    asFloat
returns -2.0816681711721685e-16
```

9.6 Conclusion

FLOATS ARE INEXACT NUMBERS. PAY ATTENTION WHEN YOU MANIPULATE THEM. THERE ARE MUCH MORE THINGS TO KNOW ABOUT FLOATS, AND IF YOU ARE ADVANCED ENOUGH, IT WOULD BE A GOOD IDEA TO CHECK THIS LINK FROM THE WIKIPEDIA PAGE "WHAT EVERY COMPUTER SCIENTIST SHOULD KNOW ABOUT FLOATING-POINT ARITHMETIC".

Part III

Web

Part IV

Libraries

Chapter 10

Files with FileSystem

with the participation of:

Stéphane Ducasse (stephane.ducasse@inria.fr)

A while ago Colin Putney announced the Filesystem framework, a nice and extensible replacement for the ugly FileDirectory classes in Pharo. While all core classes are well commented, there is a quick start missing that explains how end users are supposed to adopt the framework.

10.1 Getting started

First we need to load the package:

```
Gofer new
squeaksource: 'fs';
package: 'ConfigurationOfFileSystem';
load.
(Smalltalk at: #ConfigurationOfFileSystem) perform: #loadBleedingEdge
```

The framework supports different kinds of filesystems that can be used interchangeably and that can transparently work with each other. The most obvious one is the filesystem on your hard disk. We are going to work with that one for now:

```
| working |
working := FSFilesystem onDisk working.
```

Type the above code into a workspace and evaluate it. It assigns a reference of the current working directory to the variable `working`. References are the central object of the framework and provide the primary mechanism of

working with files and directories. They are instances of the class `FSReference`

Note that you do not use platform specific classes such as `FSUnixStore` or `FSWindowsStore`. All code below works on `FSReference` instances.

10.2 Navigating the Filesystem

Now let's do some more interesting things. To list children of your working directory evaluate the following expression:

```
working := FSFilesystem onDisk working.
working children.
```

To recursively access all the children of the current directory you can use `allChildren`

```
working allChildren.
```

To get only jpeg files you can for example

```
working allChildren select: [ :each | each basename endsWith: 'jpeg' ]
```

To get a reference to a specific file or directory within your working directory use the slash operator:

```
cache := working / 'package-cache'.
```

Navigating back to the parent is easy:

```
cache parent.
```

You can check for various properties of the cache directory by evaluating the following expressions:

<code>cache exists.</code>	<code>" → true"</code>
<code>cache isFile.</code>	<code>" → false"</code>
<code>cache isDirectory.</code>	<code>" → true"</code>
<code>cache basename.</code>	<code>" → 'package-cache'"</code>

To get additional information about the filesystem entry evaluate:

<code>cache entry creation.</code>	<code>" → 2010-02-14T10:34:31+00:00"</code>
<code>cache entry modification.</code>	<code>" → 2010-02-14T10:34:31+00:00"</code>
<code>cache entry size.</code>	<code>" → 0 (directories have size 0)"</code>

The framework also supports locations, late-bound references that point to a file or directory. When asking to perform a concrete operation, a location

behaves the same way as a reference. Currently the following locations are supported:

```
FSLocator desktop.  
FSLocator home.  
FSLocator image.  
FSLocator vmBinary.  
FSLocator vmDirectory.
```

If you save a location with your image and move the image to a different machine or operating system, a location will still resolve to the expected directory or file.

Opening Read- and Write-Streams

To open a file-stream on a file ask the reference for a read- or write-stream:

```
stream := (working / 'foo.txt') writeStream.  
stream nextPutAll: 'Hello World'.  
stream close.  
stream := (working / 'foo.txt') readStream.  
stream contents.  
stream close.
```

Please note that `writeStream` overrides any existing file and `readStream` throws an exception if the file does not exist. There are also short forms available:

```
working / 'foo.txt' writeStreamDo: [ :stream | stream nextPutAll: 'Hello World' ].  
working / 'foo.txt' readStreamDo: [ :stream | stream contents ].
```

Have a look at the streams protocol of `FSReference` for other convenience methods.

Renaming, Copying and Deleting Files and Directories

You can also copy and rename files by evaluating:

```
(working / 'foo.txt') copyTo: (working / 'bar.txt').
```

To create a directory evaluate:

```
backup := working / 'cache-backup'.  
backup createDirectory.
```

And then to copy the contents of the complete package-cache to that directory simply evaluate:

`cache copyAllTo: backup.` Note, that the target directory would be automatically created, if it was not there before.

To delete a single file evaluate:

(working / 'bar.txt') delete. To delete a complete directory tree use the following expression. Be careful with that one though.

backup deleteAll.

That's the basic API of the Filesystem library. If there is interest we can have a look at other features and other filesystem types in a next iteration.

working / 'foo.txt' readStreamDo: [:stream | stream nextPutAll: 'Hello World']. working / 'foo.txt' writeStreamDo: [:stream | stream contents].

10.3 Design

Path

Paths are the most fundamental element of the Filesystem API. They represent filesystem paths in a very abstract sense, and provide a high-level protocol for working with paths without having to manipulate Strings. Here are some examples using the methods that FSPath provides:

"absolute path"

FSPath / 'plonk' / 'feep' => /plonk/feep

"relative path"

FSPath * 'plonk' / 'feep' => plonk/feep

"relative path with extension"

FSPath * 'griffle' , 'txt' => griffle.txt

"changing the extension"

FSPath * 'griffle.txt' , 'jpeg' => griffle.jpeg

"parent directory"

(FSPath / 'plonk' / 'griffle') parent => /plonk

"resolving a relative path"

(FSPath / 'plonk' / 'griffle') resolve: (FSPath * '..' / 'feep')
=> /plonk/feep

"resolving an absolute path"

(FSPath / 'plonk' / 'griffle') resolve: (FSPath / 'feep')
=> /feep

"resolving a string"

(FSPath * 'griffle') resolve: 'plonk' => griffle/plonk

"comparing"

```
(FSPPath / 'plonk') contains: (FSPPath / 'griffle' / 'nurp')
=> false
```

Filesystem

A filesystem is an interface to some hierarchy of directories and files. "The filesystem," provided by the host operating system, is embodied by FSDiskFilesystem and it's platform-specific subclasses. But other kinds of Filesystems are also possible. FSMemoryFilesystem provides a RAM disk'a filesystem where all files are stored as ByteArrays in the image. FSZipFilesystem represents the contents of a zip file.

Each filesystem has its own working directory, which it uses to resolve any relative paths that are passed to it. Some examples:

```
fs := FSMemoryFilesystem new.
fs workingDirectory: (FSPPath / 'plonk').
griffle := FSPPath / 'plonk' / 'griffle'.
nurp := FSPPath * 'nurp'.

fs resolve: nurp      => /plonk/nurp

fs createDirectory: (FSPPath / 'plonk') => "/plonk created"
(fs writeStreamOn: griffle) close. => "/plonk/griffle created"
fs isFile: griffle.      => true
fs isDirectory: griffle => false
fs copy: griffle to: nurp => "/plonk/griffle copied to /plonk/nurp"
fs exists: nurp        => true
fs delete: griffle     => "/plonk/griffle" deleted
fs isFile: griffle     => false
fs isDirectory: griffle => false
```

Reference

Paths and filesystems are the lowest level of the Filesystem API. An FSReference combines a path and a filesystem into a single object which provides a simpler protocol for working with files. It implements the same operations as FSFilesystem , but without the need to track paths and filesystem separately:

```
fs := FSMemoryFilesystem new.
griffle := fs referenceTo: (FSPPath / 'plonk' / 'griffle').
nurp := fs referenceTo: (FSPPath * 'nurp').

griffle isFile
griffle isDirectory
```

```
griffle parent ensureDirectory.
griffle writeStreamDo: [:s]
griffle copyTo: nurp
griffle delete
```

References also implement the path protocol, with methods like `/`, `parent` and `resolve:`.

Locator

Locators could be considered late-bound references. They're left deliberately fuzzy, and only resolved to a concrete reference when some file operation needs to be performed. Instead of a filesystem and path, locators are made up of an origin and a path. An origin is an abstract filesystem location, such as the user's home directory, the image file, or the VM executable. When it receives a message like `isFile`, a locator will first resolve its origin, then resolve its path against the origin.

Locators make it possible to specify things like "an item named 'package-cache' in the same directory as the image file" and have that specification remain valid even if the image is saved and moved to another directory, possibly on a different computer.

```
locator := FSLocator image / 'package-cache'.
locator printString      => '{image}/package-cache'
locator resolve          => /Users/colin/Projects/Mason/package-cache
locator isFile            => false
locator isDirectory       => true
```

The following origins are currently supported:

- `image` - the image file changes - the changes file
- `vmBinary` - the executable for the running virtual machine
- `vmDirectory` - the directory containing the VM application (may not be the parent of `vmBinary`)
- `home` - the user's home directory
- `desktop` - the directory that hold the contents of the user's desktop documents - the directory where the user's documents are stored

Applications may also define their own origins, but the system will not be able to resolve them automatically. Instead, the user will be asked to manually choose a directory. This choice is then cached so that future resolution requests won't require user interaction.

Enumeration

References and Locators also provide simple methods for dealing with whole directory trees:

allChildren. This will answer an array of references to all the files and directories in the directory tree rooted at the receiver. If the receiver is a file, the array will contain a single reference, equal to the receiver.

allEntries. This method is similar to `allChildren`, but it answers an array of `FSDirectoryEntries`, rather than references.

copyAllTo: aReference. This will perform a deep copy of the receiver, to a location specified by the argument. If the receiver is a file, the file will be copied; if a directory, the directory and its contents will be copied recursively. The argument must be a reference that doesn't exist; it will be created by the copy.

deleteAll. This will perform a recursive delete of the receiver. If the receiver is a file, this has the same effect as `delete`.

Visitors

The above methods are sufficient for many common tasks, but application developers may find that they need to perform more sophisticated operations on directory trees.

The visitor protocol is very simple. A visitor needs to implement `visitFile:` and `visitDirectory:`. The actual traversal of the filesystem is handled by a guide. A guide works with a visitor, crawling the filesystem and notifying the visitor of the files and directories it discovers. There are three Guide classes, `FSPreorderGuide`, `FSPostorderGuide` and `FSBreadthFirstGuide`, which traverse the filesystem in different orders. To arrange for a guide to traverse the filesystem with a particular visitor is simple. Here's an example:

```
FSBreadthFirstGuide show: aReference to: aVisitor
```

The enumeration methods described above are implemented with visitors; see `FSCopyVisitor`, `FSDeleteVisitor`, and `FSCollectVisitor` for examples.

Chapter 11

Announcements: an Object Dependency Framework

11.1 A word about Component Coupling

Here is an interesting question that often comes up often when writing components. It is one that we faced when embedding our components. How do the components communicate with each other in a way that doesn't bind them together explicitly? That is, how does a child component send a message to its parent component without explicitly knowing who the parent is? Designing a component to refer to its parent is just a part of the solution since the interfaces of different parents may be different which would prevent the component from being reused in different contexts.

There is a solution based on explicit dependencies also called the change/update mechanism. Since early versions of Smalltalk, a dependency mechanism based on a change/update protocol is available and it is the foundation of the MVC framework itself. A component registers its interest in some event and that event triggers a notification.

There is a new framework called Announcement developed originally by Vassili Bykov which has been ported to several Smalltalk implementations. While the original dependency framework relied on symbols for the event registration and notification, Announcement promotes an object-oriented solution. Events are plain objects.

The main idea behind the framework is to set up announcers, define or reuse some announcements, let clients register interest in events and signal events. An event is an object representing an occurrence of a specific event. It is the place to define all the information related to the event occurrence. An announcer is responsible for registering interested clients and announcing

events.

An Example. Here is an example taken from Ramon Leon's very good Smalltalk blog. This example shows how we can use announcements to manage the communication between a parent component and its children as for example in the context of a menu and its menu items.

Method 11.1: Defining an announcer.

```
MySession>>announcer
  ↑ announcer ifNil: [announcer := Announcer new]
```

Then subclass Announcement for any interesting thing that might happen. The announcement subclass is the place to add any extra information about the specific announcement such as a context, the objects involved etc. This is why announcement objects are both more powerful and simpler than using symbols.

Class 11.2: Defining a specific Announcement.

```
Announcement subclass: #RemoveChild
  instanceVariableNames: 'child'

RemoveChild class>>child: aChild
  ↑self new
  child: aChild;
  yourself

RemoveChild>>child: anChild
  child := anChild

RemoveChild>>child
  ↑child
```

Any component interested in this announcement registers its interest using the method `on: anAnnouncementClass do: aBlock` or `on: anAnnouncementClass send: aSelector to: anObject`. The messages `on:do:` and `on:send:to:` are strictly equivalent to the messages `subscribe: anAnnouncementClass do: aValueable` and `subscribe: anAnnouncementClass send: aSelector to: anObject`. You can also ask an announcer to `unsubscribe: an object`.

In the following example, when a parent component is created, it expresses interest in the `RemoveChild` event and specifies the action that it will perform when such an event happens.

Method 11.3:

```
Parent>>initialize
  super initialize.
```

```
self session announcer on: RemoveChild do: [:it | self removeChild: it child]
```

```
Parent>>removeChild: aChild  
    self children remove: aChild
```

And any component that wants to fire this event simply announces it by sending in an instance of that custom announcement object.

Method 11.4: *Announcing an event.*

```
Child>>removeMe  
    self session announcer announce: (RemoveChild child: self)
```

Note that depending on where you place the announcer, you can even have different sessions sending events to each other, or different applications.

Announcements are not always the best way to establish communication between components. On one hand, announcements let you create loosely coupled components and thus maximize reusability. On the other hand, they introduce additional complexity when you may be able solve your communication problem with a simple message send.

Chapter 12

Sockets

with the participation of:
Noury Bouraqadi (bouraqadi@gmail.com)

Modern application are often distributed on multiple devices and collaborate through a network to achieve some task. The basic approach to set up such a collaboration is to use *sockets*. A typical use is in the World Wide Web. Browsers and servers interact through HTTP sockets.

The concept of socket was first introduced by researchers from Berkeley University in the 1960's. They defined the first socket API for the C programming language in the context of Unix operating systems. Since then, the concept of socket was spread out to other operating systems. Its API was ported to most existing programming languages including Smalltalk.

In this chapter, we present the socket's API in the context of Pharo Smalltalk. We first show through some examples how to use sockets for building both clients and servers. Then, we'll introduce `SocketStream` and how to use it. In practice, one is likely to use `SocketStream` instead of plain sockets.

12.1 Basic Concepts

Socket

A remote communication involves at least two system processes exchanging some data bytes through a network. Each process accesses the network through at least one socket (see Figure 12.1). A socket can then be defined as a *plug on a communication network*.

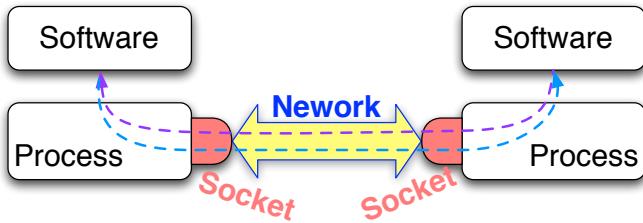


Figure 12.1: Inter-Process Remote Communication Through Sockets.

Sockets are used to achieve a bidirectional communication. They allow both sending and receiving data. Such interaction can be done according to communication protocols which are encapsulated in sockets. On the Internet and other networks such as ethernet LANs¹, two basic protocols widely used are *TCP/IP* and *UDP/IP*.

TCP/IP vs. UDP/IP

TCP/IP stands for *Transmission Control Protocol / Internet Protocol* (*TCP* for short). *TCP* uses guarantees a reliable communication (no data loss). It requires that applications involved in the communication get connected before actually communicating. Once a connection is established interacting parties can send and receive an arbitrary amount of bytes. This is often referred to as a *stream communication*. Data reach the destination in the same order of their sending.

UDP/IP stands for *User Datagram Protocol / Internet Protocol* (*UDP* for short). Datagrams are chunks of data which size can not exceed 64KB. *UDP* is an unreliable protocol because of two reasons. First, *UDP* does not guarantee that datagrams will actually reach their destination. The second reason why *UDP* is qualified as unreliable is that the reception order of multiple datagrams from a single sender to some particular receiver may arrive in an arbitrary order. However, *UDP* is faster than *TCP* since no connection is required before sending data. A typical use of *UDP* is “heart-beating” as used in server-based social application, where clients need to notify the server their status (e.g., Requesting interactions, or Invisible).

In a future version we will present UDP. But, so far UDP does not work in Pharo. In the remainder of this chapter we will focus on *TCP Sockets*. We first will present how to develop a simple client application using a socket

¹Local Area Networks.

12.2 A Simple TCP Client

We call *TCP client* an application that initiates a TCP connection to exchange data with another application: the *server*. It is important to mention that the client and the server may be developed in different languages. The life-cycle of such a client in Pharo decomposes into 4 steps:

1. Create a TCP socket.
2. Connect the socket to some server.
3. Exchange data with the server through the socket.
4. Close the socket.

Create a TCP Socket

Pharo provides a single socket class. At creation, the socket type (TCP or UDP) is provided to the socket plugin of the virtual machine. To create a TCP socket, you need to evaluate the following message

```
Socket newTCP
```

Method `newTCP` hands out the `tcp` type (which is stored in the class variable `TCPSocketType`) to set up a TCP socket.

Connect a TCP Socket to some Server

To connect a TCP Socket to a server, you need to have the object representing the IP address of that server. This address is a instance of `SocketAddress`. A handy way to create it is to use `NetNameResolver` that provides IP style network name lookup and translation facilities.

Script 12.1 provides two examples of socket address creation. The first one creates an address from a string describing the server name ('`www.esug.org`'), while the second does the creation from a string representing the server's IP address ('`127.0.0.1`'). Note that to use the `NetNameResolver` you need to have your machine connected to a network with a DNS². The only exception is for retrieving the local host address, i.e. `127.0.0.1` which is the generic address to refer to the machine that runs your software (Pharo in our case).

There are dedicated methods for retrieving the localhost address but they are buggy at the time of writing this text. The `NameLookupFailure` exception is used ONLY in `NetNameResolver class»addressForName:timeout:`. It should be more widely used.

²*Domain Name System*: basically a directory that maps device names to their IP address.

Script 12.1: Creating a Socket Address

```
| esugAddress localAddress |
esugAddress := NetNameResolver addressForName: 'www.esug.org'.
localAddress := NetNameResolver addressForName: '127.0.0.1'.
```

Now we can connect our TCP socket to the server as shown in Script 12.2. Message `connectTo:port:` attempts to connect the socket to the server which address and port are provided as parameters. ***SocketAddress can also store the port of the server. However, NetNameResolver does NOT support building socket address WITH port.***

Script 12.2: Connecting a TCP Socket to ESUG Server.

```
| clientSocket serverAddress |
clientSocket := Socket newTCP.
serverAddress := NetNameResolver addressForName: 'www.esug.org'.
clientSocket
    connectTo: serverAddress port: 80;
    waitForConnectionFor: 10.
clientSocket isConnected
```

The `connectTo:port:` message returns immediately after issuing to the system (through a primitive call) the request to connect the socket. Message `waitForConnectionFor: 10` suspends the current process until the socket is connected to the server. It waits at most 10 seconds as requested by the parameter. If the socket is not connected after 10 seconds, the `ConnectionTimedOut` exception is signaled. Otherwise, the execution can carry on by evaluating the expression `clientSocket isConnected` which obviously answers true.

Exchange Data with Server

Once connection is established, the client can send/receive data to/from the server. By data we mean a `ByteString`³. Typically, the client sends some request to the server and then expects receiving some response. Web browsers act according to this schema. A web browser is a client that issues a request to some web server identified by the URL. Such request is often the path to some resource on the server such as an html file or a picture. Then, the browser awaits the server response (e.g., html code, picture bytes).

Script 12.3: Exchanging Data with some Server through a TCP Socket.

```
|clientSocket data|
... ``create and connect the TCP clientSocket"
clientSocket sendData: 'Hello server'.
data := clientSocket receiveData.
... ``Process data"
```

³One can send a `ByteArray`, but the receiving socket will return a `ByteString` however.

Script 12.3 shows the protocol to send and receive data through a client socket. Here, we send the string 'Hello server!' to the server using the `sendData:` message. Next, we send the `receiveData` to our client socket to make it wait for data reception. Then, the contents of variable `data` is processed.

Script 12.4: Bounding the Maximum Time for Data Reception.

```
|clientSocket data|
... ``create and connect the TCP clientSocket"
[data := clientSocket receiveDataTimeout: 5.
... ``Process data"
] on: ConnectionTimedOut
do: [:timeOutException|
Transcript
cr;
show: 'No data received!';
space;
show: 'Network connection is too slow or server is down.']


```

Note that by using `receiveData`, the client waits until the server either sends no more data, or close the connection. This means that the client may wait indefinitely. An alternative is to have the client signal a `ConnectionTimedOut` exception if it had waited too much as shown in Script 12.4. We use message `receiveDataTimeout:` to ask the client socket to wait for 5 seconds. If data is received during this period of time, it is processed silently. But if no data is received during the 5 seconds, a `ConnectionTimedOut` is signaled. So, we log a description of what happened on the Transcript.

Close a Socket

A socket remains alive while devices at both ends are connected. Once the interaction is over, either the server or the client can decide to close the socket. This can be done by sending the `close` message to the socket. The image where this message is evaluated will then send a close request to the other side. The socket remains connected until the other side closes it. However, this may last indefinitely when there is an network failure or when the other side is down. This is why sockets also answer the `destroy` message, which frees system resources required by the socket.

In practice we use `closeAndDestroy`. It first attempts to close the socket by sending the `close` message. Then, if the socket is still connected after a duration of 20 seconds, the socket is destroyed. Note that there exist a variant `closeAndDestroy: seconds` which gets as a parameter the duration to wait before destroying the socket.

Script 12.5: Closing a TCP Socket After Connection to a Web Site.

```
| clientSocket serverAddress httpQuery htmlText |
```

```

httpQuery := 'GET / HTTP/1.1', String crlf,
  'Host: www.esug.org:80', String crlf,
  'Accept: text/html', String crlfcrlf.
Transcript cr; cr; show: 'Attempt to get a web page...'.
serverAddress := NetNameResolver addressForName: 'www.esug.org'.
clientSocket := Socket newTCP.
[clientSocket
  connectTo: serverAddress port: 80;
  waitForConnectionFor: 10.
clientSocket sendData: httpQuery.
htmlText := clientSocket receiveDataTimeout: 5.
Transcript cr; show: htmlText.
] ensure: [clientSocket closeAndDestroy].
Transcript cr; show: '...Done'

```

To summarize all steps described so far, we use the example of getting a web page from a server in Script 12.5. First, we retrieve the IP address of the www.esug.org server. Then, we create a TCP socket and connect it to the server. We use the IP address we get in the previous step and the default port for web servers: 80. Next we forge the HTTP⁴ query. The string corresponding to our query starts with the GET keyword, followed by a slash saying that we would like to get the root file of the server. Follows the protocol version HTTP/1.1. The second line recalls the host name and port. The third and last line of the HTTP query refers to format accepted by our client. Since, we intend to display the result of our query on the Transcript, we state that our client accepts texts with html format. After sending the http query, we wait at most 5 seconds for the html text that we display on the Transcript. Socket connection, query sending and html reception are inside a block which execution is ensured to end with cleaning up socket related resources, by means of the closeAndDestroy message.

12.3 A Simple TCP Server

Now let us build a simple TCP server. A *TCP Server* is an application that awaits TCP connections from TCP clients. Once connection established, both the server and the client can send a receive data in any order. A big difference between the server and the client is that the server uses at least two sockets. One socket is used for handling client connections, while the second serves for exchanging data with a particular client.

⁴HyperText Transfer Protocol used for web communications.

TCP Socket Server Life-cycle

The life-cycle of a TCP server in Pharo has 5 steps:

1. Create a first TCP socket, let's call it $socket_1$.
2. Wait for connections by making $socket_1$ listen on some port.
3. Accept a client request for connection. As a result, $socket_1$ will build a second socket, let's call it $socket_2$.
4. Exchange data with the client through $socket_2$. In the meanwhile, $socket_1$ can continue to wait for connections, and possibly create new sockets to exchange data with other clients.
5. Close $socket_2$.
6. Close $socket_1$ when we decide to kill the server and stop accepting client connections.

Concurrency is implicit in this life-cycle. The server listens for incoming client's connection requests through $socket_1$, while exchanging data with some clients through $socket_2$. The server can even simultaneously exchange data with multiple clients through different sockets. In the following, we first illustrate the socket serving machinery. Then, we give a complete server class and explain the server's life-cycle and related concurrency issues.

Serving Basic Example

We illustrate the serving basics through a simple example of an echo TCP server that accepts a single client request. It sends back to clients whatever data it received and quits. The code is provided by Script 12.6.

Script 12.6: Basic Echo Server.

```
| connectionSocket interactionSocket |
connectionSocket := Socket newTCP.
connectionSocket listenOn: 9999 backlogSize: 10.
interactionSocket := connectionSocket waitForAcceptFor: 60.
connectionSocket closeAndDestroy.
receivedData := interactionSocket receiveData.
Transcript cr; show: receivedData.
interactionSocket sendData: 'ECHO: ', receivedData.
interactionSocket closeAndDestroy.
```

First, we create the socket that we will use for handling incoming connections. We set it up to listen on port 9999. The backlogSize is set to 10,

meaning that we ask the Operating System to allocate a buffer for 10 connection requests. This backlog will not be actually used in this example. But, a more realistic server will have to handle multiple connections and then store pending connection requests into the backlog.

Once the connection socket is prepared, it starts listening for client connections. The `waitForAcceptFor: 60` message makes the socket wait connection requests for 60 seconds. If no client attempts to connect during these 60 seconds, the message answers `nil`. Otherwise, we get a new socket `interactionSocket` connected to the client's socket. At this point, we don't need the connection socket anymore, so we can close it (`closeAndDestroy` message).

Since the interaction socket is already connected to the client, we can use it to exchange data. Messages `receiveData` and `sendData:` presented above (see section 12.2) can be used to this end. In our example, we wait for data from the client, next we display it on the Transcript, and last we send it back to the client prefixed with the '`ECHO:`' string. Last, we finish the interaction with the client by closing the interaction socket.

Script 12.7: Echo Client.

```
| clientSocket serverAddress echoString |
serverAddress := NetNameResolver addressForName:'127.0.0.1'.
clientSocket := Socket newTCP.
[clientSocket
    connectTo: serverAddress port: 9999;
    waitForConnectionFor: 10.
clientSocket sendData: 'Hello Pharo!'.
echoString := clientSocket receiveDataTimeout: 5.
Transcript cr; show: echoString.
] ensure: [clientSocket closeAndDestroy].
```

You cannot test this script totally without having a second image that will execute the client code. Therefore use two different images, one that runs the server code and one for the client code. Indeed, since we use the user interaction process, the Pharo UI will be frozen at some points, such as during the `waitForAcceptFor:..`. Script 12.7 provides the code to run on the client image. Note that you have to run the server code first. Otherwise, the client will fail.

Echo Server Class

We define here the `EchoServer` class that deals with concurrency issues. It handles concurrent client queries and it does not freeze the UI. As we can see in the definition labelled as class 12.8, the `EchoServer` declares three instance variables. The first one (`connectionSocket`) refers to the socket used for listening to client connections. The two last instance variables (`isRunning` and

`isRunningLock`) are used to manage the server process life-cycle while dealing with synchronization issues.

Class 12.8: EchoServer *Class Definition*

```
Object subclass: #EchoServer
instanceVariableNames: 'connectionSocket isRunning isRunningLock'
classVariableNames: ''
poolDictionaries: ''
category: 'SimpleSocketServer'
```

Method 12.9: *The EchoServer»initialize Method*

```
EchoServer»initialize
super initialize.
isRunningLock := Mutex new.
self isRunning: false
```

Method 12.10: *The EchoServer»isRunning Read Accessor*

```
EchoServer»isRunning
↑isRunningLock critical: [isRunning]
```

Method 12.11: *The EchoServer»isRunning: Write Accessor*

```
EchoServer»isRunning: aBoolean
isRunningLock critical: [isRunning := aBoolean]
```

The `isRunning` instance variable is a flag that is set to true while the serving is running. As we'll see below, it can be accessed by different processes. Therefore, we use the mutex (see method 12.9) referenced using the `isRunningLock` instance variable.; This mutex ensures that only one process can read or write the value of `isRunning` (use of the `critical:` message in method 12.10 and method 12.11).

Method 12.12: *The EchoServer»stop Method*

```
EchoServer»stop
self isRunning: false
```

In order to manage the life-cycle of our server, we introduced two methods `EchoServer»start` and `EchoServer»stop`. Let's begin with the simplest one `EchoServer»stop` which definition is provided as method 12.12. It simply sets the `isRunning` flag to `false`. This will have the consequence of stopping the serving loop in method `EchoServer»serve` (see method 12.13).

Method 12.13: *The EchoServer»serve Method*

```
serve
| interactionSocket |
```

```
[self isRunning] whileTrue: [self interactOnConnection].
connectionSocket closeAndDestroy
```

The activity of the serving process is implemented in the `serve` method (see method 12.13). It interacts with clients on connections while the `isRunning` flag is true. After a stop, the connection socket is destroyed and the serving process is terminated. The creation of this serving process is the responsibility of method `EchoServer»start` (see the last line of method 12.14). eidosco

Method 12.14: The EchoServer»start Method

```
start
isRunningLock critical: [
    self isRunning ifTrue: [↑self].
    self isRunning: true].
connectionSocket := Socket newTCP.
connectionSocket listenOn: 9999 backlogSize: 10.
[self serve] fork
```

The `EchoServer»start` method first checks if the server is already running. It returns if this is case. Otherwise, the a TCP socket dedicated to connection handling is created and made to listen on port 9999. The backlog size refers to the size of the system buffer for storing pending client connection requests. This value is a trade-off that depends on how fast is the server (varies according to the VM and the hardware) and the maximum rate of client connections requests. Thus, the backlog size should be big enough to avoid loosing any connection request, but not too big to avoid wasting memory. Finally `EchoServer»start` method creates a process by sending the `fork` message to the `[self serve]` block. The created process has the same priority as the creator process (i.e. the one that performs the `EchoServer»start` method).

Method 12.15: The EchoServer»interactOnConnection Method

```
interactOnConnection
| interactionSocket |
interactionSocket := connectionSocket waitForAcceptFor: 1 ifTimedOut: [↑self].
[self interactUsing: interactionSocket] fork
```

Method `EchoServer»serve` (see method 12.13) loop interacts with clients on connections. This interaction is handled in the `EchoServer»interactOnConnection` method (see method 12.15) . First, the connection socket waits for client connections for one second. If no client attempts to connect during this period we simply return. Otherwise, we get as result another socket dedicated to interaction. To process other client connection requests, the interaction is performed in another process, hence the `fork` in the last line.

Method 12.16: The EchoServer»interactUsing: Method

```
interactUsing: interactionSocket
```

```
| receivedData |
[receivedData := interactionSocket receiveDataTimeout: 5.
Transcript cr; show: receivedData.
interactionSocket sendData: 'ECHO:', receivedData] ensure: [interactionSocket
closeAndDestroy]
```

The interaction as implemented in method `EchoServer»interactUsing:` (see method 12.16) with a client is the only application specific part⁵. The interaction boils down to reading data provided by the client and sending it back prefixed with the 'ECHO:' string. It worth noting that we ensure that the interaction socket is destroyed, whether we have exchanged data or not (timeout).

12.4 SocketStream

Client Side SocketStream

`SocketStream` is a read-write stream that encapsualtes a TCP socket. It eases the interaction with a remote server. We illustrate this facility with the code of the following code snippet (Script 12.17).

Script 12.17: *Getting the first line of a web page using SocketStream.*

```
|stream httpQuery result|
stream := SocketStream openConnectionToHostNamed: 'www.pharo-project.org' port:
80.
httpQuery := 'GET / HTTP/1.1', String crlf,
'Host: www.pharo-project.org:80', String crlf,
'Accept: text/html', String crlf.
[
  stream sendCommand: httpQuery.
  Transcript cr; show: stream nextLine.
] ensure: [
  stream close]
```

The first line creates a stream that encapsulates a newly created socket connected to the provided server. This is the responsibility of message `openConnectionToHostNamed:port:`. It suspends the execution until the connection with the server is estabilished. If the server does not respond, the socket stream signals a `ConnectionTimedOut` exception. This exception is actually signalled by the underlying socket. The timeout delay is 45 seconds (defined in method `Socket class»standardTimeout`).

⁵In an actual application we should have an abstract server superclass which subclasses implement the `interactUsing:` method.

Once our socket stream connected to the server, we forge and send an HTTP GET query. Notice that compared to script 12.5 (page 145) we skipped here (Script 12.17) one final String crlf. This is because the SocketStream »sendCommand: method automatically inserts CR and LF characters after sent data to mark line ending.

Reception of the requested web page is triggered by sending the nextLine message to our socket stream. It will wait a few seconds until data is received. Data is then displayed on the transcript. We safely ensure that the connection is closed.

In this example, we only display the first line of response sent by the server. We can easily display the full response including the html code by sending the upToEnd message to our socket stream. Note however that you'll have to wait a bit longer compared to displaying a single line.

12.5 A Basic Chat Application

12.6 Distributed Object Applications

12.7 Chapter summary

Chapter 13

The Settings Framework

As an application matures it often needs to provide variations such as a default selection color, a default font or a default font size. Often such variations represent user preferences for possible software customizations. Since the 1.1 release, Pharo has contained and used the Settings framework to manage its preferences. With Settings, an application can expose its configuration. Settings is not limited to managing Pharo preferences and we suggest using it for any application. What is nice about Settings is that it is not intrusive, it supports modular decomposition of software and it can be added to an application even after that application's inception. The Settings framework is what we will look at now.

13.1 Settings in a Nutshell

Setting supports an object-oriented approach to preferences definition and manipulation. What we want to express by this sentence is that:

1. each package or subsystem should define its own customization points (often represented as a variable or a class variable). The code of a subsystem then freely accesses such customization value and use it to change its behavior to reflect the preference.
2. Using Settings, a subsystem describes its preferences so that the end user can manipulate them. However, at no point in time, the code of a subsystem will explicitly refer to setting objects to adapt its behavior.

The control flow of a subsystem does not involve Settings. This is the major point of difference between Settings and the preference system available in Pharo1.0.

Vocabulary

A *preference* is a particular *value* which is usually accessible. Basically such a preference value is stored in a class variable or in an instance variable of a singleton and is directly managed through the use of simple accessors. Pharo contains numerous preferences such as the user interface theme, the desktop background color or a boolean flag to allow or prohibit the use of sound are currently declared as preferences. We will show how we can define a preference in Section 13.3.

A *setting* is a *declaration* (description) of a preference value. To be viewed and updated through the setting browser, a preference value must be described by a setting. Such a setting is built by a particular method tagged with a pragma (see Figure 13.1). Section 13.3 explains how to declare a setting.

Pharo users need to browse existing preferences and eventually change their value, this is the major role of the *Settings Browser* presented in Section 13.2.

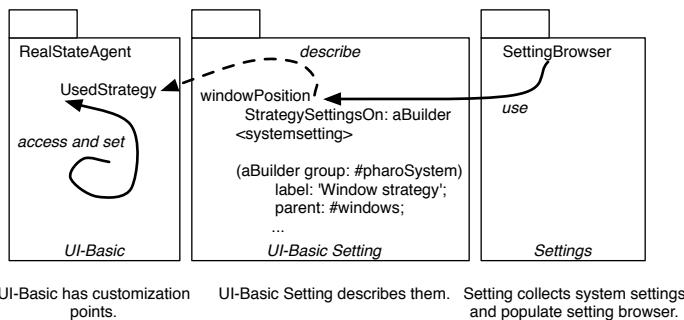


Figure 13.1: A package defines customization points. Such customization points are described with Settings instances. The *Settings Browser* collects the description and presents them to the user.

Figure 13.1 shows important points of the architecture put in place by Settings: The *Settings* package can be unloaded and a package defining preferences does not depend on the *Settings* package. This architecture is supported by the following points:

Customization points. Each application should define its customization points. For example, the class `RealStateAgent` of the package *UI-Basic* defines the class variable `UsedStrategy` which defines how the windows. The flow of the package *UI-Basic* is modular and self-contained: the class `RealStateAgent` does not depend on the settings framework. The class `RealStateAgent` has been designed to be parametrized.

Description of customization point. The Settings framework supports the description of the setting `UsedStrategy`. In Figure 13.1, the package `UI-Basic Setting` defines a method (it could be an extension to the class `RealStateAgent` or another class). The important point is that the method declaring the setting does not refer directly to `Setting` classes but describes the setting using a builder. This way this description could even be present in the `UI-Basic` package without introducing a reference.

Collecting setting for user presentation. The `Settings` package defines tools to manage settings such as a *Settings Browser* that the user uses to change his preference. The *Settings Browser* collects settings and uses their description to change the values of preferences. The control flow of the program and the dependencies are always from the package `Settings` to the package that has preferences and not the inverse.

13.2 The Settings Browser

The *Settings Browser*, shown in Figure 13.2, mainly allows one to browse all currently declared settings and to change related preference values. To open it, just use the World menu (`World > System > Settings`) or evaluate the following expression:

```
SettingBrowser open
```

The settings are presented in several trees in the middle panel. Setting searching and filtering is available from the top toolbar whereas the bottom panels show currently selected setting description (left bottom panel) and current package set (right bottom panel).

Browsing and changing preference values

Setting declarations are organized in trees which can be browsed in the middle panel. In order to get a description for a setting, just click on it: the setting is selected and the left bottom panel is updated with informations about the selected setting.

Changing a preference value is simply done through the browser: each line holds a widget on the right with which you can update the value. The kind of widget depends on the actual type of the preference value. Whereas a preference value can be of any kind, the setting browser is currently able to present a specific input widget for the following types: `Boolean`, `Color`, `FilePath`, `FileDirectory`, `Font`, `Number`, `Point` and `String`. A drop-list, a password field or a range input widget using a slider can also be used. Of course,

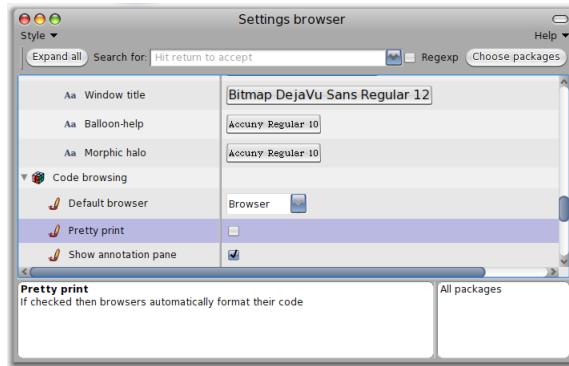


Figure 13.2: The *Settings Browser*.

the list of possible widgets is not closed as it is possible to make the setting browser support new kind of preference values or use different input widgets. This point is explained in Section 13.8.

If the actual type of a setting is either *String*, *FilePath*, *FileDirectory*, *Number* or *Point*, to change a value, the user has to enter some text in a editable drop-list widget. In such a case, the input must be confirmed by hitting the return key (or with cmd-s). If such a setting value is changed often, the drop-list widget is very handy because you can retrieve and use previously entered values in one click!

Other possible actions are all accessible from the contextual menu. Depending on the selected setting, they may be different. Three versions of it are shown in Figure 13.3.

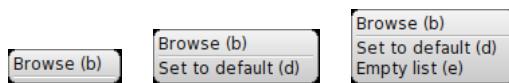


Figure 13.3: The contextual popup menu

- **Browse (b):** open a system browser on the method that declares the setting. It is also accessible via the keyboard shortcut *cmd-b* or if you double-click on a setting. It is very handy if you want to change the setting implementation or simply see how it is implemented to understand the framework by investigating some examples (how to declare a setting is explained in Section 13.3).
- **Set to default (d):** set the selected setting value to the default one. It is very handy if, as an example, you have played with a setting to observe

its effect and finally decide to come back to its default. It is also possible to set to default all settings in one single action, this is explained in Section 13.7.

- **Empty list (e):** If the input widget is an editable drop-list, this menu item allows one to forget previously entered values by emptying the recorded list.

Searching and filtering settings

Pharo contains a lot of settings and finding one of them can be tedious. You can filter the settings list by entering something in the search text field of the top bar of the *SettingsBrowser*. Then, only the settings which name or description contains the text you've entered will be shown. The text can be a regular expression if the "Regexp" checkbox is checked.

Another way to filter the list of settings is to choose them by package. Just click on the "Choose package" button, then a dialog is opened with the list of packages in which some settings are declared. If you choose one or several of them, then, only settings which are declared in the selected packages are shown. Notice that the bottom right text pane is updated with the name of the selected packages.

Depending on where and when you are using Pharo, you may have to change preferences repeatedly. As an example, when you are doing a demonstration, you may want to have bigger fonts, at work you may need to set a proxy whereas at home none is needed. Having to change a set of preferences depending on where you are and what you are doing can be very tedious and boring. With the *Settings Browser*, it is possible to save the current set of preference values in a named style that can be reloaded later. Setting style management is presented in Section 13.7.

13.3 Declaring a setting

All global preferences of Pharo can be viewed or changed using the *Settings Browser*. A preference is typically a class variable or an instance variable of a singleton. If one wants to be able to change its value from the *SettingsBrowser*, then a setting must be declared for it. A setting is declared by a particular *class* method that should be implemented as follows: it takes a builder as argument and it is tagged with the `<systemsettings>` pragma.

The argument, `aBuilder`, serves as an API or facade for building setting declarations. The pragma allows the *Settings Browser* to dynamically discover current setting declarations.

The important point is that a setting declaration should be package specific. It means that each package is responsible for the declaring of its own settings. For a particular package, specific settings are declared by one or several of its classes or a companion package. There is no global setting defining class or package (as it was the case in Pharo1.0). The direct benefit is that when the package is loaded, then its settings are automatically loaded and that when a package is unloaded, then its settings are automatically unloaded. In addition a Setting declaration should not refer to any Setting class but to the builder argument. This makes sure that your application is not dependent from Settings and that you will be able to remove Setting if you want to define extremely small footprint applications.

Let's take the example of the `caseSensitiveFinds` preference. It is a boolean preference which is used for text searching. If it is true, then text finding is case sensitive. This preference is stored in the `CaseSensitiveFinds` class variable of the class `TextEditor`. Its value can be queried and changed by, respectively, `TextEditor class>>caseSensitiveFinds` and `TextEditor class>>caseSensitiveFinds:` given below:

```
TextEditor class>>caseSensitiveFinds
↑ CaseSensitiveFinds ifNil: [CaseSensitiveFinds := false]

TextEditor class>>caseSensitiveFinds: aBoolean
CaseSensitiveFinds := aBoolean
```

To define a setting for this preference (*i.e.*, for the `CaseSensitiveFinds` class variable) and be able to see it and change it from the *Settings Browser*, the method below is implemented. The result is shown in the screenshot of the Figure 13.4.

```
CodeHolderSystemSettings class>>caseSensitiveFindsSettingsOn: aBuilder
<systemsettings>
(aBuilder setting: #caseSensitiveFinds)
target: TextEditor;
label: 'Case sensitive search' translated;
description: 'If true, then the "find" command in text will always make its searches in
a case-sensitive fashion' translated;
parent: #codeEditing.
```

Now, let's study this setting declaration in details.

The header

```
CodeHolderSystemSettings class>>caseSensitiveFindsSettingsOn: aBuilder
...
```

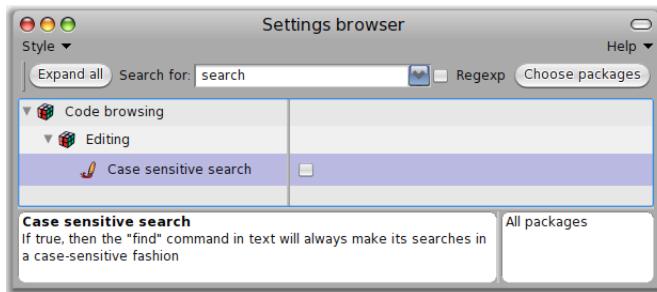


Figure 13.4: The *caseSensitiveFinds* setting

This class method is declared in the class `CodeHolderSystemSettings`. This class is dedicated to settings and contains nothing but settings declarations. Defining such a class is not mandatory; in fact any class can define settings declarations. We define it that way to make sure that the setting declaration is packaged in a different package than the one of the preference definition – for layering purposes.

This method takes a builder as argument. This object serves as an API or facade for setting buildings: the contents of the method essentially consists in sending messages to the builder to declare and organize a sub-tree of settings.

The pragma

A setting declaration is tagged with the `<systemsettings>` pragma.

```
CodeHolderSystemSettings class>>caseSensitiveFindsSettingsOn: aBuilder
<systemsettings>
...
...
```

In fact, when the settings browser is opened, it first collects all settings declarations by searching all methods with the `<systemsettings>` pragma. In addition, if you compile a setting declaration method while a *Settings Browser* is opened then it is automatically updated with the new setting.

The setting configuration

A setting is declared by sending the message `setting:` to the builder with an identifier passed as argument. Here the identifier is `#caseSensitiveFinds`. Here is an example:

```
CodeHolderSystemSettings class>>caseSensitiveFindsSettingsOn: aBuilder
```

```
<systemsettings>
(aBuilder setting: #caseSensitiveFinds)
...
```

Sending the message `setting:` to a builder creates a *setting node*. By default, the symbol passed as argument is considered as the selector used by the *Settings Browser* to get the preference value. The selector for changing the preference value is by default built by adding a colon to the getter selector (*i.e.*, it is `caseSensitiveFinds:` here). These selectors are sent to a target which is by default the class in which the method is implemented (*i.e.*, `CodeHolderSystemSettings`). Thus, this one line setting declaration would be sufficient if `caseSensitiveFinds` and `caseSensitiveFinds:` accessors were implemented in `CodeHolderSystemSettings`.

In fact, very often, these default initializations will not fit your need. Of course you can adapt the setting node configuration to take into account your specific situation.

For example, the corresponding getter and setter accessors for the `caseSensitiveFinds` setting are implemented in the class `TextEditor`. Then, we should explicitly set that the target is `TextEditor`. This is done by sending the message `target:` to the setting node with the target class `TextEditor` passed as argument as shown by the updated definition:

```
CodeHolderSystemSettings class>>caseSensitiveFindsSettingsOn: aBuilder
<systemsettings>
(aBuilder setting: #caseSensitiveFinds)
    target: TextEditor
```

This very short version is fully working and enough to be compiled and taken into account by the *Settings Browser* as shown by Figure 13.5.

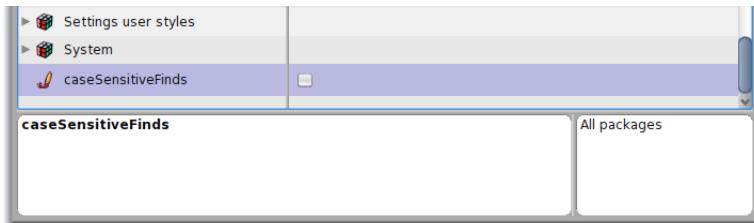


Figure 13.5: A first simple version of the `caseSensitiveFinds` setting.

Unfortunately, the presentation is not really user-friendly because:

- the label shown in the settings browser is the identifier (the symbol used to build accessors to access it),

- there is no description or explanation available for this setting, and
- the new setting is simply added at the root of the setting tree.

To address such shortcomings, you can configure more your setting node with a label and a description with respectively the `label:` and `description:` messages which take a string as argument.

```
CodeHolderSystemSettings class>>>caseSensitiveFindsSettingsOn: aBuilder
<systemsettings>
(aBuilder setting: #caseSensitiveFinds)
target: TextEditor;
label: 'Case sensitive search' translated;
description: 'If true, then the "find" command in text will always make its searches
in a case-sensitive fashion' translated;
parent: #codeEditing.
```

Don't forget to send `translated` to the label and the description strings, it will greatly facilitate the translation in other languages.

Concerning the classification and the settings tree organization, there are several ways to improve it and this point is fully detailed in the next section.

More about the target

The target of a setting is the receiver for getting and changing the preference value. Most of the time, it is a class. Indeed, typically, a preference value is stored in a class variable. Thus, class side methods are used as accessors for accessing the setting.

But the receiver can also be a singleton object. This is currently the case for many preferences. As an example, the Free Type fonts preferences, they are all stored in the instance variables of a `FreeTypeSettings` singleton. Thus, here, the receiver is the `FreeTypeSettings` instance that you can get by evaluating the following expression:

```
FreeTypeSettings current
```

So, one can use this expression to configure the target of a corresponding setting. As an example the `#glyphContrast` preference could be declared as follow:

```
(aBuilder setting: #glyphContrast)
target: FreeTypeSettings current;
label: 'Glyph contrast' translated;
...
```

This is simple but unfortunately, declaring such a singleton target like this is not a good idea. This declaration is not compatible with the *Setting style*

functionalities (see Section 13.7). In such a case, one have to separately indicate the target class and the message selector to send to the target class to get the singleton. Thus, as shown in the example below, you should use the `targetSelector:` message:

```
(aBuilder setting: #glyphContrast)
  target: FreeTypeSettings;
targetSelector: #current;
  label: 'Glyph contrast' translated;
  ...
...
```

More about default values

The way the *Settings Browser* build a setting input widget depends on the actual value type of a preference. Having `nil` as a value for a preference is a problem for the *Settings Browser* because it can't figure out which input widget to use. So, basically, in order to be properly shown with the good input widget, a preference must always be set with a non `nil` value. You can set a default value to a preference by initializing it as usual, with a `#initialize` method or with a lazy initialization programed in the accessor method of the preference.

Regarding the *Settings Browser*, the best way is the lazy initialization (see the example of the `#caseSensitiveFinds` preference given in the section 13.3). Indeed, as explained in the section 13.2, from the *Settings Browser* contextual menu, you can reset a preference value to its default one or globally reset all preference values. In fact it is done by setting the preference value to reset to `nil`. As a consequence, the preference is automatically set to its default value as soon as it is get by using its dedicated accessor.

It is not always possible to change the way an accessor is implemented. A reason for that could be that the preference accessor is maintained within another package which you are'nt allowed to change. As shown in the example below, as a workaround, you can indicate a default value from the declaration of the setting:

```
CodeHolderSystemSettings class>>caseSensitiveFindsSettingsOn: aBuilder
<systemsettings>
(aBuilder setting: #caseSensitiveFinds)
default: true;
...
```

13.4 Organizing your settings

Within the *Settings Browser*, settings are organized in trees where related settings are shown as children of the same parent.

Declaring a parent

The simplest way to declare your setting as a child of another setting is to use the parent: message with the identifier of the parent setting passed as argument. In the example below, the parent node is an existing node declared with the #codeEditing identifier.

```
CodeHolderSystemSettings class>>caseSensitiveFindsSettingsOn: aBuilder
  <systemsettings>
  (aBuilder setting: #caseSensitiveFinds)
    target: TextEditor;
    label: 'Case sensitive search' translated;
    description: 'If true, then the "find" command in text will always make its searches in
    a case-sensitive fashion' translated;
    parent: #codeEditing.
```

The #codeEditing node is also declared somewhere in the system. For example, it could be defined as a group as we will see now.

Declaring a group

A group is a simple node without any value and which is only used for children grouping. The #codeEditing node is created by sending the group: message to the builder with its identifier passed as argument. Notice also that, as shown in Figure 13.4, the #codeEditing node is not at root because it is declared itself as a child of the #codeBrowsing node.

```
CodeHolderSystemSettings class>>codeEditingSettingsOn: aBuilder
  <systemsettings>
  (aBuilder group: #codeEditing)
    label: 'Editing' translated;
    parent: #codeBrowsing.
```

Declaring a sub-tree

Being able to declare its own settings as a child of a pre-existing node is very useful when a package wants to enrich existing standard settings. But it can also be very tedious for settings which are very application specific.

Thus, directly declaring a sub-tree of settings in one method is also possible. Typically, a root group is declared for the application settings and the children settings themselves are also declared within the same method. This is simply done through the sending of the `with:` message to the root group. The `with:` message takes a block as argument. In this block, every new settings are implicitly declared as children of the root group (the receiver of the `with:` message).



Figure 13.6: Declaring a subtree in one method: the *Configurable formatter* setting example.

As an example, take a look at Figure 13.6, it shows the settings for the refactoring browser configurable formatter. This sub-tree of settings is fully declared in the method `RBCConfigurableFormatter class>>settingsOn:` given below. You can see that it declares the new root group `#configurableFormatter` with two children, `#formatCommentWithStatements` and `#indentString`:

```
RBCConfigurableFormatter class>>settingsOn: aBuilder
<systemsettings>
(aBuilder group: #configurableFormatter)
target: self;
parent: #refactoring;
label: 'Configurable Formatter' translated;
description: 'Settings related to the formatter' translated;
with: [
(aBuilder setting: #formatCommentWithStatements)
label: 'Format comment with statements' translated.
(aBuilder setting: #indentString)
label: 'Indent string' translated]
```

Optional sub-tree

Depending on the value of a particular preference, one might want to hide some settings because it doesn't make sense to show them. As an example, if the background color of the desktop is plain then it doesn't make sense to show settings which are related to gradient background. Instead, when the user wants a gradient background, then a second color, the gradient direction, and the gradient origin settings should be presented. Look at the Figure 13.7:

- on the left, the *Gradient* widget is unchecked meaning that its actual value is false; in this case, it has no children,
- on the right, the *Gradient* widget is checked, then the setting value is set to true and as a consequence, the settings useful in order to set a gradient background are shown.



Figure 13.7: Example of optional subtree. Right – no gradient is selected. Left – gradient is selected so additional preferences are available.

To handle such optional settings is simple: optional settings should be declared as children of a boolean parent setting. In this case, children settings are shown only if the parent value is true. Concerning the desktop gradient example, the setting is declared in `PolymorphSystemSettings` as given below:

```
(aBuilder setting: #useDesktopGradientFill)
label: 'Gradient';
description: 'If true, then more settings will be available in order to define the
desktop background color gradient';
with: [
  (aBuilder setting: #desktopGradientFillColor)
    label: 'Other color';
    description: 'This is the second color of your gradient (the first one is given by
the "Color" setting' translated.
  (aBuilder pickOne: #desktopGradientDirection)
    label: 'Direction';
    domainValues: {#Horizontal. #Vertical. #Radial}.
  (aBuilder pickOne: #desktopGradientOrigin)
    label: 'Origin';
    domainValues: {
      'Top left' translated -> #topLeft. ...
    }
]
```

The parent setting value is given by evaluating `PolymorphSystemSettings` class->`useDesktopGradientFill`. If it returns true, then the children `#desktopGradientFillColor`, `#desktopGradientDirection`, and `#desktopGradientOrigin` are shown.

Ordering your settings

By default, sibling settings are sorted alphabetically by their label. You may want to change this default behavior. Changing the settings ordering can be done two ways: by simply forbidding the default ordering or by explicitly specifying an order.

As in the following example of the `#appearance` group, you can indicate that no ordering should be performed by sending the `#noOrdering` message to the parent node. Then its children are let in declaration order.

```
appearanceSettingsOn: aBuilder
<systemsettings>
(aBuilder group: #appearance)
  label: 'Appearance' translated;
  description: 'All settings concerned with the look'n feel of your system' translated;
noOrdering;
  with: [...]
```

You can indicate the order of a setting node among its siblings by sending the message `order:` to it with a number passed as argument. The number can be an Integer or a Float. Nodes with an order number are always placed before others and are sorted according to their respective order number. If an order is given to an item, then no ordering is applied for other siblings.

As an example, take a look at how the `#standardFonts` group is declared:

```
(aBuilder group: #standardFonts)
  label: 'Standard fonts' translated;
  target: StandardFonts;
  parent: #appearance;
  with: [
    (aBuilder launcher: #updateFromSystem)
      order: 1;
      targetSelector: #current;
      script: #updateFromSystem;
      label: 'Update fonts from system' translated.
    (aBuilder setting: #defaultFont)
      label: 'Default' translated.
    (aBuilder setting: #codeFont)
      label: 'Code' translated.
    (aBuilder setting: #listFont)
    ...
  ]
```

In this example, the launcher `#updateFromSystem` is declared to be the first node, then other siblings with identifiers `#defaultFont`, `#codeFont`, and `#listFont` are placed according to the declaration order.

13.5 Providing more precise value domain

By default, the possible value set of a preference is not restricted and is given by the actual type of the preference. For example, for a color preference, the widget allows you to choose whatever color, for a number, the widget allows the user to enter any number. But, in some cases, only a particular set of values is desired. As an example, for the standard browser or for the user interface theme settings, the choice must be made among a finite set of classes, for the free type cache size, only a range from 0 to 50000 is allowed. In these cases, it is much more comfortable if the widget can only accept particular values. To address this issue, the domain value set can be constrained either with a range or with a list of values.

Declaring a range setting

As an example, let's consider the full screen margin preference shown in the Figure 13.8. Its value represents the margin size in pixels that is let around a window when it is expanded.



Figure 13.8: Example of range setting.

It's value is an integer but it makes no sense to set -100 or 5000 to it. Instead, a minimum of -5 and a maximum of 100 constitute a good range of values. One can use this range to constraint the setting widget. As shown by the example below, comparing to a simple setting, the only two differences are that:

- the new setting node is created with the range: message instead of the setting: message and
- the valid range is given by sending the range: message to the setting node, an Interval is given as argument.

```
screenMarginSettingOn: aBuilder
<systemsettings>
(aBuilder range: #fullScreenMargin)
    target: SystemWindow;
    parent: #windows;
    label: 'Full screen margin' translated;
```

description: 'Specify the amount of space that is let around a windows when it's opened fullscreen' translated;
range: (-5 to: 100).

Selecting among a list

When a preference value is constrained to be one of a particular list of values, it is possible to declare it so that a drop list is used by the settings browser. This drop list is initialized with the predefined valid values. As an example, consider the *window position strategy* example. The corresponding widget is shown in action within the settings browser by Figure 13.9. The allowed values are 'Reverse Stagger', 'Cascade', or 'Standard'.



Figure 13.9: Example of a list setting.

The example below shows a simplified declaration for the *window position strategy* setting.

```
windowPositionStrategySettingsOn: aBuilder
  <systemsettings>
    (aBuilder pickOne: #usedStrategy)
      label: 'Window position strategy' translated;
      target: RealEstateAgent;
      domainValues: #(#'Reverse Stagger' #Cascade #Standard)
```

comparing to a simple setting, the only two differences are that:

- the new setting node is created with the `pickOne:` message instead of the `#setting:` message and
- the list of authorized values is given by sending the `domainValues:` message to the newly declared setting node, a Collection is given as argument (the default value being the first one).

Concerning this window strategy example, the value set to the preference would be either #'Reverse Stagger' or #Cascade or #Standard.

Unfortunately, these values are not very handy. A programmer may wish another value as, for example, some kind of *strategy object* or a *Symbol* which

could directly serve as a selector. In fact, this second solution has been chosen by the `RealEstateAgent` class maintainers. If you inspect the value returned by `RealEstateAgent usedStrategy` you will realize that the result is not a `Symbol` among `#Reverse Stagger`, `#Cascade`, or `#Standard` but another symbol. Then, if you look at the way the window position strategy setting is really implemented you will see that the declaration differs from the basic solution given previously: the `domainValues:` argument is not a simple array of `Symbols` but an array of `Associations` as you can see in the declaration below:

```
windowPositionStrategySettingsOn: aBuilder
<systemsettings>
(aBuilder pickOne: #usedStrategy)
...
domainValues: {'Reverse Stagger' translated -> #staggerFor:initialExtent:world: .
'Cascade' translated -> #cascadeFor:initialExtent:world: . 'Standard' translated ->
#standardFor:initialExtent:world:};
```

From the *Settings Browser* point of view, the content of the list is exactly the same and the user can't notice any difference because, if an array of `Associations` is given as argument to `domainValues:`, then the keys of the `Associations` are used for the user interface.

Concerning the value of the preference itself, if you inspect `RealEstateAgent usedStrategy`, you should notice that the result is a value among `(#staggerFor:initialExtent:world: #cascadeFor:initialExtent:world: #standardFor:initialExtent:world:)`. In fact, the values of the `Associations` are used to compute all possible real values for the setting.

The list of possible values can be of any kind. As another example, let's take a look at the way the user interface theme setting is declared in the `PolymorphSystemSettings` class:

```
(aBuilder pickOne: #uiThemeClass)
label: 'User interface theme' translated;
target: self;
domainValues: (UITheme allThemeClasses collect: [:c | c themeName -> c])
```

In this example, `domainValues:` takes an array of associations which is computed each time a *Settings Browser* is opened. Each association is made of the name of the theme as key and of the class which implements the theme as value.

13.6 Launching a script

Imagine that you want to launch an external configuration tool or that you want to allow one to configure the system or a particular package with the

help of a script. In such a case you can declare a *launcher*. A launcher is shown with a label as a regular setting except that no value is to be entered for it. Instead, a button labelled *Launch* is integrated in the *Settings Browser* and clicking on it launch an associated script.

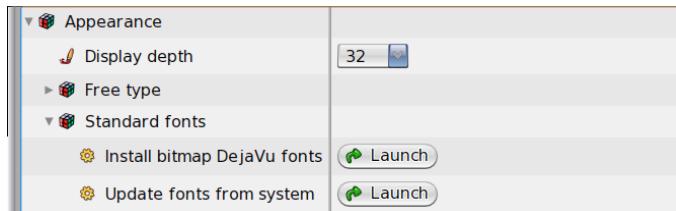


Figure 13.10: Example of launchers.

As an example, to use True Type Fonts, the system must be updated by collecting all the available fonts in the host system. This can be done by evaluating the following expression:

```
FreeTypeFontProvider current updateFromSystem
```

It is possible to run this script from the *Settings Browser*. The corresponding launcher is shown in Figure 13.10. The integration of such a launcher is quite simple. You simply have to declare a setting for it! For example, look at how the launcher for the TT fonts is declared:

```
GraphicFontSettings class>> standardFontsSettingsOn:
<systemsettings>
(aBuilder group: #standardFonts)
...
(aBuilder launcher: #updateFromSystem) ...
target: FreeTypeFontProvider;
targetSelector: #current;
script: #updateFromSystem;
label: 'Update fonts from system' translated.
```

Comparing to a simple setting, the only two differences are that:

- the new setting node is created by sending the `launcher:` message to the builder and
- the message `script:` is sent to the setting node with the selector of the script passed as argument.

13.7 Setting styles management

Even if many preferences have been removed from Pharo because they were obsolete, there are still a large number of them. And even if the *Settings Browser* is easy to use, it may be tedious to set up your own preferences even for a subset, each time you start working with a new image. A solution is to implement a script to set all your preferred choices. The best way is to create a specific class for that purpose. Then you can include it in a package that you can reload each time you want to setup a fresh image. We call this kind of class a *Setting style*.

To manage *Setting styles*, the *Settings Browser* can be helpful in two ways. First, it can help you discover how to change a preference value, and second, it can create and update a particular style for you.

Scripting settings

Because preference variables are all accessible with accessor methods, it is naturally possible to initialize a set of preferences in a simple script. For the sake of simplicity, let's implement it in a *Setting style*.

As an example a script can be implemented to change the background color and to set all fonts to a bigger one than the default. Let's create a *Setting style* class for that. We can call it *MyPreferredStyle*. The script is defined by a method of *MyPreferredStyle*. We call this method *loadStyle* because this selector is the standard hook for settings related script evaluating.

```
MyPreferredStyle->>loadStyle
| f n |
"Desktop color"
PolymorphSystemSettings desktopColor: Color white.
"Bigger font"
n := StandardFonts defaultFont. "get the current default font"
f := LogicalFontfamilyName: n familyName pointSize: 12. "font for my preferred size"
StandardFonts setAllStandardFontsTo: f "reset all fonts"
```

PolymorphSystemSettings is the class in which all settings related to *PolyMorph* are declared. *StandardFonts* is the class that is used to manage Pharo default fonts.

Now the question is how to find out that the desktop color setting is declared in *PolymorphSystemSettings* and that the *DefaultFonts* class allows fonts management? More generally where are all these settings declared and managed?

The answer is quite simple: just use the *Settings Browser*! As explained in Section 13.2, *cmd-b* or double clicking on an item open a browser on the

declaration of the current setting node. You can also use the contextual menu for that. Browsing the declaration will give you the target class (where the preference variable is stored) and the selector for the preference value.

Now we would like `MyPreferredStyle>>#loadStyle` to be automatically evaluated when `MyPreferredStyle` is itself loaded in the system. For that purpose, the only thing to do is to implement an initialize method for the `MyPreferredStyle` class:

```
MyPreferredStyle class>>initialize
    self new loadStyle
```

Integrating a style in the *Settings Browser*

Any script can be integrated in the *Settings Browser* so that it could be loaded, browsed or even removed from it. For that purpose you only have to declare a name for it and to make sure that the *Settings Browser* will discover it. Just implement a method named `styleName` on the class side of your style class. Concerning the example of previous section, it should be implemented as follows:

```
MyPreferredStyle class>>styleName
    "The style name used by the SettingBrowser"
    <settingstyle>
    ↑ 'My preferred style'
```

`MyPreferredStyle class>>styleName` takes no argument and must return the name of your style as a *String*. The `<settingstyle>` pragma is used to let the *Settings Browser* know that `MyPreferredStyle` is a setting style class.

Once this method is compiled, open the Setting Browser and popup the *Style* top menu. As shown by Figure 13.11, you should see a dialog with a list of style names comprising your own one.

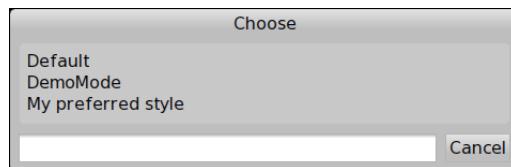


Figure 13.11: The style load dialog with your own style

13.8 Extending the *Settings Browser*

As explained in the section 13.2, the *Settings Browser* is by default able to manage a simple preference types. These default possibilities are generally enough. But there are some situations where it can be very helpful to be able to handle more complex preference values.

As an example, let focus on the text selection preferences. We have the primary selection and three other optional kinds of text selection, the secondary selection, the find and replace selection and the selection bar. For all selections, a background color can be set. For the primary, the secondary and the find and replace selection, a text color can also be chosen.

Declaring selection settings individually

So far, according to the default possibilities, a setting can be declared for each of the text selection characteristics so that each corresponding preference can be changed individually from the *Settings Browser*. Settings declared for a particular selection kind can be grouped together as children of a setting group. As an immediate improvement, for an optional text selection, a boolean setting can be used instead of a simple group.

As an example, let's take the secondary selection. This text selection kind is optional and one can set a background and a text color for it. Corresponding preferences are declared as instance variables of `ThemeSettings`. Their values can be read and changed from the current theme by getting its associated `ThemeSettings` instance. Thus, the two color settings can be declared as children of the `#useSecondarySelection` boolean setting as given below:

```
(aBuilder setting: #useSecondarySelection)
    target: UITheme;
    targetSelector: #currentSettings;
    label: 'Use the secondary selection' translated;
    with: [
        (aBuilder setting: #secondarySelectionColor)
            label: 'Secondary selection color' translated.
        (aBuilder setting: #secondarySelectionTextColor)
            label: 'Secondary selection text color' translated].
```

The Figure 13.12 shows these setting declarations in the *Settings Browser*. The look and feel is clean but in fact two observations can be made:

1. it takes three lines for each selection kind, this is a little bit uncomfortable because the view for one selection takes a lot of vertical space,
2. the underlying model is not explicitly designed, the settings for one selection kind are grouped together in the *Settings Browser* but corre-

sponding preference values are declared as separated instances variables of `ThemeSettings`. In the next section we see how to improve this first solution with a better design.

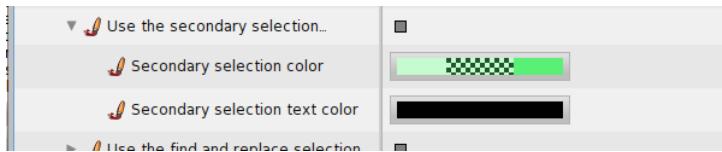


Figure 13.12: The secondary selection settings declared with basic setting values

An improved selection preference design

A better solution would be to design the concept of text selection preference. Then, we have only one value to manage for each selection preference instead of three. A text selection preference is basically made of two colors, one for the background and the second for the text. Except the primary selection, each selection is optional. Then, we could design a text selection preference as follow:

```
Object subclass: #TextSelectionPreference
instanceVariableNames: 'backgroundColor textColor mandatory used'
classVariableNames: 'FindReplaceSelection PrimarySelection SecondarySelection
SelectionBar'
poolDictionaries: ""
category: 'Settings-Tools'
```

`TextSelectionPreference` is made of four instance variables. Two of them are for the colors. If the mandatory instance variable is set to false then, the used boolean instance variable can be changed. Instead, if the mandatory is set to true, then, the used instance variable is set to true and is not changeable.

`TextSelectionPreference` have also four class variables, one for each kind of possible text selection preference. The getters and the setters have also to be implemented in order to be able to manage these preferences from the *Settings Browser*. As an example, for `PrimarySelection`:

```
TextSelectionPreference class>>primarySelection
↑ PrimarySelection
ifNil: [PrimarySelection := self new
    textColor: Color black;
    backgroundColor: (Color blue alpha: 0.5);
mandatory: true;
    yourself]
```

You can notice that the mandatory attribute is initialize to true.

Another example with the selection bar preference:

```
TextSelectionPreference class>>selectionBar
↑ SelectionBar
ifNil: [SelectionBar := self new
    backgroundColor: Color lightBlue veryMuchLighter;
mandatory: false;
yourself]
```

Here, you can notice that the preference is declared as optional and with no text color.

In order for these preferences to be changeable from the *Settings Browser*, we have to declare two methods. The first one is for the setting declaration and the second is to implement the view.

The setting declaration is implemented as follow:

```
TextSelectionPreference class>>selectionPreferenceOn: aBuilder
<systemsettings>
(aBuilder group: #selectionColors)
label: 'Text selection colors' translated;
parent: #appearance;
target: self;
with: [(aBuilder setting: #primarySelection) order: 1;
label: 'Primary'.
(aBuilder setting: #secondarySelection)
label: 'Secondary'.
(aBuilder setting: #findReplaceSelection)
label: 'Find/replace'.
(aBuilder setting: #selectionBar)
label: 'Selection bar']
```

As you can see, there is absolutely nothing new in this declaration. The only thing that changes is that the value of the preferences are of a user defined class. In fact, in case of user defined or application specific preference class, the only particular thing to do is to implement one supplementary method for the view. This method must be named `#settingInputWidgetForNode`: and must be implemented as a class method.

`#settingInputWidgetForNode`: method responsibility is to build the input widget for the *Settings Browser*. This method takes a `SettingDeclaration` as argument. `SettingDeclaration` is basically a model and its instances are managed by the *Settings Browser*.

Each `SettingDeclaration` instance serves as a preference value holder. Indeed, each setting that you can view in the *Settings Browser* is internally represented by a `SettingDeclaration` instance.

For each of our text selection preferences, we want to be able to change its colors and if the selection is optional, then we want to have the possibility to enable or disable it. Regarding the colors, depending on the selection preference value, only the background color is always shown. Indeed, if the text color of the preference value is nil, this means that having a text color does'nt makes sense and then, the corresponding color chooser is not built.

The #settingInputWidgetForNode: method can be implemeented as below:

```
TextSelectionPreference class>>settingInputWidgetForNode: aSettingDeclaration
| preferenceValue backColorUI usedUI uiElements |
preferenceValue := aSettingDeclaration preferenceValue.
usedUI := self usedCheckboxForPreference: preferenceValue.
backColorUI := self backgroundColorChooserForPreference: preferenceValue.
uiElements := {usedUI. backColorUI},
(primitiveValue textColor
ifNotNil: [ { self textColorChooserForPreference: primitiveValue } ]
ifNil: [{}]).
↑ (self theme newRowIn: self world for: uiElements)
cellInset: 20;
yourself
```

This method simply adds some basic elements in a row and returns the row. At a first place, you can notice that the actual preference value, an instance of TextSelectionPreference, is got from the SettingDeclaration instance by sending #primitiveValue to it. Then, the user interface elements can be built based on the actual TextSelectionPreference instance.

The first element is a *checkbox* or an empty space returned by the #usedCheckboxForPreference: invocation. This method is implemented as follow:

```
TextSelectionPreference class>>usedCheckboxForPreference: aSelectionPreference
↑ aSelectionPreference optional
ifTrue: [self theme
newCheckboxIn: self world
for: aSelectionPreference
getSelected: #used
setSelected: #used:
getEnabled: #optional
label: "
help: 'Enable or disable the selection']  

ifFalse: [Morph new height: 1;
width: 30;
color: Color transparent]
```

The next elements are two color choosers. As an example, the background color chooser is built as follow:

```
TextSelectionPreference class>>backgroundColorChooserForPreference:
    aSelectionPreference
    ↑ self theme
        newColorChooserIn: self world
        for: aSelectionPreference
        getColor: #backgroundColor
        setColor: #backgroundColor:
        getEnabled: #used
        help: 'Background color' translated
```

Now, in the *Settings Browser*, the user interface looks as shown in Figure 13.13, with only one line for each selection kind instead of three as in our previous version.



Figure 13.13: The text selection settings implemented with a specific preference class

13.9 Conclusion

We presented Settings, a new framework to manage preferences in a modular way. The key point of Settings is that it supports a modular flow of control: a package is responsible to define customization points and can use them locally, then using Settings it is possible to describe such customization points. Finally the Settings Browser collects such setting descriptions and present them to the user. The flow is then from the *Settings Browser* to the customized packages.

Chapter 14

Regular Expressions in Pharo

with the participation of:

Stéphane Ducasse (stephane.ducasse@inria.fr)

Oscar Nierstrasz (oscar.nierstrasz@acm.org)

Regular expressions are widely used in many scripting languages such as Perl, Python and Ruby. They are useful to identify strings that match a certain pattern, to check that input conforms to an expected format, and to rewrite strings to new formats. Pharo also supports regular expressions due to the *Regex* package contributed by Vassili Bykov. Regex is installed by default in Pharo. If you are using an older image that does not include Regex the Regex package, you can install it yourself from SqueakSource¹.

A regular expression² is a template that matches a set of strings. For example, the regular expression 'h.*o' will match the strings 'ho', 'hiho' and 'hello', but it will not match 'hi' or 'yo'. We can see this in Pharo as follows:

```
'ho' matchesRegex: 'h.*o'      →  true
'hiho' matchesRegex: 'h.*o'     →  true
'hello' matchesRegex: 'h.*o'    →  true
'hi' matchesRegex: 'h.*o'       →  false
'yo' matchesRegex: 'h.*o'       →  false
```

In this chapter we will start with a small tutorial example in which we will develop a couple of classes to generate a very simple site map for a web site. We will use regular expressions (i) to identify HTML files, (ii) to strip

¹<http://www.squeaksource.com/Regex.html>

²http://en.wikipedia.org/wiki/Regular_expression

the full path name of a file down to just the file name, (iii) to extract the title of each web page for the site map, and (iv) to generate a relative path from the root directory of the web site to the HTML files it contains. After we complete the tutorial example, we will provide a more complete description of the Regex package, based largely on Vassili Bykov's documentation provided in the package.³

14.1 Tutorial example—generating a site map

Our job is to write a simple application that will generate a site map for a web site that we have stored locally on our hard drive. The site map will contain links to each of the HTML files in the web site, using the title of the document as the text of the link. Furthermore, links will be indented to reflect the directory structure of the web site.

Accessing the web directory

 If you do not have a web site on your machine, copy a few HTML files to a local directory to serve as a test bed.

We will develop two classes, `WebDir` and `WebPage`, to represent directories and web pages. The idea is to create an instance of `WebDir` which will point to the root directory containing our web site. When we send it the message `makeToc`, it will walk through the files and directories inside it to build up the site map. It will then create a new file, called `toc.html`, containing links to all the pages in the web site.

One thing we will have to watch out for: each `WebDir` and `WebPage` must remember the path to the root of the web site, so it can properly generate links relative to the root.

 Define the class `WebDir` with instance variables `webDir` and `homePath`, and define the appropriate initialization method. Also define class-side methods to prompt the user for the location of the web site on your computer, as follows:

```
WebDir>>setDir: dir home: path
    webDir := dir.
    homePath := path

WebDir class>>onDir: dir
    ↑ self new setDir: dir home: dir pathName

WebDir class>>selectHome
```

³The original documentation can be found on the class side of `RxParser`.

↑ self onDir: fileList modalFolderSelector

The last method opens a browser to select the directory to open. Now, if you inspect the result of WebDir selectHome, you will be prompted for the directory containing your web pages, and you will be able to verify that webDir and homePath are properly initialized to the directory holding your web site and the full path name of this directory.

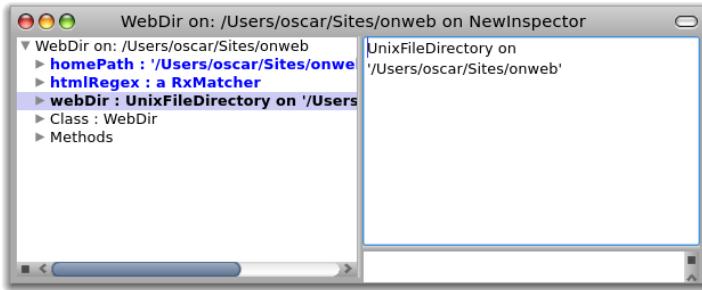


Figure 14.1: A WebDir instance

It would be nice to be able to programmatically instantiate a WebDir, so let's add another creation method.

Add the following methods and try it out by inspecting the result of WebDir onPath: 'path to your web site'.

WebDir class>>onPath: homePath
↑ self onPath: homePath home: homePath

WebDir class>>onPath: path home: homePath
↑ self new setDir: (FileDirectory on: path) home: homePath

Pattern matching HTML files

So far so good. Now we would like to use regexes to find out which HTML files this web site contains.

If we browse the FileDirectory class, we find that the method fileNames will list all the files in a directory. We want to select just those with the file extension .html. The regex that we need is '.*\.\html'. The first dot will match any character except a newline:

```
'x' matchesRegex: '.' → true
'' matchesRegex: '' → true
Character cr asString matchesRegex: '' → false
```

The `*` (known as the “Kleene star”, after Stephen Kleene, who invented it) is a regex operator that will match the preceding regex any number of times (including zero).

```
" matchesRegex: 'x*'      →  true
'x' matchesRegex: 'x*'    →  true
'xx' matchesRegex: 'x*'   →  true
'y' matchesRegex: 'x*'   →  false
```

Since the dot is a special character in regexes, if we want to literally match a dot, then we must escape it.

```
'. matchesRegex: '.'     →  true
'x' matchesRegex: '.'    →  true
'.' matchesRegex: '.'    →  true
'x' matchesRegex: '\.'  →  false
```

Now let's check our regex to find HTML files works as expected.

```
'index.html' matchesRegex: '.*\..html'  →  true
'foo.html' matchesRegex: '.*\.\.html'    →  true
'style.css' matchesRegex: '.*\.\.html'   →  false
'index.htm' matchesRegex: '.*\.\.html'   →  false
```

Looks good. Now let's try it out in our application.

 Add the following method to WebDir and try it out on your test web site.

```
WebDir>>htmlFiles
↑ webDir fileNames select: [ :each | each matchesRegex: '.*\.\.html' ]
```

If you send `htmlFiles` to a `WebDir` instance and `print it`, you should see something like this:

```
(WebDir onPath: '...') htmlFiles  →  #('index.html' ...)
```

Caching the regex

Now, if you browse `matchesRegex:`, you will discover that it is an extension method of `String` that creates a fresh instance of `RxParser` every time it is sent. That is fine for ad hoc queries, but if we are applying the same regex to every file in a web site, it is smarter to create just one instance of `RxParser` and reuse it. Let's do that.

 Add a new instance variable `htmlRegex` to `WebDir` and initialize it by sending `asRegex` to our regex string. Modify `WebDir>>htmlFiles` to use the same regex each time as follows:

```
WebDir>>initialize
  htmlRegex := '*.html' asRegex

WebDir>>htmlFiles
  ↑ webDir fileNames select: [ :each | htmlRegex matches: each ]
```

Now listing the HTML files should work just as it did before, except that we reuse the same regex object many times.

Accessing web pages

Accessing the details of individual web pages should be the responsibility of a separate class, so let's define it, and let the WebDir class create the instances.

 Define a class `WebPage` with instance variables `path`, to identify the HTML file, and `homePath`, to identify the root directory of the web site. (We will need this to correctly generate links from the root of the web site to the files it contains.) Define an initialization method on the instance side and a creation method on the class side.

```
WebPage>>initializePath: filePath homePath: dirPath
  path := filePath.
  homePath := dirPath

WebPage class>>on: filePath forHome: homePath
  ↑ self new initializePath: filePath homePath: homePath
```

A WebDir instance should be able to return a list of all the web pages it contains.

 Add the following method to `WebDir`, and inspect the return value to verify that it works correctly.

```
WebDir>>webPages
  ↑ self htmlFiles collect:
    [ :each | WebPage
      on: webDir pathname, '/', each
      forHome: homePath ]
```

You should see something like this:

```
(WebDir onPath: '...') webPages → an Array(a WebPage a WebPage ...)
```

String substitutions

That's not very informative, so let's use a regex to get the actual file name for each web page. To do this, we want to strip away all the characters from

the path name up to the last directory. On a Unix file system directories end with a slash (/), so we need to delete everything up to the last slash in the file path.

The String extension method `copyWithRegex:matchesReplacedWith:` does what we want:

```
'hello' copyWithRegex: '[elo]+' matchesReplacedWith: 'i' → 'hi'
```

In this example the regex [elo] matches any of the characters e, l or o. The operator + is like the Kleene star, but it matches exactly *one* or more instances of the regex preceding it. Here it will match the entire substring 'ello' and replay it in a fresh string with the letter i.

 Add the following method and verify that it works as expected.

```
WebPage>>fileName
↑ path copyWithRegex: '.*/' matchesReplacedWith: ''
```

Now you should see something like this on your test web site:

```
(WebDir onPath: '...') webPages collect: [:each | each fileName ]
→ #('index.html' ...)
```

Extracting regex matches

Our next task is to extract the title of each HTML page.

First we need a way to get at the contents of each page. This is straightforward.

 Add the following method and try it out.

```
WebPage>>contents
↑ (FileStream oldFileOrNoneNamed: path) contents
```

Actually, you might have problems if your web pages contain non-ascii characters, in which case you might be better off with the following code:

```
WebPage>>contents
↑ (FileStream oldFileOrNoneNamed: path)
converter: Latin1TextConverter new;
contents
```

You should now be able to see something like this:

```
(WebDir onPath: '...') webPages first contents → '<head>
<title>Home Page</title>
```

```
...
'
```

Now let's extract the title. In this case we are looking for the text that occurs *between* the HTML tags `<title>` and `</title>`.

What we need is a way to extract *part* of the match of a regular expression. Subexpressions of regexes are delimited by parentheses. Consider the regex `([^aeiou]+)([aeiou]+)`. It consists of two subexpressions, the first of which will match a sequence of one or more non-vowels, and the second of which will match one or more vowels. (The operator `^` at the start of a bracketed set of characters negates the set.⁴) Now we will try to match a *prefix* of the string 'pharo' and extract the submatches:

```
re := '([^aeiou]+)([aeiou]+)' asRegex.
re matchesPrefix: 'pharo'    → true
re subexpression: 1          → 'pha'
re subexpression: 2          → 'ph'
re subexpression: 3          → 'a'
```

After successfully matching a regex against a string, you can always send it the message `subexpression: 1` to extract the entire match. You can also send `subexpression: n` where $n - 1$ is the number of subexpressions in the regex. The regex above has two subexpressions, numbered 2 and 3.

We will use the same trick to extract the title from an HTML file.

Define the following method:

```
WebPage>>title
| re |
re := '[w\W]*<title>(.*)</title>' asRegexIgnoringCase.
↑ (re matchesPrefix: self contents)
  ifTrue: [ re subexpression: 2 ]
  ifFalse: [ ('', self fileName, '-- untitled') ]
```

There are a couple of subtle points to notice here. First, HTML does not care whether tags are upper or lower case, so we must make our regex case insensitive by instantiating it with `asRegexIgnoringCase`.

Second, since dot matches any character *except a newline*, the regex `*<title>(.*)</title>` will not work as expected if multiple lines appear before the title. The regex `w` matches any alphanumeric, and `\W` will match any non-alphanumeric, so `[w\W]` will match any character *including newlines*. (If we expect titles to possibly contain newlines, we should play the same trick with the subexpression.)

⁴NB: In Pharo the caret is also the return keyword, which we write as `↑`. To avoid confusion, we will write `^` when we are using the caret within regular expressions to negate sets of characters, but you should not forget, they are actually the same thing.

Now we can test our title extractor, and we should see something like this:

```
(WebDir onPath: '...') webPages first title → 'Home page'
```

More string substitutions

In order to generate our site map, we need to generate links to the individual web pages. We can use the document title as the name of the link. We just need to generate the right path to the web page from the root of the web site. Luckily this is trivial — it is simple the full path to the web page minus the full path to the root directory of the web site.

We must only watch out for one thing. Since the `homePath` variable does not end in a `/`, we must append one, so that relative path does not include a leading `/`. Notice the difference between the following two results:

```
'/home/testweb/index.html' copyWithRegex: '/home/testweb' matchesReplacedWith: "
    → '/index.html'
'/home/testweb/index.html' copyWithRegex: '/home/testweb/' matchesReplacedWith: "
    → 'index.html'
```

The first result would give us an absolute path, which is probably not what we want.

 *Define the following methods:*

```
WebPage>>relativePath
↑ path
    copyWithRegex: homePath , '/'
    matchesReplacedWith: "

WebPage>>link
↑ '<a href=""', self relativePath, '">', self title, '</a>'
```

You should now be able to see something like this:

```
(WebDir onPath: '...') webPages first link → '<a href="index.html">Home Page</a>'
```

Generating the site map

Actually, we are now done with the regular expressions we need to generate the site map. We just need a few more methods to complete the application.

 *If you want to see the site map generation, just add the following methods.*

If our web site has subdirectories, we need a way to access them:

```
WebDir>>webDirs
↑ webDir directoryNames
  collect: [ :each | WebDir onPath: webDir pathName , '/' , each home: homePath ]
```

We need to generate HTML bullet lists containing links for each web page of a web directory. Subdirectories should be indented in their own bullet list.

```
WebDir>>printTocOn: aStream
self htmlFiles
ifNotEmpty: [
  aStream nextPutAll: '<ul>'; cr.
  self webPages
    do: [:each | aStream nextPutAll: '<li>';
      nextPutAll: each link;
      nextPutAll: '</li>'; cr].
  self webDirs
    do: [:each | each printTocOn: aStream].
aStream nextPutAll: '</ul>'; cr]
```

We create a file called “toc.html” in the root web directory and dump the site map there.

```
WebDir>>tocFileName
↑ 'toc.html'

WebDir>>makeToc
| tocStream |
tocStream := webDir newFileNamed: self tocFileName.
self printTocOn: tocStream.
tocStream close.
```

Now we can generate a table of contents for an arbitrary web directory!

```
WebDir selectHome makeToc
```

14.2 Regex syntax

We will now have a closer look at the syntax of regular expressions as supported by the Regex package.

The simplest regular expression is a single character. It matches exactly that character. A sequence of characters matches a string with exactly the same sequence of characters:

'a' matchesRegex: 'a'	→	true
-----------------------	---	------

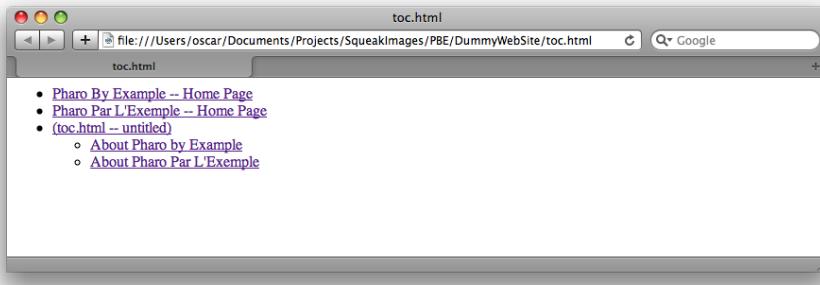


Figure 14.2: A small site map

```
'foobar' matchesRegex: 'foobar'    →  true
'blorpel' matchesRegex: 'foobar'   →  false
```

Operators are applied to regular expressions to produce more complex regular expressions. Sequencing (placing expressions one after another) as an operator is, in a certain sense, “invisible”—yet it is arguably the most common.

We have already seen the Kleene star (*) and the + operator. A regular expression followed by an asterisk matches any number (including 0) of matches of the original expression. For example:

```
'ab' matchesRegex: 'a*b'        →  true
'aaaaaab' matchesRegex: 'a*b'   →  true
'b' matchesRegex: 'a*b'        →  true
'aac' matchesRegex: 'a*b'      →  false  "b does not match"
```

The Kleene star has higher precedence than sequencing. A star applies to the shortest possible subexpression that precedes it. For example, ab* means a followed by zero or more occurrences of b, not “zero or more occurrences of ab”:

```
'abbb' matchesRegex: 'ab*'    →  true
'abab' matchesRegex: 'ab*'    →  false
```

To obtain a regex that matches “zero or more occurrences of ab”, we must enclose ab in parentheses:

```
'abab' matchesRegex: '(ab)*'  →  true
'abcab' matchesRegex: '(ab)*' →  false  "c spoils the fun"
```

Two other useful operators similar to * are + and ?. + matches one or more instances of the regex it modifies, and ? will match zero or one instance.

'ac' matchesRegex: 'ab*c'	→	true
'ac' matchesRegex: 'ab+c'	→	false "need at least one b"
'abbc' matchesRegex: 'ab+c'	→	true
'abbc' matchesRegex: 'ab?c'	→	false "too many b's"

As we have seen, the characters *, +, ?, (, and) have special meaning within regular expressions. If we need to match any of them literally, it should be escaped by preceding it with a backslash \. Thus, backslash is also special character, and needs to be escaped for a literal match. The same holds for all further special characters we will see.

'ab*' matchesRegex: 'ab*'	→	false "star in the right string is special"
'ab*' matchesRegex: 'ab*'	→	true
'a\c' matchesRegex: 'a\\c'	→	true

The last operator is |, which expresses choice between two subexpressions. It matches a string if either of the two subexpressions matches the string. It has the lowest precedence—even lower than sequencing. For example, ab*|ba* means “a followed by any number of b’s, or b followed by any number of a’s”:

'abb' matchesRegex: 'ab* ba*'	→	true
'baa' matchesRegex: 'ab* ba*'	→	true
'baab' matchesRegex: 'ab* ba*'	→	false

A bit more complex example is the expression c(a|d)+r, which matches the name of any of the Lisp-style car, cdr, caar, cdr, ... functions:

'car' matchesRegex: 'c(a d)+r'	→	true
'cdr' matchesRegex: 'c(a d)+r'	→	true
'cadr' matchesRegex: 'c(a d)+r'	→	true

It is possible to write an expression that matches an empty string, for example the expression a| matches an empty string. However, it is an error to apply *, +, or ? to such an expression: (a|)* is invalid.

So far, we have used only characters as the *smallest* components of regular expressions. There are other, more interesting, components. A character set is a string of characters enclosed in square brackets. It matches any single character if it appears between the brackets. For example, [01] matches either 0 or 1:

'0' matchesRegex: '[01]'	→	true
'3' matchesRegex: '[01]'	→	false
'11' matchesRegex: '[01]'	→	false "a set matches only one character"

Using plus operator, we can build the following binary number recognizer:

```
'10010100' matchesRegex: '[01]+' → true
'10001210' matchesRegex: '[01]+' → false
```

If the first character after the opening bracket is `^`, the set is inverted: it matches any single character *not* appearing between the brackets:

```
'0' matchesRegex: '['01]' → false
'3' matchesRegex: '['01]' → true
```

For convenience, a set may include ranges: pairs of characters separated by a hyphen (`-`). This is equivalent to listing all characters in between: `[0-9]` is the same as `[0123456789]`. Special characters within a set are `^`, `-`, and `]`, which closes the set. Below are examples how to literally match them in a set:

```
'^' matchesRegex: '[01^]' → true "put the caret anywhere except the start"
'-' matchesRegex: '[01-]' → true "put the hyphen at the end"
']' matchesRegex: '[]01]' → true "put the closing bracket at the start"
```

Thus, empty and universal sets cannot be specified.

Character classes

Regular expressions can also include the following backquote escapes to refer to popular classes of characters: `\w` to match alphanumeric characters, `\d` to match digits, and `\s` to match whitespace. Their upper-case variants, `\W`, `\D` and `\S`, match the complementary characters (non-alphanumeric, non-digits and non-whitespace). We can see a summary of the syntax seen so far in Table 14.1.

As mentioned in the introduction, regular expressions are especially useful for validating user input, and character classes turn out to be especially useful for defining such regexes. For example, non-negative numbers can be matched with the regex `\d+`:

```
'42' matchesRegex: '\d+' → true
'-1' matchesRegex: '\d+' → false
```

Better yet, we might want to specify that non-zero numbers should not start with the digit 0:

```
'0' matchesRegex: '0|([1-9]\d*)' → true
'1' matchesRegex: '0|([1-9]\d*)' → true
'42' matchesRegex: '0|([1-9]\d*)' → true
'099' matchesRegex: '0|([1-9]\d*)' → false "leading 0"
```

We can check for negative and positive numbers as well:

Syntax	What it represents
a	literal match of character a
.	match any char (except newline)
(...)	group subexpression
\	escape following special character
*	Kleene star — match previous regex zero or more times
+	match previous regex one or more times
?	match previous regex zero times or once
	match choice of left and right regex
[abcd]	match choice of characters abcd
[^ abcd]	match negated choice of characters
[0-9]	match range of characters 0 to 9
\w	match alphanumeric
\W	match non-alphanumeric
\d	match digit
\D	match non-digit
\s	match space
\S	match non-space

Table 14.1: Regex Syntax in a Nutshell

'0' matchesRegex: '(0 ((\+ \-)?[1-9]\d*))'	→	true
'-1' matchesRegex: '(0 ((\+ \-)?[1-9]\d*))'	→	true
'42' matchesRegex: '(0 ((\+ \-)?[1-9]\d*))'	→	true
'+99' matchesRegex: '(0 ((\+ \-)?[1-9]\d*))'	→	true
'-0' matchesRegex: '(0 ((\+ \-)?[1-9]\d*))'	→	false "negative zero"
'01' matchesRegex: '(0 ((\+ \-)?[1-9]\d*))'	→	false "leading zero"

Floating point numbers should require at least one digit after the dot:

'0' matchesRegex: '(0 ((\+ \-)?[1-9]\d*))(\.\d+)?'	→	true
'0.9' matchesRegex: '(0 ((\+ \-)?[1-9]\d*))(\.\d+)?'	→	true
'3.14' matchesRegex: '(0 ((\+ \-)?[1-9]\d*))(\.\d+)?'	→	true
'-42' matchesRegex: '(0 ((\+ \-)?[1-9]\d*))(\.\d+)?'	→	true
'2.' matchesRegex: '(0 ((\+ \-)?[1-9]\d*))(\.\d+)?'	→	false "need digits after ."

For dessert, here is a recognizer for a general number format: anything like 999, or 999.999, or -999.999e+21.

```
'-999.999e+21' matchesRegex: '(\+|\-)?\d+(\.\d*)?((e|E)(\+|\-)?\d+)?' → true
```

Character classes can also include the grep(1)-compatible elements listed in Table 14.2.

Syntax	What it represents
[:alnum:]	any alphanumeric
[:alpha:]	any alphabetic character
[:cntrl:]	any control character (ascii code is < 32)
[:digit:]	any decimal digit
[:graph:]	any graphical character (ascii code >= 32)
[:lower:]	any lowercase character
[:print:]	any printable character (here, the same as [:graph:])
[:punct:]	any punctuation character
[:space:]	any whitespace character
[:upper:]	any uppercase character
[:xdigit:]	any hexadecimal character

Table 14.2: Regex character classes

Note that these elements are components of the character classes, *i.e.*, they have to be enclosed in an extra set of square brackets to form a valid regular expression. For example, a non-empty string of digits would be represented as `[[[:digit:]]]+`. The above primitive expressions and operators are common to many implementations of regular expressions.

```
'42' matchesRegex: '[[[:digit:]]]+' → true
```

Special character classes

The next primitive expression is unique to this Smalltalk implementation. A sequence of characters between colons is treated as a unary selector which is supposed to be understood by characters. A character matches such an expression if it answers true to a message with that selector. This allows a more readable and efficient way of specifying character classes. For example, `[0–9]` is equivalent to `:isDigit:`, but the latter is more efficient. Analogously to character sets, character classes can be negated: `^:isDigit:` matches a character that answers false to `isDigit`, and is therefore equivalent to `[^0–9]`.

So far we have seen the following equivalent ways to write a regular expression that matches a non-empty string of digits: `[0–9]+`, `\d+`, `\[d]+`, `[[[:digit:]]]+`, `:isDigit:+`.

```
'42' matchesRegex: '0-9+' → true
'42' matchesRegex: '\d+' → true
'42' matchesRegex: '\[d)+' → true
'42' matchesRegex: '[[[:digit:]]]+' → true
'42' matchesRegex: ':isDigit:+' → true
```

Matching boundaries

The last group of special primitive expressions is shown in Table 14.3, and is used to match boundaries of strings.

Syntax	What it represents
^	match an empty string at the beginning of a line
\$	match an empty string at the end of a line
\b	match an empty string at a word boundary
\B	match an empty string not at a word boundary
\<	match an empty string at the beginning of a word
\>	match an empty string at the end of a word

Table 14.3: Primitives to match string boundaries

```
'hello world' matchesRegex: '.*\bw.*'    → true   "word boundary before w"
'hello world' matchesRegex: '.*\bo.*'     → false  "no boundary before o"
```

14.3 Regex API

Up to now we have focussed mainly on the syntax of regexes. Now we will have a closer look at the different messages understood by strings and regexes.

Matching prefixes and ignoring case

So far most of our examples have used the String extension method `matchesRegex:`.

Strings also understand the following messages: `prefixMatchesRegex:`, `matchesRegexIgnoringCase:` and `prefixMatchesRegexIgnoringCase:`.

The message `prefixMatchesRegex:` is just like `matchesRegex`, except that the whole receiver is not expected to match the regular expression passed as the argument; matching just a prefix of it is enough.

```
'abacus' matchesRegex: '(a|b)+'           → false
'abacus' prefixMatchesRegex: '(a|b)+'      → true
'ABBA' matchesRegexIgnoringCase: '(a|b)+'   → true
'Abacus' matchesRegexIgnoringCase: '(a|b)+'  → false
'Abacus' prefixMatchesRegexIgnoringCase: '(a|b)+' → true
```

Enumeration interface

Some applications need to access *all* matches of a certain regular expression within a string. The matches are accessible using a protocol modeled after the familiar Collection-like enumeration protocol.

`regex:matchesDo:` evaluates a one-argument aBlock for every match of the regular expression within the receiver string.

```
list := OrderedCollection new.  
'Jack meet Jill' regex: '\w+' matchesDo: [:word | list add: word].  
list → an OrderedCollection('Jack' 'meet' 'Jill')
```

`regex:matchesCollect:` evaluates a one-argument aBlock for every match of the regular expression within the receiver string. It then collects the results and answers them as a SequenceableCollection.

```
'Jack meet Jill' regex: '\w+' matchesCollect: [:word | word size] →  
an OrderedCollection(4 4 4)
```

`allRegexMatches:` returns a collection of all matches (substrings of the receiver string) of the regular expression.

```
'Jack and Jill went up the hill' allRegexMatches: '\w+' →  
an OrderedCollection('Jack' 'and' 'Jill' 'went' 'up' 'the' 'hill')
```

Replacement and translation

It is possible to replace all matches of a regular expression with a certain string using the message `copyWithRegex:matchesReplacedWith:..`

```
'Krazy hates Ignatz' copyWithRegex: '<[:lower:]>' matchesReplacedWith: 'loves'  
→ 'Krazy loves Ignatz'
```

A more general substitution is match translation. This message evaluates a block passing it each match of the regular expression in the receiver string and answers a copy of the receiver with the block results spliced into it in place of the respective matches.

```
'Krazy loves Ignatz' copyWithRegex: 'b[a-z]+b' matchesTranslatedUsing: [:each | each  
asUppercase] → 'Krazy LOVES Ignatz'
```

All messages of enumeration and replacement protocols perform a case-sensitive match. Case-insensitive versions are not provided as part of a String protocol. Instead, they are accessible using the lower-level matching interface presented in the following question.

Lower-level interface

When you send the message `matchesRegex:` to a string, the following happens:

1. A fresh instance of `RxParser` is created, and the regular expression string is passed to it, yielding the expression's syntax tree.
2. The syntax tree is passed as an initialization parameter to an instance of `RxMatcher`. The instance sets up some data structure that will work as a recognizer for the regular expression described by the tree.
3. The original string is passed to the matcher, and the matcher checks for a match.

The Matcher

If you repeatedly match a number of strings against the same regular expression using one of the messages defined in `String`, the regular expression string is parsed and a new matcher is created for every match. You can avoid this overhead by building a matcher for the regular expression, and then reusing the matcher over and over again. You can, for example, create a matcher at a class or instance initialization stage, and store it in a variable for future use. You can create a matcher using one of the following methods:

- You can send `asRegex` or `asRegexIgnoringCase` to the string.
- You can directly invoke the `RxMatcher` constructor methods `forString:` or `forString:ignoreCase:` (which is what the convenience methods above will do).

Here we send `matchesIn:` to collect all the matches found in a string:

```
octal := '8r[0-9A-F]+' asRegex.
octal matchesIn: '8r52 = 16r2A'    →  an OrderedCollection('8r52')
```

```
hex := '16r[0-9A-F]+' asRegexIgnoringCase.
hex matchesIn: '8r52 = 16r2A'    →  an OrderedCollection('16r2A')
```

```
hex := RxMatcher forString: '16r[0-9A-Fa-f]+' ignoreCase: true.
hex matchesIn: '8r52 = 16r2A'    →  an OrderedCollection('16r2A')
```

Matching

A matcher understands these messages (all of them return `true` to indicate successful match or search, and `false` otherwise):

`matches: aString` – `true` if the whole argument string (`aString`) matches.

```
'\w+' asRegex matches: 'Krazy' → true
```

`matchesPrefix: aString` – true if some prefix of the argument string (not necessarily the whole string) matches.

```
'\w+' asRegex matchesPrefix: 'Ignatz hates Krazy' → true
```

`search: aString` – Search the string for the first occurrence of a matching substring. (Note that the first two methods only try matching from the very beginning of the string). Using the above example with a matcher for `a+`, this method would answer success given a string 'baaa', while the previous two would fail.

```
'\b[a-z]+\b' asRegex search: 'Ignatz hates Krazy' → true "finds 'hates'"
```

The matcher also stores the outcome of the last match attempt and can report it: `lastResult` answers a Boolean: the outcome of the most recent match attempt. If no matches were attempted, the answer is unspecified.

```
number := '\d+' asRegex.  
number search: 'Ignatz throws 5 bricks'.  
number lastResult → true
```

`matchesStream:`, `matchesStreamPrefix:` and `searchStream:` are analogous to the above three messages, but takes streams as their argument.

```
ignatz := ReadStream on: 'Ignatz throws bricks at Krazy'.  
names := '<[A-Z][a-z]+>' asRegex.  
names matchesStreamPrefix: ignatz → true
```

Subexpression matches

After a successful match attempt, you can query which part of the original string has matched which part of the regex. A subexpression is a parenthesized part of a regular expression, or the whole expression. When a regular expression is compiled, its subexpressions are assigned indices starting from 1, depth-first, left-to-right.

For example, the regex `((\d+)\s*(\w+))` has four subexpressions, including itself.

- | | | |
|----|------------------------------|-----------------------------------|
| 1: | <code>((\d+)\s*(\w+))</code> | "the complete expression" |
| 2: | <code>(\d+)\s*(\w+)</code> | "top parenthesized subexpression" |
| 3: | <code>\d+</code> | "first leaf subexpression" |
| 4: | <code>\w+</code> | "second leaf subexpression" |

The highest valid index is equal to 1 plus the number of matching parentheses. (So, 1 is always a valid index, even if there are no parenthesized subexpressions.)

After a successful match, the matcher can report what part of the original string matched what subexpression. It understands these messages:

`subexpressionCount` answers the total number of subexpressions: the highest value that can be used as a subexpression index with this matcher. This value is available immediately after initialization and never changes.

`subexpression`: takes a valid index as its argument, and may be sent only after a successful match attempt. The method answers a substring of the original string the corresponding subexpression has matched to.

`subBeginning`: and `subEnd`: answer the positions within the argument string or stream where the given subexpression match has started and ended, respectively.

```
items := '(\d+)\s*(\w+))' asRegex.
items search: 'Ignatz throws 1 brick at Krazy'.
items subexpressionCount   → 4
items subexpression: 1     → '1 brick'  "complete expression"
items subexpression: 2     → '1 brick'  "top subexpression"
items subexpression: 3     → '1'        "first leaf subexpression"
items subexpression: 4     → 'brick'    "second leaf subexpression"
items subBeginning: 3      → 14
items subEnd: 3            → 15
items subBeginning: 4      → 16
items subEnd: 4            → 21
```

As a more elaborate example, the following piece of code uses a MMM DD, YYYY date format recognizer to convert a date to a three-element array with year, month, and day strings:

```
date := '(Jan|Feb|Mar|Apr|May|Jun|Jul|Aug|Sep|Oct|Nov|Dec)\s+(\d\d?)\s*,\s*\d\d)' asRegex.
result := (date matches: 'Aug 6, 1996')
ifTrue: [{(date subexpression: 4) .
          (date subexpression: 2) .
          (date subexpression: 3)}]
ifFalse: ['no match'].
result → #('96' 'Aug' '6')
```

Enumeration and Replacement

The String enumeration and replacement protocols that we saw earlier in this section are actually implemented by the matcher. RxMatcher implements the following methods for iterating over matches within

strings: matchesIn:, matchesIn:do:, matchesIn:collect:, copy:replacingMatchesWith: and copy:translatingMatchesUsing::

```

seuss := 'The cat in the hat is back'.
aWords := '\<([^\aeiou][a])+\>' asRegex. "match words with 'a' in them"
aWords matchesIn: seuss
    → an OrderedCollection('cat' 'hat' 'back')
aWords matchesIn: seuss collect: [:each | each asUppercase ]
    → an OrderedCollection('CAT' 'HAT' 'BACK')
aWords copy: seuss replacingMatchesWith: 'grinch'
    → 'The grinch in the grinch is grinch'
aWords copy: seuss translatingMatchesUsing: [ :each | each asUppercase ]
    → 'The CAT in the HAT is BACK'

```

There are also the following methods for iterating over matches within streams: matchesOnStream:, matchesOnStream:do:, matchesOnStream:collect:, copyStream:to:replacingMatchesWith: and copyStream:to:translatingMatchesUsing::

```

in := ReadStream on: '12 drummers, 11 pipers, 10 lords, 9 ladies, etc.'.
out := WriteStream on: ''.
numMatch := '\<\d+\>' asRegex.
numMatch
copyStream: in
to: out
translatingMatchesUsing: [:each | each asNumber asFloat asString ].
out close; contents → '12.0 drummers, 11.0 pipers, 10.0 lords, 9.0 ladies, etc.'

```

Error Handling

Several exceptions may be raised by RxParser when building regexes. The exceptions have the common parent RegexError. You may use the usual Smalltalk exception handling mechanism to catch and handle them.

- RegexSyntaxError is raised if a syntax error is detected while parsing a regex
- RegexCompilationError is raised if an error is detected while building a matcher
- RegexMatchingError is raised if an error occurs while matching (for example, if a bad selector was specified using ':<selector>:' syntax, or because of the matcher's internal error)

```

[+' asRegex] on: RegexError do: [:ex | ex printString ]
    → 'RegexSyntaxError: nullable closure'

```

14.4 Implementation Notes by Vassili Bykov

What to look at first. In 90% of the cases, the method `String»matchesRegex:` is all you need to access the package.

`RxParser` accepts a string or a stream of characters with a regular expression, and produces a syntax tree corresponding to the expression. The tree is made of instances of `Rxs*` classes.

`RxMatcher` accepts a syntax tree of a regular expression built by the parser and compiles it into a matcher: a structure made of instances of `Rxm*` classes. The `RxMatcher` instance can test whether a string or a positionable stream of characters matches the original regular expression, or it can search a string or a stream for substrings matching the expression. After a match is found, the matcher can report a specific string that matched the whole expression, or any parenthesized subexpression of it. All other classes support the same functionality and are used by `RxParser`, `RxMatcher`, or both.

Caveats The matcher is similar in spirit, but *not* in design to Henry Spencer's original regular expression implementation in C. The focus is on simplicity, not on efficiency. I didn't optimize or profile anything. The matcher passes H. Spencer's test suite (see "test suite" protocol), with quite a few extra tests added, so chances are good there are not too many bugs. But watch out anyway.

Acknowledgments Since the first release of the matcher, thanks to the input from several fellow Smalltalkers, I became convinced a native Smalltalk regular expression matcher was worth the effort to keep it alive. For the advice and encouragement that made this release possible, I want to thank: Felix Hack, Eliot Miranda, Robb Shecter, David N. Smith, Francis Wolinski and anyone whom I haven't yet met or heard from, but who agrees this has not been a complete waste of time.

14.5 Chapter Summary

Regular expressions are an essential tool for manipulating strings in a trivial way. This chapter presented the `Regex` package for Pharo. The essential points of this chapter are:

- For simple matching, just send `matchesRegex:` to a string
- When performance matters, send `asRegex` to the string representing the regex, and reuse the resulting matcher for multiple matches

- Subexpression of a matching regex may be easily retrieved to an arbitrary depth
- A matching regex can also replace or translate subexpressions in a new copy of the string matched
- An enumeration interface is provided to access all matches of a certain regular expression
- Regexes work with streams as well as with strings.

Part V

Source Management

Chapter 15

Versioning your code with Monticello

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

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

Many different versioning systems are available. CVS¹ and Subversion² are probably the most popular. In principle you could use them to manage the development of Pharo software projects, but such a practice would disconnect the versioning system from the Pharo environment. In addition, CVS-like tools only version plain text files and not individual packages, classes or methods. We would therefore lack the ability to track changes at the appropriate level of granularity. If the versioning tools know that you store classes and methods instead of plain text, they can do a better job of supporting the development process.

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

¹<http://www.nongnu.org/cvs>

²<http://subversion.tigris.org>

In this chapter, you will learn how to use use Monticello and Squeak-Source to manage your software. We have already met Monticello briefly in earlier chapters³. This chapter delves into the details of Monticello and describes some additional features that are useful for versioning large applications.

15.1 Basic usage

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

Running example — perfect numbers

We will use a small running example of perfect numbers⁴ in this chapter to illustrate the features of Monticello. We will start our project by defining some simple tests.

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

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

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

Launching Monticello

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

In Figure 15.1 we see that the Monticello Browser consists of two list panes and one button pane. The left pane lists installed packages and the right panes shows known repositories. Various operations may be performed via the button pane and the menus of the two list panes.

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

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

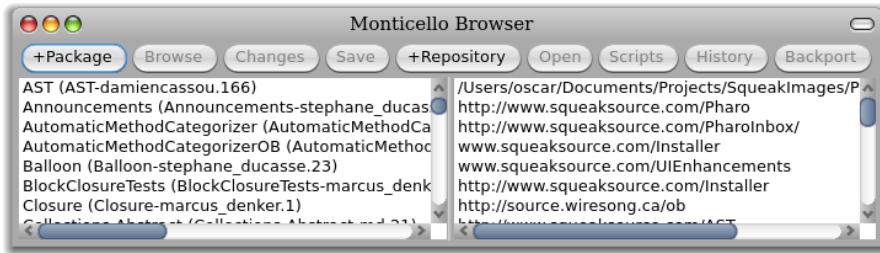


Figure 15.1: The Monticello Browser.

Creating a package

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

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

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

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

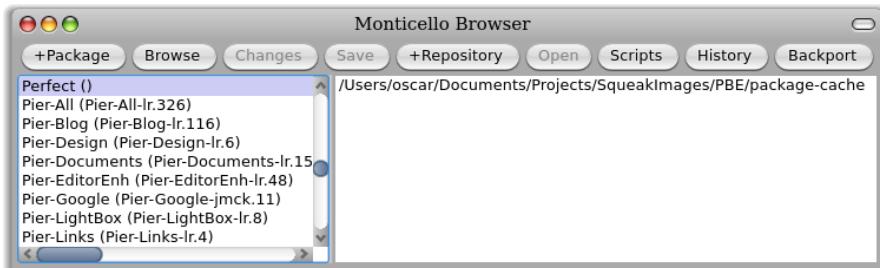


Figure 15.2: Creating the Perfect package.

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

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

- All methods belonging to *any* class (in any category) that are defined in a protocol named `*perfect` or `*Perfect`, or in protocols whose names start with `*perfect-` or `*Perfect-`. Such methods are known as *extensions*. We don't have any yet, but we will define some very soon.
- All methods belonging to any classes in the category `Perfect`, or in categories whose names begin with `Perfect-`, *except* those in protocols whose names start with `*` (*i.e.*, those belonging to *other* packages). This includes our `testPerfect` method, since it belongs to the protocol running.

Committing changes

Note in Figure 15.2 that the `Save` button is disabled (greyed out).

Before we save our `Perfect` package, we need to specify *where* we want to save it. A *repository* is a package container, which may either be local to your machine or remote (accessed over the network). Various protocols may be used to establish a connection between your Pharo image and a repository. As we will see later (Section 15.5), Monticello supports a large choice of repositories, though the most commonly used is HTTP, since this is the one used by SqueakSource.

At least one repository, called `package-cache`, is set up by default, and is shown as the first entry in the list of repositories on the right-hand side of your Monticello browser (see Figure 15.1). The `package-cache` is created automatically in the local directory where your Pharo image is located. It will contain a copy of all the packages you download from remote repositories. By default, copies of your packages are also saved in the `package-cache` when you save them to a remote server.

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

 Select the directory repository named `package cache`, press `Save`, enter an appropriate log message, and `Accept` to save the changes.

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

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

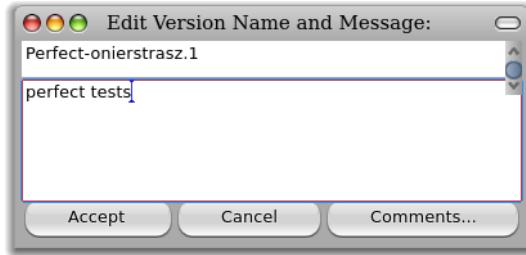


Figure 15.3: You may set a new version name and a commit message when you save a version of a package.

confirm that a file `Perfect-XX.1.mc2` was created in your package cache. XX corresponds to your name or initials.⁵

A *version* is an immutable snapshot of a package that has been written to a repository. Each version has a unique version number to identify it in a repository. Be aware, however, that this number is *not* globally unique—in another repository you might have the same file identifier for a *different snapshot*. For example, `Perfect-onierstrasz.1.mc2` in another repository might be the *final*, deployed version of our project! When saving a version into a repository, the next available number is automatically assigned to the version, but you can change this number if you wish. Note that version branches do not interfere with the numbering scheme (as with CVS or Subversion). As we shall see later, versions are by default ordered by their version number when viewing a repository.

Class extensions

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

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

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

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

PerfectTest»testPerfectBoundary
```

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

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

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

“Clean” and “Dirty” packages

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

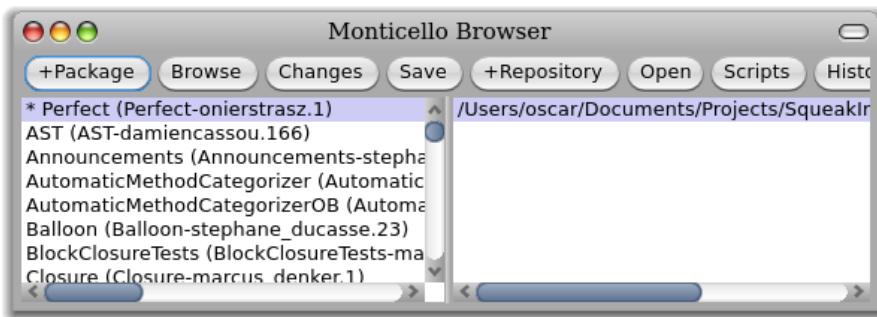


Figure 15.4: Modifying our *Perfect* package will “dirty” it.

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

Try the `Browse`, `History` and `Changes` buttons to see what they do⁶. `Save` the changes to the *Perfect* package. Confirm that the package is now “clean” again.

The Repository inspector

The contents of a repository can be explored using a repository inspector, which is launched using the `Open` button of Monticello (cf Figure 15.5).

Select the package–cache repository and open it. You should see something like Figure 15.5.

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

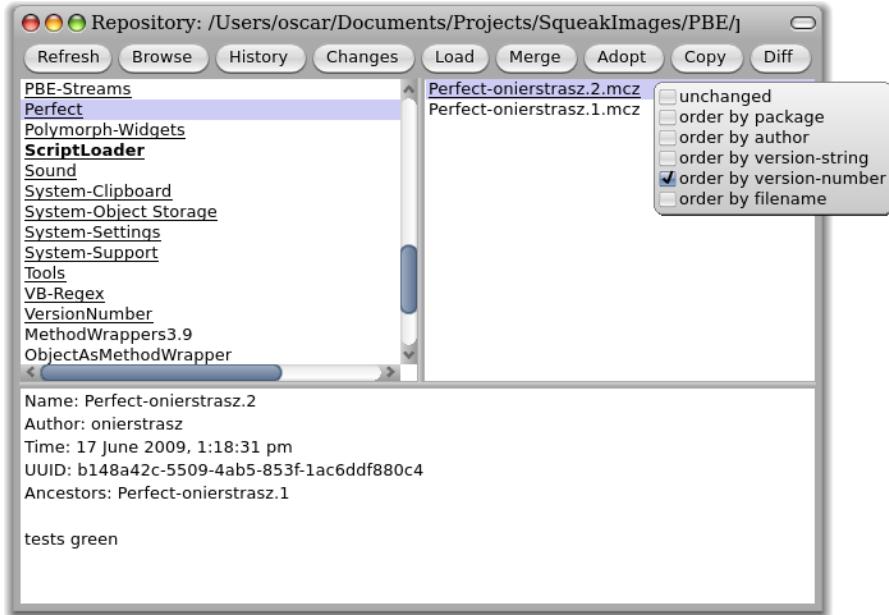


Figure 15.5: A repository inspector.

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

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

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

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

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

Loading, unloading and updating packages

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

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



Figure 15.6: Unloading a package.

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

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

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

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

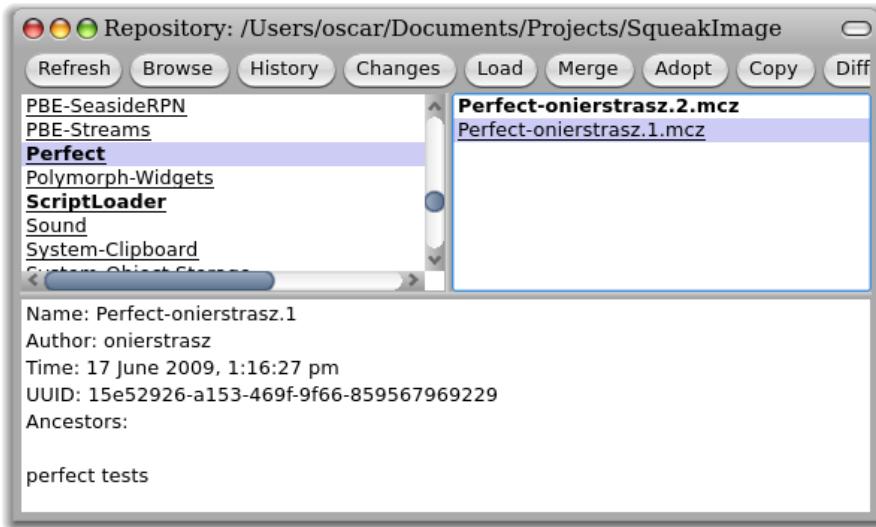


Figure 15.7: Loading an earlier version.

Now the tests should be green again.

Branching

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

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

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

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

```

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

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

```

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

- ⌚ Attempt to load version 2 of the Perfect package.

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



Figure 15.8: Unsaved changes warning.

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

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

- ⌚ If you still have the repository inspector open, **Refresh** it to see the new version (Figure 15.10).

Merging

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

```
Preferences disable: #useNewDiffToolsForMC
```

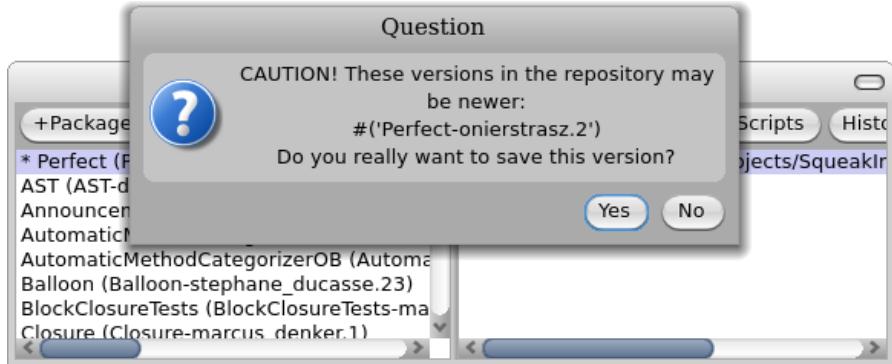


Figure 15.9: Newer versions warning.

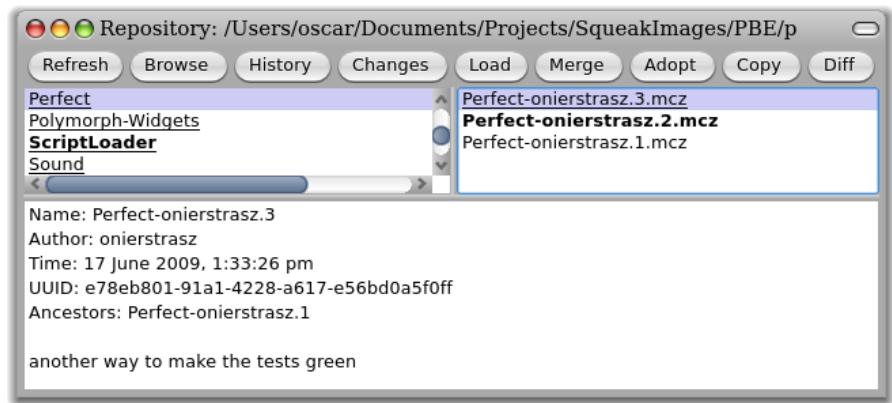


Figure 15.10: Versions 2 and 3 are separate branches of version 1.

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

Consider the current situation with our *Perfect* package, as illustrated at the left of Figure 15.11. We have published a new version 3 that is based on version 1. Since version 2 is also based on version 1, versions 2 and 3 constitute independent branches.

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

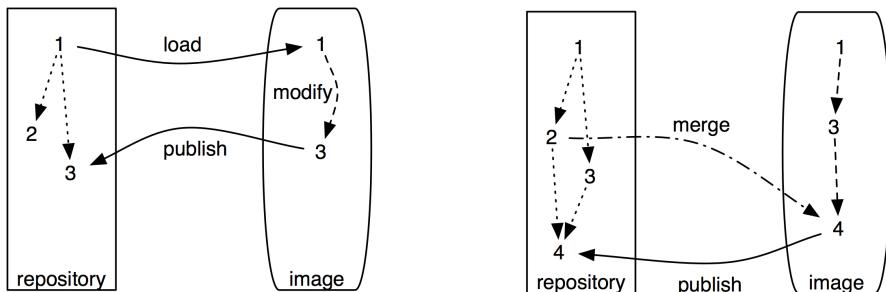


Figure 15.11: Branching (left) and merging (right).

currently loaded, we would like to merge in changes from version 2, and publish a new, merged version 4, as illustrated at the right of Figure 15.11.

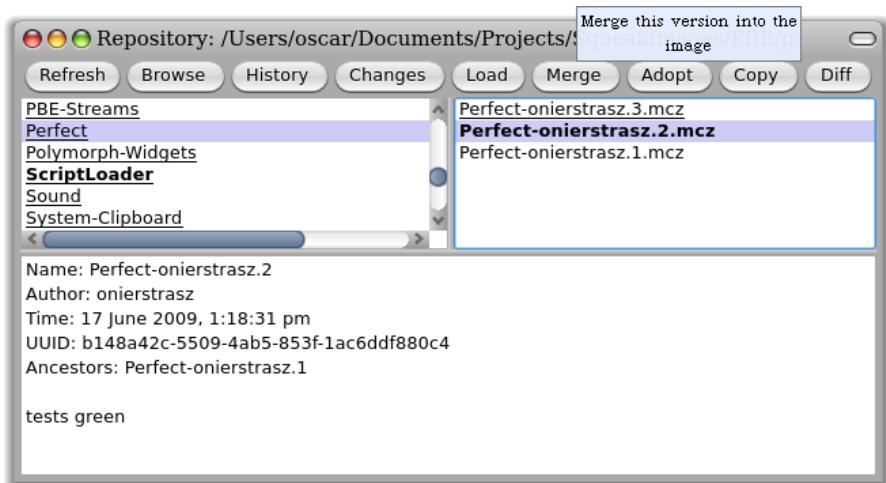


Figure 15.12: Select a separate branch (in bold) to be merged.

Select version 2 in the repository browser, as shown in Figure 15.12, and click the **Merge** button.

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

In Figure 15.13 we see the three differences between versions 2 and 3 of the Perfect package. The method `PerfectTest»testPerfectBoundary` is new, and the two indicated methods of `Integer` have been changed. In the lower pane

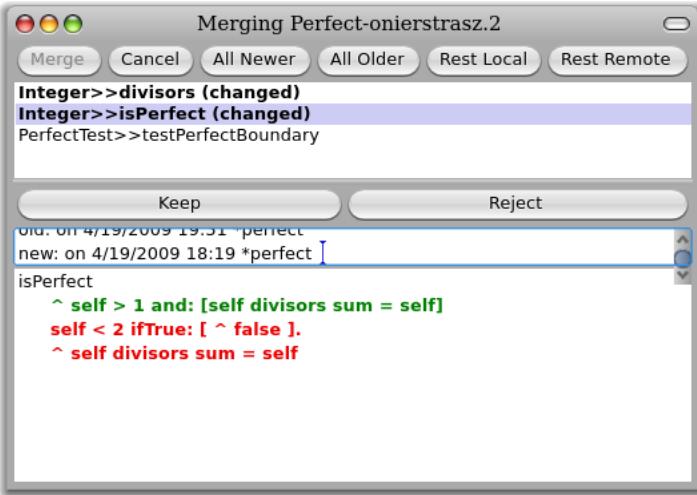


Figure 15.13: Version 2 of the Perfect package being merged with the current version 3.

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

A method or a class is in conflict if its definition has been altered. Figure 15.13 shows 2 conflicting methods in the class `Integer`: `isPerfect` and `divisors`. A conflicting package element is indicated by being underlined, ~~barred~~, or **bold**. The full set of typeface conventions is as follows:

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

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

Underlined=Repository replace current. An underlined element will be kept and replace the current element in the image. In Figure 15.14 we see that `Integer»isPerfect` from version 2 has been kept.

Barred=Repository version rejected. A ~~barred~~ element has been rejected, and the local definition will not be replaced. In Figure 15.14 `Integer»divisors` from version 2 is rejected, so the definition from version 3 will remain.

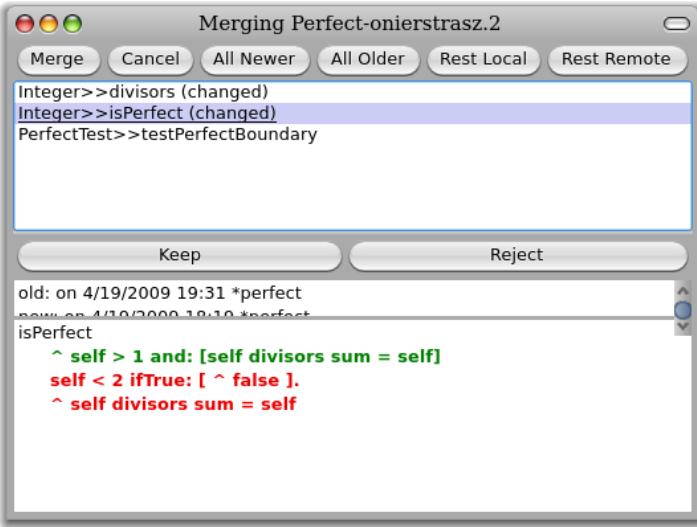


Figure 15.14: Keeping and rejecting changes.

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

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

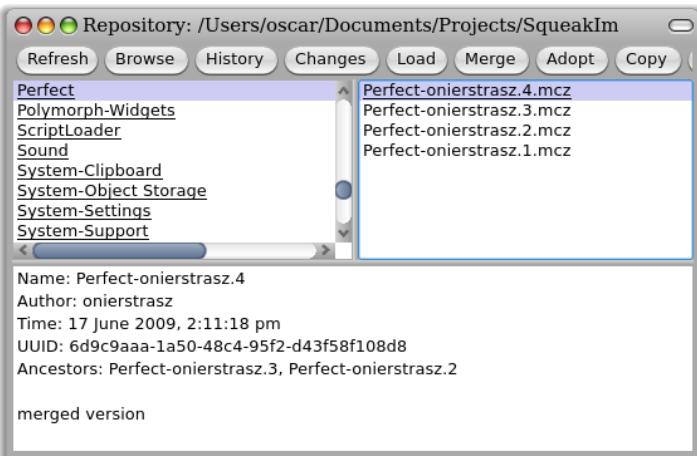


Figure 15.15: All older versions are now ancestors of merged version 4.

If you now refresh the repository inspector, you will see that there are no more versions shown in bold, *i.e.*, all versions are ancestors of the currently loaded version 4 (Figure 15.15).

15.2 Exploring Monticello repositories

Monticello has many other useful features. As we can see in Figure 15.1, the Monticello browser window has nine buttons. We have already used four of them—**+Package**, **Save**, **+Repository** and **Open**. We will now look at **Browse**, **Changes** and **History**, which are used to explore the state and history of repositories

Browse

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

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

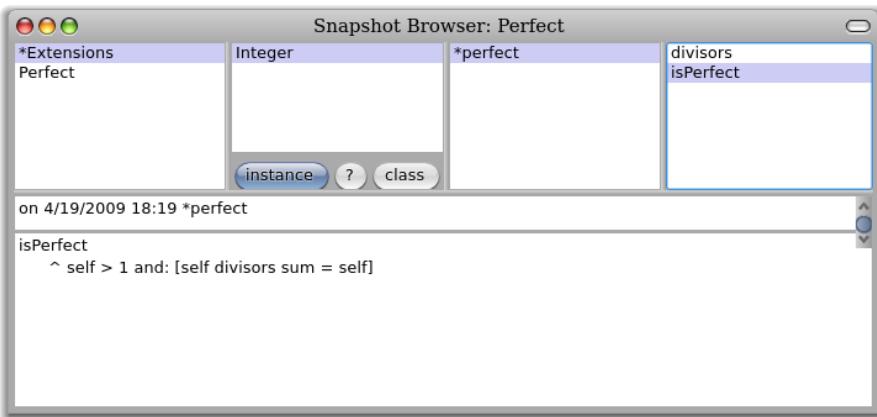


Figure 15.16: The snapshot browser reveals that the *Perfect* package extends the class *Integer* with 2 methods.

For example, Figure 15.16 shows the class extensions defined in the *Perfect* package. Note that code cannot be edited here, though by action-clicking, if your environment has been set up accordingly) on a class or a method name you can open a regular browser.

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

Changes

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

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

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

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

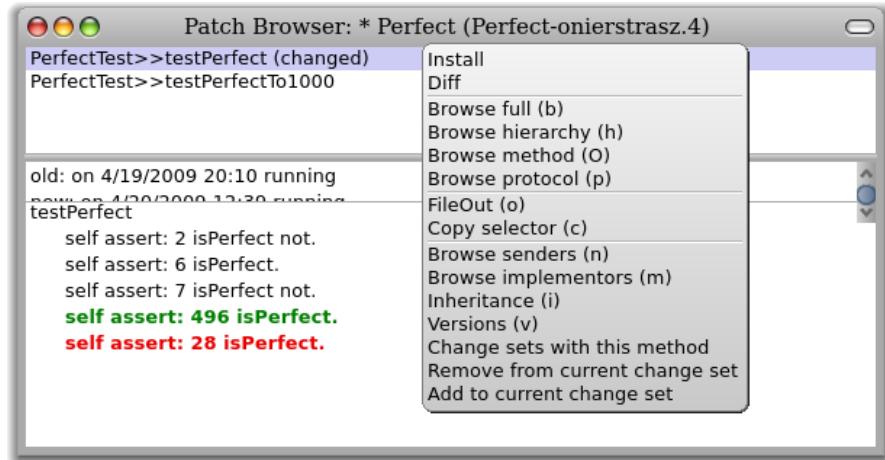


Figure 15.17: The patch browser shows the difference between the code in the image and the most recently committed version.

Figure 15.17 shows that the Perfect package has been locally modified with one changed method and one new method. As usual, action-clicking on a change offers you a choice of contextual operations.

History

The **History** button opens a version history viewer that displays the comments committed along with each version of the selected package (see Figure 15.18). The versions of the package, in this case Perfect, are listed on the left, while information about the selected version is displayed on the right.

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

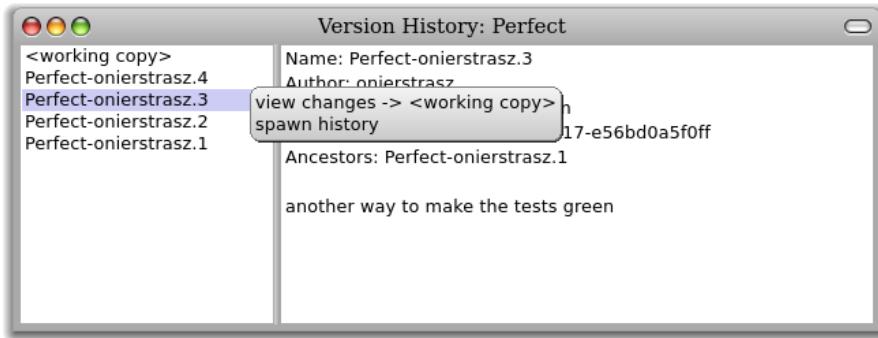


Figure 15.18: The version history viewer provides information about the various versions of a package.

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

15.3 Advanced topics

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

Backporting

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

The process is illustrated in Figure 15.19. Suppose that the main branch of our software system consists of versions A and B, maintained by Manny. A

contributor, Conny, has developed a separate experimental branch, C, with changes X and Y. Change X fixes a nasty problem in versions A and B, so Manny asks Conny to prepare a backported branch D containing *only* change X. Now Manny can merge B and D to produce a new version E that fixes the defect. Conny can continue to further develop her independent branch C.

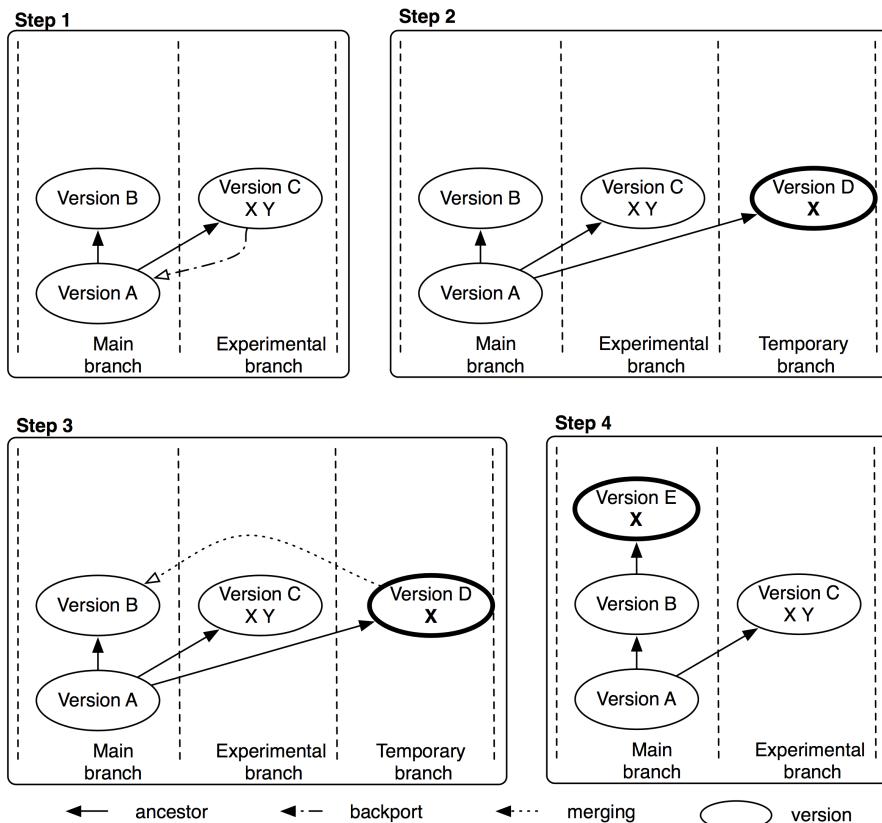


Figure 15.19: Change X is backported from version C to version A, producing a new branch D. D can then be merged into B, without affecting C.

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

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

Let us see how this works in practice. Recall that we earlier rejected the implementation of `isPerfect` when we merged versions 2 and 3 of the `Perfect` package. Now we will recover that change as a backport to version 1. (Versions 1, 2 and 3 play the roles of versions A, B and C, respectively, in Figure 15.19.)

 *Unload the `Perfect` package. Now open a repository inspector on your package `-cache` and load `Perfect` version 3. In the Monticello browser select `Perfect` and click on the `Backport` button. Select version 1 as the ancestor. You should be able to browse the changes between version 3 and 1, as shown in Figure 15.20. Now select the `Integer»isPerfect` method and click on `Select`.*

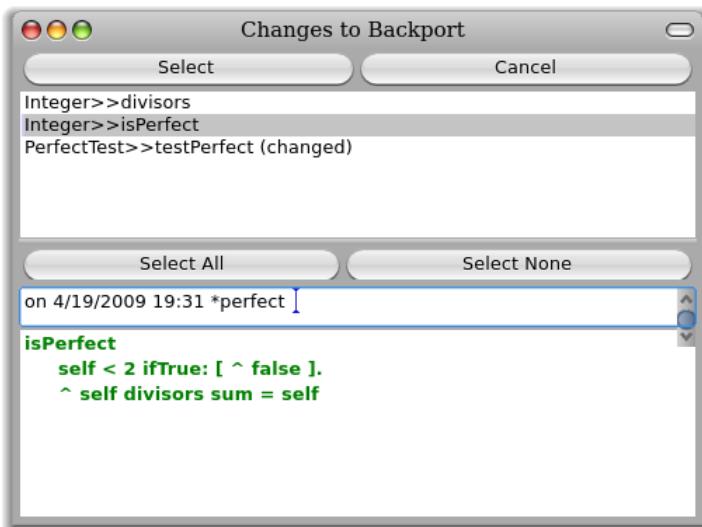


Figure 15.20: Backporting changes from version 3 to version 1 of the `Perfect` package.

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

Dependencies

Most applications cannot live on their own and typically require the presence of other packages in order to work properly. For example, let us have a look at Pier⁷, a meta-described content management system. Pier is a large piece of software with many facets (tools, documentations, blog, catch strategies, security, ...). Each facet is implemented by a separate package. Most Pier packages cannot be used in isolation since they refer to methods and classes defined in other packages. Monticello provides a dependency mechanism for declaring the *required packages* of a given package to ensure that it will be correctly loaded.

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

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

Figure 15.21 illustrates how this works in Pier. Package Pier-All is an *empty package* that acts as a kind of umbrella. It requires Pier-Blog, Pier-Caching and all the other Pier packages.

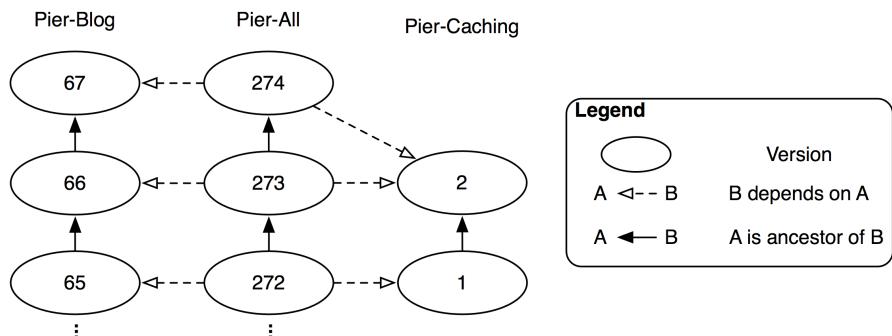


Figure 15.21: Dependencies in Pier.

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

⁷<http://source.lukas-renggli.ch/pier>

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

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

 Take the following steps:

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

Note that when NewPerfect-All is selected in the Monticello browser, the dependent packages are displayed in bold (see Figure 15.22).

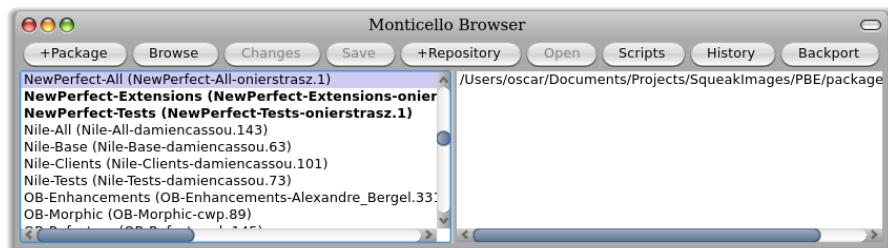


Figure 15.22: NewPerfect-All requires NewPerfect-Extensions and NewPerfect-Tests.

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

Here is the reason why:

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

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

Class initialization

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

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

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

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

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

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

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

15.4 Getting a change set from two versions

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

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

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

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

15.5 Kinds of repositories

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

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

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

HTTP repositories may be used with an HTTP server other than SqueakSource . For example, a simple configuration⁸ turns Apache into a Monticello repository with restricted access rights:

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

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

⁸<http://www.visoracle.com/squeak/faq/monticello-1.html>

GOODS. This repository type stores versions in a GOODS object database. GOODS is a fully distributed object-oriented database management system that uses an active client model⁹. It's a read-write repository, so it makes a good “working” repository where versions can be saved and retrieved. Because of the transaction support, journaling and replication capabilities offered by GOODS, it is suitable for large repositories used by many clients.

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

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

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

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

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

⁹<http://www.garret.ru/goods.html>

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

Using SqueakSource

SqueakSource is a online repository that you can use to store your Monticello packages. An instance is running and accessible from <http://www.squeaksourc e.com>. At the time this chapter is being written, over 1500 projects are registered on SqueakSource and nearly 2000 people have an account. Figure 15.23 shows the main web page.

The screenshot shows the homepage of SqueakSource. The header features the "SqueakSource" logo and a "up to date" status indicator. A navigation bar at the top includes links for Home, Projects, Tags, Members, Groups, and Help. On the left, a sidebar contains sections for Actions (RSS feed, Register Member, Register Group, Register Project) and Authentication (Login). The main content area is titled "Home". It welcomes users to the site, explaining the purpose of SqueakSource as a Monticello code repository for Squeak. It encourages users to register and manage their accounts, projects, and versions. Below this, a note states that the service is free and backed up daily, with a reminder to set up proper backups. It also mentions the Squeak Mailing-List and the support of the Software Composition Group and the University of Bern. A "Statistics" section shows member counts and recent activity. The "Projects" section lists 1533 projects, including recently created, used, active, and downloaded ones. At the bottom, there are links for XHTML, CSS, and RSS, along with a timestamp of 22 April 2009.

Figure 15.23: SqueakSource, the online Monticello code repository.

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

```
MCHttpRepository
location: 'http://www.squeaksourc e.com/PharoByExample'
user: ""
password: "
```

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

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

The screenshot shows the 'Register Project' page on the SqueakSource website. The left sidebar has links for Home, Projects, Tags, Members, Groups, and Help. The main content area has a heading 'Actions' with a 'Back' link. Below that is a sidebar with links for Oscar Nierstrasz, Logout, Edit Account, and several projects like Coral, Enigma Machine, Fame for Squeak, JMethods, Nanola, Omni Dojo, PEG, SCGPier, SOI, SOI Davos Training Camp, SplitJoin, Squeak Examples, Squeak by Example, Tables in Squeak, and Traits Model. The main form area has a heading 'Register Project'. It contains the following fields:

- Name:** [Text input field]
- Title:** [Text input field]
- Description:** [Text area]
- License:** [Select dropdown] with options: seaside, server, teaching, testing, xml. The 'None' option is selected.
- Enable Blessings:** [Select dropdown] with options: no, yes. The 'no' option is selected.
- Global:** [Select dropdown] with options: Read, Write, Admin. The 'Read' option is selected.
- Send emails to:** [Text input field] with placeholder 'address:'
- Reply-To Address:** [Text input field] with value 'squeak-dev@lists.squeakfc.org'.
- Subscriptions:** [Checkboxes]
 - Recieve commit notifications by email
- Administrators:** [Text input field] with value '(add)' followed by a '(remove)' link.
- Developers:** [Text input field] with value '(add)'.
- Guests:** [Text input field] with value '(add)'.

At the bottom are 'Save' and 'Cancel' buttons.

Figure 15.24: Repositories under SqueakSource are highly configurable.

Monticello offers a large range of options (cf. Figure 15.24) to configure a project repository: tags may be assigned, a license may be chosen, access for people who are not part of the project may be restricted (read/write, read, no access), emails may be sent upon commits, mailing list may be managed, and users may be defined to be members of the project (as administrator, developer, or guest).

15.6 The .mcz file format

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

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

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

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

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

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

- *Snapshot*. A Snapshot is a record of the state of the package at a particular time. Each mcz file contains a directory named “snapshot/”. All the members in this directory contain definitions of program elements, which when combined form the Snapshot. Current versions of Monticello only create one member in this directory, called “source.st”.
- *Dependencies*. A version may depend on specific version of other packages. An mcz file may contain a “dependencies/” directory with a member for each dependency. These members will be named after each package the Monticello package depends upon. For example, a Pier-All mcz file will contain files named Pier-Blog and Pier-Caching in its dependencies directory.

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

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

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

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

15.7 Chapter Summary

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

- Monticello are mapped to Smalltalk categories and method protocols. If you add a package called Foo to Monticello, it will include all classes in categories called Foo or starting with Foo-. It will also include all methods in those categories, except those in protocols starting with *. Finally it will include all *class extension* methods in protocols called *foo or starting with *foo- anywhere else in the system.
- When you modify any methods or classes in a package, it will be marked as “dirty” in Monticello, and can be saved to a repository.
- There are many kinds of repositories, the most popular being HTTP repositories, such as those hosted by SqueakSource.

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

Chapter 16

Gofer: Scripting package loading

Pharo proposes powerful tools to manage source code such as semantics-based merging, tree diff-merge, and a git-like distributed versioning system. In particular as presented in the Monticello Chapter, Pharo uses a package system named Monticello. In this chapter after a first reminder of the key aspects of Monticello we will show how we can script package using Gofer. Gofer is a simple API for Monticello. It is used by Metacello, the language to manage package maps that we present in Chapter Metacello.

16.1 Preamble: Package management system

Packages. A package is a list of class and method definition. In Pharo a package is not associated with a namespace. A package can extend a class defined in another package: it means that a package, for example Network can add methods to the class String, even though String is not defined in the package Network. Class extensions support the definition of layers and allows for the natural definition of packages.

To define a package, you simply need to declare one using the Monticello browser and to define a class extensions, it is enough to define a method with a category starting with '*' followed by the package name (here '*network').

```
Object subclass: #ButtonsBar
instanceVariableNames: ""
classVariableNames: ""
poolDictionaries: ""
category: 'Zork'
```

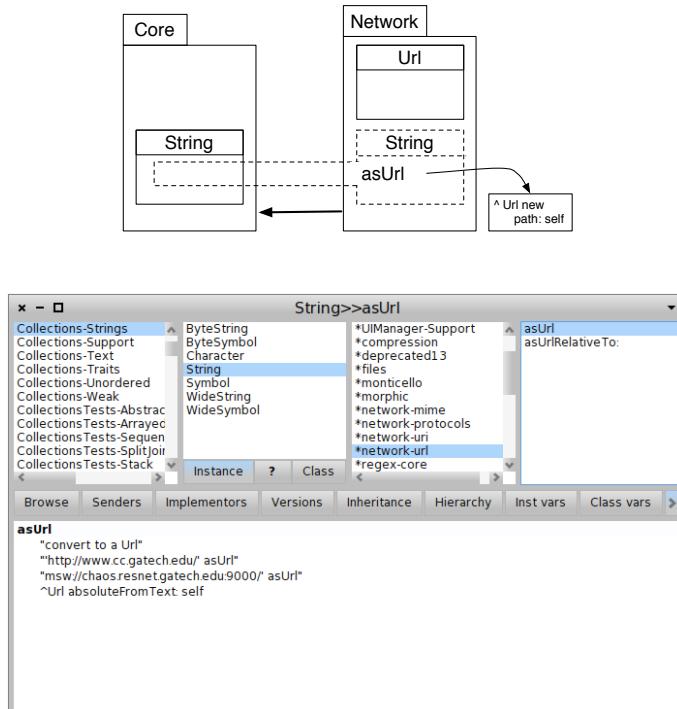


Figure 16.1: The browser shows that the class `String` gets the methods `asUrl` and `asUrlRelativeTo`: from the package `network-url`

We can get the list of changes of a package before publication by simply selecting the package and clicking on the `Changes` of the Monticello Browser.

Package Versioning System. A version management system helps for version storage and keeps an history of system evolution. Moreover, it provides the management of concurrent accesses to a source code repository. It keeps traces of all saved changes and allows us to collaborate with other engineers. More a project grows, more it is important to use a version management system.

Monticello defines the package system and version management of Pharo. In Pharo, classes and methods are elementary entities which are versioned by Monticello when actions were done (superclass change, instance variable changes, methods adding, changing, deleting ...). A source is an HTTP server which allows us to save projects (particularly packages) managed by Monticello. This is the equivalent of a forge: It provides the management of contributors and their status, visibility information, a wiki with RSS

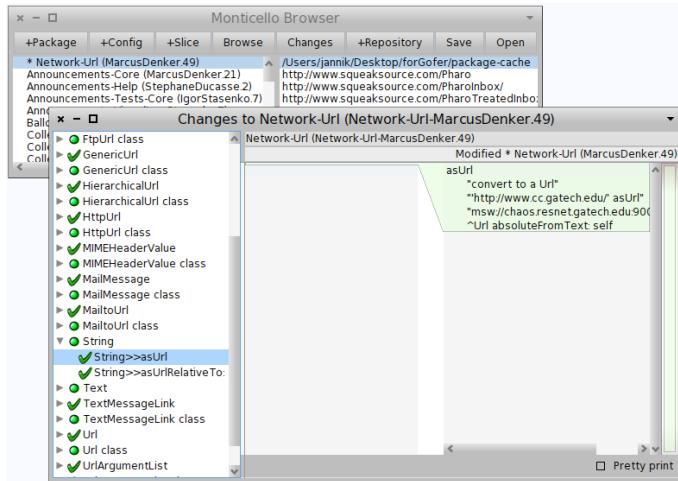


Figure 16.2: The change browser shows that the method `String>>asUrl` has changed.

feed. A source open to everybody is available at <http://www.squeaksource.com/>.

Distributed architecture of Monticello. Monticello is a distributed version control management system like git but dedicated to Smalltalk. Monticello manipulates source code entities such as classes, methods,... It is then possible to manage local and distributed code servers. Gofer allows one to script such servers to publish, download and synchronize servers.

Monticello uses a local cache for packages. Each time a package is required, it is first looked up in this local cache. In a similar way, when a package is saved, it is also saved in the local cache. From a physical point of a view a Monticello package is a zipped file containing meta-data and the complete source code of package. To be clear in the following we make the distinction between a package loaded in the pharo image and a package saved in the cache but not loaded. A package currently loaded is called a working copy of the package. We also define the following terms: image (object and bytecode executed by the virtual machine), loaded package (downloaded package from a server that is loaded in memory), dirty package (a loaded package with unsaved modifications). A dirty package is a loaded package.

For example, in Figure 16.3 the package a.1 is loaded from the server squeaksource. It is not modified. The package b.1 is loaded from the server yoursource.com but it is modified locally in the image. Once b.1 which was dirty is saved on the server yoursource.com, it is versioned into b.2 which is

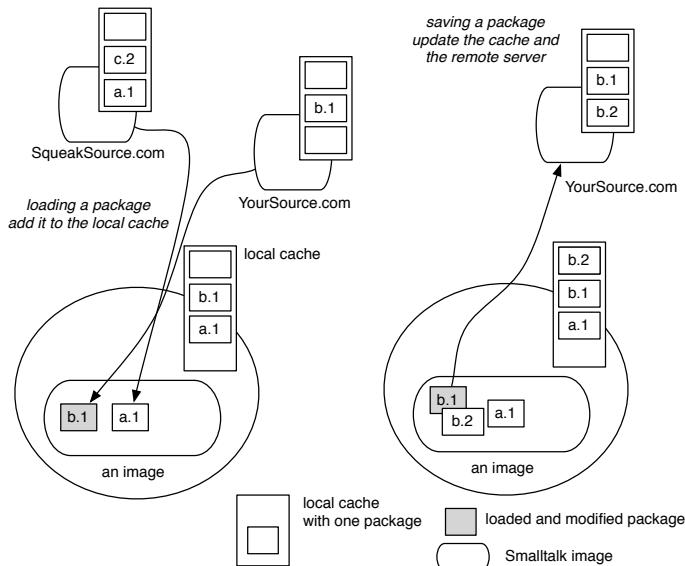


Figure 16.3: (left) Typical setup with clean and dirty packages loaded and cached — (right) Package published.

saved in the cache and the remote server.

16.2 What is Gofer?

Gofer is a scripting tool for Monticello. It is developed by Lukas Renggli and it is used by Metacello (the map and project management system built on top of Monticello). Gofer supports the easy creation of scripts to load, save, merge, update, fetch ... packages. In addition, Gofer makes sure that operations let the system in a clean state. Gofer allows one to load packages located in different repositories in a single operation, to load the latest stable version or the currently developed version. Gofer is part of basis of Pharo since Pharo 1.0. Metacello uses Gofer as its underlying infrastructure to load complex projects. Gofer is more adapted to load simple package.

You can ask Gofer to update it itself by executing the following expression:

```
Gofer gofer update
```

16.3 Using Gofer

Using Gofer is simple: you need to specify a location, the package to be loaded and finally the operation to be performed. The location often represents a file system been it an HTTP, a FTP or simply an hard-disc. The location is the same as the one used to access a Monticello repository. For example it is '<http://www.squeaksourc.com/MyPackage>' as used in the following expression.

```
MCHttpRepository
  location: 'http://www.squeaksourc.com/MyPackage'
  user: 'pharoUser'
  password: 'pharoPwd'
```

Here is a typical Gofer script: it says that we want to load the package PBE2GoferExample from the repository PBE2GoferExample that is available on <http://www.squeaksourc.com>.

```
Gofer new
  url: 'http://www.squeaksourc.com/PBE2GoferExample';
  package:'PBE2GoferExample';
  load
```

When the repository (HTTP or FTP) requires an identification, the message `url:username:password:` is available. The message `directory:` supports the access to local files.

```
Gofer new
  "we work on the project PBE2GoferExample"
  url: 'http://www.squeaksourc.com/PBE2GoferExample'
  "we specify a user name"
  username: 'pharoUser'
  "we specify a password"
  password: 'pharoPwd';
  "we define the package to be loaded"
  package:'PBE2GoferExample';
  "we disable the lookup of package in the local cache"
  disablePackageCache;
  "we stop the error raising"
  disableRepositoryErrors;
  "we load the specified package"
  load.
```

Since we often use the same public servers, there are some shortcuts in the API to use them. For example, `squeaksourc:` is a shortcut for `http://www.squeaksourc.com`, `wiresong:` for `http://source.wiresong.ca/` and `gemsource:` for `http://seaside.gemstone.com/ss/`. In such a case you do not need to specify the full path as shown with the next snippet:

```
Gofer new
squeaksource: 'PBE2GoferExample';
package: 'PBE2GoferExample';
load
```

In addition, when Gofer does not succeed to load a package in a specified URL, it looks in the local cache which is normally at the root of your image. It is possible to force Gofer not to use the cache using the message `disablePackageCache` or to use it using the message `enablePackageCache`.

In a similar manner, Gofer returns an error when one of the repositories is not reachable. We can instruct it to ignore such errors using the message `disableRepositoryErrors`. To enable it the message we can use the message `enableRepositoryErrors`.

Package Identification

Once an URL and the option are specified, we should define the packages we want to load. Using the message `version:` defines the exact version to load, while the message `package:` should be used to load the latest version available in all the repositories.

The following example load the version 2 of the package.

```
Gofer new
squeaksource: 'PBE2GoferExample';
version: 'PBE2GoferExample-janniklaval.1';
load
```

We can also specify some constraints to identify packages using the message `package: aString constraint: aBlock` to pass a block.

For example the following code will load the latest version of the package saved by the developer named janniklaval.

```
Gofer new
squeaksource: 'PBE2GoferExample';
package: 'PBE2GoferExample' constraint: [ :version | version author = 'janniklaval' ];
load
```

16.4 Gofer Actions

Loading several packages

We can load several packages from different servers. To show you a concrete example, you have to load first the configuration of Seaside using its

Metacello configuration.

```
Gofer new
"we will load the latest version of the configuration of Seaside "
squeaksources: 'MetacelloRepository';
package: 'ConfigurationOfSeaside30';
load.
```

"Now to load seaside you need to ask its configuration."
(Smalltalk at: #ConfigurationOfSeaside30) load.

Pay attention that the last expression will load the complete seaside application, *i.e.*, around 70 packages, so it can take a moment.

The following code snippet loads multiple packages from different servers. The loading order is respected: the script loads first Pier–All, then Picasa–Model, and finally Picasa–Seaside.

```
Gofer new
renggli: 'pier';
package: 'Pier–All';
squeaksources: 'PicasaClient';
package: 'Picasa–Model';
package: 'Picasa–Seaside';
load.
```

This example may give the impression that Pier–All is looked up in the Pier repository of the server renggli and that Picasa–Model and Picasa–Seaside are looked up in the project PicasaClient of the squeaksources server. However this is not the case, Gofer does not take into account this order. In absence of version number, Gofer loads the most recent package versions found looking in the two servers.

We can then rewrite the script in the following way:

```
Gofer new
squeaksources: 'PicasaClient';
renggli: 'pier';
package: 'Pier–All';
package: 'Picasa–Model';
package: 'Picasa–Seaside';
load.
```

When we want to specify that package should be loaded from a specific server, we should write multiple scripts.

```
Gofer new
squeaksources: 'PicasaClient';
package: 'Picasa–Model';
package: 'Picasa–Seaside';
```

```

load.
Gofer new
squeaksource: 'PicasaClientLightbox';
package: 'Picasa-Model';
package: 'Picasa-Seaside';
load.

```

Note that such scripts load the latest versions of the packages, therefore they are fragile since if a new package version is published, you will load it even if this is inappropriate. In general it is a good practice to control the version of the external components we rely on and use the latest version for our own current development. Now such problem can be solved with Metacello which is the tool to express configurations and load them.

Other protocols

Gofer supports also FTP as well as loading from a local directory. We basically use the same messages than before with some changes.

For FTP, we should specify the URL using 'ftp' as heading

```

Gofer new
url: 'ftp://wtf-is-ftp.com/code';
...

```

To work on a local directory, the message `directory:` followed by the absolute path of the directory should be used. Here we specify that the directory to use is reachable at `/home/pharoer/hacking/MCPackages`

```

Gofer new
directory: '/home/pharoer/hacking/MCPackages';

```

Finally it is possible to look for packages in a repository and all its sub-folders using the keen star.

```

Gofer new
directory: '/home/pharoer/hacking/MCPackages/*';
...

```

Once a Gofer instance is parametrized, we can send it messages to perform different actions. Here is a list of the possible actions. Some of them are described later.

load	Load the specified packages.
update	Update the package loaded versions.
merge	Merge the distant version with the one currently loaded.
localChanges	Show the list of changes between the bases version and the version currently modified.
remoteChanges	Show the changes between the version currently modified and the version published on a server.
cleanup	Cleanup packages: System obsolete information is cleaned.
commit / commit:	Save the packages on a distant server – with a message log.
revert	Reload previously loaded packages.
recompile	recompile packages
unload	Unload from the image the packages
fetch	Download the remote package versions from a remote server to the local cache.
push	Upload the versions from the local cache to the remote server.

Working with remote servers

Since Monticello is a distributed versioning control system, it is often useful to synchronize versions published on a remote server with the ones locally published in the MC local cache. Here we show the main operations to support such tasks.

The merge, update and revert operations. The message merge performs a merge between a remote version and the working copy (the one currently loaded). Changes present in the working copy are merged with the code of the remote one. It is often the case that after a merge, the working copy gets dirty and should be republished. The new version will contain the current changes and the changes of the remote version. In case of conflicts the user will be warned, else the operation will happen silently.

```
Gofer new
squeaksource: 'PBE2GoferExample';
package: 'PBE2GoferExample';
merge
```

The message update loads the remote version in the image. The modifications of the working copy are lost.

The message revert resets the local version, *i.e.*, it loads again the current version. The changes of the working copy are then lost.

The commit and commit: operations. Once we have merged or changed a package we want to save it. For this we can use the messages `commit` and `commit:`. The second one is expecting a comment - this is in general a good practice.

```
Gofer new
```

```
"We save the package in the repository"
squeaksource: 'PBE2GoferExample';
package: 'PBE2GoferExample';
"We comments the changes and save"
commit: 'I try to use the message commit: '
```

The localChanges and remoteChanges operations. Before loading or saving a version, it is often useful to verify the changes made locally or on the server. The message `localChanges` shows the changes between the last loaded version and the working copy. The `remoteChanges` shows the differences between the working copy and the last published version on the server. Both return a list of changes.

```
Gofer new
```

```
squeaksource: 'PBE2GoferExample';
package: 'PBE2GoferExample';
"We check that we will publish only our changes by comparing local changes versus
the packages published on the server"
localChanges
```

Using the messages `browseLocalChanges` and `browseRemoteChanges`, it is possible to browse the changes using a normal code browser.

```
Gofer new
```

```
squeaksource: 'PBE2GoferExample';
"on ajoute la derni re version de PBE2GoferExample"
package: 'PBE2GoferExample';
"on navigue dans les changements effectu s sur le serveur"
browseRemoteChanges
```

The unload operation. The message `unload` unloads from the image the packages. Note that using the Monticello browser you can delete a package but such operation does not remove the code of the classes associated with the package, it just destroys the package. Unloading a package destroys the packages and the classes it contains.

The following code unloads the packages and its classes from the current image.

```
Gofer new
```

```
squeaksource: 'PBE2GoferExample';
package: 'PBE2GoferExample';
unload
```

Note that you cannot unload Gofer itself that way. Gofer gofer unload does not work.

The fetch and push operations. Since Monticello is a distributed versioning system, it is good to save locally all the versions you want, without being forced to published on a remote server - this is especially true when working off-line. Now this is tedious to synchronize all the local and remote published packages. The messages `fetch` and `push` are there to support you in this task.

The message `fetch` copies from the remote server the packages that are missing in your local server. The packages are not loaded in Pharo. After a `fetch` you can load the packages even if the remote server breaks down.

```
Gofer new
squeaksource: 'PBE2GoferExample';
package: 'PBE2GoferExample';
fetch
```

Now if you want load your packages locally remember to set up that the lookup should consider local cache and disable errors as presented in the beginning of this chapter (messages `disableRepositoryErrors` and `enablePackageCache`).

The message `push` performs the inverse operation. It published to the remote server the packages locally available. All the packages that you published locally are then pushed to the server.

```
Gofer new
squeaksource: 'PBE2GoferExample';
package: 'PBE2GoferExample';
push
```

As a pattern, we always keep in our local cache the copies of all the versions of our projects or the projects we used. This way we are autonomous from any network failure and the packages are backed up in our regular backup.

With these two messages, it is easy to write a script sync that synchronize local and remote repositories.

```
Gofer new
squeaksource: 'PBE2GoferExample';
package: 'PBE2GoferExample';
push.
```

```
Gofer new
squeaksources: 'PBE2GoferExample';
package: 'PBE2GoferExample';
fetch
```

Automating Answers

Sometimes package installation asks for information such as passwords. With the systematic use of a build server, packages will probably stop to do that, but this is important to know how to supply answers from within a script to these questions. The message `valueSupplyingAnswers:` supports such a task.

```
[ Gofer new
  squeaksources: 'Seaside30';
  package: 'LoadOrderTests';
  load ]
  valueSupplyingAnswers: {
    {'Load Seaside'. True}.
    {'SqueakSource User Name'. 'pharoUser'}.
    {'SqueakSource Password'. 'pharoPwd'}.
    {'Run tests'. false}.
}
```

This message should be sent to a block giving a list of questions and their answers as shown by the previous examples

16.5 Conclusion

Gofer provides a robust and stable implementation to script the management of your packages. Now when you project grows you should really consider to use Metacello (see Chapter 17).

Chapter 17

Managing projects with Metacello

with the participation of:

Dale Henrichs (dale.henrichs@gemstone.com)

Mariano Martinez Peck (marianopeck@gmail.com)

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 configuration* 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.

17.1 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.

17.2 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 clearly as possible. Gofer is a scripting API to Monticello (See Chapter 16).

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.

17.3 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.

17.4 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:.
- 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.

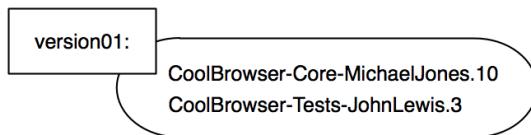


Figure 17.1: Simple version.

- 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 user-name 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 `version02:`

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

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

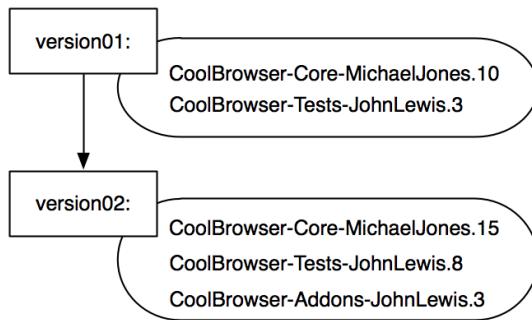


Figure 17.2: A second simple version.

17.5 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.squeaksources.com/Pharo10MetacelloRepository>, <http://www.squeaksources.com/Pharo11MetacelloRepository> were created. They contain the configurations of several tools and applications and serve as a central repository.

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. A process for publishing configurations in specific distribution repositories is under definition at the time of writing.

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' }.
```

Note that if you want to load simulate the loading of a configuration but not load it, you should use record: instead of load:

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

But instead of doing a "do it", do a "print it", send it the message loadDirective.

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

loadDirective.

17.6 Managing package 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

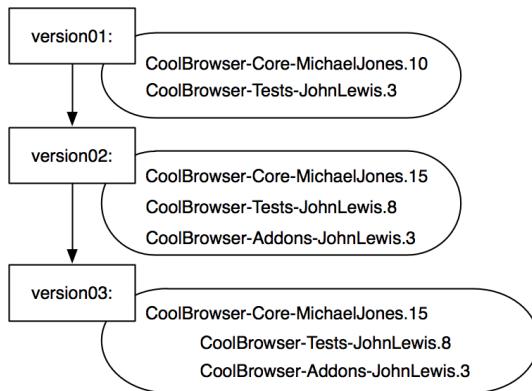


Figure 17.3: A version expressing requirements between packages.

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 17.3):

```

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 `CoolBrowser` 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 pack-

ages 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*.

17.7 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.

```
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' ] ].
```

Baseline `baseline04:` will be used across several versions as for example the version '`0.4`' defined below (see Figure 17.4). 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';
```

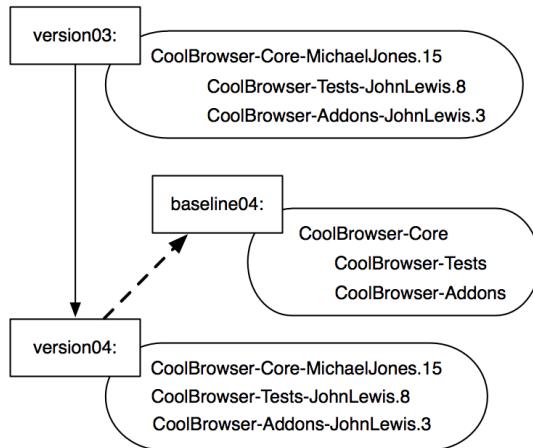


Figure 17.4: A version now imports a baseline that expresses dependencies between packages.

```

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 baseline version so that you get the latest work from all of the project members.

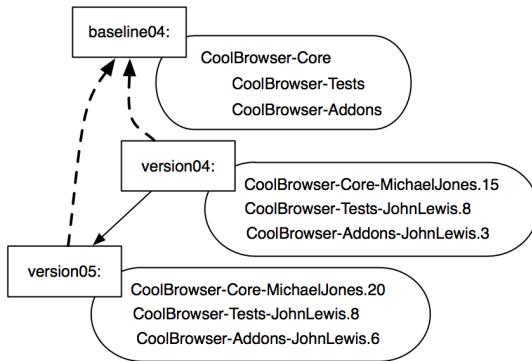


Figure 17.5: A second version imports again the baseline.

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 17.5).

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.

17.8 Groups

Suppose that now the CoolBrowser project is getting better and someone wrote tests for the addons. We have a new package named 'CoolBrowser-AddonsTests'. It depends on 'CoolBrowser-Addons' and 'CoolBrowser-Tests' as shown by Figure 17.6.

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

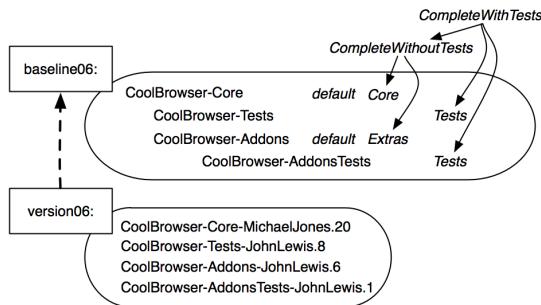


Figure 17.6: A baseline with groups: default, Core, Extras, Tests, CompleteWithoutTests and CompleteWithTests.

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, 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 17.6).

To define a group we use the method `group: groupName with: group elements`. The `with:` argument can be a package name, a project, another group, or 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'.

    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 a group is a structural component. The version 0.6 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.
```

Finally 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'.
```

17.9 Project Configuration Dependencies

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:

```
``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

Now let us focus on the case where projects are described using Metacello configuration. Let us 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 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 the recommended ConfigurationOfCoolBrowser. This will be better to understand each parameter.

The version is just a normal version. It imports a baseline.

```
ConfigurationOfCoolToolSet>>version01: spec
<version: '0.1' imports: #('0.1-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 17.7).

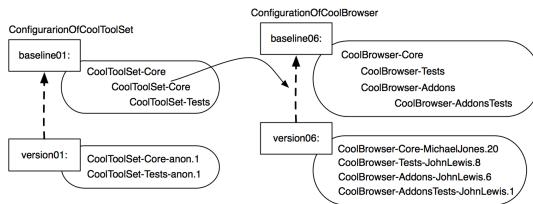


Figure 17.7: Dependencies between configurations.

- The message `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 message `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 could be the same and be for example '`ConfigurationOfCoolBrowser`'. Here since we mentioned that the project does not follow the convention we have to specify all the information.

Finally, the `versionString:` and `loads:` message specify 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.

We've named the project reference '`CoolBrowser ALL`' and in the specification for the '`CoolToolSet-Core`' package, we've specified that '`CoolBrowser ALL`' is required. The name of the project reference is arbitrary, you can select the name you want, although it's recommended to put a name that makes sense to that project reference.

Now we can now download `CoolToolSet` like this:

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

Note that the entire `CoolBrowser` project is loaded before '`CoolToolSet-Core`'. To separate the test package from the core packages, 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` you can see that we used `versionString: '1.0'`. if the project changes often and you want to follow the changes, you may be forced to update often your baseline and this is not really adequate. Depending of your context you can specify the `#versionString:` in the version method instead of in the baseline 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 at all for the references '`CoolBrowser default`' and '`CoolBrowser Tests`' in the version method, then the version specified in the baseline is used. If there is no version specified in the baseline method, then Metacello loads the latest version of the project.

17.10 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 `preLoadDoIt:` and `postLoadDoIt:`. 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
    package: 'CoolBrowser-Core' with: [
        spec
        file: 'CoolBrowser-Core-MichaelJones.20';
        preLoadDolt: #preloadForCore;
        postLoadDolt: #postloadForCore:package: ];
    ...
    package: 'CoolBrowser-AddonsTests' with: 'CoolBrowser-AddonsTests-
JohnLewis.1'].
```

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

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

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

As you can notice there, both methods, **preLoadDolt**: and **postLoadDolt**: receive a selector that will be performed before or after the load. You can also note that the method **postloadForCore:package:** takes two parameters. The *pre/post* load methods may take 0, 1 or 2 arguments. The *loader* is the first optional argument and the loaded *packageSpec* is the second optional argument. Depending on your needs you can choose which of those arguments do you want.

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

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

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

```

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

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

    spec
        package: 'CoolBrowser-Core' with: [
            spec
                file: 'CoolBrowser-Core-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.

17.11 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 to do that, we need to give Metacello platform specific information using the method for:do: as shown in the following example.

```

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

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

spec for: #gemstone do: [
    spec package: 'CoolBrowser-Platform' with: 'CoolBrowser-PlatformGemstone-
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 uniform way. Thus, we have a lot of flexibility. At runtime, when you load CoolBrowser, Metacello automatically detects in which dialect the load is happening and loads the specific package for that dialect.

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.

17.12 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 `symbolicVersion:` 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 expressions such as `(ConfigurationOfOmniBrowser project version: #'stable')` load to a project reference in a baseline version:

```
baseline10: spec
<version: '1.0-baseline'>
spec for: #squeakCommon do: [
    spec blessing: #baseline.
    spec repository: 'http://seaside.gemstone.com/ss/GLASSClient'.
    spec
        project: 'OmniBrowser' with: [
            spec
                className: 'OmniBrowser';
                versionString: #'stable';
                repository: 'http://www.squeaksource.com/MetacelloRepository' ].
    spec
        package: 'OB-SUnitGUI' with: [
            spec requires: #( 'OmniBrowser' )];
        package: 'GemTools-Client' with: [
            spec requires: #( 'OB-SUnitGUI' )];
        package: 'GemTools-Platform' with: [
            spec requires: #( 'GemTools-Client' )]].
```

Project Blessing and Loading

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.

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 to use under development (i.e., whose blessing is development). Typically a development version is used by developers for managing pre-release activities as the project transitions from bleedingEdge to stable. There are a number of MetacelloToolBox methods that take advantage of the development symbolic version.

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.

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

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

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

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

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

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

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

The following is the definition for the `bleedingEdge` symbolic version:

```
bleedingEdge
<defaultSymbolicVersion: #bleedingEdge>

| bleedingEdgeVersion |
bleedingEdgeVersion := (self project map values select: [ :version |
version blessing == #baseline ]) detectMax: [ :version | version ].
bleedingEdgeVersion ifNil: [ bleedingEdgeVersion := self project latestVersion ].  

bleedingEdgeVersion
ifNil: [ self versionDoesNotExistError: #bleedingEdge ].  

↑ bleedingEdgeVersion versionString
```

Hints.

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

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

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

Loading Pharo 1.2.2–baseline would cause the `\ct{#stable}` version for each of those projects to be loaded ...

but remember over time the `\ct{#stable}` version will change and incompatibilities between packages can creep in.

By using `\ct{#stable}` versions you will be in better shape than using `\ct{#bleedingEdge}` because the `\ct{#stable}` version is known to work.

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

```
\begin{code}{}}
spec
project: 'OB Dev' with: '1.2.4';
project: 'ScriptManager' with: '1.2';
project: 'Shout' with: '1.2.2';
....].
\end{code}
```

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

#stable} versions may have drifted over time.

If you are just bringing up a PharoCore1.2 image and would like to load the Pharo dev code, you should load the \ct{#stable} version of Pharo (which may be 1.2.2 today and 1.2.3 tomorrow).

If you want to duplicate the environment that someone is working in, you will ask them for the version of Pharo and load that explicit version to reproduce the bug or whatever

\section{Script and Tool Support}

Metacello comes with an API to make the writing of tools for Metacello easier. Two classes exist: \ct{MetacelloBaseConfiguration} and \ct{MetacelloToolBox}.

\subsection{Development Support}

The \ct{MetacelloBaseConfiguration} class is aimed at eventually becoming the common superclass for all Metacello configurations. For now, though, the class serves as the location for defining the common default symbolic versions (\ct{bleedingEdge} at the present time) and as the place to find development support methods such as the following ones:

\begin{description}

- \item \ct{compareVersions}: Compare the \ct{#stable} version to \ct{#development} version.
 - \item \ct{createNewBaselineVersion}: Create a new baseline version based upon the \ct{#stable} version as a model.
 - \item \ct{createNewDevelopmentVersion}: Create a new \ct{#development} version using the \ct{#stable} version as model.
 - \item \ct{releaseDevelopmentVersion}: Release \ct{#development} version: set version blessing to \ct{#release}, update the \ct{#development} and \ct{#stable} symbolic version methods and save the configuration.
 - \item \ct{saveConfiguration}: Save the mcz file that contains the configuration to it's repository.
 - \item \ct{saveModifiedPackagesAndConfiguration}: Save modified mcz files, update the \ct{#development} version and then save the configuration.
 - \item \ct{updateToLatestPackageVersions}: Update the \ct{#development} version to match currently loaded mcz files.
 - \item \ct{validate} Check the configuration for Errors, Critical Warnings, and Warnings.
- \end{description}

\subsection{Metacello Toolbox API}

The \ct{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.

For an overview of the Metacello Toolbox API, you can look in the HelpBrowser at the 'Metacello>>API Documentation' section. The instance-side methods for MetacelloToolBox support the 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'.

\section{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:

\subsection{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:

```
\begin{code}{}}
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). GemTools itself is made up of 5 mcz files from the <http://seaside.gemstone.com/ss/GLASSClient> repository and de-

pends upon 4 other projects: FFI, OmniBrowser, Shout and HelpSystem.

- OB-SUnitGUI: requires 'OmniBrowser'.
- GemTools-Client: requires 'OmniBrowser', 'FFI', 'Shout', and 'OB-SUnitGUI'.
- GemTools-Platform: requires 'GemTools-Client'.
- GemTools-Help: requires 'HelpSystem' and 'GemTools-Client'.

Project Startup

Create Configuration and Initial Baseline

Here we 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:...` message copies the class `MetacelloConfigTemplate` to `ConfigurationOfGemToolsExample` and creates a `#baseline10:` method that looks like the following:

```
ConfigurationOfGemToolsExample>>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 `#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.

Create Initial Literal Version

Now we use the toolbox API to create the initial literal version of the project (by literal we mean with numbers identifying the package versions). The toolbox method `createDevelopment:...` 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:

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

Note how 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:

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

Validation

Whenever you finish editing a configuration you should validate it to check for mistakes that may cause problems later on. The Metacello ToolBox provides the validation via the message `validateConfiguration:.` The following expression show you possible errors: (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 message: (MetacelloToolBox descriptionForValidationReasonCode: #noVersionSpecified) inspect.

The fourth line lists the configurationClass, *i.e.*, 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).

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'.
```

Development Cycle

Now let us look at a typical iteration: testing, releasing, and saving the configuration.

Platform Testing

To finish the validation of your configuration, you need to do some test loads on your intended platforms. For GemTools we can 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.
```

Now this is the moment to run the unit tests. Note that for the GemTools unit tests you need to have GemStone installed.

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 version to #release.
- sets the #development version to #notDefined.
- sets the #stable 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:

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

The `stable:` method ends up looking like this:

```
ConfigurationOfGemToolsExample>>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'.
```

Open New Version for Development

Now we are ready to start new development. The method `createNewDevelopmentVersionIn:...` performs the necessary modification to be in a state that reflects it.

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

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'.
```

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 we just left it out of the previous example). To add a new package a project you need to:

- (1) create a new baseline version to reflect the new package and dependencies,
- (2) update existing #development version to reference the new baseline version, and
- (3) 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.
```

17.13 Load types

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

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

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

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

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

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

A possible problem with linear mode is the following: suppose project *A* depends has dependencies on other two projects *B* and *C*. *B* depends on the project *D* version 1.1 and *C* depends on project *D* version 1.2 (the same project but another version). First question, which *D* version does *A* have at the end? By default (you can change this using the method operator: in the project method), Metacello will finally load version 1.2.

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

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

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

```
ConfigurationOfCoolToolSet >>project
```

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

17.14 Conditional loading

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

Suppose that our CoolToolSet starts to provide much more features. We first split the core in two packages: 'CoolToolSet-Core' and 'CoolToolSet-CB'. CoolBrowser can be present in one image but not in another one. We want to load the package 'CoolToolSet-CB' by default only and if CoolBrowser is present.

The mentioned conditionals are achieved in Metacello by using the *project attributes* we saw in the previous section. They are defined in the *project* method. Example:

```
ConfigurationOfCoolBrowser >>project
```

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

```

project := constructor project.
projectAttributes := ((Smalltalk at: #CBNode ifAbsent: []) == nil
    ifTrue: [ #( #'CBNotPresent' ) ]
    ifFalse: [ #( #'CBPresent' ) ]).
project projectAttributes: projectAttributes.
project loadType: #linear.
project ]

```

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

```

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

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

```

You can notice that the way to use project attributes is through the existing method `for:do:`. Inside that method you can do whatever you want: de-

fine groups, dependencies, etc. In our case, if CoolBrowser is present, then we just add 'CoolToolSet-CB' to the default group. If it is not present, then 'CoolBrowser default' is added to dependency to 'CoolToolSet-CB'. In this case, we do not add it to the default group because we do not want that. If desired, the user should explicitly load that package also.

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

17.15 Project version attributes

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

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

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

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.
```

17.16 Conclusion

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

Part VI

Tools

Chapter 18

Optimizing Application

Since the beginning of software engineering, programmers have faced issues related to application performance. Although there has been a great improvement on the programming environment to support better and faster development process, addressing performance issues when programming still requires quite some dexterity.

In principle, optimizing an application is not particularly difficult. The general idea is to make slow and frequently called methods either faster or less frequently called. Note that optimizing an application usually complexifies the application. It is therefore recommended to optimize an application only when the requirements for it are well understood and addressed. In other term, you should optimize your application only when you are sure of what it is supposed to do. As Kent Beck famously formulated: *1 - Make It Work, 2 - Make It Right, 3 - Make It Fast.*

18.1 What does profiling mean?

Profiling an application is a term commonly employed that refers to obtaining dynamic information from a controlled program execution. The obtained information is intended to provide important hints on how to improve the program execution. These hints are usually numerical measurements, easily comparable from one program execution to another.

In this chapter, we will consider measurement related to method execution time and memory consumption. Note that other kind of information may be extracted from a program execution, in particular the method call graph.

It is interesting to observe that a program execution usually follows the universal 80-20 rule: only a few amount of the total amount of methods (let's

say 20%) consume the largest part of the available resources (80% of memory and CPU consumption). Optimizing an application is essentially a matter of tradeoff therefore. In this chapter we will see how to use the available tools to quickly identify these 20% of methods and how to measure the progress coming along the program enhancements we bring.

Experience shows that having unit tests is essential to ensure that we do not break the program semantics when optimizing it. When replacing an algorithm by another, we ought to make sure that the program still do what it is supposed to do.

18.2 A simple example

Consider the method `Collection>>select:thenCollect:..`. For a given collection, this method selects elements using a predicate. It then applies a block function on each selected element. At the first sight, this behavior implies two runs over the collections: the one provided by the user of `select:thenCollect:` then an intermediate one that contains the selected elements. However, this intermediate collection is not indispensable, since the selection and the function application can be performed with only one run.

The method `timeToRun`. Profiling one program execution is usually not enough to fully identify and understand what has to be optimized. Comparing at least two different profiled executions is definitely more fruitful. The message `timeToRun` may be sent to a bloc to obtain the time in milliseconds that it took to evaluate the block. To have a meaningful and representative measurement, we need to “amplify” the profiling with a loop.

Here are some results:

```
| coll |
coll := #(1 2 3 4 5 6 7 8) asOrderedCollection.
[ 100000 timesRepeat: [ (coll select: [:each | each > 5]) collect: [:i | i * i]]] timeToRun
"Calling select:, then collect: - → ~ 570 – 600 ms"

| coll |
coll := #(1 2 3 4 5 6 7 8) asOrderedCollection.
[ 100000 timesRepeat: [ coll select: [:each | each > 5] thenCollect:[:i | i * i]]] timeToRun
"Calling select:thenCollect: - → ~ 397 – 415 ms"
```

Although the difference between these two executions is only about few hundred of milliseconds, opting for one method instead of the other could significantly slow your application!

Let's scrutinize the definition of `select:thenCollect:..`. A naive and non-optimized implementation is found in `Collection`. (Remember that `Collection`

is the root class of the Pharo collection library). A more efficient implementation is defined in `OrderedCollection`, which takes into account the structure of an ordered collection to efficiently perform this operation.

```
Collection>>select: selectBlock thenCollect: collectBlock  
    "Utility method to improve readability."
```

```
    ↑ (self select: selectBlock) collect: collectBlock
```

```
OrderedCollection>>select: selectBlock thenCollect: collectBlock  
    " Utility method to improve readability.  
    Do not create the intermediate collection. "
```

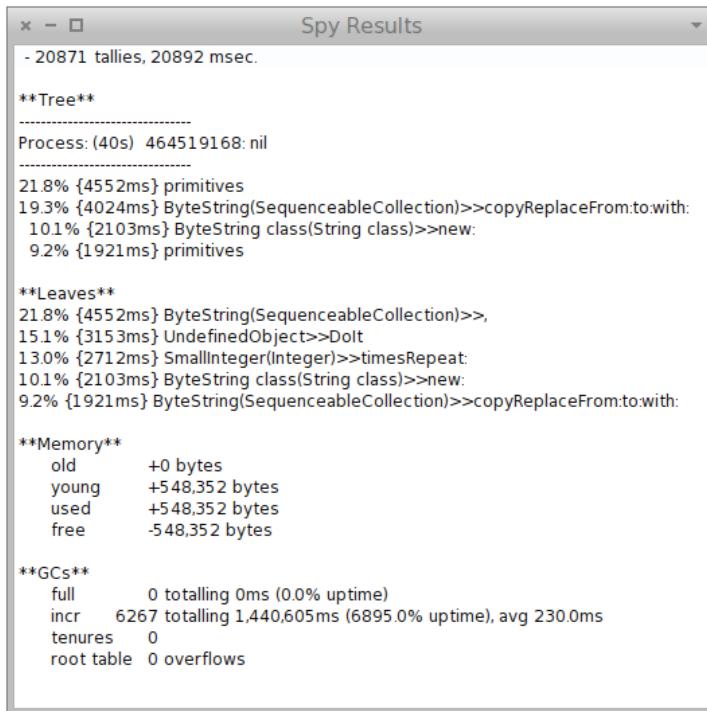
```
| newCollection |  
newCollection := self copyEmpty.  
firstIndex to: lastIndex do: [:index |  
    | element |  
    element := array at: index.  
    (selectBlock value: element)  
        ifTrue: [ newCollection addLast: (collectBlock value: element) ]].  
↑ newCollection
```

As you have probably guessed already, other collections such as `Set` and `Dictionary` do not benefit from an optimized version. We leave as an exercise an efficient implementation for other abstract data types. As part of the community effort, do not forget to submit your contribution to Pharo if you come up with an optimized and better version of `select:thenCollect:` or other methods. The Pharo team really value such effort.

The method `bench`. When sent to a block, the `bench` message estimates how many times this block is evaluated per second. For example, the expression `[1000 factorial] bench` says that `1000 factorial` may be executed approximately 350 times per second.

18.3 Code profiling in Pharo

The `timeToRun` method is useful to tell how long an expression takes to be executed. But it is not really adequate to understand how the execution time is distributed over the computation triggered by evaluating the expression. Pharo comes with `MessageTally`, a code profiler to precisely analyze the time distribution over a computation.



```

x - □ Spy Results
- 20871 tallies, 20892 msec.

**Tree**
-----
Process: (40s) 464519168: nil
-----
21.8% {4552ms} primitives
19.3% {4024ms} ByteString(SequenceableCollection)>>copyReplaceFrom:to:with:
10.1% {2103ms} ByteString class(String class)>>new:
9.2% {1921ms} primitives

**Leaves**
21.8% {4552ms} ByteString(SequenceableCollection)>>
15.1% {3153ms} UndefinedObject>>DoIt
13.0% {2712ms} SmallInteger(Integer)>>timesRepeat:
10.1% {2103ms} ByteString class(String class)>>new:
9.2% {1921ms} ByteString(SequenceableCollection)>>copyReplaceFrom:to:with:

**Memory**
old      +0 bytes
young    +548,352 bytes
used     +548,352 bytes
free     -548,352 bytes

**GCs**
full      0 totalling 0ms (0.0% uptime)
incr     6267 totalling 1,440,605ms (6895.0% uptime), avg 230.0ms
tenures   0
root table 0 overflows

```

Figure 18.1: MessageTally in action.

MessageTally

MessageTally is implemented as a unique class having the same name. Using it is quite simple. A message spyOn: needs to be sent to MessageTally with a block expression as argument to obtain a detailed execution analysis. Evaluating MessageTally spyOn: [*"your expression here"*] opens a window that contains the following information:

1. a hierarchy list showing the methods executed with their associated execution time during the expression execution.
2. leaf methods of the execution. A leaf method is a method that does not invoke other methods (*e.g.*, primitive, accessors).
3. statistic about the memory consumption and garbage collector involvement.

Each of these points will be described later on.

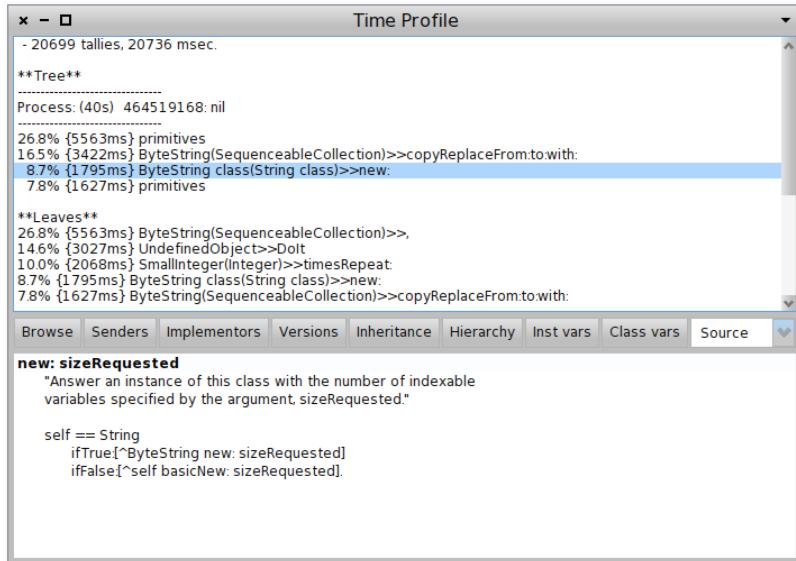


Figure 18.2: TimeProfiler uses MessageTally and navigates in executed methods.

Figure 18.1 shows the result of the expression `MessageTally spyOn: [20 timesRepeat: [Transcript show: 100 factorial printString]]`. The message `spyOn:` executes the provided block in a new process. The analysis focuses on one process, only, the one that executes the block to profile. The message `spyAllOn:` profiles all the processes that are active during the execution. This is useful to analyze the distribution of the computation over several processes.

A tool a bit less crude than `MessageTally` is `TimeProfileBrowser`. It shows the implementation of the executed method in addition (Figure 18.2). `TimeProfileBrowser` understand the message `spyOn:`. It means that in the below source code, `MessageTally` can be replaced with `TimeProfileBrowser` to obtain the better user interface.

Integration in the programming environment

As shown previously, the profiler may be directly invoked by sending `spyOn:` and `spyAllOn:` to the `MessageTally` class. It may be accessed through a number of additional ways.

Via the World menu. The World menu (obtained by clicking outside any Pharo window) offers some profiling facilities under the System submenu

(Figure 18.3). Start profiling all Processes creates a block from a text selection and invokes `spyAllOn:`. The entry Start profiling UI profiles the user interface process. This is quite handy when debugging a user interface!

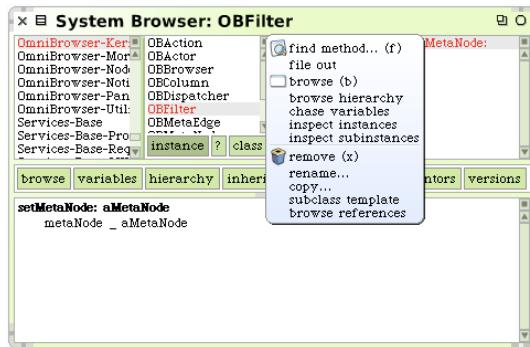


Figure 18.3: Access by the menu.

Via the Test Runner. As the size of an application grows, unit tests are usually becoming a good candidate for code profiling. Running tests often is rather tedious when the time to run them is getting too long. The Test Runner in Pharo offers a button Run Profiled (Figure 18.4).

Pressing this button runs the selected unit tests and generates a message tally report.

18.4 Read and interpret the results

The message tally profiler essentially provides two kinds of information:

- execution time is represented using a tree representing the profiled code execution (**Tree**). Each node of this tree is annotated with the time spent in each leaf method (**Leaves**).
- memory activity contains the memory consumption (**Memory** and the garbage collector usage (**GC**)).

For illustration purpose, let us consider the following scenario: the string character 'A' is cumulatively appended 9 000 times to an initial empty string.

```
MessageTally spyOn:
[ 500 timesRepeat: [
  | str |
```

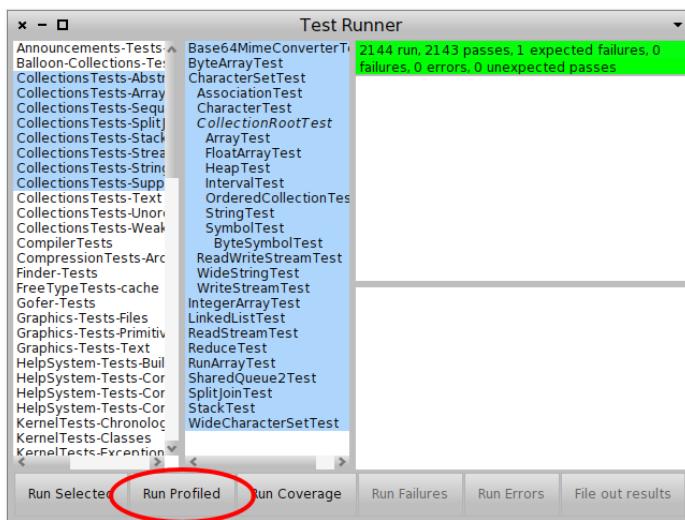


Figure 18.4: Button to generate a message Tally in the TestRunner.

```
str := ".
9000 timesRepeat: [ str := str, 'A' ]]].
```

The complete result is:

– 24038 tallies, 24081 msec.

Tree

Process: (40s) 535298048: nil

```
29.7% {7152ms} primitives
11.5% {2759ms} ByteString(SequenceableCollection)>>copyReplaceFrom:to:with:
5.9% {1410ms} primitives
5.6% {1349ms} ByteString class(String class)>>new:
```

Leaves

```
29.7% {7152ms} ByteString(SequenceableCollection)>>,
9.2% {2226ms} SmallInteger(Integer)>>timesRepeat:
5.9% {1410ms} ByteString(SequenceableCollection)>>copyReplaceFrom:to:with:
5.6% {1349ms} ByteString class(String class)>>new:
4.4% {1052ms} UndefinedObject>>Dolt
```

Memory

old	+0 bytes
young	+9,536 bytes

```

used      +9,536 bytes
free      -9,536 bytes

**GCs**
full      0 totalling 0ms (0.0% uptime)
incr     13875 totalling 3,122,594ms (12967.0% uptime), avg 225.0ms
tenures   0
root table 0 overflows

```

The first line gives the overall execution time and the number of samplings (also called *tallies*, we will come back on sampling at the end of the chapter).

****Tree**: Cumulative information**

The ****Tree**** section represents the execution tree per processes. The tree tells the time the Pharo interpreter has spent in each method. It also tells the different invocation using a call graph. Different execution flows are kept separated according to the process in which they have been executed. The process priority is also displayed, this helps distinguishing between different processes. The example tells:

```

**Tree**
-----
Process: (40s) 535298048: nil
-----
29.7% {7152ms} primitives
11.5% {2759ms} ByteString(SequenceableCollection)>>copyReplaceFrom:to:with:
5.9% {1410ms} primitives
5.6% {1349ms} ByteString class(String class)>>new:

```

This tree shows that 11.5% of the total execution time is spent in the method `SequenceableCollection>>copyReplaceFrom:to:with:`. This method is called when concatenating character strings using the message comma (,), itself indirectly invoking `new:` and some virtual machine primitives.

The execution takes 11.5% of the execution time, this means that the interpreter effort is shared with other processes. The invocation chain from the code to the primitives is relatively short. Reaching hundreds of nested calls is no exception for most of applications. We will optimize this example later on.

****Leaves**: leaf methods**

The **** Leaves**** part lists *leaf methods* for the code block that has been profiled. A leaf method is a method that does not call other methods. More exactly,

it is a method `m` for which no method invoked by `m` have been “detected”. This is the case for variable accessors (e.g., `Point»x`), primitive methods and methods that are very quickly executed. For the previous example, we have:

```
**Leaves**
29.7% {7152ms} ByteString(SequenceableCollection)>>,
9.2% {2226ms} SmallInteger(Integer)>>timesRepeat:
5.9% {1410ms} ByteString(SequenceableCollection)>>copyReplaceFrom:to:with:
5.6% {1349ms} ByteString class(String class)>>new:
4.4% {1052ms} UndefinedObject>>Dolt
```

Memory

The statistical part on memory consumption tells the observed changes on the quantity of memory allocated and the garbage collector usage. To fully understand this information, one needs to keep in mind that Pharo’s garbage collector (GC) is a scavenging GC, relying on the principle that an old object has greater chance to live even longer. It is designed following the fact that an old object will probably be kept referenced in the future. On the contrary, a young object has greater chance to be quickly dereferenced.

Several memory zones are considered and the migration of a young object to the space dedicated for old object is qualified as tenured. (Following the metaphor of American academic scientists, when a permanent position is obtained.)

An example of the memory analyze realized by `MessageTally`:

```
**Memory**
old      +0 bytes
young    +9,536 bytes
used     +9,536 bytes
free     -9,536 bytes
```

`MessageTally` describes the memory usage using four values:

1. the `old` value is about the grow of the memory space dedicated to old objects. An object is qualified as “old” when its physical memory location is in the “old memory space”. This happens when a full garbage collector is triggered, or when there are too many object survivors (according to some threshold specified in the virtual machine). This memory space is cleaned by a full garbage collection only. (An incremental GC does not reduce its size therefore).

An increase of the old memory space is likely to be due to a *memory leak*: the virtual machine is unable to release memory, promoting young objects as old.

2. the young value tells about the increase of the memory space dedicated to young objects. When an object is created, it is physically located in this memory space. The size of this memory space changes frequently.
3. the used value is the total amount of used memory.
4. the free value is the remaining amount of memory available.

In our example, none of the objects created during the execution have been promoted as old. 9 536 bytes are used by the current process, located in the young memory space. The amount of available memory has been reduced accordingly.

GCs

The **GCs** provides statistics about the garbage collector. An example of a garbage collector report is:

```
**GCs**  
full      0 totalling 0ms (0.0% uptime)  
incr     13875 totalling 3,122,594ms (12967.0% uptime), avg 225.0ms  
tenures   0  
root table 0 overflows
```

Four values are available.

1. The full value totals the amount of full garbage collections and the amount of time it took. Full garbage collection are not that frequent. They results from often allocating large memory chunks.
2. The incr is about incremental garbage collections. Incremental GCs are usually frequent (several times per second) and quickly performed (about 1 or 2 ms). It is wise to keep the amount of time spent in incremental GCs below 10%.
3. The number of tenures tells the amount of objects that migrated to the old memory space. This migration happens when the size of the young memory space is above a given threshold. This typically happens when launching an application, when all the necessary objects haven't been created and referenced.
4. The root table overflows is the amount of root objects used by the garbage collector to navigate the image. This navigation happens when the system is running short on memory and need to collect all the objects relevant for the future program execution. The overflow value identifies the rare situation when the number of roots used by the incremental GC is greater than its internal table. This situation forces the GC to promote some objects as tenured.

In the example, we see that only the incremental GC is used. As we will subsequently see, the amount of created objects is quite relevant when one wants to optimize performances.

18.5 Illustrative Analysis

Understanding the result obtained when profiling is the very first step when one wants to optimize an application. However, as you probably started to feel, understanding why a computation is costly is not trivial. Based on a number of examples, we will see how comparing different profiling results greatly helps to identify costly message calls.

The method `","` is known to be slow since it creates a new character string and copy both the receiver and the argument into it. Using a Stream is a significant faster approach to concatenate character strings. However, `nextPut:` and `nextPutAll:` must be carefully employed!

Using a Stream for string concatenation. At the first glance, one could think that creating a stream is costly since it is frequently used with relatively slow inputs and outputs (e.g., network socket, disk accesses, Transcript). But replacing the string concatenation employed in the previous example by a stream operation is almost 10 times faster! This is easily understandable since concatenating 9000 times a character strings creates 8999 intermediate objects, each being filled with the content of another. Using a stream, we simply have to append a character at each iteration.

```
MessageTally spyOn:  
[ 500 timesRepeat: [  
    | str |  
    str := WriteStream on: (String new).  
    9000 timesRepeat: [ str nextPut: $A ]]].
```

– 807 tallies, 807 msec.

Tree

Process: (40s) 535298048: nil

Leaves

33.0% {266ms} SmallInteger(Integer)>>timesRepeat:
21.2% {171ms} UndefinedObject>>DoIt

Memory

```

old      +0 bytes
young    -18,272 bytes
used     -18,272 bytes
free     +18,272 bytes

**GCS**
full      0 totalling 0ms (0.0% uptime)
incr     1168 totalling 75,789ms (9391.0% uptime), avg 65.0ms
tenures   0
root table 0 overflows

```

String preallocation. Using `OrderedCollection` without a preallocation of the collection is well known for being costly. Each time the collection is full, its content has to be copied into a larger collection. Carefully choosing a preallocation has an impact of using ordered collections. The message `new: aNumber` could be used instead of `new`.

```

MessageTally spyOn:
[ 500 timesRepeat: [
    | str |
    str := WriteStream on: (String new: 9000).
    9000 timesRepeat: [ str nextPut: 'A' ]].

```

For this example, it is possible to improve the script by using the method `atAllPut:`. The script below takes only a couple of milliseconds.

```

MessageTally spyOn:
[ 500 timesRepeat: [
    | str |
    str := String new: 9000.
    str atAllPut: $A ]].

```

An experiment. Doing benchmarks shines when different executions are compared. In the previous piece of code, replacing the value 9000 by 500 is very valuable. The time taken with 9000 iterations is 2.7 times slower than with 500. Using the string concatenation (*i.e.*, using the `,` method) instead of a stream widens the gap with a factor 10. This experiment clearly illustrates the importance of knowing the list of the string characters manipulated.

The time of the profiled execution is also an important quality factor for the result. `MessageTally` employs a sampling technique to profile code. Per default, `MessageTally` samples the current executing thread each millisecond per default. It is therefore necessary that all the methods involved in the computation are executed a “fair” amount of time to appear in the result report. If the application to profile is very short (few milliseconds only), then executing it a number of times help improving the accuracy of the report.

18.6 Counting messages

The profiling we have realized so far is focused on method execution time. The advantage of method call stack sampling is that it has a relatively low impact on the execution. The disadvantage is the relatively imprecision of the result. Even though the obtained results are sufficient in most of the case, they are always an approximation of the real execution.

MessageTally allows for a profiling based on program interpretation. The idea is to use a bytecode interpreter instead of execution sampler. The main advantage is the exactness of the result. The information obtained with the message `tallySends:` indicates the amount of time each method involved in a computation has been executed. Figure 18.5 gives the result obtained by executing

```
MessageTally tallySends:[ 1000 timesRepeat: [3.14159 printString]].
```

The downside of `tallySend:` is the time taken to execute the provided block. The block to profile is executed by an interpreter written in Pharo, which is slower than the one of the virtual machine. A piece of code profiled by `tallySends:` is about 200 times slower. The interpreter is available from the method `ContextPart»runSimulated: aBlock contextAtEachStep: block2.`

18.7 Memorized Fibonacci

As a small application of the techniques we have seen, consider the Fibonacci function ($fib(n) = fib(n - 1) + fib(n - 2)$ with $fib(0) = 0, fib(1) = 1$). We will study two versions of it: a recursive version and a memorized version. Memoizing consists in introducing a cache to avoid redundant computation.

Consider the following definition, close to the mathematical definition:

```
Integer»fibSlow
  self assert: self >= 0.
  (self <= 1) ifTrue: [ ^ self].
  ^ (self - 1) fibSlow + (self - 2) fibSlow
```

The method `fibSlow` is relatively inefficient. Each recursion implies a duplication of the computation. The same result is computed twice, by each branch of the recursion.

A more efficient (but also slightly more complicated) version is obtained by using a cache that keeps intermediary computed values. The advantage is to not duplicate computations since each value is computed once. This classical way of optimizing program is called memoizing.

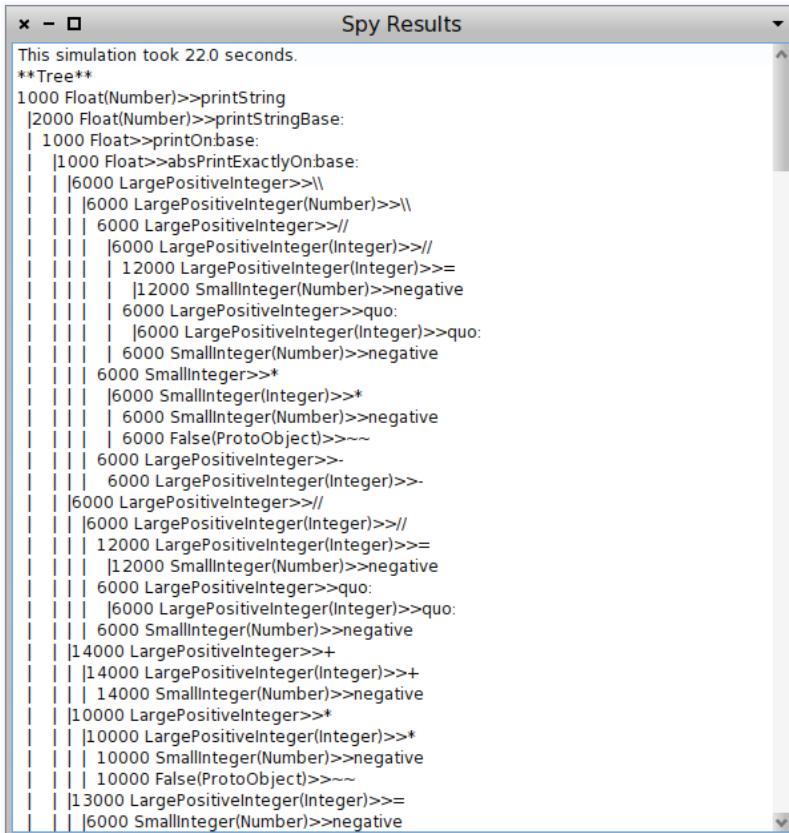


Figure 18.5: All executed messages during an execution.

```

Integer»fib
↑ self fibWithCache: (Array new: self)

Integer»fibLookup: cache
| res |
res := cache at: (self + 1).
↑ res ifNil: [ cache at: (self + 1) put: (self fibWithCache: cache ) ]

Integer»fibWithCache: cache
(self <= 1) ifTrue: [ ↑ self].
↑ ((self - 1) fibLookup: cache) + ((self - 2) fibLookup: cache)

```

As an exercise, profile 1200 fibSlow and 1200 fib to be convinced of the gain of memoizing.

18.8 SpaceTally for Memory Consumption per Class

It is often important to know the amount of instances and the memory consumption of a given class. The class SpaceTally offers this functionality.

The expression `SpaceTally new printSpaceAnalysis` runs over all the classes of the system and gathers for each of them its code size, the amount of instances and the total memory space taken by the instances. The result is sorted along the total memory space taken by instances and is stored in a file named `STspace.txt`, located next to the Pharo image.

It is not surprising to see that strings, compiled methods and bitmaps represents the largest part of the Pharo memory.

Class	code space	# instances	inst space	percent
ByteString	2217	91946	6325763	26.0
CompiledMethod	21186	60807	3704137	15.2
Bitmap	3893	319	3685532	15.1
Array	2478	96671	3015172	12.4
ByteSymbol	920	40109	1009703	4.1

SpaceTally may also be executed on a reduced set of classes. Consider:

```
((SpaceTally new spaceTally: (Array with: TextMorph with: Point))
    asSortedCollection: [:a :b | a spaceForInstances > b spaceForInstances])
```

18.9 Few advices

We have seen a number of strategies to measure and optimize a program. The examples we have used are relatively small. Optimizing a program is not always an easy task. Identifying method candidate for inserting a cache is simple and efficient once (i) you know when to invalidate the cache and (ii) when you are aware of the impact on the overall execution when inserting the code.

In general, it is more valuable to understand the overall algorithm than trying to optimize leaf methods. The way data are structured may also provide opportunities for optimization. For example, using an ordering collection or a linked list may not be appropriated to represent acyclic graphs. Using a set may offer better performance or a dictionary in the case that hash values are reasonably well distributed.

The memory consumption may plays an important role. The overall performance may significantly decreases if the garbage collector is often so-

licited. Recycling objects and avoiding unnecessary object creations helps reducing the solicitation of the garbage collector.

18.10 How MessageTally is implemented?

MessageTally is a gorgeous example on how to use Pharo's reflecting capabilities. The method `spyEvery: millisecs on: aBlock` contains the whole profiling logic. This method is indirectly called by `spyOn:`. The `millisecs` value is the amount of milliseconds between each sample. It is set at 1 per default. The block to be profiled is `aBlock`.

The essence of the profiling activity is given by the following code excerpt:

```
observedProcess := Processor activeProcess.
Timer := [
  [ true ] whileTrue: [
    | startTime |
    startTime := Time millisecondClockValue.
    myDelay wait.
    self
      tally: Processor preemptedProcess suspendedContext
      in: (observedProcess == Processor preemptedProcess
            ifTrue: [ observedProcess ] ifFalse: [ nil ])
      by: (Time millisecondClockValue - startTime) // millisecs].
  nil] newProcess.
Timer priority: Processor timingPriority-1.
```

Timer is a new process, set at a high priority, that is in charge of monitoring `aBlock`. The process scheduler will therefore favorably active it (`timingPriority` is the process priority of system processes). It creates an infinite loop that waits for the amount of necessary milliseconds (`myDelay`) before snapshooting the method call stack. The process to observe is `observedProcess`. It is the process in which the message `spyEvery: millisecs on: aBlock` has been sent.

The idea of profiling is to associate to each method context a counter. This association is realized with an instance of the class `MessageTally` (the class defines the variables `class`, `method` and `process`).

At a regular interval (`myDelay`), the counter of each stack frame is incremented with the amount of elapsed milliseconds. The stack frame is obtained by sending `suspendedContext` to the process that has just been preempted.

The method `tally: context in: aProcess by: count` increments each stack frame by the amount of milliseconds given by `count`.

The memory statistic are given by differentiating the amount of con-

sumed memory, before and after the profiling. Smalltalk, an instance of the class `SmalltalkImage`, contains many accessing methods to query the amount of available memory.

18.11 Chapter Summary

In this chapter, we see the basic of profiling in Pharo. It has presented the functionalities of `MessageTally` and introduced a number of principles for re-sorbing performance bottleneck.

- The method `timeToRun` and `bench` offer simple benchmarking and should be sent to a block.
- `MessageTally` is a sampling-based code profiler.
- Evaluating `MessageTally spyOn: ["an expression"]` executes the provided block and display a report.
- Accuracy is gained by increasing the execution time of the profiled block.
- The Pharo programming environment gives several convenient ways to profile.
- Counting messages is slow but accurate profiling technique.
- Memoization is a common and efficient code pattern to speed up execution.
- `SpaceTally` reports about the memory consumption.