Chapter 1

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. Gofer was developed by L. Renggli and further extended by E. Lorenzano and C. Bruni. It is used by Metacello, the language to manage package maps that we present in Metacello Chapter (see Chapter **??**).

1.1 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, for example, that a package 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: ''
```

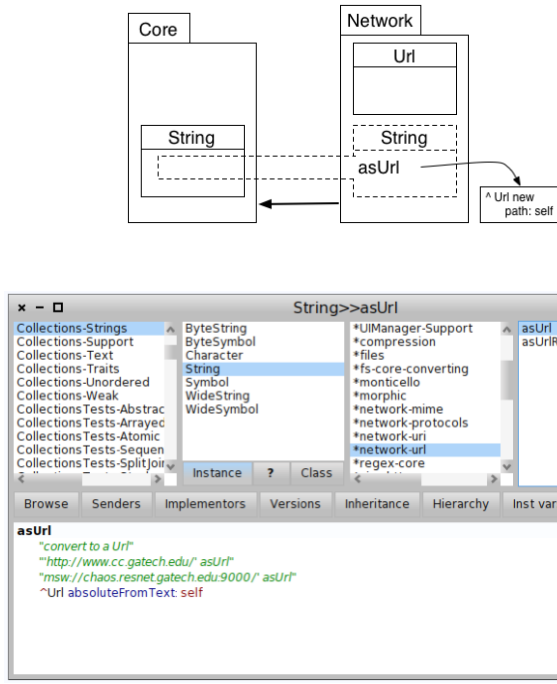

Figure 1.1: The browser shows that the class String gets the methods asUrl and asUrlRelativeTo: from the package network−url

```
classVariableNames: ''
poolDictionaries: ''
category: 'Zork'
```

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. The more a project grows, the more important it is 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

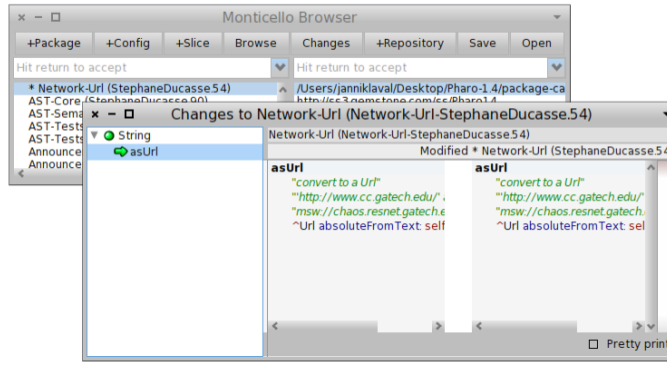

Figure 1.2: The change browser shows that the method String>>asUrl has changed.

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 feed. A source open to everybody is available at http://www.smalltalkhub.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 1.3 the package a.1 is loaded from the server smalltalk.com. 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

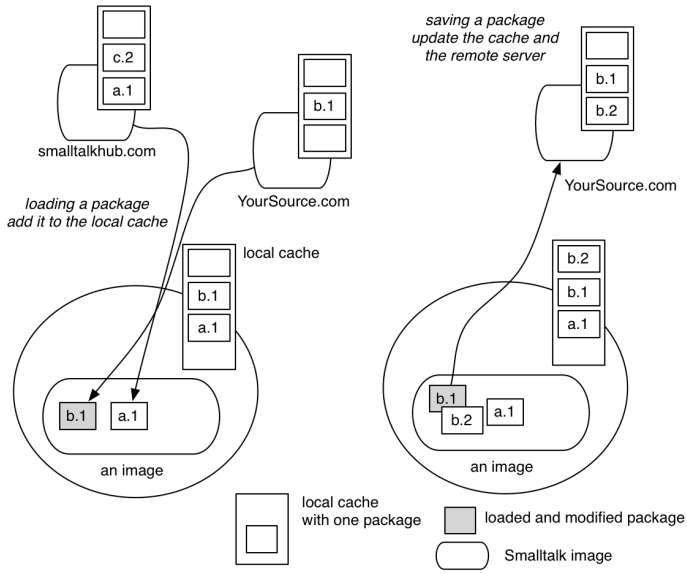

Figure 1.3: (left) Typical setup with clean and dirty packages loaded and cached — (right) Package published.

saved in the cache and the remote server.

1.2 What is Gofer?

Gofer is a scripting tool for Monticello. It has been 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 the system stays 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 the 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 packages.

You can ask Gofer to update it itself by executing the following expression:

```
Gofer gofer update
```

1.3 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 a hard-disc. The location is the same as the one used to access a Monticello repository. For example it is 'http://smalltalkhub.com/mc/MyAccount/MyPackage/main' as used in the following expression.

```
MCHttpRepository
   location: 'http://smalltalkhub.com/mc/MyAccount/MyPackage/main'
   user: ''
   password: ''
```

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.smalltalkhub.com in the account of JannikLaval.

```
Gofer new
   url: 'http://smalltalkhub.com/mc/PharoBooks/GoferExample/main';
   package: 'PBE2GoferExample';
   load
```

When the repository (HTTP or FTP) requires an identification, the message url:username:password: is available. Pay close attention as this is a single message, so do not put cascade in between. The message directory: supports the access to local files.

```
Gofer new
   url: 'http://smalltalkhub.com/mc/PharoBooks/GoferExample/main'
   username: 'pharoUser'
   password: 'pharoPwd';
   package: 'PBE2GoferExample';
   load.
```

```
"we work on the project PBE2GoferExample and provide credentials"
Gofer new
   url: 'http://smalltalkhub.com/mc/PharoBooks/GoferExample/main/PBE2GoferExample'
   username: 'pharoUser'
   password: 'pharoPwd';
   package: 'PBE2GoferExample';        "define the package to be loaded"
   disablePackageCache;                "disable package lookup in local cache"
   disableRepositoryErrors;            "stop the error raising"
   load.                               "load the package"
```

Since the same public servers are often used, Gofer's API offers a number of shortcuts to shorten the scripts. Often, we want to write a script and give

it to other people to load our code. In such a case having to specify a password is not really adequate. Here is an example for smalltalkHub (which has some verbose urls such as 'http://smalltalkhub.com/mc/PharoBooks/GoferExample/main' for the project GoferExample). We use the smalltalkhubUser:project: message and just specify the minimal information. In this chapter, we also use squeaksource3: as a shortcut for http://ss3.gemtalksystems.com/ss.

```
"Specifying a user but no password"
Gofer new
   smalltalkhubUser: 'PharoBooks' project: 'GoferExample';
   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
   smalltalkhubUser: 'PharoBooks' project: 'GoferExample';
   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
   smalltalkhubUser: 'PharoBooks' project: 'GoferExample';
   package: 'PBE2GoferExample'
   constraint: [ :version | version author = 'janniklaval' ];
   load
```

1.4 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 OSProcess using its Metacello configuration.

```
Gofer new
   "we will load a version of the configuration of OSProcess "
   url: 'http://www.squeaksource.com/MetacelloRepository';
   package: 'ConfigurationOfOSProcess';
   load.

"Now to load OSProcess you need to ask for its configuration."
((Smalltalk at: #ConfigurationOfOSProcess) project version: #stable) load.
```

The following code snippet loads multiple packages from different servers. The loading order is respected: the script loads first Collections−Arithmetic, then the necessary for Sound, and finally Phratch, a port of the well Scratch visual programming language.

Take note that it will load the complete Phratch application, and may take a moment.

```
Gofer new
   url: 'http://smalltalkhub.com/mc/PharoExtras/CollectionArithmetic/main';
   package: 'Collections−Arithmetic';
   url: 'http://smalltalkhub.com/mc/PharoExtras/Sound/main';
   package: 'Sound';
   package: 'Settings−Sound';
   package: 'SoundScores';
   package: 'SoundMorphicUserInterface';
   url: 'http://smalltalkhub.com/mc/JLaval/Phratch/main';
   package: 'Phratch';
   load
```

This example may give the impression that Collections−Arithmetic is looked up in the CollectionArithmetic repository of the server smalltalkhub and that Phratch is looked up in the project Phratch of the smalltalkhub 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
   url: 'http://smalltalkhub.com/mc/PharoExtras/CollectionArithmetic/main';
   url: 'http://smalltalkhub.com/mc/PharoExtras/Sound/main';
```

```
  url: 'http://smalltalkhub.com/mc/JLaval/Phratch/main';
  package: 'Collections−Arithmetic';
  package: 'Sound';
  package: 'Settings−Sound';
  package: 'SoundScores';
  package: 'SoundMorphicUserInterface';
  package: 'Phratch';
  load
```

When we want to specify that package should be loaded from a specific server, we should write multiple scripts.

```
Gofer new
  url: 'http://smalltalkhub.com/mc/PharoExtras/CollectionArithmetic/main';
  package: 'Collections−Arithmetic';
  load.
Gofer new
  url: 'http://smalltalkhub.com/mc/PharoExtras/Sound/main';
  package: 'Sound';
  package: 'Settings−Sound';
  package: 'SoundScores';
  package: 'SoundMorphicUserInterface';
  load.
Gofer new
  url: 'http://smalltalkhub.com/mc/JLaval/Phratch/main';
  package: 'Phratch';
  load
```

Note that such scripts load the latest versions of the packages and are therefore fragile, because if a new package version is published, you will load it even if this is unstable. 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, 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 as before, with some changes.

For FTP, we should specify the URL using 'ftp' as the 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 subfolders 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 basis 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: Obsolete system information is cleaned.
commit / commit:	Save the packages to a distant server – with a message log.
revert	Reload previously loaded packages.
recompile	Recompile packages
unload	Unload the packages from the image
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.

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 copy. 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 or else the operation will happen silently.

```
Gofer new
   smalltalkhubUser: 'PharoBooks' project: 'GoferExample';
   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 the current version again. 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 - in general this is a good practice.

```
Gofer new
   "We save the package in the repository"
   smalltalkhubUser: 'PharoBooks' project: 'GoferExample';
   package: 'PBE2GoferExample';
   "We comment 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
   smalltalkhubUser: 'PharoBooks' project: 'GoferExample';
   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
   smalltalkhubUser: 'PharoBooks' project: 'GoferExample';
   "we add the latest version of PBE2GoferExample"
   package: 'PBE2GoferExample';
   "we browse the latest version published on the server"
   browseRemoteChanges
```

The unload operation. The message unload unloads the packages from the image. Note that using the Monticello browser you can delete a package, but such an 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
   package: 'PBE2GoferExample';
   unload
```

Note that you cannot unload Gofer itself that way. Gofer gofer unload does not work.

Fetch and push operations. Since Monticello is a distributed versioning system, it is a good idea to save all the versions you want locally, without being forced to published on a remote server. This is especially true when working off-line. As it 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 the packages that are missing from the remote server 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
   smalltalkhubUser: 'PharoBooks' project: 'GoferExample';
   package: 'PBE2GoferExample';
   fetch
```

Now, if you want to load your packages locally remember to set up the lookup so that it takes into account the local cache and disables errors as presented in the beginning of this chapter (messages disableRepositoryErrors and enablePackageCache).

The message push performs the inverse operation. It publishes locally available packages to the remote server. All the packages that you published locally are then pushed to the server.

```
Gofer new
   smalltalkhubUser: 'PharoBooks' project: 'GoferExample';
   package: 'PBE2GoferExample';
   push
```

As a pattern, we always keep the copies of all the versions of our projects or the projects we used in our local cache. 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 synchronizes local and remote repositories.

```
Gofer new
   smalltalkhubUser: 'PharoBooks' project: 'GoferExample';
   package: 'PBE2GoferExample';
   push.
Gofer new
   smalltalkhubUser: 'PharoBooks' project: 'GoferExample';
   package: 'PBE2GoferExample';
   fetch
```

As mentioned earlier, you can have multiple packages to be pushed and fetched from.

```
Gofer new
   smalltalkhubUser: 'PharoBooks' project: 'GoferExample';
   package: 'PBE2GoferExample';
   package: 'PBE2GoferExampleSecondPackage';
   push.
Gofer new
   smalltalkhubUser: 'PharoBooks' project: 'GoferExample';
   package: 'PBE2GoferExample';
   package: 'PBE2GoferExampleSecondPackage';
   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 it is important to know how to supply answers from within a script to these questions. The message valueSupplyingAnswers: supports such a task.

```
[ Gofer new
   squeaksource: '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 in previous examples

Configuration Loading

Gofer also supports Metacello configuration loading. It provides a set of the following messages to handle configurations: configurationOf:, loadVersion:, loadDevelopment, and loadStable.

In this example, loading the development version of NativeBoost. There you need only to specify the NativeBoost project and you will load the ConfigurationOfNativeBoost and execute the loading the development version.

```
Gofer new
    smalltalkhubUser: 'Pharo' project: 'NativeBoost';
    configuration;
    loadDevelopment
```

When the repository name does not match the name of the configuration you should use configurationOf: and provide the name of the configuration class.

1.5 Some useful scripts

Gofer offers a nice facility to gather all the packages together in a given repository via the message allResolved.

Script 1.1: *Getting the number of packages in a repository.*

```
(Gofer new
    smalltalkhubUser: 'Pharo' project: 'NativeBoost';
    allResolved) size
```

The following script gathers the package versions by packages and returns a dictionary.

Script 1.2: *Grouping versions by package names.*

```
((Gofer new
    smalltalkhubUser: 'Pharo' project: 'NativeBoost';
```

```
   allResolved)
   groupedBy: [ :each | each packageName])
```

Script 1.3: *Getting the package list for the Kozen project hosted on SS3.*

```
((Gofer new
   squeaksource3: 'Kozen';
   allResolved)
   groupedBy: [ :each | each packageName]) keys
```

Fetching packages

Here is a script to fetch all the packages of a given repository. It is useful for grabbing your files and having a version locally.

Script 1.4: *Fetching all the packages of a repository*

```
| go |
go := Gofer new squeaksource3: 'Pharo20'.
go allResolved
   do: [ :each |
         self crLog: each packageName.
         go package: each packageName;
            fetch]
```

Script 1.5: *Fetching all the refactoring packages from the Pharo2.0 repository*

```
| go |
go := Gofer new.
go squeaksource3: 'Pharo20'.
(go allResolved select: [ :each | 'Refactoring*' match: each packageName])
   do: [ :pack |
      self crLog: pack packageName.
      go package: pack packageName; fetch]
```

Publishing local files

The following script publishes files from your local cache to a given repository.

Script 1.6: *How to publish package files to a new repository using Pharo 1.4*

```
| go |
go := Gofer new.
go repository: (MCHttpRepository
   location: 'http://ss3.gemtalksystems.com/ss/Pharo14'
   user: 'pharoUser'
```

```
   password: 'pharoPwd').

((FileSystem workingDirectory / 'package-cache')
   allEntries
      select: [ :each | '*.mcz' match: each])
            do: [ :f | go version: ('.' join: (f findTokens: $.) allButLast); push]
```

The following script uses the new filesystem library, we also show how we can get the package name and not the versions. The script also pays attention to only publish mcz files. It can be extended to publish selectively specific packages.

Script 1.7: *How to publish package files to a new repository using Pharo 20*

```
| go |
go := Gofer new.
go repository: (MCHttpRepository
   location: 'http://ss3.gemtalksystems.com/ss/rb-pharo'
   user: 'pharoUser'
   password: 'pharoPwd').

(((FileSystem disk workingDirectory / 'package-cache')
   allFiles select: [:each | '*.mcz' match: each basename])
      groupedBy:  [:each | (each base copyUpToLast: $-) ])
         keys do: [:name | go package: name; push]
```

Script 1.8: *How to publish Fame to the Moose Team on SmalltalkHub*

```
|go repo|
repo := MCSmalltalkhubRepository
  owner: 'Moose'
  project: 'Fame'
  user: 'pharoUser'
  password: 'pharoPwd'.

go := Gofer new.
go repository: repo.
(((FileSystem disk workingDirectory / 'package-cache')
   allFiles select: [:each | '*Fame*.mcz' match: each basename])
      groupedBy: [:each | (each base copyUpToLast: $-) ]) keys
         do: [:name | go package: name; push]
```

All in one page

Since we love simple scripts where we do not have to think too much, here is a full version to migrate between Monticello repositories.

Script 1.9: *Script to synchronize from one Monticello repository to another*

```
| source goferSource destination goferDestination files destinationFiles |

source := MCHttpRepository
        location: 'http://www.squeaksource.com/MyProject'.
destination := MCSmalltalkhubRepository
   owner: 'TheOwner'
   project: 'YourPackage'
   user: 'YourName'
   password: ''.

goferSource := Gofer new repository: source.
goferDestination := Gofer new repository: destination.

files := source allVersionNames.
"select the relevant mcz packages"
goferSource allResolved
   select: [ :resolved | files anySatisfy: [ :each | resolved name = each ] ]
   thenDo: [ :each | goferSource package: each packageName ].
"download all mcz on your computer"
goferSource fetch.

"check what files are already at destination"
destinationFiles := destination allVersionNames.
"select only the mcz that are not yet at destination"
files
   reject: [ :file | destinationFiles includes: file ]
   thenDo: [ :file | goferDestination version: file ].
"send everything to SmalltalkHub"
goferDestination push.

"check if we have exactly the same files at source and destination"
destination allVersionNames sorted = files sorted.
```

1.6 Chapter summary

Gofer provides a robust and stable implementation to script the management of your packages. When your project grows, you should really consider using Metacello (see Chapter **??**).

In this chapter, we introduce how we can script package with Gofer.

- The method `load` allows us to load packages from sources given with the method `url:` and `package:`.
- The method `url:` supports FTP and local directory access.

- The API provides some useful shortcuts: smalltalkhubUser:project: is a shortcut for http://www.smalltalkhub.com, squeaksource3: for http://ss3.gemtalksystems.com/ss/ and gemsource: for http://seaside.gemstone.com/ss/.
- We can load several packages by calling the method package: multiple times before calling load.
- Once a Gofer instance is parametrized, one can perform a number of relevant operations: update, merge, push, ...