Chapter 1

Files with FileSystem

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.

1.1 Getting started

First we need to load the package:

```
Gofer new
    wiresong: 'mc';
    package: 'Filesystem';
    load.
```

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 := FSDiskFilesystem current 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 not use platform specific classes such as FSUnixFilesystem or FSWindowsFilesystem. All code below works on FSReference instances.

1.2 Navigating the Filesystem

Now let's do some more interesting things. To list children of your working directory evaluate the following expression:

```
working := FSDiskFilesystem current working.
working children.
```

To 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:

```
cache exists.          " ⟶ true"
cache isFile.          " ⟶ false"
cache isDirectory.     " ⟶ true"
cache basename.        " ⟶ 'package−cache'"
```

To get additional information about the filesystem entry evaluate:

```
cache entry creation.     " ⟶ 2010−02−14T10:34:31+00:00"
cache entry modification. " ⟶ 2010−02−14T10:34:31+00:00"
cache entry size.         " ⟶ 0 (directories have size 0)"
```

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.

ThatÕs 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].

1.3 Design

Stéf ►*should add class comments and a uml diagram*◄

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"
(FSPath / 'plonk') contains: (FSPath / '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 FSDisk-

Filesystem and it's platform-specific subclasses. But other kinds of Filesystems are also possible. FSMemoryFilesystem provides a RAM diskÐa 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: (FSPath / 'plonk').
griffle := FSPath / 'plonk' / 'griffle'.
nurp := FSPath * 'nurp'.

fs resolve: nurp            => /plonk/nurp

fs createDirectory: (FSPath / '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: (FSPath / 'plonk' / 'griffle').
nurp := fs referenceTo: (FSPath * '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 my 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.