Project Report
Group Badgers

Java and C# in depth, Spring 2013

Thomas Frick (03-150-927)
Matthias Ganz (04-862-850)
Philipp Rohr (04-397-030)

April 7, 2013

1 Introduction

This document describes the design and implementation of the *Personal Virtual File System* of group *Badgers*. The project is part of the course *Java and C# in depth* at ETH Zurich. The following sections describe each project phase, listing the requirements that were implemented and the design decisions taken. The last section describes a use case of using the *Personal Virtual File System*.

2 VFS Core

VFS Core is a library that provides an implementation of a virtual file system. The API that a client of this library can use consists of three interfaces that are described in 2.2.1. The VFS Core provides functionality to create/open/dispose new virtual disks and allows the management of files and directories within such a disk. Furthermore it provides a simple way to import/export files from/to the host file system.

The library internally works with a virtual disk that is divided into a header, index and data section, having the index represented as B-tree. Such a design allows access with good performance to the data inside the disk.

2.1 Requirements

Below one will find a list of the requirements implemented in this project.

2.1.1 disk management

- *The virtual disk must be stored in a single file in the working directory in the host file system.*
- *VFS must support the creation of a new disk with the specified maximum size at the specified location in the host file system.*
- *VFS must support several virtual disks in the host file system.*
- *VFS must support disposing of the virtual disk.*
- *VFS must support querying of free/occupied space in the virtual disk.*

These requirements are met with the classes

`ch.eth.jcd.badgers.vfs.core.VFSDiskManagerImpl`
`ch.eth.jcd.badgers.vfs.core.config.DiskConfiguration`

allow creation/deletion and opening of the disk on the host file system. Clients of the library have to pass a *DiskConfiguration* to the *VFSDiskManagerImpl* when calling *create* or *open*

2.1.2 file management

- *VFS must support creating/deleting/renaming directories and files.*
- *VFS must support navigation: listing of files and folders, and going to a location expressed by a concrete path.*
- *VFS must support moving/copying directories and files, including hierarchy.*
- *VFS must support importing files and directories from the host file system.*
- *VFS must support exporting files and directories to the host file system.*

These requirements are met with the classes

`ch.eth.jcd.badgers.vfs.core.VFSEntryImpl`
`ch.eth.jcd.badgers.vfs.core.VFSPathImpl`
`ch.eth.jcd.badgers.vfs.core.VFSFileInputStream`
`ch.eth.jcd.badgers.vfs.core.VFSFileOutputStream`

The *VFSEntryImpl* allows copy, move (and rename), delete, listing of children and going to the parent (navigation). Together with the streams it also supports importing/exporting to any location clients of the VFS core library wish to. The classes

`ch.eth.jcd.badgers.vfs.ui.VFSConsole`
`ch.eth.jcd.badgers.vfs.ui.VFSUIController`

demonstrate this by importing and exporting to the host file system.

2.1.3 bonus features

- *Elastic disk: Virtual disk can dynamically grow or shrink, depending on its occupied space.*

The implementation only allows growing if more files are imported. Shrinking is not supported.

- *Compression, if implemented with 3d party library.*
- *Compression, if implemented by hand (you can take a look at the arithmetic compression)*

  The classes

`ch.eth.jcd.badgers.vfs.compression.BadgersLZ77CompressionInputStream`
`ch.eth.jcd.badgers.vfs.compression.BadgersLZ77CompressionOutputStream`
`ch.eth.jcd.badgers.vfs.compression.BadgersRLECompressionInputStream`
`ch.eth.jcd.badgers.vfs.compression.BadgersRLECompressionOutputStream`

implement streams that can be wrapped around *VFSFileInputStream* and *VFSFileOutputStream*. The *DiskConfiguration* allows to switch compression on and to declare which algorithm shall be chosen. This allows easy configuration of any 3rd party compression streams (which was not chosen to implement, because of the implementation of our own compression algorithm).

- *Encryption, if implemented with 3rd party library.*
- *Encryption, if implemented by hand.*

The classes

`ch.eth.jcd.badgers.vfs.encryption.CaesarInputStream`
`ch.eth.jcd.badgers.vfs.encryption.CaesarOutputStream`

show how encryption can be implemented in the library. It was chosen to implement encryption similar to compression with streams, which allows easy configuration via *DiskConfiguration*. These streams are mainly for demonstration how encryption should work and shall not be used in high security environments :-)

2.2 Design

This section describes the main aspects of the VFS core library. It shows the implementation of the core interfaces and classes, explains the mock classes and tests and eventually describes the file format and its management classes.

2.2.1 Core Classes

Figure 1 gives an overview of the main interfaces and classes that were implemented. The interfaces *VFSDiskManager*, *VFSEntry* and *VFSPath* can be used by clients using the library implemented here.

- *VFSDiskManager* The implementations of this interface provide mainly a way to open, create and dispose new virtual disks. Additionally one can get the root entry of the file system and get additional information about an opened disk.

- *VFSEntry* Represents a directory or file on the file system. A *VFSEntry* provides all the required methods to manipulate files and directories and importing/exporting files into the virtual file system. The general meaning of *VFSEntry* is, that such objects usually exist on the filesystem.

- *VFSPath* Represents a path on the file system to a given *VFSEntry*. It has a slightly looser coupling to the file system as a path does not imperatively need to exist.

- *FindInFolderCallback* The *find* and *findInFolder* methods on *VFSDiskManager* and *VFSEntry* require a callback object where the find mechanisms provided by the mock and the real implementations will notify the caller when a new entry is found. So a client can start the search asynchronously and can handle the notifications for example with updating lists. The console implementation simply lists the absolute paths to the found entries.

The intention of those interfaces is to hide the real implementation of the virtual file system from a client. With that in mind it should be simple to add a network layer upon the real implementation without changing client code. The classes *VFSDiskManagerImpl, VFSEntryImpl (and its descendants) and VFSPathImpl* implement all the management for actually using the VFS on a virtual disk.

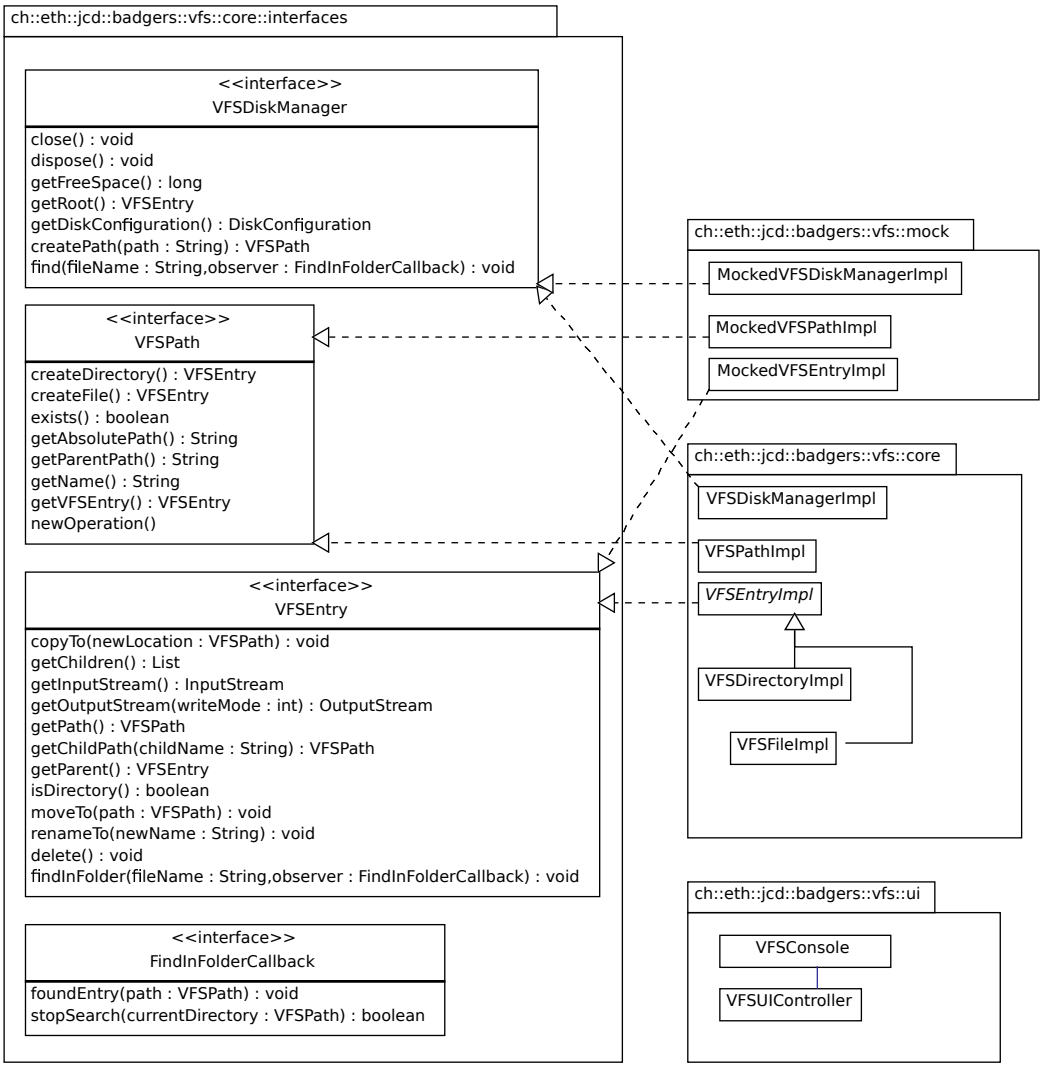

Figure 1: core classes

2.2.2 Mocking

For discussing the semantics of the virtual file system and the development of the interfaces explained in 2.2.1 it was decided to implement a mock that works against the host file system. The mock classes implement all the interfaces and were very helpful for acquiring a common understanding of how the interface shall be used by clients. In a further step it was very useful to have the mock classes while developing the console application. Hence the console could be developed independently from the real implementation.

2.2.3 Test

During the development a bunch of test cases came to life. The tests solely depend on the interfaces and thus they can run against the mock classes and the real implementation. This was a huge help in finding bugs in the slightly more complicated real implementation.

2.2.4 The *real* implemenation

Eventually some code was developed that handles a virtual disk. This code and some details about the file format are described in this section.

The classes Figure 2 shows the overview over the classes implemented for the disk handling and file management.

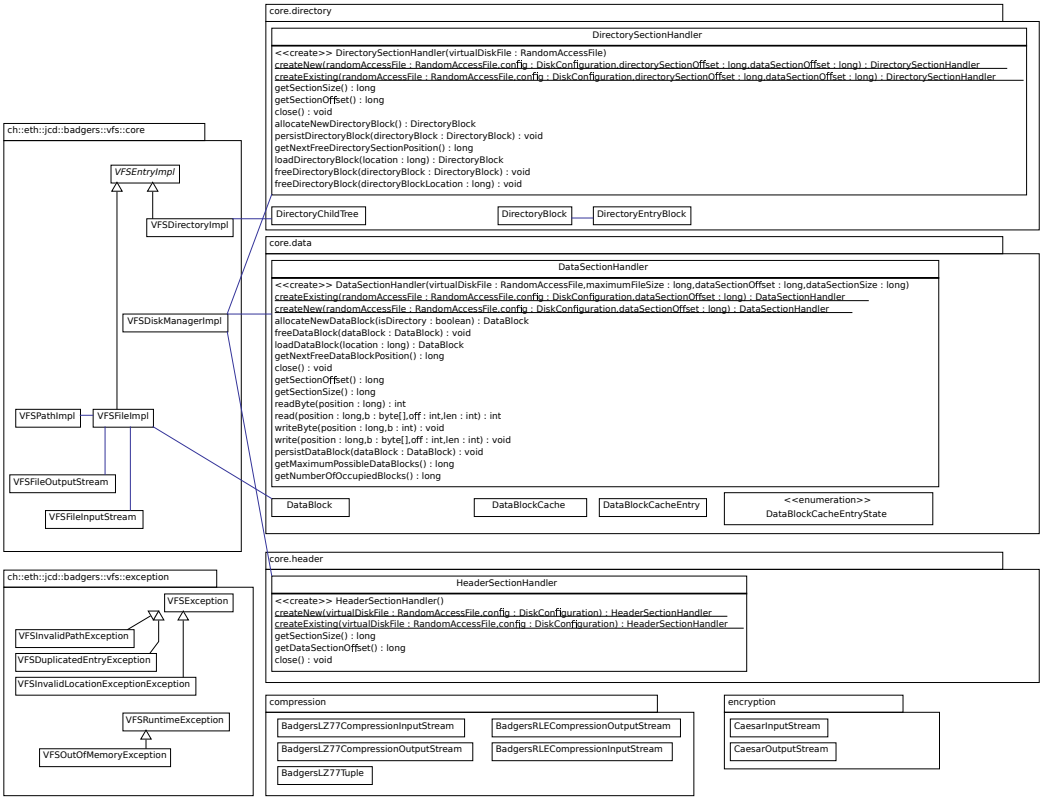

Figure 2: Implementation of the VFS core classes

The classes were divided into several packages which are explained here in more detail.

core The *core* package contains the implementation of the interfaces mentioned in section 2.2.1. Clients of the VFS core library operate on these classes to manipulate the virtual file system.

exception While manipulating the file system some exceptional behaviour might occur and thus some exceptions will be thrown by the core

classes. These exception are propagated to the clients of the VFS core libraries.

core.header As the virtual disk is divided into three sections (header, directory and data) the classes that handle the management of the header section are put in the package *core.header*.

core.data *core.data* contains the classes that manipulate the data section on the file system. This mainly means managing data blocks and actually writing/reading the raw data bytes of files to the virtual disk.

core.directory *core.directory* contains the classes that manage the index of the file system. That means that the whole directory structure is maintained in here. This is done in a B-Tree that contains all references to directories and files currently saved in the file system. More about the internals of directory section can be found in section 2.2.4

encryption The encryption package shows some demo classes that implement a ceasar cipher[1]. These classes are mainly to show how encrpytion in VFS core can be implemented and configured. To enable encryption in a new virtual disk one has to enable it in the *DiskConfiguration* that is passed to the *VFSDiskManagerImpl* at creation. Upon selecting the encryption algorithm the encryption streams will be wrapped around the *VFSFileInput- and VFSFileOutputStream*s.

compression To reduce the data volume within the virtual disk, compression on each file can be enabled. As mentioned earlier, compression is implemented as Input- and OutputStreams and thus is wrapped around the *VFSFileInput- and VFSFileOutputStream*s. Currently available compression algorithms are run length encoding [1] and LZ77 [2].

- *Run Length Encoding* The available 8bit run length encoding(rle) algorithm is a very simple form of data compression where multiple occurrence of the same byte were stored as a single byte value and the corresponding count. It is useful for simple graphic images like line drawings and icons.
- *LZ77* Abraham Lempel and Jacob Ziv introduced the LZ77 lossless compression algorithm in 1977. Newer compression methods such as

[1] http://en.wikipedia.org/wiki/Caesar_cipher

GZIP or DEFLATE often use LZ77-based algorithms. The compression is achieved by replacing the data with a reference to an earlier existing copy in the data input stream. For that a window of a certain size is held in memory where existing copies of the current data are searched.

The File Format This section describes the binary file format used by the file system inside a virtual disk. The file is separated into three major parts. The header, index and the data section. Each of them is described below. Figure 3 gives an overview, of how the sections are distributed in the virutal disks and what contents they have.

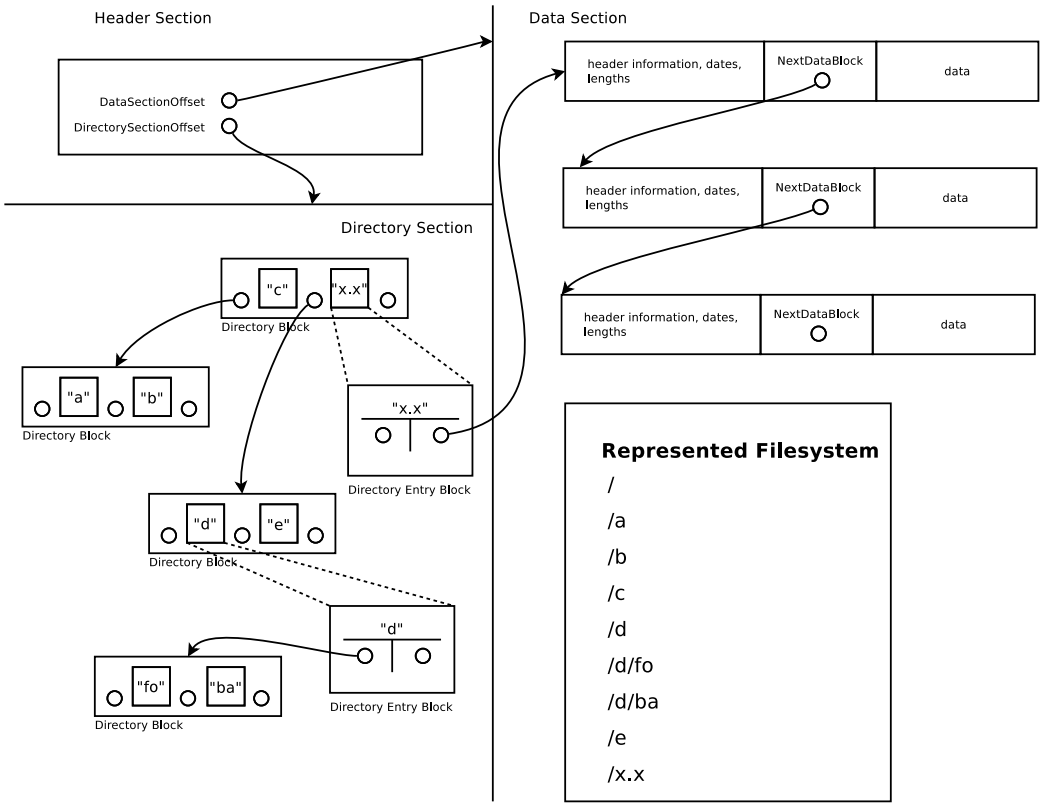

Figure 3: Overview of a disk

Header Section The header section contains some general information about the currently opened virtual disk. The details can be found in the following table:

Name	Length	Description
Info	50 byte UTF-8 String	Contains something like Badger VFS 2013 V1.0
Version	10 byte UTF-8 String	Contains something like "1.0"
Compression used	20 byte UTF-8 String	null or indicates compression used for this file
Encryption used	20 byte UTF-8 String	null or indicates encryption used for this file
DirectorySectionOffset	long (8 byte)	File offset where the directory section starts
DataSectionOffset	long (8 byte)	File offset where our data section starts
SaltString	8 bytes	Salt used to hash username and password randomly string generated while creating this file. (Not implemented yet)
Password	xxx bytes	CryptoHash (SHA-whatever) of Password+SaltString (Not implemented yet)

Directory Section The directory section describes which files and folders belong to which parent directory. This section has a fixed size and contains so called *DirectoryBlocks* which also have a fixed size. This makes management and manipulation easy. To each directory belongs a B-Tree structure which lists all contained entries.

DirectoryBlock One *DirectoryBlock* represents a node in the B-Tree of order 2.

Name	Length	Description
DirectoryHeader	1 byte	Header information. This header makes it easy to determine whether a *DirectoryBlock* is in use or not (memory management)
DirectoryEntryBlock1	128 byte	The smaller key inserted into the B-Tree
DirectoryEntryBlock2	128 byte	The bigger key inserted into the B-Tree
DirectoryBlockLink1	8 byte	Points to another DirectoryBlock which contains keys smaller than DirectoryEntryBlock1
DirectoryBlockLink2	8 byte	Points to another DirectoryBlock which contains keys bigger than DirectoryEntryBlock1 but smaller than DirectoryEntryBlock2
DirectoryBlockLink3	8 byte	Points to another DirectoryBlock which contains keys bigger than DirectoryEntryBlock2

DirectoryEntryBlock Represents a single directory or file.

Name	Length	Description
Filename	112 byte	UTF-8 file name String
DataBlockLocation	8 byte	Pointer to a DataBlock located in the Data Section. This DataBlock holds some meta information about the current directory
DirectoryEntryTreeRoot	8 bytes	Pointer to a DirectoryBlock located in the Directory Section. This referenced DirectoryBlock is the Root Block of a B-Tree containing all entries of that directory specified by the current Directory Entry Block. **This field containing a 0 indicates that this entry represents a file not a directory**

Data Section The data section is split into blocks where each of them is 1024 bytes long. Each block contains some amount of data and points to a subsequent block (Simple linked list).

Block layout

Name	Length	Description
BlockHeader 0) Header-Bit (LSB) 1) not used 2) not used 3) not used 4) not used 5) not used 6) not used 7) not used	1 byte	 If set to 1 this is the first DataBlock of a file.
NextDataBlock	8 byte	Points to the start address of the next DataBlock (linked list). 0 if this is the last DataBlock of a certain file or folder.
CreationDate	8 byte	UTC Time when this file was created *This field only exists if Header-Bit is set to 1*
DataLength	4 byte	Indicates the number of data saved on this DataBlock.
Data	n byte	user data (may be encrypted/compressed)

The root directory TODO: TEXT OR REMOVE

3 Quick Start Guide

3.1 the eclipse project

The project requires to be compiled with JAVA 7. It also depends on the maven plugin which pulls in all the required libraries.

3.2 Command line client

The command line client allows the usage of the VFS core and is mainly intended to test the basic functionalities. The console runs either in management mode or in file system mode. The management mode is entered automatically when starting the command line client. It allows creating and opening virtual disks. The file system mode is entered as soon as a virtual disk is opened.

3.2.1 startup

The command line client can be started as follows:

```
java -jar VFSCore.jar ch.eth.jcd.badgers.vfs.ui.VFSConsole
```

or by starting `ch.eth.jcd.badgers.vfs.ui.VFSConsole` in eclipse.

This gives a console prompt where the following commands can be used in.

3.2.2 commands

Following commands can be used with the command line client in management mode:

- **create c:\path\to\disk.bfs [size]** creates virtual disk with a maximum quota of [size] megabytes on the host system. The file may grow up to [size] megabytes. There is curently no way to change encryption or compression by using the console application. By default no encryption and the LZ77 compression will be used.
- **open c:\path\to\disk.bfs** opens filesystem mode for the given virtual disk
- **exit** exits the console program

following commands can be used in file system mode:

- **ls** lists the contents of the current directory
- **pwd** shows the path to the current directory
- **df** shows the usage of the current virtual disk space
- **cd dst** changes current directory to *dst* which must be either a child directory of the current path or ".."
- **find searchString** lists absolute paths of all files containing *searchString* in their file name
- **mkdir dirName** creates a new directory *dirName* in the current path
- **mkfile fileName** creates a new empty file *fileName* in the current path - this is rather not useful, as the "import" creates a file with content
- **rm file** deletes the entry denoted as *file*, it must be a child of the current path
- **cp src dst** copies the *src* file to *dst* as a child of the current path
- **mv src dst** moves the *src* file to *dst*
- **import ext_src dst** imports a *ext_src* from the host system to *dst*
- **export src ext_src** exports a *src* file to the host system *ext_dst*
- **find searchString** lists all filesystem entries below the current entry containing *searchString*
- **dispose** deletes the currently opened virtual disk
- **close** closes the file system mode, from now on management mode commands can be executed

4 Glossary

VFS core The main Java library, that handles all the interaction with virtual disks and importing/exporting/storing files. It is used by the command line client and the gui.

Virtual Disk A virtual disk denotes a container file that is stored on the host file system. A virtual disk can be opened with the software that is developed during this project and stores the actual files. The file extension of the virtual disk is "*.bfs".

References

[1] *Run Length Encoding.*

[2] A. Lempel and J. Ziv. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory 23, 337-343*, 1997.