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 4, 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 files system. The API that a client of this library can use consists of three interfaces that are described in 2.2. 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 performant access to the data inside the disk.

2.1 Requirements

Describe which requirements (and possibly bonus requirements) you have implemented in this part. Give a quick description (1-2 sentences) of each requirement. List the software elements (classes and or functions) that are mainly involved in implementing each requirement.

2.2 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.

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.

2.3 Mocking

For discussing the semantics of the virtual file system and the development of the interfaces explained in 2.2 it was decided to implement a mock that works against the host file system. The mock classes implement all the interfaces and was very helpful in getting 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 which could be developed independently from the real implementation.

2.4 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.

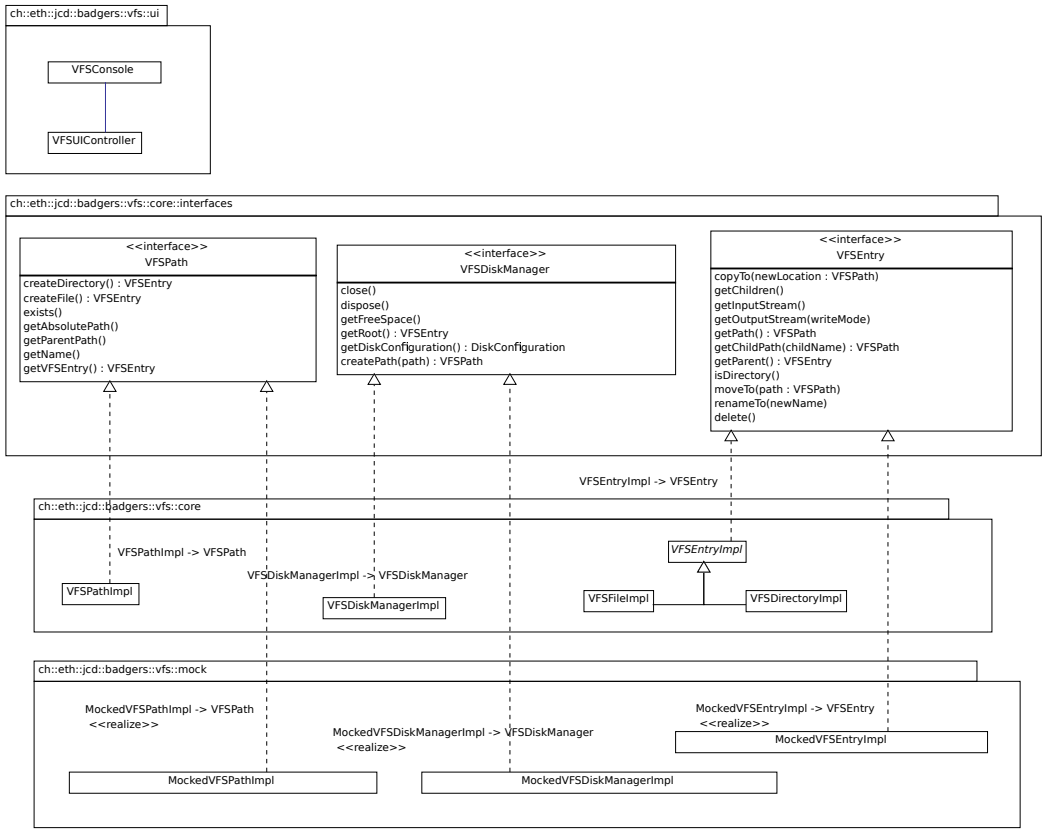

Figure 1: core classes

2.5 The *real* implemenation

Eventually some code that handles a virtual disk was developed and is described in this section.

2.5.1 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.

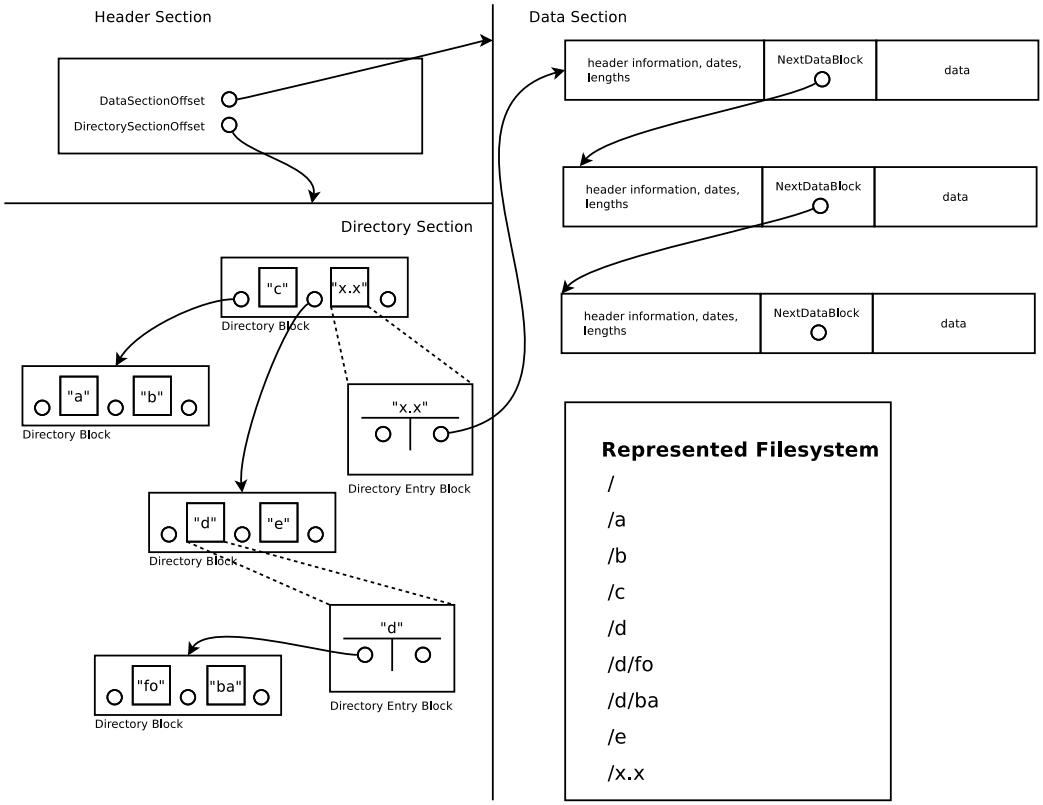

Figure 2: Overview of a disk

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 our 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
Password	xxx bytes	CryptoHash (SHA-whatever) of Password+SaltString

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 an manipulation easy. To each directory belongs a B-Tree structure which lists all contained entries.

Directory Block One Directory Block represents a Node in our B-Tree of order 2.

Name	Length	Description
DirectoryHeader	1 byte	Header information. This header makes it easy to determine whether a DirecotryBlock is in used or not (memory management)
DirectoryEntryBlock1	128 byte	The smaller key inserted into our B-Tree
DirectoryEntryBlock2	128 byte	The bigger key inserted into our 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

Directory Entry Block Represents a single directory or file.

Name	Length	Description
Filename	112 byte	UTF-8 Filename 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 is a file not a directory**

Data Section The data section is split into blocks where each of them is X bytes long. Each block contains some amount of data and points to a subsequent block

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

2.5.2 compression layer

To reduce the data volume within the virtual disk, compression on each file can be enabled. 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 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.

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 filesystem mode. The management mode is entered automatically when starting the command line client. It allows creating and disposing virtual disks. The filesystem 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 1024** creates virtual disk with a maximum quota of 1024 megabytes on the host system. The file may grow up to 1024 megabytes. **TODO**: more parameters are needed (encryption, compression, password if there is encryption)
- **open c:\path\to\disk.bfs** opens filesystem mode for the given virtual disk
- **exit** exits the console program

follwing commands can be used in filesystem mode:

- **ls** lists the contents of the current directory

- **pwd** shows the path to the current directory
- **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 filesystem 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.