Project Report
Group twitchplayseth

Java and C# in depth, Spring 2014

Leonhard Helminger
Marc Gähwiler
Philipp Gamper

April 7, 2014

1 Introduction

This document describes the design and implementation of the *Personal Virtual File System* of group *twitchplayseth*. 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

The *VFS Core* provides a basic interface that allows to create and remove a *VFS container file* and perform certain operations on an existing *VFS container file* that are required to fulfill all requirements mentioned in the next section.

The main API classes are

VDisk
: Provides an interface to create a new *VFS container file*, delete an existing *VFS container file* or perform certain operations on an existing *VFS container file*.

VDirectory
: Represents a directory in the *VFS*. Provides an interface to get the

entries of the directory, add and remove entries and copy or delete the whole directory including it's entries.

VFile
: Represents a file in the *VFS*. Provides an interface to write and read from the file and to copy or delete the file.

VStats
: Provides information about the *VFS*. This includes the size of the *VFS*, used and free space and blocks and the number of directories and files.

2.1 Requirements

In this section all requirements, that were implemented in the first part, are listed. For each requirement there is a short description of the requirement and how it is implemented in the project.

The virtual disk must be stored in a single file in the working directory in the host file system
The idea behind the virtual disk is to store a virtual file system in a single file in an existing file system (which is called the host file system from now on). This requirement is implemented in `VDisk`.

***VFS* must support the creation of a new disk with the specified maximum size at the specified location in the host file system**
A new *VFS* can be created using the `VDisk` class. To create a new *VFS* it is necessary to format the *VFS file* using the `format` method of `VDisk` before it is used for the first time.

***VFS* must support several virtual disks in the host file system**
Once a *VFS* file has been created and formatted, it can be opened by using the constructor of `VDisk`. The implementation allows to create and open an unlimited amount of *VFS files* which each contain their own *VFS*. There is no limitation of how many *VFS files* are opened in parallel at runtime (excluding system limitation like amount of available RAM or physical diskspace).

***VFS* must support disposing of the virtual disk**
A previously created *VFS file* can be deleted with the `discard` method of `VDisk`. It simply removes the *VFS file* from the host's filesystem.

***VFS* must support creating/deleting/renaming directories and files**
All necessary interfaces that handle file/directory creation, deletion and renaming are implemented as methods of `VDisk`, namely `touch`/`mkdir`, `delete` and `rename`.

***VFS* must support navigation: listing of files and folders, and going to a location expressed by a concrete path**
To keep the `VDisk` class stateless in regard to the current file or directory, the `VDisk` class provides a `resolve` method that expects a path in the *VFS* and returns a `VDirectory` instance if the path is valid. To list the files and sub-directory, that a directory contains, the `list` method of `VDisk` can be used in combination in a previously resolved `VDirectory`.

***VFS* must support moving/copying directories and files, including hierarchy**
To move or copy a `VDirectory` or `VFile` in the *VFS* there exist the `move` and `copy` methods in the `VDisk` class. Both methods support both copying/moving a `VDirectory`/`VFile` without renaming it (thus keeping it's original name) or renaming it in the same step. Copying a directory results in a recursive copy of the directory and all it's content.

***VFS* must support importing files and directories from the host file system**
To import a file or directory from the host file system into the *VFS*, the `importFromHost` method of `VDisk` can be used.

***VFS* must support exporting files and directories to the host file system**
To export a `VDirectory` or `VFile` from the *VFS* into the host file system, the `exportToHost` method of `VDisk` can be used.

***VFS* must support querying of free/occupied space in the virtual disk**
To receive information about the *VFS* the `stats` method of `VDisk` can be used. It returns a `VStats` instance that contains all information about the queries *VFS*

2.2 Design

In this section we are describing the general design of the application. Additionally we supply an abstract illustration how the VFS is implemented and what the different layers are as well as a detailed view on the used design patterns and how the file system works.

2.2.1 General Design

Our VFS has three different Layers, illustrated in figure (1). All in the document specified interfaces such as create, rename, move, etc. are implemented in the VDisk class, thus this is the only Object the user should see/use. Additionally to the VUtil and VStats helper classes (see section 2), there are VFile and VDirectory. Objects of those two classes representing a file resp. a directory in the virtual file system. One layer below we use Blocks to save the data on the hard disk. We differentiate between Directory-, File-, Super-, and BitMapBlock whereas the last two are just once on the VirtualDisk and are necessary to store the information about free/used blocks and properties of the file syste like the blocksize or the address of the first root directory.
In the lowest layer there is just one class, called FileManager. Only this class has direct access to the VDisk file, and thus provides all important methods like readInt, readBytes as well as writeInt, writeBytes, writeString, etc.

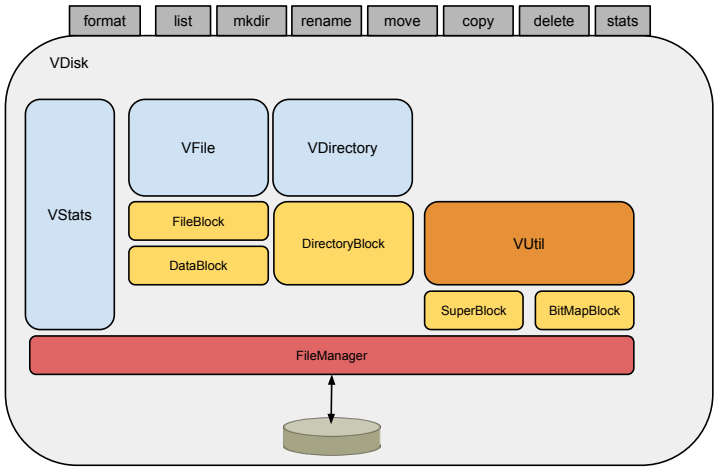

Figure 1: Illustration of VDisk

2.2.2 VFS File Format

A well formated VDisk-file has a fixed header, which consists of a SuperBlock, a BitmapBlock and a DirectoryBlock called RootDirectoryBlock as you can see in figure (2). As soon as one creates and formats a new VDisk, this 3 Blocks will be created.

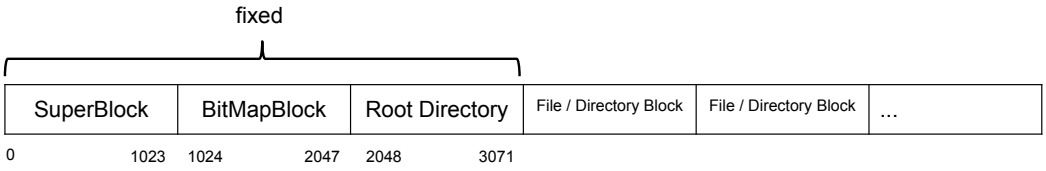

Figure 2: first three Blocks of a VDisk file

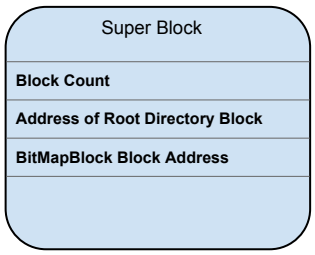

Figure 3: representation of our SuperBlock, which contains all necessary properties of the VFS

Figure (4) illustrates the BitMapBlock. As in the first part of this section described the first three bits are always used by Super-, BitMap- and RootDirectory Block. To avoid huge buffered data in memory we read the data byte after byte. Important to mention is which bit is for which address. Since we read just one byte at the time, we have 8 bits in memory. But, whereas we use *BitSet* to set and unset the bits, we changed the order of the bits. Thats the reason why in figure (4) the arrangement of the bits is in reverse order.

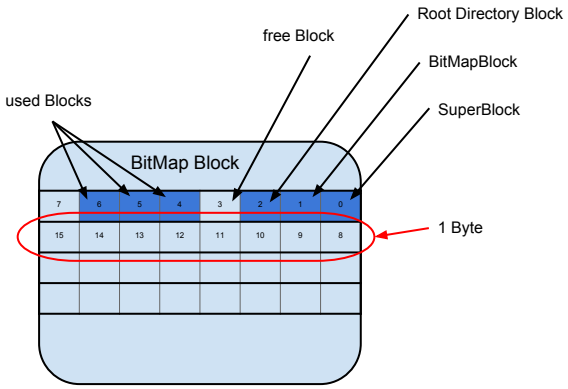

Figure 4: representation of our BitMapBlock which keeps track of the used and free block of the VFS

2.2.3 Design Patterns

3 Quick Start Guide

[optional: This part has to be completed by April 8th.]

If you have a command line interface for your VFS, describe here the commands available (e.g. ls, copy, import).

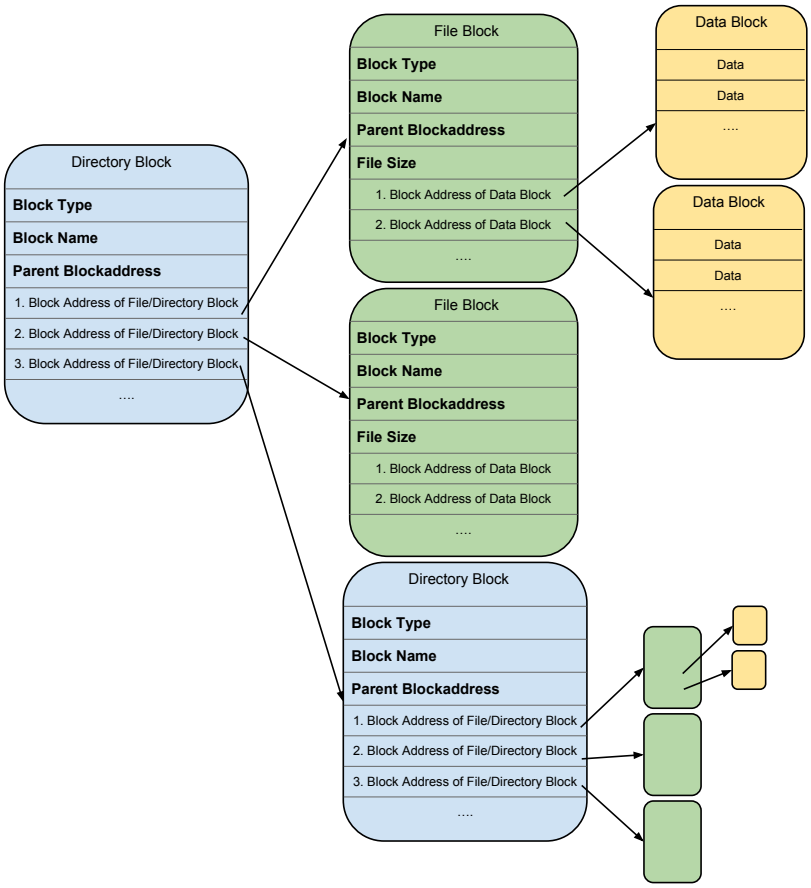

Figure 5: Schematically illustration how the blocks are arranged