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

This section describes the general design of the first part of the project. In addition to this it supplies a detailed overview of the *VFS* implementation including the different abstraction layers it uses, an explenation how the *VFS* is stored on the lowest leven and a list of the design pattern it utilizes.

2.2.1 General Design

Our *VFS* has three different layers, illustrated in figure (1). All previously mentioned interfaces (such as create, rename, move, ...) are implemented in the `VDisk` class which serves as the only interface a user should call directly. Files and directories in the *VFS* are represented with the `VFile` and `VDirectory` classes respectively.

In addition to this there exists a fourth user facing class `VStats` which provide the user with different kinds of information about the *VFS*. Internally the *VFS* uses a helper class `VUtil`, that offers a mid level API for different operations on the *VFS* (like allocating and freeing blocks).

One layer below the file system uses different block classes that serve as the last abstraction above the physical disk level. Most importantly there exist implementations of `DirectoryBlock`, `FileBlock`, `SuperBlock`, and `BitMapBlock`.

The last two blocks exist once for each *VFS* and store specific metadata such as the block count, information about free/used blocks or the block address of the root directory block.

The final level that offers multiple methods to read and write specific types (like integers, byte arrays or UTF-8 strings) from/to the *VFS file* is called `FileManager`.

2.2.2 VFS File Format

A formatted *VFS file* has a fixed-size header, which consists of a super block (represented by the previously mentioned `SuperBlock` class, a bitmap block (`BitMapBlock` and a single directory block (`DirectoryBlock`) called `rootDirectoryBlock`. The layout is visible in figure (2).

2.2.3 Design Patterns

3 Quick Start Guide

To easily see the VFS in action, it comes with a simple console application. The usage of this command line tool is simple:

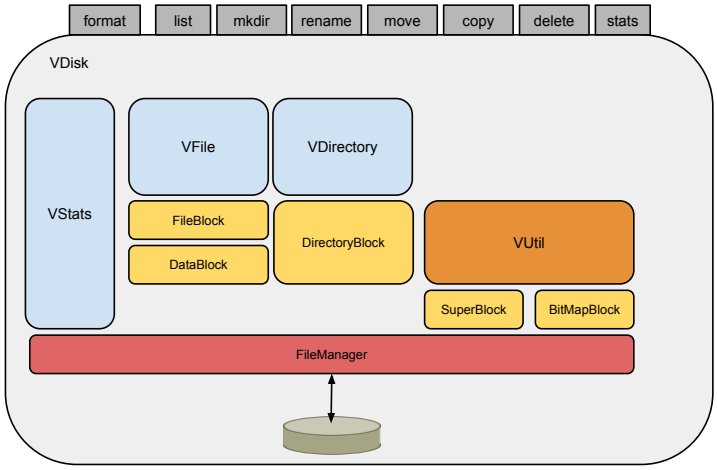

Figure 1: Illustration of VDisk

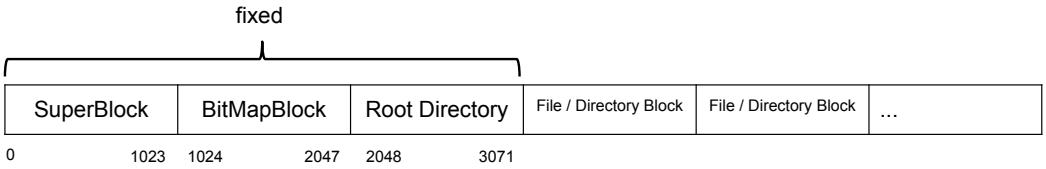

Figure 2: first three Blocks of a VDisk file

1. open a terminal
2. navigate to the VFS root directory
3. launch the console by typing the following command into your prompt

$$> java - jar VFS.jar data/console.vdisk$$

The command above make the console loading an existing VDisk. If you want to create a new one, you have to pass the number of blocks to trigger the console to create a new VDisk. The command therefore is

$$> java - jar VFS.jar data/console.vdisk < number of block to allocate >$$

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

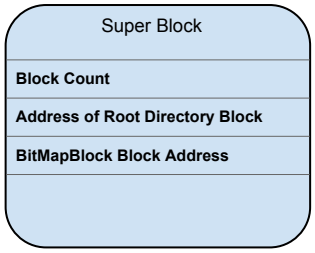

Figure 3: representation of our SuperBlock, which contains all necessary properties of the VFS

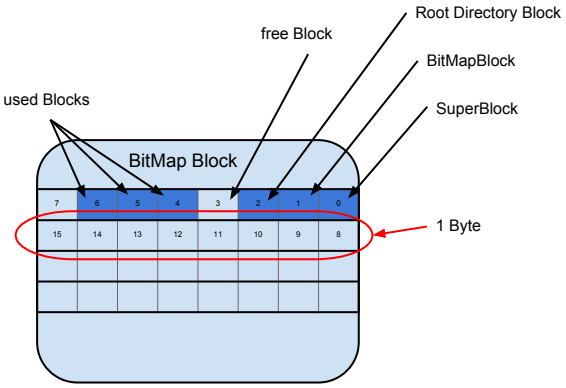

Figure 4: representation of our BitMapBlock which keeps track of the used and free block of the VFS

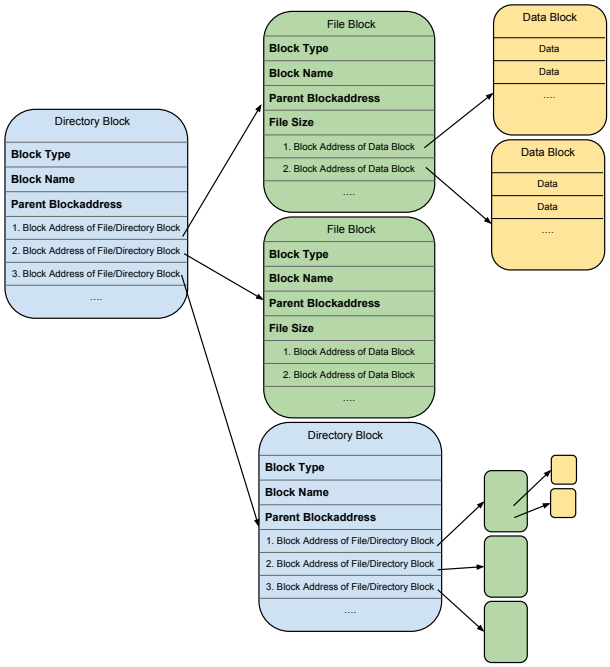

Figure 5: Schematically illustration how the blocks are arranged