Project Report
Group Batman

Java and C# in depth, Spring 2013

Benjamin Steger
Gregor Wegberg

April 8, 2013

1 Introduction

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

Our VFS core consists of two parts and an additional package that implements a Command Line Interface (`.cli`). The first big part is `.vdisk` which contains interfaces for a virtual disk implementation and an implementation of it. The other part is `.io` that contains convenient classes inspired by `java.io.*` to work with the virtual disk in a typical Java I/O manner.

Note: Monospaced text with a leading dot are package names of our project. As our full package name would be to long to write every time. The base for all these package names is: `ch.se.inf.ethz.jcd.batman`.

2.1 Requirements

2.1.1 Virtual Disk in Single File

This requirement is best visible in the implementation of our "create" CLI command (`.cli.command.CreateCommand`) which only needs a single file path to create the disk.

The implementation of the logic behind this command can be viewed at `.vdisk.impl.VirtualDisk.create(String)` or `.vdisk.impl.VirtualDisk .load(String)`. Both methods take a single file path and use it to create/load the virtual disk.

2.1.2 Multiple Virtual Disk Support

Using the CLI a user can load and unload different disks. Therefore it is no problem to have multiple disks on the same system.

As visible inside our virtual disk implementation (`.vdisk.impl.*`) our classes don't have any global state and it's therefore possible to have multiple instances of `VirtualDisk` at the same time. `VDiskFile` (our `java.io.File` inspired class) even checks if the operations are performed on the same disk or not.

2.1.3 Disposing Disk

We've implemented a disposing command called "destroy" for our CLI (`.cli.command.DestroyCommand`). However, as the disk is a single file it can just be deleted. Our "destroy" command has an additional check that examins the file for our magic number before deleting the file.

2.1.4 Creating/Deleting/Renaming Directories and Files

Creating a file for example is handled by `.io.VDiskFile.createNewFile(int)`. A directory can be created by using `.io.VDiskFile.mkdir()`. Deleting a file or a directory can be achieved by using `.io.VDiskFile.delete()`. To rename a file or directory one uses preferably `.io.VDiskFile.renameTo(VDiskFile)`.

Of course all this methods of `.io.VDiskFile` use the underlying interface of the virtual disk and it's content.

2.1.5 Navigation and Listing of Files/Directories

The easiest way to navigate is by using `.io.VDiskFile`. To showcase the navigation we implemented a "change directory" command (`.cli.command .ChangeDirectoryCommand`).

To get the content of a directory one can use `.io.VDiskFile.list()` or `.io.VDiskFile.listFiles()`. Inside our CLI the "list members" command (`.cli.command.ListMembersCommand`) is available.

As before these methods and classes work on the virtual disk interface and just wrap around it for convenience. Our virtual disk implementation has the notion of a directory and a file. It also includes the concept of virtual disk entries belonging to a parent. However, our virtual disk does not enforce a specific path scheme. It just makes sure that the special character "/" is not used inside a name to allow the implementation of a path scheme.

2.1.6 Moving/Copying Directories and Files

Moving a file or directory is implemented as the "move" CLI command (`.cli.command.MoveCommand`). The logic behind this is implemented inside `.io.VDiskFile.renameTo(VDiskFile)` and uses these virtual disk methods to complete the task: `.vdisk.IVirtualDirectory .removeMember(IVirtualDiskEntry)`, `.vdisk.IVirtualDirectory .addMember(IVirtualDiskEntry)` and `.vdisk.IVirtualDiskEntry .setName(String)`.

Copying a file or directory is provided by the "copy" CLI command (`.cli.command.CopyCommand`). Which uses the `.io.VDiskFile .copyTo(VDiskFile)` method to achieve the task.

2.1.7 Importing / Exporting Files and Directories

For this task we provide the user with an "import" (`.cli.command .ImportCommand`) and "export" (`.cli.command.ExportCommand`) CLI command. Those commands use the `.io.util.HostBridge` class which takes care of the needed logic to import/export single files or whole directory structures.

2.1.8 Querying Virtual Disk Information

Our virtual disk implementation provides multiple methods to query for different metrics about the virtual disk. We made the required and most interesting ones available as CLI commands.

The "size" Command (`.cli.command.SizeCommand`) returns the size a file or directory occupies on the disk.

The "query" Command (`.cli.command.QueryCommand`) takes an argument that specifies which virtual disk metric should be returned:

"occupied" returns the used space inside the virtual disk

"free" returns the already allocated but free space inside the virtual disk

"total" returns the size the virtual disk occupies on the host system

2.1.9 Bonus: Compression

As we implemented our own `java.io.InputStream` and `java.io.OutputStream` it is easy to decorate them with compression. To showcase this we implemented `.io.util.GZIPMover` which implements `.io.util.DataMover`. Therefore it's possible to import uncompressed files from the host system and compress them before writing the data onto the virtual disk.

The compression is tested by `.io.HostBridgeTest` inside our `test` directory.

2.1.10 Bonus: Encryption

As with compression, we implemented `.io.util.EncryptedMover` which allows to encrypt the data before writing it onto the virtual disk or decrypt it before exporting it back.

For encryption we use the standard Java streams: `javax.crypto .CipherOutputStream` and `javax.crypto.CipherInputStream`.

The encryption is tested by `.io.HostBridgeTest` inside our `test` directory.

2.1.11 Bonus: Elastic disk

Our disk is always elastic and changes it's size corresponding to the needed space. The corresponding implementation can be found in `.vdisk.impl .VirtualDisk.allocateBlock(long)` `.vdisk.impl.VirtualDisk.extend(long)` and `.vdisk.impl.VirtualDisk.shrink(long)`.

2.1.12 Bonus: Large Data

We provide a test case inside `.io.BigImportExportTest` which is located inside our `slowtests` directory. This test will create a 15 GiB file on the host's disk and import it afterwards. The time it took to import the large file is recorded and printed out into the standard output. On a MacBook Pro (current generation) it took a bit less than three minutes to import the 15 GiB. During the import the Java runtime used always less than 20 MByte of main memory.

As the MacBook Pro has 16 GByte of memory we tested the same test case on a Sony Vaio notebook with 6 GByte of memory. The test was successful. The Java runtime used a bit less than 17 MByte of memory.

2.2 Design

2.2.1 Packages

As already mentioned we use different packages to separate the different parts of our implementation. A short overview follows.

`.vdisk` This package contains interfaces that describe the contract between a virtual disk implementation and it's users. It also contains exceptions used by the interfaces.

`.vdisk.impl` Contains the implementation of our virtual disk and it's file system.

`.vdisk.util` Consists of classes that come in handy while using the `.vdisk` interfaces. Those classes allowed us to follow the DRY principle and prevents duplicate code.

`.io` Contains classes which were inspired by the `java.io` package.

`.io.util` This package contains utility classes providing easy to use interfaces for often needed functionality. At the same time they hide to some extend the logical complexity of some processes. Again, this helps to follow the DRY principle and prevents duplicate code.

`.cli` This package contains the only class with a `main` method (`Main.main`). This class starts the CLI which can be used to explore, create and modify virtual disks.

`.cli.command` Contains the implementation of the different CLI commands.

2.2.2 Virtual Disk Structure

Our virtual disk starts with a super block. This block contains basic information about the disk itself.

The super block is followed by virtual blocks (`.vdisk.IVirtualBlock`). Those blocks may be a data block (`.vdisk.IDataBlock`) or a free block (`.vdisk.IFreeBlock`).

In the following we will talk about positions/addresses inside the virtual disk. These are implemented as `long` values that represent the location of the first byte of the addressed object inside the file representing the virtual disk.

Superblock The super block starts with our magic number. This can be used to check if the file is likely to be a virtual disk.

After the magic number follows the position of the root directory. This is followed by eight bytes which are reserved for future use and 21 addresses pointing to the first elements of segregated free lists.

The super block can be accessed through `.vdisk.IVirtualDisk`.

Virtual Block A virtual block can be a data block or a free block. Every virtual block has a header and a footer, each being 8 bytes long. The first bit of the header/footer indicates if the virtual block is free or not. The rest of the eight byte header and footer contains the length of the block. Our virtual blocks do not have a fixed size.

The minimum size is limited by the header and footer size. The maximum size is limited by the length that can be represented inside the header/footer which is $2^{64-1} - 1$.

The virtual block can be used through `.vdisk.IVirtualBlock`.

Data Block A data block is a special kind of virtual block. Therefore it starts/ends with the virtual block header/footer.

Additionally the data block has it's own header. The header contains an address to the next data block creating a singly linked list. After that follows the size of the data stored inside the data block. This is needed because the data block could be much bigger than the data stored inside it.

The data block can be used through `.vdisk.IDataBlock`.

Free Block A free block is a special kind of virtual block. Therefore it starts/ends with the virtual block header/footer.

Additionally the free block has it's own header. The header contains two addresses, the first pointing to the previous free block (or 0x0 if none) and the second pointing to the next free block (or 0x0 if none). Therefore free blocks create a doubly linked list.

The free block can be used through `.vdisk.IFreeBlock`.

Virtual Disk Space As mentioned data blocks build a singly linked list. So we decided to implement a simple abstraction for them. A virtual disk space consists of a list of data blocks and provides an interface that allows to work with them as if they were one continuous block.

In addition the virtual disk space takes care of expanding and shrinking depending on how much space is needed. It is important to understand that the virtual disk space is a simplification and does not come up inside the virtual disk file itself.

The virtual disk space can be used through `.vdisk.IVirtualDiskSpace`.

Disk Entry A disk entry is an abstract description for objects stored inside the virtual disk. It is the common interface for files and directories. Every disk entry has a directory as its parent. Only the root directory has no parent.

Disk entries contain a reference to the previous and next disk entry, creating a doubly linked list. This list contains entries which are on the same logical level inside the hierarchy. In other words: all elements inside the list have the same parent.

The disk entry can be used through `.vdisk.IVirtualDiskEntry`.

Directory A directory is a specialisation of a disk entry. It describes an object that has a name and contains children of type disk entry.

The metadata used by the directory are stored inside a virtual disk space.

The directory can be used through `.vdisk.IVirtualDirectory`.

File A file is a specialisation of a disk entry. It describes an object that has a name and contains metadata and data.

The metadata and data are stored in separate virtual disk spaces. This allows to implement hard and soft links at a later point.

The file can be used through `.vdisk.IVirtualFile`.

2.2.3 InputStream / OutputStream

An important goal for us was to create an implementation that is easy to understand for Java developers familiar with Java's I/O. Therefore we implemented `.io.VDiskFileInputStream` that implements `java.io.InputStream` and `.io.VDiskFileOutputStream` that implements `java.io.OutputStream`. Because of this design choice, it is very easy to encrypt, compress or transform the data in any possible way while reading or writing data.

A good example of such stream decorations are our compression and encryption implementations, which can be found at `.io.util.GZIPMover`

and `.io.util.EncryptedMover`. The implementation of both classes is, as you can see, very simple and readable.

3 VFS Browser

[This section has to be completed by April 22nd.]

Give a short (1-2 paragraphs) description of what VFS Browser is.

3.1 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.

3.2 Design

Give an overview of the design of this part and describe in general terms how the implementation works. You can mention design patterns used, class diagrams, definition of custom file formats, network protocols, or anything else that helps understand the implementation.

3.3 Integration

If you had to change the design or API of the previous part, describe the changes and the reasons for each change here.

4 Synchronization Server

[This section has to be completed by May 13th.]

Give a short (1-2 paragraphs) description of what VFS Browser is.

4.1 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.

4.2 Design

Give an overview of the design of this part and describe in general terms how the implementation works. You can mention design patterns used, class

diagrams, definition of custom file formats, network protocols, or anything else that helps understand the implementation.

4.3 Integration

If you had to change the design or API of the previous part, describe the changes and the reasons for each change here.

5 Quick Start Guide

5.1 How to run the CLI

To run the Command Line Interface just run `.cli.Main.java`. A host system path is a path that is valid for the host system. A virtual disk path is a valid path for the virtual disk.

5.2 Commands

Following a list of available commands with a description what they do.

stop Stops the CLI and shuts down the application. Example: `stop`

create Takes a valid file path (host system) where it will create a new, empty virtual disk. Example: `create /home/user/test.vdisk`

destroy Takes a valid file path (host system) where a virtual disk is located. After successfully checking for the magic number it will delete the disk from the host's system. Example: `destroy /home/user/test.vdisk`

load Takes a valid file path (host system) where a virtual disk is located. Will load the given file as a virtual disk. Example: `load /home/user/test.vdisk`

unload Unloads the currently loaded virtual disk. Example: `unload`

ls Takes an *otional* argument with an absolute or relative path (virtual disk). Will print out all children (files and directories) inside the given path or for the current location. Example `ls /`

mkdir Takes an absolute or relative path (virtual disk) and creates a directory at the given location. It is possible to pass "-p" as the first argument and a path as the second one, in this case it will create all needed directories to satisfy the given path.
Example: `mkdir -p /test/some/stuff` will create the directories /test, /test/some and /test/some/stuff.

cd Takes an absolute or relative path (virtual disk) as its argument. Changes the current directory to the given path. Example: `cd /test/some`

size Takes an *optional* relative or absolute path as its argument. Returns the size of the object represented by the given path (or the size of the current one, if no path provided). Example: `size /test`

query Takes "occupied", "free" or "total" as its first argument. Example: `query total`

- **occupied** Shows the amount of used space inside the virtual disk
- **free** Shows the amount of not used space inside the virtual disk
- **total** Shows the size of the disk on the host system

delete Takes an absolute or relative path to a directory or file inside the virtual disk. Deletes the given object. Example: `delete /test`

move Takes two absolute or relative paths (virtual disk). The first being the source and the second being the target. Moves the source file or directory to the given target. Example: `move /test/some/file /test/` would move the file "/test/some/file" into the directory "/test/" (resulting in a new file "/test/file")

copy Takes two absolute or relative paths (virtual disk). The first being the source and the second being the target. Creates a copy of the source at the given target location. Example: `copy /test/some/file /test/file`

import Takes two absolute or relative paths. The first one being a source (host system) and the second one being a target (virtual disk). Will import the file or directory from the host system into the given target inside the virtual disk. Example: `import /home/user/movies/awesome.avi /awesome.avi`

export Takes two absolute or relative paths. The first one being a source (virtual disk) and the second one being a target (host system). Exports the given source file or directory into the given target. Example: `export /awesome.avi /home/user/movies/`

[This part has to be completed by May 13th.]

Describe how to realize the following use case with your system. Describe the steps involved and how to perform each action (e.g. command line executions and arguments, menu entries, keyboard shortcuts, screenshots). The use case is the following:

1. *Start synchronization server on localhost.*
2. *Create account on synchronization server.*
3. *Create two VFS disks (on the same machine) and link them to the new account.*
4. *Import a directory (recursively) from the host file system into Disk 1.*
5. *Dispose Disk 1 after the synchronization finished.*
6. *Export the directory (recursively) from Disk 2 into the host file system.*
7. *Stop synchronization server.*