

Project Report

Group jcd_dumplings

Java and C# in depth, Spring 2013

Elmer Lukas
Nussbaumer Ivo

May 13, 2013

1 Introduction

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

2.1 Code Coverage

To measure the code coverage of the VFSBase and VFSBaseTests correctly, the DiskServiceReference has to be excluded from the coverage, because this code is generated automatically and therefore is irrelevant for the test coverage (WCF service reference). Figure 1 shows how this is done for the VFSBase project.

The code coverage of the code was at 94% on Monday, 2013-05-13 at 4.46 pm, see Figure 2.

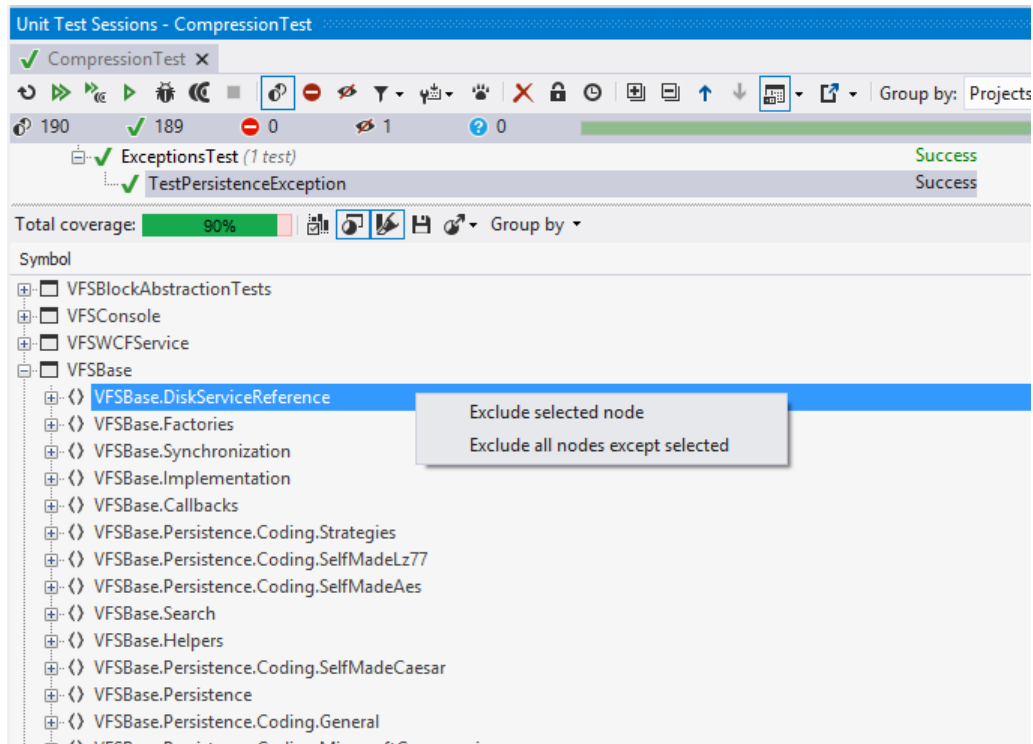


Figure 1: Exclude Tests

2.2 Static Code Analysis

The static code analysis was run with the JCD C# ruleset. The Visual Studio didn't recognise that some methods in the view model are being used by the view in the xaml-files, which resulted in some warnings in the analysis, which were suppressed in the suppression file or directly in the code. Beside this few warnings there were none.

2.3 Code Metrics

Microsoft Visual Studio 2012 provides a functionality to measure the quality of the code¹. To ensure good code quality and especially good maintainability, code metrics for the solutions were generated. The results are displayed in Figure 3.

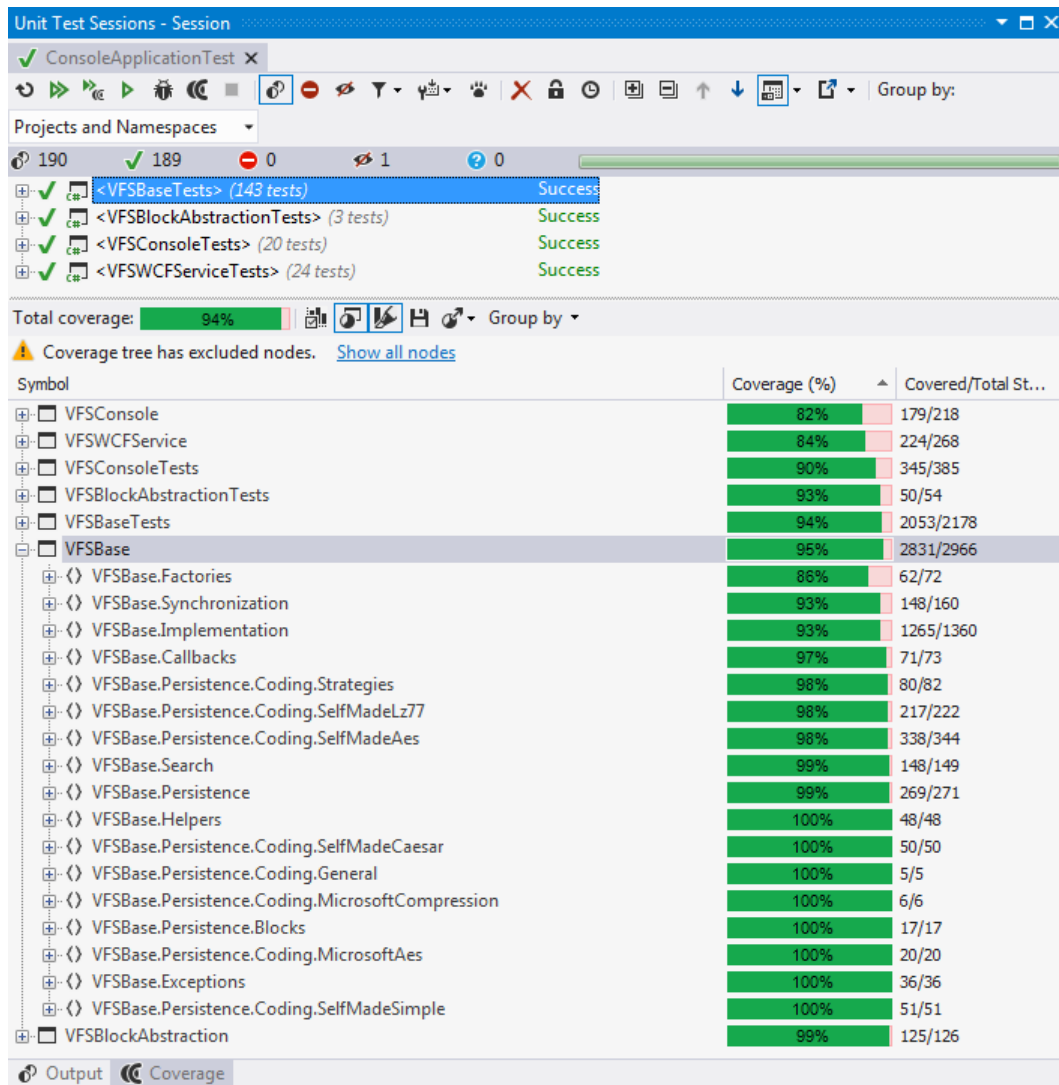


Figure 2: Code Coverage

2.4 Code Statistics

For a quick overview, some code statistics were extracted, see Figure 4.

3 VFS Base

The VFSBase is the base of the file system. It implements the functionality to creating, renaming, deleting, importing, exporting, files and folders. It also

¹<http://msdn.microsoft.com/en-us/library/bb385914.aspx>

Hierarchy	Maintain...	Cyclomat...	Depth of I...	Class Co...	Lines of Code
Startup (Debug)	83	7	3	13	14
VFSBase (Debug)	85	978	3	143	1'891
VFSBaseTests (Debug)	75	508	2	101	1'571
VFSBlockAbstraction (Debug)	85	35	2	15	74
VFSBlockAbstractionTests (Debug)	79	14	1	17	36
VFSBrowser (Debug)	90	587	9	163	945
VFSConsole (Debug)	85	69	1	36	142
VFSConsoleTests (Debug)	76	91	1	35	285
VFSWCContracts (Debug)	96	43	1	13	32
VFSWCService (Debug)	80	84	2	45	205
VFSWCServiceTests (Debug)	71	64	1	25	198

Figure 3: Code Metrics

LoC Total	5393
LoC Implementation (including views)	3303
LoC Implementation (excluding views)	2358
LoC Views (excluding WPF XML)	945
LoC Tests	2090
$\frac{LoC Tests}{LoC Implementation (excluding views)}$	approx. 0.89

Figure 4: Code statistics

provides functionality to query disk information like free space or occupied space. To store the files, there are several compression and encryption options available. Additionally, it creates a new version for every action on the file system, and thus provides a history for the file system. The file system also handles big data by storing all the things on the disk instead of storing them in the RAM before writing it to the disk.

3.1 Requirements

3.1.1 Introduction: About the FileSystem and the FileSystem-TextManipulator

Most of the features discussed in this section are about the FileSystem and the FileSystemTextManipulator. The first question might be: the FileSystem and the FileSystemTextManipulator are very similar and provide nearly the same interface, why are the separate? The answer to this question is: separation of concerns and encapsulation.

The `FileSystemTextManipulator` provides a *beautiful* interface which is easy to use and to understand. It wraps the `FileSystem` and abstracts away certain tasks, like converting a given path to an object to the object itself (e.g. it converts `"/a/b/c"` to the directory object `"c"`). Also, using the `FileSystemTextManipulator`, only a certain part of the actual file system is extracted. This way, the file system can handle more files than the RAM size of the current client is. Also, the `FileSystemTextManipulator` uses some structures only known to the `VFSBase` assembly, and it encapsulates all the internal objects by only returning values that should be available publicly.

The `FileSystem` is the core of the file system functionality. It implements all the internal structure to store the files and folders to the disk. Finally, there are the `ThreadSafeFileSystem` and the `ThreadSafeFileSystemTextManipulator` classes. They also originate from the separation of concerns design principle and provide a thread safe implementation of the `FileSystem` and the `FileSystemTextManipulator` respectively. The thread safety is guaranteed through a `ReaderWriterLock`², which allows multiple concurrent reads, and exclusive writes.

In this section, if the referenced classes are `FileSystem` and `FileSystemTextManipulator`, then of course the `ThreadSafeFileSystemTextManipulator` and the `ThreadSafeFileSystem` are relevant too. Furthermore, if there is a method `FileSystem.SomeMethod` in the referenced methods, then the Methods `ThreadSafeFileSystem.SomeMethod` `FileSystemTextManipulator.SomeMethod` `ThreadSafeFileSystemTextManipulator.SomeMethod` are not mentioned, but of course, they are relevant too.

3.1.2 Store data in single file

All the data is stored in a single file and in blocks. There are logical block numbers, which serve as pointers. All blocks are, once written, immutable to implement the file history.

Classes: `BlockManipulator`, `BlockAllocation`

Methods: `BlockManipulator.WriteBlock` `BlockManipulator.ReadBlock`, `BlockAllocation.Allocate`

3.1.3 Creation of new disk

The file system supports the creation of new disks at a specified location of the host file system. Because the disk size grows dynamically (elastic disk),

²<http://msdn.microsoft.com/en-us/library/system.threading.readerwriterlockslim.aspx>

no maximum size has to be specified³.

Classes: FileSystemFactory, FileSystemTextManipulatorFactory

Methods: FileSystemTextManipulatorFactory.Create, FileSystemTextManipulatorFactory.Open, FileSystemFactory.Create, FileSystemFactory.Open

3.1.4 Several virtual disks in the host file system

Multiple virtual disks can be created on the host file system.

3.1.5 Disposing of the virtual disk

To dispose a virtual disk, the virtual file system can be closed. After closing a file system, the virtual disk can be deleted, as it is a normal file on the host system.

Classes: FileSystem, FileSystemTextManipulator

Methods: FileSystem.Dispose

3.1.6 Creating, deleting, renaming directories and files

The user can create, delete and rename directories. Files can be imported, and then be renamed or deleted. Classes: FileSystem, FileSystemTextManipulator

Methods: FileSystem.CreateFolder, FileSystem.Rename, FileSystem.Delete

3.1.7 Listing and navigation

The files and folders can be listed and navigated through. The file system supports going to a location expressed by a concrete path.

Classes: FileSystem, FileSystemTextManipulator

Methods: FileSystem.List, FileSystem.Folders, FileSystem.Files, FileSystem.List

3.1.8 Moving / copying directories and files

The files and directories can be copied. Because of the history implementation, some performance optimizations are possible. Thus, the copy time does not depend on the directory or file size, but solely on the directory depth.

Classes: FileSystem, FileSystemTextManipulator

Methods: FileSystem.Move, FileSystem.Copy

³<https://piazza.com/class/spring2013/2520284001/26>

3.1.9 Import and export

The VFS supports importing and exporting files and directories from and to the host system respectively. Of course, the import and export functionality is recursive for directories.

Classes: `FileSystem`, `FileSystemTextManipulator`

Methods: `FileSystem.Import`, `FileSystem.Export`

3.1.10 Querying of free and occupied space

The file system allows to query for free and occupied space. The free space corresponds to the free space on the host file system. In the GUI, the view to the free and occupied space can be found in the main menu under "Info"/"Disk Info".

Classes: `FileSystemOptions`

Methods: `FileSystemOptions.DiskOccupied`, `FileSystemOptions.DiskFree`

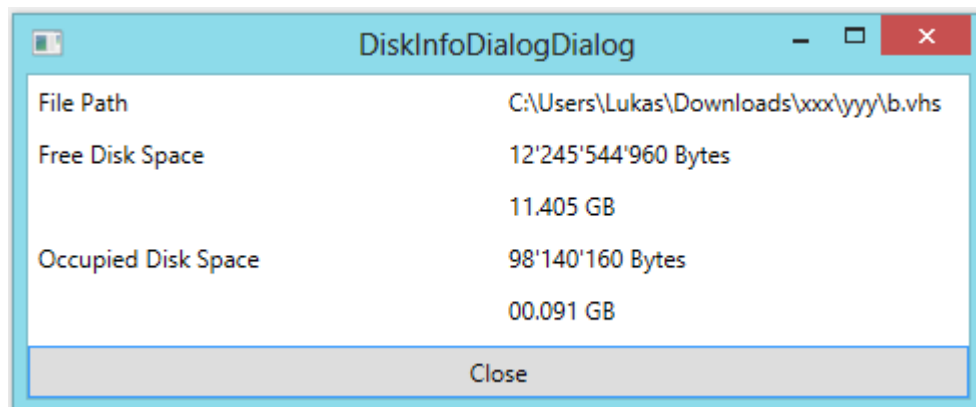


Figure 5: Free and occupied space

3.1.11 Compression with 3rd party library

There are multiple compression types available. One of them is a third party implementation of the Microsoft deflate stream⁴.

Patterns: Decorator pattern, Null object pattern (for no encryption)

Classes: `MicrosoftStreamCompressionStrategy`, `DeflateStream`

Methods: `DecorateToVFS`, `DecorateToHost`

⁴<http://msdn.microsoft.com/en-us/library/system.io.compression.deflatestream.aspx>

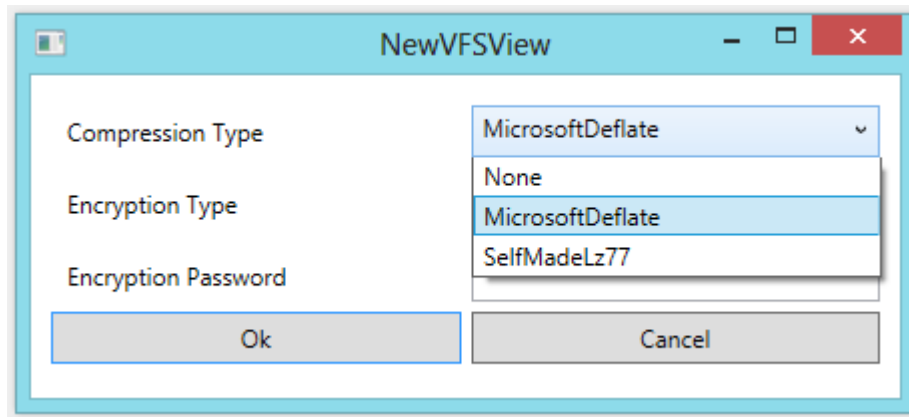


Figure 6: Compression types

3.1.12 Compression implemented by hand

Compression by hand is implemented. The basic idea behind the compression is the LZ77 algorithm⁵.

Due to the fact that the implementation is in C#, the implementation is slow compared to a 3rd party library. Also, no special optimizations were made. Therefore, it is recommended to use the 3rd party libraries for optimal performance.

Patterns: Decorator pattern, Null object pattern (for no encryption)

Classes: SelfMadeLz77Stream, Lz77Triple, Lz77Constants, CircularBuffer, SelfMadeLz77StreamCompressionStrategy

Methods: SelfMadeLz77Stream.Read, SelfMadeLz77Stream.Write

3.1.13 Encryption with 3rd party library

There are multiple encryption types available. One of them is a third party implementation of the Microsoft Rijndael, the AES standard⁶.

Additionally, the encryption needs a user defined password. This password is then used to calculate the encryption / decryption key. This means, that a virtual disk without this password cannot be decrypted.

Due to the fact that the implementation is in C#, the implementation is slow compared to a 3rd party library. Also, no special optimizations were made. Therefore, it is recommended to use the 3rd party libraries for optimal performance. Furthermore, the Microsoft implementation of AES is likely

⁵http://www.ieeeeghn.org/wiki/index.php/Milestones:Lempel-Ziv_Data_Compression_Algorithm,_1977

⁶<http://msdn.microsoft.com/en-us/library/system.security.cryptography.rijndael.aspx>

better tested than this self made implementation and thus should be preferred for security reasons.

Patterns: Decorator pattern, Null object pattern (for no encryption)

Classes: Rijndael

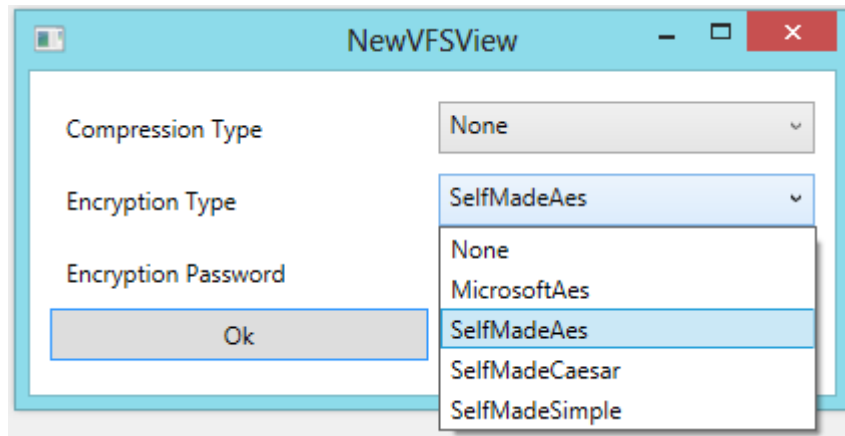


Figure 7: Encryption types

3.1.14 Encryption implemented by hand

Encryption was also implemented by hand. There is a simple encryption and a caesar encryption which are fast, but very weak. Therefore, the AES-256 with CBC⁷ was implemented.

Patterns: Decorator pattern, Null object pattern (for no encryption)

Additionally, the encryption needs a user defined password. This password is then used to calculate the encryption / decryption key. This means, that a virtual disk without this password cannot be decrypted.

Classes: SelfMadeAes256Cryptor, AesHelperMethods, AesConstants

Methods: SelfMadeAes256Cryptor.EncryptBlocks, SelfMadeAes256Cryptor.DecryptBlocks

3.1.15 Elastic disk

The elastic disk is implemented. The file system only allocates disk space when it is required and there are no "practical" limit for the file size or the disk size. It probably will be limited either by the host file system or the hardware itself. The file system is designed in such a way that it does not need to reorganize itself to be fast (known as defragmentation).

⁷part of the AES standard, <http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf>

Classes: BlockAllocation
 Methods: BlockAllocation.Allocate

3.1.16 Large data

As mentioned above, large data is supported. File contents are written directly to the disk, and does not store whole files in memory. Therefore, the file system does not need much RAM, but can handle huge files tough.

Classes: BlockManipulator, FileSystem, FileSystemTextManipulator
 Methods: BlockManipulator.WriteBlock, BlockManipulator.ReadBlock, FileSystem.Import, FileSystem.Export

3.2 Design

3.3 File Format

The design of the file format is related to the Unix File System (UFS)⁸ and the New Technology File System (NTFS)⁹. The file is partitioned into two sections:

- one super block
- many normal blocks

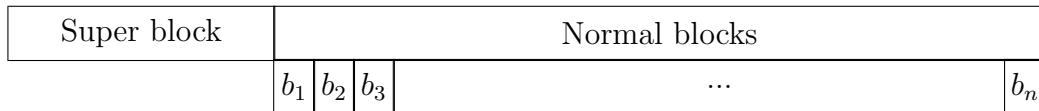


Figure 8: Partitions of the virtual disk with n normal blocks.

3.3.1 Super Block

The super block has a constant size of 32 KB. Although most of the space is not needed yet, it can be used to save meta information about the file system:

- block number of the latest root node
- used block amount and block size
 $(block_amount * block_size + super_block_size = disk_size)$

⁸http://en.wikipedia.org/wiki/Unix_File_System

⁹<http://en.wikipedia.org/wiki/NTFS>

- version of the current file (for file history)
- version of the file format
- ID of the virtual disk (for synchronization)
- selected compression algorithm
- file encryption algorithm(s)
- encrypted key for file encryption
- hashed seed/password for file encryption
- etc.

3.3.2 Normal Blocks

The normal blocks consist of many blocks of a fixed block size (e.g. 8KB). The block size is stored in the file system meta information, which is stored in the super block. The block size is the same for every block and cannot be changed after creating the file. Though the block size could be configured at creation time of the disk, which is not available in the GUI, because it was not a requirement.

Normal Block Types

One normal block can be of one of the following types:

- Index Node
- File Node
- Folder Node
- Indirect Node
- File Content Node

Index Node

The Index Node contains the file type (file or directory, 1 byte) and the name (max. 255 bytes, can contain any character excluding '/' and '\0') of one object. The next 64 bytes contain the block count of the blocks, which are referenced (through the Indirection Nodes) by this Index node. The next 64 bytes contain the indirection node number, which is a reference to the indirection node. The next 64 bytes are reserved for the version of the object (important for the history implementation). The next 64 bytes are reserved for the predecessor object reference (also for the history implementation). If the Index Node is a file, then the next 32 bytes describe the length of the last block, so the file system knows which data of the last block is relevant. If the Index Node is a folder, then the next 64 bytes describe the amount of blocks used. This is important for the root folder and the synchronization.

The latest root node can be at any position. The reference to the root node is stored in the options, which are stored in the master block size. The root node is a special folder without a name. The latest root node always has a back reference to the previous root node, unless it is the first root node with version 0.

The block b_1 is reserved for other meta information, also organized as a folder. Until now, this block is unused, but it might be useful in the future.

If the file type is a directory, then the Index Node contains references to other Index Nodes (files and directories). These are considered sub-files and sub-directories.

If the file type is a file, then the Index Node contains references to File Content Nodes. These File Content Nodes must be accessible in the same order they were put into the file system. Because of the concept of the Indirection Nodes, random access is possible. Therefore, seeking to any block in the file takes $\mathcal{O}(1)$, which is very good, especially if only a very small part of a huge file is needed.

The File Content Node contains only file content. This content can be encrypted and/or compressed. The last content node of a file is filled with padding, so block numbers can identify blocks in the file system. If only very small files are stored, it is recommended to reduce the block size.

The Indirect Nodes are used to address multiple blocks. There are three levels of Indirect Nodes. First, the one which is referenced by the Index

Node (File Node or Folder Node), is the 1st level Indirect Node. This one references to multiple 2nd level indirect nodes. Any of the 2nd level Indirect Nodes references to multiple 3rd Indirect Nodes. And every 3rd Indirect Node references to multiple blocks, which are of type Index Node. For folders, these addressed blocks are File Nodes or Folder Nodes. For files, these addressed blocks are File Content Nodes.

3.4 Formulas

$$\text{block reference size} = 64 \text{ Byte} = \text{sizeof}(\text{long}) \quad (1)$$

$$\text{references per IndirectNode} = \frac{\text{block size}}{\text{block reference size}} \quad (2)$$

$$\text{maximal file size} = \left(\frac{\text{block size}}{\text{block reference size}} \right)^3 \quad (3)$$

$$\text{maximal amount of blocks} = 2^{\text{block reference size}} \quad (4)$$

3.5 Implementation

Most of the basic file system implementation can be found in the VFSBase project. The VFSBlockAbstraction abstracts all the block operations, so the VFSBase can use the block abstraction and write any data to any block which can be addressed with a block number.

Therefore the VFSBase depends on the VFSBlockAbstraction. Additionally, the VFSBase depends on the VFSWCFCContracts project, where the interface of the WCF disk service and the Data Transfer Objects (DTOs) are defined.

4 VFS Browser

The VFS Browser is the GUI (Graphical User Interface) for the VFS Core. With the browser it is possible to browse through existing or new file systems and perform the usual file system actions like delete, copy, move files or folders. Furthermore it serves as a client to the synchronization server which provides the user with the possibility to synchronize the file system with a remote server.

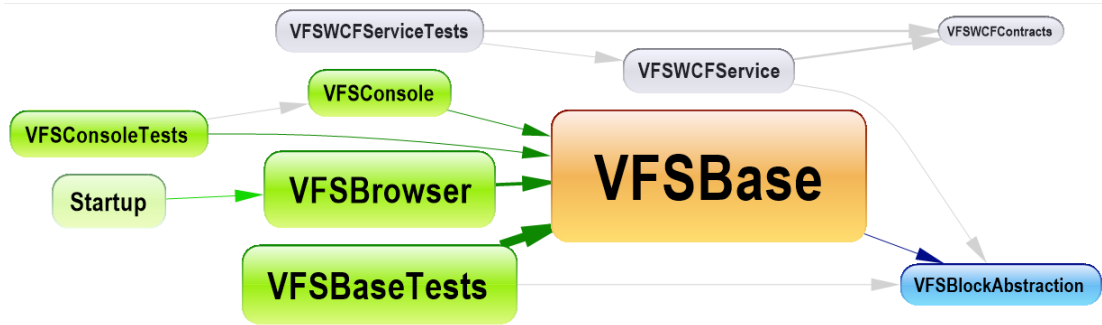


Figure 9: Dependencies of the VFSBase

4.1 Requirements

4.1.1 Platform

The browser is implemented as a desktop application and is written in C# and uses WPF (Windows Presentation Foundation).

Classes: *all classes in the VFSBrowser project*

4.1.2 Operations

The browser supports all operations of the VFS core.

- Copy / Move / Rename / Delete
- Create folder
- Import / Export
- Create / Open / Close file system

Classes: *MainViewModel*

Methods: *Copy, Move, Rename, Delete, NewFolder, ImportFile, ImportFolder, Export, OpenVfs, NewVfs, CloseVfs*

4.1.3 Selection

Selecting a file or a folder is done by clicking with the left mouse button on the item in the grid. Multiple selection of files or folders is possible by pressing and holding down the Control- or Shift-Key and clicking with the left mouse button or with the Up- and Down-Key.

Classes: *MainWindow.xaml*

View Items: *ItemsGrid*

4.1.4 Keyboard navigation

Most of the command are also executable by keyboard. All the other actions are accessible in the application menu, which can be opened by pressing the Alt-Key.

The keyboard commands which are available are listed in Figure 10

Classes: *MainWindow.xaml*

View Items: *ItemsGrid*

Command	Action
Left / Back	Go to parent folder
Right / Enter	Go to child folder / open file
Up / Down	Select previous / next item
Delete	Delete selected items
F2	Rename selected item
Ctrl + C	Copy selected items
Ctrl + X	Cut selected items
Ctrl + V	Paste items

Figure 10: Keyboard short cuts

4.1.5 Mouse navigation

All the commands which are accessible by keyboard are also available with mouse navigation. Most of them are found in the application menu or the context menu of the grid. Opening a folder or a file is achieved by double clicking the item with the left mouse button and to navigate to the parent folder, the folder ".." has to be double clicked.

Classes: *MainWindow.xaml*

View Items: *ItemsGrid*

4.1.6 Search

The line beneath the application menu (see Figure 12) offers all the necessary tools to search for files or folder in the file system. In the drop down menu of the split button are the options listed in Figure 11 adjustable.

Classes: *MainWindow.xaml*, *MainViewModel*,

Methods: *Search*, *CancelSearch*

Option	Description
Case Sensitive	If checked, the search is case sensitive, otherwise insensitive.
Global	If checked, the search is always started from the root folder, otherwise it is restricted to the current folder.
Recursive	If checked, all results in the sub folders are also found, otherwise just the items in the current folder.

Figure 11: Search options

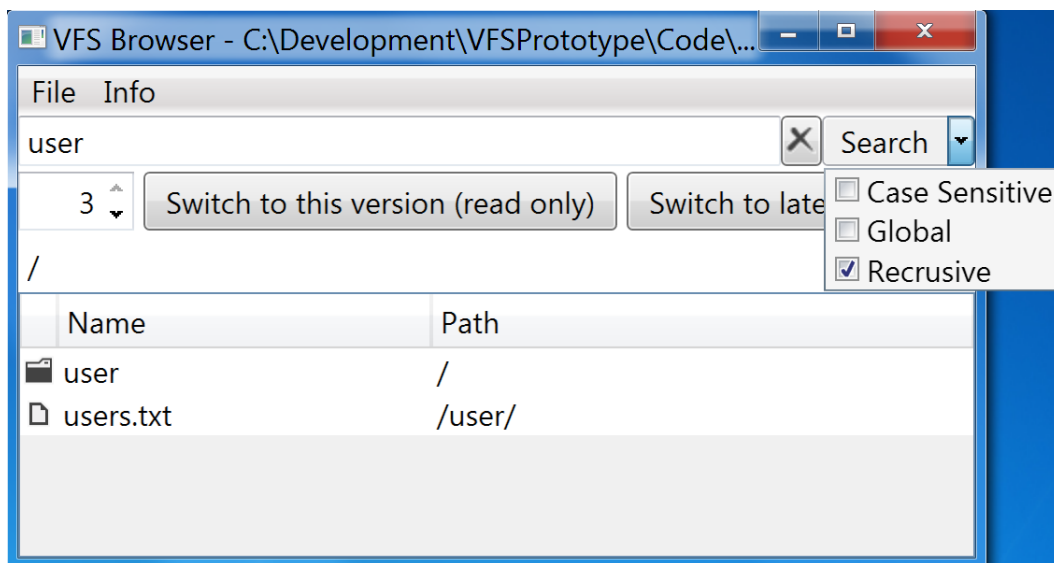


Figure 12: Search functionality

4.2 Bonus features

4.2.1 Responsive UI

The VFS Browser includes a progress view (see Figure 13), which is shown during long-running operations (i.e. import, copy). These operations can be cancelled by clicking the cancel button on the progress view.

Classes: *OperationProgressView.xaml*, *OperationProgressViewModel*, *CopyCallbacks*, *ImportCallback*, *ExportCallbacks*, *CallbacksBase*, *FileSystemTextManipulator*

Methods: *Import*, *Export*, *Copy*

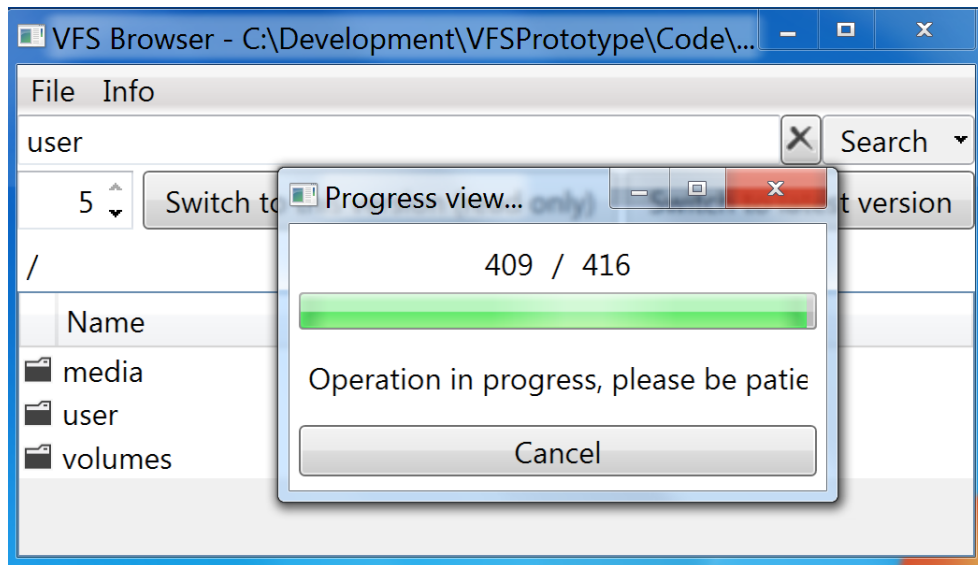


Figure 13: Progress view

4.2.2 Advanced search

Searching is implemented on a suffix basis. Therefore it is possible to find files by searching after a substring of the file name, which makes searching easier, because the knowledge of the full file name isn't required.

Classes: *SearchService*

Methods: *Search*

4.2.3 Operation progress

The progress of long lasting operations like copying and importing is displayed in a separate window (see Figure 13). The view shows the user the number of files involved in the operation and how many are already processed. This is displayed with text and also with a progress bar.

Classes: *OperationProgressView.xaml*, *OperationProgressViewModel*, *CopyCallbacks*, *ImportCallback*, *ExportCallbacks*, *CallbacksBase*, *FileSystemTextManipulator*

Methods: *Import*, *Export*, *Copy*

4.2.4 Drag-and-Drop

Drag and Drop is implemented for importing files. It is possible to drag files and folders from the explorer into the grid in the VFSBrowser. This copies them into the virtual file system.

Classes: *MainWindow.xaml, MainViewModel, DropBehaviour*
Methods: *Drop*

4.2.5 Efficient search

The efficiency of the file search was improved, by building up a index with the file names. The index stores all file names and its suffixes and links them with the corresponding files. So it is possible to locate the files in $O(n)$ time with n =keyword length.

Classes: *IndexService, SuffixTree, SuffixTreeNode, SearchService, FileSystemTextManipulator*
Methods: *Search, AddToIndex, Index*

4.3 Design

4.3.1 Design Patterns

The GUI is implemented with the MVVM design pattern. The main advantages of this pattern is that the view is almost completely separated from the view model, except the data bindings. Events from the view are bound with commands in the view model, and therefore no code-behind is necessary. Furthermore manual UI test aren't needed, just the view model has to be tested, assuming that the binding to the view works.

4.3.2 Search Index

The search index is stored in a suffix tree. For each suffix of all file and folder names a sub tree is generated, and on each node the path to the file or folder is added. With this data structure it is possible to search for substring of file names in $O(n)$ time, with n =length of keyword. The index is generated in a separate thread after opening a file system, and is kept up-to-date after each modification.

For example if the folder *user* is added to the index, the tree in Figure 14 is build, and the path to the folder is added to each node. After adding the folder *usa* to the index, the tree would look like Figure 15, and the path to the folder is added to the bold nodes.

The search algorithm now just follows the letters of the keyword down the tree and returns the list attached to the node were it ends up. After that it has to check if the nodes are in the folder, which is specified in the search options, and it is also checked if the item exists in the current version.

If the search is case insensitive, the algorithm follows on each node two path, one with the next lower case and one with the next upper case letter.

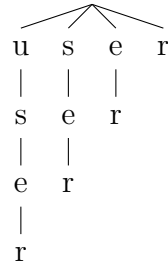


Figure 14: Search index *user*

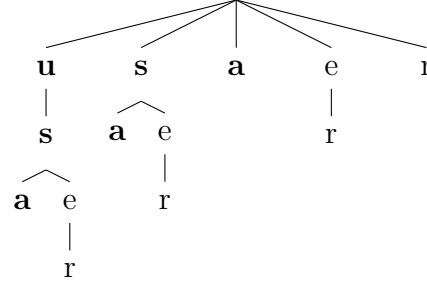


Figure 15: Search index *user, usa*

4.4 Integration

4.4.1 Search Index

For implementing the search index, the index service had to be integrated into the *FileSystemTextManipulator*. So that after adding, copying or importing files, the file names of the new or changed files would be added to the index.

5 Synchronization Server

The synchronization server propagates changes from one machine to another using a network connection. The management of the distributed VFSs is made on an account basis, i.e. in order to use the server, a user must have an account and link a virtual disk to this account. Therefore the server allows user to register and login after successful registration. Once the user enters the online mode (see GUI part) the currently open disk is synchronized to the server automatically. After initial synchronization, every synchronized disk gets a unique ID, so the disk can be identified. A user can have multiple disks, while a disk belongs to a single user. Unlinked disks are not synchronized. There is an automatic conflict recognition so the user can resolve conflicts by rolling back to a specific version, which is not conflicted, and then restart the synchronization again.

Technically the server is implemented as a WCF ¹⁰ service and it allows multiple parallel connections. As an optional requirement, automatic persistence was implemented using SQLite. The usage of SQLite for this optional

¹⁰<http://msdn.microsoft.com/en-us/library/dd456779.aspx>

requirement was allowed by Alexey Kolesnichenko ¹¹.

5.1 Requirements

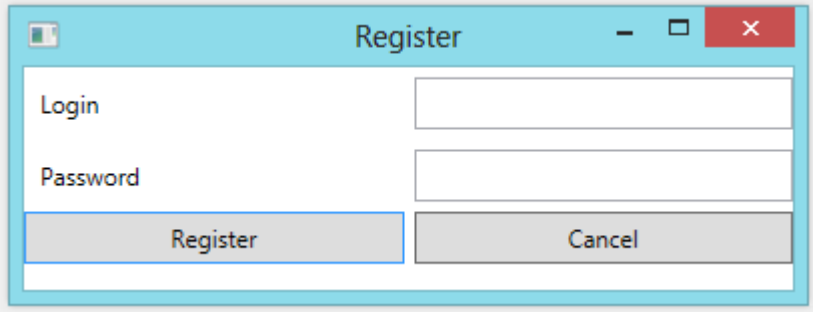
Describe which requirements (and possibly bonus requirements) you have implemented in this part. Give a quick description (1-2 sentences) of each requirement. List the software elements (classes and or functions) that are mainly involved in implementing each requirement.

5.1.1 Registration and login

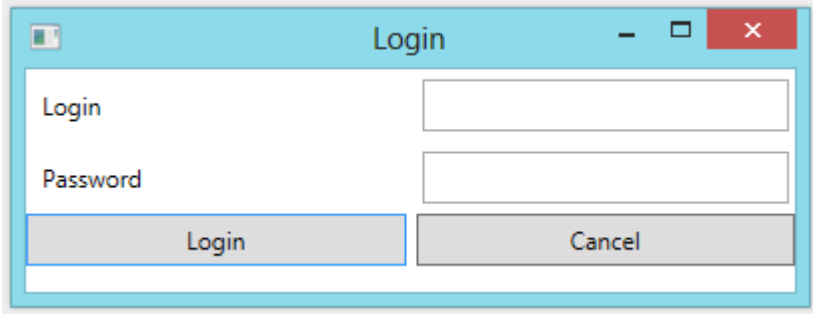
There are two forms for registration and login.

Classes: MainViewModel, DiskServiceClient

Commands: LoginCommand, LogoutCommand

A screenshot of a Windows-style dialog box titled "Register". It has a light blue header bar with standard window controls (minimize, maximize, close). The dialog contains two text input fields: the first is labeled "Login" and the second is labeled "Password". Below these fields are two buttons: "Register" on the left and "Cancel" on the right. The "Register" button is highlighted with a blue border.

(a) Registration functionality

A screenshot of a Windows-style dialog box titled "Login". It has a light blue header bar with standard window controls (minimize, maximize, close). The dialog contains two text input fields: the first is labeled "Login" and the second is labeled "Password". Below these fields are two buttons: "Login" on the left and "Cancel" on the right. The "Login" button is highlighted with a blue border.

(b) Login functionality

Figure 16: Registration and login

¹¹<https://piazza.com/class/spring2013/252028400l/41>

5.1.2 Online / offline mode

The user can switch between online and offline mode.

Classes: MainViewModel, DiskServiceClient, SynchronizationViewModel, SynchronizationService

Commands: SwitchToOnlineModeCommand, SwitchToOfflineModeCommand

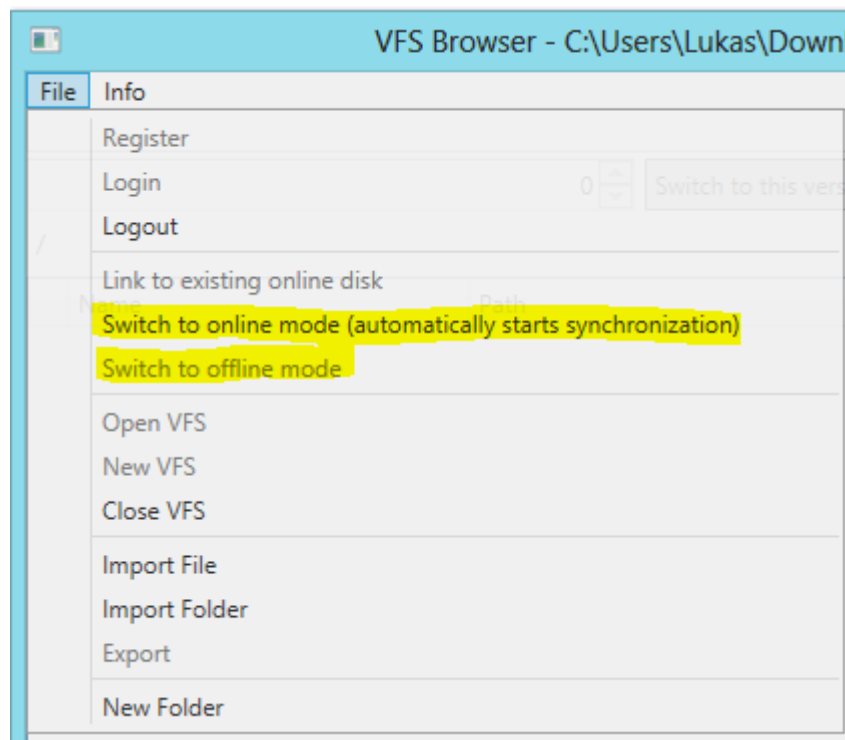


Figure 17: Online and offline switch

5.1.3 Binding of existing disk

The user can bind an existing disk to the local machine. Synchronization will then start automatically.

Classes: MainViewModel, DiskBrowserViewModel, DiskServiceClient

Commands: LinkDiskCommand

5.1.4 Registration of unique accounts

The server provides a registration of unique user accounts. Each account includes name and password. To make registered users persistent, SQLite is

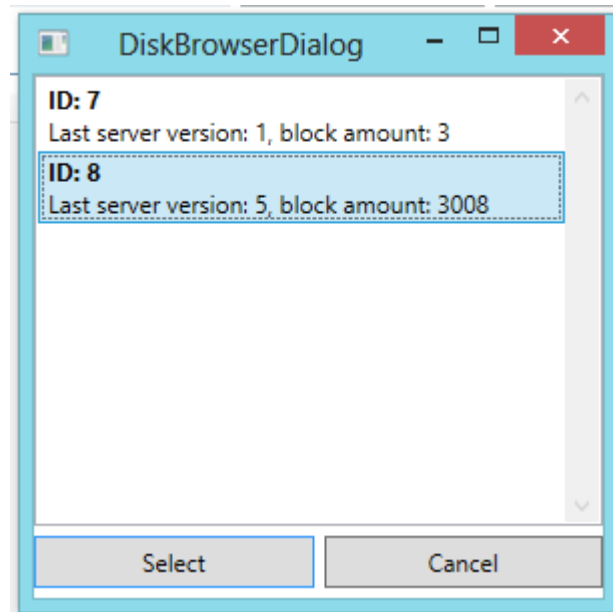


Figure 18: Binding an existing virtual disk

used. If persistence would not have been implemented, a dictionary with the login as key could have been used.

Classes: UserDto, DiskServiceImpl, PersistenceImpl

5.1.5 Automatic synchronization across machines

Once a disk is bound to a server and the client is in online mode, the local changes are synchronized across the machines automatically. If remote changes occur, the client automatically downloads all changes from the server.

5.1.6 Mocked unit tests

Because of the IoC pattern, mocked unit tests are fairly easy to implement. For example, in the VFSConsoleTests project, the FileSystemTextManipulatorMock mocks the FileSystemTextManipulator. Therefore the console can be tested without the FileSystemTextManipulator. Additionally, there are InOutMocks to mock the input / output.

5.1.7 Conflict resolution

There is an automatic conflict recognition so the user can resolve conflicts by rolling back to a specific version, which is not conflicted, and then restart

```

/// <summary>
/// Creates a user.
/// </summary>
/// <param name="login">The login.</param>
/// <param name="hashedPassword">The hashed password.</param>
/// <returns></returns>
public UserDto CreateUser(string login, string hashedPassword)
{
    var u = new UserDto { Login = login, HashedPassword = hashedPassword };
    _db.Insert(u);
    return u;
}

/// <summary>
/// Finds the user.
/// </summary>
/// <param name="login">The login.</param>
/// <returns></returns>
internal UserDto FindUser(string login)
{
    return _db.Find<UserDto>(u => u.Login == login);
}

```

Figure 19: Code to persist and to find a user

the synchronization again.

While the file system is conflicted, read operations are still possible. This way, the user can export a file he changed, roll back to a older version, synchronize the disk, and then import the file he changed again.

5.1.8 Automatic updates

The browser is updating automatically when changes to the disk occur. To make this possible, the file system provides an event, which fires when changes to the file system occur. The GUI registers to this event. This is the .NET / WPF implementation which is very similar to the observer pattern.

Classes: FileSystem, FileSystemTextManipulator, MainViewModel

Events: FileSystemChanged



Figure 20: The synchronization dialogue

5.1.9 Simultaneous synchronization

The server is implemented as a WCF ¹² service and it allows multiple parallel connections.

For every new connection, a new `IDiskService` is instantiated. This allows multiple connections to the server and thus parallel synchronization of changes. This is achieved by setting the `ServiceBehavior.ConcurrencyMode` to `ConcurrencyMode.Multiple`. To avoid additional overhead, the service is instantiated `PerSession`, meaning that every client creates a new service instance on the server.

Classes: `IDiskService`, `DiskServiceImpl`, `ServiceContract`

¹²<http://msdn.microsoft.com/en-us/library/dd456779.aspx>


```

public class InOutMocks : IDisposable
{
    private readonly MemoryStream _memoryIn;
    private readonly MemoryStream _memoryOut;
    private readonly StreamWriter _inWriter;
    private readonly StreamReader _outReader;

    public StreamWriter Out { get; private set; }

    public StreamReader In { get; private set; }

    public InOutMocks()
    {
        _memoryIn = new MemoryStream();
        _inWriter = new StreamWriter(_memoryIn);
        In = new StreamReader(_memoryIn);

        _memoryOut = new MemoryStream();
        _outReader = new StreamReader(_memoryOut);
        Out = new StreamWriter(_memoryOut);
    }

    public void FakeInLine(string line, bool seekToBeginning = false)
    {
        _inWriter.WriteLine(line);
        _inWriter.Flush();
        if (seekToBeginning) _inWriter.BaseStream.Position = 0;
    }

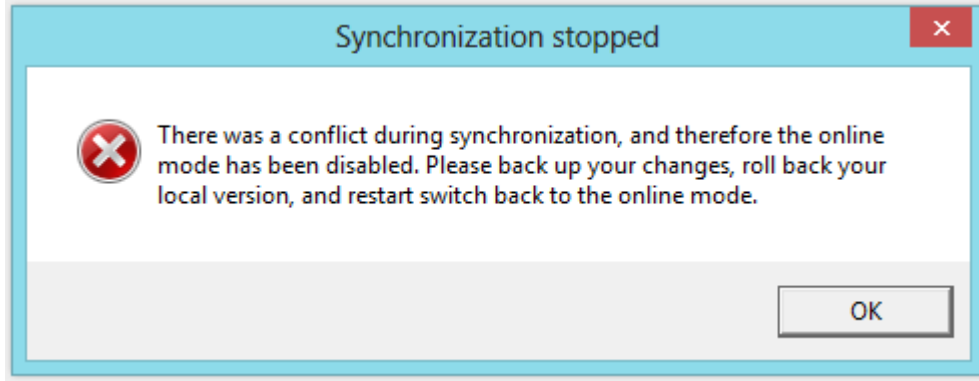
    public string FakeOutline(bool seekToBeginning = false)
    {
        Out.Flush();
        if (seekToBeginning) _outReader.BaseStream.Position = 0;
        return _outReader.ReadLine();
    }

    public void Dispose()
    {
        Dispose(true);
        GC.SuppressFinalize(this);
    }

    protected virtual void Dispose(bool disposing)
    {

```

Figure 21: The input / output mocks to test the console application



(a) Conflict recognition



(b) Conflict resolution

Figure 22: Conflict recognition and resolution

5.1.10 Incremental changes

The synchronization only requires to synchronize the blocks which have not been synchronized before.

For this, the DiskDto (persisted on the server) stores the NewestBlock which is to synchronize. Locally, every file system stores the blocks used in the root folder (BlocksUsed property) of the current version. When the synchronization starts, only

$$\text{Math.abs}(\text{NewestBlock}_{\text{OnServer}} - \text{BlocksUsed}_{\text{Locally}})$$

have to be synchronized.

Classes: SynchronizationService

5.1.11 File history

There is a file history available. The user can switch to an older version and export files from there, and he also can roll back to an older version, discarding all changes in later versions.

Technically, the file history is implemented like this:

In the beginning, there is one root node. Every change in the file system then leads to a new root node, which points back to the old root node. The new root node contains block references to all children of the old root node which have not been changed during the update of the file system. Additionally, the new root node contains references to the newly created nodes. So, for

```

public void OnFileSystemChanged(object sender, FileSystemChangedEventArgs e)
{
    Root = LatestRoot = ImportRootFolder();
    if (FileSystemChanged != null) FileSystemChanged(sender, e);
}

public event EventHandler<FileSystemChangedEventArgs> FileSystemChanged;

private void InitializeFileSystem()
{
    Root = LatestRoot = ImportRootFolder();
}

```

(a) Automatic updates: the event implementation in the file system class

```

_manipulator = _container.Resolve<IFileSystemTextManipulatorFactory>().CreateFileSystemTextMa
_manipulator.FileSystemChanged += FileSystemChanged;

```

(b) Automatic updates: the MainViewModel registers to updates when the file system is changed

```

private void FileSystemChanged(object sender, FileSystemChangedEventArgs e)
{
    ViewModelHelper.InvokeOnGuiThread(() => SwitchToLatestVersion(null));
    ViewModelHelper.InvokeOnGuiThread(RefreshCurrentDirectory);
}

```

(c) Automatic updates: Is executed when the file system is changed

Figure 23: Automatic updates

every version of the history, a new root node is created. The file system options, which are stored at the start of the file system, point to the latest root node in the file system.

Example

Given: The *rootfolder_{v1}* contains a *foldera_{v1}* and *b_{v1}*. *foldera_{v1}* also contains a subfolder *folderx_{v1}*. The options point to the *rootfolder_{v1}*.

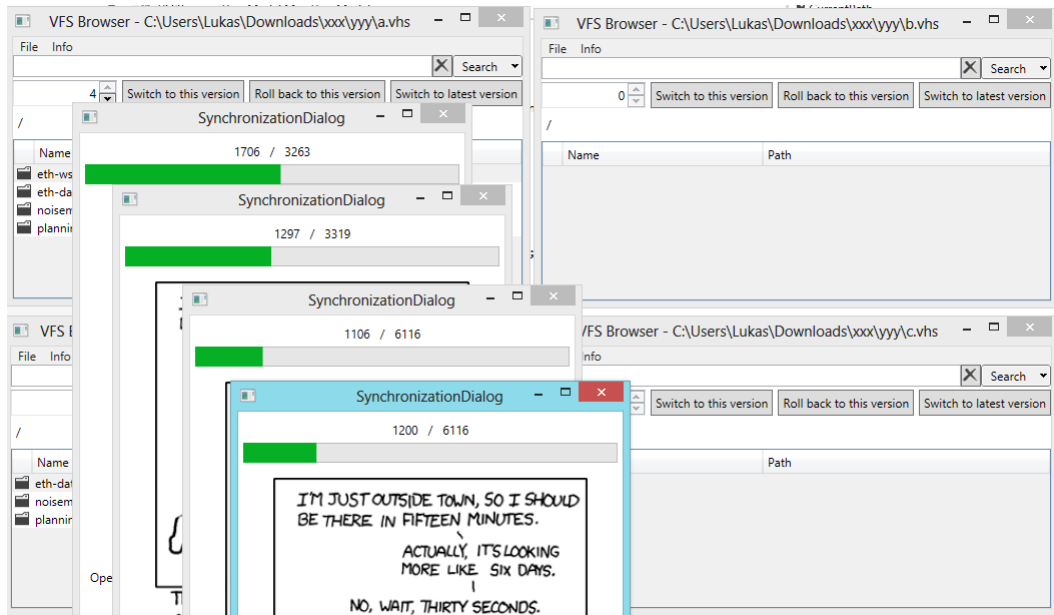
Scenario: Now a new subfolder *y_{v2}* should be created within the folder *a_{v1}*.

Action: Then a new folder *a_{v2}* is created, which references to the new subfolder *y_{v2}* and to the old folder *foldera_{v1}*. Next, a new *rootfolder_{v2}* is created. It creates a back link to *rootfolder_{v1}*, so it can switch back to an older version. Additionally, the new *rootfolder_{v2}* contains a reference to the folder *b_{v1}* (old reference) and a reference to *a_{v2}* (new reference).

Termination: After this, the options now point to the *rootfolder_{v2}*.

```
[ServiceBehavior(
    InstanceContextMode = InstanceContextMode.PerSession,
    ConcurrencyMode = ConcurrencyMode.Multiple)]
public sealed class DiskServiceImpl : IDiskService, IDisposable
```

(a) Attributes to enable parallelism of the disk service



(b) Demonstration of multiple simultaneous synchronization with one running server and two different disks (three of them linked, one of them separate)

Figure 24: Simultaneous synchronization

This gives us multiple advantages: Firstly, a written block never will change again (immutable object pattern). Therefore, synchronization is faster, because only the newly created blocks have to be synchronized. Furthermore, copy operations do not depend on the file size or the amount to copy. Thus the time to copy only depends on the length of the path of the files to copy.

Classes: FileSystem, BlockList

Methods: FileSystem.ArchiveAndReplaceRoot, BlockList.CopyReplacingReference

5.2 Design

For the synchronization server, a new project called VFSWCFSservice was added to the solution. It implements a IDiskService. The contracts which

```

private void DoSynchronize()
{
    var state = FetchSynchronizationStateInternal();

    if (state == SynchronizationState.LocalChanges) SynchronizeLocalChanges();
    if (state == SynchronizationState.RemoteChanges) SynchronizeRemoteChanges();
    if (state == SynchronizationState.Conflicted)
        throw new VFSException("Synchronization is conflicted, please roll back");
}

```

(a) Synchronize remote changes or local changes

```

_fileSystem.SwitchToVersion(remoteDisk.LocalVersion);
var fromBlockNr = _fileSystem.Root.BlocksUsed;
_fileSystem.SwitchToLatestVersion();
var untilBlockNr = _fileSystem.Root.BlocksUsed;

_callbacks.ProgressChanged(0, fromBlockNr - untilBlockNr);
for (var currentBlockNr = fromBlockNr; currentBlockNr <= untilBlockNr; currentBlockNr++)
{
    _diskService.WriteBlock(_user, _disk.Id, currentBlockNr, _fileSystem.ReadBlock(currentBlockNr));
    _callbacks.ProgressChanged(currentBlockNr - fromBlockNr, untilBlockNr - fromBlockNr);
}

```

(b) Only synchronize blocks which have not been synchronized yet

Figure 25: Incremental changes

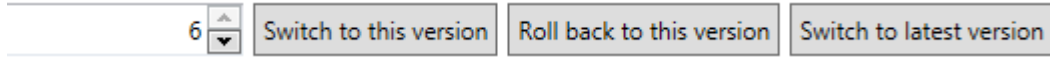
are shared among the clients have been extracted to the VFSWCFCContracts Project. It contains the data contracts for the Data Transfer Objects (DTO) ¹³ and the interface for the service contract ¹⁴ IDiskService.

Additionally, the BlockManipulator and the BlockAllocation were extracted from the VFSBase to the new project VFSBlockAbstraction. This way, the server can use the BlockManipulator and the BlockAllocation to store the blocks, which he receives from the clients, to the disk.

Basically, the synchronization works like this: First, the client creates a disk and then chooses to make it a synchronized disk. For this, he creates an account with login and password. Then he can synchronize his disk for the first time. Here, the client checks if the disk is not already linked to the server (he can do this by checking if the disk id is 0) and then start the initial synchronization. All the blocks are sent to the server, and so are the options

¹³<http://msdn.microsoft.com/en-us/library/ff649585.aspx>

¹⁴<http://msdn.microsoft.com/en-us/library/ms732002.aspx>



(a) The file history GUI

```
public void SwitchToVersion(long version)
{
    if (version < 0) throw new VFSEException(string.Format("Version {0} is not a positive version", version));
    if (LatestRoot.Version < version) throw new VFSEException(string.Format("Version {0} does not exist", version));
    if (Root.Version < version) Root = LatestRoot;

    while (version < Root.Version)
    {
        var blockNr = Root.PredecessorBlockNr;
        var readBlock = _blockManipulator.ReadBlock(blockNr);
        Root = _blockParser.ParseFolder(readBlock);
        Root.BlockNumber = blockNr;
    }
}

public void RollBackToVersion(long version)
{
    SwitchToVersion(version);
    LatestRoot = Root;
    _options.RootBlockNr = Root.BlockNumber;
    WriteConfig();
}
```

(b) The file history implementation (FileSystem)

```
/// <summary>
/// Copies the toCopy index node, and replaces the toReplace node with the replacement
/// </summary>
/// <param name="toCopy">To index node to copy.</param>
/// <param name="toReplace">To node to be replaced. Can be set to null if only a node should be appended and no one should be replaced.</param>
/// <param name="replacement">The node to replace the node toReplace. Can be set to null for the delete action.</param>
/// <param name="newVersion"></param>
/// <returns></returns>
public Folder CopyReplacingReference(Folder toCopy, IIndexNode toReplace, IIndexNode replacement, long newVersion)
{
    var toReplaceNr = toReplace == null ? 0 : toReplace.BlockNumber;
    var replacementNr = replacement == null ? 0 : replacement.BlockNumber;

    var newFolder = new Folder(toCopy.Name)
    {
        //BlocksCount = newBlocksCount,
        PredecessorBlockNr = toCopy.BlockNumber, //toReplaceNr,
        BlockNumber = _blockAllocation.Allocate(),
        Version = newVersion
    };
    _persistence.Persist(newFolder);

    var b = new BlockList(newFolder, _blockAllocation, _options, _blockParser, _blockManipulator, _persistence);

    // This algorithm section can be improved. We don't have to copy all references, we only have to copy the references that are different.
    foreach (var reference in AsEnumerable())
    {
        var blockNumber = reference.BlockNumber == toReplaceNr ? replacementNr : reference.BlockNumber;
        if (blockNumber != 0) b.AddReference(blockNumber);
    }
    if (toReplace == null && replacement != null) b.AddReference(replacementNr);
    if (replacement != null) replacement.Parent = newFolder;

    return newFolder;
}
```

(c) The file history implementation (BlockList.CopyReplacingReference)

Figure 26: File history

of the file system. After pushing all blocks to the server, the client sends the current root version to the server, which then is stored on the server. This way, the server can check if there are local changes, remote changes, conflicts, or if everything is up to date. Also, the maximum block number is stored on the server. This way, the client knows which blocks he has sent to the server and which blocks are not synchronized yet. This allows incremental updates by only transferring the blocks which have not been synchronized yet.

As you can see in the Figure 27, the only dependencies of the VFSWCF-Service are the VFSWCFContracts and the BlockAbstraction. There is one hidden dependency though: The service client for the VFSBase is generated automatically from the VFSWCFContracts. Therefore, there is a dependency from VFSBase to VFSWCFContracts, which is hidden on this diagram.

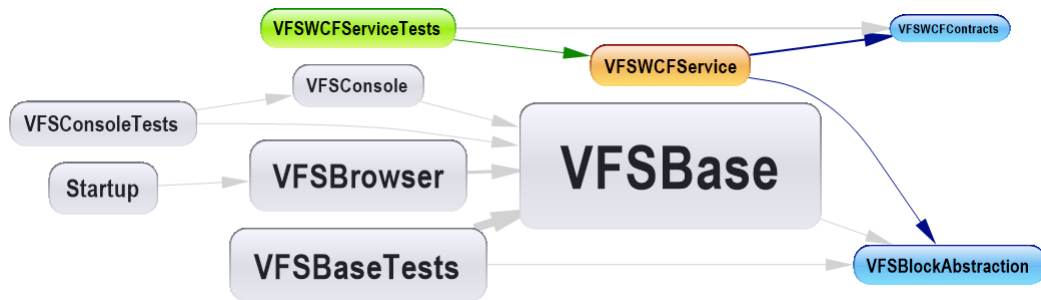


Figure 27: WCFService dependency diagram

The network protocol is defined on a very high level and can very easily be replaced, because to change the network protocol, only the configuration of the WCF service has to be changed. The current network protocol is set to the WCF default, which is HTTPS. The available operations can be looked up through a WSDL file, which is provided by the server.

5.3 Integration

Most of the existing design / API did not change, but it had to be enhanced for the server synchronization part. Especially, the FileSystemManipulator and the FileSystem now had to be thread safe, because the synchronization can run in the background.

Additionally, some existing functionality was replaced / dropped because of the history functionality and the immutable blocks. For example, the free memory command is not necessary because with the history, there is no scenario where memory is freed.

One very good thing were the tests. They only changed very little since they were written, and they provided immediate feedback, if something went wrong during a refactoring or by enhancing the system. They also run very fast and are run parallel by the Visual Studio 2012.

6 Quick Start Guide

6.1 Installation

Fist, install the Git client of your preference ¹⁵.

Then, start the command line, navigate to the folder where you would like to checkout the repository, and enter:

```
git clone git://github.com/lukaselmer/VFSPrototype.git
```

Now, there should be a directory VFSPrototype. In the further installation instructions, the path to this folder will be referenced as *repository checkout location*.

Before the project can be started, the following software has to be installed. Even tough the software could run on other configurations too, there is no guarantee that the software will run (correctly) with a different configuration.

1. Windows 8 Professional, 64 Bit
2. Visual Studio 2012 Ultimate Edition (VS2012)
3. NuGet packet manager addon¹⁶ for VS2012 (restart VS2012 after installing the addon)

Additionally, it is required to add the folder that contains sqlite.dll to the system path. This folder is (relative to the Git repository root): "*repository checkout location/Code/VFSPrototype/sqlite/*". After this, **your pc has to be restarted**, otherwise it will probably not work.

In case you would not like to add this directory to the system path, you can copy the sqlite.dll and the sqlite.def to the Windows/System32 directory.

Then, after installing the sqlite.dll and the NuGet packet manager addon, open the solution at "*repository checkout location/Code/VFSPrototype/VFSPrototype.sln*". After the project is loaded, right click on the solution and click on "Enable NuGet Package Restore", see Figure 28 on page 34. If this button is not available, then the NuGet package manager probably is not installed correctly.

¹⁵e.g. <https://code.google.com/p/msysgit/>

¹⁶<http://visualstudiogallery.msdn.microsoft.com/27077b70-9dad-4c64-adcf-c7cf6bc9970c>

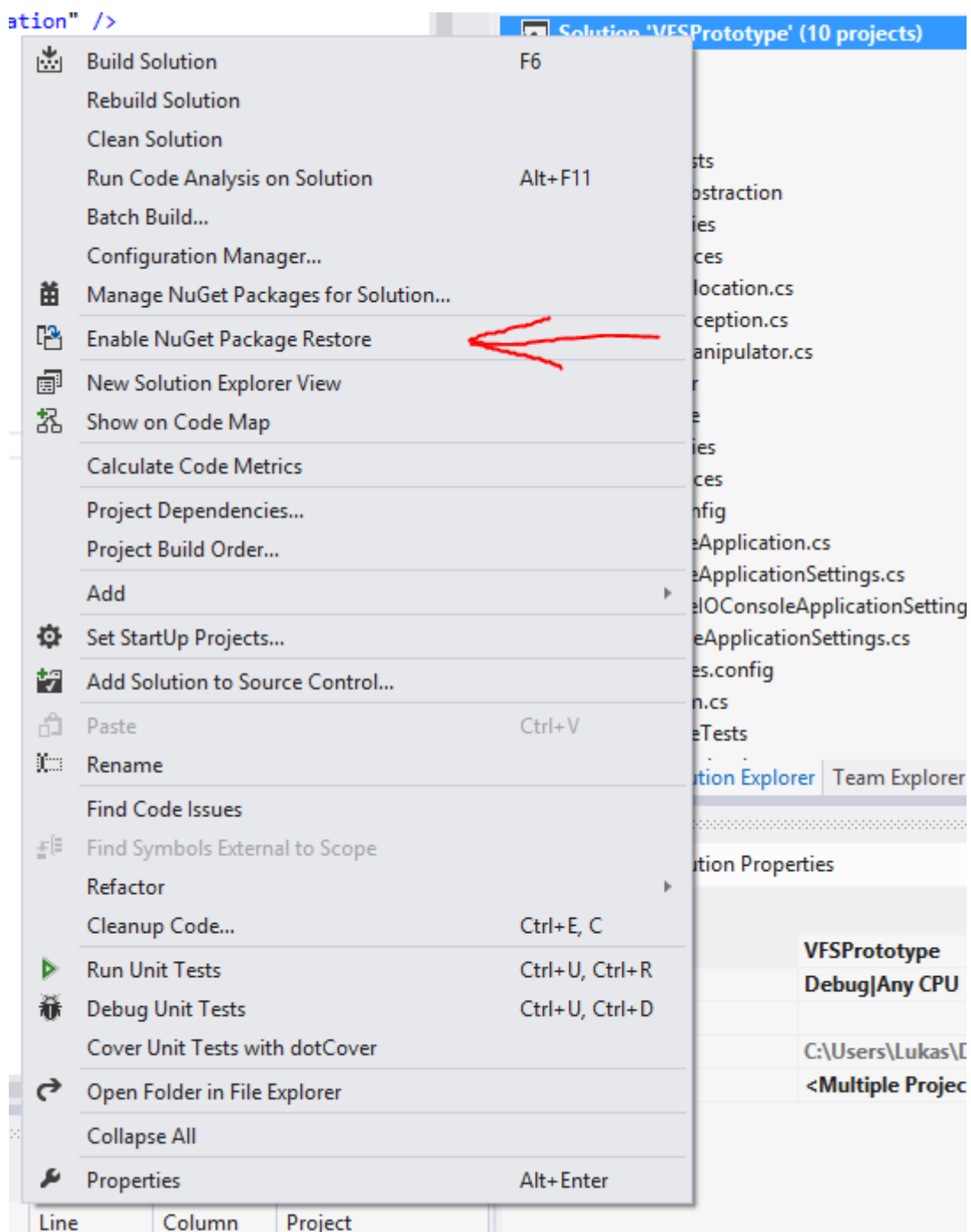


Figure 28: EnableNuGet Package Restore

To start the server, right click on the project "VFSCWCFContracts", click

"Debug"/"Start new instance". Now the server should be running and a new browser window should open.

To start the client, right click on the project "Startup", click "Debug"/"Start new instance". Now the client should be running and a new window with the file system browser should open.

You can start as many clients a you would like.

There is one more catch, literally. Try the login functionality with some wrong credentials. Now the VS2012 will witch to the debug mode, because a `FaultException` was thrown. This is inconvenient, because these exceptions are propagated to the client, because of the `FaultContracts`. To disable this behaviour, stop the program, and click on "Debug"/"Exceptions" in the VS2012 main menu bar. Then a new window should occur. Click on "Find..." and enter "FaultException". You should find two items "FaultException" and "FaultException'1". Disable all flags of these two exceptions, especially the "User-unhandled" checkbox. Press ok, and you will not see this exception again.

6.2 Console Application

The console application was developed at the beginning of the project. Therefore, many options available in the GUI are not available in the console.

Available commands:

- cd
- delete
- exists
- exit
- export
- help
- import
- ls
- mkdir

6.3 Scenario

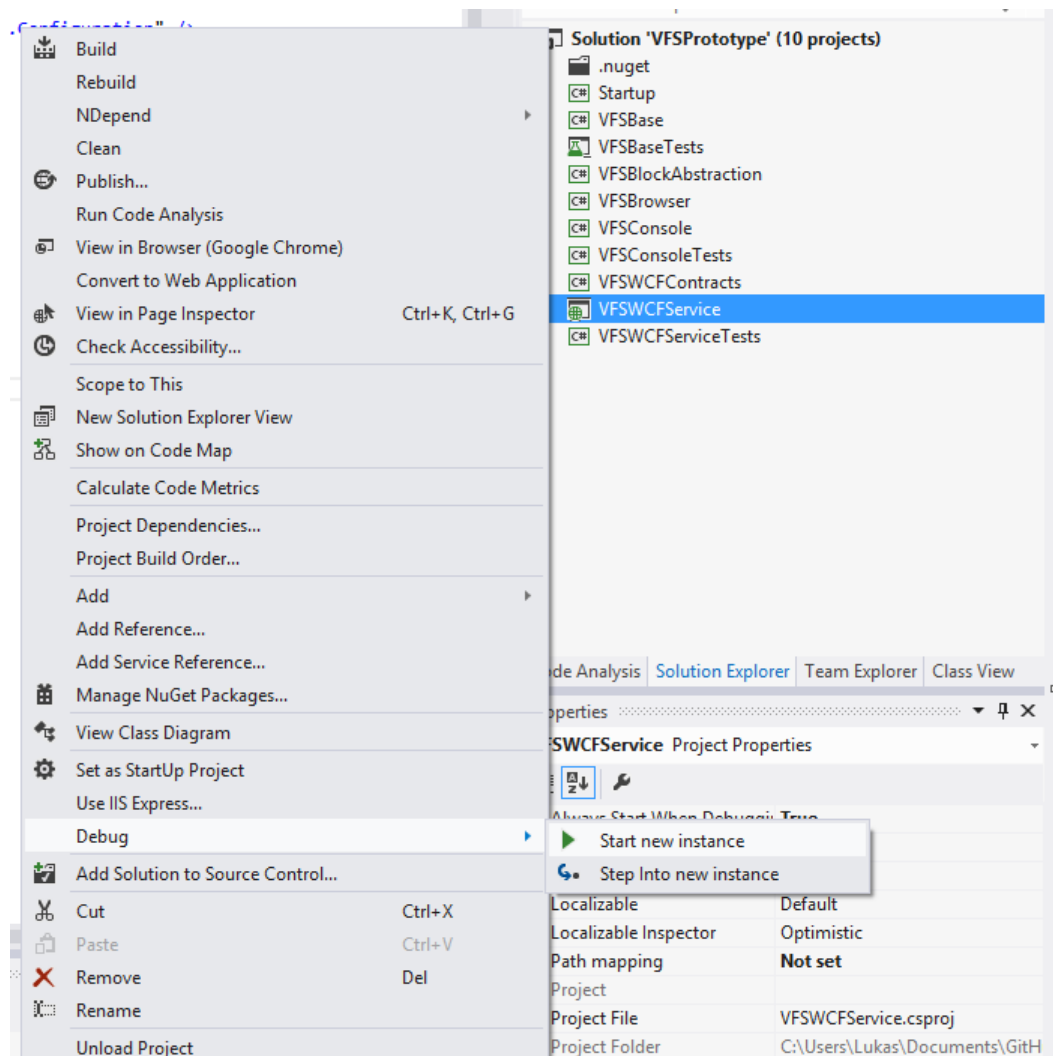


Figure 29: synchronization server on localhost.

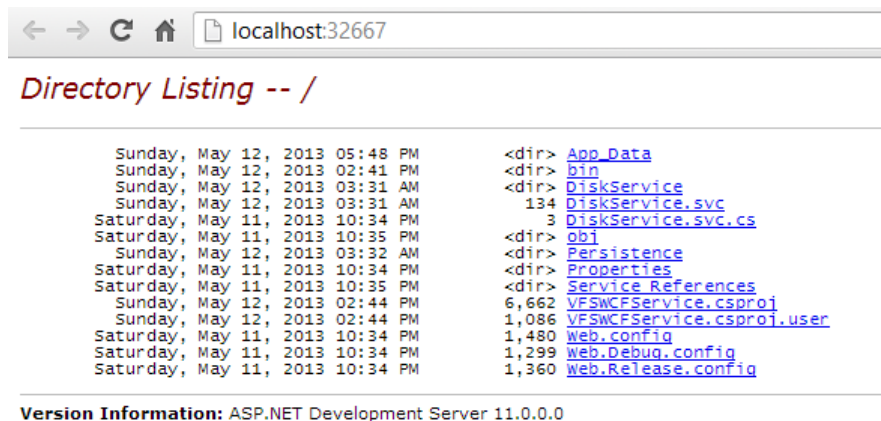


Figure 30: You should see this.

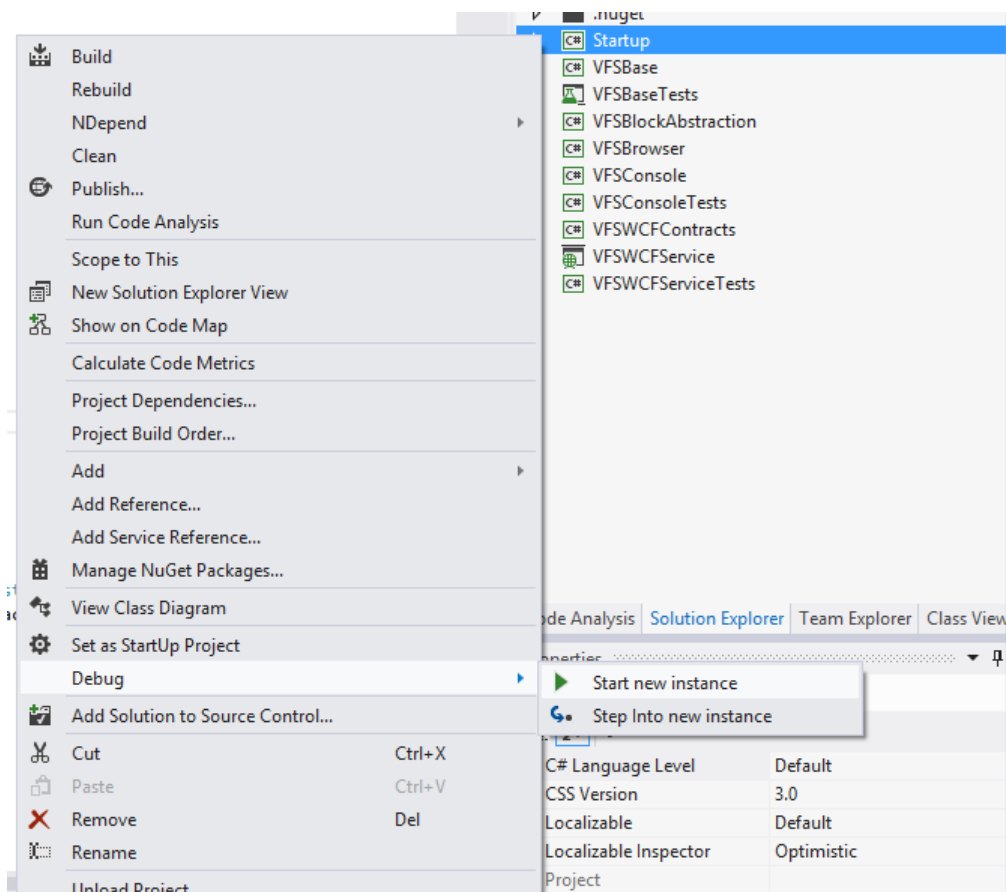


Figure 31: Start the client (do this two times to start two clients).

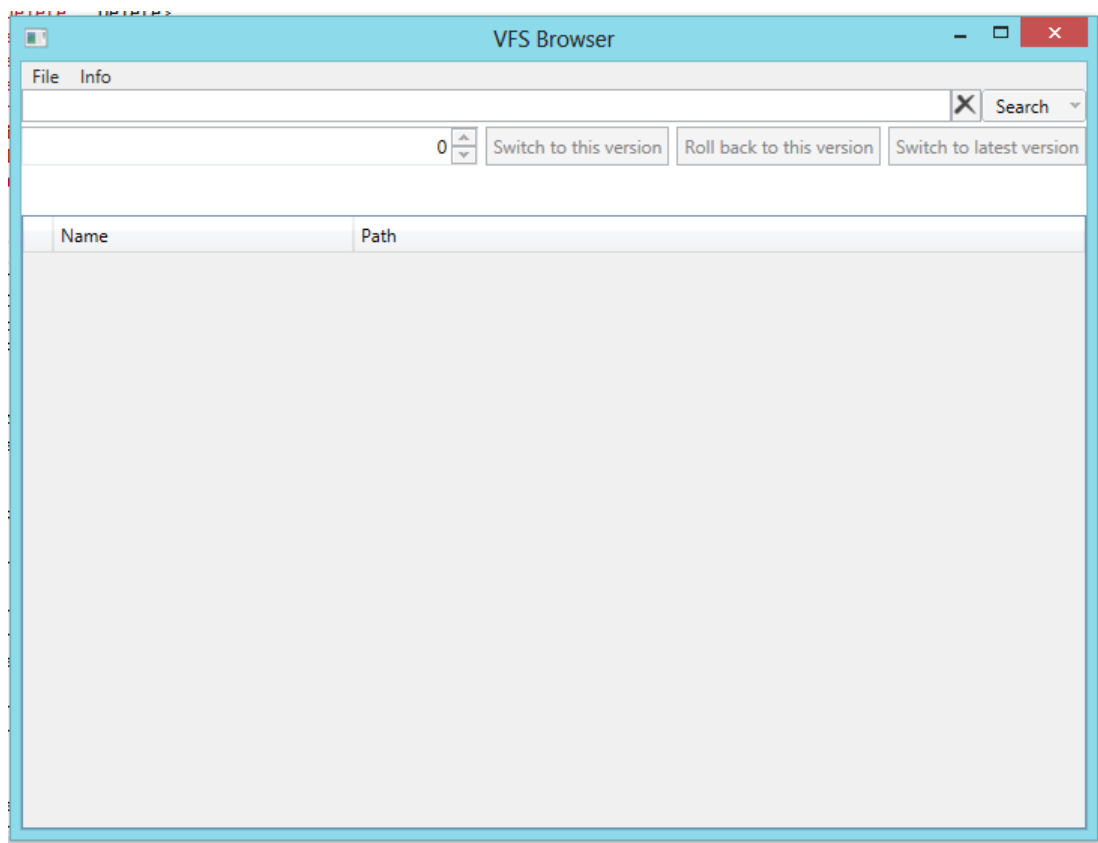


Figure 32: You should see this (two times if the client has been started two times).

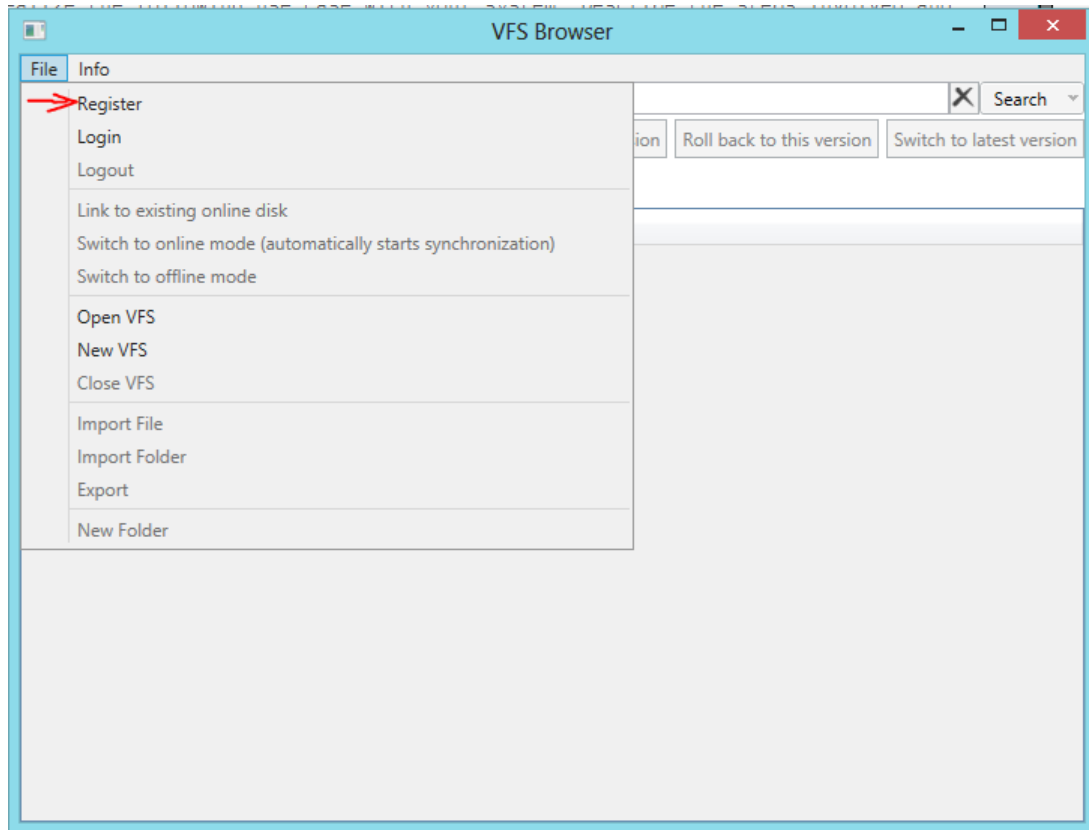


Figure 33: Register a new user account.

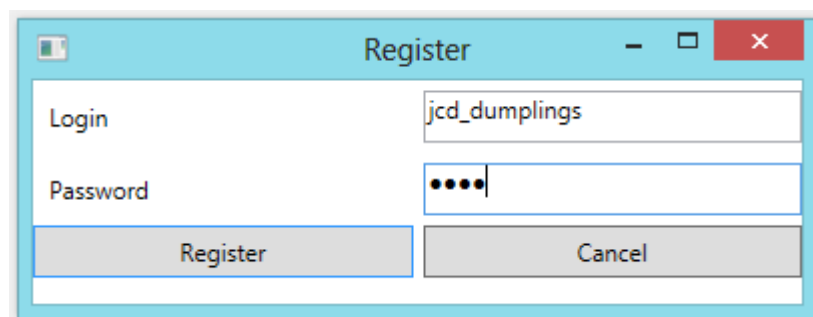


Figure 34: Enter credentials.

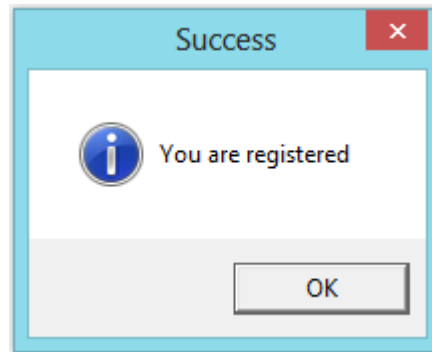


Figure 35: Registration successful.

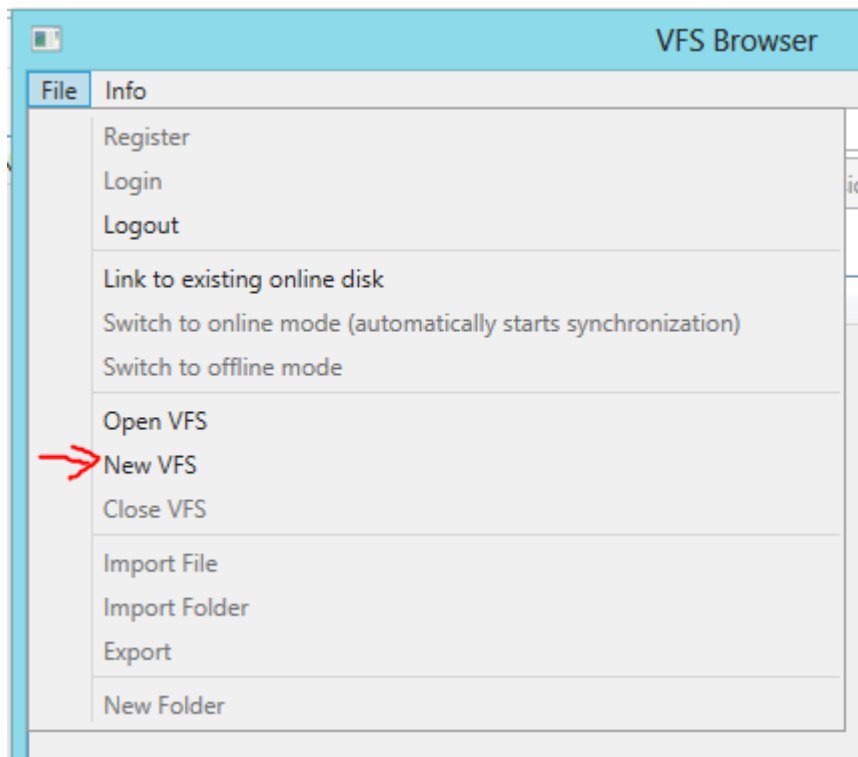


Figure 36: Create a new (local) VFS.

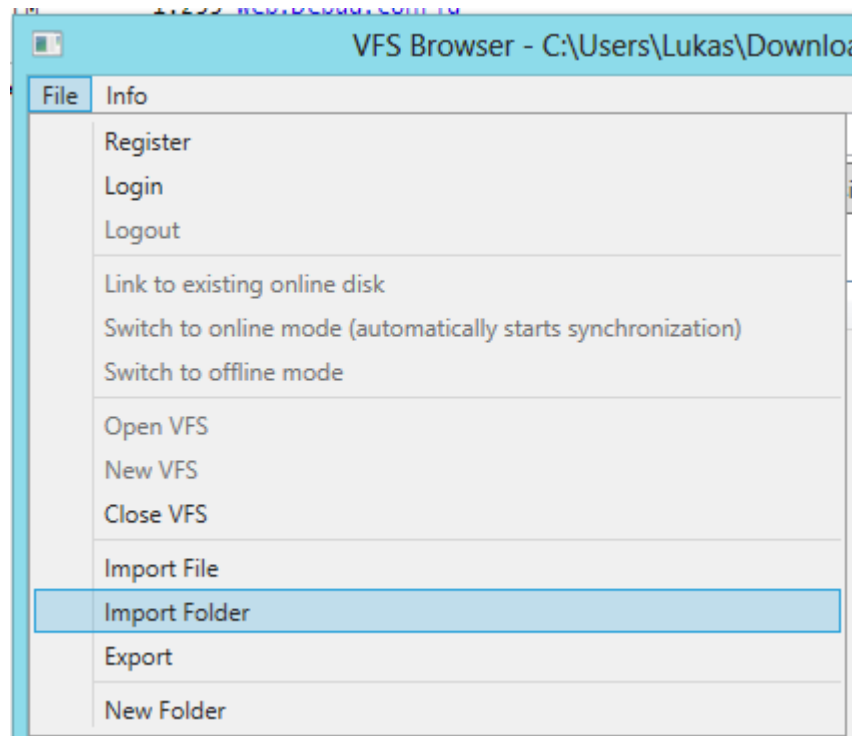


Figure 37: Import any folder.

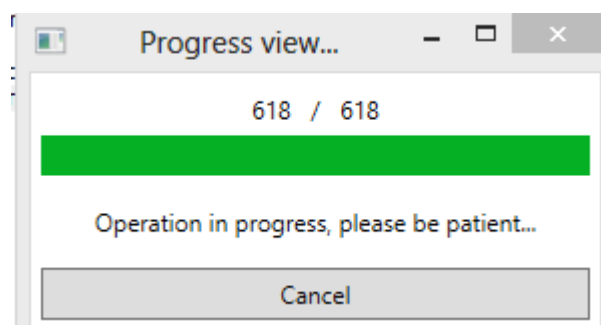


Figure 38: You should see the progress bar from the import.

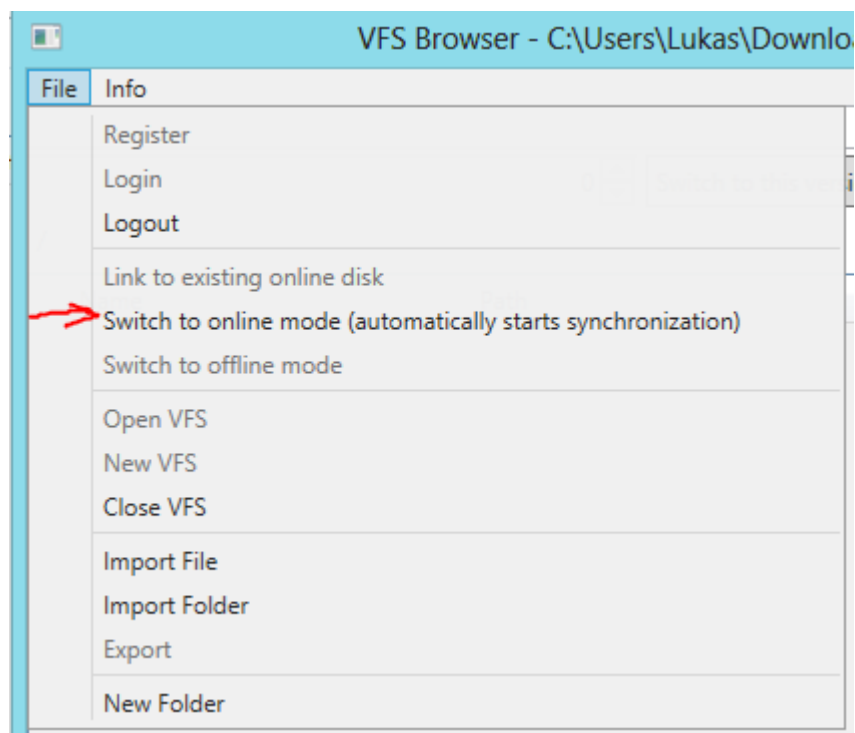


Figure 39: Switch to online mode to start synchronization.



Figure 40: When the synchronization starts, you should see this dialogue.

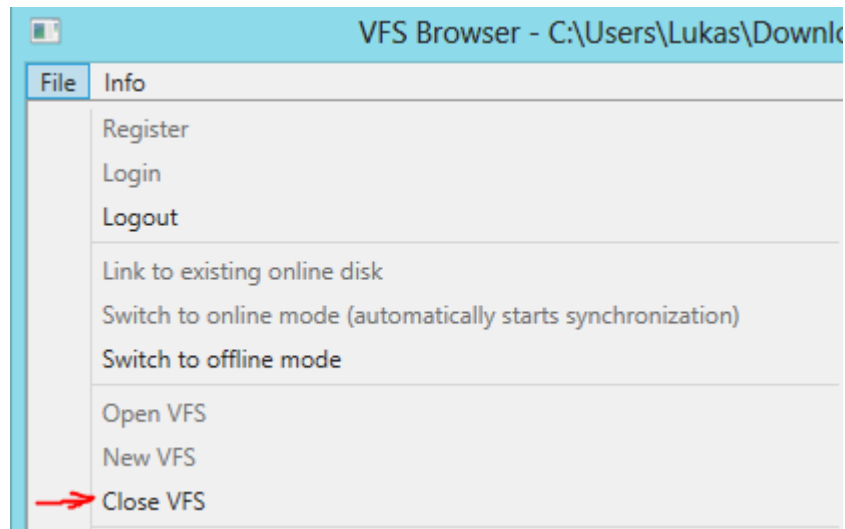


Figure 41: Close the VFS.

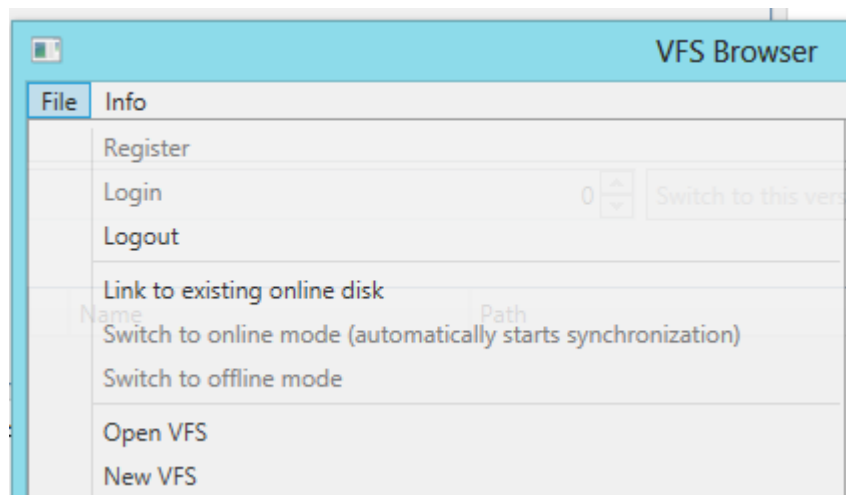


Figure 42: Switch to the other client, login with your credentials, and choose Link to existing disk.

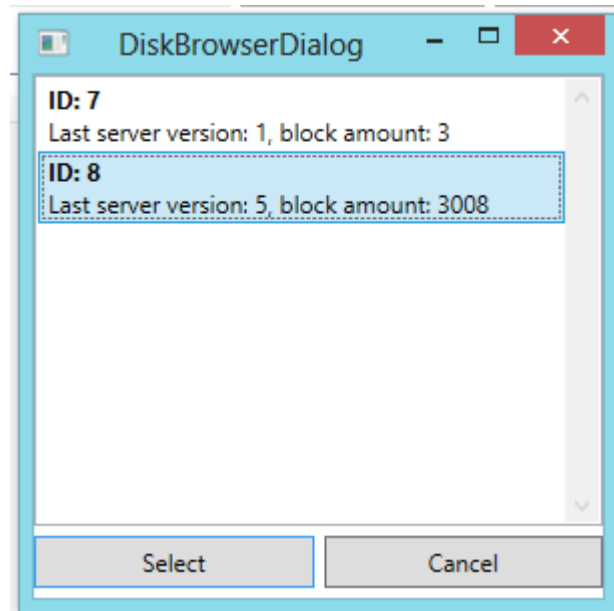


Figure 43: Select a disk.



Figure 44: When switching to online mode again, you should see the synchronization dialogue.

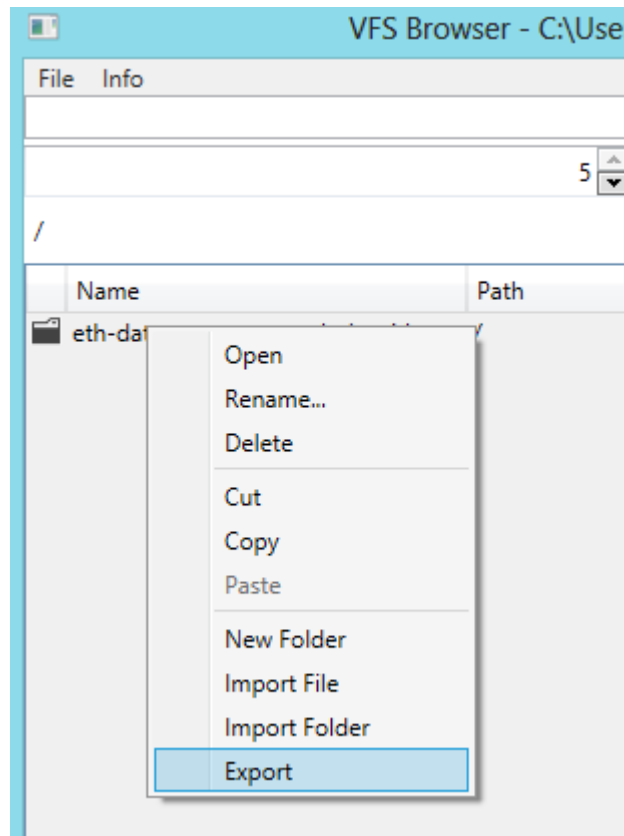


Figure 45: As soon as the dialogue is closed, right click on the synchronized folder to export it.

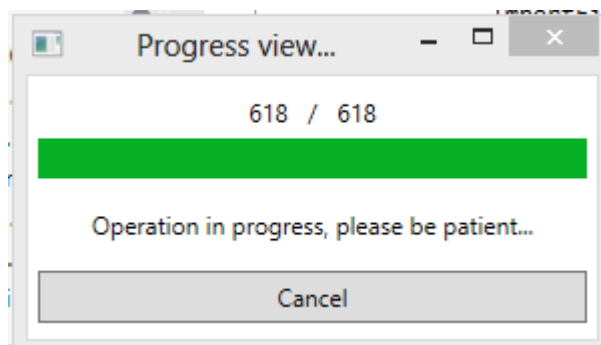


Figure 46: Once again, a progress report shows the export progress. After the export has finished, the folder you imported now is on the host system.