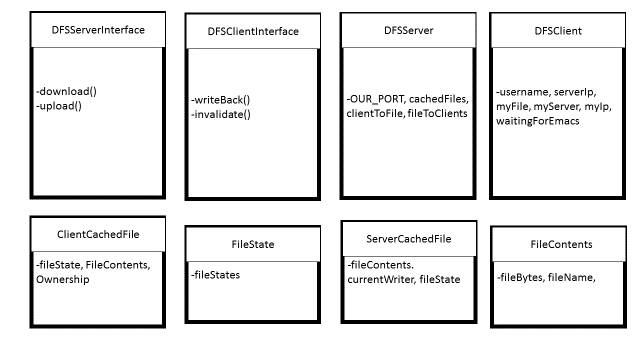
# Documentation

Figure 1: Classes



## Server

### Brief

The server follows the logic descried by the project description; ensuring that all the clients’ file-states adhere to logic described in the state transition diagrams. It’s how we factored the code that makes our project stand out.

The server relies on the setting of its currently cached file’s file-state, represented in the enum FileState.

### Flow

Instantiating a server is easy. We use a default port so you only need to pass the server its’ own IP as the first argument.

The server is factored out into a collection of functions that worked to simplify debugging of the program and manage the collection of cached files. There’re are three main data structures: two hash-maps; a map of client to file, and a map of file to clients; and one vector of cached files.

The cached files are stored as instances of ServerCachedFile which contains an instance of the provided FileContents class as well as a FileState and a string with the current client with write permissions.

Whenever the methods upload or download are called, the state of the ServerCachedFile is updated as well as the hash-maps with references to what client has what file cached. These updates take place after guaranteeing all other clients currently caching that file now have to correct FileState. In the case of ownership change of a file, the server changes the FileState of a cached file to Ownership\_Change and blocks. The download() function remains blocked until the current owner of a file responds to a writeBack() call and uploads a new version of the file.

## Client

### Brief

Client maintains the program flow dictated by the state transition diagram shown in the project documentation. By factoring out the code into many discrete functions for ease of debugging.

### Flow

Opening a client requires two arguments, its’ own IP, and the server IP. The port is a constant the client sources from the server class. The client then runs in two discrete stages, a startup, and a main loop.

The startup is executed with the main function. Where here error checking takes place and the client creates an rmi connection to the server as well as adding itself to the rmi.

The second part of the client works in three parts.

First the client gets a file from the server, then writes the file into its’ own cache. The client’s file state is guaranteed by the server. The client will always download a new version of the file from the server, unless the file state is not invalid and the ownership of the file has not been altered. Before writing reading a file. The client checks if it needs to upload its currently cached file.

Next the client runs emacs.

Finally the client checks if it has a file-state of Release\_Ownership, in this case it uploads its current file and changes its file state to Read\_Shared.

## Other Classes

The two interfaces DFSServerInterface and DFSClientInterface are there to allow the use of the java rmi package.

ServerCachedFile and ClientCachedFile wrap FileContents and hold additional information about a file particular to either the server or the client.

FileContents was provided.

FileStates is an enum with the a collection of file-states used by both the server and the clients.

# Output

## 1

## 2

## 3

## 4

## 5

## 6

## 7

# Discussion

## Functional Improvements

Improving our program would start with exception handling, currently we handle exceptions only generally and usually terminate to process. A more robust design would handle individual exceptions and in general not terminate the process. We would factor out the enums into two different groups, one for the server and one for the client. Our ServerCachedFile is not properly factored and does not use geters and setters. Writing our comments in a standard for that can auto generate documentation would also be a considerable improvement.

## Performance Improvements

Our program preforms well and completes the test. Making use of multiple threads for the server could allow for handling more clients. Where one thread handles downloads, and another handles uploads. Using semaphores and interrupts instead of spin locking the on ownership\_change would likely yield considerable improvements.

Working with emacs proved difficult and while a solution was found, there is likely a more efficient way to handle spinning up emacs.