Milestone 3 Assumptions Report

This report contains assumptions made in our implementation. Including clarifications made to the applications specification.

Overall Assumptions

1) Our shell is mainly tested on **Windows environment**. It may or may not work correctly on a unix based platform like (mac or linux). Our assumptions will be mainly based on the Windows platform.

2) We assume there would **not be any scenarios dealing with file or folder permissions** hence, we always **assume that the correct permissions are set to allow commands to be executed successfully** for example there should not be scenarios where file or folder permissions are the cause of the command not behaving as expected for example a scenario where rm command is unable to execute due to the only read permission allowed in the file. This is due to issues in Java API with regards to setting file permissions for Windows.

   Although Java provides setExecutable, setReadable, setWriteable, etc to change file permissions, it seems to work fine if the underlying java application is running on unix platform but there are problems when dealing with setting permissions in Windows due to windows ACL which you can read more about the issue regarding Java API with regards to set permissions in Windows [StackOverflow] or [Javadoc on permissions]

   Quote from Java official documentation (Javadoc):
   “A file system may implement restrictions to certain operations on the actual file-system object, such as reading, writing, and executing. These restrictions are collectively known as access permissions. The file system may have multiple sets of access permissions on a single object. For example, one set may apply to the object's owner, and another may apply to all other users. The access permissions on an object may cause some methods in this class to fail.”

   There are workarounds but it will be too complex to implement as it depends on the underlying file system and operating system.

Shell specification Assumptions

Call Command

Call Command follows the grammar specified in the specifications and allows for command substitution.

Pipe Operator

1) As stated in the specification, if an exception occurred in any of the parts, the execution of the other part is terminated hence for example, lsa | echo hello world where lsa is an invalid application. The lsa would throw an exception hence, the expected output of the above command is shell: lsa: Invalid app. In other words, once an exception is encountered, the exception is thrown and the rest of the parts are terminated.
   The above behaviour differs from in unix where it will still echo hello world despite the exception thrown.
2) **Note:** We ignore return a non zero exit code as this is similar to the Pipe Operator case regarding non zero exit code mentioned in the Q&AReport that we can ignore.

Sequence Operator

1) As stated in the specification, if an exception occurred in the execution of the first command, the execution of the second part can continue. Our shell would output the exception message followed by a new line for the first command and also execute the second command. **Note:** We ignore return a non zero exit code as this is similar to the Pipe Operator case regarding non zero exit code mentioned in the Q&AReport that we can ignore.
   For example: lsa; echo hello world
   Where lsa is an invalid application.
   Our shell behaviour (outputs):
   shell: lsa: Invalid app
   hello world

IO Redirection

1) For input redirection, if the input is a directory, then it will throw an File not Found Exception. For example, ls < src where source is a folder will throw File not found exception..

Command Substitution

1) When the shell executes “echo `exit`”, the program will exit automatically. This is different from the Unix shell where the console will display an empty line.
2) Command substitution back quoted is not nested. It is continuous
    - E.g. echo `echo `echo test``
    - Output: echo test

Globbing

1) Globbed files that does not exist for example if testFile does not exist, globbing will return args to command “testFile*”. This is allowed on UNIX based system but not Windows as Windows do not accept invalid characters such as * in path name, so on Windows machines, it will throw InvalidPathException error such as Illegal char <*> at index 8: testFile* whereas in UNIX machines it will still work as normal.

Application specification Assumptions

rm

1) Note: In rm command in Windows platform (as stated in overall assumption that we assumed our shell is run on Windows platform), input arguments such as:
    - .. refers to parent of current directory
    - . refers to the current directory
    - / or \ refers to the root directory (often system drive such as C:\)

2) The rm command when used with the **-rd** or **-r -d** or **-r** flags (including the permutation of the flags) would attempt to delete all files and/or folders input as arguments for the rm command together with the contents of non empty folders.

3) The rm command would only throw one exception (the latest exception) when there are potentially one or more exceptions due to the files and/or folders passed into the rm command arguments. This behaviour defer from unix which throws more than one exception when there are more than one exceptions due to the files and/or folders passed into the rm command arguments.

An example: *rm hello 1.txt hello2* where hello and hello2 are directories and 1.txt is a file which exists.
Behaviour in our shell: 1.txt will be deleted while one exception will be thrown highlighting that a directory is passed as an argument as rm without any flags cannot remove folders. hello and hello2 which are directories will not be removed. This is inline with the behaviour in unix. Note: Only the latest exception will be thrown.

4) The rm command would throw an exception ERR_IS_CURR_DIR when the input argument passed is the same as the current directory. This behaviour differs from unix shell as unix shell allows removal of current directory while still remaining in the path of the deleted directory.
Behaviour in our shell: If current directory is C:\Users\<COMPUTER_USER_NAME>\Documents\hello

<prompt>: *rm C:\Users\<COMPUTER_USER_NAME>\Documents\hello*
rm: This is the current directory
<prompt>: *rm -d .*
rm: This is the current directory

5) In rm command it is important to note that *rm / or rm* \ basically if the input argument of rm is a forward slash / or backslash \ it is referring to removing the root directory which in WIndows is usually the Windows default system drive for example if the current directory is C:\Users\<COMPUTER_USER_NAME>\Documents\hello, the system drive is C:\

6) The rm command does not support inputs which is a subpath of the current directory. Basically, removal of any folders from the current hierarchy of folders (parent directories cannot be removed).
This is intentional to avoid removing possibly all the ancestor/parent directories of the current directory which may result in potential problems such as potential side effect of removing the current directory when removing the ancestor/parent directories of the current directory which will result in the current directory be removed despite the prompt showing that we are in that directory. Our shell will throw an RmException with the message This is a sub path of the current path. This differs from unix shell behaviour which allows such removal while still allowing the current working directory to be in the already deleted path.
An example: If the current directory is C:\Users\<COMPUTER_USER_NAME>\Documents\hello
Behaviour of Our Shell:
<prompt>: *rm C:\Users\<COMPUTER_USER_NAME>\Documents*
rm: This is the sub path of the current path
<prompt>: *rm ..*
rm: This is the sub path of the current path

Note: An example to demonstrate what we consider sub path. A list of sub paths of the current directory (C:\Users\<COMPUTER_USER_NAME>\Documents\hello):

- C:\Users\<COMPUTER_USER_NAME>\Documents\
- C:\Users\<COMPUTER_USER_NAME>\
- C:\Users\
- C:\

Technically C:\Users\<COMPUTER_USER_NAME>\Documents\hello is also a subpath of itself but in rm command the error message thrown is different so as to allow users to have more information. In the case of our shell, as mentioned in earlier rm command assumption, if the current directory is C:\Users\<COMPUTER_USER_NAME>\Documents\hello and *rm C:\Users\<COMPUTER_USER_NAME>\Documents\hello* it will throw an error with message This is the current directory.

7) Note even if you try testing file permission (despite the fact that we stated in overall assumptions we will assume file permissions are correct) in WIndows machine it may or may not work. Please note that our implementation using Windows file attribute supports adding ReadOnly https://web.csulb.edu/~murdock/attrib.html there is no WriteOnly or ExecuteOnly etc like in unix. It is not flexible as in unix in terms of file permission. If it works it will either throw an IOException message or ERR_NO_PERM message though there is no guarantee. Note: In normal unix shell, unix shell cannot remove directory with write only permissions when it has empty directory flag but without recursive flag as it will show directory is not empty error though we try to group it for our shell under ERR_NO_PERM but as stated earlier there is no guarantee it will work especially on Windows. However, in normal unix shell, unix shell can remove directory with write only permissions with recursive and empty directory flag hence, we try but no guarantee (refer to bug mentioned earlier in Java API), to support removing folder with write only permissions with recursive flag (Might not work on Windows)

echo

1) If there are no arguments passed to the echo command, it will output an empty line. This behaviour is similar to in unix where an empty argument passed to echo command also outputs an empty line.
   An example: *echo*
   Behaviour in our shell:
   <prompt>: echo

   <prompt>:
   Note: As shown above, echo with empty arguments outputs an empty line.

2) If *echo "A*B*C"* it will output A*B*C without double quotes similar to in unix shell.

Behaviour in our shell:
<prompt>: echo “A*B*C”
A*B*C
<prompt>:
Behaviour in unix shell:
<prompt>: echo “A*B*C”
A*B*C
<prompt>:

exit

1) As stated in the project specification and skeleton code comments for exit application, the input arguments passed to exit application is not used so regardless of what or how many arguments are passed to exit application, our shell will just exit with status code 0.

   This differs from unix shell behaviour which accepts exit code as optional parameters and throw error if more than one input arguments stating that there are too many arguments.

   Note: This is also due to the fact that the provided skeleton ExitException.java provided comments stated that ExitException is used to send a signal to the shell to exit so it is in theory not used to throw error messages instead is used as a signal for shell to exit.
   Behaviour in our shell:
   <prompt>: exit

   Process finished with exit code 0

cd

For cd command, we generally test our shell on Windows platform and since Windows platform and unix platform have many differences in terms of PATH and allowed filenames, we will try to focus on the behaviour of our shell on Windows as much as possible.

1) The cd command PATH argument when it is input forward slash / or backslash \, in our shell the directory will remain unchanged when runned on Windows platform but it will change to the / directory when run in unix if the input is forward slash /. If the input is backslash \ in unix, it will attempt to switch to a directory with filename \ if it exists, else it will throw an error that the directory does not exist.

In actual unix shell, *cd /* will direct to the / directory of unix on the other hand if *cd* \ is inputted it will request for further input to change to the input directory for example *cd* \ followed by input *hello* where hello is an existing directory in the current directory, it will change to the hello directory.

2) The cd command must take in one argument. (cd without arguments is not supported) as mentioned in the CdApplication.java comments in the skeleton code.
   For example: *cd*
   The above example is not allowed in our shell and would throw a CdException of missing arguments message.

3) If our shell is being run on Windows it will not allow invalid path characters such as colon :
   For example: **In our shell,** *cd C:* is invalid when our shell is run on Windows due to the invalid path character colon : as Windows does not allow : as a path character or as a filename or folder names. It will throw an error with regards to illegal character by Windows. Since our overall assumption stated to test using Windows environment, file names or path names that are not allowed in Windows would throw an exception by Windows.

   Note: However, *cd C:/* will work if our shell is run on Windows assuming your system drive is C:/ (an existing drive letter) despite the colon as it is referring to the system drive of the current machine.

   On the other hand if our shell is run on unix, as colon is allowed as valid file names or folder names in unix hence, for example *cd :* in unix if there is a directory with the name :, it will change to the colon directory hence, similarly our shell if run on unix allows *cd :*

4) If cd command has more than one argument similar to in ubuntu, for example, *cd hello world*, would throw a CdException with Too many arguments.
5) As stated in our overall assumption, we assume that all permissions are correct and there are no instances like wrong permissions (for example no read permissions or no execute permissions) related scenarios due to issues with Java API in setting file permissions for different operating systems and file systems as highlighted earlier. Note: If you still choose to check, please note we try to add check for no execute permission (This is similar to unix shell which does not accept changing to directory with no execute permission) but as stated earlier it does not work for all operating systems and file systems so there is no guarantee.

Cp

1) If the destination directory does not exist, throw CpException ERR_FILE_NOT_FOUND

2) If the destination file is the same as the source file, for example, *cp 1.txt 1.txt*, where 1.txt exists, will throw a CpException with ERR_SRC_DEST_SAME message The source file and the destination file are the same file. This behaviour is similar to unix which would highlight that both files are the same file.
3) If the destination file already exists, for example, *cp 1.txt 2.txt* where 1.txt and 2.txt already exists, overwrite 2.txt with the contents of 1.txt.
4) If the source file input is a directory, for example, *cp hello hello2*, where hello and hello2 are existing directories, will throw a CpException with message This is a directory. This behaviour differs from unix which supports copying of directory to another directory when used with -r flag. Our shell does not support -r flag based on project application specification hence, copying directory is not supported.
5) If the input source files have at least one source file that does not exist, for example, *cp 1.txt 2.txt dest* where 1.txt is a non existing file while 2.txt is an existing file and dest is an existing directory, it should throw an CpException with No such file or directory but the 2.txt is still copied to hello directory. This is similar to unix behaviour.
   Another example, *cp 1.txt 2.txt dest* where 1.txt is an existing file while 2.txt is a non existing file and dest is an existing directory, it should throw a CpException with no such file or directory but the 1.txt is copied to the hello directory. This is similar to unix behaviour.
6) Our shell throws the latest CpException if it occurs for example *cp hello 1.txt 2.txt* dest where hello and dest are existing directories while 1.txt is a non existing file and 2.txt is an existing file. Our shell would throw the latest CpException which in this case is ERR_FILE_NOT_FOUND and at the same time copy 2.txt into the dest directory.
7) If the destination directory contains a file with the same file name as the file to be copied, overwrite the file in the directory with the file to be copied contents.
   For example: *cp 1.txt dest* where 1.txt is an existing file and dest is an existing directory containing another existing 1.txt file.
   Expected Behaviour: Overwrites the 1.txt file in the dest directory with the contents of the 1.txt file to be copied.
8) Note even if you try testing file permission (despite the fact that we stated in overall assumptions we will assume file permissions are correct) in WIndows machine it may or may not work. Please note that our implementation using Windows file attribute supports adding ReadOnly https://web.csulb.edu/~murdock/attrib.html there is no WriteOnly or ExecuteOnly etc like in unix. It is not flexible as in unix in terms of file permission. If it works it will throw either an ERR_NO_PERM message or an IOException message.

Paste

1) In paste, there are two forms of input allowed, namely filenames and stdin.
2) When filename is not given or put dash (“-“) after the paste command instead of file name, paste will read from standard input (i.e. <). Example: paste - file.txt < fileOne.txt. For this case, when merged, the file contents from the standard input will appear before

that of the filename. Only one dash is supported, and it can be placed before filenames or after filenames. Multiple dashes are not supported.

3) Note that when multiple files cannot be found/read etc, only one exception will be thrown.
4) Paste is expected to work with IO redirection (input '<' and output '>' for this case). Basic Stdin example: Paste < file.txt. The contents of the file will be in the inputstream, and will then be printed out by the paste command for this example.
5) If there is only one file provided, be it from Stdin or from filename, paste will print the contents of the file.
6) Stdin inputs can be command substitutions. For example, paste `echo file.txt` will print out the contents of file.txt.
7) Files with different numbers of lines will paste as per normal. Example, fileA has 5 lines, fileB has 3 lines, paste will still work.
8) Paste will throw an exception when Stdin is empty, invalid/empty/null filename is provided.
9) When both stdin and filename is provided, paste will always print the contents of the stdin before those contents of the file based on filename.
10) In terms of file type, theoretically, any text document file type will be acceptable. However, only ".txt" file type has been extensively tested. Therefore, we cannot guarantee that the performance of other types of text document file types will be consistent with the information stated here.
11) Note that when a file has more than one line, paste does not concatenate the lines with a tab character separating. It follows Linux behavior of using a newline to separate the different lines. Tab is only used to separate the content of different files on the same line.
12) If the file before the next file has an empty line, while the next file does not, there will not be any tab, when paste merges them. This follows Linux behavior.
13) A file with only newlines, will be treated like an empty file (except when input is provided via stdin). This is due to the use of bufferedreader.

Sed

1) Only single quotation marks commands will be accepted.
2) Other separators characters acceptable, for example: '/' or '|' or '\' or ',' or '.' or ';' or ':'. Note that only one form of separator characters could be used.
3) Any number of files can be provided. The files will be printed in sequence with the replacements made, and separated by newline (note: only 1 newline, therefore there will not be an empty gap).
4) When filename is not given after the sed arguments, sed will read from standard input (i.e. <).
5) Sed will throw an exception when either the regex, replacement index, or replacement string given is invalid or null or empty.

6) Note that a replacement index that is greater than the words in each line will not throw an exception. It will simply print out the file contents without replacement. This follows Linux shell's behaviour.

cut

1) The order of argument in the cut command is fixed.
    a) Options LIST [FILES]
2) Options must be either -b or -c only.
3) LIST must be either a single number or a range of numbers or 2 comma-separated numbers
4) Filename must only contains either:
    a) Alpha-numerical characters.
    b) Alpha-numerical characters with special characters where special characters cannot be the starting letter.
5) There can only be 2 real positive whole numbers provided for comma-separated numbers and number range.
6) It is not mandatory to have [FILES]. If FILE is not specified or '-', read from standard input.
    a) Shell can execute echo "baz" | cut -b 2
    b) Note that if you specified multiple '-', Shell will only take the first '-' and ignore subsequent '-' in the command.
7) Here are the list of file extension types that are accepted for cut
    a) .txt
    b) .html
    c) .sh
    d) .bat
    e) …

Mv

1) Assumption is that for a folder destination move, the folder does not contain a file with a similar name.

Ls

1) Ls follows Linux implementation where folder1 will be shown
    a) ls -R folder1
    folder1:
    file1.txt subfolder1

2) For multiple arguments, in linux,
    a) for example ls folder1 folder2 file3.txt
    b) The order will follow the input commands instead of being sorted to which directory comes first.
3) File can not start with dot(.)
4) -d and -R can not be true at same time. So ls -d -R FOLDERS is invalid
5) -d will only display folders and not any file
6) If -d or -R with no folders specified, use current directory
7) For globbing for Ls, the listed files will be in alphabetical order, unlike UNIX where files are listed first before folders.
8) For files with recursion, there might be more new lines depending on number of recursive calls made, which is different from Shell. But items listed should still be the same.
9) Ls file path with globbing will give you the file names of files that are globbed instead of the file path (different from shell)

Find

1) Fixed format: <foldernames separated by space> -name <filename>
2) In the FIND application, the argument after '-name' must be quoted with double quotes and only can exist 1 file name.
3) If no files are found, an empty string will be returned.
4) For our FInd application : find A -name “A*” will list all the files in the folder ‘A’ which is different from shell’s version where it also list A folder as well.

Sort

1. The order of argument in the sort command is fixed.
    a. [-n][-r][-f] [FILES]
    b. If you put [FILES] first, [-n][-r][-f] will be interpreted as files, if declared, and will print “sort: No such file or directory” exception message.
    c. If the flag arguments of [-n][-r][-f] is not declared, our program will interpreted the command as sort [FILES].
2. Our implementation of sort is different from Unix implementation of sort
    a. Based on unix implementation by default, `sort` sorts character by character using a locale specified sort order. Generally that's pretty close to ASCII order, but there may be some regional variations. From the man page:
    “ *** WARNING *** The locale specified by the environment affects sort order. Set LC_ALL=C to get the traditional sort order that uses native byte values. “

    b. Refer to http://man7.org/linux/man-pages/man1/sort.1.html
3. The order of output will be as followed based on the first character
    a. Special characters followed by numbers followed by capital letters and followed by small letters
4. Unlike Unix shell, our program does not allow sort application FILES to be a dash.
5. Filename must only contains either:
    a. Alpha-numerical characters.
    b. Alpha-numerical characters with special characters where special characters cannot be the starting letter.

Wc

1. The order of argument in the wc command is fixed.
    a. [-c][-l][-w] [FILES]
    b. If you put [FILES] first, [-c][-l][-w] will be interpreted as files, if declared, and will print “wc: No such file or directory” exception message.
    c. If the flag arguments of [-c][-l][-w] is not declared, our program will interpreted the command as wc -clw/-c -l -w [FILES].
2. Unlike Unix shell, our program does not allow wc application FILES to be a dash.
3. Filename must only contains either:
    a. Alpha-numerical characters.
    b. Alpha-numerical characters with special characters where special characters cannot be the starting letter.
4. Like Unix Shell, if any file that does not exist in the file system or the file is a directory, our program will print error message

Diff

1. Only text files (.txt) have been extensively used in the testing for diff. Although some other file formats may or may not work on this application, we strongly recommend using only text files.
2. When the input consists of only file names, only 2 file names are allowed to be provided.
3. Filename can be provided via stdin. To indicate that there is a stdin input, use “-”. This dash can be placed in any position of the argument list (i.e. any index of the argList).
4. Only subdirectories one level down from the directory names provided will be compared. This means if “subAFolder” is the immediate subdirectory of “AFolder”, and “subsubAFolder” is the immediate subdirectory of “subAFolder”, and 2 levels down from “AFolder”, “subsubAFolder” will not be considered during the comparison. Only “subAFolder” will be considered.
5. Options can be used in any order or combination, example: diff -sBq, diff -s -B …, diff -qs -B. The -B flag implies that all blank lines are removed after the files are compared for

their differences. The -s flag will cause the DiffApplication to output a message when the two files are identical. Without this -s flag, in the event of identical files, no output will be printed. The -q flag will cause the output to print out a message stating that the files are different rather than printing the actual differences. Without the -q flag, the output will be the actual differences.

6. For directories, the 3 flags are able to be used. However, by default, without the presence of any of the flags (e.g. diff A.txt B.txt), Diff will show the common directories, differences between files with the same name across the 2 directories. It will also show files that exist only in each of the directories.
7. Note that Diff between directories, in the output that prints the differences between the file contents, the output will have any space in between the '>' or '<' and the actual content. For example, "< for u in $ul". This follows the format that is present in the final diff example in the project description.
8. Differences between directories (those that only appear in one folder and not the other) are not sorted alphabetically based on the file or folder name. We deviate from Linux's exact sorting behavior because we believe it in this context, it is sufficient to organize based on folders alone rather than filenames. This means all the file names and folder names that are in folder A that do not exist in folder B will be printed first, followed by those file names and folder names of folder B. However, the order among the files will not be sorted.
9. As per linux behavior, when -s and -q flag is used and stdin is one of the input, the output will be "[filename] and - are identical/differ"
10. For binary files, we assume that there are only two possible binary file types that will be supplied: .bmp, and .bin. During the comparison of the binary files, if there are no differences, nothing will be printed. If there are differences, the different message (similar to that of the q flag) will be printed.
11. Due to the nature of bufferedreader, it stops reading when a newline character is present. As such, for the purpose of this project, diff will not be able to detect a newline at the End of File (EOF).

Grep

1. The -i and -c flags cannot be combined. Example: "-ic" is not acceptable. Also, these flags have to be exact, they cannot include other characters, else an exception will be thrown. Only "-i" and "-c" is allowed. Note that these flags can only be placed before the filenames.
2. Stdin is not indicated by dash.
3. If there are no filenames, then stdin may be used.
4. When more than one filename is provided, if the pattern matches, grep will print out the filename before printing out the line that matches.