Design your own shell – HW1

Team Members:

Jules Baud // Valentin Pichavant // Antoine Regnier

Installation:

```
$ ./Makefile
```
will generate six executable files.

```
$ ./main
```
Is the normal application

```
$ ./main_debug
```
Execute the program with the debug output

```
$ ./main_test
```
Execute the tests

```
$ ./main_loop
```
Allows to test the alarm and shows that the timer works

```
$./main_test_debug
```
Execute both the test and debug

```
$./main_debug_alarm
```
Test that the alarm is triggered

How to use

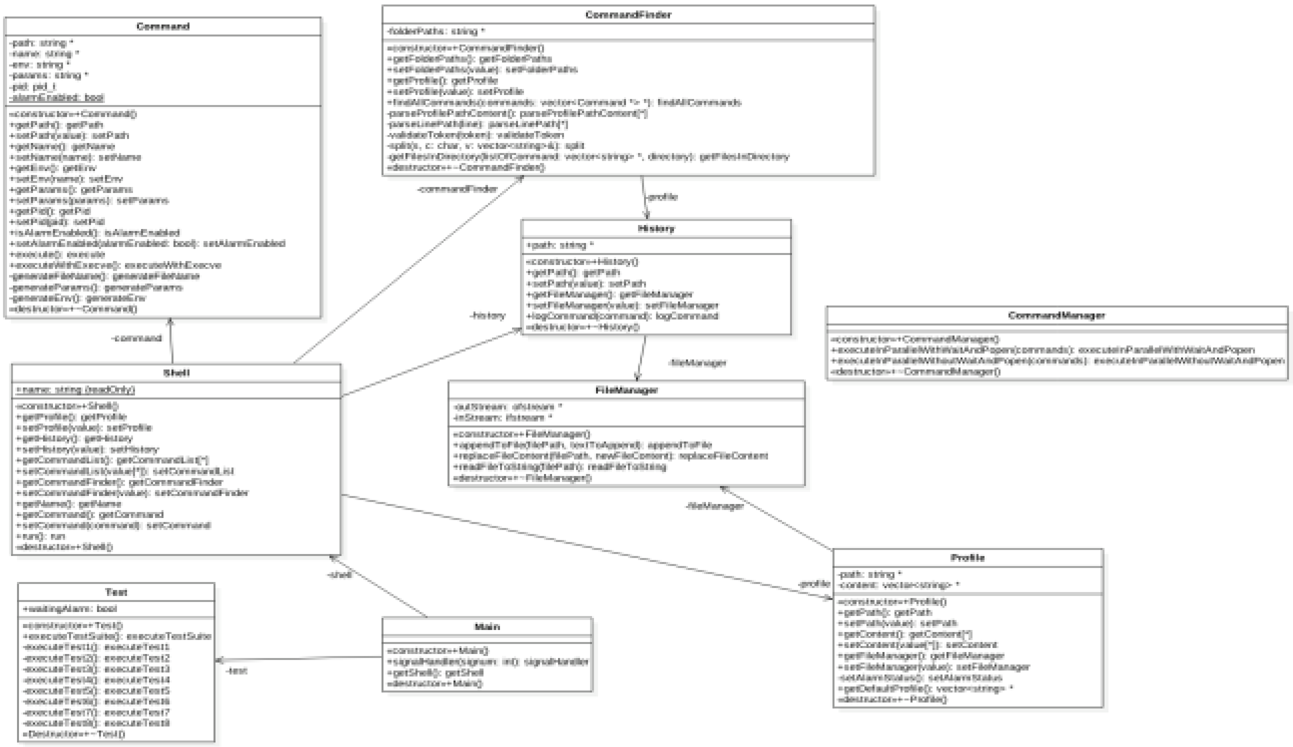
Once launched:
ESC to exit
Left Arrow and Right arrow to modify the input command
Tab initiate the autocomplete
Up Arrow and Down Arrow to iterate through the available suggestions

Why C++?

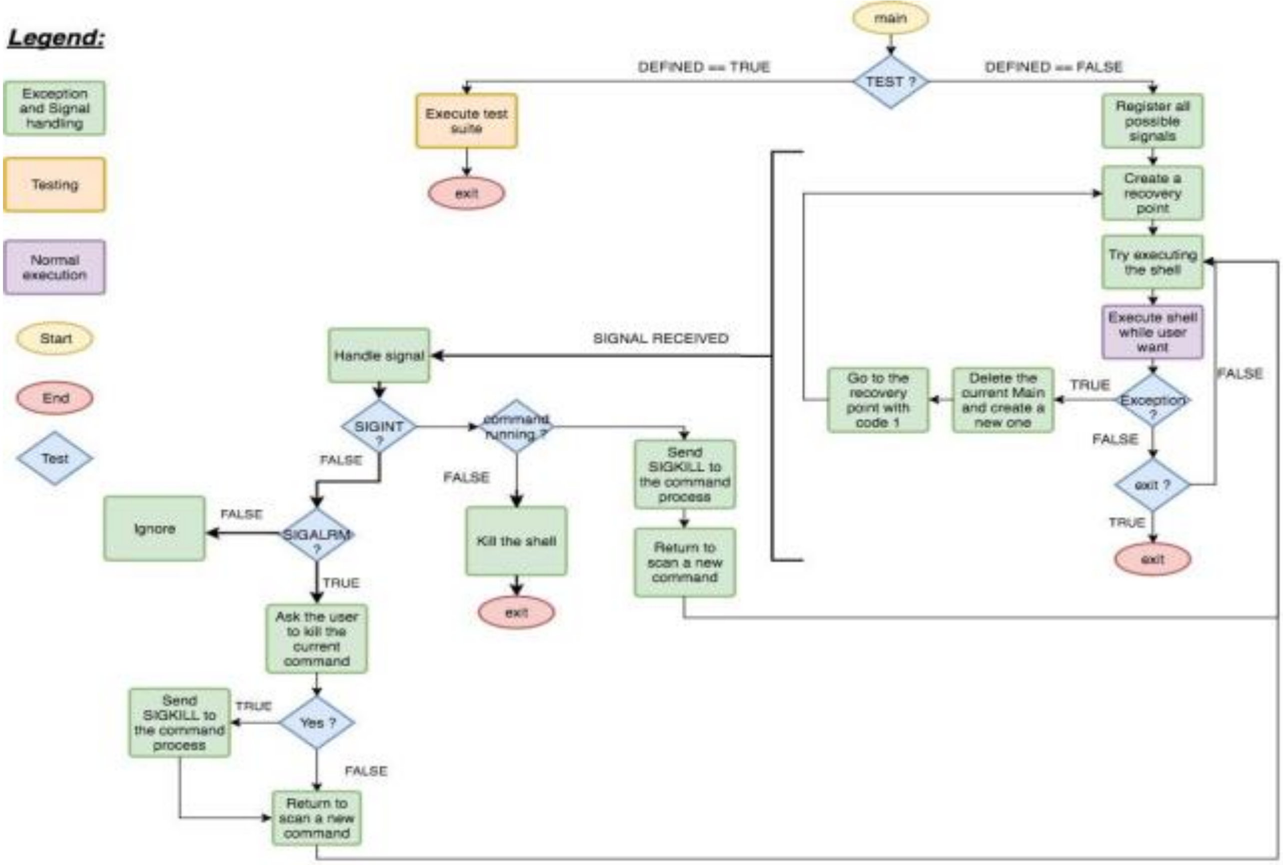
Using an object-oriented object allows to gain access to advanced capabilities that C++ offers which are absent in native C. First its object-oriented nature allows to decouple the source code and to increase the programmability. Furthermore, the advanced error catching allows to handle multiple system messages and/or I/O errors flawlessly.

Specifications:

Application UML:


Command
-path: string *
-name: string *
-env: string *
-params: string *
-pid: pid_t
-alarmEnabled: bool
«constructor»+Command()
+getPath(): getPath
+setPath(value): setPath
+getName(): getName
+setName(name): setName
+getEnv(): getEnv
+setEnv(name): setEnv
+getParams(): getParams
+setParams(params): setParams
+getPid(): getPid
+setPid(pid): setPid
+isAlarmEnabled(): isAlarmEnabled
+setAlarmEnabled(alarmEnabled: bool): setAlarmEnabled
+execute(): execute
+executeWithExecve(): executeWithExecve
-generateFileName(): generateFileName
-generateParams(): generateParams
-generateEnv(): generateEnv
«destructor»+~Command()
CommandFinder
-folderPaths: string *
«constructor»+CommandFinder()
+getFolderPaths(): getFolderPaths
+setFolderPaths(value): setFolderPaths
+getProfile(): getProfile
+setProfile(value): setProfile
+findAllCommands(commands: vector<Command *> *): findAllCommands
-parseProfilePathContent(): parseProfilePathContent[*]
-parseLinePath(line): parseLinePath[*]
-validateToken(token): validateToken
-split(s, c: char, v: vector<string>&): split
-getFilesInDirectory(listOfCommand: vector<string> *, directory): getFilesInDirectory
«destructor»+~CommandFinder()
-commandFinder
-profile
History
+path: string *
«constructor»+History()
+getPath(): getPath
+setPath(value): setPath
+getFileManager(): getFileManager
+setFileManager(value): setFileManager
+logCommand(command): logCommand
«destructor»+~History()
-history
-command
-fileManager
CommandManager
«constructor»+CommandManager()
+executeInParallelWithWaitAndPopen(commands): executeInParallelWithWaitAndPopen
+executeInParallelWithoutWaitAndPopen(commands): executeInParallelWithoutWaitAndPopen
«destructor»+~CommandManager()
Shell
+name: string (readOnly)
«constructor»+Shell()
+getProfile(): getProfile
+setProfile(value): setProfile
+getHistory(): getHistory
+setHistory(value): setHistory
+getCommandList(): getCommandList[*]
+setCommandList(value[*]): setCommandList
+getCommandFinder(): getCommandFinder
+setCommandFinder(value): setCommandFinder
+getName(): getName
+getCommand(): getCommand
+setCommand(command): setCommand
+run(): run
«destructor»+Shell()
FileManager
-outStream: ofstream *
-inStream: ifstream *
«constructor»+FileManager()
+appendToFile(filePath, textToAppend): appendToFile
+replaceFileContent(filePath, newFileContent): replaceFileContent
+readFileToString(filePath): readFileToString
«destructor»+~FileManager()
-fileManager
-shell
-profile
Profile
-path: string *
-content: vector<string> *
«constructor»+Profile()
+getPath(): getPath
+setPath(value): setPath
+getContent(): getContent[*]
+setContent(value[*]): setContent
+getFileManager(): getFileManager
+setFileManager(value): setFileManager
-setAlarmStatus(): setAlarmStatus
+getDefaultProfile(): vector<string> *
«destructor»+~Profile()
Test
+waitingAlarm: bool
«constructor»+Test()
+executeTestSuite(): executeTestSuite
-executeTest1(): executeTest1
-executeTest2(): executeTest2
-executeTest3(): executeTest3
-executeTest4(): executeTest4
-executeTest5(): executeTest5
-executeTest6(): executeTest6
-executeTest7(): executeTest7
-executeTest8(): executeTest8
«Destructor»+~Test()
-test
Main
«constructor»+Main()
+signalHandler(signum: int): signalHandler
+getShell(): getShell
«destructor»+Main()

Application's Workflow:


Legend:
Exception and Signal handling
Testing
Normal execution
Start
End
Test
main
TEST ?
DEFINED == TRUE
DEFINED == FALSE
Execute test suite
exit
Register all possible signals
Create a recovery point
Try executing the shell
Execute shell while user want
Exception ?
TRUE
FALSE
Delete the current Main and create a new one
Go to the recovery point with code 1
exit ?
FALSE
TRUE
exit
SIGNAL RECEIVED
Handle signal
SIGINT ?
command running ?
Send SIGKILL to the command process
Return to scan a new command
FALSE
Kill the shell
exit
FALSE
SIGALRM ?
Ignore
FALSE
TRUE
Ask the user to kill the current command
Yes ?
TRUE
Send SIGKILL to the command process
FALSE
Return to scan a new command

Exception handling:

- Every c++ exception of the shell instance is catched on the main. When they occur the main will get back at the beginning with a new Main instance and will try to start from scratch.

- All signals are caught on the main, that's why only a SIGKILL can kill the program. Otherwise we handle SIGINT to end the shell and SIGALRM to offer the possibility to kill the process. Other signals make the program restart from the beginning with a new Main instance.

Exception avoiding:

- Each constructor initializes the class dynamic attributes either with a NULL or with a new to avoid bad memory access (even if a SIGFAULT is catched by the signal handler and make the program restart).
- Each destructor delete only the NOT NULL dynamic objects.
- Setters delete also NOT NULL dynamic objects before setting the new value to avoid memory leak.
- Every pointer is checked as NOT NULL before access. For instance, in the Main before handling SIGALRM we check if there is a shell then if this shell contains a command and then we send the kill signal otherwise we didn't send it because a SIGALRM could came from another reason.
- The code requires at least to be compiled in C++11 which means more security because instead using for loop over collection, we use the for each loop to avoid NULL exception.
- We have created several mechanisms to control that every step of execution is secure, the most significant example is the parsing of the profile. The parsing can handle multiple PATH definition, if no PATH is found then this try to restore the. profile file, if that fail, the default profile will be used even if the. profile is not stored as a file. For each line of PATH, the validity of each token in the line is tested. We have a similar system for the ALARM value, we try to find if there is an ALARM in the. profile, if no we force alarm to be enable, else if we find ON or a string containing ON we set ALARM as ON otherwise if OFF is found are a string containing OFF we set the alarm to OFF. If multiple definition of ALARM are detected then the user is informed of the corrupted file and the last occurrence of ALARM is used.
- File access are managed by the File Manager class which means that the way that file are open/closed is abstract for other classes, and inside we use C++ stream to secure the way file are open for instance the instream open will call the C++ method basic_ifstream<_CharT, _Traits>::open(const string& __s, ios_base::open mode __mode) then basic_filebuf<_CharT, _Traits>::open(const string& __s, ios_base::openmode __mode) then basic_filebuf<_CharT, _Traits>::open(const char* __s, ios_base::openmode __mode) and finally the fopen C function and will verify that everything is ok to read the file.
- The shell is design to securely scan only authorized characters, every other character are ignored (for instance special char).

Maintainability:

- The program code use preprocessor instruction to handle if it's supposed to execute test cases or/and in DEBUG mode (which means to print debug information) and if it executed on MINIX or not (at the beginning we used thread to implement parallels process but we found that Minix does not implements pthread.h so we keep the code that work on UNIX.
- The code follow object oriented design which make it easy to use and to change.
- Most of the source code is documented with comments / javadoc like documentation for the methods / debug printing.
- The code is modular so we can assume to make it more like a real shell in the future, for instance for each command we could imagine define an environment or add an environment into the PROFILE.
- All class can be displayed as a string due to the overload of every ostream (like toString() in java).

Functionality:

- We have added the parsing of potential commands in the folder defined as PATH to suggest commands to the user merging them with the history (only executable file on the PATH folder are added).
- We can remove by using delete while we scan the command.
- Using left arrow, we can get before to change a char, for instance if we have "Hello" by pressing left arrow to time "Hel" will be displayed and we will be able to change the second l character and then we are able to use the right arrow to go back to the end of the string.

Testing:

- Most of the functionalities were tested individually and then together to be sure that every module was working before more integration thanks to the C++ object oriented design.