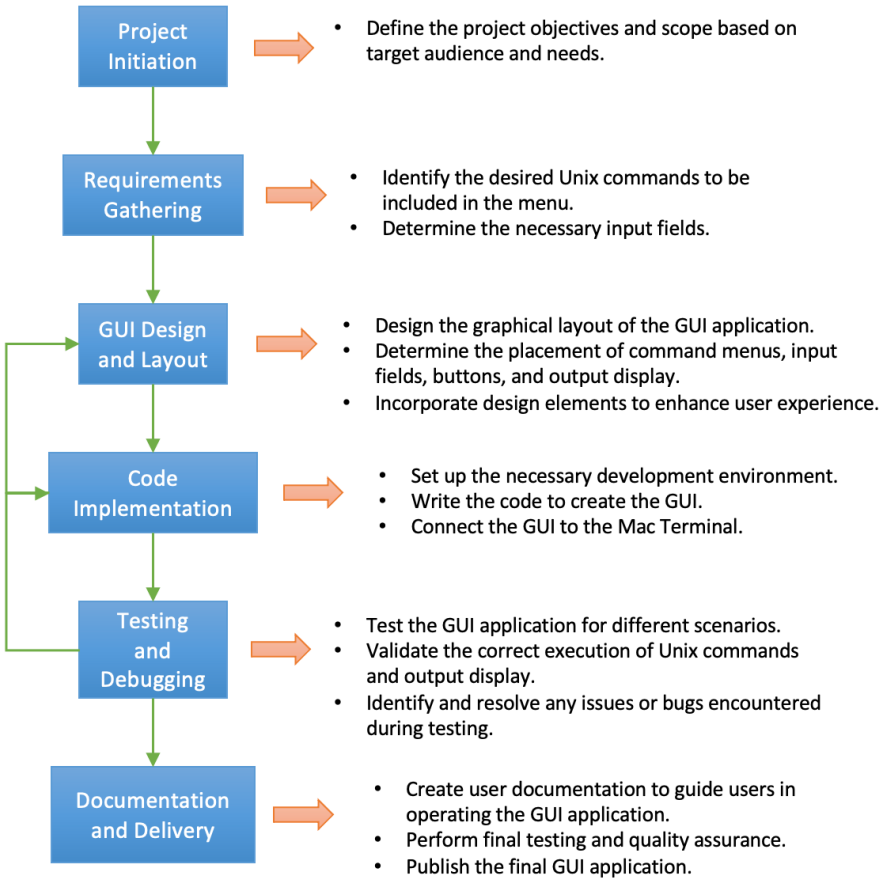
Sravya Kuchipudi
Professor Kaisler, Hasanov
CSCI 6917
10 July 2023

Midterm Project Report: Building a GUI for Unix Commands Execution on MacOS Terminal

The goal of this project is to provide a straightforward graphical user interface (GUI) that enables users to choose Unix commands from menus, enter the required parameters, run the commands, and show the results. This GUI offers a simple and user-friendly alternative for MacOS's command prompt for executing Unix commands.

This project can be considered as using a qualitative strategy since the focus is on creating a graphical user interface (GUI) to facilitate user interaction with Unix commands, rather than performing extensive data analysis or numerical computations. The emphasis is on improving the user experience and providing a user-friendly interface for executing commands. The qualitative approach allows for in-depth exploration of user needs, preferences, and feedback to inform the design and development process. The research strategy emphasizes understanding user requirements, gathering feedback, and iteratively refining the GUI application. The plan developed for this research strategy is presented in the following flow diagram, outlining the key steps involved in developing the GUI application. The blue block arrows indicate more details about each step while the thin orange arrows indicate the flow of work with loops included where relevant.



The first few weeks were taken to better define the goals and bounds of the project as well as testing different tools and libraries that can be applied. Due to an existing proficiency in Python, it was decided there was a need to then research a few different libraries to figure out the best approach. Originally, PySimpleGUI was chosen for the project since it is relatively easy to use which lowers the learning curve. It's a cross-platform library, which means it supports multiple operating systems, including MacOS, which aligns with the project's goal of building a GUI for MacOS, so a GUI application that works consistently across different platforms without requiring significant modifications can be developed. Since it is customizable, the existing widgets could be used in several ways to address the needs of this project. While these are the original reasons that PySimpleGUI was used for the project, it is important to highlight that it does have a rather blocky and simplistic nature which makes it difficult to blend into the design style of MacOS.

Creating the PySimpleGUI test code to work on formatting took a few different steps. After importing the library, the GUI layout was defined using PySimpleGUI's layout syntax. There is a window created that serves as the main structure for the GUI components. Then an event loop is entered where it waits for user interaction or events to occur. Within the event loop, the code handled specific events such as button clicks or window closures. If the user clicks the Execute button, the code retrieved the command number and arguments entered by the user. To demonstrate the functionality, a placeholder message was generated as the output, however, as seen in the future implementation, the selected Unix command would be executed, and its output would be captured. Then the output display area in the GUI window shows the captured output, allowing the user to see the results of the executed command.

The initial layout followed a rather simple format with clear sections for the command selection menu, argument input, execute button, and output display. The format was clear and intuitive to allow users to easily identify and understand the various GUI components. Input prompts for the command number and arguments made it clear to users what information is expected, reducing confusion, and ensuring accurate input. The format also included a designated area for displaying the output, providing users with convenient access to the results of the executed command. This compact design was useful for initial testing since it optimized space usage, resulting in a clean and usable interface without overwhelming the user with excessive clutter. While there were later changes to improve functionality and aesthetics, this was the initial choice to cover the different requirements of the input and output.

The next stage was to connect the graphical user interface (GUI) to the Mac Terminal to allow for integration of user interactions and command execution. For this, a few different approaches were researched to determine the best fit. The main approaches that were reviewed were the subprocess module, AppleScript, and the pty module. Each option had its own advantages and considerations which was why this was a needed portion of the project to figure out which would be the best to integrate.

The first potential approach involved using AppleScript, a scripting language provided by Apple for automation on macOS. With AppleScript, the user can directly send commands to the Terminal application. The integration between GUI and Terminal is achieved by creating an AppleScript script that takes input from the GUI and executes it as a Terminal command. This approach eliminates the need for a subprocess, making it a lightweight solution. Since it is specific to MacOS and requires knowledge of AppleScript syntax this was an added portion of work that was subsequently considered as not efficient for the current time constraints. Another approach was the pty module which offered a different approach by allowing pseudo-terminal utilities. Instead of executing a separate Terminal process, a virtual terminal session is created using the pty.openpty() function. This approach allows for interaction with the Terminal directly from the GUI application. The pty module provided greater control and flexibility, but it required a certain amount of additional code to handle the output and update the GUI accordingly which was not an efficient focus. Finally, the subprocess module in Python provided a straightforward method to execute

commands in the Terminal from the GUI application. By using the subprocess.run() function, input from the GUI can be passed as command-line arguments to the Terminal. This approach was simple to implement and leveraged the existing functionality of the operating system and therefore was considered as the best choice for the project.

After choosing the best way to integrate the GUI to the Terminal, there was brief shift to choose which of the top Unix commands would be best to add for the testing phase. Using help from the articles and readings provided by Dr. Kaisler as well as other collected knowledge and research, six different Unix commands were chosen to initially work with on the GUI. The first choice was ls which is used to list the files and directories in the current working directory. This was a necessary inclusion since it would help facilitate testing and checking the other commands. Then commands that would help manipulate directories were chosen. The cd command is used to navigate the file system by changing the current working directory in the Unix shell by taking the desired directory as an argument. The mkdir command is used to create directories or folders in the file system by taking one or more directory names as arguments and creates them in the current working directory. Finally, some commands that can be used on files or directories were chosen. The cp command is used to copy files and directories from one location to another by taking two arguments: the source item name and the destination name . The mv command is used to move or rename files and directories by taking two arguments: the source item name and either the new destination or item name. Lastly, the rm command is used to remove or delete files and directories by taking either the file name or -r directory name as the argument.

The next part was to finally integrate the GUI with the Terminal so the functionality could be tested. For this the subprocess module was chosen to provide a straightforward method to execute commands in the Terminal from the GUI application. By using the subprocess.run() function, input from the GUI could be passed as command-line arguments to the Terminal. Basically, this was done by creating a run_command function that takes a command as input, runs it in the Terminal using the subprocess.check_output method, and returns the output.

While it was not too difficult to code this, during the testing phase it was noticed that the commands were running as individual shells at each execution. Basically, if the cd command was run to change the directory and then the ls command was run to check for proper execution, the shell would still show the original working directory and not the selected directory. From here some research and experimentation was needed to figure out how the execution could be done properly. Instead of running each command as a separate process, a single shell session was tested to maintain the state of the current working directory using the initialize_shell() function, which creates a subprocess running the Bash shell. This approach kept causing the GUI to not respond and attempts at troubleshooting it were causing other errors so some more investigation was done to find a different way which led to the os module. The os module is a built-in python module that helps interact with the operating system. The os module was used in conjunction with the subprocess module to fix the issue with changing directories. When the selected command is cd, the os.chdir function is used instead of subprocess.run because changing the directory of the current process is not possible using subprocess.run alone. The os.chdir function allows the current working directory to be changed within the current Python process.

At this point the GUI had been created and linked to the MacOS Terminal such that a handful of commands could be successfully executed. However, the appearance of the user interface was rather simplistic and blocky, so it was decided that a change was necessary. For PySimpleGUI there was not much a designer could do to change the aesthetic other than the color themes or placement of button, but another more widely used tool library, that was the original second choice for a GUI package, was tkinter. Although the code looked technical it was mainly just formatting, so while there was a bit of practice that was needed, there was a lot of material online that helped with picking up the difference in the syntax

quickly. Also, the structure of the integration using subprocess and os as well as the commands dictionary didn't really need to be changed, so the main work was just creating the window.

From this point the project was focused on adding and improving different features in the graphical user interface GUI since there were testing capabilities to assess that the additions were working. There were several features that were important to add but some of the main ones were the separate output window, templates for commands, and additional functionalities. One significant addition to the code was the introduction of a separate output window. Previously, the output was displayed directly in the main GUI window. However, displaying the output in a separate window offered a few advantages since it allows for a cleaner separation between input and output making it easier for users to focus on their commands and view the resulting output without any visual clutter. It provides more space for displaying the output compared to the limited area within the main GUI window, Also, it enables the user to interact with the main GUI while viewing the output simultaneously. This output window was implemented by creating a new Toplevel window. Within this window, a Text widget was added to display the command output. The widget was also configured to be read-only which ensures that the output cannot be modified by the user either accidentally or otherwise. One small error that occurred when initially coding this was that each command execution led to a new output window being created. So, to ensure that each time a new command is run, its output is appended to the existing output window, the output_text widget was made a global variable so that it could be accessed and modified from the run_button_click function. Within the run_button_click function, the output window was configured to allow modifications, and the output of the command was appended to the output_text widget. After appending the output, the widget is set back to read-only mode to prevent any user modifications.

The command's dictionary in the code was updated to include additional information for each command. Each command now contains a description and a template. The description provides a brief explanation of the command's functionality, while the template displays the command syntax, indicating how the command should be used with its respective arguments. These additions enhance the usability of the GUI by providing users with clear information on each command's purpose and usage. The update of the command's dictionary also led to the addition of a command information window. This window provides users with descriptions and templates for each available command, helping them understand the purpose and usage of each command reducing the likelihood of errors and facilitating their interaction with the GUI. While it isn't overly informative, it serves as a quick reference and guide, improving the usability and effectiveness of the application. When the Command Info button is clicked, a new Toplevel window is created. Within this window, a Text widget is added that holds information about each command, including the command description and template for use. The information is retrieved from the commands dictionary in the code, which stores the details for each command. Overall, these decisions were made with the aim of enhancing the overall usability, functionality, and user experience of the Unix Commands GUI. By considering user needs, interface design principles, and the specific requirements of a command-line interface in a graphical environment, these choices contribute to a more intuitive, efficient, and informative application. The addition of an output window allows for better separation of input and output, the command information window provides users with valuable details about each command, and the updated functionality ensures that the output is appended to the existing output window for each new command execution. These improvements make the GUI more user-friendly and informative, facilitating the execution of Unix commands in a graphical interface.

After adding these features, a few more commands were integrated into the commands dictionary. The chosen additional commands were pwd, cat, more, head, and tail. The pwd command stands for print working directory and when executed, it displays the full path of the current working directory by the user simply typing pwd in the terminal. It is essential for understanding the current location within the file system hierarchy and helps navigate through directories efficiently. The cat command, short for concatenate, is used to display the contents of one or more files on the terminal and is often used to read

text files or to combine multiple files into one. The word cat is used with the file name(s) so cat file.txt will display its contents. The more command is used to view the contents of a file one page at a time and can be executed by more file.txt. The head command is used to display the first few lines of a file by entering head file.txt. By default, it shows the first ten lines, but a different number can be specified (such as head -n 5 file.txt to display the first five lines). The tail command is similar to head but displays the last few lines of a file using tail file.txt. This command is frequently used for monitoring log files, where the latest entries or changes appended to the file can be observed and helps track real-time updates without having to open and read the entire file.

The next steps of the project at this point are to improve the error checking mechanisms. To enhance the code by adding error checking, several steps can be taken such as input validation and detailed error responses when possible. First, input validation will be implemented to ensure that the user provides valid commands and arguments. This can involve checking if the entered command exists in the commands dictionary and verifying that the arguments meet any required format or constraints. Next, potential errors that may occur during command execution will be handled. For example, when using subprocess.run() the generated process error can be displayed as an appropriate error message to the user, providing details about the error that occurred. Additionally, specific file-related errors might need to be handles. When using file operations, such as opening or reading files, there can be catch exceptions like FileNotFoundError or PermissionError to inform the user about the specific file-related issue encountered. It can be beneficial to add these informative error messages through the GUI application rather than just generated into the IDE as they currently are received. Displaying error messages in the output window or using message boxes can help the user understand and troubleshoot any issues that occur during command execution and help them learn more about Unix commands to provide a smoother experience and better feedback to the user in case of errors.