Design Document

Cayden Lund (u1182408)
Eduardo Valdivia (u1105168)
Jaden Gill (u1259060)
Preston Hales (u1171135)

CS 4000—Senior Capstone Design
6 October 2023

1. Executive Summary

Commander was born from the frustrations we experienced with traditional scripting environments, especially those relating to command line automation. Common command line scripting environments include Bash and Powershell. While great for their direct connection to the command line and using/parsing command outputs, they also have a limited set of actions the user may perform, and not very friendly syntax. And for Bash, the language is not strongly typed which makes things confusing for programmers coming from a more structured language, and makes tasks such as doing math more involved due to having to parse strings. In comparison, programming languages targeted toward a general audience like Python and Perl are powerful with better syntax and typing, but lack the simple directness of a command line scripting environment and easy parsing of output from commands.

The first of many differences with Commander, compared to its contemporaries, is the language's syntax. The goal of Commander is to be a simple language with an interpreter, like Python, but include additional syntax for user clarity. Much of the syntax is going to take inspiration from C and its derivative languages like C#, Java, and Javascript since these languages are pretty familiar to most programmers, thus allowing for a less steep learning curve for people new to Commander. Commander allows users to quickly develop scripts for their projects while maintaining readable code. Moreover, language interpreters such as Python's are notorious in certain groups for a variety of reasons, often low performance. The Commander interpreter is written in C++ for faster interpretation and execution time.

Similarly, scripting environments are primarily exclusive to a specific platform. Powershell, for instance, is included by default on Windows as the go-to scripting environment. While it is possible to install and run Powershell on Linux, one must go through the process of installation, and even once installed will still have to use Bash to utilize certain dependencies tied exclusively to it. One benefit of Commander compared to other environments is its ability to be run on multiple platforms, including Windows, macOS, and Linux, via the multi-platform interpreter and run commands independent of the primary scripting environment. Additionally, no additional dependencies will be required to run the interpreter using the pre-built executables that will be released for each version of Commander on the GitLab page. Upon completion, the project will also be released as open-source under the MIT license, allowing other users to compile the application on their machines or make contributions as they feel inclined. If time permits, we will also create plugins for various text editors and IDEs in order to provide nice features including syntax highlighting for the Commander language. The result is ultimately an environment that can be built and run anywhere the user would like with the tools needed to easily program their Commander scripts.

Our goal for Commander is to provide a scripting environment which resolves the aforementioned problems that present implementations have. Users will want to use Commander due to its accessibility and ease along with the benefits of a traditional programming language. It is our hope that Commander will enable veterans and new command-line users to develop the command line scripts that they desire.

2. Background and Technical Requirements

a. Similar Languages

Projects similar to ours include *AngelScript*, *Bash*, *Batsh*, *Lua*, *Perl*, *PowerShell*, and *Python*. A brief overview of each of these languages is provided below.

Most of the languages we have observed attempt to provide general purpose features rather than emphasize command execution. Meanwhile, the other scripting languages for command-line programs tend to have unintuitive syntax with a steep learning curve. In the case of Batsh, which does both of these things well, the creators only provide a transpiler to Bash/Batch, so the code cannot be directly executed across multiple operating systems as it can with an interpreter. The goal of Commander is to be a scripting-first language that users will find simple to understand, while offering a speedy interpreter that will be able to execute commands across the three main operating systems: Windows, MacOS, and Linux. A summary of the languages we have observed are as follows:

AngelScript: A flexible scripting library designed to be functional through external scripts. It was designed to be easier for both application and script writers. It implements types like C++ and has similar syntax. A caveat we have identified is AngelScript's usage in game development. In other words, the language is designed with a specific use case in mind.

Bash: Most commonly used in Linux systems, Bash is the GNU Project's shell (*B*ourne *A*gain *SH*ell). It encompasses both a shell environment and a scripting language that allows for users to automate their daily tasks through the shell. Its steep learning curve and confusing syntax may result in unreadable script files. Additionally, it is weakly typed, which makes certain things like math operations more difficult to program.

Batsh: A command-line scripting language with C-like syntax which transpiles to either Bash or Windows Batch. There exists an online tool provided by the creators as well which allows you to write a Batsh script and it will output the resulting Bash/Batch translation. However, it lacks an interpreter or other methods for directly executing the code.

Lua: A scripting language that supports object-oriented programming, functional programming, data-driven programming, and data description. It is used in a wide range of applications like games, web, and image processing. Similarly to Python, Lua is often used as a general-purpose programming language. The language's interaction with the command line for any given operating system is unrefined.

Perl: A general-purpose interpreted scripting language. Its own philosophy is to enable developers to quickly create and test their code. Like most general-purpose scripting languages, interaction with the command line may be indirect or discouraged.

PowerShell: A modern shell-scripting language developed by Microsoft. It is similar to Bash in many respects, and shares some of the same drawbacks. In short, it can directly execute commands well, but its syntax is verbose and not very readable, and it is primarily limited to Windows. (It can run on other platforms, but has some compatibility issues and many dependencies.) One of PowerShell's benefits over Bash is that it is strongly typed, which makes it easier to avoid bugs and do certain things like math.

Python: A general purpose programming language with high emphasis on code readability. The language is very general-purpose, though, and lacks the directness for automating the command line (that is, it takes a lot of code just to run a single command). Other criticisms include its generally slower performance compared to other languages and its used of indentation for defining scopes.

b. Required Technology

While we develop Commander, we intend to use the following technologies:

C++: We plan on programming the interpreter in a low-level language, both for direct access to system files and streams and for runtime efficiency. We selected C++ as our team is all familiar with the language.

Unit Testing: We plan on writing extensive unit tests in order to ensure the accuracy of the Commander interpreter. These tests will use the GoogleTest unit testing library for ease-of-use, reproducibility, and reliability.

File I/O and Commands: The interpreter will require access to and utilize the file system on the operating system of choice when it is run. Not only will this be required to read in and run the script files, but to also find and execute the referenced commands that are stored as executable binaries in the system.

Text Editors/IDEs: We plan on utilizing powerful text editors and IDEs such as Vim, Emacs, VSCode, and CLion in order to program the interpreter. These editors often provide many tools and plugins such as Intellisense, syntax highlighting, and debuggers to help make programming easier and less error-prone. If time permits, we will write plugins for these editors in order to help people programming in Commander to have access to some of these same features.

GitLab/GitHub: For version control, we will use GitLab. GitLab has many tools for managing features and goals the project will have. Upon completing CS4500, we will move the repository to Github as an open source project for future version control. This

choice is due to the sheer number of users working with Github, which will allow more developers to have easier access to the project.

c. Software/Hardware Requirements

In order to compile and build the interpreter, a C++ compiler such as g++ or clang++ is required. However, pre-built executables will be provided, so there are no software requirements or dependencies that clients will need to install. Commander will be multi-platform; that is, it will run on Windows, MacOS, and Linux computers. The development focus will be on making it work on x86-64 architectures, but it will likely be portable to ARM or other architectures as well. There are no further anticipated hardware requirements.