SCFE: Project Report

Matthieu Stombellini

Mathieu Rivier

François Soulier

Rakhmatullo Rashidov

Contents

1 Introduction

This is the final report for our second semester project: SCFE (Salamanders' Console File Explorer). It provides details on the original idea, the entire process of making the project, as well as a description of the end result.

1.1 Technical terms used in this document

This document might be read by people who are not accustomed to terms related to software engineering. Here is a short list of understandable definitions for the words we will use throughout this document:

- Library: a set of functions, methods and other resources created by developers for other developers to use
- Framework: a library with a much more consistent and complete base, often used as a scaffold for an entire application
- Implementation: a way to code definitions of functions. For example, if definitions were the abstract of a book, the implementation would be the actual content of the book.
- `.NET`: a set of frameworks, languages and related tools by Microsoft for creating applications
- `.NET Framework`: the oldest implementation of the `.NET` standard, which only runs on Windows
- `Mono`: a third-party project that aims to make `.NET Framework` available on non-Windows machines
- `.NET Core`: a relatively recent implementation of the `.NET` standard that is fully cross-platform and much faster than `.NET Framework`
- `C#`: a programming language made for `.NET`. Other languages created for the `.NET` technologies include `F#` or `VB.NET`

- `UI` and `GUI`: respectively User Interface and Graphical User Interface

1.2 General description of the project

The goal of SCFE is to provide a file explorer that is easy to use for everyone, from beginners to more advanced users.

The idea was born out of a collection of problems from both sides of the usual way of manipulating files: on the one hand, graphical file explorers (like Finder on Mac OS or the File Explorer on Windows) are very clear, provide a lot of information, but are fairly slow to use since they require to use the mouse, and can also be a bit limited when it comes to having more advanced functionalities. In short, it is a great option for beginners and people who like to be in a fairly painless environment. On the other hand, consoles and terminals are very popular options and, although they are not file explorers per se, they are still used as such. They contain all of the different features of file explorers and much more, while also being extremely useful for productivity, the next action only being a few key presses away. It is a great option for power-users who love to get things done fast and do not mind having a quite steep learning curve, but is a terrible option for beginners, who have to learn a lot of new commands and have to deal with a certain lack of clarity in most places, where you have to search on the Internet to understand how to do something which you could do in only a few clicks on a regular file explorer.

We decided to go ahead and create a tool that would provide the best of both worlds: a tool that is easy to use yet allows you to get things done very quickly, is easy to learn, and always has your back in case you forgot how to do something. It is also fairly intuitive and does not require much knowledge to be able to access all of its powerful features.

All in all, we wanted to create a piece of software that was primarily useful, something that we wanted to use ourselves. Because we find it useful for ourselves, we believe others will find it useful as well.

To increase this "usefulness factor", one of our goals was to make SCFE cross-platform, meaning that it could be used on any platform: Windows, Mac OS and Linux. This was a major constraint and caused some issues during the realization of the project.

These are the core values that we followed when creating SCFE, and the end result respects them as much as possible.

We decided to build SCFE as a console application for multiple reasons:

- It made cross-platform compatibility easier and, overall, consoles are available on any computer, meaning that we had almost no dependencies other than the core framework
- It gave us almost unlimited flexibility
- It was much easier to work with compared to full-blown graphical interfaces, and generally provides a smoother and faster experience, due to how bare-bones a console is
- It provides better integration for users used to the console environment. For others, it can still be opened with the app icon.

The very core of SCFE is its modal architecture. There are three main modes of operation for the application, which we developed and expanded upon throughout the project. The modes themselves are described in the Final Results section as well as in the Book of Specifications.

1.2.1 Features

Here are some features which we wanted to implement:

- Opening files based on the system's default application (e.g. opening .docx files with Word)

- Compatibility with basic file operations (e.g. renaming, deleting)
- Multi-file selection support and copying and moving multiple files at once
- Support for directly going to a specific folder
- Cross-platform compatibility for Windows, Mac OS and Linux
- Compatibility with console commands
- Compatibility with Git
- Multiple modes system: `NAV` mode for navigation, `SEA` mode for search, `COM` mode for entering console commands
- Easy to use, intuitive shortcuts
- Simple yet complete documentation
- Overall smoothness with as little hiccups as possible

1.2.2 Technical details

SCFE is based on the `.NET Core` framework and is written in the `C#` programming language. We chose the former for its great cross-platform compatibility, and the latter because it was the closest to other languages we had experience with.

It also uses additional libraries which are listed below:

- `JetBrains.Annotations`, a library that gave us hints while developing and allowed for better error detection. It is only used while developing and has no impact for the end-user.
- `MoreLINQ`, a library which provides more shortcuts for dealing with data collections in `C#`. It made it particularly easy to code requests like "Get me the maximum value of this sequence", which were usually reduced to a single line of code thanks to this library.
- `LibGit2Sharp`, a library which we used for providing integration with the Git version control system (VCS) tool. It is the de facto official library

for dealing with Git repository under the `.NET Core` framework.

It was coded using the Rider IDE, and tested under Windows and Mac OS as well as Linux through the WSL system on Windows.

For distributing the application, we chose to have different solutions available:

- An installer for Windows, with the `.NET Core 3` framework included
- A full version for Mac OS X and Linux with the `.NET Core 3` framework included
- A light version for Windows that only includes the barebones `.dll` files and an `.exe` for launching the application. This version requires the correct version on the `.NET Core` framework to be installed on the machine.

2 Book of Specifications follow-up

This section will describe the exact changes and tweaks compared to the book of specifications.

We have respected the book of specifications as much as we could throughout the entire project. Our priorities when designing and coding the application stayed the same, and we implemented all of the features we wanted to have.

We were not sure of which library to use for showing the user interface, which is why we decided to create our own, based on the simple yet reliable Console API provided by `.NET Core`. The project being written in C#, we original went for `.NET Framework` on Windows and `Mono` on other platforms, but the amount of cross-platform compatibility issues, especially for detecting inputs, led us to prefer `.NET Core` for the task.

The library we created, which we named `Viu`, is detailed in later sections of the document.

Additionally, two other libraries are actively used: `LibGit2Sharp`, which provides easy way to interact with Git repositories, and `JetBrains.Annotations`, which was mostly used for internal quality assurance, as it allowed our editors to warn us when writing potentially crash-prone code.

While nothing extreme changed, in order to maximize user productivity, we had to make a few changes on some of the details.

- The functionality for going down or up fast in files (through the Shift key) was changed: instead of going 10 by 10, it now acts as a “page up”/“page down” modifier, moving the selection cursor at the bottom of the screen, focusing the first element that was not visible (or just to the last element).

- We announced that the `COM` mode would allow to change the current directory. We found that this would be too complex to implement and a bit too long: we created a dedicated shortcut instead for the "go to folder" functionality, which allows the user to go to any folder they wish. It also takes less time: with the `COM` mode: you have to press Ctrl+Enter to enter the mode, type "cd" then space, then type your path, for a total of 5 additional key strokes beside the path itself. With the shortcut we created (G or Ctrl+G), this number is reduced to either 1 or 2 key strokes. It is more efficient and more reliable as it does not depend on any underlying shell, hence making it a better solution than what we planned in the book of specifications.

- Selecting files was planned to be with the S/Ctrl+S key. We decided to also add the Space key for selection, since it was a big, easy to reach key that would have otherwise been unused

- We originally planned to completely block the screen when big files or folders were being moved or copied. This idea was very simple to implement, but we decided to go for a more complex but more efficient solution: the operation is handled on a separate thread, which then sends back information to our user interface thread to notify the user that things are happening in the background.

- We specified that we wanted to add color schemes to the application, but we did not say in the Book of Specifications how we were going to switch between them. Since we only have two color schemes at present (the regular one which differentiates folders, files, hidden folders and hidden files and the one for Git, explained later on in this report), we decided to simply automatically switch between them. Hence, the Git color scheme is used when in a Git repository and the regular one is used everywhere else.

- Finally, while we respected most of the key shortcuts we announced, we modified some of them: e.g. instead of Ctrl+T to change the file sorting method and Ctrl+O to toggle between increasing order and decreasing order, we are usin Shift+S for the former and Shift+Q for the latter. Moreover, while we did implement E and Ctrl+E as the shortcuts to switch between modes, we decided to also add M and Ctrl+M for the same purpose, the reasoning behind it being that the M key is close to the HJKL keys which can be used in SCFE to navigate between files.

3 Evolution of the project

This section will go over the development progress of the application. It will be presented by "step" (e.g. progress before the first presentation, between the first and the second presentation...), then detailed for each task. As a reminder, here are the tasks for our project and the progression we originally planned in the book of specifications.

Task	Presentation 1	Presentation 2	Final Presentation
Core UI	70%	90%	100%
Navigation modes	20%	70%	100%
File I/O	30%	70%	100%
User input	40%	50%	100%
Tool & OS integration	0%	30%	100%
Website	0%	50%	100%
Documentation	20%	50%	100%

- Core UI corresponds to low-level UI work, basic components and the overall development of the Viu library which SCFE is based on.
- Navigation modes corresponds to the implementation of the various modes that can be used to navigate through the app
- File I/O corresponds to the implementation of operations on files
- User input corresponds to the handling and routing of key presses from the user
- Tool & OS integration corresponds to the interoperability with existing tools and systems
- Website corresponds to the development of the website
- Documentation corresponds to the writing of the documentation for our application

3.1 Before the first defense

The goal of this time period was to get everyone accustomed to the various tools they would have to interact with over the course of the next few months, as well as creating the various bases that we needed early: the Viu library and the files system we wanted to have in place were by far the most important steps.

3.1.1 Core UI

In order to actually build a console application, we needed to have something that could provide an easy way to display components, as well as handling inputs exactly as we wanted. Unfortunately, almost all of the C# libraries we could find lacked something crucial: it was often flexibility or cross-platform compatibility issues which made libraries unusable in our project.

As a reminder, cross-platform compatibility was one of our priorities with this project. Originally, two out of the four team members were Mac users, but another member also replaced his Windows machine with a Macbook. With 75% of the team actively using the Mac OS system, it was obvious that cross-platform compatibility simply had to be there.

For the examples we mentioned in our book of specifications:

- The `gui.cs` library, while perfectly functional, was also extremely unclear, and not flexible enough. We also had doubts about its cross-platform capabilities.
- The `CursesSharp` library was just too low-level to be efficiently used, and also added numerous concerns over the portability and cross-platform capabilities of our code, the library requiring very specific bindings to

system libraries which differed depending on the platform (UNIX or Windows).

This is why we decided to make our own library, based solely on the `Console` API available in `.NET`. It is a simple, straight-forward and (mostly) painless set of functionalities that were tried and tested. The only downside was that it was a bit slow compared to other methods which would call system libraries directly.

This sub-project of SCFE is one of its most important components: in order to match the flexibility and performance requirements we had in the Book of Specifications, a lot of work had to be done to make it as smooth as possible. The library was given a name, `Viu`, and was completely separated from the code of the application itself. The idea was that `SCFE` depended on `Viu`, not the other way around. While this separation might be seen as a step away from the original goal of the project, it is very much the opposite: the way in which we built `Viu` makes implementation of more high-level features of SCFE way easier.

The `Viu` library is heavily inspired by the Swing toolkit in Java: the validate-then-print (i.e. plan where everything is, and show them after everything is placed) workflow, component hierarchy and layout strategies resemble Swing, but no code has been taken from it, and, when actually creating interfaces with `Viu`, only some of the functions share characteristics with the Java system.

The main ideas developed throughout this first period were the creation of visual components (labels, text fields, tables, buttons...) as well as the implementation of various layout strategies. All of this was made harder by the fact that the interface always has a varying size, and layout strategies had to be flexible enough such that we would never have to touch low-level layout code and manipulating position coordinates directly when building SCFE on

top of `Viu`.

Thanks to this and some light multithreading to catch when the window is resized, components dynamically resize themselves as the space around them changes.

`Viu` includes multiple "layout strategies", allowing us to have components shown exactly as we want them to be. All strategies are able to smartly place components in them, always ensuring that they perfectly fit in, dynamically wrapping their components if necessary. The strategies illustrated in figure 1 are:

- The Border Strategy, laying out 4 components at each border and one in the middle
- The Line Strategy, organizing components into a horizontal or vertical line
- The Flow Strategy, putting components one after the other, wrapping them like a text if necessary, although components are of course not restricted to text only and can be any `Viu` component.

The most useful components were created, including simple texts, text fields (which listen to user input), buttons and tables among others. As with layout strategies, they can all dynamically resize themselves. Tables can even resize each column individually with different widths for their content (provided that it has different sizes at its disposal) in order to either grow or shrink.

For extra flexibility, and if we wish to step away from the basic Console API for something that allows us to have a more fine-grained control over the output, `Viu` comes with its own abstraction layer above any console related code. Components never call the `Console` API directly, only calling our abstraction layer, for which the concrete implementation is given by a simple pass-through

to the basic `Console` API.

`Viu` components were built by Matthieu, who had experience with the Swing system (hence the resemblance), for the code that lays component out, the general appearance of components and the overall hierarchy of the `Viu` system. The user input side of `Viu` is described in its own task.

3.1.2 Navigation modes

We built `Viu` with very few shortcuts available right out of the box. We only added the essential obvious ones (e.g. arrow and enter keys). A few additional shortcuts were added, making use of the Input Map and Action Map systems described in the User Input section. For the Action Map, most of the work done was hooking up a few shortcuts to action names in the NAV mode, as well as preparing the table used in the prototype to receive constantly changing input bindings. This was *not* fully implemented as it was unnecessary at this stage, but has proved itself to be a fairly thorough test for the input system, to see how flexible it would be.

The existing shortcuts were mapped to their corresponding action name, in a dictionary, ready to be implemented into the main application for when it would be out of the simple prototype state. This task was realized by François.

3.1.3 File I/O

While file input and output is an essential part of SCFE, it was far from being a priority for the first period. We planned to primarily focus on the basis of what has become `Viu`, making everything else later.

As such, François coded an object-oriented representation of the file system entirely based on the various classes provided by `.NET`. Unfortunately, these

were either only using static methods and strings, or did not have enough features to satisfy our needs. This meant that instead of code that would look rather explicit like `myFile.GetName()`, we would have much less clear code like `Path.GetName(myPath)`. Moreover, the errors sent by the `.NET` classes were either non-explicit or unpredictable.

Having our own implementation also meant that we were able to copy and paste files or even entire folders using a simple function in the file representation. (e.g. `myFolder.CopyTo(somewhereElse)`). This was a feature that was surprisingly absent from basic `.NET` classes.

In addition, the methods implemented in this task such as `Copy` or `Move` use the System.IO classes in such a way that the File class from SCFE can represent either a file or a folder. That specific property was particularly difficult to handle in the implementation, because each and every case had to be taken into account, and thus imply a need for recognizing the kind of file (folder or file) that was being dealt with. `.NET` has two specific classes for handling either files or folders, making it impossible to just say "copy this thing to there". We would have to say "if the thing is a folder, copy it over there, if it is a file, copy it over there", which would be inconvenient seeing how file operations are essential to our application.

Our implementation thus merges `System.IO.File` and `System.IO.Directory` functions, with some added bonuses, like directly determining the folder in which a file is, or other handy shortcuts.

It is important to note that the functionalities of the File class were not used to their full extent at this stage: they were only used to display the content of files. We intended to use what was coded more extensively for the second presentation.

François was responsible for the entirety of the task, with outside help from

Matthieu for fixing a few bugs and properly using .NET APIs, since it required a deep dive into the documentation provided by Microsoft to handle each and every case properly.

3.1.4 User input

The main goal of this task for the first period was to provide interactivity for `Viu`: reacting to key inputs and managing focus states of all components.

A focus system was added, which allows the user to navigate between the various elements of the interface and throughout the component hierarchy using arrow keys (or other custom key bindings). This was done through smart use of C# interfaces on all components, which can declare the fact that they can be focused and handle key presses by simply implementing an interface. The focus system can be compared to the logic behind normal applications which allows them to determine which component to select next when pressing `Tab` or `Shit+Tab`.

To create this focus system, we simply extended layout strategies to also include logic that tells which component should be the next one to be focused when going left, right, up or down.

Another aspect of the user input task was the InputMap/ActionMap system. Once again inspired by the Swing system from the Java language, this allows us to separate shortcuts from actions. The idea is that `InputMap`s provide a binding between key presses and action names, while `ActionMap`s provide a binding between the action names and the actual action. This system provides a fantastic "buffer" layer between what the user presses and what the program does. This is the exact system that was used for the Navigation Modes task for the second period to switch back and forth between different shortcuts without fundamentally changing the actual actions themselves.

A diagram describing the process can be found at figure 2.

Finally, a few components which were entirely based on user interaction were added, namely text fields and buttons. These were entirely custom built by handling individual key presses, since we were not really able to rely on the "regular" way of reading text from the console. Text fields provide a sizeable implementation of shortcuts which are common in text editors in order to simply make the text field component easier to use. Examples of such key presses include the Delete key or Ctrl+Arrow shortcuts (to jump between words instead of just between letters/symbols).

Heavy testing for this part was crucial to make sure that the input system was robust.

Mathieu was responsible for most of the task, implementing the input system (under Matthieu's supervision in order to make it fit nicely into the existing component hierarchy) with ActionMap and InputMaps as well as the focus system (once again with help from Matthieu), and Rakhmatullo was in charge of the "input reaction" part of a few components, including buttons (e.g. performing a predetermined action when pressing a button).

3.1.5 Tool & OS integration

Nothing was done for this task specifically, as was planned in the book of specifications. It represented a more advanced part of the application, and was far beyond the simple "laying the basis down" step which we were in the middle of for the first period.

3.1.6 Website

Once again, not much had been done for the website at that time, the goal mostly being to prepare a basis for every other task. Mathieu did gather some information and made some mockups in order to gain some time for the second period.

3.1.7 Documentation

Most of the work that was done for this part was simply collecting intentions from the book of specifications to make them into a user-readable format. This is done in Markdown at the moment, but will be published onto the website later on.

Rakhmatullo never touched the Markdown format before, so he looked into it and at how to make it work for us.

3.1.8 Overall state

At the end of the first period, we had achieved quite a lot: we had a solid basis in the form of `Viu` as well as prototypes which showcased the various components we implemented. The goal was to have a something ready for a smooth transition into full functionality.

A prototype of the application was produced and is in figure 3. While it did not "do" anything, it was possible to go up and down in the interface and select files, but that was it. It served more as a demonstration of the capabilities of the Viu library we created than an actual product.

3.2 Between the first and second defense

Now that we had a basis for the application, including a home-made library for showing user interfaces in the console, it was time to create the real application on top of that.

3.2.1 Core UI

While building the main application, some unexpected bugs with the code we use to build graphical user interfaces in the console were encountered. The goal for this period was to, logically, fix them, as well as adding functionality here and there. Some components got "smarter" with a logic that properly handles being "squished". The components would behave incorrectly if they did not have enough space around them: each component normally expects and requests a certain amount of space to be attributed. If the console was too small, it was impossible for the component to print itself properly, and a number of issues could happen: either the text would "overflow" to the next line, or the entire application would crash with a terrifying (but admittedly pretty stupid) out-of-bounds error. This was fixed and most components now support being "squished" and can show an ellipsis (...) at a configurable spot: strings can be cut from the left (...like this), from the right (like this...) or from the center (like...this), depending and what makes most sense. Another component that was improved was the table component (which is used as the file list in SCFE): it is now possible to scroll through the list if there is not enough space to display it.

An additional improvement done in this period was the introduction of proper multithreading to the UI. In previous tests, all of the actions done on the interface were done from multiple processes at the same time. This kind of concurrency could be extremely problematic in the future and led to crashes

on Linux and Mac OS. In this period, all components were now forced to be printed on a single thread (which we will call the graphics thread), meaning that operations related to showing and refreshing the user interface were all done in a single process, eliminating a lot of bugs we had at the same time. The way it was done was to simply limit the access to the abstraction layer that was developped in the first period: components needed access to the layer to actually print things in the console, and we simply restricted the availability of said layer to operations done in the graphics thread.

The interface itself was entirely built using `Viu`: an excellent stress-test for the basis which grew stronger and more stable as a result. It was part of this task, but mostly consisted in putting everything together like Lego bricks.

The work on this task was entirely done by Matthieu.

3.2.2 Navigation modes

Two of the three planned modes were implemented: the `NAV`igation mode as well as the `SEA`rch mode.

The `NAV` mode, which is more like usual graphical file explorers, consists in using the arrow keys and HJKL keys to move around the interface. Many shortcuts were implemented to perform actions on the various files. Some examples of fully implemented and functional operations:

- Pressing R to rename a file
- Pressing N to create a new file
- Pressing Shift+N to create a new folder
- Pressing the Delete key to delete a file
- Pressing C to copy a file, X to cut a file and V to paste a file

In order to make the application more fool-proof, destructive operations are either not performed (e.g. pasting where a file already exists with the same name), showing an error, or ask for confirmation. This way, deletion requires the user to type "yes" and press Enter in the text box to ensure that they do want to delete the file(s) they selected.

The `SEA` mode also has these shortcuts, although they require using the Control key to access them (e.g. while you can simply press R to rename a file in `NAV` mode, you have to press Ctrl+R in `SEA` mode). This is because the regular letter keys do not redirect to the usually expected actions: they instead jump the focus to the text box at the bottom of the interface, and allow the user to enter letters for searching specific files. The user can then:

- Press Enter to either directly open the only file or folder that was found with this name
- If multiple files match the search keywords, pressing Enter will re-focus the file area instead and let the user choose which file they actually want.
- Pressing Escape cancels the search, empties the textbox and returns to the file list.

The `SEA` mode corresponds more to what console users expect: quick access to files by using the file names. It should be noted that the search performed is not recursive: only files in the current folder are searched. This is because the whole point of the `SEA` mode is to navigate through files as fast as possible by entering their name, and searching throughout the entire tree of files would be extremely tedious and slow down the application as a whole. In order to avoid this (and also avoid confusion from random files buried deep inside folders surfacing in a simple search), entering text while in `SEA` mode only searches in the current directory. In this way, it is better to think of the `SEA` mode as a "file view filter" rather than an actual heavy search mode.

Almost all of the actions described above are supported for multiple files: pressing Space or S allows the user to select files, and most actions will act accordingly (e.g. copying multiple files to the clipboard).

All of this was implemented using the Input map (turning keys into action names) and Action map (turning action names into actual handlers which perform the action) system which is described in the part about the progress done before the first defense. We now had a very flexible system which is able to hot-swap key bindings while leaving the actual actions untouched.

Moreover, actions such as pasting or deleting can take a lot of time. In order to properly support these, we used multithreading to perform the operation on one thread while allowing the other threads to handle the user's inputs freely. The thread in which the deletion happens then tells the graphical thread to either print a message, or reprint the entire folder if the content changed.

As a side note, an interesting thing to note is that on the dev team we each have our own preferences on how to navigate in SCFE: some prefer the `SEA` mode for quick access to most files while others prefer the `NAV` mode for the ease of use and quick access to actions!

This task was done by both Matthieu and François. Since Matthieu was mostly responsible for the Core UI task and François for the File I/O task, Matthieu took care of linking the navigation mode to the interface components as well as the most complex elements like multithreading, while François took care of linking the various actions to their implementation. All actions have two "branches": showing the user that some action is happening or has happened, and actually doing the action on the files. Matthieu did the former, while François took care of the latter.

3.2.3 File I/O

In order to have a more complete display for the various properties of each files, the current implementation had to be able to get more information from the files, e.g. access to the last modification date.

The system also needed to sort and filter files for a few reasons:

- The order of the files returned by the `.NET APIs` is not stable nor practical, and can be quite chaotic for large folders.
- *ALL* files are shown by default, meaning that system folders that are not even readable by anyone but the system were visible, which ended up adding more clutter to the UI and causing more confusion as to which files were accessible and useful.

We, therefore, ended up with a pretty flexible system of re-routable filters and "smart sorting".

- The base filter for showing files depends on a few variables, including whether hidden files should be shown or not, or whether we are in `SEA` mode and need to apply the search terms or not. Detecting hidden files is done in both the Windows and Linux/Mac OS way at the same time: the file can have the "Hidden" attribute (like on Windows) or can start with a dot (like on Linux and Mac OS).
- The sorting conditions we use take into account a few attributes: first whether the file is a folder or an actual file, and then the names of the files. This is not customizable at present, but we might allow the user to also choose whether they want to sort by size, modification date, or other.

Color schemes were also implemented. They simply consist in differentiating files (in white) from folders (in green) and hidden files (darker shades) from regular files (regular shades).

As a side feature, the File implementation got a new system for creating relative representations of file paths, which is explained in the "Tool & OS Integration" part.

This task was done by François.

3.2.4 User Input

The text box at the bottom of SCFE is the basic way for the user to interact with the system. It is based on the same text box component as before, and most of the work done this time was correctly focusing and defocusing it in order to avoid having the user wrongly inputting text in it.

When the text box receives an "enter" key press or receives a new letter, it performs actions that depend on the state of the application: if an action is in progress (e.g. the user is entering the new name for a file because he triggered the "rename" sequence by pressing `R`), then the text box redirects the input to the current action handler, which is different for every kind of action. If no actions are in progress, then the text box either does nothing (in `NAV` mode) or filters the file view (in `SEA` mode) just like any search would. In `NAV` mode, the text box is usually not accessible, and if it is, it has no real action (except when actions like renaming are in progress of course).

Moreover, the "input system" (i.e. input and action maps) was hooked to the table in the application with this task, allowing the user to actually do things with his keyboard. However, actually defining the different bindings was not the goal of this task, as that was a job for the Navigation Modes task.

This task was done by Mathieu.

3.2.5 Tool & OS integration

In order to reduce the amount of clutter in the interface and avoid extremely long paths, in some cases, the path might be "relativized". The only currently implemented instance of this is when the user is in a folder that is a subfolder of their home directory. For example: On Windows, if my user name is "MyName", instead of "C:/Users/MyName/MyFolder", SCFE displays "~/MyFolder", which is more helpful (and uses a well-known standard of replacing the home folder of the user by `~`). The folder's name is deduced from what the OS publicizes as the home directory, so it will work for any username under operating systems which are correctly supported by `.NET Core`.

Because a file explorer that cannot open files is quite useless, SCFE provides a lightweight integration with the underlying system through being able to open files. This uses the default action from the OS and opens the same app that would open if the user double clicks on the file in the system's file explorer.

This was done by Rakhmatullo.

3.2.6 Website

At this point, while we had some ideas for the website, Mathieu was in charge of the creation and maintenance of the website. The website is available at the following address:

https://salamanders.dev/

At this point, it only had basic information on the project: what it is, who the members of the team are, and some placeholders where needed. It is fully

responsive and works on mobile. At this point, it was ready to welcome more content.

Its design is close to the one we have used throughout the documents and the presentations: black and yellow accents with logos for both SCFE and Viu. Viu did not originally have a logo, but since it was separated in the code base from SCFE, we figured that it could be considered to be another product from our team. The Viu logo was created for this occasion.

The website was done in raw (but clean) HTML and CSS using the Bootstrap toolkit and is hosted on OVH. The HTTPS certificate was obtained through OVH as well. This gives us total control over the website's content, although the creation and upload can be a little tedious at times.

The documentation is also hosted on the website and well integrated with the current theme. A Downloads page was also present but did not actually provide the downloads at the time: these were added for the last defense.

The work on the website as well as the theming for the documentation were entirely done by Mathieu.

3.2.7 Documentation

Basic usage of the application were documented and included information on regular use of the app in the most basic workflows.

The documentation clearly explains the use of the `SEA` and `NAV` modes, ensuring to explain when and where to do something in a clear manner so as not to completely lose newcomers in useless details. It also provides a helpful load of key bindings, which are also available from within the application.

The application itself also has a healthy amount of information: the user

can open special “option panels” which displays all of the options available for a file, along with the key bindings that would allow one to perform the action. This help is dynamically created, so that if we want to have customizable key bindings in the future, the shortcuts shown in the application reflect the chosen key bindings – but as handy as it is, the help remains minimal, both because it should not interrupt the workflow and the console is a poor environment when it comes to reading text, mostly due to the limited amount of space and lack of proper font decorations.

The documentation that is outside of the application is written in Markdown and is available on the website under the Documentation tab. It is, of course, converted into HTML before being published, and some manual adjustments to the HTML code had to be made. Online documentation was done by Rakhmatullo, while in-app documentation and help was done by Matthieu. The theming of the documentation on the website was done by Mathieu.

3.2.8 Overall state

At this point, the application truly existed with almost all of its features. Thanks to our solid basis, we were able to accumulate all of the various simple features we wanted at a fairly fast pace. Three screenshots are available in figures 4, 5 and 6, showing respectively the application working on Windows, on Windows while in the search mode and on Mac OS. Cross-platform compatibility, being one of our priorities, worked, but had a few bugs here and there which were fixed in the period that followed.

Moreover, the websites and documentation had started to actually take shape and came to existence. They were light, as they were not our priority for this period, but still worked fine and did their job.

3.3 Between the second and final defenses

A lot of work had been done between the first and final defense: so much so that there was not much left to do for the final defense, apart from the most advanced features, and the usual bug fixing.

3.3.1 Core UI

Most of the work done for this task in this period was refactoring and bug fixing of what was happening behind the scenes. A few additional internal functionalities were added, mostly to help with multithreading and properly going back and forth between the graphics process and other processes which took care of file management. An example of such functionalities is an all-in-one "RequestSync" function, which is called in file management processes and is responsible for blocking the file process, asking the user for an input, waiting for said input, and passing back said input and freeing the previously blocked process, unless the user canceled the action (e.g. by pressing escape).

Moreover, some important bugs were fixed for Viu: due to bugs in `.NET Core 2.2`, which is the framework we use, we were forced to clear and reprint the entire user interface every single time the user provided input through the keyboard. We therefore made Viu compatible with both `.NET Core 3` *and* the 2.2 version. We could not just drop support for the 2.2 version, as the version 3 is only available as a pre-release and should be released towards the end of 2019.

Another feature that was added was the ability to change the order of columns through a configuration file. Said configuration file, located at `~/.SCFE/columns.txt`, takes the form of the name of the columns separated by commas. The default layout is `name,git,size,date`. While it is a simple

configuration file format, we understand that it might be a little tricky to do for beginners – but we decided against creating a full blown configuration screen to our application due to both time constraints and how overkill such a functionality would be for simple reordering of columns. We do not have many columns available in SCFE: the addition of this reordering mechanism is only a commodity and not a standout feature of the program.

Matthieu was responsible for the entirety of this task.

3.3.2 Navigation modes

Navigation modes received some attention in this part, but not much. It was mostly about adding a few additional shortcuts and providing links for the Git integration. For example, a "Select All" and a "Toggle Selection" functions were added.

Additionally, the user is now able to change the sorting method for the files through two shortcuts: `Shift+S` cycles through the different sorting methods available (name, then by file extension, then by size, then by date, then back to name), and `Shift+Q` reverses the order. The order reversal does not just "flip the entire order": it does so more intelligently. For example, in the name sorting (which is the default), folders go first from A to Z, then files go from A to Z. In the reversed name sorting, folders are still first but go from Z to A and only then do files go from Z to A. This way, folders still remain at the top, which would not be the case if we just reversed the original order.

François and Matthieu cooperated on this task. Matthieu linked the actual shortcuts to the interface while François made the modifications required to the sorting system we already had in place. These modifications are detailed in the next section.

3.3.3 File I/O

The file I/O task saw a new addition which we did not actually plan on the Book of Specifications because we were not sure of its feasibility. One of the major flaws of our application was the lack of a "hot reload" feature: if the user (or another program) created, removed or modified a file, it was impossible for SCFE to detect it and refresh itself. The user had to manually press the `Shift+R` shortcut, which triggers a reload of the folder.

We added a new functionality in this period: auto-refresh. Thanks to a feature of `.NET` called file-system watchers, we are able to catch any change in the folder that is currently opened in SCFE. If a change is detected, we automatically reload the user interface to show the modification, still focusing the file that was previously focused. This process is very fast and makes SCFE extremely useful in many more situation. It also saves two key presses from the user, which is always a nice addition since the application focuses on productivity.

The File I/O got some modifications here and there to support the addition of new sorting options which the user can cycle through, as described in the Navigation modes part of this period.

The original sorting method was also changed to support natural sorting. The original, classic way of sorting names is through a purely alphabetical point of view. In this way, the names "Hello1", "Hello2" and "Hello10" would be sorted as "Hello1", "Hello10" and "Hello2", which is not logical. The logical way of sorting the strings is called "natural sort" and replaces the original sort method we had in place. This way, the sorting is less confusing and more enjoyable for the user, as we believe it is much cleaner.

This task was done by François.

3.3.4 User input

Not much had to be done in this part. The coding of the `COM` mode, which we will detail later on in this document, is based on the exact same mechanism that allowed us to have users provide details for operations like renaming files. We re-used this system to allow the user to enter a command, and then route that command accordingly to what was coded in the Tool & OS Integration part.

An additional ability was added to the text field component, which we use in SCFE as the text box that is available at all times at the bottom of the screen. It is now able to hide its input in case of the user entering sensitive information: this was added in order to allow the integration with Git to request a password from the user in such a way that the password is not displayed in clear on the screen.

Some bugs we encountered with key presses not being registered were fixed by simply upgrading our `.NET Core` version to the pre-releases of version 3.

This task was done by Mathieu.

3.3.5 Tool & OS Integration

At this stage in the project, tool and OS integration were *the* priority. We had a fairly basic file explorer, now we needed powerful features for advanced users.

However, because the goal of our application is to be a file explorer, we decided against having heavy integration with every tool. The idea is that some lightweight integration for the most used tool would be more than enough, and users who need to perform more complex actions should use the tool directly instead of using the tool through SCFE.

With this mindset, we set out to add the two features which we would find most useful: Git integration and a COM mode for executing commands in the command line.

3.3.5.1 Git integration

Git is a "version control system": a tool that allows users to manage their files (usually code) in repositories, version them (i.e. have saved file states), navigate through the file history easily and reliably, while also being able to share the files and their modifications history online on sites like GitHub and GitLab. Git is used at EPITA for submitting code during programming practical work, and we have been using it since the very beginning of the project to collaborate on the code. It is the most widely used tool in its category, so integrating it into SCFE would be an obvious benefit for developers.

Git integration in SCFE is simple yet extremely useful. It is based on the LibGit2Sharp library, and comes with both the ability to visualize the repository and actions on it. When in a Git repository, SCFE automatically loads and caches the repository's status, and displays the status of each file in a separate column. This column only appears when in a Git repository and is automatically hidden everywhere else.

Additionally, a few common actions were added: staging and unstaging a file (which tells Git that they should be added to the next save), creating a commit (i.e. creating a save), as well as pushing and pulling (i.e. sending and receiving saves from a server). These are supported only on a fairly basic level, and we went the safe route. These operations have different variants that require extra input from the user, and Git can usually detect conflicts between files: should any conflict or extra input happen, we simply inform the user that an error happen, and we stop there. This follows the idea that SCFE users should not be in SCFE to perform any complicated action.

An example of Git integration can be found in figure 7.

There is still another way of launching more complex commands from SCFE, in the form of the COM mode.

3.3.5.2 COM mode

The COM mode is the third and probably the most straightforward one: you simply press Ctrl+Enter (or Shift+M), type in the name of the command you wish to launch just like in a regular terminal, and you then simply press enter to launch the command. Like any other text input in SCFE, you can just press Escape to cancel the action.

The command itself is rerouted to an underlying shell, which is PowerShell on Windows and bash on Mac OS and Linux. If the process writes something in its output (e.g. to show a message), the message is intercepted and routed to the interface where all of the messages are usually shown in SCFE.

This system works well and is done in a multithreaded fashion, meaning that the main interface is not blocked while the command is ran in the background. It is still fairly basic and does not support processes which require input from the user, but it is not intended to be a replacement for a full terminal. It can be useful for small tasks though. For example, we are able to build SCFE inside SCFE through a single command in COM mode.

An example can be found in figure 8.

Work on both git integration and COM mode was made by Rakhmatullo with heavy assistance from Matthieu. The rest of the team also helped testing the new additions, which was crucial since some features like the `COM` mode heavily rely on platform-dependent behaviors.

3.3.6 Website

Even though the website got some good progress between the first and second defense, there was still some work to do. Some animations were added and more theming was done overall for a cleaner look.

The download page also got a few new additions, including a landing page inspired by the front page of the Atom editor (https://atom.io), as well as more details on the software like release notes and more.

The work on the website was entirely done by Mathieu.

3.3.7 Documentation

Additional parts of the project were documented. Like in the previous period, every new feature is displayed in a specific panel that shows all of the available actions.

The documentation that is on the website also got a few improvements, mostly documenting new features that appeared here, as well as documenting more of the features that had not been documented yet although they were present.

Documentation was done by Rakhmatulo.

3.3.8 Overall state

At this point, the application, website and documentation were fully finished. The final results are explained in the following section.

The application also received an installer for Windows and for Mac OS, a light version (that does not include the `.NET Core` framework and thus requires

a separate download) for Windows, and the binaries for Linux. The installer for Windows was made by Matthieu, the one for Mac OS by Mathieu.

4 Final results

In this section, we will go over what was created for our project and the end results.

4.1 A library: Viu

As we mentioned before, we did not like any of the various options we had for a us.

Although not the main goal for our application, we did end up creating a separate library for creating interfaces while in the console. The motivation behind not simply re-using something that already existed was that the other libraries did not give us the flexibility we wanted, had problems on non-Windows platforms, or had miscellaneous quirks that we could not fix easily.

Viu provides a great way to create an application in the console. It is fully resizable, multi-threaded and has an abstraction layer which allows it to run on any sort of output. By simply replacing this abstraction layer with something else, we could theoretically write the user interface to a printer, an actual graphical interface (although it would still keep its text look), a different console implementation, or pretty much anything else.

Viu features a powerful input system which we have described in the evolution of the project. It has greatly helped us in the process of creating the application itself, providing an easy way to swap between bindings.

4.2 An application: SCFE

The SCFE application has grown to be exactly what we had predicted, and more.

One of the features we were able to add although it was not initially planned is an auto-refresh system. When adding or modifying files from inside SCFE, the application originally refreshed itself, ensuring that files that were created were shown. However, this system did not take into accounts files which might have been modified from elsewhere, e.g. from another application or command line utility.

In an attempt to add some extra functionalities, Matthieu looked into the possibility of using the file system watcher feature of `.NET`, which was surprisingly easy to add since we had already implemented multi-threaded related features. In the end, SCFE supports reloading folders on the fly when they are modified from somewhere else.

As for all of the features that were planned, they are all there, including the mode system. SCFE was fully built with this mode system in mind. There are three modes usable, following a "modal" approach:

- `NAV` (Navigation) mode, for easily navigating between files. Shortcuts are easy to reach in this mode, only taking a single keypress. The `NAV` mode is built for productivity for folders with a limited amount of files, and for people who prefer the regular, slower approach of file navigation, where you can only go up and down between files.
- `SEA` (Search) mode, for quickly finding a specific file you are looking for. The shortcuts in this mode are the same as with the navigation mode, the difference being that the search mode requires the Ctrl key to be held down for all shortcuts. This is because pressing any key without the Ctrl

key will search for files instead of triggering a shortcut. The user can type the name of a shortcut from anywhere and easily and quickly access said file. It makes looking for files extremely efficient, and resembles more what console users would expect.

- `COM` (Command) mode, for launching more complex commands that one would normally enter in a terminal. Commands are great for launching scripts or other processes which are more technical or flexible than what can usually be done in SCFE.

A few different workflows exist with this model:

- Using only the `NAV` mode with the Options panel we implemented. This panel provides access to all of the available actions in a list, and displays all of their associated shortcuts. It is a great way to get started with SCFE and allows beginners and those who do not want to learn everything there is to know about the app to use the application to its full extent.
- Using `NAV` mode with all of the shortcuts and sometimes switching to `SEA` mode to navigate in big folders. The shortcuts in `NAV` mode are particularly fast to use since they do not require the Ctrl key to be pressed. Copy pasting, which is usually a Ctrl+C and Ctrl+V simply because a key press on C for copying and a key press on V for pasting. Of course, should the user forget a shortcut, he can open the options panel to get a reminder. This is the workflow most of the people of the team use.
- Only using the `SEA` mode. This makes navigation through folders extremely fast, but might be more taxing on the hand for launching shortcuts. Moreover, you have to know the names of your folders by heart in order to really be efficient. This workflow is the closest to the console way of navigating through files.

The `COM` mode, not truly being a "navigation" mode but more of an "operation" mode, can still be used in conjunction with all of the other modes without any issue and in any workflow.

SCFE provides all of the regular operations one would expect from a file explorer, and more:

- Direct lightweight integration with commands through the `COM` mode
- Lightweight integration with Git with reduced lag
- Compatible with all platforms with almost no differences in operation
- Auto-refresh on folder modification in other applications

Other file explorers rarely feature all four of these features at the same time, which makes us very proud of the resulting product.

Another interesting note about SCFE is that, even though it is written in a framework that is not necessarily known for its speed, it remains fairly fast and perfectly usable throughout the life of the application, which is especially important when going through Git repositories. Using Git repositories is a bit taxing for the system, since the way the Git system works forces us to reload the entire index of the repository in order to get the latest status of the different files and showing an accurate representation of file status.

4.3 A website: salamanders.dev

Along with SCFE, the website we created, while simple, still respects what we consider to be our priorities: something clean that goes straight to the point but does not lose you in oversimplifications.

The various download links and the documentation are posted to the website, which features a clean, minimalistic and modern style that still features our

graphical theme and font of choice (PT Mono). The downloads are built for all of the platforms we are targetting, and it includes a Windows installer that Matthieu built using the Nullsoft Scriptable Install System (NSIS) tool. NSIS is a tried-and-tested solution that has been around for 19 years and has a great userbase: almost all of the installers that we have nowadays are built using NSIS. As a final perk, NSIS is completely free, which prevents us from adding costs to our project.

The website itself, being hosted through OVH, is fully static, which is an excellent option as it requires no maintenance, stays flexible, and we do not have to worry too much about databases or other more complex web components. Being written in pure HTML, it is a bit heavy to manage, but Mathieu has had experience with writing pure HTML websites and knows how to properly manage them.

5 Overall opinion

5.1 The good

Overall, we are very happy with what we have managed to produce.

All of what we did was new for members of the group: for example, managing a team was a first for Matthieu, while Mathieu and François had very little experience with coding prior to joining EPITA. This gave us new experiences, which is always a good thing.

The project also allowed us to use tools in a more advanced way: while Git was only used for simple submissions in our programming course, we used it to have it integrated inside a different application – and also used it for ourselves to track the project's progress.

5.2 The bad

One particular annoyance throughout this project was one of our most important goals: cross-platform compatibility. While everything worked fine on Windows machines, the other members of the team had a much tougher time ironing out bugs. Terminals on non-Windows platforms tend to "catch" a lot of key presses, rendering some key sequences completely useless. All of the key sequences that involve pressing the Alt key simply did not work most of the time on non-Windows platforms. There are no workarounds for this other than diving to a much more low-level approach to communicating with the terminal for catching these key strokes, but it would have been way too time consuming for us.

In the end, the various panels we have in place for selecting actions from

a list end up also being a last resort if a key binding fails for some reason on non-Windows platform. This is a very unpredictable issue since all systems will catch different key strokes and, when the underlying system is not responsible for the issue, it is the `.NET` framework we are using that does not want to cooperate with us.

On a more human level, Matthieu struggled a lot with delegating tasks to the rest of the team, since he had more experience with this kind of project. Coding features was usually faster if he did it by himself rather than letting others do it and having to explain them.

5.3 Final thoughts

This project was a first for all of us, and we are all very proud of the end result. We believe that we achieved what we had in mind when we first started, and that we successfully delivered a product that we want to use ourselves. This has been an interesting experience which allowed us to manipulate software development tools outside of the usual programming courses that we have, and a rather new team experience.

6 Appendix

This section contains additional images and screenshots of our application.

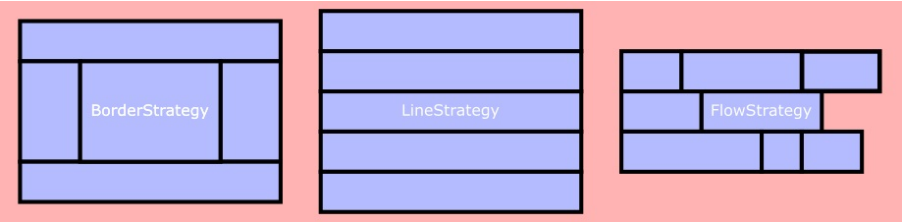

Figure 1: Layout strategies for components (Border, Line and Flow strategies)

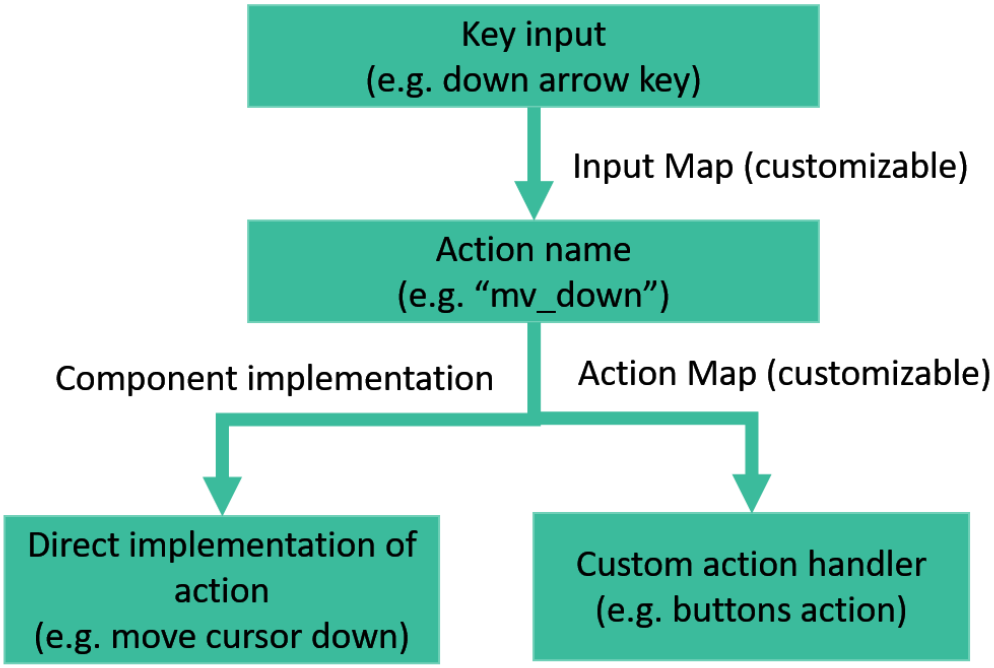

Figure 2: Illustration of how `InputMap` and `ActionMap` work

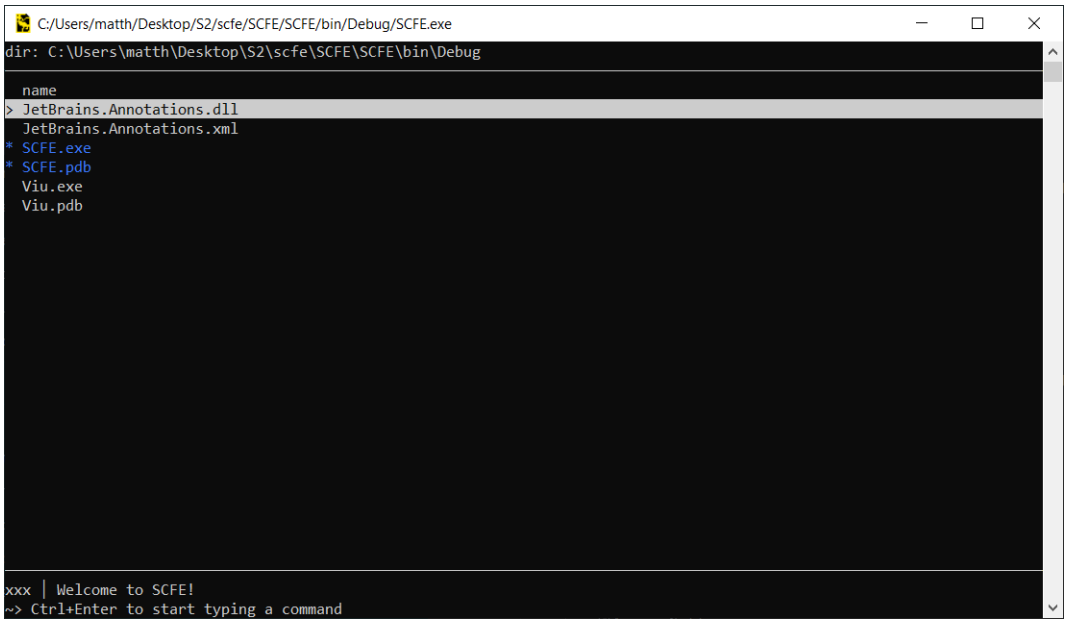

Figure 3: Prototype of the application (First defense)

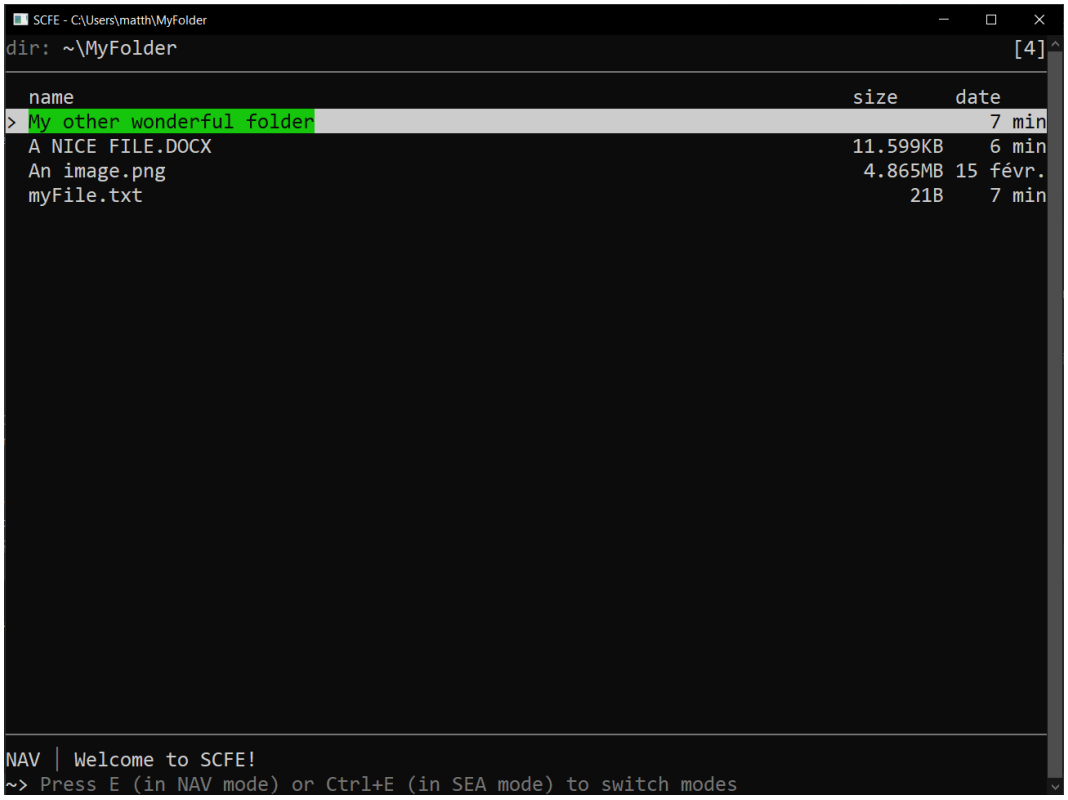

Figure 4: The application on Windows (Second defense)

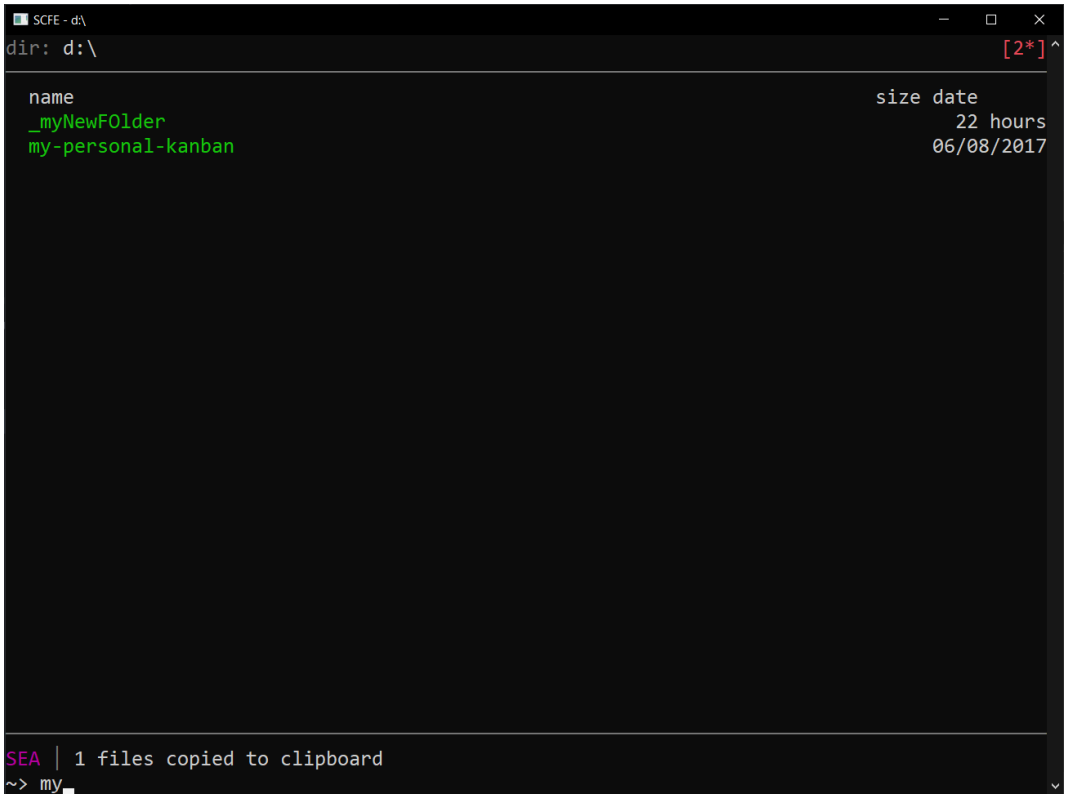

Figure 5: Using the SEA (search) mode

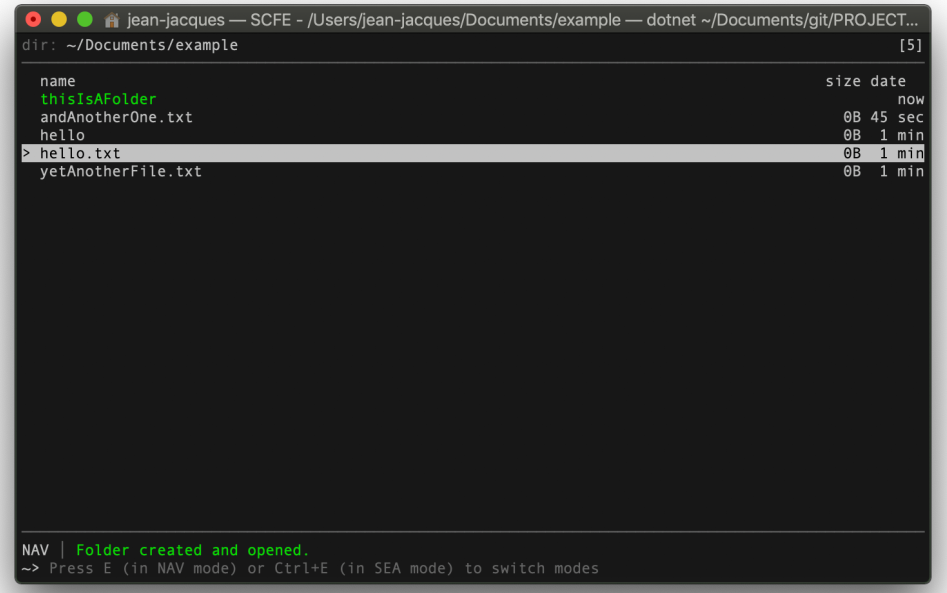

Figure 6: The application running on Mac OS (Second defense)

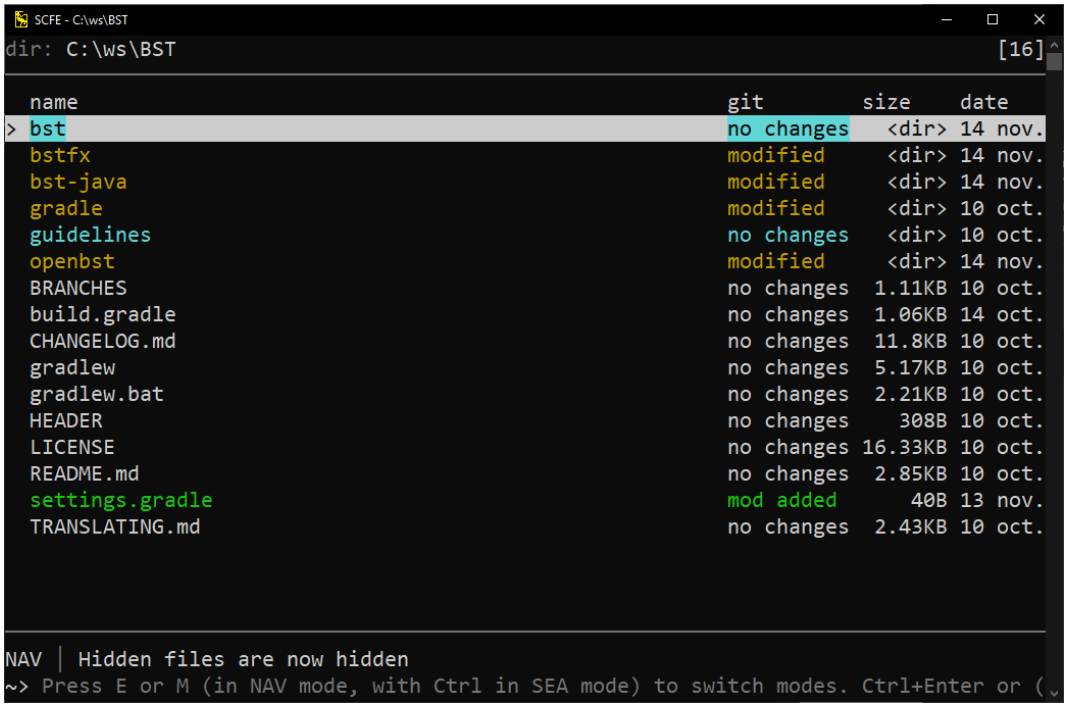

Figure 7: The final application in a Git repository

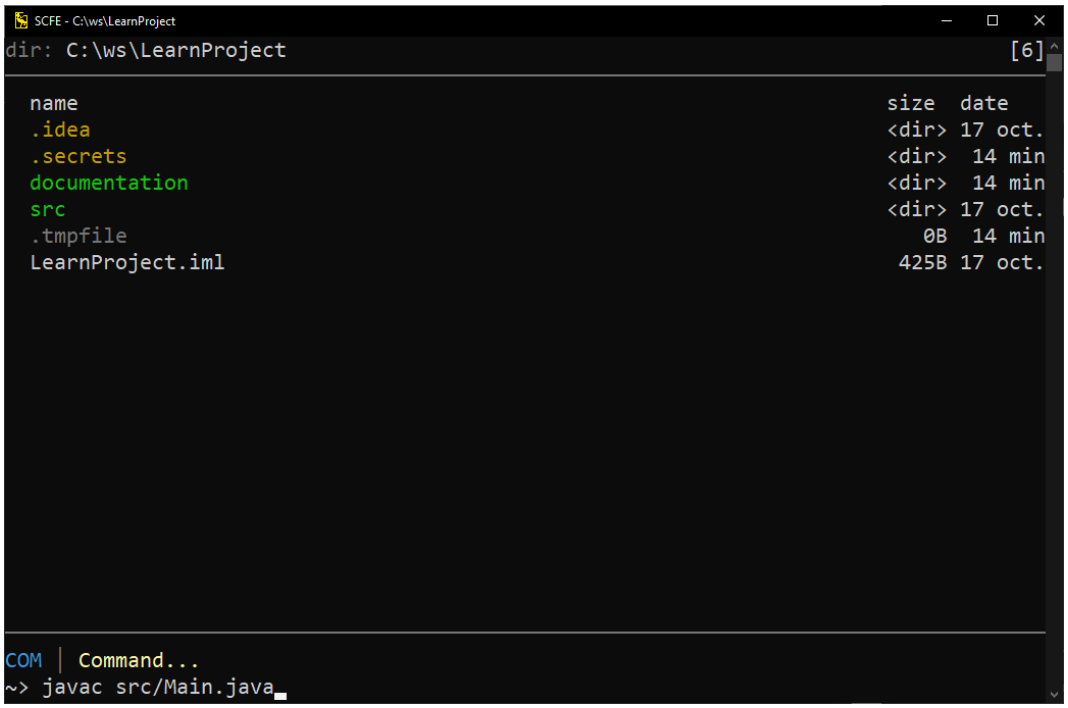

Figure 8: The final application in COM mode, in a folder with hidden files