CLI Usability

Tyler Cecil, Louis Jencka, Jesse Maes

May 4, 2016

1 Background

In the lecture from the first day of Human Computer Interaction, we talked about the history of computer user interfaces. The command line interface (CLI) was mentioned in a slide, as an example of a historical type of interface. It was described as “efficient, precise, and fast.”

Unfortunately, this advanced interface has gained its power at the expense of ease-of-use. Utilizing the CLI entails a large overhead to learning set of commands, which is what has led to peculiar design goals for command line interfaces. Commands and options are generally terse abbreviations, which are very easy to memorize, but are not necessarily easy to learn.

As Computer Scientists, the CLI isnt just relic of history; we make heavy use it on a daily basis. Its a powerful and highly flexible tool, useful for a menagerie of tasks. For the beginner user, the shell can be an incomprehensible nightmare – its blinking cursor against a blackened sky a stand-in for man’s helpless struggle against death. Just as text editors have made leaps and bounds in communicating essential contextual information, we believe the CLI can be improved for beginning and advanced users alike.

2 Problem Definition

Shown in Figure (1) is the problem sketch for our project. The frames are color-coded according to the model-view-controller (MVC) architecture design pattern: green for the model; blue for the view; red for the controller.

The sketch begins with a user, of any experience level, who interacts with the system via keyboard and mouse. The main user interface of the system is an ordinary command line window, which will read and execute commands. In addition, our program will have an adjacent contextual information window, which provides heads up information about the state of the system, and can preview the results of commands before they are run.

A user’s input in passed from the interface to an interaction engine. The interaction engine

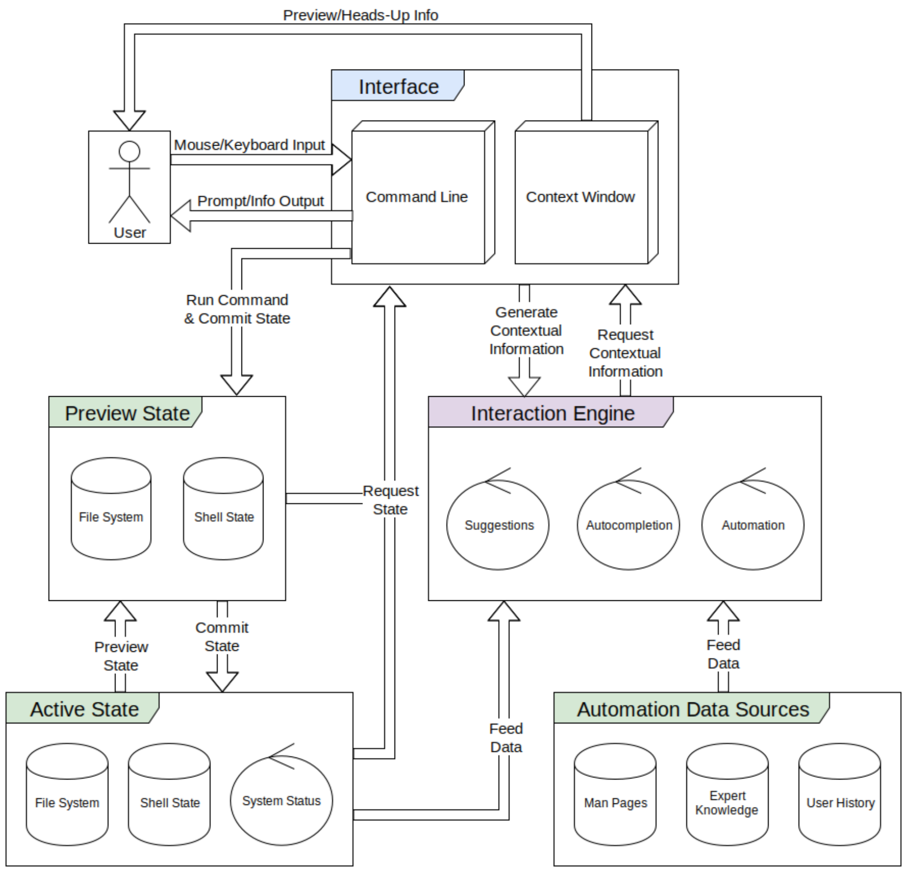


Figure 1: A sketch of our problem design.

processes this text for command suggestions, autocompletion, and automation, returning this information to the command line interface. The OS's man pages, the package's built-in patterns, the user's history, and the current state of the shell are all used by the interaction engine to intelligently provide these features.

This system uses a two-step design to model the changes that the command line applies to the operating system. This allows us to provide an "escape hatch" before running commands, which is important because it's possible for users to make destructive mistakes when using the command line! Every time a command is run, the changes to the filesystem or OS state are first staged in the "preview state." The staged changes are visible to the user through the context window, where they can be approved or rejected. If the user approves of the action, then the preview state is flushed and the changes are sent to the "active state," which is synonymous with the actual underlying operating system.

3 Potential Users

3.1 Beginners

Our first imagined user is brand new to the shell, if not Linux itself. A freshman CSE student, for example, would need to learn how to navigate a UNIX operating system via its main interface - the shell. As a first-time Linux/BSD user they would need to learn some basic information about how their system is organized, and how to manipulate it. From their previous computer usage they could be expected to understand the hierarchical nature of file systems, but nothing more. What this user needs is to reach a baseline fluency in CLI usage: how to navigate their file system; how to create and modify files; how to learn more.

1. No experience with UNIX operating systems
2. No experience with a CLI
3. No experience with the state inherent to a shell, mainly the current working directory
4. No knowledge of what programming is, or how to automate tasks
5. No knowledge of the anatomy of a command (command name, arguments, options, et cetera)
6. No knowledge of the software available on their system
7. No knowledge of basic OS tasks, such as installing software
8. Has experience with common GUI techniques - click-to-open, drag-and-drop, et cetera
9. Is uncertain

3.2 Casual Users

Another potential user would be for scientists, engineers, or anyone with experience as a technical computer user whose workflow could be more efficient if they knew how to use the shell. The CLI can make it easier for users to automate tasks and make precise changes to the files they are working on. These users may already know how to complicated tasks on their computers, or even be able to write programs. Many powerful tools insulate their users from the details they would need to use the shell, such as Matlab, Maple, or Excel. An information-rich user interface could help casual power users make the transition to understand how to use their terminals for more complicated tasks such as shell scripting and task automation.

1. Understands the basics of how computers work and filesystems are organized
2. Understands what computer programs are or has knowledge of how to program

3. Is only familiar with a GUI, or is more comfortable with pointing and clicking than typing
4. Regularly needs to perform some repetitive task on their computer, but does not imagine the task could be automated
5. Finds themselves lost if/when they use the CLI - doesnt understand the larger context of the system surrounding the shell
6. Is unaware of most of the programs, commands, and command line options they could be using (has only seen the tip of the iceberg)
7. Does not know how to apply technical knowledge or programming skills to system tasks in the CLI

3.3 Advanced User

We also believe that advanced users who already know their way around the command line could benefit from an information-rich shell. This user can be expected to know by heart the commands they need to use, the details of how their system is organized, and how to program in a shell language. Where the CLI could stand improvement for these users is in providing more details on the state of their system, and allowing a terser search of console-oriented documentation.

1. Deep understanding of OS structure and interfaces
2. Mastery of command line usage
3. Commonly constructs aliases and automates tasks
4. Consults system documentation to find advanced command options
5. Understands that task inference is frequently achievable from context in the shell

4 Tasks

4.1 User Stories

As a beginner,

- I want to compile my source code.
- I want to navigate to my source code.
- I want to be able to edit my source.
- I want easy access to information about what I am doing.

- I want to install the tools I need to make code.
- I want to know what I can do.
- I want to be able to open everything like I was able in windows.
- I want to know what my command is going to do.
- I want the same visual feedback a file browser gives me.
- I want to know how to utilize my shells history.
- I want to be able to easily and obviously perform common tasks.
- when I type a filename, I want the most common task associate with that file to be suggested.
- when I drag a file into the command line, I want that file name to be typed.

As a moderate user,

- I want to automate my workflow.
- I want the system to alert me when automation is an option.

As an expert user,

- I want tab-completion that is file-type aware
- I want heads up information about my system as I use it

As any user,

- I want to be able to copy and paste.
- I want to be able to drag a link into my shell and have the file be added to the current directory.
- I want the system to learn about my most common tasks.
- I want to know what options are available to me for a given command.
- I want contextual information displayed to me when composing commands.
- I want to view files in the current directory as I move through the system.
- I want to be able to undo what I've done.

4.2 Hierarchical Analysis of Selected Tasks

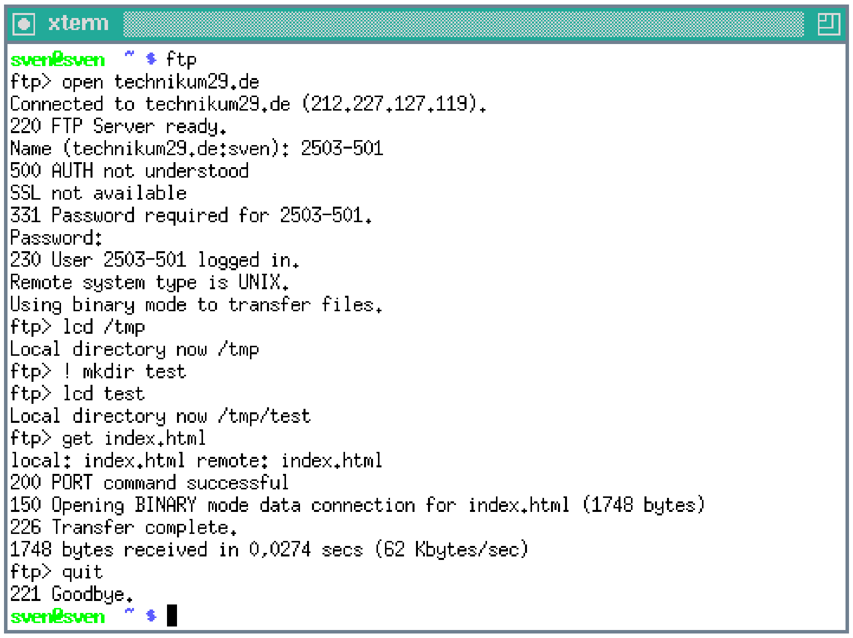
1. As any user, I want to be able to undo what I've done.
   (a) Run a command
      i. Know what command to run
      ii. Write the command
      i. Understand the results of the command
         A. View what files/directories/settings were changed
         B. View how they were changed
   (b) Undo a command
2. As a beginner, I want to know what I can do with a file.
   (a) Specify a file
      i. See what files are in my current working directory
      ii. Select one of those files
   (b) View the available commands
      i. View a list of what commands would be useful for that file
      ii. View what each command does
   (c) Run a command on the file
      i. Select a command
3. As a casual user, I want to automate my workflow.
   (a) Specify a task which needs to be run many times
   (b) View suggestions on how to run that task
   (c) Select an automation technique
4. As a beginner, I want to know what the command is going to do.
   (a) Specify a command
   (b) View that commands effects
      i. View which files are going to be affected
      ii. View how those files are going to be affected
   (c) Chose whether to run the command or not
5. As an expert user, I want heads-up information about my system as I use it.
   (a) See what the state of my session is
      i. View background jobs
      ii. View possible commands based on the current context
         A. Take local files into account
      iii. View the ways the current session has been altered – non-normal environment variables, virtual environments, et cetera.

5 Existing Systems

While there exists no application that cover all of the contextual and adaptive features discussed in this paper, the following existing systems tackle many of our ideas for a richer shell.

5.1 xterm

While there exists no application that covers all of the contextual and adaptive features discussed in this paper, the following existing systems tackle many of our ideas for a richer shell.

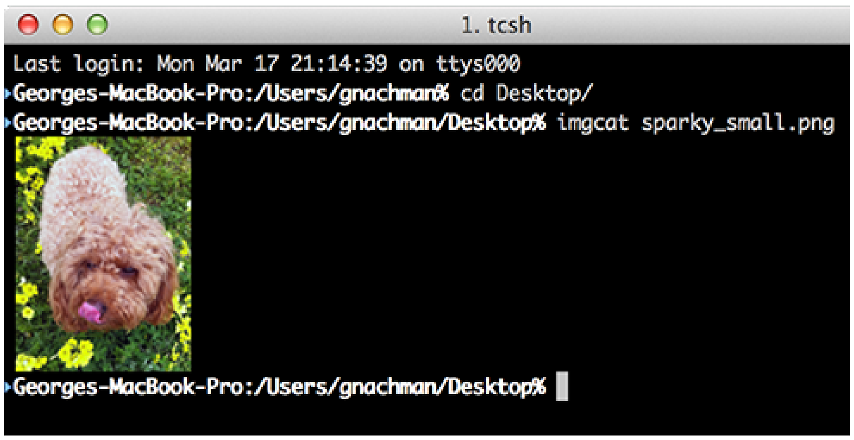


```
sven@sven ~ $ ftp
ftp> open technikum29.de
Connected to technikum29.de (212.227.127.119).
220 FTP Server ready.
Name (technikum29.de:sven): 2503-501
500 AUTH not understood
SSL not available
331 Password required for 2503-501.
Password:
230 User 2503-501 logged in.
Remote system type is UNIX.
Using binary mode to transfer files.
ftp> lcd /tmp
Local directory now /tmp
ftp> ! mkdir test
ftp> lcd test
Local directory now /tmp/test
ftp> get index.html
local: index.html remote: index.html
200 PORT command successful
150 Opening BINARY mode data connection for index.html (1748 bytes)
226 Transfer complete.
1748 bytes received in 0.0274 secs (62 Kbytes/sec)
ftp> quit
221 Goodbye.
sven@sven ~ $
```

Figure 2: An xterm window.

5.2 iterm2

iterm2, like xterm, essentially just provides a terminal emulator, which in turn provides a shell. However, it is able to provide far more contextual information than that of xterm. The shell can alert the terminal emulator itself about the state of the system (such as current git status), as well as embed images into the text result.



```
Last login: Mon Mar 17 21:14:39 on ttys000
Georges-MacBook-Pro:/Users/gnachman% cd Desktop/
Georges-MacBook-Pro:/Users/gnachman/Desktop% imgcat sparky_small.png
Georges-MacBook-Pro:/Users/gnachman/Desktop% 
```

Figure 3: An iTerm 2 window, with inline images.

5.3 zsh / Fish

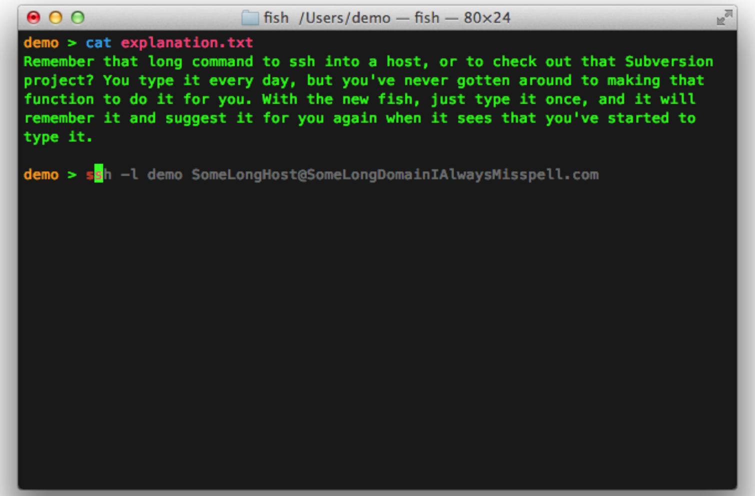
Both zsh and fish are shell programs designed to be more usable alternatives to bash. A large part of this usability revolves around autocompletion and suggestion. Fish will suggest command as you type (though it is not bash-compatible). Zsh allows arbitrary autocompletion to be added to the system.

5.4 Explain Shell

Explain shell is an educational tool which will break up a command, and explain how it worked and what it means. This tool does not itself provide a shell, but it does provide a tool which can be used to learn and understand bash and related programs.

5.5 Jupyter

Project jupyter is a modular, web based shell, designed for use with data analytic languages (such as Julia, Python, and R). It allows graphical response to commands, and lets users distribute entire sessions of use. An important feature is once a command is run, the user can edit the command. This chance will then propagate through the history.



```
demo > cat explanation.txt
Remember that long command to ssh into a host, or to check out that Subversion
project? You type it every day, but you've never gotten around to making that
function to do it for you. With the new fish, just type it once, and it will
remember it and suggest it for you again when it sees that you've started to
type it.

demo > ssh -l demo SomeLongHost@SomeLongDomainIAlwaysMisspell.com
```

Figure 4: A fish shell, showing a suggested input.

	This	`xterm`	iTerm 2	`zsh/fish`	Explain Shell	Jupyter
Undo	V	X	X	X	X	V
File Actions	V	X	X	O	X	O
Automation Assistance	X	X	X	X	X	X
Command Results	V	X	X	X	V	O
Heads-Up Information	V	O	V	V	X	V

Table 1: Comparision of features. V – able to perform the task, X – unable to perform the task, O – able to perform the task with poor interactive design.

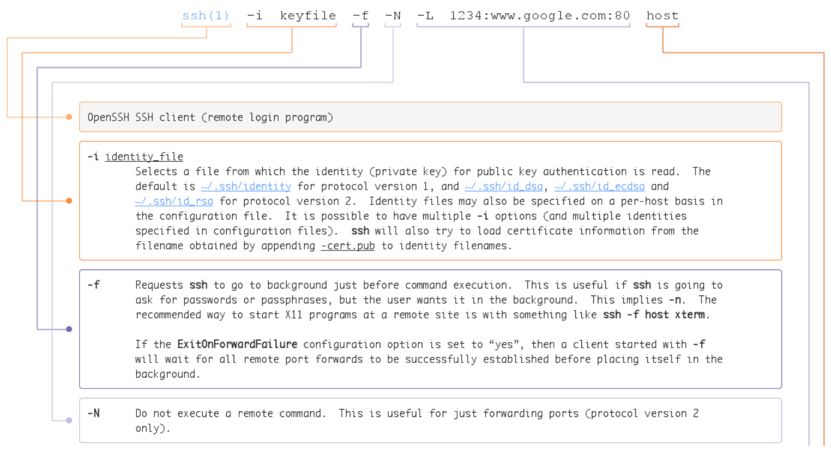
5.6 Comparison of Features

6 Task Navigation (Design Alternatives)

For each of the main tasks provided in Section 4, we analyze the ways in which these tasks could be actualized. Each actualization is represented with three separate diagrams showing the flow of actions.

6.1 Undoing Actions

In this task, the user needs to be able to be able to revert an action they have just performed (something that is particularly useful for beginners to the command line environment).



```
ssh(1)  -i  keyfile  -f  -N  -L  1234:www.google.com:80  host
```

```
OpenSSH SSH client (remote login program)
```

```
-i identity_file
        Selects a file from which the identity (private key) for public key authentication is read.  The
        default is ~/.ssh/identity for protocol version 1, and ~/.ssh/id_dsa, ~/.ssh/id_ecdsa and
        ~/.ssh/id_rsa for protocol version 2.  Identity files may also be specified on a per-host basis in
        the configuration file.  It is possible to have multiple -i options (and multiple identities
        specified in configuration files).  ssh will also try to load certificate information from the
        filename obtained by appending -cert.pub to identity filenames.
```

```
-f      Requests ssh to go to background just before command execution.  This is useful if ssh is going to
        ask for passwords or passphrases, but the user wants it in the background.  This implies -n.  The
        recommended way to start X11 programs at a remote site is with something like ssh -f host xterm.

        If the ExitOnForwardFailure configuration option is set to “yes”, then a client started with -f
        will wait for all remote port forwards to be successfully established before placing itself in the
        background.
```

```
-N      Do not execute a remote command.  This is useful for just forwarding ports (protocol version 2
        only).
```

Figure 5: A screen shot of explain shell breaking down a command.

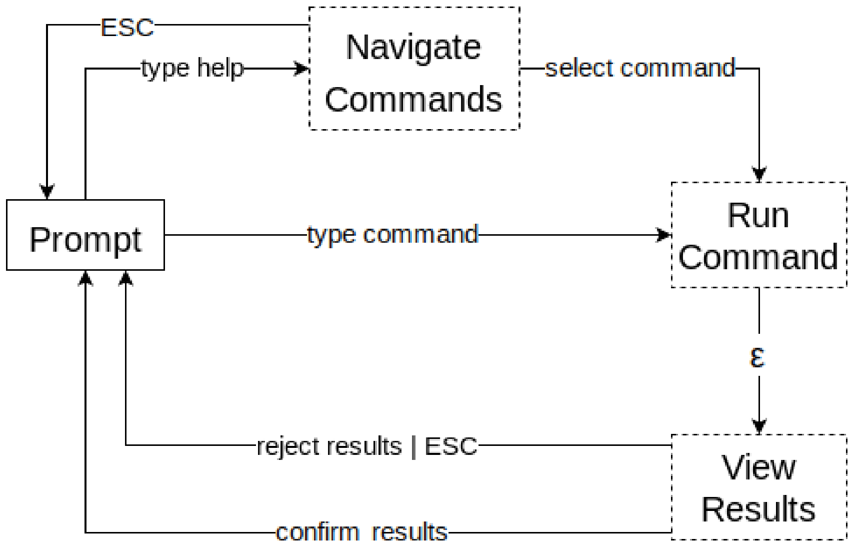
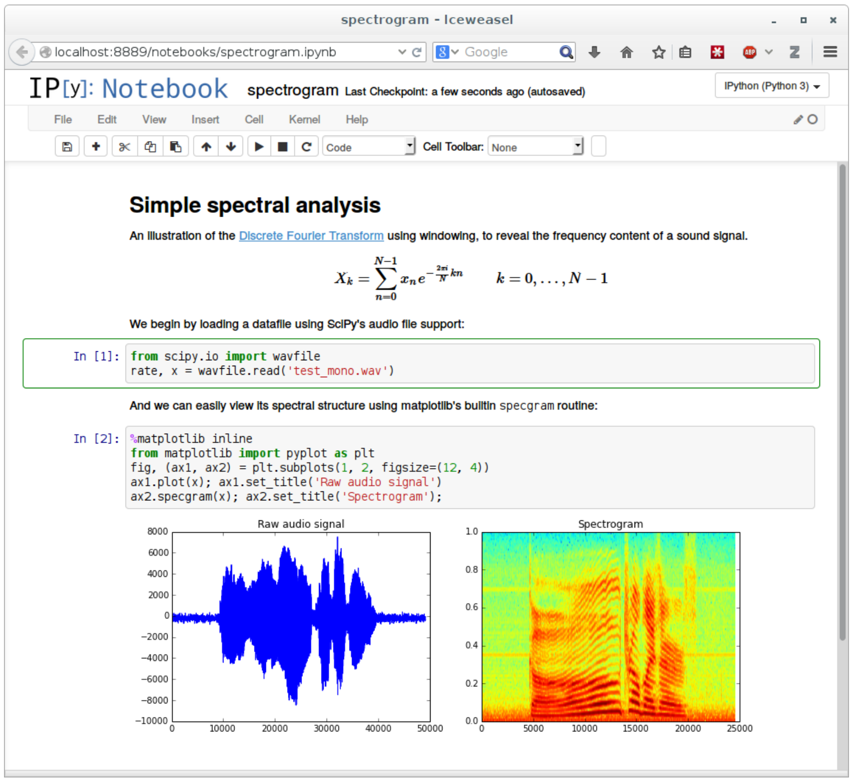


Figure 7: A breakdown of the procedure for undoing actions.

Figure 7 demonstrates a method which allows the user to see the results of their actions, and then chose whether or not to carry it out.



Simple spectral analysis

An illustration of the Discrete Fourier Transform using windowing, to reveal the frequency content of a sound signal.

$$X_k = \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi i}{N} kn} \qquad k = 0, \dots, N-1$$

We begin by loading a datafile using SciPy's audio file support:

```
In [1]: from scipy.io import wavfile
        rate, x = wavfile.read('test_mono.wav')
```

And we can easily view its spectral structure using matplotlib's builtin `specgram` routine:

```
In [2]: %matplotlib inline
        from matplotlib import pyplot as plt
        fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 4))
        ax1.plot(x); ax1.set_title('Raw audio signal')
        ax2.specgram(x); ax2.set_title('Spectrogram');
```

Figure 6: An interactive jupyter session.

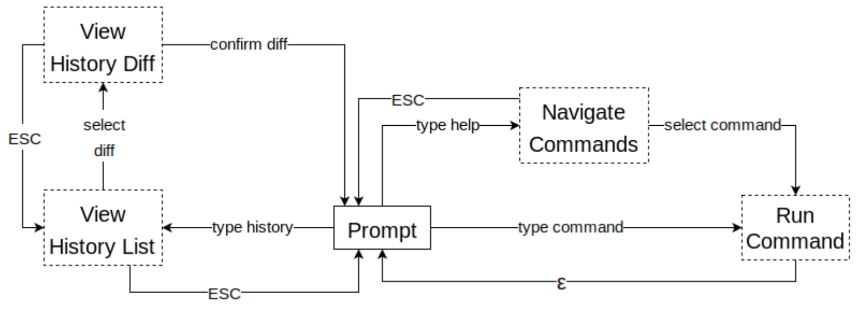


Figure 8: A breakdown of the procedure for undoing actions.

A scheme that gives perhaps more undo ability (yet less pedagogical value) is the ability to brows the history of your commands and the states they represented. Figure 8 shows such a scheme.

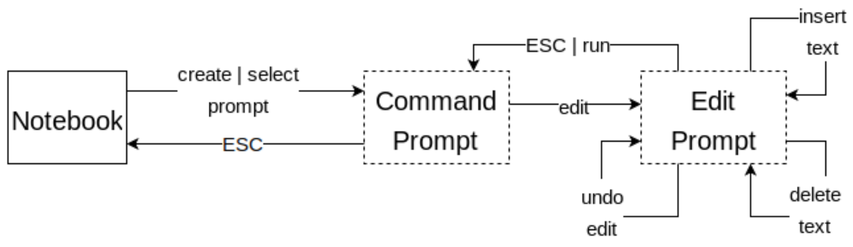


Figure 9: A breakdown of the procedure for undoing actions.

Another common way to deal with this problem is the style that is used by the *iPython* notebook – displaying the whole command history, and allowing the user to change the text of the commands that were previously entered. Figure 9 shows such a scheme.

6.2 Discovery

Users commonly want to know what they can do with given file types. We want to provide the user with the knowledge of how to continue with their files.

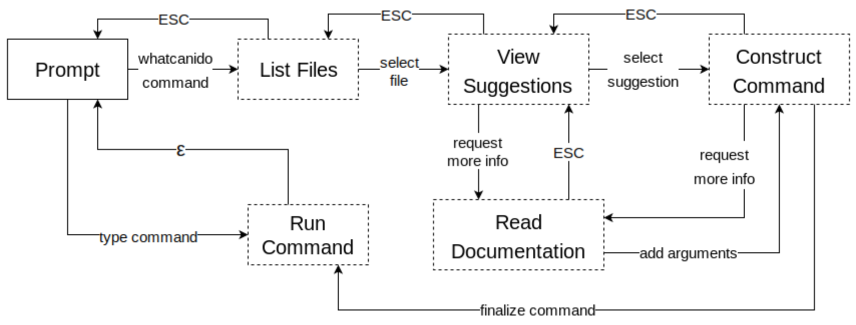


Figure 10: A breakdown of the procedure for undoing actions.

In Figure 10 we present a model where first a file is selected. Once the selection has been made, the shell can provide a list of “what to do next” options. This represents a more reverse polish style of shell navigation.

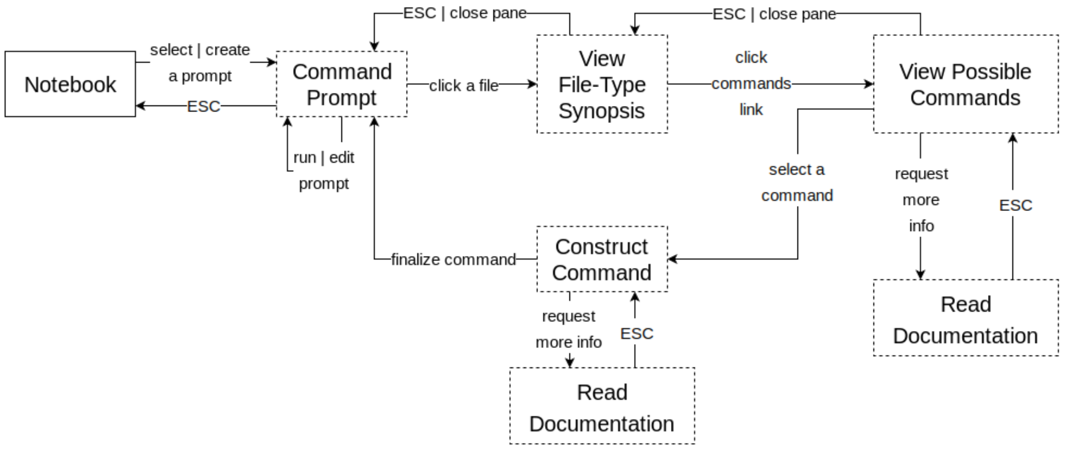


Figure 11: A breakdown of the procedure for undoing actions.

An alternative to this, again, would be a notebook style interface, with a mouse oriented approach. In this, we can select files, and can optionally be presented with a list of actions.

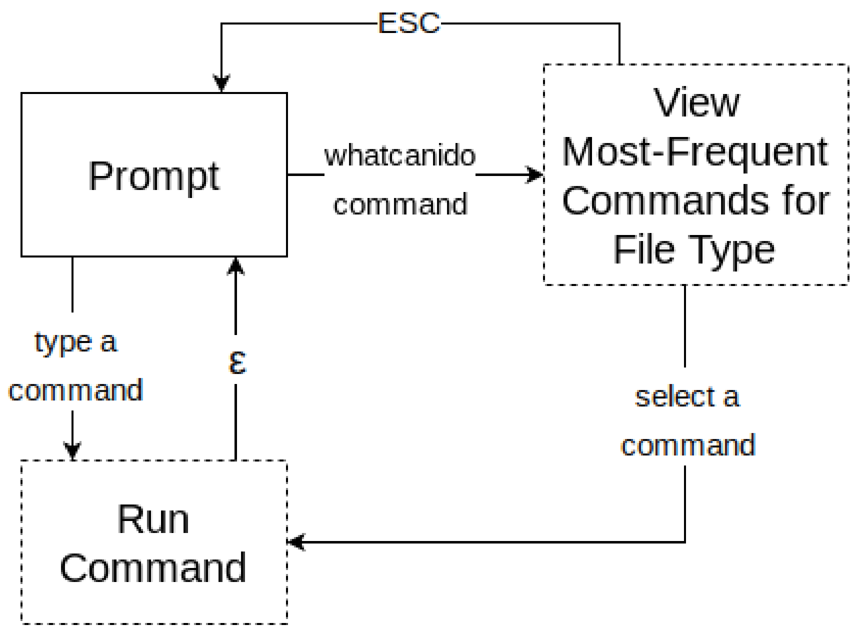


Figure 12: A breakdown of the procedure for undoing actions.

Figure 12 shows the simplest way to implement this feature – a special command to ask "what is the most common task to perform on this file type."

6.3 Workflow Automation

Commonly users are performing repetitive tasks on a related set of files. They may not understand that their workflow could be automated, or how to automate things. The onus would then be on our software to detect common situations as well as repetitive situations, and alert and assist the user in streamlining their work.

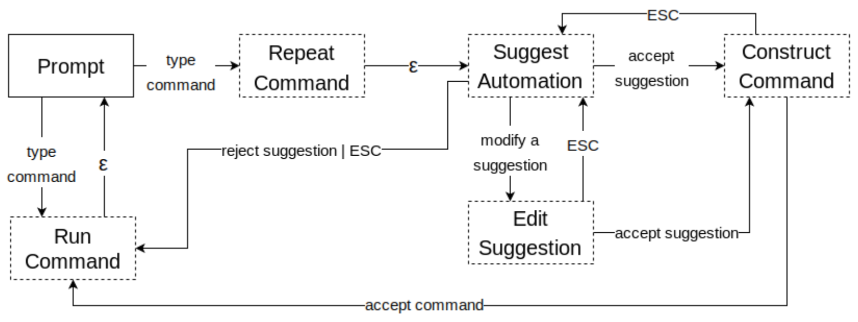


Figure 13: A breakdown of the procedure for undoing actions.

In Figure 13 we show perhaps the most obvious way to perform this sort of automation – detect repeated commands, and suggest an automation path (such as building a for loop). This will serve as a good, non-invasive method, that will alert the user of features of the shell.

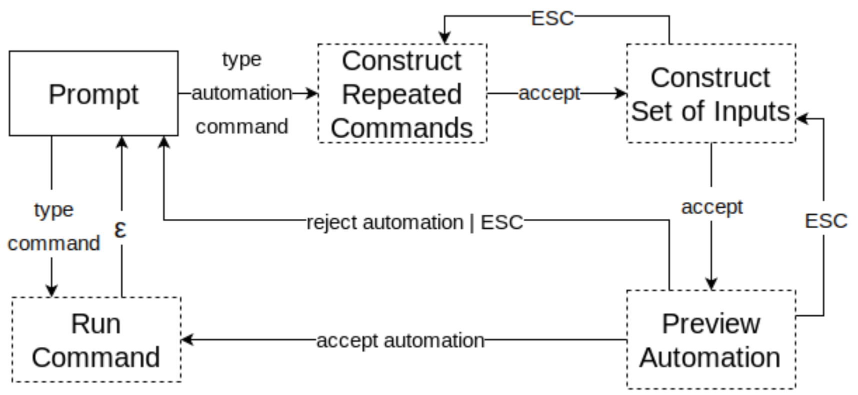


Figure 14: A breakdown of the procedure for undoing actions.

The user, however, could be provided with an “automate” command, prompting this system to provide automation suggestions. This would be an easier path to pursue, but leave more responsibility for the user. Figure 14 shows such a scheme.

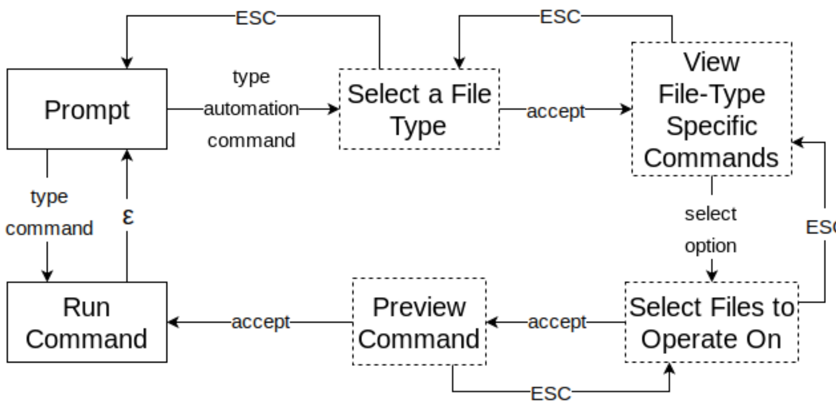


Figure 15: A breakdown of the procedure for undoing actions.

Finally, we could boil all automation commands down to batch processes on related files. Figure 15 shows how the user may select related files, and then build up a command to operate on all files.

6.4 Preview Command

Commonly users are not sure how what the results of their commands will be. We feel being provided with a "preview" of commands may help build confidence in the shell.

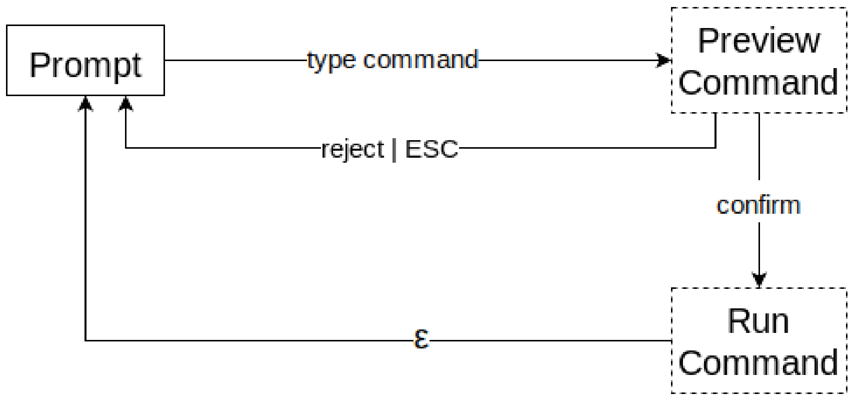


Figure 16: A breakdown of the procedure for undoing actions.

An easy way to achieve this would be to preview the command while it is being constructed, perhaps in the context window we have described. The user can then fluidly work in the shell, seeing results before they occur.

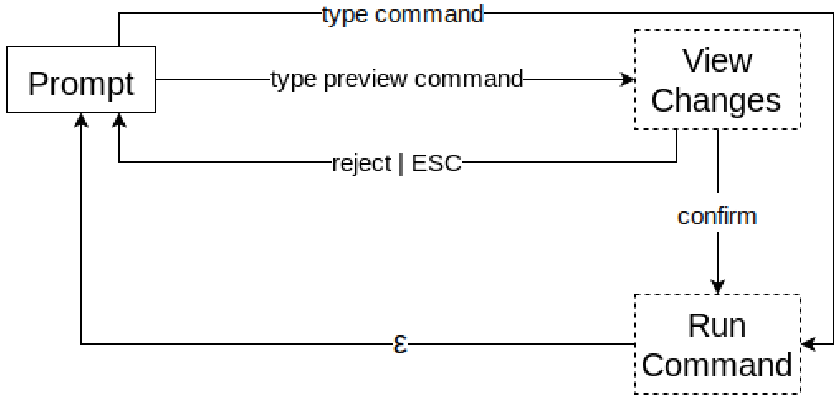


Figure 17: A breakdown of the procedure for undoing actions.

Figure 17 however would remove this as being the default behavior, and allow the users to request a preview. The rest of the workflow would remain unchanged.

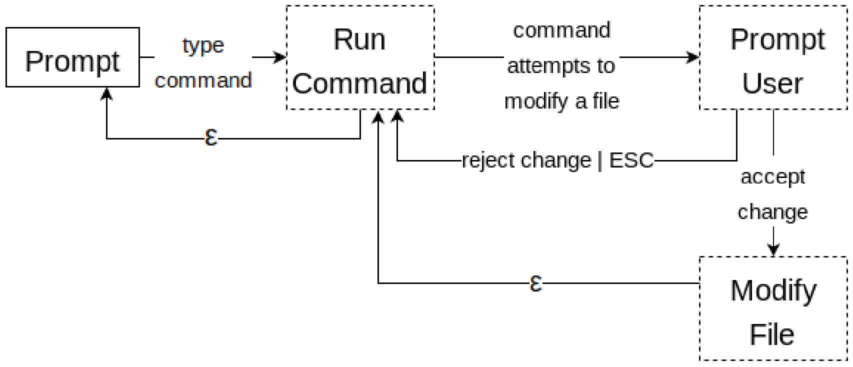


Figure 18: A breakdown of the procedure for undoing actions.

Finally, Figure 18 shows a case when previews are only used when the shell detects a chance will be made to any files (probably the time when users are most uncomfortable). This way the users are not constantly burdened with previews, and only shown then when they really matter.

7 Possible Interaction Techniques

As this is an addition to the CLI, and not a departure from it, we intend to keep the keyboard as the main focus. All new interfaces should be navigable by keyboard, if not least primarily so. For users new to a CLI, however, they can be assumed to be most comfortable with interacting via a mouse. We imagine that for the novice and casual user alike, a hybrid interface could be highly useful.

The Gestalt Principles

Proximity – In the shell there is a fixed amount of space between lines and individual characters is fixed. However, textual output can still take advantage of the principle of proximity by using whitespace to separate items that should not be grouped together. This achieves the goal of grouping related objects in a text-based interface.

Similarity – Text mainly conveys semantic information, and we can use a few techniques to visually tag that information so that the user can easily determine the type of the text theyre reading and writing. Specifically, automatically making use of output coloring and syntax highlighting can help CLI a user pick apart information and group similar words within the buffer.

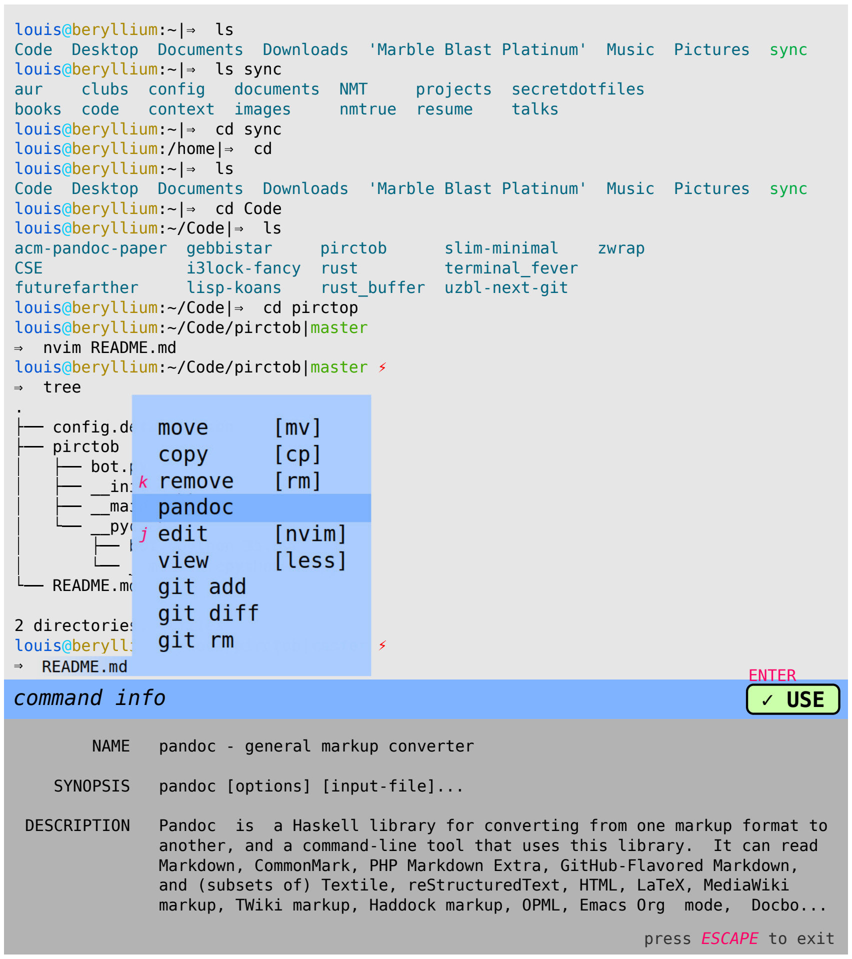
Past Experience – Most importantly, we need to appeal to the experiences of both novice and advanced users. Those who have never used the shell will expect their filesystem to be presented to them in a way that reminds them of a file browser - which is why we want to include a graphical pane which can be used to fill in commands equivalent to point and click actions. More advanced users will be more familiar with the basics of common commands, and can benefit from command completion that will make it easier to type the things that they type most often.

Color and light patterns could be used to convey information about the current session. In class the structure of eyes was discussed, with a focus on what sort of light is most easily noticed by different sections of they eye. In a text-based interface this theory could be used to craft notifications that catch the users attention based on the distance of the notification from the users likely focus on the screen.

8 Interface Design

We developed several mockups to illustrate the important design elements which we consider important for developing our prototype.

8.1 Command Info



```
louis@beryllium:~|⇒  ls
Code  Desktop  Documents  Downloads  'Marble Blast Platinum'  Music  Pictures  sync
louis@beryllium:~|⇒  ls sync
aur    clubs  config   documents  NMT     projects  secretdotfiles
books  code   context  images     nmtrue  resume    talks
louis@beryllium:~|⇒  cd sync
louis@beryllium:/home|⇒  cd
louis@beryllium:~|⇒  ls
Code  Desktop  Documents  Downloads  'Marble Blast Platinum'  Music  Pictures  sync
louis@beryllium:~|⇒  cd Code
louis@beryllium:~/Code|⇒  ls
acm-pandoc-paper  gebbistar     pirctob      slim-minimal    zwrap
CSE               i3lock-fancy  rust         terminal_fever
futurefarther     lisp-koans    rust_buffer  uzbl-next-git
louis@beryllium:~/Code|⇒  cd pirctob
louis@beryllium:~/Code/pirctob|master
⇒  nvim README.md
louis@beryllium:~/Code/pirctob|master ⚡
⇒  tree
.
├── config.de
├── pirctob
│   ├── bot.p
│   ├── __in
│   ├── __ma
│   └── __py
│       ├──
│       └──
└── README.mc

2 directorie
louis@berylli                        ⚡
⇒  README.md
```

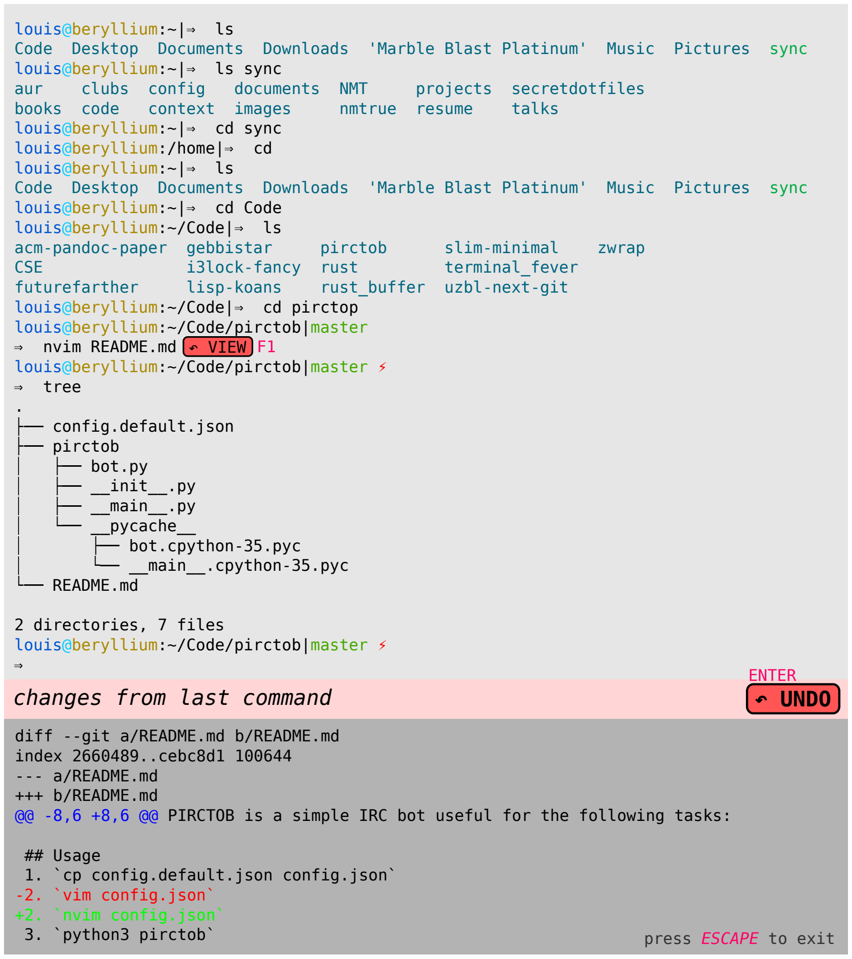
```
        NAME   pandoc - general markup converter

    SYNOPSIS   pandoc [options] [input-file]...

 DESCRIPTION   Pandoc  is  a Haskell library for converting from one markup format to
               another, and a command-line tool that uses this library.  It can read
               Markdown, CommonMark, PHP Markdown Extra, GitHub-Flavored Markdown,
               and (subsets of) Textile, reStructuredText, HTML, LaTeX, MediaWiki
               markup, TWiki markup, Haddock markup, OPML, Emacs Org  mode,  Docbo...

                                                                  press ESCAPE to exit
```

Figure 19: A mockup of the autocomplete dialog with the command info window

Figure 19 is illustrates our concept for the autocomplete dialog. The user can access the dialog by tab-completing the command they are typing, or clicking on a the name of a file in the output of any command. The dialog box can show a list of possible commands, determining which suggestions are most likely based on command history, the current working context, or the type of the file to be acted on. The autocomplete open remains open, leaving the user a trail of "breadcrumbs" which can lead them back to other commands' information windows.

The command info window is parsed from the man pages of the suggested commands, and the layout is right/left aligned to make it easier to read. The window is mainly a menu page for the particular command the user is interested in, but doubles as an input form for the CLI. The possible command line options that will appear in the documentation can be treated as form controls, (conceptually similar to checkboxes), which allow the user to interactively build commands by clicking or tabbing through.

8.2 Undo Window



```
louis@beryllium:~|⇒  ls
Code  Desktop  Documents  Downloads  'Marble Blast Platinum'  Music  Pictures  sync
louis@beryllium:~|⇒  ls sync
aur    clubs  config   documents  NMT     projects  secretdotfiles
books  code   context  images     nmtrue  resume    talks
louis@beryllium:~|⇒  cd sync
louis@beryllium:/home|⇒  cd
louis@beryllium:~|⇒  ls
Code  Desktop  Documents  Downloads  'Marble Blast Platinum'  Music  Pictures  sync
louis@beryllium:~|⇒  cd Code
louis@beryllium:~/Code|⇒  ls
acm-pandoc-paper  gebbistar     pirctob      slim-minimal    zwrap
CSE               i3lock-fancy  rust         terminal_fever
futurefarther     lisp-koans    rust_buffer  uzbl-next-git
louis@beryllium:~/Code|⇒  cd pirctop
louis@beryllium:~/Code/pirctob|master
⇒  nvim README.md
louis@beryllium:~/Code/pirctob|master ⚡
⇒  tree
.
├── config.default.json
├── pirctob
│   ├── bot.py
│   ├── __init__.py
│   ├── __main__.py
│   └── __pycache__
│       ├── bot.cpython-35.pyc
│       └── __main__.cpython-35.pyc
└── README.md

2 directories, 7 files
louis@beryllium:~/Code/pirctob|master ⚡
⇒
```

```
diff --git a/README.md b/README.md
index 2660489..cebc8d1 100644
--- a/README.md
+++ b/README.md
@@ -8,6 +8,6 @@ PIRCTOB is a simple IRC bot useful for the following tasks:

 ## Usage
 1. `cp config.default.json config.json`
-2. `vim config.json`
+2. `nvim config.json`
 3. `python3 pirctob`
```

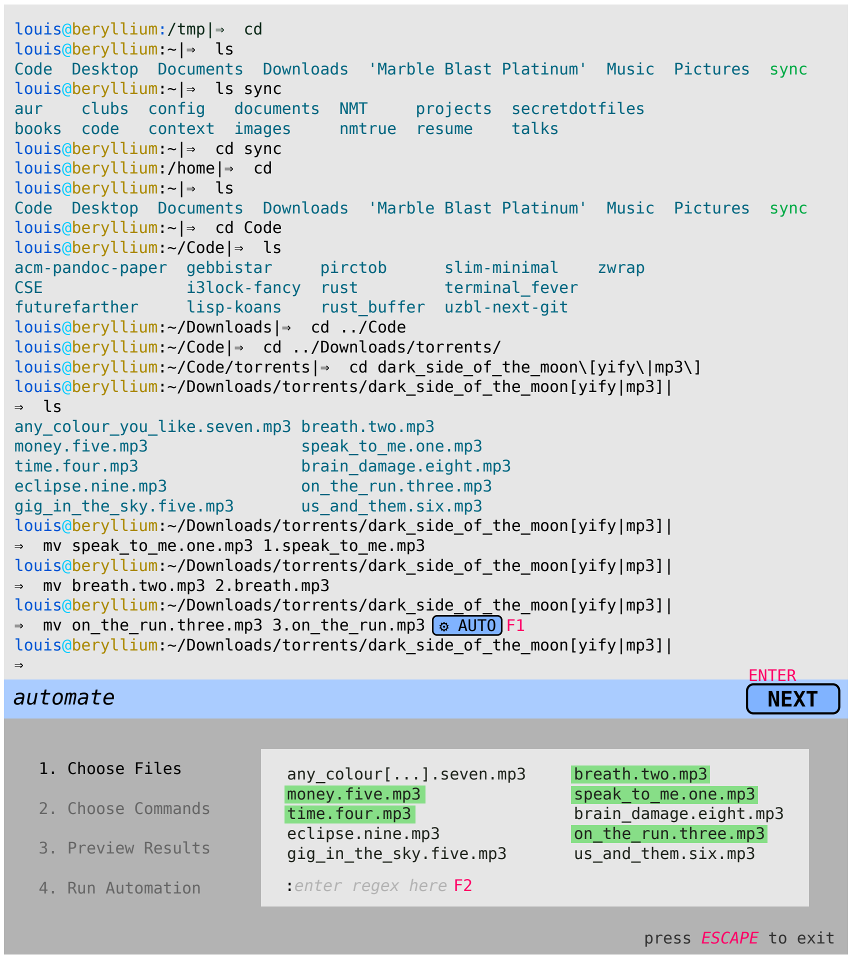
Figure 20: A mockup of the undo window with undo option

In Figure 20 we can see an example interface for the undo window for our terminal. After a command is entered that changes something, this shows the user exactly what changes have taken place, and makes it possible to undo the last set of changes. After any state-altering command is entered at the CLI, a button is appended to the command in the log which brings up the shown dialog. The same dialog can be made available from the heads-up display in the bottom, or the terminal can be configured to show the undo window after destructive commands by default.

The point of this interface is to create an escape hatch to allow the user to return the entire system to the state before the previous command. This is an example of responsive disclosure, because the information is only made available to the user when they indicate that they want to know how to undo a command. The system is also inherently limited in that it can only undo a single command, so as more commands are entered the button should disappear from older commands that can no longer be undone. This follows the principle of responsive enabling.

The undo button closes the pane by undoing the command, and pressing `ESC` simply returns the user to the normal context window without undoing anything. Both are examples of having a prominent "done" button, so the user knows how to tell the terminal they're done with the undo dialog. The red color on the dialog window also serves as a visual warning to the user to indicate that what they are doing will ultimately result in a real change to the state of the system, to make them more likely to exercise proper caution.

8.3 Automation Helper



```
louis@beryllium:/tmp|⇒  cd
louis@beryllium:~|⇒  ls
Code  Desktop  Documents  Downloads  'Marble Blast Platinum'  Music  Pictures  sync
louis@beryllium:~|⇒  ls sync
aur    clubs  config   documents  NMT     projects  secretdotfiles
books  code   context  images     nmtrue  resume    talks
louis@beryllium:~|⇒  cd sync
louis@beryllium:/home|⇒  cd
louis@beryllium:~|⇒  ls
Code  Desktop  Documents  Downloads  'Marble Blast Platinum'  Music  Pictures  sync
louis@beryllium:~|⇒  cd Code
louis@beryllium:~/Code|⇒  ls
acm-pandoc-paper  gebbistar     pirctob      slim-minimal    zwrap
CSE               i3lock-fancy  rust         terminal_fever
futurefarther     lisp-koans    rust_buffer  uzbl-next-git
louis@beryllium:~/Downloads|⇒  cd ../Code
louis@beryllium:~/Code|⇒  cd ../Downloads/torrents/
louis@beryllium:~/Code/torrents|⇒  cd dark_side_of_the_moon\[yify\|mp3\]
louis@beryllium:~/Downloads/torrents/dark_side_of_the_moon[yify|mp3]|
⇒  ls
any_colour_you_like.seven.mp3 breath.two.mp3
money.five.mp3                speak_to_me.one.mp3
time.four.mp3                 brain_damage.eight.mp3
eclipse.nine.mp3              on_the_run.three.mp3
gig_in_the_sky.five.mp3       us_and_them.six.mp3
louis@beryllium:~/Downloads/torrents/dark_side_of_the_moon[yify|mp3]|
⇒  mv speak_to_me.one.mp3 1.speak_to_me.mp3
louis@beryllium:~/Downloads/torrents/dark_side_of_the_moon[yify|mp3]|
⇒  mv breath.two.mp3 2.breath.mp3
louis@beryllium:~/Downloads/torrents/dark_side_of_the_moon[yify|mp3]|
⇒  mv on_the_run.three.mp3 3.on_the_run.mp3 ⚙ AUTO F1
louis@beryllium:~/Downloads/torrents/dark_side_of_the_moon[yify|mp3]|
⇒
```

```
any_colour[...].seven.mp3   breath.two.mp3
money.five.mp3              speak_to_me.one.mp3
time.four.mp3               brain_damage.eight.mp3
eclipse.nine.mp3            on_the_run.three.mp3
gig_in_the_sky.five.mp3     us_and_them.six.mp3

:enter regex here F2

press ESCAPE to exit
```

Figure 21: An example use of the automation window

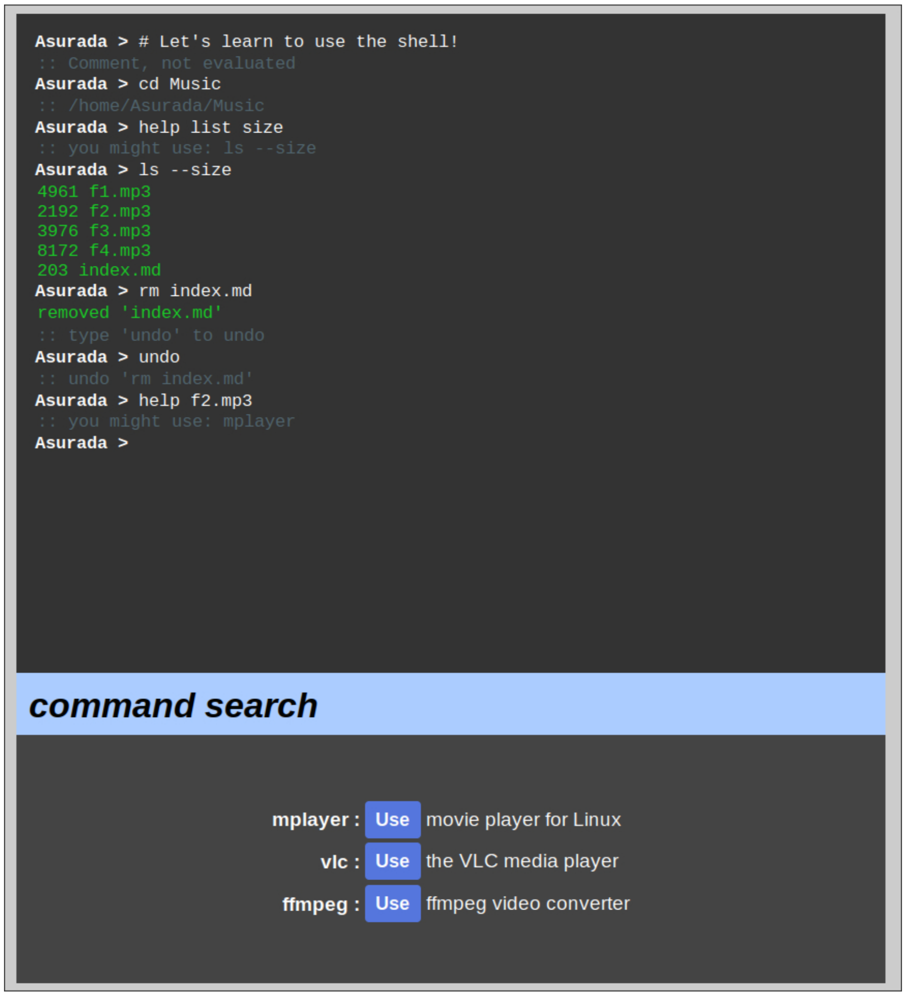
One of the main things that makes the command line powerful, and one of the hardest features to use effectively, is the ability to automate common system tasks. Figure 21 shows an example interface for a feature which can guide users through the process of automating tasks. The interface makes use of the users command history to detect when the same command has been used several times in the row, and presents the user with a button to bring up the automation window – this follows from the concept of responsive enabling.

The automation window is a sequence map that walks the user through the steps of creating a loop in their shell's language to repeat a task multiple times. In this example, the terminal is helping the user iterate over file names, so it creates a form where the user can select files, (again, similar to checkboxes), to add to the loop. There is also a textbox where the user can input formatted text to select files with shell wildcards – this is indicated to the user by adding an input hint to the form.

8.4 Prototype

To better demonstrate our interface design we've developed a web-based prototype. This prototype reproduces some of the behavior discussed in the above three examples. Namely the highly visible contextual information, the ability to undo state changing commands, and assistance in automating tedious tasks.

We've constructed this tool using web technologies (HTML, CSS, and JavaScript), along with the jquery and jquery-console libraries. This prototype can be accessed at the following URL: `http://infohost.nmt.edu/~ljencka/terminal_fever/`. We've included a few screenshots of our prototype to show what an early implementation might look like.

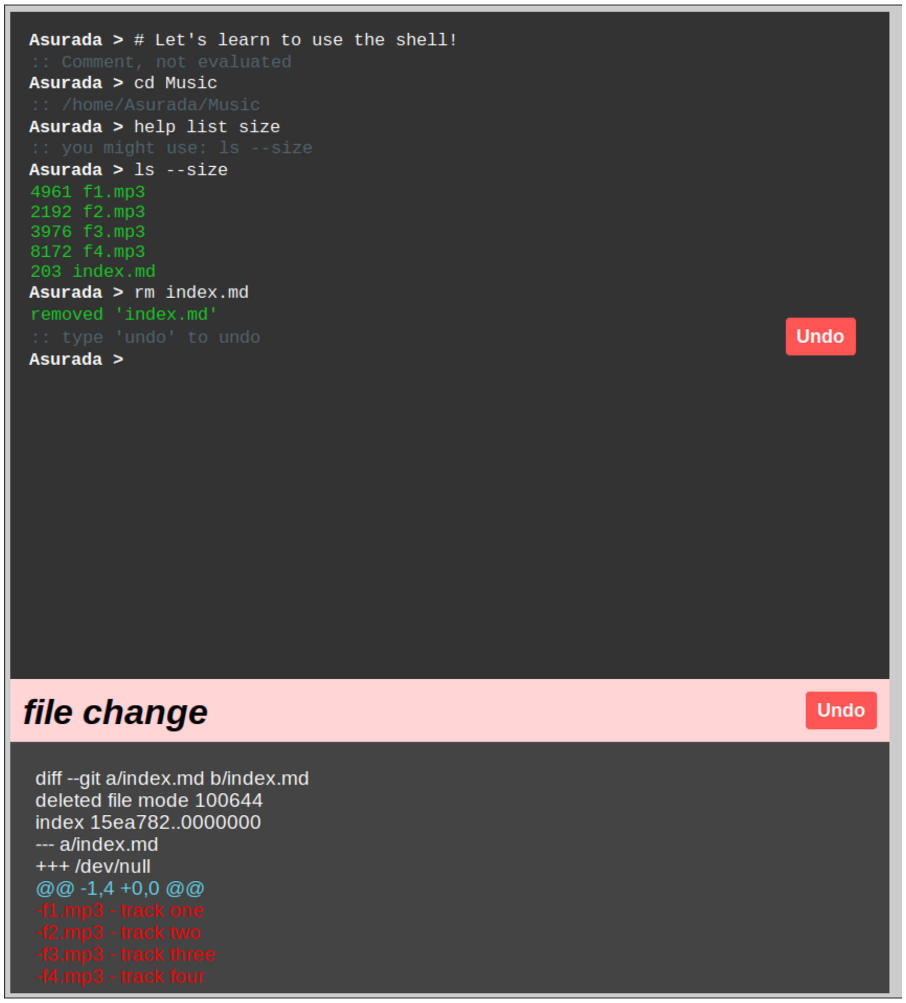


```
Asurada > # Let's learn to use the shell!
:: Comment, not evaluated
Asurada > cd Music
:: /home/Asurada/Music
Asurada > help list size
:: you might use: ls --size
Asurada > ls --size
4961 f1.mp3
2192 f2.mp3
3976 f3.mp3
8172 f4.mp3
203 index.md
Asurada > rm index.md
removed 'index.md'
:: type 'undo' to undo
Asurada > undo
:: undo 'rm index.md'
Asurada > help f2.mp3
:: you might use: mplayer
Asurada >
```

command search
mplayer : Use movie player for Linux
vlc : Use the VLC media player
ffmpeg : Use ffmpeg video converter

Figure 22: A prototype displaying the command search function

In Figure 22 we see the result of user typing `help` in the shell. The context window displays search results suggesting what the user might want to do with the given input file. Each suggested command can be autocompleted in the shell by clicking the "Use" button, or tabbing through the suggestions.



```
Asurada > # Let's learn to use the shell!
:: Comment, not evaluated
Asurada > cd Music
:: /home/Asurada/Music
Asurada > help list size
:: you might use: ls --size
Asurada > ls --size
4961 f1.mp3
2192 f2.mp3
3976 f3.mp3
8172 f4.mp3
203 index.md
Asurada > rm index.md
removed 'index.md'
:: type 'undo' to undo
Asurada >
```

```
diff --git a/index.md b/index.md
deleted file mode 100644
index 15ea782..0000000
--- a/index.md
+++ /dev/null
@@ -1,4 +0,0 @@
-f1.mp3 - track one
-f2.mp3 - track two
-f3.mp3 - track three
-f4.mp3 - track four
```

Figure 23: A prototype displaying the option to undo a command

Figure 23 shows a command that changed the state of the system – specifically, the user deleted a markdown file. Our system can use git to backup the state of the system as the user works, so it is possible to undo these changes. If the user clicks "Undo" or types it at the command line, then the file will be restored.

9 Evaluation Planning

Evaluating the effectiveness of our design is an important step in understanding how we can and have effected users abilities with and knowledge of the command line. An important metric for this would be *unique command count* — the measure of how many unique commands the user is able to use. This includes things such as looping, options, as well as actual commands. Our goal is to understand the impact our designs could have on unique command count, as well as general user experience.

We hypothesize the following:

- Users who use our program will have a higher unique command count than the control group.
- Users who use our program will report a more positive user experience.
- Users who use our program will maintain their unique command count outside of our shell.

These captor the idea that the users should begin using the shell better, have less fear of the shell, and will be able to carry this knowledge to other systems.

9.1 Collecting Data

Collecting data to this end will take place in two parts:

- A user survey to gauge users emotional experience — how much they enjoyed using the command line.
- A data reporting study, which would report command history for a number of users.

These would each need to be run for our two groups: control and enhanced. Likely our study would require each group (of mixed experience levels), to use our software for a period of time. Then we would give them a number of tasks to perform. After they have completed the tasks, we can collect statistics, and conduct the survey.

Our user survey would need to focus in on the emotional experience of our users. It would ask questions such as

- Rank your intimidation of the system.
- Rank how hard it was to perform a given task.
- Rank how well you think you used the system.
- Rank how happy you were with your experience.

Automated usage statistics would be much easier. All we would need to do is get the history file for each user, and look at the unique command usage.

Together we can use these to test the first two of our hypotheses. For the third, we would require users to switch from our system to a traditional system after an extended period. At that point we could likely re-conduct our analysis to see if users have learned skills in a way they can reproduce. Figure 24 shows something of a timeline on how this would work to

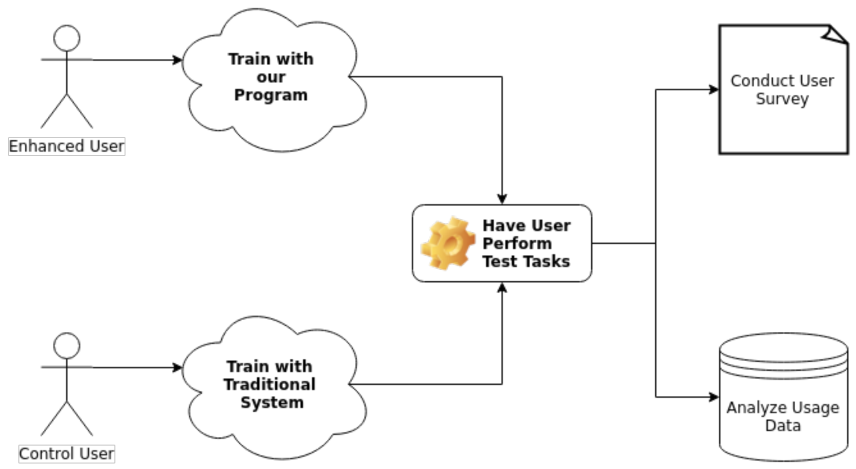


Figure 24: This shows a timeline of how we would evaluate our design.

develop our understanding. Again, in it we have two groups of users. Each receives training in a different command line system. Each will be given a set of tasks after some period of time. After these tasks, we can begin to analyze the results.

10 Results

10.1 Data Collection

8a. Explain how you collect the data.

10.2 Data Analysis

8b. Describe the tools you used in analyzing the collected data.

10.3 Statistical Analysis

8c. List the statistics of participants' demography.

10.4 Qualitative Analysis

8d. Describe the quantitative/qualitative analysis results with proper statistics test/grounded theory. You should also indicate whether or not the analysis result can support the hypotheses in Section 7.