Lightening the Cognitive Load of Shell Programming

Ishaan Gandhi[*] **Anshula Gandhi**[†]

November 2020
CMU-ISR-20-115B

Institute for Software Research
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

*This paper was presented at PLATEAU 2020:
The 11th Annual Workshop on the Intersection of HCI and PL
November 2020, Co-located with SPLASH 2020*

[*]Department of Computer Science, Harvey Mudd College, Claremont, CA, USA
[†]Department of Brain and Cognitive Sciences, Massachusetts Institute of Technology, Cambridge, MA, USA

Keywords: Terminal, Shell, Scripting Languages

Abstract

Terminal emulators, or simply terminals, are used ubiquitously by developers. While many have proposed alternatives, this paper examines the fundamental reasons why shell programming, especially when using a terminal as a programming environment, can be difficult, as understood through the Cognitive Dimensions Framework. We will present a task analysis of the shell programming language itself (which we'll refer to as "the shell") and the application most often used to interact with it (which we'll refer to as "the terminal"). We lay out many usability problems of interactive programming via shell in the hopes that tool developers may be able to build upon this analysis in the future.

Lightening the Cognitive Load of Shell Programming

Ishaan Gandhi
Harvey Mudd College, Claremont, CA, US
igandhi@hmc.edu

Anshula Gandhi
MIT, Cambridge, MA, US
anshula@mit.edu

Abstract

Terminal emulators, or simply terminals, are used ubiquitously by developers. While many have proposed alternatives, this paper examines the fundamental reasons why shell programming, especially when using a terminal as a programming environment, can be difficult, as understood through the Cognitive Dimensions Framework [7]. We will present a task analysis of the shell programming language itself (which we'll refer to as "the shell") and the application most often used to interact with it (which we'll refer to as "the terminal"). We lay out many usability problems of interactive programming via shell in the hopes that tool developers may be able to build upon this analysis in the future.

2012 ACM Subject Classification Software and its engineering → Scripting languages; Human-centered computing → Command line interfaces

Keywords and phrases Terminal, Shell, Scripting Languages

Acknowledgements We want to thank Will Crichton, to whom we owe many structural ideas of this paper. We also thank Lucas Bang for his valuable feedback.

1 Introduction

The "principles of cognitive dimensions" create a way to evaluate programming languages or programming environments based on how the notation of the language adds or alleviates the cognitive load on a programmer. These dimensions include, among others, **consistency** (how much of the usage of the language can be inferred), **progressive evaluation** (if partially complete programs can still be executed to aid the developer), and **hard mental operations** (if the user might need to take separate notes to make sense of the notation). Ultimately, "the understandability of a programming language depends on the match between the way it is structured and the type of question to be answered" [7]. These dimensions help to measure this match. Throughout the paper, we will emphasize these dimensions in bold type for clarity. In this paper, we will provide a task analysis of shell programming via the terminal, emphasizing difficulties, and including justifications from both the cognitive dimensions theory and anecdotal programmer experience. What assumptions about shell programming are implicit in the design of the terminal? We draw attention to areas where the terminal provides a poor fit for the kinds of tasks users want to undertake. While the focus of this paper is on interactive shell programming, many of the observations will be applicable to other manners of using the shell language (e.g. shell scripts and make files).

Programming in shell via a terminal involves two tasks:

- Writing a single *shell command.* A command containing only a program name and its flags, or a shell builtin, like `cd` or `source` is called *simple.* A command can also contain constructs like pipelines, lists, loops, conditionals, and groups. [5] We will call this second type of command *compound*, and any single command, whether simple or compound, a *statement.*

- Issuing multiple statements in order to accomplish a task. As we will see, unique challenges arise when commands are issued as individual statements, (for example, `cd /tmp/`, then `rm -rf`) instead of as an equivalent compound command, (`cd /tmp/ && rm -rf`.)

We break this paper down, accordingly, into a task analysis of writing single commands, and a task analysis of issuing multiple statements.

2 Task Analysis

2.1 What makes writing each command hard?

The terminal emphasizes issuing commands as one-offs as opposed to repetitive actions

Yet, in practice, many commands are actually needed repeatedly. Within a project, a programmer frequently runs the same few commands, which are often long. Think tight loops of compile, test, debug, edit.

How can a programmer find a command they previously ran in order to run it again? Remembering which previously used command pairs with which action is a **hard mental operation**. In our experience, terminal users complete such a task by piecing together evidence from their shell's history. If they are lucky, they might remember a substring of the command and find it with a reverse search. Without such luck, they are relegated to repeatedly pressing the up key. In either case, they must filter out irrelevant commands along the way and remember the correct order to run the relevant ones.

Both command names and flags are hard to remember. To keep terminal usage concise, users often prefer to use shortened flags or commands instead of the full version. These shorthands make the terminal productive but are not **role expressive**. Would a programmer be able to scroll back through their terminal history and understand what they did? Would they be able to find the command they previously used with a reverse search? For many programmers, the answer is "probably not".

Suppose a programmer wants to write a compound command by composing together 5 simple commands. If they are typing them in a terminal, they might first make sure that simple command `A` does what they want it to do. Then, they'd want to make sure `A` and `B` work correctly together, perhaps typing `B | A`. Then they might add in `C`, `B | A && C`. In general, building up to a series of N simple commands in this way would require typing on the order of N^2 commands. The user might therefore decide to just write out all the commands in one go (`B | A && C | D; E`) to spare themselves this process. The terminal is not conducive to **progressive evaluation**. The programmer had to retype commands because the outputs were needed by later commands. Could a terminal let programmers use the outputs of previous commands as inputs into future commands?

Progressive evaluation is important because programmers prefer to code "in little spurts (possibly corresponding to mental chunks or schemas) which are knitted into what has been written so far" [7] [6]. Novice programmers especially rely on progressive evaluation: "the less experienced the programmer, the smaller the amount that is produced before it must be evaluated" [7]. Thus, shell programming with software that is less-than-conducive to progressive evaluation may prove to be a stumbling block allowing more entry of programmers into the field. What would software that emphasized progressive evaluation look like? Would programmers make fewer mistakes if they could view the entirety of a command's output before it gets piped into the next command?

The terminal forces users to trade off between terseness and visibility. This makes commands, especially when compound, hard to debug.

Expansion occurs in a way that is not observable, and the behavior of the program is not visible from reading the statement. For example `rm -r $(./files_to_delete)` does not tell you what files are going to be deleted. While powerful, shell expansions like these make it easy to make mistakes. A better terminal might allow a user to expand some expressions as the shell evaluates the code.

The conflict between **terseness** and **visibility** is also apparent in the ubiquity of aliases [9]. Aliases provide a way for programmers to reuse code without retyping an entire command. However, when a command is aliased, there is almost no visibility into what command the shell will run. The command being run could have come from any number of sourced files.

The terminal has a textual user interface (TUI) as opposed to the more common graphical user interface (GUI). The standard idioms for interaction used by most other applications do not work in the terminal.

The TUI on terminals (on MacOS, the Emacs key bindings, shortcuts, and commands) are in principle perfectly productive. In practice, most programmers do not understand them well enough to use them productively.

Suppose a programmer needs to change a character in the middle of a 40 character command, and their cursor is on the last character. A savvy Emacs user can quickly perform the desired change. In practice, only 4.5% of developers use Emacs as their editor [13]. The rest of us are pressing the left-arrow key 20 times, a form of **repetition viscosity**.

Programmers might make their terminal experience more similar to what they are used to by, for example, using alt-click to move their cursor with the mouse, or remapping `Ctrl-R` to the more idiomatic `Ctrl-F`. Even skilled typists have little explicit knowledge of key locations on the keyboard, and instead type reflexively. Novel tasks like learning a new keybinding, by contrast, require loading explicit knowledge into working memory [15]. One might hypothesize that users can therefore use the wrong shortcut through muscle memory, even if they consciously know the shortcut is different on different programs. The issue at hand is the **closeness of mapping** between what the user expects the notation to be, and what it is. A study of how terminal designers might make these controls more discoverable to programmers accustomed to GUIs would be a valuable contribution.

The shell interpreter in a terminal is a read-eval-print-loop, or REPL

REPLs, by definition, offer the programmer only 1 statement of editable text at a time. Once a statement is issued, it is no longer editable. The space to edit a command is restricted to a single line at the bottom of the screen. However, large windows of spaces to edit are important to programmers. Research into programmer experience shows that “an editor which allows easy access to a large window of code makes the [editing] cycle easier...” [7] [4].

2.2 Why is issuing multiple statements hard?

Writing a command requires knowledge of every command that has come before it in the shell, since any could potentially alter the shell’s state.

The state of a shell session can affect the semantics of a command. For example, environment variables, such as `PATH` or `PWD` can make the same command run a completely different program. This is a **hidden dependency**: the relationship between the command a programmer ran and the environment variable the command made use of is not visible from just the command. This can make it hard to reproduce the result of a command.

Although most programming languages support both global and local variables, programmers should default to locals, using globals only when necessary. Could we make terminals that more closely mirror conventional wisdom in most languages?

Context is lost when searching for individual statements in a command history

In Jupyter notebook, Python code is often chunked into semantic blocks, with a few lines grouped together for one clear purpose, and the next few grouped together for another purpose. By contrast, in a Python REPL, it can be more difficult to figure out what the larger purpose of a line of code is.

The same difficulty of understanding the context of a previously run command exists in terminal. The design of the terminal emphasizes *temporal* **visibility**; it can show a programmer inputs and outputs in the order they were produced in. However, it won't show which commands should be run consecutively or why they should be run. Notebook computing applied to shell programming might aid in code readability, as it does for Python users.

Comparing multiple commands along an axis is limited by terminal

Suppose a programmer has run a particular command numerous times, with numerous parameters and flags. If that user then wants to see all the different flags they used the program with, their best option might be to use the `history` command. However, this can be cognitively demanding for the user, as they scroll through various unrelated commands, only looking for and comparing the ones that they are interested in.

We can justify this readability difficulty theoretically. Suppose the programmer has run a command K different times with various flags. They want to search through their shell history for all K variants to compare them, but their terminal window will only show them $N << K$ commands at a time. As they scroll off to another part of the screen, they can likely only keep about 4 commands in their working memory [3]. However, they need to remember all $K - N$ other commands in order to do the comparisons. Thus, the poor **juxtaposability** of this interface adds an additional cognitive burden to the programmer.

Easing this burden might involve showing the bash history of only the commands in the project or "notebook" are currently working on, or showing commands sorted by command prefix rather than time executed.

3 Potential Remedies

We can't get away from the command line. Even though Netflix has decided to forgo shell scripts and migrate towards Jupyter notebooks with Python, the software they're using to organize the execution of these notebooks requires heavy use of the command line [14, 12]. That is, they are likely still using a shell to execute the Jupyter notebook, to execute the Python that executes the Jupyter notebook, and to upload the notebook to a remote server.

Making terminal alternatives is a long-standing tradition in the developer community. One such alternative aims to make the terminal more visual by outputting images from the browser, maps from the internet, and visualizations of working directories and disk space [16]. A second alternative increases the visibility of shell computations by letting users step through command expansions and evaluations [8]. Another alternative involves using a Jupyter notebook with a bash kernel [10].

In addition, efforts by the zsh community help address some of the issues mentioned in this paper. For example, one widely used history-based autocompletion plugin unobtrusively presents users with previously run commands as they type, easing the burden of repeatedly recalling frequently used actions [2]. Fish shell addresses the visibility of aliases problem by replacing aliases with their definitions once executed [1].

Terminal designers might look to the Python community for inspiration. While Python ships with a REPL, many programmers who want to interactively use the language use Jupyter notebooks. While the primitive input in a REPL is a command, the primitive input in notebook computing is the cell. A cell is simply a series of commands that can be run together via a button or a keyboard shortcut. Notebook computing recognizes that commands Python users ran weren't necessarily one-offs, and lets users save their work. It also uses a more IDE-like UI, and shows context around individual statements. As with terminal, however, notebooks require users to have knowledge of every cell previously run to keep track of global state. Furthermore, the fact that the Python kernel is used ubiquitously with Jupyter, unlike the bash kernel, might indicate some other weaknesses of notebook computing for the interactive shell programming domain.

4 Conclusion

Programmers who get past the initial steep learning curve and appreciate the power of shell programming in the terminal might prefer the terminal to doing the same tasks through GUIs. Indeed, the move away from the command line and towards GUIs "has resulted in programs that are easier to learn and use, but harder to automate and reuse." [11].

However, "learnability" of software need not be in conflict with the automation and reusability benefits of that software for power users. For example, consider the addition of idiomatic keybindings to the terminal. A terminal using `Ctrl-F` instead of the default `Ctrl-R` for searching through command history, as mentioned in section 2.1, might increase "learnability" without harming reusability.

We advocate creating terminal alternatives that apply the design principles mentioned in this paper, in order to create programs that are still easy to learn and use, without sacrificing the automation advantage that shell provides. We hope these design principles based in cognitive science will allow shell computing to be less of a cognitive burden, freeing up bandwidth for the more impactful activities in a programmer's life.

References

1 *Friendly interactive shell*, (accessed 2020). URL: `https://fishshell.com/`.

2 *Zsh-autosuggestions*, (accessed 2020). URL: `https://github.com/zsh-users/zsh-autosuggestions`.

3 Nelson Cowan. The magical number 4 in short-term memory: A reconsideration of mental storage capacity. *Behavioral and brain sciences*, 24(1):87–114, 2001.

4 Simon P. Davies. Externalising information during coding activities: Effects of expertise, environment and task. *ESP*, 93(744):42–61, 1993.

5 Free Software Foundation. *Bash Builtin Commands*, (accessed 2020). URL: `https://www.gnu.org/software/bash/manual/html_node/Bash-Builtins.html`.

6 Thomas R. G. Green, Rachel K. E. Bellamy, and J.M. Parker. Parsing and gnisrap: A model of device use. In *Human–Computer Interaction–INTERACT'87*, pages 65–70. Elsevier, 1987.

7 Thomas R. G. Green and Marian Petre. Usability analysis of visual programming environments: a 'cognitive dimensions' framework. *Journal of Visual Languages Computing*, 7(2):131–174, 1996.

8 Michael Greenberg. Smoosh: the symbolic, mechanized, observable, operational shell, 2020. Current version: 0.1. URL: `https://github.com/mgree/smoosh`.

9 Seth Kenlon. *Bash aliases you can't live without*, (accessed 2020). URL: `https://opensource.com/article/19/7/bash-aliases`.

10 Thomas Kluyver. A jupyter kernel for bash, (accessed 2020). URL: `https://github.com/takluyver/bash_kernel`.

11 Robert C. Miller and Brad A. Myers. Integrating a command shell in a web browser. In *Proceedings of 2000 USENIX Annual Technical Conference*, 2000.

12 Nteract. Papermill: Parameterize, execute, and analyze notebooks, (accessed 2020). URL: `https://github.com/nteract/papermill`.

13 Stack Overflow. *Developer Survey Results*, 2019 (accessed 2020). URL: `https://insights.stackoverflow.com/survey/2019#development-environments-and-tools`.

14 Matthew Seal, Kyle Kelley, and Michelle Ufford. *Scheduling Notebooks at Netflix*, (accessed 2020). URL: `https://netflixtechblog.com/scheduling-notebooks-348e6c14cfd6`.

15 Kristy M Snyder, Yuki Ashitaka, Hiroyuki Shimada, Jana E Ulrich, and Gordon D Logan. What skilled typists don't know about the qwerty keyboard. *Attention, Perception, & Psychophysics*, 76(1):162–171, 2014.

16 Pramod Verma. Gracoli: a graphical command line user interface. In *CHI'13 Extended Abstracts on Human Factors in Computing Systems*, pages 3143–3146. 2013.