Skinning the Command Line

Improving the Unix Command Line Experience for Undergraduate Computer Science Students

Christopher Özbek
Martin Modahl
Greg Conti

CS6750 Human Computer Interaction - Phase 4

1. Introduction

The primary goal of this project is to improve the utility and usability of the command line interface for undergraduate computer science students. Within the traditional Unix system there exists a great deal of knowledge in the environment that is accessible through a variety of tools. The student can access help through the *man, info, apropos* and *--help* commands as well as utilize web search engines, books and cheat sheets to help them accomplish their tasks. For the expert Unix command line user these tools are efficient and effective ways to acquire the information they need, unfortunately these tools can be frustrating for the novice. The learning curve for the Unix command line interface is notoriously steep.

We focus our research on undergraduate computer science students who are novice Unix command line users, but require additional skills for more advanced work within their discipline. We put particular emphasis at the critical point in the computer science curriculum where they are introduced to the Unix command line interface. We believe that enabling a smooth transition at this juncture will greatly improve their acceptance of and facility with the command line interface. Efficient and effective use of the command line interface is critically important for their future success in the more advanced parts of many undergraduate computer science programs. By learning the command line interface quicker they will be able to spend more time focusing on their tasks rather than learning to use the tools.

To summarize, the important characteristics of our target user group are:

- Significant background and understanding of high-level computer use, but minimal understanding of the syntax to execute at the command line.
- Anxiety
- Limited understanding of the Unix structure
- Limited ability to seek out and understand help using traditional sources
- Limited Unix vocabulary
- High error rate using the command line
- Uncertainty about the success or failure of their attempts
- Desire to become “elite” power users
- Ability to use only a limited subset of tools, but need to know more

These characteristics of users together with the tasks that they perform provide the fundamental focus of our system.

The environment we have specified is that of a stable computer environment where the user has direct access to the command line as well as access to online help tools. This could be in a computer lab, home office, library or study. The task environment is the command line itself. The command line can be reached remotely or locally from a myriad of computing platforms.

We developed several concepts that augment the command line and allow novice Unix computer science students to immediately use the system more effectively as well as gently shepherding them on their way to becoming expert users.

- *1. A Graphical User Interface (GUI) adapter using skins*
  This idea is based on a complete transparent addition to the command line that separates the command line window in two halves, where the upper half graphically represents the text-based lower half. If the user starts to type *tar* in the lower half, this action would cause the upper half of the command line window to transform into a graphical user interface for *tar*, specifying the most important command options, offering help and exploreability methods.

- *2. A gesture based interface*
  Gestures represent an almost untapped resource of interface possibilities. A user, for example, may hold their ears to signal a desire to reduce the volume of the speakers. Other examples include: pressing both hands flat against each other to signal compression, pulling them apart for uncompression, scissor movements with the fingers to cut and tapping various points on the body to access different files.

- *3. A game environment using a wizard school metaphor*
  This concept transforms the command line into a medieval setting much like a role-playing game. Spells, weapons and tools represent commands and are introduced level by level to the player/user. Simple commands such as *ls, cd, mv* and *rm* would be be learned early in the game. More advanced spells would be acquired later when the user has reached a certain level of proficiency with the system.

During this stage we combined the knowledge gained from Parts 1, 2 and 3; as well as new material from class and feedback to thoroughly evaluate our prototype. It is important to note that our prototype is not just a trivial mock-up. It is a fully operational implementation of our design built inside a Unix terminal application. It required extensive programming and is on the order of ~1000 lines of C code. At the completion of our project, it is our intent to release this code (under the GNU Public License (GPL)) and publish our results within the computer science education community.

Section 2 is a detailed description of the prototype which includes screen dumps and text to explain how users will interact with it, implementation challenges, justification on why we chose this design, what is special about the design with regard to our problem and what we wanted to build, but were unable to. Section 3 includes our usability specification and evaluation plan. Section 4 contains the results from our evaluation of the prototype we developed in part 3. Appendix A includes the raw data from our evaluation, Appendix B is our raw data summary spreadsheet, Appendix C is our IRB paperwork and Appendix D includes screen shots of our prototype. Appendix E is the set of slides from our in-class presentation.

2. Detailed Description of the Prototype

Introduction
If a user wants to use a specific Unix command, they must remember a specific set of options and arguments to the command. Normally, a user would need to search for these options and their format in *help* files, *man* pages and the *info* program. In this prototype, the system provides the user with built in help functionality by adding extra functionality to an existing interface. We began examining the command line by looking at the state of the art and realized that a simple, intuitive help system was one of the key points making the command line difficult to learn and use. We do not want to provide only an intuitive help interface, but a help interface that enables users to interactively build the command line commands. By overlaying interactive help functionality over an existing command line interface, we expect to give users a chance to get intuitive help and visible feedback building their command line commands.

After the advent of the Windows Icons Menus Pointers (WIMP) interface, users have become comfortable with pointing and clicking their way through their computer world. We hope to exploit this familiarity by giving users the option of selecting and editing the command on the command line by using standard WIMP abstractions. The literature diagnoses what the user needs in these visual WIMP interfaces and has informed our design.

Because of the nature of the command line, the interface is more of a language than an interface. Learning this language is the step new users must take to become power users. By giving users a WIMP interface to construct sentences in this language we can add significant power to the interface and give users help learning the command line language.

There are several clear advantages and disadvantages to this interface style:

Advantages:

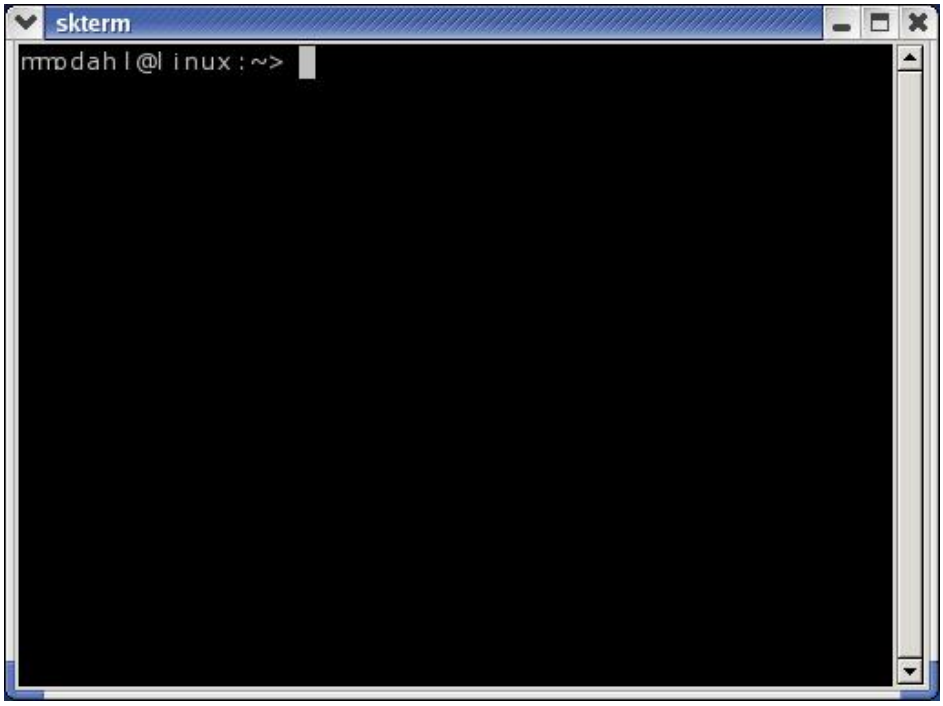
- Familiar interfaces joined together to create a new interface.
- Interface gives user feedback in the form of a command line that changes to show selected options and arguments.
- Exploreable interface lets users undo changes to command line.

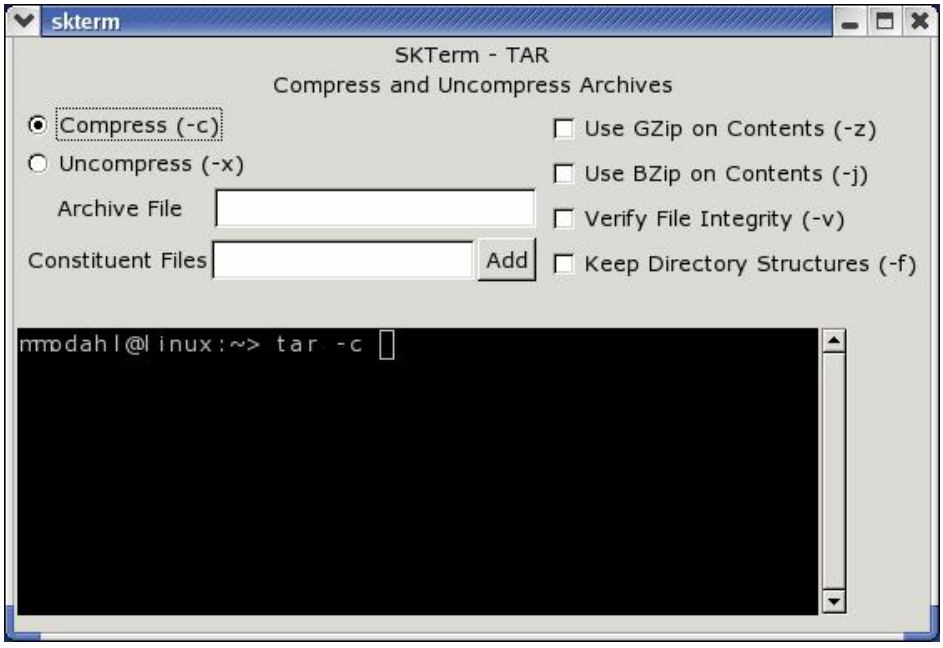
Disadvantages:

- Users may become reliant on the enhanced interface and never learn the command line commands the interface is intended to teach.
- The WIMP Interfaces that overlay the command line must be hand built or dynamically created by the system for each command line command a user might need help with. An open architecture that allowed a community of users to build and share these templates would help mitigate this concern.

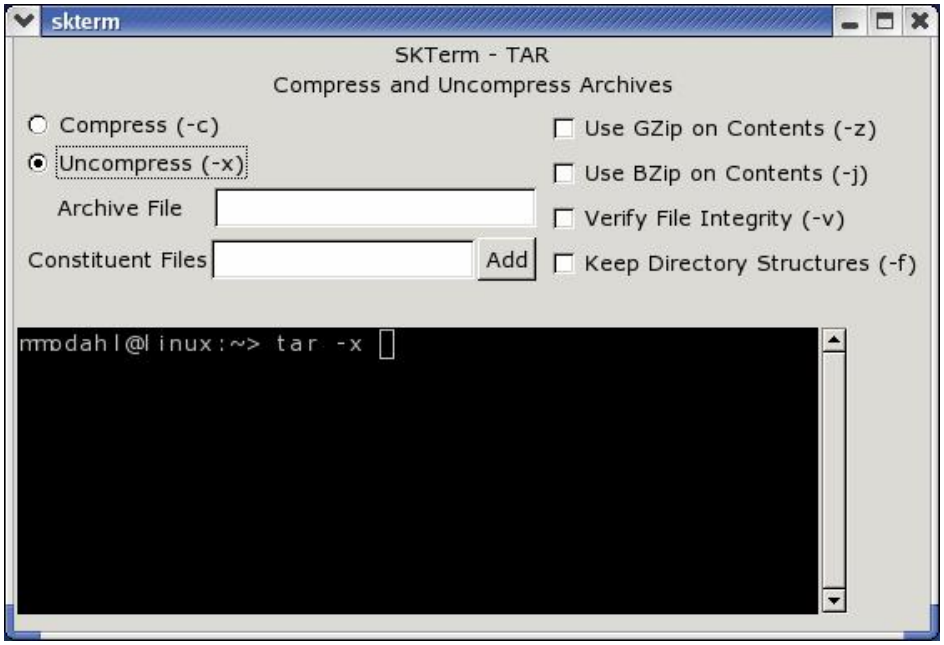
Narrative Walk-Through

To illustrate our design, the following is a sample scenario. In this scenario, the user is presented with a tarball archive (Unix slang for a group of files compressed using the *tar* command into a single file) of a homework assignment. She knows that she should use the *tar* command to uncompress the image. Once she has identified the command to use, she types it into the command line and presses the shortcut key to activate our program. The following figures illustrate this process.

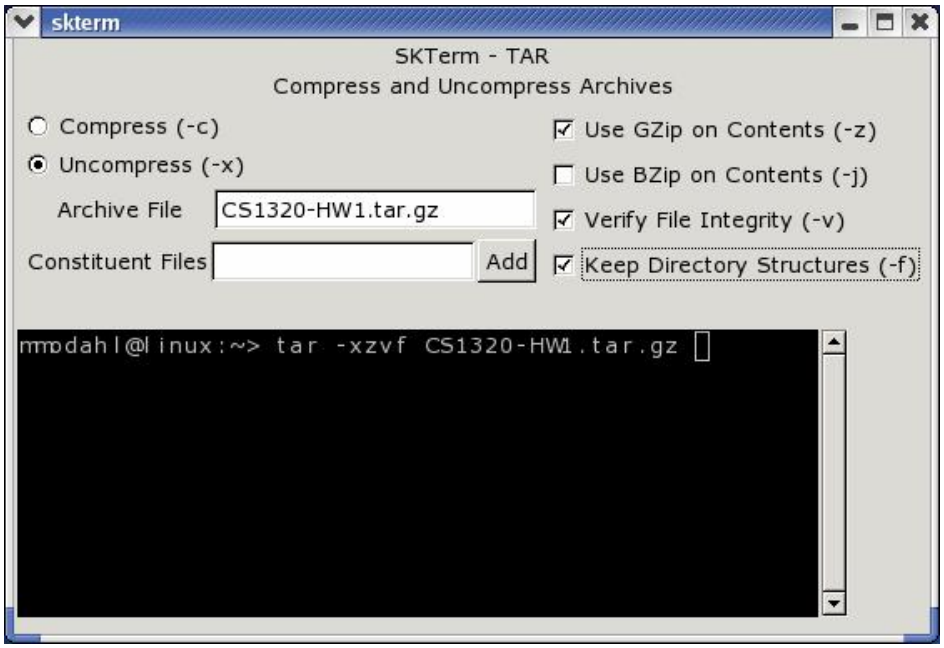




Once the overlay has been raised, the user can select the basic options of the command.



The user then adds the appropriate file as the archive in question and the interface completes the most common command line command for the file selected. The user then hits the *Enter* key.



With this interface, we expect that users would begin to internalize the knowledge of how the commonly used commands are formed by the GUI interface. By repeatedly using the GUI interface and watching how the GUI interface builds the command line, the user can begin to learn to use the command line.

Prototype description

When starting the prototype application the user is presented with a conventional terminal window that displays the current shell and waits for user input. But in contrast to traditional command-line application, the user of the prototype is not constrained to stay with the text view alone he always can request the window-screen to be split in two halves. While the terminal stays on the bottom half, the upper half displays the extended information panel which is a concise representation of the current terminal state.

Imagine for instance that the user just started the session and is in her home directory. In this initial state our prototype will tell the user which programs and functions it supports along with a short description of their purpose. The user can then either select one option by clicking on the provided short-cut button or proceed with the usual text-based command-line interfacing.

We have pushed our prototype mainly in the direction of helping the user to work with complicated command-line switches and added support for the programs `tar` and `ln`. When the user wants to use these commands the split-view will display check and text boxes alongside a short textual description of the program which should guide the user to perform the most usual tasks associated with the command.

Continuing our example we can imagine the user faced with an archive file she wants to uncompress. Selecting tar from the general description page the command line on the lower half is updated and now reads "`tar`". Since the user never uncompressed a file with tar before she usually would have to resort to the Linux/Unix manual system using the `man` or `--help` facilities. With the split-screen enabled the prototype shows her the most important and common options in a concise way. We feel that this is a major improvement from the traditional help system which failed to group knowledge in a descending order of importance and left the user searching in very technical and precise, yet voluminous, set of information. The user utilizes the mouse to select the respective switches for the un-gzip, extract and verbose options and then enters the file-name in the text-field. Each change, made in either view, is directly mirrored in the other view.

Justification:

When we had to decide on a prototype to implement, after our poster presentation in phase two, the following attributes influenced us and are special about the design:

- ***Feasibility***
  For a prototype to be implemented inside our hard time constraints and still retain the amount of real world usability we wanted a feasible design. The gesture interface to the command line from phase two clearly failed here to offer any real alternative. A prototype for this would clearly have overextended our capabilities and would have forced us to retreat onto the terrain of a pure mock-up without any "real" function. The split-screen prototype on the other side provided a real possibility for implementation because the relative independence of the individual screen for each application. By constraining on only a small number of programs

we wanted to support and evaluate we could accomplish our desired result within our time and resource constraints.

- ***Integration***
  This point is based on the insight, that software development is essentially evolutionary and coexisting instead of revolutionary and replacing. With Linux, this situation is especially aggravated by the fact that there is no central authority or company which has the resources and money to create a new reference standard. This forces software developers in the Open Source world to adhere to implicitly defined de-facto standards. Competition is highly embedded in this culture, i.e. despite the technical superiority world-class Linux applications such as the Apache web-server, they have essentially remained command-line tools. Recognition of this inherent nature of the Linux world is essential for the success of our prototype. The split-screen interface, with its reliance on an existing interface, showed clear superiority against the other two ideas for command-line improvement.

- ***Impact***
  A critical criterion for an academic prototype must always be the impact of the chosen design. This impact may be of scientific or real-world flavor, yet the message remains the same: A successful design changes the way in which a problem or issue is seen and solved by an essential part of the user population. Evaluating our ideas from phase two under the light of this criterion pushed us strongly in the direction of the split-screen. Only this idea seems ready to have a huge impact. In fact we believe that this idea is the first real improvement that the command-line interface experiences when ignoring the (not doubt useful) extended shell features like completion, history, etc.

- ***Power-user and beginner compatibility***
  When we began the contraction of our design-space after the expansion in the brainstorm phase we quickly agreed on this requirement: Our solution has to be compatible both with beginners and power-users. The former because they constitute our target user-population which are meant to benefit from our development and the latter because they will be teaching the former.

  To attract the beginning users who are unfamiliar with the command-line metaphor and overcome their anxiety, the prototype tries to provide as much WIMP-scaffold as possible without removing the command-line completely. Ideally we believe our system could engulf all possible usages of the command line using the windowing interface and render the textual description into insignificance if the user does not want to be bothered by it.

  On the other hand, failure to make power-users comfortable with the provided software, will result in rejection and feed back into the beginner population. We draw this conclusion from our initial user assessment in phase one. The power-user and the “Linux hacker” embody the image of in-depth “elite” knowledge that is the goal of many students. The creation of a tool that is not part of the “real”

Linux community and just an academic or educational “toy” will most probably lead to rejection by the beginners.

We feel that the split-view can bridge this gap in a very convenient way: Since the additional scaffolding view is purely optional and only positively extends the functionality of the existing solutions we should not hear any complaints that the technology interferes or even “taints” the elegance of existing terminal software.

- ***Extensibility:***
  The path that has been begun with the prototype leads directly to an open, flexible and extensible framework structure for augmenting the command-line experience. To accommodate the diverse development community based on the GNU Public License that clearly encourages breadth of solutions, the solution provided has to be very easily to customize by both users and developers. User must be able to customize split-screen views for their favorite tools, while developers don’t want to have added another time burden they have to satisfy. The tools that will come with a real world version of our prototype would address both of these problems in a pragmatic way: The split-screen definition for each application would be stored in a modifiable XML-file which can be automatically generated from both *man* and *–help* or manually constructed. This makes an involvement of the developer unnecessary unless specific modifications are required and gives the user a comfortable and verifiable means to alter existing configurations.

Implementation Issues:

Our prototype was developed using the Gnome Toolkit (GTK) and an X windows terminal component in the C programming language.

- ***Container Change:***
  The first hurdle was the dynamically change of the size and content of the container for the split-view. For this we programmatically had to modify the tree structure which contains the container-hierarchy for the windows components depending on the selected program and whether the split-view was requested by the user or not. Our prototype contains the definitions for these views in a hard-coded fashion that is not changeable without modification at the source code level. As mentioned before, this is not our ultimate vision for the project, but rather a forced implementation issue since the complete definition of a XML-schema combined with respective parsing and process capacities would have exceeded our time-limit considerably. With a dynamic configuration-file based view-creation, our prototype would have been already been in close proximity to a final working solution.

- ***Mediation of changes between the two different views***
  To achieve the desired effect of two synchronized views on the same sequence of input characters we employed the model + (view/controller) (MVC) pattern. We group view+controller together in modern GUI applications because the input and representation of data is often too tightly coupled to encourage more fine grained subdivision.

The model holds the current valid input sequence and parses it into separate components. As an example let us start with the following input:

```
tar --directory=~/downloaded -xzvf archive.tar.gz
```

The model parses the line and returns the following interpretation:

- Program-name:
  `tar`
- Long-options:
  Name: `directory` Value: `~/downloaded`
- Short-options:
  `x`, `z`, `v`, `f`
- File-attributes:
  `archive.tar.gz`

Using this information, the model is able to signalize each view to update. The terminal component, in this case, just reassembles the options to re-generate the original input sequence while the split view displays extended information and controls for the program identified by the parsing operation. In our example, this means that the program presents a view for the `tar`-command. Therein the options `x`, `v`, `z` and `f` are checked and the file-attribute `archive.tar.gz` is presented in a textbox (which also allows browsing the file system). The long-option `directory` is not shown in the split-windows since it is categorized as a minor-option which is too specialized to be allowed and would pollute the conciseness of the split-view. One could imagine an extension to this design, where all options are present but grouped into different sub-views of the split-view: The main view could concentrate on the core functionality and display only the three or four most important switches, attributes and usages.

All further synchronization between the two views is relayed over the model part of the program: Each key press of the user in the terminal view and mouse click onto the split-view is continuously monitored and passed on to the model which then updates all views.

3. Usability Specification and Evaluation Plan

The design of our prototype springs from our analysis of users and their tasks. The following usability principles have emerged from this analysis and are useful to evaluate the effectiveness of the design.

- *Predictability and Synthesizability* – The Unix system does a poor job of displaying state. Our system should improve this visibility, both before and after each interaction. Part of this improved visibility should be positive and constructive feedback.

- *Familiarity* – The system should take advantage of the real-world and computer related expertise that they already have, bridging the gap between these current skills to more advanced levels.

- *Generalizability* - One of the core goals of the system is to guide the user from novice to higher levels of expertise. A major component is to encourage the user to build skills that can be transferred to non-aided command line use. The system should provide scaffolding to extend the knowledge of the user, including knowledge in the environment (*man, apropos, info, --help*). It should also provide mappings between abstract concepts and program options. Through these techniques, the system should build confidence and reduce anxiety in the user.

- *Consistency* – Our design prototype should provide a consistent form of input expressions and output responses. The help should be given in a known, consistent manner.

- *Dialog Initiative* – The user should be able to preempt the system at any time. We should greatly restrict, if not eliminate, the system's ability to preempt the user.

- *Multi-threading* – While the command line interface typically supports only one user interaction at a time, we should be alert for opportunities that would support multiple tasks. This might include allowing multiple instances of the system to remain visible throughout parallel operations. While not technically parallel processing, it may guide the user to take advantage of Unix's powerful capability to chain multiple commands together.

- *Task Migratability* – The system should slowly fade support as the user internalizes more advanced knowledge of the command line.

- *Substitutivity* – We should consider allowing users to make semantic requests instead of syntactically correct Unix commands. Examples include allowing the user to type one of a variety of commands to access help (perhaps *help, man, info, apropos*) and be given their choice of the most appropriate assistance. If

implemented, the list of synonyms to access help (and other supported tasks) would come from user research.

- *Customizability* – The system should be expandable and flexible. Users should be able to easily customize the system to their needs and update the system with new task capabilities.

- *Observability* – In addition to the desire for visibility mentioned earlier, we should make as much of the process visible and persistent. Defaults should be chosen wisely, based on user needs.

- *Recoverability* – If possible, we should allow users to undo their actions.

- *Responsiveness* – While we expect to have better response time than most graphical programs, we should design the system to communicate as rapidly as possible with the user. The system should also consume few system resource (lightweight) and be very stable.

- *Task conformance* – The program should provide support for the tasks the user wishes to perform. It should also include additional tasks that the user may not be aware of, but based on expert analysis, may still require. The overall effect of the system should be to decrease the error rate and increase speed to complete tasks.

To measure the success of our prototype against our usability criteria, we have designed an experiment that includes both quantitative and qualitative measurements. Section 3A is an outline of the experiment. Section 3B is a copy of the task sheet we gave to each participant. Section 3C is a worksheet we used to gather data as each participant worked through the tasks. Section 3D is the survey we administered immediately following the experiment.

Section 3A – Outline of the Experiment

For the experiment, we conducted benchmark tasks to gather quantitative data and followed each evaluation with a survey (Section 3D) to gather more qualitative data. Participants were given a script of tasks (Section 3B) that required a file to be uncompressed using the *tar* command, the creation of a symbolic link using the *ln* command and then recompressed using the *tar* command. While the participant worked through the script, we counted the number of errors and the time to complete the tasks. We used the worksheet shown in Section 3C to assist us.

Hypothesis: Participants will more quickly complete tasks and make fewer errors using our GUI augmented command line terminal.

Null Hypothesis: There will be no difference between the results from the normal command line and our GUI augmented command line terminal.

Experimental Design: Within subjects. With each new participant, we reversed the order in which they used the normal and augmented terminals to decrease learning effects.

Location: Undergraduate computer labs. This location was chosen because it is the target environment for real-world use of our system.

Independent variable: Interface feature (normal Unix command line terminal vs. our GUI augmented command line terminal).

Dependent variables:

- time to complete script,
- number of errors made while completing script

Participants: We sought out undergraduate computer science students. These students were the exact target population for which we designed the system.

Prior to the study, we:

- explained the importance of the research
- explained that participation was voluntary
- filled out the IRB approved consent form
- made sure each participant was comfortable
- explained that we expected the session to take less than 20 minutes
- explained how errors or failures were not the participant's problem, but are instead, places where the interface needs to be improved

During the study, we

- maintained a relaxed atmosphere
- never indicated displeasure or anger
- gathered data using the form in section 3C
- were in the same room and directly observed the participants
- used single person sessions
- stayed detached
- asked participants to "think aloud," to gain a better appreciation of their thought process

After the study, we

- used the questionnaire shown in Section 3D to gather more data to cover the range of our usability criteria
- thanked participants
- showed them how to complete missed tasks
- reiterated the importance of the research
- maintained privacy of participants and data
- anonymously stored the data

We believe this plan was appropriate because it utilized research study best practices as well as provided data on the usability of our interface against the original command line interface. The demographics of our participants were the same as our target users. The benchmarks provided hard quantitative data that allowed us to directly measure the effectiveness of our prototype against the original interface. The questions on the questionnaire supplemented this information by covering the breadth of our usability principles. More in-depth analysis can be found in section 4.

Skinning the Commandline

Task Description: You have created an archive (hw4.tar.gz) of a recent homework, but you forgot to rename your Makefile. Uncompress the archive and create a symbolic link named Makefile targeted at the existing file Makefile.dloptest then re-compress the archive using BZip2 instead of GZip compression.

Steps:

(1) Uncompress Archive (use 'tar').

(2) Change directories to hw4.

(3) Create a symbolic link named Makefile targeting Makefile.dloptest (use 'ln').

(4) Change directories back up (..).

(5) Compress the entire directory hw4 into a file named hw4.tar.bz2 using BZip2 compression (use 'tar').

Note: In all compression/decompression operations, be sure to verify file integrity and maintain directory structure.

Section 3C – Experiment Worksheet

Participant Number: _____ Date _____ Time: _____

Major: Computer Science Computer Engineering Electrical Engineering
Other:_____

Class Year:

Personal Evaluation of Unix Command Line expertise: Beginner 1 2 3 4 5 6 Expert

Task 1 – *tar*	
Time to complete task	
Number of Errors	
Notes:	

Task 2 – *ln*	
Time to complete task	
Number of Errors	
Notes:	

Section 3D – Post Experiment Survey

Participant Number _____ Date: _____ Time: _____

I received adequate feedback on the success of each command (tar & ln).	Strongly Disagree <1 2 3 4 5 6 > Strongly Agree
I received more feedback than I would normally receive from the Unix command line.	Strongly Disagree <1 2 3 4 5 6 > Strongly Agree
I found the graphical user interface easy to use.	Strongly Disagree <1 2 3 4 5 6 > Strongly Agree
I could execute the same commands again without the program's help.	Strongly Disagree <1 2 3 4 5 6 > Strongly Agree
For the two commands tested (*tar* and *ln),* I would prefer using the traditional Unix help commands (*man, apropos, info, --help*) instead of the program.	Strongly Disagree <1 2 3 4 5 6 > Strongly Agree
I found the program confusing.	Strongly Disagree <1 2 3 4 5 6 > Strongly Agree
The program ran smoothly and did not crash or lock up.	Strongly Disagree <1 2 3 4 5 6 > Strongly Agree
The capabilities provided by the GUI for the *tar* command were what I need to meet my everyday needs.	Strongly Disagree <1 2 3 4 5 6 > Strongly Agree
The capabilities provided by the GUI for the *ln* command were what I need to meet my everyday needs.	Strongly Disagree <1 2 3 4 5 6 > Strongly Agree
The program was very responsive and quick.	Strongly Disagree <1 2 3 4 5 6 > Strongly Agree
The GUI's preset defaults will speed-up my common tasks.	Strongly Disagree <1 2 3 4 5 6 > Strongly Agree
I was able to easily exit the program.	Strongly Disagree <1 2 3 4 5 6 > Strongly Agree
I would like additional ways to start the program, beyond the keystroke combination I used in the experiment.	Strongly Disagree <1 2 3 4 5 6 > Strongly Agree

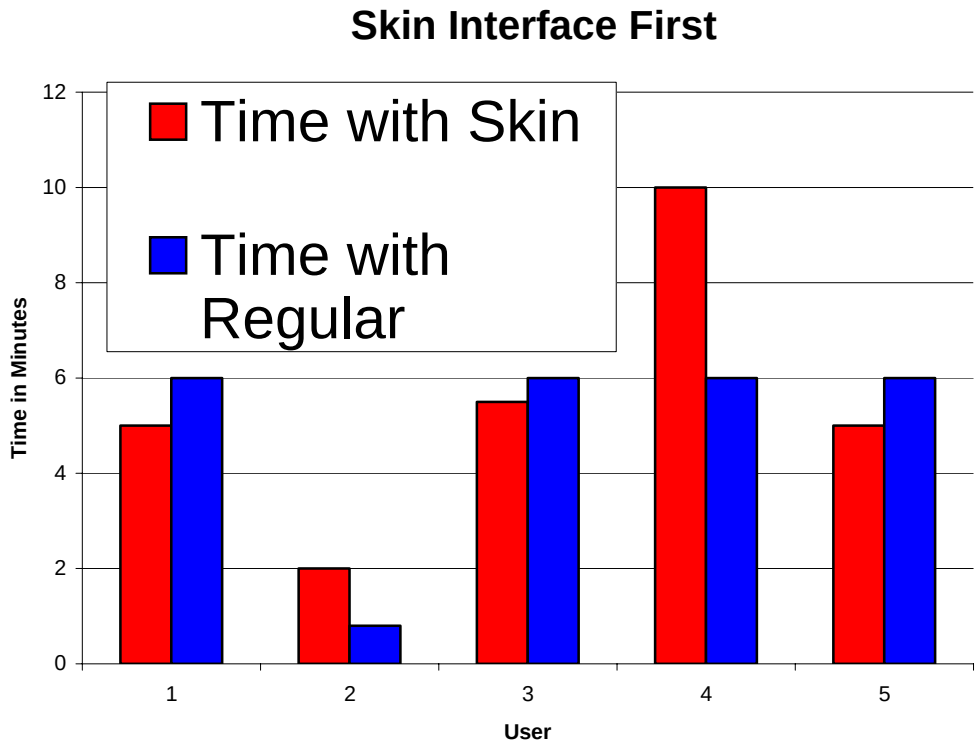
4. Results

Introduction

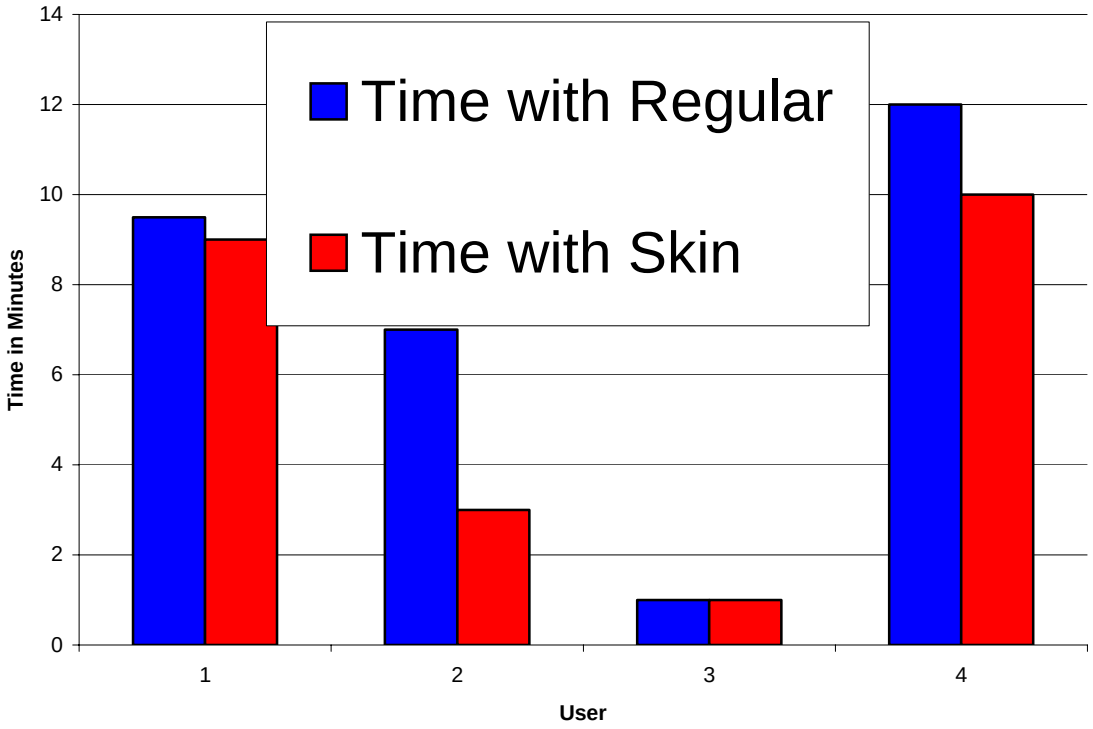
In this final part of the project, our group conducted an evaluation of the prototype developed during the last stage. Section 3 described the design rationale for the evaluation task, our materials and the evaluation measures we employed. In this section, we use these evaluation measures to gauge the success of our design. This section furthers the discussion of evaluation techniques, tasks and users that we began in Section 3.

Study Overview

The experiment was conducted in a common college of computing computer lab using undergraduate computer science and electrical/computer engineering students. Each volunteer read and consented to the research documented on our approved IRB research proposal. They each then completed a task related to the processing of a "homework deliverable." This task was described with a narrative explaining motive and overview as well as with step-by-step instructions that included hints. Section 3B shows the task description. After completing the task using both the standard gnome-terminal command line interface and our prototype skin command line interface, each volunteer answered a questionnaire (Section 3D).



Regular Interface



Half of the research subjects (5) were first given our prototype skin interface to use to complete the assigned task before attempting to do the task again using a regular terminal (see the "Skin Interface First" Chart). This order served two purposes. The first was to compare the effectiveness of our design against the traditional command line interface. The second purpose was to gain some insight into the degree our program "taught" the user how to execute the command from the command line (one of our design goals). We projected that participants would show improvement after the switch. Such an improvement would likely signify that users learned something from our prototype that could carry over to using the regular command line interface. We also projected that there would also relatively large speed-up of task completion when users utilized our GUI. This proved to be only partially correct. We found it difficult to measure the learning effect of our program without conducting additional experiments. A single run through our program did not appear to be enough to allow the student to "learn" the command line. The results in the chart show that indeed two of the five participants ran the second trial using the standard command line interface faster than our prototype interface. We attribute this primarily to the user's existing knowledge of the command line. More interestingly, the differences between the time required to do the trial using the standard interface after using the prototype was significantly less than the time required to do the same trial using the standard interface before using our prototype. By using the second half (5) of the participants (see the "Regular Interface" chart) as a control we determined that our prototype did give novice users an improvement in speed when asked to repeat a task using the standard command line interface. We are optimistic that our prototype design was effective, but feel that additional study is required to gather statistically valid results. Overall, we feel our results justify our experiments and our prototype design.

After the participants did the experiment we administered a post experiment questionnaire (methodology described in section 3). This questionnaire was approved by IRB and inquired about the subjective impressions of the participants concerning our prototype. All questions allowed answers ranging from one meaning "Strongly Disagree" to six meaning "Strongly Agree". The following table gives the average responses along with the standard deviation:

Number	***Question***	***Average***	***Std. Deviation***
1	I received adequate feedback on the success of each command (tar & ln).	4,50	1,18
2	I received more feedback than I would normally receive from the Unix command line.	4,50	0,71
~~***3***~~	~~I could more easily find the files required for each command than with the Unix command line.~~	~~3,00~~	~~1,41~~
4	I found the graphical user interface easy to use.	5,30	0,67
5	I could execute the same commands again without the program's help.	3,60	1,71
6	For the two commands tested (*tar* and *ln),* I would prefer using the traditional Unix help commands (*man, apropos, info, --help*) instead of the program.	2,30	1,34
7	I found the program confusing.	1,80	0,79
8	The program ran smoothly and did not crash or lock up.	3,20	1,81
9	The capabilities provided by the GUI for the *tar* command were what I need to meet my everyday needs.	5,00	0,82
10	The capabilities provided by the GUI for the *ln* command were what I need to meet my everyday needs.	5,00	0,87
11	The program was very responsive and quick.	5,40	0,70
12	The GUI's preset defaults will speed-up my common tasks.	4,13	1,46
13	I was able to easily exit the program.	4,20	1,75
14	I would like additional ways to start the program, beyond the keystroke combination I used in the experiment.	3,90	1,97

It should be noted that question #3 was not administered to all the participants because we didn't have the respective functionality implemented in our prototype.

Each question was chosen to evaluate one or more of our design criteria. After reviewing the survey results we interpreted the survey as follows.

The program ran smoothly and did not crash or lock up.	Fair
The capabilities provided by the GUI for the *tar* command were what I need to meet my everyday needs.	Excellent
The capabilities provided by the GUI for the *ln* command were what I need to meet my everyday needs.	Excellent
The program was very responsive and quick.	Excellent
The GUI's preset defaults will speed-up my common tasks.	Very Good
I was able to easily exit the program.	Very Good
I would like additional ways to start the program, beyond the keystroke combination I used in the experiment.	Very Good

I received adequate feedback on the success of each command (tar & ln).	Very Good
I received more feedback than I would normally receive from the Unix command line.	Very Good
I found the graphical user interface easy to use.	Excellent
I could execute the same commands again without the program's help.	Fair
For the two commands tested (*tar* and *ln)*, I would prefer using the traditional Unix help commands (*man*, *apropos*, *info*, *--help*) instead of the program.	Excellent
I found the program confusing.	Excellent

After further analysis against our design criteria and consolidating our results we determined that :

- Feedback Very Good
- GUI Excellent
- Learning Fair
- Functionality Excellent
- Ease of Use Excellent
- Stability Fair

We hereby can infer that our prototype shows promising capabilities for enhancing the command-line experience of the student user. The graphical user interface is easy to use and complies with the general functional task-set the user-population wants to perform. The prototypical stage of the development prevented us from getting better ratings in stability but future versions should be able to alleviate these problems.

The only really critical point that we could deduce from the questionnaire was the fair ranking for the learning aspect of the prototype. This is a major failure for our design that hoped to promote transfer between the GUI-world and the traditional command-line. We are unsure what caused this problem. Possible reasons include the one-time-usage of the prototype, the mode of operation that caused the participants to focus on completion speed and not on learning and the missing of an explicit scaffold that make the connection between button and checkboxes and actual command-line parameters more visible.

Improvements

We did not specifically ask our participants to suggest improvements but several hinted on extension possibilities. Most common was the usage of browse-buttons for file-selection, something we already had considered but which felt prey to the time-constrains of the prototype development. Another participant suggested adding a status-bar as being always visible which mediated between the two extremes of having the GUI-interface taking up half the screen-estate and not having it at all. Inspired by the different perspective view on the command another participant got the idea to display the files available in the current directory as sort of an explorer-view while no command was entered. This would eliminate the need to do the obvious ls-command each time a directory is entered while still keeping the actual command-line "calm" and ready for typing.

Given these suggestions and our own analysis, we consider the following to be appropriate ways to improve our design:

- Move from our hard coded, closed design to an open architecture to that would allow users to build and share GUI templates.
- Improve stability
- Put more focus on the layout of widgets in the GUI window.
- Continue testing on a larger pool of undergraduate computer science students, perhaps in conjunction with an applicable 200 level introduction to computer science course.
- Explore ways to dynamically generate GUI templates from existing help in the Unix environment, perhaps by parsing the *man* files.