Re-imagining the Command Line User Experience for Problem Solving

Rachel B. Cabot

Bachelor of Science in Computer Science with Honours
The University of Bath
May 2017

Re-imagining the Command Line User Experience for Problem Solving

Submitted by: Rachel B. Cabot

Abstract

Though they appear to be arcane and outdated tools by modern standards, traditional command line interfaces (CLIs) still find heavy use by sysadmins, software developers and power users. This is largely due to the fact that for many of these users, graphical user interfaces (GUIs) are not often designed to scale to the functionality and control requirements of modern software systems.

While CLIs can support powerful and efficient action, it is clear that they challenge users in many ways. In examining deficiencies with CLIs, we can observe a principle problem which affects the extent to which CLI users are able to effectively and efficiently accomplish their goals, and, in particular perform action specification. This problem is characterised by a need for users to engage in exploratory activities in order to successfully execute valid commands.

The question explored within this dissertation is how this problem of exploration may be addressed, and in doing so, determine how CLIs can better support the solution of novel problems. This is explored through several studies involving expert CLI users, and deriving, testing and evaluating a design for a CLI with usability-enhancing features.

Users were found to react well to command suggestion mechanisms, achieving faster task success and engaging in less documentation checking. However, their inclusion often lead to a less engaging experience for users, and there are still challenges for the integration of these in real-world CLI software systems. As a consequence of these findings and others, we are able to arrive at an informed theoretical model of CLI user action specification, which might be used to better understand how the experience of CLI users might be improved.

Contents

List of Figures

List of Tables

Acknowledgements

I wish to express my sincerest gratitude to my supervisor Dr Leon Watts for his continued support and interest in my undergraduate dissertation. Besides my supervisor, I would like to thank other lecturers and research fellows, including Dr Stephen Payne and Ken Cameron, for their insightful comments and encouragement.

In addition, I would like to thank the participants of my studies for their cooperation, without whom these findings would not have been possible. My thanks also extend to my friends and family for helping with the daunting task of proof-reading.

Services such as PopupArchive, the ACM Digital Library and Github were incredibly valuable resources during my research, and I would not have been as effective without them. Special thanks to Mikhail Yurasov, for open-source provision of the frontend boilerplate used for integrating the artefacts in this dissertation.

Glossary

accessibility The extent to which something can be useful to people of any background, ability or expertise. 97, 144

action specification The process a user undergoes when they are determining how to affect their intentions upon the environment. 10, 98, 130–132, 135, 138, 139, 141

activity theory An umbrella term for a particular line of social science theories and research. 59

affordance Something that you can do with a particular object. 39, 45, 125, 138

agency The ability to make decisions. 35, 64, 75, 85, 86, 127, 138, 145

aliasing The provisioning of multiple labels for a command or referent. 73

answer service A question answering system; a system which automatically answers questions posed by humans in a natural language. 21, 30

CLI Command Line Interface; an interface which enables a user to issue commands in the form of symbolic input; an interaction paradigm based on a control language. 1, 4, 5, 7, 9, 10, 28, 29, 32–34, 36, 37, 39, 40, 44, 47–49, 51, 52, 54–56, 59, 61, 63, 64, 67–70, 72–74, 76, 79, 84, 92, 94–98, 103, 107, 124–127, 131, 132, 135–139, 141–146

cognitive model An approximation of human cognitive processes for the purposes of comprehension and prediction. 36

command A set of referent labels, structured in some way by the syntax of a control language, which expresses a particular intention of a user. 9, 10, 12, 21, 27, 28, 30, 37–39, 41–45, 48, 49, 51–62, 65, 70, 73–79, 81–90, 94, 95, 97, 98, 106, 108–110, 115, 117, 118, 121–123, 125, 127, 130, 133, 134, 136–138, 141, 143, 164

conceptualise To form a mental model of some domain. 38, 137

connective syntax Symbols within a control language which allow the structuring of many labels and domain objects in a single command (eg pipes, flags). 132, 134

control The extent to which an agent is able to affect their environment. 11, 27, 36, 37, 57, 64, 69, 74, 79, 82–86, 104, 105, 120, 127, 130, 134, 138, 141, 146

control language A set of labels for domain referents and a connective syntax. 49, 67–69, 130, 136, 137, 141, 145

customisation Alteration of an artefact in order to better support a user's aims or preferences. 52, 64, 73, 144

digraph Directed graph; a set of nodes and directional links between them. 31

direct manipulation Physical, incremental operations upon a continuous representation of an object of interest. 26

discovery The act of finding some artefact or thing pertinent to the success of an activity. 44, 54, 55, 58, 74, 75, 109, 118, 121, 122, 125, 130–133, 135, 136

DMI An interface which permits direct manipulation. 26

domain A specific sphere of work or activity. 27, 29, 35, 38, 45, 59, 60, 63, 64, 68–75, 90, 95–98, 103, 108, 109, 115, 117, 119, 122, 124, 125, 130, 131, 136–138, 141, 143

domain context pattern recognition The classification of, by machine learning processes subject to big data, different areas of use within a domain. 143

domain object A pertinent object of a domain of work (eg file, folder). 130

effective Successful in the context of an activity. 28, 34, 36, 40, 60, 62, 96, 127, 136–138, 141, 143–145

efficient Performed rapidly and with low effort. 41, 54, 64, 68, 78, 89, 97, 130, 135

emergent Unintended in a fashion which was not priorly predicted. 137

end-user programming Refers to activities and tools that allow end-users - people who are not professional software developers - to program computers. 28, 40

engagement The extent to which a person mentally participates in an activity. 124, 127, 138

error state A state which arises due to some incorrect action, and prohibits or hinders progress towards a favourable outcome in the context of some activity. 78, 89

expert A person with sufficient knowledge to effectively work with a particular artefact. 4, 38, 48, 51, 53, 59, 63–65, 72, 73, 81, 84, 141, 144, 145

exploration The process of searching an environment with the intent to discover something. 31, 45, 64, 81, 95, 131–134

Exploration Problem The problem of conducting efficient explorative activities during CLI action specification. 47, 55, 65, 68–70, 78, 94, 130, 131, 135, 139, 141

exploratory activity Some process that is undertaken with the intention of discovering something within an environment. 134, 141

flag An option which modifies the behaviour of a command. 58, 131

flow A state in which a user is able to fluently and effortlessly translate their intentions into action. 40, 53, 68, 73, 77, 79, 82, 84, 89, 127, 135

groupware Software artefacts designed to facilitate collective working by a number of different users. 136, 144

guessability How easily users are able to propose a correct speculation for something. 28, 59, 69, 125, 136

GUI Graphical User Interface; An interface which permits direct manipulation through graphical means. 1, 26, 30

Gulf of Execution The gap between a user's goal for action and the means to execute that goal. 54

heuristic A rule of thumb; a standard method which is deployed, but not guaranteed to work. 21, 34, 36

highly interactive Provides a user with a tight feedback loop. 144

innate Natural, or overlearnt as to feel natural. 135

interactive Allowing a two-way flow of information between a computer and a computer-user; responding to a users input. 9, 21, 30, 34, 39–41, 84, 85, 130

interface An artefact enabling a user to communicate with a computer. 10–12, 21, 29, 33, 43, 68–70, 89, 90, 94, 98, 105, 110, 112–116, 120, 123, 126, 130, 138, 145, 162–164

knowledge-primed Subject to prior knowledge or well-known experiences of a particular user. 135

label A symbol that is input by a user in order to access a particular referent. 131, 136, 137

look and feel The visual design and dynamic behaviour of elements within a user interface. 145

mainstream product Artefacts that are used by most people and regarded as normal or conventional. 144

meaningfulness How well a particular thing fits within a person's preconceived mental model of a particular system. 137

memorability The extent to which a thing lends itself to begin remembered by a person. 58, 59, 69, 137

mental model A person's internal representation of some system, by which they perceive consequences of their actions within the system. 60, 137, 138

natural language A language that has developed naturally in use (as contrasted with an artificial language or computer code). 30, 38, 138

NDM Naturalistic decision making; decision making in the context of dynamic and complex real-world scenarios. 33, 34

needs-driven Not recreational; only on the basis of working activity. 137

operational documentation Documentation which provides a user with information on how to operate an interface. 71, 89, 91, 94, 99, 108, 117–119, 121, 123, 133

package management system A system which automates the process of installing, upgrading, configuring, and removing computer programs. 133

parameter A domain object upon which a particular function operates. 58, 59, 69, 132, 137

peer A person working within the same domain of activity as a particular user. 84, 133, 134, 136, 137, 141, 145

power user A computer user who uses advanced features of computer hardware, operating systems, programs, or web sites which are not used by the average user. 1

powerful Affecting large change with low effort. 1, 21, 28, 41, 56–58, 63, 64, 68

problem solving The process of manipulating an environment into a desirable state in the context of some activity. 24, 31–36, 38, 40, 44, 47, 52, 60, 68, 70, 75, 79, 96, 127, 145

problem space A set of states that a problem domain can be in, connected by domain actions which transform the state of the domain. 31, 32, 35, 37, 38, 40, 96

purely functional programming A programming paradigm that does not rely on mutable state, and treats all computation as the evaluation of mathematical functions. 146

realtime Relating to a system in which input data is processed within milliseconds so that it is available virtually immediately as feedback to the process from which it is coming. 9, 33, 37, 38, 138

recognition-primed Automatically produced on the basis of classification of a specific situation. 54, 130, 135

recovery The act of returning to a normal state from an erroneous state. 41, 61, 70, 137

referent A unit of functionality specifically referred to by a label. 30, 55, 58, 97, 98, 130, 133, 136, 141

routine task A task which can be solved exclusively on the basis of recognition-primed decisions. 11, 102, 135

RPD Recognition-primed decision; reacting on the basis of prior experience to a situation. 33, 39, 40, 54

scriptability The extent to which an artefact enables end-user programming. 21

shell A user interface for access to an operating system's services. 29, 142

signifier Some indication of affordance. 138

suggestion An idea or plan put forward for consideration. 9, 34, 45, 49, 72, 75, 77, 79, 88, 95, 108, 126, 136, 138, 143

symbol Some unit representation of an element of a control language. 27

syntax corpus A representation of the operatonal details of a particular control language. 11, 98, 99, 143

sysadmin Systems administrator; a person who is responsible for the upkeep, configuration, and reliable operation of computer systems. 1

terminal emulator A program that emulates a video terminal within some other display architecture. 142

traditional CLI Shells or terminal emulators, in contrast with search engines or answer services. 33, 92, 119

trial and error The process of experimenting with various methods of doing something until one finds the most successful. 55, 59, 69, 134, 136

understandable Can be readily assimilated into knowledge. 138

Unix Philosophy A philosophical approach to minimalist, modular software development. 36, 37, 57, 146

usability The ease of use and learnability of an artefact. 21, 24, 28, 29, 36, 63, 67, 68, 85, 94, 141, 142

vanilla Unmodified; not subject to end-user customisation. 51, 65

viewmodel Component of the MVVM (model-view-viewmodel) software architecture which links parts of an underlying model to view components. 90

websocket A computer communications protocol, providing full-duplex communication channels over a single TCP connection. 90

wildcard A character that will match any character or sequence of characters in a search. 9, 40–42, 131

Chapter 1

Introduction

Graphical user interfaces (GUIs) are often considered the de-facto standard for establishing an interactive link between a user and a computer system. However, as [Norman, 2007] points out, command line interface (CLI) paradigms have achieved mainstream ubiquity in the form of "answer service" search engines. It would appear that this search-oriented paradigm has been well explored in real world software products and mainstream usage, but this presents a limited worldview on the possibilities that the CLI paradigm provides. In particular, the benefits that power users can derive from command lines are often forgotten in modern software systems. In addition to this, usability of CLIs has received very little contemporary research attention [Heron, 2015].

As [Norman, 2007] puts it, GUIs fail to scale to the demands of modern software systems. Indeed, their inflexibility limits the user's ability to solve novel problems or scale to large functionality sets [Groth and Gil, 2009]. Aspects of CLIs afford the advantage of more "efficient" interactions for advanced users [Murillo and Sánchez, 2014], and expose several powerful utilities. One such feature is the ability to compose unit functionality, whereby user actions can be chained together into complex control structures via connective syntax such as pipes. Another is scriptability — this allows for end-user programming, providing experts with sophisticated tools to automate their work and decrease cognitive and physical effort. By features such as these, CLIs should allow a user to move rapidly towards the solution to a novel problem, where other interface types may incapable of doing so.

But this comes at a cost; learning command syntax represents a significant overhead, as well as the increased demand on user memory and chance for human error [Miller et al., 2008]. It is these disadvantages which are most likely to account for a reluctance to adopt command line style interactions outside of search and answer service paradigms.

It is the intention of this research project to understand how CLIs can better be harnessed as tools for composing a wide range of functionalities to arrive at a solution to a problem. The approach taken will be to investigate what users of CLIs value in their interactions under circumstances which require them to solve novel problems, and what deficiencies exist that prevent them from being effective. Through this, it may be possible to discover new design heuristics and descriptive models of users' experiences within CLIs. In deriving these, we may enable usability improvements to existing CLIs, and encourage the integration of powerful features of CLIs within mainstream software products.

Chapter 2 presents a review of prior work on alternative types of user interface in relation to human problem-solving strategies. In this, we find that exploratory action is fundamental for all human-computer interactions, and so users of CLIs would benefit from better support for command discovery and reflection on the effects of prior command actions.

Chapter 3 builds on this by grounding these general concepts in the examination of the experience of a set of expert CLI users. This is done by interviewing a group of expert CLI users, and analysing transcripts of these via thematic analysis. In this, we discover what features these users value, what effects their experience, and what kinds of problems CLIs are particularly suited to. In particular, we find that the effectiveness of CLI use is limited by the extent to which a user's intent can be easily and rapidly specified as a command.

Chapter 4 summarises these notions and derives a specification which captures desirable qualities of an enhanced CLI artefact for problem solving. Several designs are proposed, which address the design problems outlined by proposing features such as artificial intelligence, suggestion mechanisms and information visualisation. These are evaluated by expert users, and the best candidate interface is implemented. The selected interface provides context-sensitive command suggestions, a visualisation method for undo states, and continuous representations of sections from operational documentation.

Chapter 5 proposes an evaluative study to investigate the effectiveness of the enhanced CLI design which was implemented. This is done by designing an experiment domain and a set of tasks — constituting a text-adventure game — which attempts to accurately model real-world use of CLIs and command shells in a controlled fashion. By comparing the use of the enhanced CLI with that of a CLI without certain enhancing features, we are able to conclude that intelligent suggestion mechanisms for CLIs are a promising design space that deserve further investigation.

Chapter 6 aims to bring together these diverse findings into a formalised specification of the Exploration Problem, and a formal model of action specification of CLI users. We find we are able to identify several kinds of exploratory activity which occur when CLI users are determining how to specify an intent to act as a command.

Chapter 7 summarises the findings as presented in chapter 6 and proposes a wide array of future works suggested by certain aspects explored within the project as a whole.

Chapter 2

Literature Review

In this investigation, we aim to better understand barriers to CLI usage from a Human-Computer Interaction perspective, so that novel CLI design ideas can be identified and tested. In order to investigate how the usability of CLIs might be enhanced for problem solving activities, a review of the relevant literature must initially be performed.

Therefore, in this chapter, we shall first establish a conceptual foundation by examining some key concepts relating to the broader field of Human-Computer Interaction. We will then focus more upon facets of interface types, with the intention of distinguishing CLIs from other sorts of interfaces. With specific considerations to design issues of CLIs, we shall then examine conceptual models of methods and strategies for human problem solving. This will allow us finally to appraise methods which have been priorly used to address the design issues identified under the specific context of problem solving.

From these examinations, a conceptual basis for further investigations shall be achieved, as well as a focus for the specific aims of this dissertation.

2.1 Interaction

Before thoroughly engaging in literature related to CLIs and problem solving, it is pertinent to identify particular fundamental, underpinning theories relating to the nature of human interaction with software artefacts.

2.1.1 Human-Computer Interaction

Human-Computer Interaction (HCI) is a field of study which is concerned with research into the design and use of computer technology, focusing on hardware and software interfaces. Within this dissertation, we define interaction as the process by which a user of a computer system conveys and receives information to and from a computational artefact. This computational artefact might comprise hardware, software, or both.

The degree to which a computational artefact is interactive depends upon how effectively — or indeed, how easily — a user is able to engage in this two-way flow of information with it.

2.1.2 The Human Action Cycle

Donald Norman, in his seminal works within HCI [Norman, 1988], defines many theoretical models which assist the description and structuring of reasoning about interactive artefacts.

One such theory, the Human Action Cycle [Norman, 1988], details how a human might go about engaging in an interactive loop with an artefact. First, it asserts that a person goes through a stage of goal formation, whereby the high-level aims of an overarching activity are determined with respect to the artefact in question. Then, the person must engage in a stage of execution, whereby their goals are translated into a set of tasks required to achieve those goals and specified as a sequence of tasks — this particular process is known as action specification. Once this occurs, this action sequence is then physically executed by the user. Finally, the user must engage in a stage of evaluation, whereby the user physically perceives of their executed actions, interpreting the actual outcomes on the basis of the expected outcomes and then reflectively comparing actual and expected outcomes to determine if the desired effect was achieved.

This theory underpins much of the analysis performed within the investigations presented here.

2.1.3 Affordances and Signifiers

In the context of Norman [1988]'s Human Action Cycle, affordances are defined here as the possibilities for action that a user perceives that they can perform with an artefact [Norman, 1988]. This necessarily depends upon the context in which the artefact is perceived and used — a user may perceive a wider or narrower range of affordances for a given object for different contexts.

Features of an artefact which allow a user to perceive affordances are known as signifiers [Norman, 1988]. An example of such a feature is a handle on a teapot — the handle signifies to the user that the teapot affords grasping or tipping.

2.1.4 Referents and Function

For an interactive artefact to be useful, it must perform a number of functions - units of capability which a user can harness to perform some useful action. Particularly in the case of software artefacts, computational functions require some means by which a user to access them — in the context of CLIs, these are known as referents [Furnas et al., 1987]. Referents are symbolic labels — typically textual — for a piece of unit functionality afforded by a CLI.

2.2 Understanding Interface Types

Very little contemporary literature exists on the study of CLIs, supporting claims that they deserve further investigation. Nevertheless, it is still possible to explore the nature of CLIs through traditional literature, with the aim to derive a working definition of this type of interface. This can be done, in part, by differentiating definitional CLIs from other types of interfaces, such as direct manipulation interfaces (DMIs). Features that typify CLIs can then be explored with respect to the field of problem solving.

2.2.1 Direct Manipulation Interfaces

Direct manipulation interfaces (DMIs), described by Hutchins et al. [1985], are interfaces which enable the direct impression of computation objects by actions of a user in real-time.

That is to say, they enable "direct manipulation", which Hutchins et al. [1985] defines by a number of measures. For an object of interest to be directly manipulated, it must afford "physical actions or labelled button presses", specifically distinct from descriptions of action. These actions constitute rapid, incremental and reversible operations, whose impact on the object of interest is immediately visible via some continuous representation.

It is clear that these types of interfaces are highly interactive, allowing a user to respond quickly to change, and promote a congruence of physical affordance — that is, controls within this paradigm often present analogies with physical real-world objects and controls, which readily suggest possible actions.

Graphical user interfaces (GUIs) are a sort of DMI which manifests interaction through visual means. The distinction is important when considering DMIs for blind or vision impaired users, which operate in tactile or sonic mediums. A common implementation of GUI is the Windows, Icons, Menus, Pointer (WIMP) paradigm, whereby windows define action areas for particular applications, icons are used to represent computation entities, and mouse-driven pointers create an impression of physical action on those entities.

Advantages

Shneiderman [1983] describes several virtues of DMIs as a paradigm for establishing an interactive link with a user. Novices are able to learn basic functionality quickly via demonstration, while experts can work rapidly to carry out a wide range of tasks. Operational concepts can also be retained and transferred to other DMIs.

In addition, error messages are rarely needed, because users can immediately see if their actions further their goals, allowing them to change the direction of their activities if not. As a consequence of this, users have "reduced anxiety" as the system is comprehensible and actions are easily reversible.

However, it can be argued that these advantages do not universally apply, especially in practical implementations.

Disadvantages

Hutchins et al. [1985] explored many disadvantages of DMIs. These included the fact that "DMIs have difficulty handling variables, or distinguishing the depiction of an individual element from a representation of a set or class of elements". Repetition of tasks, in particular, is poorly supported, because it can be difficult to continuously chain together fine motor operations for a large set of graphical objects.[1]

DMIs also have problems with accuracy — responsibilities of accurate action are often placed on the user, where they might be better handled through the intelligence of a system or communicated symbolically. As a consequence, DMIs must directly support how users think about a domain, and this can sometimes be difficult to achieve. Directly supporting a user domain restricts how it is possible to think about the problem, missing the potential for new technology to provide new ways to interact with a domain. Hutchins et al. [1985] particularly emphasises that DMIs "give up" on conversational models, and thus give up on dealing in descriptions of the domain — this is particularly detrimental for contexts or domains which require fine control over specification of action.[2]

2.2.2 Command Line Interfaces

As distinct from DMIs, command line interfaces (CLIs) are interfaces which enable a user to issue commands in the form of successive lines of text. Thus, they are interfaces which are transactional in nature, requiring full specification of operation, target and constraints before initiation.

More broadly, as Norman [2007] puts it, they are "interaction paradigms based on a control language", where users enter commands and their respective arguments. This definition is proposed with the intention to draw together diverse sorts of CLIs, including contemporary uses of aspects of CLIs within mainstream products.

Where DMIs enable users to explicitly act upon objects in a domain, CLIs allow a user to be descriptive about their intentions to act within that domain, prior to committing those intentions to action.

[1]I.e. clicks, drags and drops.

[2]For example, system administration or software deployment.

Advantages

CLIs have been described by some sources [Kampe] as having several virtues. Typically, they require few resources, or often work well in minimal environments — that is, fewer resources to run than graphical interfaces [Kampe]. This contributes to the fact that they are expert friendly — that is, they present a better user experience to those that are more experienced. This is because many sysadmins and software developers must often use CLIs such as terminal emulators to operate server technologies.

Related to this, they are concise and powerful in their function; that is to say, they can service a huge array of functionality in a minimal fashion. This power is further increased by permitting automation via scripting and end-user programming. CLIs are often described as capable of enabling experts to quickly achieve their goals [Murillo and Sánchez, 2014]. This is because CLIs excel in areas where expressive power is required; very little often needs to be written to do a huge manual task.

In addition, most CLI interactions are keyboard-only — many sources [Omanson et al., 2010][Lane et al., 2005] stipulate the efficiency of keyboard interactions over mouse interactions. However, it is clear that this heavily depends on the context of use.

Disadvantages

Many disadvantages of command lines have also been described by sources such as Kampe. A principal disadvantage is that they require intense cognitive effort from a user, requiring the understanding of highly abstract concepts to enable effective use, as well as memorisation of some control language.

Their minimal nature brings another disadvantage. CLIs are not typically visually rich, which can lead to a disengaging experience, and, in particular, a lack of signification of afforded commands. In particular, Hewett [2005] discusses how long-term memory requires cues for effective retrieval. The relative minimalism of CLIs is unlikely to permit retrieval cues — this typically hampers usability in terms of memorability.

They are also cited as having very low learnability because they are often designed with poor guessability [Wobbrock et al., 2005]. Typically, new users are required to trawl through tutorials, manual pages, help guides and forum posts in order to perform even the simplest of tasks.

Taken together, these reasons seem to demonstrate why CLIs are seen as intimidating, beginner-unfriendly, and having a high barrier to entry for novices.

2.2.3 Distinctions

While this section does set out to expose a contrast between DMIs and CLIs, many real implementations of these paradigms actually exist on a continuum between the two.

The aims of this study constitute a desire to explore interfaces which at least incorporate some aspects of CLIs in pursuit of their advantages.

Though the interfaces explored may hybridise aspects of DMIs and CLIs, the focus is nevertheless on the enhancement of the CLI paradigm. Such interfaces should, however, promote input of symbolic commands as the dominant method of interacting with a problem environment if they are to be considered within the CLI paradigm.

To provide a core definition for the purposes of this dissertation, a CLI will be treated as any interface based on the formulation and composition of linguistic statements intended to result in the manipulation of objects in a problem environment. Manipulation in this sense simply refers to moving, reading or writing data, rather than notions which are suggested by direct manipulation.

2.2.4 Types of CLI

It is clear from examinations of definitive descriptions of CLIs that further clarification must be provided upon the variety of different interface technologies that are classified as CLIs. Thus, it is pertinent at this juncture to define different sorts of CLI with respect to the examination of example CLI artefacts that exist.

Traditional CLIs

We define definitionally traditional CLIs here as those by which users solely interact with a domain on the basis of successive lines of textual input. This definition sets out to encompass shell environments and terminal emulators such as Bash [GNU], Powershell [Microsoft] and Terminal [Apple, b].

Enhanced CLIs

We define enhanced CLIs as traditional CLIs which have integrated usability enhancements. This definition sets out to include shell environments such as Fish [Doras] and Zsh [Falstad], which provide enhancing features — those intended to enhance support for user action and comprehension — such as autosuggestions [Doras][de Arruda, 2013].

GUI Integrations

We define GUI integrations of CLIs as those whose main methods of interaction are via graphical direct manipulations, but may, on occasion, permit interaction via descriptive input. This definition mostly encompasses the use of search prompts within GUI applications for location of certain referents — a pertinent example of this is the "Spotlight Search" mechanism within Apple's MacOS [Apple, a].

Hybrid Interfaces

We define hybrid interfaces as those which readily provide multiple methods of interaction which, categorically, are of CLIs as well as GUIs. This definition includes interfaces such as AutoCAD [AutoDesk], which allows operation via a built-in command prompt or through interactive drawing tools.

Answer Services

We define answer services, or question answering services, as systems which provide an information retrieval service by performing natural language processing upon natural language queries. This definition includes the answer service functionalities within search engines such as Google, and intelligent personal assistant software such as Apple's Siri.

2.2.5 Key Interface Concepts for Reconsidering CLI Design

It is clear that, from examining the deficits of the CLI paradigm and the way CLI-style interactions are integrated in real-world software systems, there are a few key interface concepts which may need to be addressed within the design of a CLI.

One aspect is the fact that CLIs, by their nature, lack signifiers of an afforded space of commands. It seems that, for one to address this, the often combinatorial space of possible commands must be narrowed by context and presented to a user by some means, without compromising desired minimality.

Another is the ability to visualise and harness a history of action. Users are more likely to be able to effectively recover from states of error, or better understand the consequences of their actions, if a history of their action is better visualised. In addition, the ability to in some way exploit or harness action history items to lessen physical or cognitive effort appears to be a fruitful design avenue.

2.3 Models of Problem Solving

The CLI design challenge warrants discussion of general human problem solving, to better understand what users might do with the information they are furnished within the context of CLI use. We define problem solving here as the process of manipulating an environment into a desirable state in the context of some activity. This definition also sets out to encompass the activities which are necessary for a user to plan or understand how to solve their problem.

We shall first consider rational models, before drawing on naturalistic models to consider circumstances and environment as a significant factor in CLI interactions.

2.3.1 Rational Problem Solving

The works of Simon and Newell [1971] are set out with the intention to explain and generate a theory of human problem solving, such that computer programs could be created to help humans solve problems. The shape of their theory could be captured as such:

- Some characteristics of a human information processing system are “invariant over task and problem solver”.
- These characteristics permit the representation of the task environment as a “problem space” in which all problem solving occurs.
- The structure of the task environment determines the possible structures of the problem space.
- The structure of a problem space determines the possible programs that can be used for problem solving.

It was their proposition that the act of problem solving was the deliberate process of searching a proposed “problem space” [Simon and Newell, 1971] to discover a path to a desirable outcome.

In lieu of this theory of problem solving, a General Problem Solver (GPS) [Newell et al., 1969] was proposed — the intentions of such were to create a computer program which could solve any problem. Problem spaces needed to be expressed formally as a set of “Well Formed Formulae”, which constituted a digraph of axioms and desired outcomes. Any problem formalised this way could in principle be solved by the GPS.

However, search-based problem solving suffers from a combinatorial explosion of the problem space for many real-world applications, making deliberative, rational exploration of a problem space intractable without heuristics or pruning techniques.

Simon and Newell [1971] also neglect to examine certain human elements in their theory of problem solving [Klein, 1995]. The following aspects are assumed by Simon and Newell [1971]'s theory, but are unsuitable when dealing with human actors and real-world environments:

- Rational action of actors — the consistent performance of precisely optimal action, without regard for operational situation.[3]
- Perfect knowledge of the problem space — as a consequence of a completely accurate systemic mental model.
- Infinite capacity for computation — able to complete arbitrarily large computations in a trivial amount of time.
- Calculative rationality — assumed stasis of the problem space while calculation occurs.

It is clear that the antithesis or lack of these aspects may have to be accounted for within CLI problem solving activities. A particular problem here that can be identified is the lack of perfect knowledge over a combinatorial problem space, particularly for CLIs which expose large sets of referents.

2.3.2 Naturalistic Decision Making

It is clear that how human beings make decisions underlies the processes by which they solve problems. Naturalistic Decision Making (NDM) is cited as "an attempt to understand how humans actually make decisions in complex real-world settings" [Zsambok and Klein, 1996]. Works in this field have explored scenarios exhibiting the following features [Zsambok and Klein, 1996]:

- Ill-defined goals and ill-structured tasks
- Uncertainty, ambiguity, and missing data
- Shifting and competing goals
- Dynamic and continually changing conditions
- Action-feedback loops (real-time reactions changed conditions)
- Time stress
- High stakes
- Multiple players
- Organisational goals and norms

[3]Such as fatigue, emotional state, time pressure and social setting.

- Experienced decision makers

Classical approaches to problem solving and decision making, which are analytical or systematic in nature, deteriorate or are unsuitable when these features are present.

It is unclear to what extent that domains operated by CLIs present these sorts of features in the context of problem solving. Indeed, it seems that many of these aspects perhaps apply to interface paradigms that are more oriented to realtime interaction through direct manipulation.

2.3.3 Recognition Primed Decision Models

Klein [1995] defines the Recognition-Primed Decision (RPD) model for rapid decision making. The model works on the assumption that humans act and react on the basis of prior experience — types of "case" are identified and reacted to with plans of action that have typically worked in the past. The implementation of a plan is monitored, to determine flaws, or discover what might go wrong. If a problem is foreseen, the chosen plan is modified or rejected in favour of the next most typically successful plan.

This class of model, in contrast to Simon and Newell [1971]'s works, allows us to understand that problem solving in the real world is interactive. That is, problems in the real world are often solved by iteratively taking incremental action based on decisions made via NDM processes until the problem is solved.

It is unclear to what extent these processes are applicable within the context of CLI interaction — indeed, it seems that the interactive, real-time nature of this sort of decision making process is almost entirely prohibited by traditional CLI interaction. However, the notion of recogition seems to highlight the importance of command output in prompting forthcoming possibilities as well as the effect of the last action — that is, command output may remind users of other courses of action that had not initially occurred to them.

2.3.4 Dual Process Theory

In the context of works on rational problem solving and naturalistic decision making, Zsambok and Klein [1996] proposes that techniques for decision making actually lie on a "decision continuum". On one end lies the conscious, deliberated and highly analytical methods, such as Multi-Attribute Utility Analysis (MAUA) [Albany University, 2003]. At the other end lies non-optimising and compensatory strategies like RPD.

In fact, later works [Evans, 2008] actually prescribe a dual model of decision, where different methods of human decision making work in parallel, known as Dual Process Theory. The suggestion is “that there may be two architecturally (and evolutionarily) distinct cognitive systems underlying these dual-process accounts”[Evans, 2008], one system supporting “slow, deliberative, and conscious” decisions with explicit rationale, and the other supporting “fast, automatic, and unconscious” decisions based on heuristics and pattern recognition based judgements.

It is clear that, in the context of these works, in order to compliment the natural duality of decision making and planning processes used by human beings, CLIs must provide both interactive and descriptive interactions to be effective problem solving tools. Indeed, represented collections of possible commands, as well as of recent command output, could engage automatic recognition and decision processes, as well as supporting deliberative reflection on the status of a user’s work.

2.3.5 Adaptive Toolbox

Complementary to these works, Gigerenzer [2003] describes the “Adaptive Toolbox”, as a way of thinking about ecological or bounded rationality — that is, how they act on the basis of incomplete knowledge about a problem domain. One’s Adaptive Toolbox is a measure of one’s ability to make decisions with limited time or knowledge.

The model is constituted thusly:

- A set of rules for using heuristics.
- A set of core mental capacities.
- A set of heuristics for solving a particular problem.

Thus, the Adaptive Toolbox prescribes how humans integrate heuristics into their decision making and problem solving processes.

It is clear that, given the propensity for CLI domains to expose huge sets of referents, that human problem solvers may almost always act on the basis of incomplete knowledge when using CLIs.

Todd and Gigerenzer [2001] also describes how NDM processes can be formally modelled using the idea of Adaptive Toolboxes, using the example of a prescription of a model for heuristic search.

2.3.6 Collaborative Models

It is possible to think of computation helpers of problem solving as qualified agents which collaborate with their human counterparts. New facets of a problem solving exercise emerge when artefacts with agency are considered.

Klein et al. [2004] explores some requirements for collaborative human-agent systems:

- Parties enter into an agreement that they wish to work together (the "Basic Compact" [Klein et al., 2004]).
- There must be some interpredictability between the actions of parties.
- Parties must be able to direct each other.
- "Common ground" — that is, a shared understanding of information available to all members — must be maintained between parties.

Various issues that arise when dealing with human-agent systems are also discussed in light of these requirements. These mainly touch on the difficulties in fulfilling the basic requirements for collaboration.

One issue in particular is the ability for agents to adequately model the intents and actions of other participants. This is vital in allowing agents to assist in problem-solving exercises — understanding when a human participant is struggling to solve a problem or how they are proceeding down a standard path to a solution.

Understanding must be developed in all parties through successive interactions with one another. While humans engage in learning by exploring a problem space, intelligent artefacts can learn about the human actors within a problem space, too. This may be in conjunction with learning about the problem space itself from a human actor.

These sorts of intelligent artefacts can utilise the information they learn to assist human actors in problem solving activities. Indeed, in the context of shifting attitudes towards hybridising graphical interfaces, and the incorporation of intelligent artefacts in these, such as chatbots [Štolfa, 2016], these issues seem extremely pertinent to contemporary development of CLIs.

2.3.7 CLIs for Problem Solving

It seems that the role of CLIs within problem solving remains to be discovered. It is possible to explore CLIs in their roles as windows into a specific domain or problem environment.

However, they might also be appraised as a way to expose collection of tools which can be used to assist problem solving within a domain.

The way in which humans use tools is a topic explored thoroughly from an ecological standpoint [Gibson and Ingold, 1993], as well as from a cognitive psychological standpoint. Artefacts are recruited to help solve problems, and in doing this, they become tools for problem solving. This recruitment of tools, particularly within CLIs, might be understood as an "exploration" of sorts — gaining an understanding of what referents and objects in your environment can do, and how they can help achieve some goal.

How human beings make decisions directly affects the design of a CLI tool for the purposes of problem solving. The question remains as to what nature of decision making CLIs can support in order to augment the problem solving process. Perhaps they must support both deliberative and heuristic methods to be effective.

2.4 Command Line Features

By exploring key issues surrounding the CLI paradigm and examining models of human problem solving, we are able to identify particular design aims which must be addressed. In order to gain an understanding of how these usability defects can be mitigated, some enhancing features of CLIs will be explored, and appraised with respect to their ability to support problem solving processes.

2.4.1 The Unix Philosophy

The Unix Philosophy is presented here as a philosophy which lends some perspectives on the design of effective control languages for CLIs. Kernighan and Mashey [1979] initially proposed aspects of this design philosophy, which were later formalised by Bergeron et al. [2003] as the following three rules:

- Write programs that do one thing and do it well.
- Write programs to work together.
- Write programs to handle text streams, because that is a universal interface.

These rules are proposed as "cultural norms" which Unix developers should abide by, such that tools can be easily maintained and repurposed.

However, Norman [1981] criticised the initial form of the philosophy detailed by Kernighan and Mashey [1979], stating that the way engineers comprehend and form a cognitive model of a system should necessarily be different from cognitive models held by end-users.

It seems that some sentiments of the Unix Philosophy can be observed to be beneficial to users, as they place importance upon the simplicity of referents, but it is pertinent to bear in mind that it was originally proposed for the benefit of software engineers and not end-users.

2.4.2 Unlimited Aliasing

Wobbrock et al. [2005] details an approach to designing CLI syntax known as "Unlimited Aliasing", where any command word can be typed and interpreted with respect to a referent. This permits the so-called "Sloppy Syntax"[Miller et al., 2008] paradigm, where users do not need to learn or memorise the syntax of a control language. In this paradigm, labels and connectives can be in any order or omitted, parameters are automatically matched and synonyms for labels can be used.

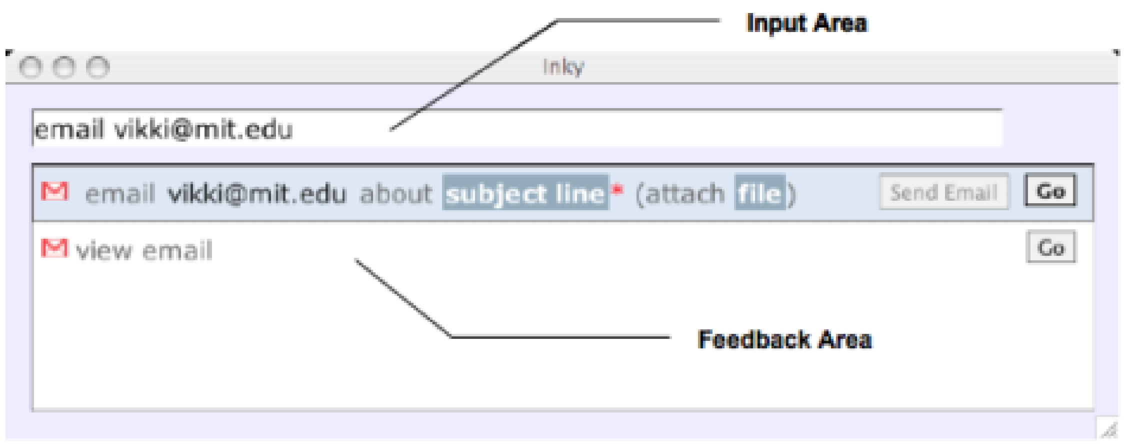


Figure 2.1: Miller et al. [2008]'s web CLI, Inky. Incorporates "Sloppy Syntax" and realtime visual feedback. (Source: Miller et al. [2008])

This paradigm makes the CLI significantly more beginner friendly — new users are able to operate within and explore a problem space almost instantly, with no learning overhead. This is because Unlimited Aliasing solves the vocabulary problem of CLI interaction. However, it is also not without it's disadvantages — users no longer have the safety net of syntax errors to prevent unwanted action from occurring. Indeed, there is possibility for the artefact's interpretation of a user's request to be wrong in potentially catastrophic ways.

2.4.3 Natural Language Processing

Similar to Unlimited Aliasing, the aims of Natural Language Processing (NLP) are to derive intention or meaning from natural language command phrases issued by a user. Real world application of this technology is typically seen in intelligent personal assistant applications such as Apple's Siri.

NLP, as distinct from Unlimited Aliasing, aims to act on fully formed phrases or pieces of human language, rather than simple unstructured sets of keywords — structural elements of language, such as grammar, play an important role.

Groth and Gil [2009] details an approach to the design of a tool which utilises NLP for specifying scientific workflow — this is an example of a specialised, reflective problem solving activity. The study concludes that their system must deal with well-known phenomena related to NLP, including "ambiguities, lack of referents, and unresolved attachments". It is suggested that systems based on controlled natural language, with well defined grammar, might be more feasible, whilst still retaining the benefits of NLP.

Groth and Gil [2009] also seems to suggest that allowing a user to define their intent to action in their own words might enable them to more effectively conceptualise about the problem space they operate within.

2.4.4 Domain-Specific Languages

Domain Specific Languages (DSLs) can and have been utilised for symbolic input with CLIs. Freudenthal [2010] describes DSLs as executable specification languages that offer expressive power focused on a particular domain. They typically constitute a formal grammar and a set of command symbols which resonate with their domain expert users.

DSLs allow a different approach to operability of a command syntax than that of NLP and Unlimited Aliasing. They allow users to define their intent to action in terms of the domain, but also promote rationalised and uniform structure. This suggests support for decision making within rational problem solving processes.

2.4.5 Visual Feedback

Miller et al. [2008] details an approach to integrating graphical elements within CLIs. These graphical elements permit realtime visual feedback as the user types commands, and in some cases enhances functionality.

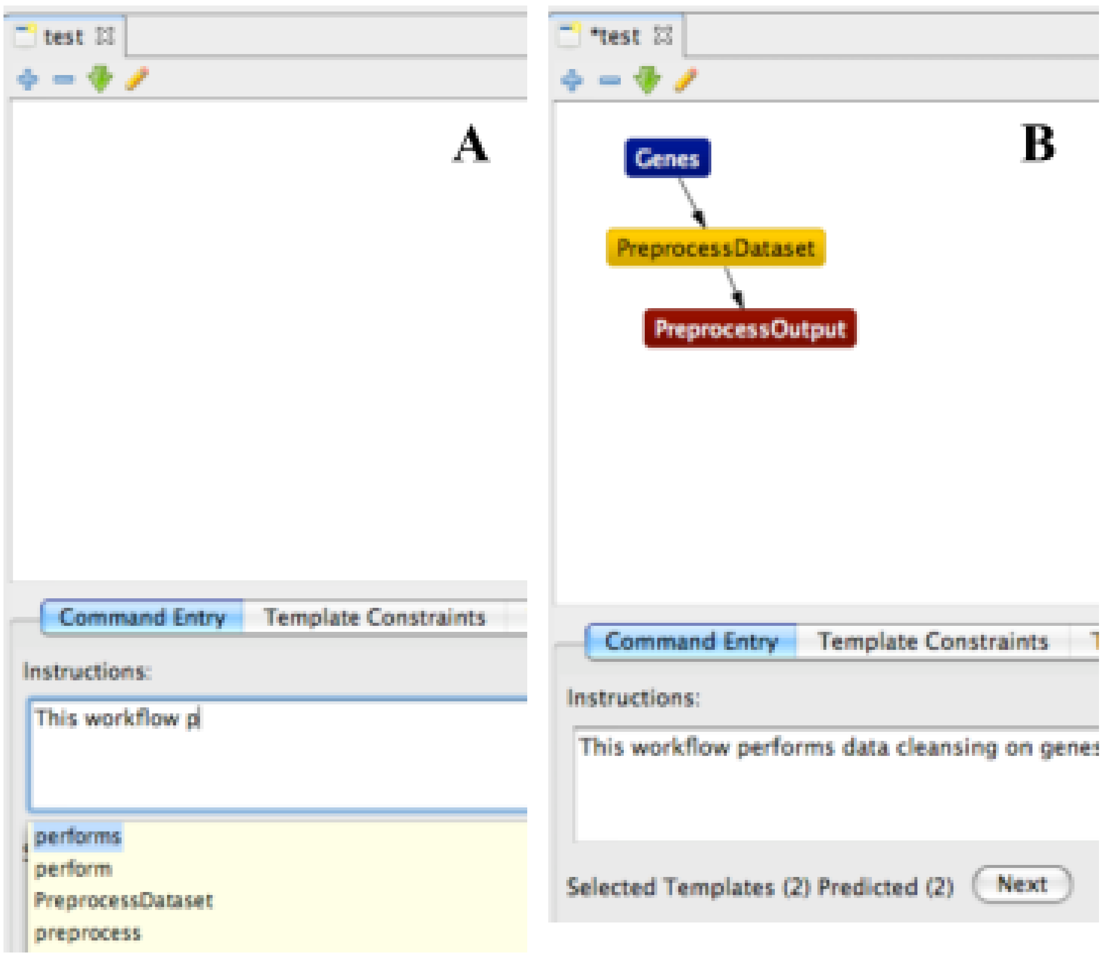


Figure 2.2: Groth and Gil [2009]'s CLI tool for constructing scientific workflows. Integrates NLP and visual function composition. (Source: Groth and Gil [2009])

This increases the interactivity and engagement of the CLI user experience. The internal state of the application is exposed to the user as the interpretations of the user's command are continuously revised.

This allows users to incrementally revise their commands in turn, allowing them to see where errors have been made or parameters have been missed out before the command is submitted. This solves the poor visibility issues that CLIs typically exhibit, and more greatly enables users to search the affordance space of the interface.

The real-time interactive aspects of this sort of feature promote reflexive decision making as prescribed by RPD models [Klein, 1995].

Visual Function Composition

As discussed priorly, Groth and Gil [2009] details an approach to a tool for the construction of scientific workflows. These workflows, when constructed, are shown visually as flow diagrams.

This visually externalises the flow of the user's data through different application functionality. It is clear that this may be an effective method of allowing a user to plan, iterate on, and reason about solutions within a problem space.

2.4.6 End-user programming

End-user programmers are defined as those who create programs which "are not the end in itself, but rather a means to accomplish their own tasks or hobbies" [Cao et al., 2010]. Many CLIs support end user programming via scripting. On Unix based systems, for example, this can be done via Shell scripting, or extensions thereof. End-user programming enables CLI users to effectively automate complex manual tasks for a massive number of cases.

Shells are cited [Kernighan and Mashey, 1979] as an interactive form of end-user programming — commands are applied iteratively to a problem space to achieve a goal state. This is an interesting perspective in the context of problem solving. The combination of rational problem solving benefits of a programming language with interactive elements permitting RPD warrants inclusion in a general design approach to CLIs.

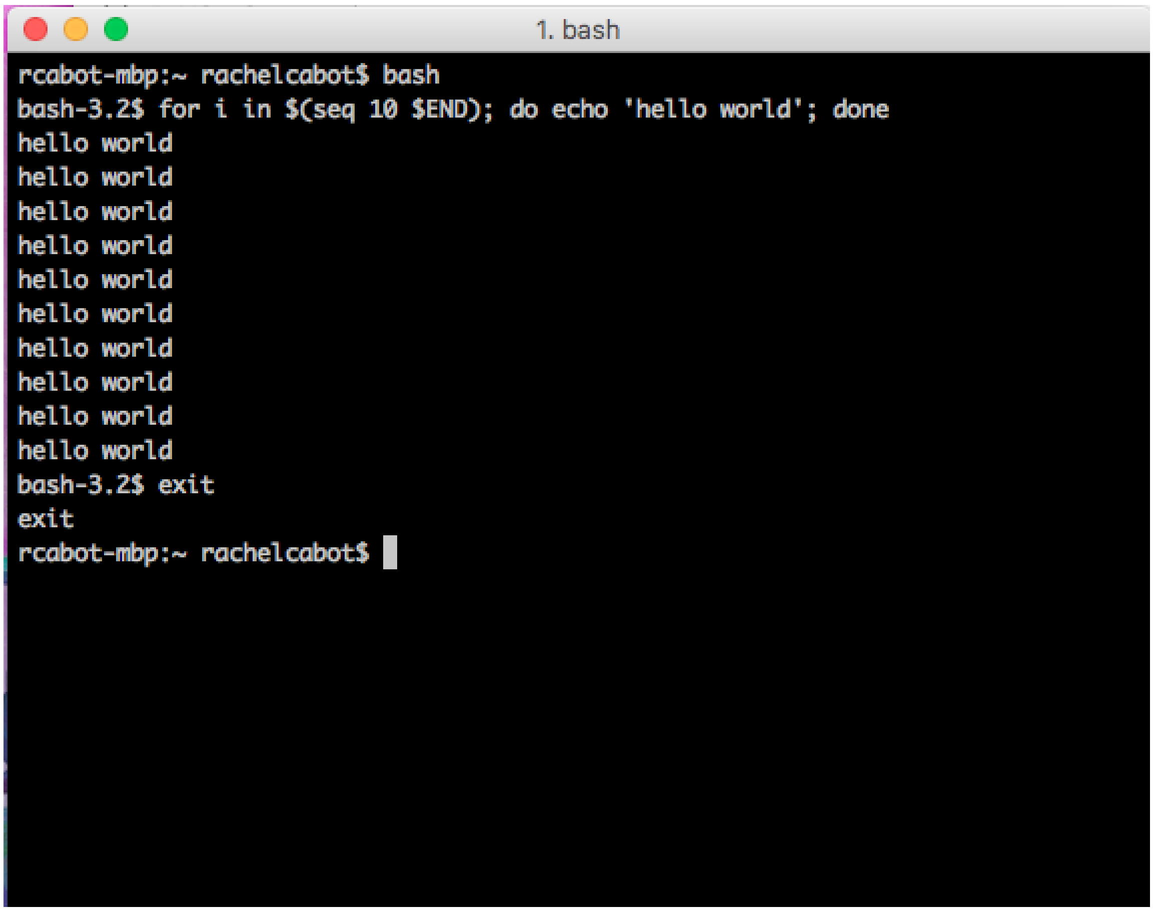
Macroinstructions

Macroinstructions (or simply macros), are rules which specify how a set of inputs should be mapped to a set of outputs according to a defined procedure [Greenwald, 1958]. These procedures can form part of a user defined functionality within CLIs. Users apply an alias to a set of commands which execute together to produce an intended output for given inputs. Features permitting this might enhance planning processes within problem solving scenarios.

2.4.7 Wildcard Expansion

Wildcard expansion is a feature within many CLIs, particularly in file based systems which use a specific instance of this, known as file "globbing" [Kerrisk].

A wildcard is a character which is used to represent multiple characters. Therefore strings with wildcard characters actually represent a set of strings. Wildcard expansion, therefore, is the process of finding all objects whose names exist in the set of strings represented by



```
rcabot-mbp:~ rachelcabot$ bash
bash-3.2$ for i in $(seq 10 $END); do echo 'hello world'; done
hello world
hello world
hello world
hello world
hello world
hello world
hello world
hello world
hello world
hello world
bash-3.2$ exit
exit
rcabot-mbp:~ rachelcabot$
```

Figure 2.3: The Bash interactive shell [GNU]. Enables interactive end-user programming.

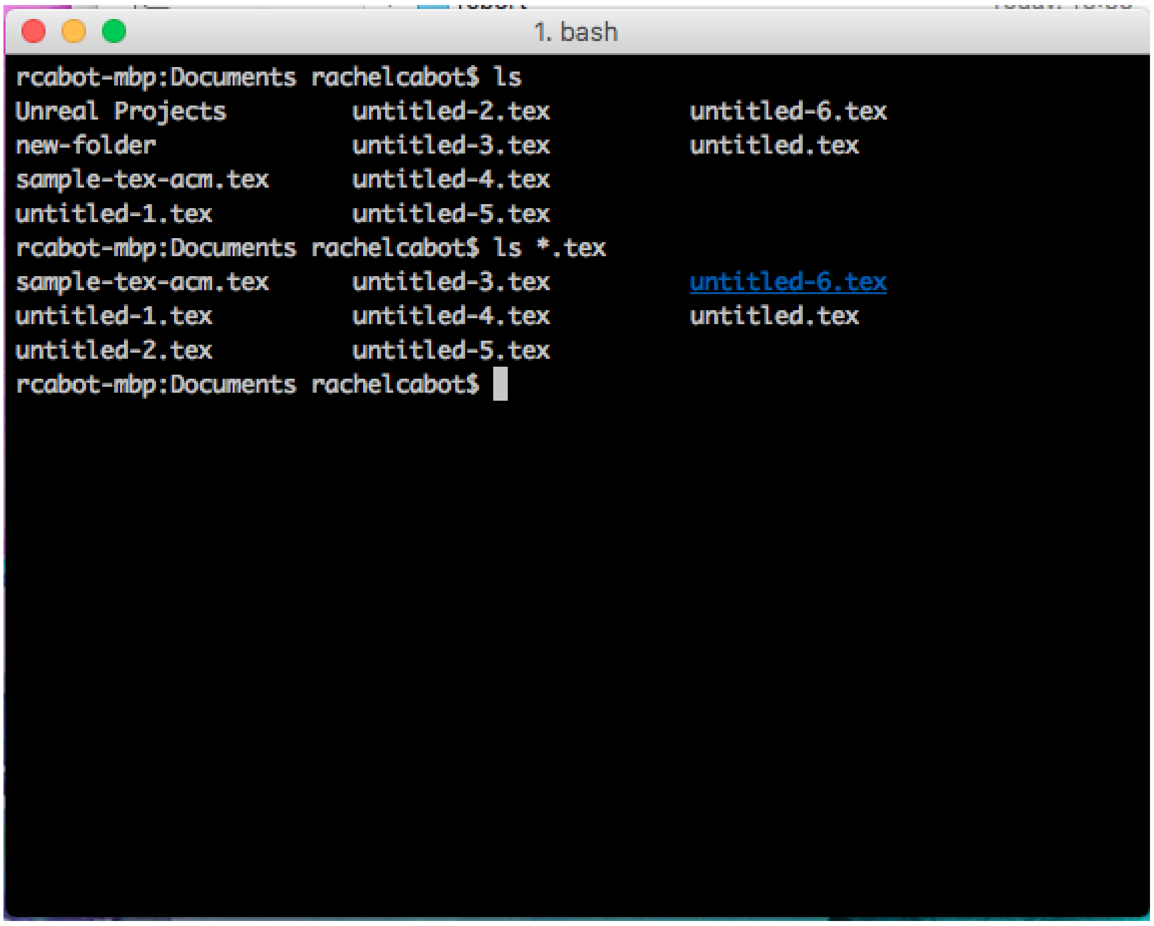
the wildcard string.

Wildcard expansion is a powerful form of search which allows an end-user programmer to obtain and manipulate a set of objects within a problem space. This powerful mechanism may aid a user in thinking about problems in a set-oriented way — one in which tedious tasks can be transformed into efficient iterative processes.

2.4.8 Command History

Most CLIs permit the visibility of commands that have been typed previously in some way.

Unix command line tools, for instance, allow a user to press the up and down arrow keys to scroll through their previously typed commands [TLDP]. These can be edited and executed in the normal way. This feature actually affords a form of error recovery — when an



1. bash

```
rcabot-mbp:Documents rachelcabot$ ls
Unreal Projects         untitled-2.tex          untitled-6.tex
new-folder              untitled-3.tex          untitled.tex
sample-tex-acm.tex      untitled-4.tex
untitled-1.tex          untitled-5.tex
rcabot-mbp:Documents rachelcabot$ ls *.tex
sample-tex-acm.tex      untitled-3.tex          untitled-6.tex
untitled-1.tex          untitled-4.tex          untitled.tex
untitled-2.tex          untitled-5.tex
rcabot-mbp:Documents rachelcabot$
```

Figure 2.4: Example of wildcard expansion in Terminal [Apple, b]. Providing the option `*.tex` to `ls` lists all files with a .tex extension.

erroneous command is issued, the user can simply press the up key, correct the command and then re-run it.

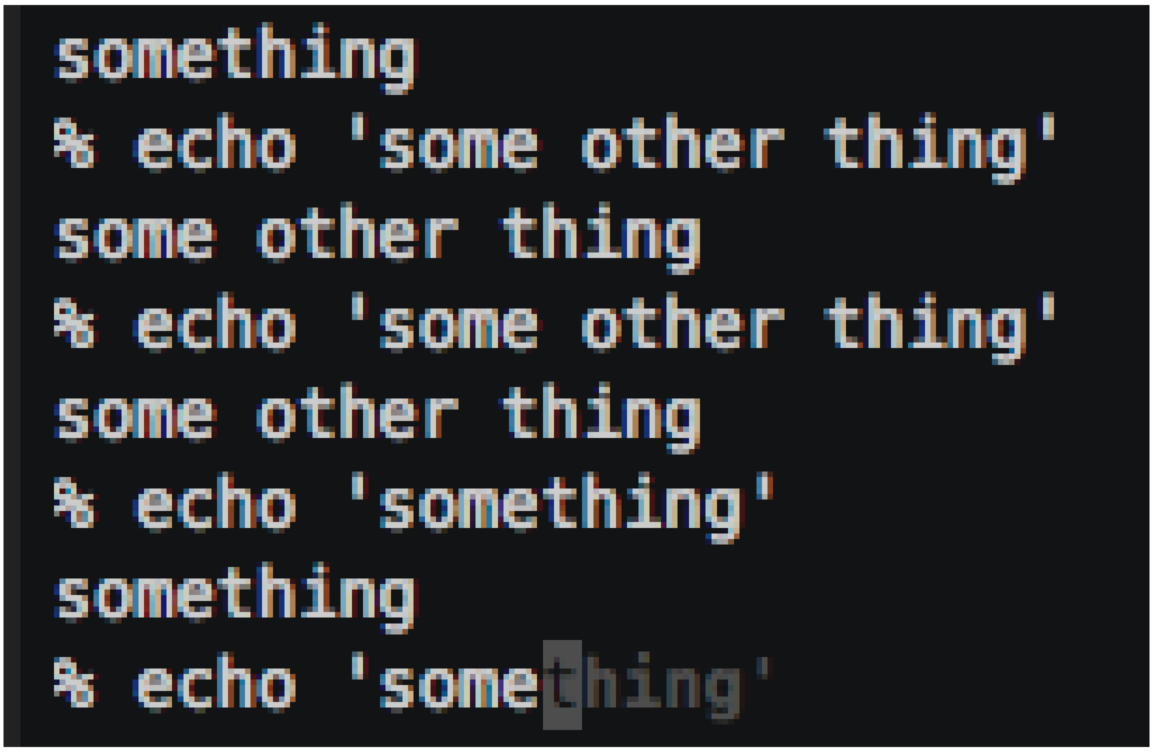
Some Unix shells also provide a history command, which shows a complete list of the previous commands [TLDP]. There are also certain shorthand commands which allow a user to re-execute the nth previous command [TLDP].

Helping a user remember what they have done previously may promote an understanding of their journey towards the solution to a particular problem. It seems that command history, when presented back to a user, might be conducive to recognition-based reasoning, as previously discussed (section 2.3.3).

2.4.9 Contextual Autocompletion

Shell enhancements such as Fish [Doras] and zsh-autosuggestions [de Arruda, 2013] integrates command suggestions into CLIs. These suggestions appear interactively in a dim colour as a user types, and are based on the context in which they are typed, and the user's command history. Pressing a confirmation key such as tab or left arrow automatically fills in the suggestion. This essentially affords a more rapid way of accessing command history.

These sorts of autosuggestions offer users the opportunity to more rapidly search for and re-execute commands they have typed before to CLIs than they would otherwise.



```
something
% echo 'some other thing'
some other thing
% echo 'some other thing'
some other thing
% echo 'something'
something
% echo 'something'
```

Figure 2.5: Demonstration of zsh-autosuggest [de Arruda, 2013].

Such a feature can be extended to showing a user what is afforded by the interface by displaying any possible or relevant commands. This may help support decision making processes the user undertakes when solving a problem, or help users to learn what actions are possible within a problem domain.

2.4.10 Spelling Autocorrection

The goal of spelling correction is to validate typed user input with respect to a given dictionary. Autocorrectors do this by automatically changing incorrectly spelt words in a user's input to "the most likely correct word" [Peterson, 1980].

Durham et al. [1983] touches on the uses of spelling autocorrection within user interfaces. These sorts of mechanisms can help prevent the entry of erroneous commands within CLIs.

2.4.11 Reversal Commands

Certain applications with CLIs, enhance robustness by providing undo functionality, such as AutoCAD's "oops" command [AutoDesk, 2015]. This command reverts destructive changes to the drawing environment, but not constructive ones — this preserves progress which might have been achieved since a mistake was made. This is an excellent example of how CLIs can support robust interactions.

2.5 Summary

In this chapter, a particular understanding of how users approach problem solving with CLIs has been explored. This has been appraised from many perspectives by exploring various models, success criteria, definitions and exemplar artefacts.

2.5.1 Key Issues for Effective CLI Design

An aspect which draws much of this research together is that of better enabling users to explore operational details of CLIs. In the context of this, we may provide a cursory definition for a so-called "Exploration Problem", characterised by the following aspects:

- CLIs which expose large functionality sets typically by definition lack signification of referents.
- A user must somehow browse or explore a set of valid commands in order to discover a required set of commands pertinent to solving a problem.

The main focus of this body of research is in the discovery of how CLIs can be enhanced to solve this Exploration Problem, and of characterising facets of this problem.

Avenues for enabling this through design of a CLI artefact for problem solving include the following:

- Syntax design — the goals of which are to make it easier for a user to comprehend an underlying domain model, and provide meaningful language for description of aspects of the problem domain.
- Visualisation — interactively presenting the system's interpretation of a command in order to enable search of an affordance space and support reflexive decision making as described by Recognition-Primed Decision models.
- Command chronologies — displaying the history of executed commands to allow users to explore patterns in their actions, and understand the path of their exploration of a problem space.
- Command suggestion — showing a user what is possible by providing signifiers of afforded commands, given a particular scenario or context.

Chapter 3

Exploratory Study

An understanding of the nature of human problem solving and CLIs has been explored through literature. Consequently, it has been concluded that the focus of this body of work should be upon the Exploration Problem of CLIs. In particular, chapter 2 identified syntax, command chronologies, and suggestions as areas of potential.

To better inform attempts at the design and implementation of an enhanced CLI, primary research of expert CLI users was conducted to deepen and expand on findings from the literature. This chapter reports an assessment on the salient and sensible aspects of the goals set out in the Literature Review, which will enable the formation of success criteria for the design of a CLI.

To begin, the research questions we wish to address will be formally defined with respect to the literature previously examined. Next, the design and procedure of the study undertaken will be specified. The main themes which emerge from these procedures will then be identified and discussed. Finally, these results will be discussed with relevance to the study questions, and we shall detail a critical analysis of the procedure carried out.

3.1 Research Questions

In order to clearly set out the objectives of the study, several research questions are presented here, which informed the method carried out. These questions are based on the aims previously discussed, in order to cement the foundation of knowledge previously obtained and to explore other avenues of interest.

Mirel [2004] highlights the importance of the examination of usefulness when it comes to designing software artefacts that help solve problems, as well as secondary concerns such as usability. Complex problem solving applications should, after all, be recognised as a "distinct class of software with its own usefulness demands" [Mirel, 2004].

Thus, it is important to question whether users find CLIs useful for solving problems at all. It may be suspected that this only applies to certain sorts of problems, or problems that are characterised by aspects which CLIs readily address.

Expert users may also have CLI tools of choice for solving day-to-day or specialised problems — these might be examined to attain an insight into how CLIs can be useful.

Therefore, this study should set out to explore, in principle, the following questions:

- In what way do CLI users recognise the utility of CLI tools for solving problems?

- What characteristics of CLI tools make them helpful for solving problems?
- What affects the experience of CLI users when problem solving?
- What are the characterising features of problems which users find CLIs are helpful in solving?
- What do expert users value in their CLI experiences?

3.2 Design of Study

This study mainly comprised of interviews with a group of expert users of CLIs. They were asked to informally demonstrate typical usage of a familiar CLI environment, and then a series of semi-structured questions were posed relating to the demonstration.

Recordings of interviews were qualitatively analysed through Thematic Analysis [Braun and Clarke, 2006]. This technique was suitable because it emphasises the examination patterns or themes within qualitative data, which allows meaningful and salient aspects relating to the research questions to effectively be discovered. Screen capture recordings were also be taken to contextualise and supplement this data.

In doing this, we aim to gain the perspectives of expert users of CLIs, which help to answer the research questions posed.

3.3 Procedure

Here follows a detailed description of the procedure carried out. This procedure was designed in accordance with the University of Bath Computer Science Ethics checklist (see form in appendices).

3.3.1 Participant Recruitment

Participants were acquired through personal interaction with members of the University of Bath Computer Science department. Only those who showed interest in or claimed to have prior experience of command line technologies were selected. Each participant signed a consent form — these can be found in the digital appendices, and a copy can be found in the digital appendices.

3.3.2 Interviews

Participants were each interviewed for 1-3 hours in quiet, undisturbed environments. They were encouraged to bring laptops which had a CLI that they were particularly familiar with. If consented to, participants were recorded via audio and screen capture, but notes were also taken to supplement these.

The interview was conducted in a semi-structured manner, centring around a few key aspects. Participants were first asked to reflect on the CLI technologies they use, and asked to demonstrate tasks they typically do or have done recently with a CLI — fabricated tasks may be introduced to demonstrate more nuanced functionality. There followed a discussion about their experience and use, with respect to various aspects or themes pertinent to the research goals, including the following prompts:

- Exploration — how do participants browse for resources required for task success?
- Discovery — what methods of discovering features pertinent to task success do participants find useful?
- Usability — what makes a CLI more or less usable?
- Efficiency — how do participants engage in CLI interaction with low cognitive or physical effort?
- Flow — how do participants experience or achieve states of fluent action?
- Syntax design — how does the design of a control language affect a participant's experience?
- Information visualisation — how do participants feel about visual elements within CLIs?
- Command history — do participants find command histories useful, and if so, why?
- Command suggestion — would or do participants find command suggestion mechanisms useful?

3.3.3 Audio and video transcription

After the data was gathered, PopupArchive's automatic transcription service was applied to the audio recordings. The transcription was edited where appropriate in order to improve the quality.

3.3.4 Codification of Data

The codification of the transcript data was carried out largely on the basis on procedures outlined by Braun and Clarke [2006]. The transcription was read and re-read along with the recordings to garner a high-level understanding of the data gathered. “Low level notes” were taken, whereby the pertinent aspects of every conversation were highlighted and captured. This text was annotated at the relevant parts of the transcript to achieve proper situation within the data.

After this process was completed, the notes were collapsed into codes — short two or three word phrases which successfully captured the overarching sentiment of each conversation point. Efforts were made in places to transcribe variations in speech, such as emphasis or inflexion, in order to correctly capture the sentimental content, by again listening to the recordings in real time. This process produced a document of codes, which effectively describe how the data addresses the research questions.

These codes were reviewed again, and unified within a set of thematic titles which accurately represent at least two or more captured codes. The process undertaken was to produce some initial themes based on a current sense of the data, sort codes into those, or into a miscellaneous section where no other section was appropriate, and then repeat this recursively on the miscellaneous section until appropriate. References to the person whose conversation produced the code through colour-coding, as well as the order of appearance of codes, were maintained. The themes and their “force” were then analysed with respect to the surrounding literature. Claims that are made about the data were verified via “member checking”, whereby the original codes were used to trace specific conversation points being used to support the claim.

In light of this, great efforts were made to achieve necessary submersion in the data; this process alone took around 40-50 hours.

The documents generated during this process can be found in the digital appendices.

3.4 Participant Profiles

Table 3.1 displays some information about the recruited participants, the technologies they decided to demonstrate and how many years they had been using the technologies described. Whether participants consented to audio or screen capture recordings is also indicated.

Participants provided no explicit consent to the use of their name within this report, and therefore their names have been replaced in the interests of privacy.

Name	CLI Technologies of Choice	Experience	Audio	Screen Capture
Dannielle	Bash [GNU], Git [Torvalds]	3 years	Yes	No
Yvan	Fish [Doras]	2 years	Yes	Yes
Amy	Zsh [Falstad], TMux [Marriott], Konsole [Doelle]	6 years	Yes	No
Rowan	Bash [GNU], Curl [Stenberg], Python [van Rossum] REPL	3 years	Yes	No
Ellen	Linux Shell, Vim [Moolenaar]	10 years	Yes	No

Table 3.1: Table displaying participant information.

3.5 Thematic Analysis

Here follows a description of the themes produced from encoding and unifying processes.

3.5.1 Enhancing the Base Experience

All participants discussed enhancements of a "base experience" or vanilla distribution through customisation, and those that opted to not customise their experience were already using enhanced CLI tools.

It is unclear to what extent a CLI tool will always need to be customised by a user, or if there is something intrinsically inadequate about the CLI experience that is consistently resolved by expert users. When questioned in what way they would improve the base experience of CLIs, participants often alluded to adding intelligence or contextual awareness to the command line in some form, though they did not always agree on whether this would be effective.

Many participants expressed their annoyance with reconfiguration when moving to different command line tools, and this might be in part attributed to that fact that users must re-solve problematic features of the base command line experience.

User Customisation

Participants appeared to customise their CLI tools to varying degrees, ranging from none at all, to heavily alterations from the base experience. Where users did not customise their tools, it was typically because they used command line tools which came with optimising features such as autocomplete — such was the case with Yvan and his usage of Fish [Doras]:

"I haven't changed much about my setup", "has nice extra features".

Some participants discussed their use of "dotfiles", such as the standard bashrc file within Unix systems, for specialised startup actions and environment variables, but there is no evidence to suggest that they are utilised ubiquitously.

Ellen in particular touched upon how customisation can allow a command line user to streamline their experience by narrowing down their feature sets to only the tools they understand or know they want to use: "You can customise it easily so... you don't have to have a bunch of stuff you don't want". This has some interesting implications for conceptualisation; perhaps smaller feature sets within a CLI are easier to learn?

Many participants made it clear that they appreciated the ability to personalise and customise visual aspects of the command line, such as colour and font, and some touched upon the value of certain aesthetic choices such as monospace readability.

Shell Intelligence

All participants offered their perspectives on context aware intelligence resident within a command line tool, sometimes as a response when questioned how they would improve the base experience of CLIs. Such an intelligence could provide suggestions or recommendations based on the tasks being performed by the user.

Participants often stated that there are problems with the CLI paradigm that such intelligence would solve. However counterpoints to this were also offered — intelligent intentions could disrupt or fault the user's own intentions, for instance, or incorrect suggestions could be costly in the context of a critical task.

> "Possibly some sort of context aware help mode... recognises what state you're in" — Dannielle
>
> "I don't need something offering suggestions to me" — Amy

Indeed, it seems pertinent at this juncture to refer to the works of Klein et al. [2004], which details how artificially intelligent agents can play a collaborative role in human problem solving — it is clear that there are some challenging aspects to overcome, but there is significant benefit to be achieved from these methods.

3.5.2 Reflexive vs Reflective Action

Several participants referred to certain actions as “innate”, meaning sufficiently overlearned so that they were produced automatically when certain situations or trigger conditions were encountered.

User action within a command line appears to lie on a continuum of this sort of “innateness”, which dictates how rapidly a user can carry out their intent. There is evidence to suggest that there is a translation step which occurs which transforms a user’s intent into a command they must type, although in some cases this action might be more nuanced.

Many basic options, such as navigating, gaining visibility and searching have a short or instant translation step, whereas commands required to solve nuanced problems may require a longer translation step, sometimes even requiring the aid of documentation.

At this point, we might invoke works from Evans [2008] on Dual Process Theory — it is clear that some distinction on these lines can be observed in the sorts of actions described by participants. Other works which increase the force of these notions are aspects of Norman [2004]’s levels of design.

Basic Operations

Participants described basic operations as being those which have entirely predictable and deterministic behaviour, and are often similar or consistent across different shells.

Some participants indicated that certain actions were almost habitual, and in some cases described a negative effect on their experience from habits which were suboptimal or degenerate — typing the `ls` command too often, or clearing the screen and losing vital information, are amongst those described.

Every participant touched upon the concept of workflows, whereby commands are typed in a defined sequence consistently to accomplish a recurring task. Yvan in particular indicated that these workflows habitually emerge and examples cited included compilation or file management.

Many participants discussed “innateness” of certain actions when working with a command line. This was often described as being second nature and without reflection, the tool they used simply “fading into the background”. These sorts of factors suggest ideas of a virtuous flow experienced by expert users of command lines — Amy in particular alludes to this, stating that for such action, “... intent is translated fairly fluently into action with no conscious planning process".

Indeed, it would seem that parallels may be drawn between these notions and aspects of RPD models proposed by Klein [1995]. "Innate" actions seem to be those which are taken on the basis of highly recognition-primed decisions.

Intent-Command Translation

The idea of a translation of intent to a command is brought up by most participants — that there was a syntax barrier to overcome, or that their plain-english intentions had to be reformulated into command language. One participant stated, however, that no such translation step was experienced by them — their intents manifested directly as commands. It is not unreasonable to assume that this is down to a high level of expertise, and as such the perceived translation is instant.

Participants described situations where they knew what it was they needed to do, but were not sure how to perform the action with a command line. This can be attributed to the discoverability problems of CLIs.

Often, users must defer to trial and error, or seek help from documentation or help from peers. This hampered act of discovery directly opposes the efficiency goals described by the participants because it often constitutes a context switch away from the CLI tool.

In some cases, participants reported that they were discouraged from looking for better ways to solve problems, and often made do with less efficient uses of more well-known commands in order to solve a nuanced problem.

> "Sometimes it's quite annoying to find out how to do something... you might know a really easy solution but you don't know what the [command] is... and sometimes it takes such a long time to find the correct solution that it would have been faster to do it in a different way." — Rowan

It seems that Rowan is describing how the formulation of his intentions is easily done, but action specification is not always immediately possible. Indeed, we may invoke works such as Norman [1988]'s Human Action Cycle, which details steps in which a goal is translated into a sequence of actions, to provide force for these claims. It would seem that the notions relating to the Gulf of Execution [Norman, 1988] are applicable here too.

Novel Problems

There is evidence to suggest that discoverability problems with CLIs are most pronounced when users of CLIs are faced with nuanced problems — these are tasks which participants describe as being outside their habits, atypical or complex. This can likely be attributed to the fact that these tasks require tools or referents which lie outside of a user's immediate expertise.

Participants stated that it is often faster to resort to trial and error when attempting to discover a required piece of functionality. In this regard, they stated that they enjoyed "did you mean" command suggestions within error states[1], and that their environments were forgiving of mistakes — commands may be used without prior understanding or assurances of validity.

Dannielle, in particular, postulates that discovery through trial and error aids in learning processes and helps one remember a solution to a problem. We may at this juncture invoke Thorndike [1927]'s foundational theory of Connectionism, which provides force for claims that CLI users learn through trial and error processes.

Some participants discussed the inadequacies of manual pages, the native documentation format, often stating they were unsearchable, not well laid out or simply inadequate for discovery of functionality. Dannielle emphasises this: "I would also google it... `-help` is good when you're asking 'what can I do' and bad for 'how can I do this'".

Help flags are praised by some participants as allowing them to perform quick documentation checks on a known command — the minimisation of context switching is often cited as an important advantage. This highlights an interesting distinction for the Exploration Problem previously set out — knowing if a piece of functionality exists, versus knowing how to carry out one's intent with the command language provided. Does the referent exist, and if so, what's the command for it?

The inadequacies of in-application documentation are further highlighted by the fact the participants state that they often defer to searching online for a solution to a nuanced problem. As Amy highlights, "If it's something esoteric... I'd probably google it".

Though participants tend to state that this can be a good learning mechanism, or useful for discovering the best solution to a problem, it can be argued that this worsens the Exploration Problem by increasing overhead of referral. We can invoke works such as Pashler [1994] that increase the force of claims stating the deficiencies and overheads

[1]Git workflow errors were often cited by participants as a good example of this.

of context switching. Rowan in particular states that switching to a browser to check online documentation is distracting, and constitutes a costly context switch. This inadequacy obstructs the command line user's most important goal, speed and efficiency of action.

It is unclear, however, whether in-browser documentation should continue to play a role in a command line user's experience. Many participants state they prefer well-formatted documentation presented online, and that the context switch isn't a severe problem for them as long as it is short in duration. Perhaps the solution to this is to provide better formatted in-application documentation for command-line users.

3.5.3 Efficiency

Across all participants, the goal of all participants was to achieve a fast or streamlined experience with the command line which required the minimum amount of effort.

Most participants discussed the strengths of command lines in terms of how they provide them with the ability to complete tasks swiftly, and provided them with an unobstructive, unobtrusive and undistracted experience. This can be attributed to the fact that CLIs expose a large amount of functionality structured by languages with powerful syntactic structures — that is, they allow users to complete huge tasks simply by typing a few short phrases or symbols. However, many aspects of the CLI experience also stand in the way of this goal.

> "...things that would streamline my usage, help make things go quicker and make things easier to understand and guess." — Yvan

Completing Tasks Quickly

Participants describe obstructions, obtrusions and distractions as distinct factors which prevent them completing tasks quickly.

Obstructions are discussed by the participants as being factors which stand in the way of useful action. Examples such as unexpected effects, organisational procedures and understanding a problem are cited as obstructive factors.

Participants describe obtrusive factors as those which disrupt the formation of their intent in some way — this is often brought up in discussions surrounding context-aware suggestions or help mechanisms for the command line.

Distracting factors are those which participants commonly described as "breaking their flow" — presentation of unwanted information or context switching are strong examples of these. It is clear that these distractions could occur at a visceral level — due to eye-catching or rapidly changing visual features — but these may also arise to a reflective need to self-interruption. The force of this claim is supported by works on multitasking from Payne et al. [2007].

Some participants discuss the importance of only having to use a keyboard in their interactions with the command line, in the context of rapid use. Amy highlights a particular benefit: “The mouse is annoying... there’s a lot to be said for not having to move between keyboard and mouse... I would prefer not to have to use the mouse”. This is amongst other discussed benefits, such as muscle memory, “home-row” navigation and the speed of touch-typing as promoting rapid and natural expression of intent.

Power of Expression

Power of expression is a large part of what drives the efficiency of CLIs, and can be attributed to various features of traditional CLIs — participants often touched upon these in the context of what made CLIs useful for solving problems.

> “It’s way more powerful you save so much time” — Ellen

Composition — the ability to combine pipelines of tools — is the most often cited example of a powerful method for solving complicated multi-step problems. Amy, in particular, discussed this feature in the context of the Unix Philosophy — the provision of a large set of small, composable tools is evidently conducive to deriving solutions to novel problems. Having a standardised format of input and output appears to be central to the usefulness of this feature.

Many participants stated that they felt they had unprecedented control over their environments of operation when using a command line, and that it was important to them to feel in control. There is evidence to suggest this manifests itself in the ability to acquire complete information, deterministic action and complete knowledge of a particular set of tools.

Some participants stated that they enjoyed the ability to script complex actions in efforts to automate the solution to a recurring problem, though not all participants described their use of scripting.

Most participants touched upon ideas of utilising multiple sessions — in some ways used to divide up different projects, but also used concurrently for the same task, suggesting a maintenance of foreground and background processes.

A few participants discussed the usefulness of re-triggering commands they had used in the past, either by pressing the up arrow, or through a reverse lookup mechanism. This allows very complex commands to be repeated instantly on demand — a powerful and essential feature of the command line paradigm.

Many participants also discussed the merits of package management as a tool for being able to rapidly integrate new and helpful functionalities into their workflow.

3.5.4 Learning

Descriptions from the participants show that the learning experience of command lines is highly dependent upon memorability of command syntax or names.

Most participants claimed that their initial usage was or felt forced, but continued to use them due to advice from peers. This may be attributed to the fact that commands are very often difficult to remember.

In addition to these facets, the learning and discovery of new commands is often described as "needs-driven" and thus never done ahead of tasks that the participants had to perform.

Memorability

Participant's responses often highlighted an interesting distinction between the memorability of commands and remembering that a particular referent exists. This offers some additional insight to the Exploration Problem.

> "There are so many commands that don't mean anything... [flags] are sometimes really unhelpful... weird orders... they're just inconsistent" — Rowan

> "Not all of the commands are logically named... completely different operations can have the same name... ambiguity is really common" — Dannielle

Almost every participant had something to say for what made syntax forgettable, citing deficiencies such as meaningless names, unlearnable acronyms, parameter orders and inconsistent flag characters/formats. Particular details of tools and commands which were ambiguous in their purpose were also noted as oft-forgotten. These factors perhaps indicate a greater problem with efficiency-optimised syntax and its subsequent detriment to the command line experience. The force of these claims relating to deficiencies is supported by notions discussed by Norman [1981].

All participants offered their opinion on what made command syntax memorable — these factors centred around higher guessability and cues to memory. Guessability factors described included well labelled commands and flags, similarities and consistency between tools, being able to safely experiment via trial and error and flexible or predictable parameter order. Cues to memory described included meaningful command acronyms, command aliases and helpful errors.

The importance of factors such as memorability and guessability within interaction design is supported by works such as Wobbrock et al. [2005] and design heuristics proposed by Preece et al. [2007].

Learning Paths

> "There's so much more to know, I just haven't had cause to know it... it's opportunistic skill improvement, you improve because you need to do something... my mentor said to use '`awk`'" — Amy

All participants eluded in some way to peers being a driving force behind their learning efforts, stating that they received helpful suggestions for commands or advice on nuanced aspects of the command line. In some cases, participants also stated that they liked seeing how other people approached problems, or discover a more expert user's approach to find the most optimal solution to a problem. The force of these notions is supported by works within the field of activity theory [Kuuti, 1996], which proposes that community plays a vital role within a domain of activity.

Some participants stated that their learning was often only by necessity or was "opportunistic" — commands are not typically learnt ahead of when they are needed and this learning is typically driven by means of their use. This introduces a recurring obstruction to the efficiency goals of command line users.

A common and interesting theme was how peers or a work environment "forced" participants to initially use command lines in order to convince them of the efficiency benefits they bring. Participants stated that, as new users, they felt the CLI experience was less friendly, and that the learning curve was steep.

3.5.5 Conceptualisation

Perspectives offered by the participants demonstrated the value of conceptualisation when tackling a problem with a CLI. It appears that the participants' approaches to problems are typified by splitting up the problems into parts which can be solved by individual commands — the conversational interaction provides a framework for understanding the solution to a problem.

The participants also discussed how the composition of commands enabled them to succinctly phrase the solution to a problem. Dannielle in particular stated that the command paradigm itself aids in understandability — that typing commands helps one understand their goals, allows their intents to be clearly specified and thus clarified, in an almost self-questioning mode of action.

Understandability

All participants in some way alluded to the value of understanding the tools that were at their disposal within a command line, often pointing out when certain aspects did not help or hindered their understanding of a particular aspect. The participants often valued an idea of complete systemic knowledge of their tools of choice. Dannielle highlights this by stating "[commands] make sense if you understand the underlying model". The force of these claims is supported by works into mental models, such as Halasz and Moran [1983], which demonstrate the positive effect of an effective mental model on problem solving activities.

Many participants discussed how they conceptualised or visualised certain aspects of CLIs mentally as an internal model. One such example, provided by Ellen, prescribes navigation of folder structures as cave exploration, with deep and branching paths.

Participants offered various ways in which command lines could support a better understanding of a systemic model, making particular reference to ways in which information is represented. These included output which is "directly" parsable, syntax highlighting and use of domain specific language.

Visibility of particular information was also well a covered theme when discussing aids to understanding — aspects such as processes, side-effects, context, status and static information must be shown to a user when convenient. Design heuristics proposed by Preece et al. [2007] add force to the notion that, for CLIs to be effective, they must provide timely visibility of system state.

Another part of user understandability is being able to know what one wishes to do next. Most participants stated that they typically understood what the next action in a sequence of actions was, but that factors such as 'did you mean' error messages or command suggestions could plausibly assist this process.

Divide, Compose and Conquer

Some participants discussed a clear “divide and conquer” approach to solving problems with their command line tools, and there is evidence to suggest this is the prevailing methodology for other participants. Yvan states that the command line “makes it easy to compartmentalise issues... helps you split down a problem”.

Participants also often discussed composition of commands in a way which supports this proposed methodology; separate and standard functionalities, which each solve part of the problem, being combined in a thoughtful and ordered way to achieve a solution.

Most participants stated the usefulness of their action history when attempting to remind themselves how a particular process worked, or picking off where they left off from a previous session.

3.5.6 GUIs in Contrast with CLIs

Most participants acknowledged the importance of GUIs as a better paradigm for certain tasks over CLIs. Reasons given included that they were more intuitive, or worked better for very complex tasks with visual elements.

A lot of participants also described their rejection of GUIs, claiming that visual elements were gratuitous, or that they just disliked them.

3.5.7 Security from Error States

Participants’ descriptions emphasise the importance of dealing with error states within a CLI — actions were described often as destructive, and where this went wrong diagnostic elements or recovery mechanisms needed to be utilised.

Some participants stated that aspects such as destructive action verification and incorrect syntax highlighting can help prevent errors before they occur.

The crux of the desire for reversal appears to be to abort undesired states that a user finds themselves in. When asked how to reverse a particular action, Amy simply stated “snapshots” — it is unclear whether snapshots were seen as viable reversal options by other participants, however. Others stated the addition of an “undo” command would be useful, provided that the use of such a command had a transparent effect on the state of the system. Indeed, studies such as Cass et al. [2006] add force to claims that undo functionality is only effective if users are able to understand it.

Participants had various suggestions for what allowed easier diagnosis of error states — these were mostly concerned with the form and understandability of the error messages produced by the command line, and the incorporation of suggestions for consequent action. The way in which participants also discussed action history also suggests that this is another mechanism which can provide some form of diagnosis.

Some participants touched upon how safe or forgiving command line environments can be conducive to their experience as a whole — this highlights the value of security from error states. In light of this, participants also stated that critical actions within a command line made them fearful of costly incorrect actions or assumptions.

3.5.8 Miscellaneous Codes

Other miscellaneous points identified by one or two participants included considerations of computation complexity of operations, and aesthetic value of command lines.

3.5.9 Summary of Themes

In this section, a broad range of themes which exist within the data gathered have been presented. It seems that crucial aspects which relate to the broader aims of this dissertation have been explored, including artificial intelligence, knowledge-priming of actions, command action specification, efficiency of actions and conceptualisation.

3.6 Participant Review of Findings

All participants who reviewed the themes defined above agreed with the points made, and did not request that changes be made.

3.7 Discussion

To address the research questions previously set out, we can summarise the overarching sentiments of the findings of this study.

3.7.1 In what way do CLI users recognise the utility of CLI tools for solving problems?

Experts find the power and expressiveness of many CLIs particularly useful for problem solving. A common method experts use to exploit powerful expression is divide and conquer.

However, it falls upon an expert user to appraise the usefulness of particular commands in a given context or domain.

3.7.2 What characteristics of CLI tools make them helpful for solving problems?

Customisation is often a part of an expert CLI user's experience. Arguably, expert users are drawn to this due to a lack of usability features in many CLIs — the base experience of CLIs appears to be inadequate or dissatisfactory in terms of usability requirements.

In particular, the extent to which experts are able to specify their intent via the commands available is a key factor to determining how helpful they find a CLI artefact.

3.7.3 What affects the experience of CLI users when problem solving?

Experts often find it easier to solve problems when they can tackle them effectively via trial and error. However, some provision must exist for security from domain error states in order to safely perform commands speculatively.

The ability to which expert users can access operational CLI documentation is an important factor in their experience, but is arguably a flawed method for discovery. This is because the act of documentation browsing obstructs useful action, and breaks down task flow.

Experts also value being able to form mental models of the domains they work within. An explanation for this is that models of the domain assist in formulation of intent when solving a domain problem.

3.7.4 What are the characterising features of problems which users find CLIs are helpful in solving?

It seems that experts tend to utilise a CLI for two types of activity.

The first are routine tasks, which constitute well-known or over-learnt sequences of commands to fulfil workflow, maintenance or navigation goals. These are solved in a rapid, almost reflexive manner, such that the decisions which underpin their solutions are recognition-primed.

The second are novel problems, which arise from a scenario where an unknown sequence of commands must be applied to achieve a specialised goal. It may be the case that a user understands a domain solution to a problem such as this, but they do not know the specific syntax which allows them to specify their intent. These problems require a user to take a reflective or analytical approach, or to discover useful, perhaps previously unknown, syntax.

It would seem that traditional CLIs are more primed to help a user achieve their goals if the activities required fall into the first type described.

3.7.5 What do expert users value in their CLI experiences?

A fundamental desire that expert users of CLIs hold is the ability to carry out efficient action to achieve rapid task success. They frequently achieve this through heavy customisation, automation and exploitation of powerful or expressive syntax.

Another aspect that they value is agency and control over their working CLI environment. That is, the ability to successfully specify their intentions on the basis of their own decisions, but to also personalise and tailor their experiences via customisation and end-user programming.

In addition, expert users often desire security from error states, which result from issuing invalid commands, or commands which lead to domain errors. This is because many actions taken within CLIs are potentially highly destructive, or involve institutionally sensitive domain objects.

3.8 Critical Analysis

It is likely that the findings here are highly subject to researcher bias, due to the qualitative and interpretive nature of the data. Thus, findings related to the overall aims of the research, particularly those relating to themes of exploration and discovery, should not be treated in an unlimited fashion. It is indeed possible that the data gathered could be interpreted in an entirely different manner to that which is presented here under different research conditions.

These findings are also limited by the fact that only a small participant sample was taken, all of whom are computer science students at the University of Bath. This negatively affects the universality of the claims.

3.9 Summary

In this chapter, we have demonstrated an in-depth investigation into perspectives of expert users of CLIs and addressed the research questions set out.

We have also discussed factors which show that there is indeed force behind the notion of difficulties performing action specification within CLIs, the need for operational documentation checking in these contexts and also the desire to complete these processes in an efficient manner. This provides force and insight into the Exploration Problem previously set out. These ideas can be taken forward to derive a specification for a command line artefact which improves upon the vanilla experience of command line users.

Chapter 4

Designing and Implementing a CLI

In this chapter, a design-focused synthesis of an enhanced CLI for problem solving, which addresses the Exploration Problem, will be presented.

Building upon the findings from the Literature Review and the Exploratory Study, we will begin with the derivation of a design specification. Several designs will then be proposed which aim to address the interface-oriented factors of this specification. A final design will be selected, developed further, and a formal claims analysis on the enhancing features of this design will be presented. This chapter closes with some brief notes on the implementation of the final design.

4.1 Deriving a Specification

When examining the literature and the findings made at this time, one can derive a detailed specification of requirements for a CLI which addresses the Exploration Problem.

The aim of this specification is to provide a loose framework for the derivation of initial designs for an enhanced CLI. They, in effect, act as a set of design principles which broadly address the usability and usefulness requirements of CLI users.

4.1.1 Understanding Basic CLI features

It is very easy to say that CLI artefacts, by definition, have some area for textual input, and some area for textual output. However, it is pertinent to the goals of this specification to understand why archetypal CLIs fulfil this definition.

To offer more insight into this, we must refer to the fundamental principles previously discussed in the review of the literature. The CLI is an interaction paradigm which provisions vast functionality structured by a control language [Norman, 2007]. In effect, this control language could take the form of any sort of descriptive symbolic representation, but it is reasonable to propose that the most universal representation is via text.

Therefore, for the purposes of this specification, we assume that the artefacts to be designed support the entry of textual input and afford the interpretation of output via text. We can specify this further by also stating that, this notion of output is only presented as a consequence of an action of input which preceded it.

4.1.2 Problems to be Solved

It is possible to arrive at a number of broad aims which an enhanced CLI artefact must address in order to solve the Exploration Problem, or provide a better experience to a CLI user. These are explored in the context of design principles for complex problem solving proposed by Mirel [2004], and more general interface design principles proposed by Preece et al. [2007].

Promote Efficient User Action

Evidence from the literature and the investigation priorly carried out suggests that a primary goal of CLI users is the efficiency with which they can carry out useful action. This aim is, in effect, a maintenance goal of the Exploration Problem — indeed, the possibility of efficient action should be maintained/enhanced/not eliminated by an artefact which aims to solve the Exploration Problem.

This aim might broadly be fulfilled by enabling commands to be performed more rapidly, or the expression of commands to become more powerful — they achieve more work for less input. Reflexively executed commands which form the basis of routine tasks can also in fact be actively supported similarly by some means. Automation in some forms, including provision of end user programming capabilities, is a potential avenue for this.

Repetitive tasks or actions can be supported via some automatic iterative capabilities or by allowing past commands to be re-executed.

To afford cognitive and physical efficiency, a focus should also be placed upon the use of the keyboard as an input device, as a mouse or trackpad does not effectively contribute to the modality of textual input.

In addition, aspects which break task flow, such as elements which are distracting or obtrusive to a user, should be minimised; obstruction to useful action must critically be limited. That is, elements which promote other usability goals should be judiciously added as not to interfere with a user's efficiency goals.

Promote Understandability of the Domain

It is clear that the design of a CLI artefact can have some effect on the extent to which the salient aspects of a domain are made clear, though aspects of the control language provided seem to have the most effect here. Norman [1981] proposed that a principal aim of interface design in this context is to make the underlying model of the system clear.

Many participants of the Exploratory Study stated the importance of the formation and maintenance of clear underlying mental models which encapsulate domain knowledge. Indeed, an intuitive model that seems to emerge from CLI use is the divide and conquer approach to solving problems. A reasonable theory for this is that well designed commands should perform a singular or unitary function, and that these can be combined over one or many statements of action to construct the solution to a problem. This appears to be a principle aspect that should be supported in some manner through design.

In conjunction with this, efforts should also be made to increase the parsability of output text, in part to support associations made between the actions users undertake and their results, but also to stimulate better understanding of the output content.

Enable Discovery of Useful Commands

Seemingly the crux of the Exploration Problem at hand, a design must address how a user can discover a required command, where their intended action may or may not be known. This might involve some sort of parsable presentation of the commands which are possible, and the enhancement of the user's ability to appraise the usefulness of presented commands, both with respect to the context the user finds themselves in. The fundamental goal here is to reduce the time it takes for a user to translate their intentions into the control language, by limiting the need to refer to external documentation.

Another way to do this is to support the guessability of the control language, though these concerns largely lie with the conceptualisation of the domain or provision of language level constructs such as aliasing, flexible parameter orders, or simply by labelling the commands in a domain-oriented manner.

Another possible approach is to support an environment where trial and error becomes a safer activity, possibly by providing security methods such as undo or versioning.

Support Learning

Reviews of the literature have indicated that CLIs are exceptionally beginner-unfriendly, principally due to lack of visibility, but also because of the presence of a control language which must be learnt. Efforts should be made to promote the process of learning of the control language, not just via features of the language such as memorability of labels, aliases and consistency of format, but also provision of interface level features. These notions are reinforced by similar concepts explored by Norman [1981].

It is pertinent to bear in mind the insights about learning paths attained from the Exploratory Study — CLI users typically learn through their peers or in a task-driven way. Thus, we can imagine that artefacts for collaboration or crowd-sourced insights might play a role in a learnability-enhanced CLI, as well as the narrowing of guidance based on task context.

Promote Visibility

The quality of visibility of possible commands appears to be a vehicle for a potential solution to the Exploration Problem — indeed, a step towards this is in the provision of a representation of what commands are possible in a given context.

There are also several auxiliary aspects of problem solving exercises that CLI users care about which require provision of greater visibility than that which the base CLI experience provides. These include ongoing processes, side-effects of action, the task context, statuses and static information.

With respect to this, the gratuitousness of visual elements should be limited, such that the benefits of minimality that CLIs provide are preserved.

Enable Security of Action and Recoverability

Another principal desired quality is the extent to which an enhanced CLI can provide recovery or security from error states. This can be broadly achieved by ensuring users understand what constitutes a valid and invalid action so that error states are prevented, but this cannot always be achieved. That said, certain mechanisms, such as verification for particular destructive actions, or highlighting incorrect syntax, can help serve this purpose.

Highly diagnosable error messages and visibility on context and command history are mechanisms which can provide support for users who find themselves in an error state. The ultimate solution for this, however, would be to allow the transparent reversal of erroneous commands via an undo functionality, or through built-in versioning.

4.2 Formal Design Specification

By defining the problems that an enhanced CLI must solve, we are able to identify the principal aspects which allow us to formulate our design goals into a formal specification. Individual aspects will relate in varying capacities to different components of the proposed CLI artefact — syntax, interface and domain model — though they are presented here with an aim to promote largely domain agnostic ideals. Syntactic and domain model requirements will not be addressed by the proposed designs, but are included here for the purposes of completeness and further development.

The full specification is detailed in table 4.1 below.

Requirement	**Aims**	**Pertinent Components**
The artefact must provide some rapid means by which valid syntax can be browsed, limited or filtered by the problem context	Discoverability	Interface
The artefact must provide the user with a means to appraise the usefulness of particular commands given a particular problem context	Discoverability	Syntax, Interface
The artefact should allow some means by which syntax can be effectively guessed by a user given a domain context	Discoverability	Syntax, Interface, Domain
The artefact must afford some means by which actions can be taken in a low-effort manner	Efficiency	Syntax, Interface
The artefact must afford some means by which actions that allow the user to influence a significant portion of the domain objects can be performed	Efficiency	Syntax
The artefact must afford some means by which composites of multiple actions can be performed	Efficiency	Syntax
The artefact must not require the user to use a pointing device	Efficiency	Interface
The artefact should by some means limit the extent to which operational documentation must be checked	Efficiency	Interface
The artefact must provide some means which enhance a peer-driven learning process	Learning	Interface
The artefact must provide a non-intimidating new-user experience	Learning	Syntax, Interface, Domain
The artefact must provide commands with labels which are memorable	Learning	Syntax, Domain
The artefact must provide commands with domain-relevant labels	Learning	Syntax, Domain

The artefact must provide some means by which a user can reverse or recover from an erroneous domain state	Recovery	Interface, Domain
The artefact must by some means make the underlying domain model clear	Understandability	Interface
The artefact must by some means make the underlying interaction model clear	Understandability	Interface
The artefact must provide commands which perform atomic functionality	Understandability	Syntax
The artefact must present output in a clearly and in a parsable and highly visible manner	Visibility	Interface, Domain
The artefact must provide a simple method by which users can visualise their action history	Visibility	Interface

Table 4.1: Requirements specification.

4.3 Generative Discussion

In order to begin generating initial designs, it is useful to speculate about potential features of an enhanced CLI which fulfils the requirements set out above.

4.3.1 Context Awareness/Classification/Recognition

To allow the artefact to provide a relevant representation of valid commands given a context, it must be capable of intelligently analysing the history of the user's action, or indeed must be provided with some indication of the task that the user will be attempting to undertake. Indeed, many existing suggestion engines [de Arruda, 2013][Doras] perform on the basis of a user's action history. These notions are also proposed in the context of works on plan recognition such as Charniak and Goldman [1993] and Carberry [2001].

However, there is a possibility here to harness action history from other users to enable a widely distributed artefact to learn to classify context from widespread expert use. This in particular may be an especially useful mechanism in enhancing peer-driven learning.

These capabilities could be wielded to provide sorting or filtering criteria for a condensed help menu, or a list of possible commands. It is particularly important for such an artefact to be capable of filtering a list of valid commands as, given a set of domain objects, a command space could be intractably large due to combinatorics.

4.3.2 Customisability

Many expert users stated the importance of CLI customisation during the Exploratory Study. Customisation is a tool for allowing a user to design their own optimised experiences, and so may enable an enhanced CLI to fulfil certain efficiency goals. One can imagine custom aliasing in particular to be amongst a feature of this kind.

4.3.3 Alternative Layouts

In the interest of exploring an innovative CLI experience, one can imagine providing variations on the traditional layout of CLIs, particularly of output, in the interests of understandability and parsability. One such method might be the provision of a visualised graphical space, where nodes are input-output tuples, and links demonstrate reactionary ties between them, particular benefits being the flexibility of such a layout, and potential for visual grouping between input and output to strengthen a user's model of causality.

Another pertinent aspect is investigating direction of flow of action history — should this be presented left-to-right, top-to-bottom, bottom-to-top? It is unclear which direction may be optimal. Models described by Bradley [2011] seem to indicate that, in the absence of a visual hierarchy, or the abundance of text-heavy content, a reader's eye will sweep down across a page, due to "reading gravity", and in the direction of an "axis of orientation". This suggests that, if a design does not visually indicate otherwise, information should flow in a downwards fashion across a viewing window.

4.3.4 Indicators and Visualisers

It would indeed be possible to provide a more visual experience for command line users, wielding colour or iconography to express semantic content. The internal state of a particular domain environment could also be made permanently visible to the user — many participants within the Exploratory Study stated their frustrations with having to routinely perform visibility commands. This does, however, present a clear trade-off between visibility and potential obtrusiveness through visual clutter.

4.3.5 Undo Schemes

As previously discussed, a mechanism by which a user can reverse the effect of the most recently executed command will be very useful for users to ensure their security from domain error states, or erroneously executed commands. The most obvious way to provide this is via a command which performs this function, although other schemes can be imagined, such as visualised versioning systems or snapshots.

An extended notion brought up in the Exploratory Study is a tree-undo scheme, which preserves undone actions as a branching point from the current state. However, accessing and redoing separate undone states in a tree may potentially require more nuanced control, and is also of niche usefulness.

4.3.6 Queried Hints and Documentation

Another potential solution to fulfilling the discoverability goals set out is to provide a user with a mechanism or command which allows them to receive hints for further action given a particular query. This solves the problem in a simpler fashion than the so-proposed context-awareness mechanisms, and places more control over the information presented in the hands of the user. In a similar way, the discovery aims might also be solved by allowing documentation to be conveyed on the basis of queries.

4.3.7 Commitment Deferral

To allow a user to rapidly appraise the usefulness of a particular presented command, a "shadow execution" mechanism can be imagined which simulates the execution of a typed command which has not yet been executed, and presents the result to a user. This affords the benefit of allowing a user to understand the effect a command might have before committing to its execution.

4.4 Proposed Designs

In light of the speculative discussions presented, a number of proposed designs for an enhanced CLI are described formally below, along with discussion points that were generated when presenting them to participants from the Exploratory Study. These are, again, presented as divorced from a domain model or the corresponding syntax for such.

A number of sub-designs and variations will also be described and presented as variations on the overarching designs for each, as reactions to informal discussions with the participants of the Exploratory Study[1] in the role of prospective/target users. In these informal discussions, user were demonstrated how each design would work, and were asked various unstructured questions to prompt their views on the suitability of various design features, such as "How would you execute a command in this situation?" Quotes and sentiments they expressed will be presented to illustrate design issues that were encountered.

4.4.1 Context-aware Suggestions

A consistent aspect of all the designs presented is the incorporation of suggestions which are context-aware. As previously discussed, these present a stronger notion than suggestion engines provided by zsh-autosuggest [de Arruda, 2013] or Fish [Doras], which simply act as a tool for rapid history retrieval.

The way these would function is on the basis of how other users have acted or reacted in similar situations to the one the user is currently situated in. Information such as the user's command history, and the state of the environment after the last command will be used to produce one to several suggestions for useful commands. This is in the assumed practical context of using big data and machine learning methods to teach a distributed artefact to recognise domain contexts. Based on these, the artefact may judiciously or intelligently select from a combinatorial space of valid commands to present a short list of suggestions.

These suggestions are proposed to serve multiple functions: primarily as a peer-driven learning tool, but also to promote the discovery of new commands which a user may find useful. Suggestions are not proposed as a tool for entirely automating the process of problem solving — thus the intention is to deliberately provide fuzzy suggestions which the user has agency to choose from.

User Discussion

Review participants had positive comments about the notion of command suggestions — that they would be "good for a new area" and useful for "recovering from error states" — but had some criticisms.

Many questioned the helpfulness of the suggestions in the context of other methods of discovery, or where suggestions produced common command aliases used by others which were not assigned on a particular user's machine, and vice-versa:

> "Would other user's commands be helpful? Not the same tasks as my task runner..."

[1]And two others, Computer Science students from the University of Bath with CLI experience.

One participant stated that they might be more helpful if they took the form of a "plugin based architecture that could add specialised suggestions" based on a particular domain. It seems that there are technical aspects to the implementation of such a feature which extend past the scope of this dissertation.

4.4.2 Enhanced Traditional CLI Prototype (v1.0)

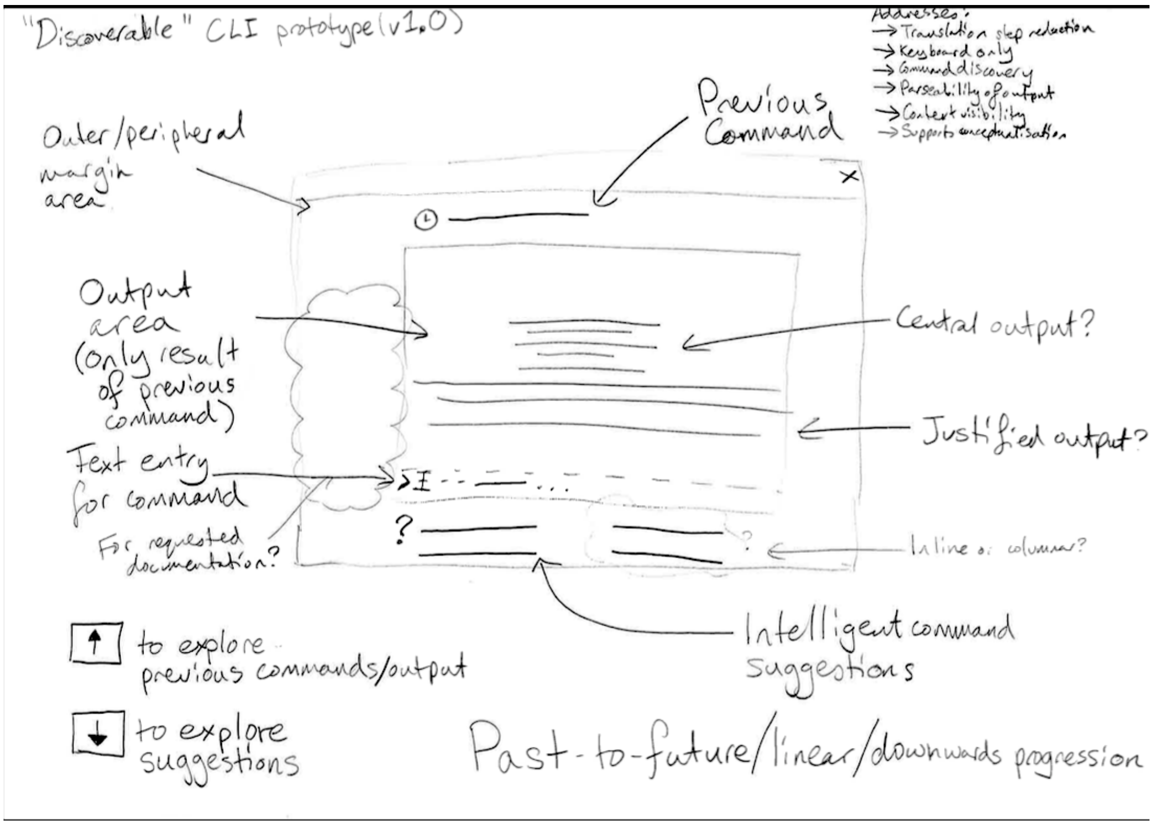

"Discoverable" CLI prototype (v1.0)
Addresses:
→ Translation step reduction
→ Keyboard only
→ Command discovery
→ Parseability of output
→ Context visibility
→ Supports conceptualisation
Previous Command
Outer/peripheral margin area
Output area (Only result of previous command)
Central output?
Justified output?
Text entry for command
For requested documentation?
In line or column?
Intelligent command suggestions
to explore previous commands/output
to explore suggestions
Past-to-future/linear/downwards progression

Figure 4.1: V1.0 introduces a paradigm focused on discovery.

This design alters the archetypal layout of traditional CLIs by adding a wide margin around the operating area. This does two things: the output of the previous command has a narrower column width, which is known to aid legibility [Dyson and Haselgrove, 2001], but it also allows peripheral information to be shown to the user.

The previous command is shown within the top margin, and only the result of that previous command is shown in the output area, with the input area shown below that as usual. It may be hypothesised that this aids conceptualisation by allowing results to appear in a consistent place. It may also provide extra focus on the most recent information.

In order to access history further back, the user may press the up arrow, which will change a user's "position" in the history, the output area changing to reflect the output of the previous command. This will also allow a user to obtain a historic command for re-executing by automatically filling the input area with the previous command.

Below the input area lie context-aware suggestions. These can be explored by pressing the down arrow, which will automatically fill the input area. Another core idea behind this design is idea of a conceptual flow from future-to-past as the information moves from bottom-to-top. It can be hypothesised that this will promote a spatial association of linear progression to the upper and lower parts of the screen.

Initial Criticisms

The problem with this design is that it potentially obscures too much history and context — this is pertinent when considering the importance of action history to CLI users (see section 2.4.8). What if a user wished to make a decision on the basis of multiple past commands? It is not clear how far back in the history of a user's action contextual information starts becoming non-useful — perhaps even the idea of a limit is not meaningful.

In addition, the actions of contextual history searching and obtaining previous commands for re-execution are conflated into one action. What if a user wishes to see the context of their actions without re-executing the command that produced them?

Greater History Visibility

The design was altered to remove the restrictive view on the output content, and now shows as much history as possible — however it now visibly presents the inputs as paired to the output they produced. One can argue that this visual association may help users identify required context, but also assist learning efforts by building upon the user's mental associations of a command and their expectation of the output.

These are traversed in much the same way as before — thus exhibiting the same problems as the previous iteration.

Multipurpose Help Region

The altered design also diversifies the purpose of the suggestion region. When the user has not typed a command, the region will present context-sensitive command suggestions, which can be navigated to by pressing the down arrow. However, when the user types or selects a command, the suggestions will disappear, and the area will display brief/summarised

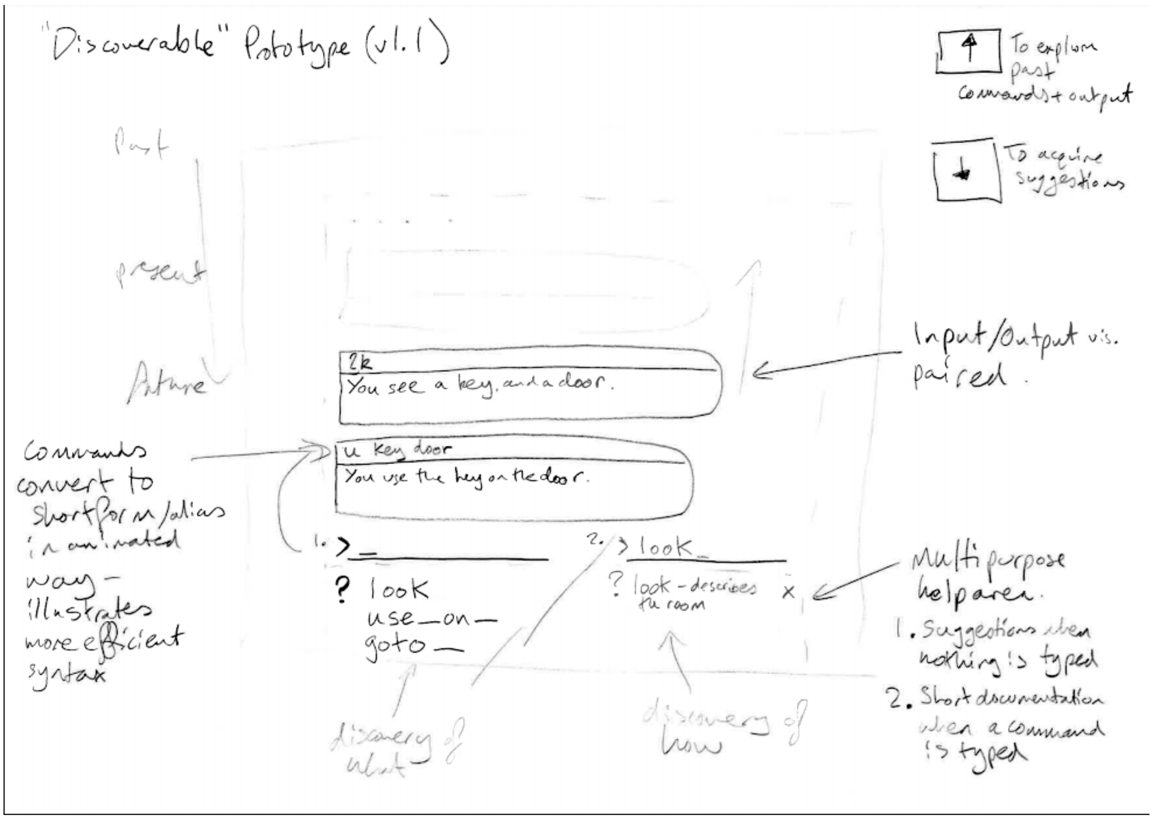

"Discoverable" Prototype (v1.1)
To explore past commands + output
To acquire suggestions
Past
present
future
?k
You see a key, and a door.
u key door
You use the key on the door.
Input/Output vis. paired.
Commands convert to shortform/alias in animated way – illustrates more efficient syntax
1. >_
? look
use_on_
goto_
2. > look_
? look - describes the room
x
Multipurpose helparea.
1. Suggestions when nothing is typed
2. Short documentation when a command is typed
discovery of what
discovery of how

Figure 4.2: V1.1 introduces paired input-output tuples, multipurpose help region, and syntax transformation.

documentation relating to the commands in the input area. These are provided to help a user remember the details of a particular command, or discover how a new command works or what function it performs.

This diversity works well in theory, because it addresses two facets of the Exploration Problem directly — allowing a user to discover what there is, and discover how it works. There may however be a problematic case where users select a command which they think they might need, but see from the presented documentation that this was not what they were looking for. Frustrating and time-consuming, perhaps, but an error state has been prevented.

Syntax Transformation

This iteration also adds a feature whereby commands which are typed using long-form aliases are transformed on execution to their shorter counterpart alias. The idea here is to make more efficient aliases more visible to a user, and potentially help them build semantic associations between short and long form aliases for commands.

This transformation might be made more obvious via the use of animations or visual indicators such as colouration.

User Discussion

Review participants had some problems with the way arrow keys we proposed to be used:

> "Wouldn't reach for the down arrow, not my expectation — used to tab completions"

Another suggested that arrow keys could become conflated for suggestion and history purposes.

There were also mixed opinions on the visual pairing of inputs and outputs. Some review participants stated that such visual elements were potentially unhelpful and not necessary, due to reasons such as particularly long output lines and screen-clearing habits. One target user even stated the following:

> "The majority of my commands are more interesting than the output of them"

However, they also stated that they had "never realised this was a problem" and that they were often confused where an output ended or began. Another stated that they would enjoy the ability to pin a particular input-output tuple to maintain the context of particularly useful outputs.

Review participants also suggested the incorporation of more control over suggestions and help areas, such as special command keys or clickable elements to show and hide them.

4.4.3 Spacial Exploration CLI Prototype (v2.0)

This design visualises command line interactions as a directed graph environment/space of input-output tuples. It can be hypothesised that visualising an interaction sequence as a graphical space can promote the conceptualisation of one's exploratory activities within a solution space or a problem solving flow. The tuples allow strong associations to be made between an input command and the result it produced, while making clear the flow that is produced by these interactions. This is intended to strengthen the visibility of cause and effect of commands to promote learning.

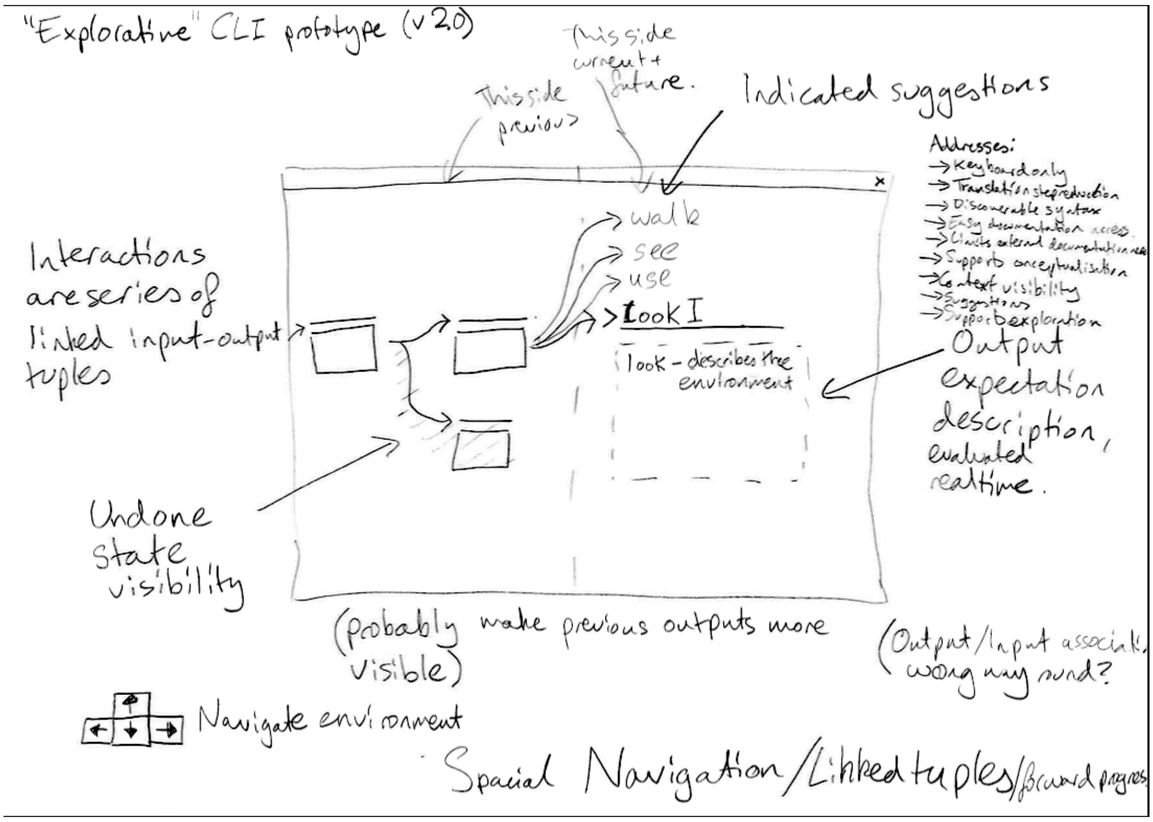

"Explorative" CLI prototype (v2.0)
This side previous
This side current + future.
Indicated suggestions
Addresses:
→ Keyboard only
→ Translation step reduction
→ Discoverable syntax
→ Easy documentation access
→ Limits external documentation need
→ Supports conceptualisation
→ Context visibility
→ Suggestions
→ Supports exploration
walk
see
use
>look I
look - describes the environment
Interactions are series of linked input-output tuples
Output expectation description, evaluated realtime.
Undone state visibility
(probably make previous outputs more visible)
(Output/Input association wrong way round?
Navigate environment
Spacial Navigation/Linked tuples/Forward progress

Figure 4.3: V2.0 introduces a paradigm focused on exploration.

The screen is divided into two: the left side for historic/contextual information and the right side for input or future action. It can be hypothesised that the left-to-right order is readily associated with past-to-future chronology, at least amongst English speakers.

On the left, tuples are presented in some way in which relevant context and previous action can be clearly seen. They are visibly linked by strands or arrows, which themselves create an association of how the user reacted to the previous output.

On the right, a working area is presented. This constitutes an input area, but also a few other enhancing features. Contextual suggestions, as well as the input area, are shown as being attached by strands to the most recent output to signify that they represent possible interpretations of the output. Underneath the input area, there is a space for "expected effect/output", which will display short explanations of what the user can expect the various commands they have typed to do. As well as providing consistency of the input-output tuple scheme, this area is provided with the intention to promote better visibility of in-application documentation in a succinct way.

The graph will also provide visibility on lost undo states as alternative interpretations of a previous output state. These will be slightly greyed out or blurred to show that, though they no longer apply, they are still accessible.

Navigation of the space is done by means of the arrow keys, shifting the focus spatially between various elements, including suggestions and past input-output tuples. This will allow the user to auto-fill the input field with suggestions and past commands.

Initial Criticisms

One potential issue with the visual aspect of this design is that commands and their outputs, as lines of text, are typically horizontal/wide in nature — this limits how they can be usefully visualised as nodes in a graph. There may be a way to symbolically summarise a previous command and output such that it becomes easier to represent in a more rounded fashion, although there is nothing to suggest such a representation would work or be beneficial.

It is also the case that this proposal may be too radically different to a traditional CLI experience to be useful for expert users — obtuse visualisation is amongst the candidates of these, though it is unclear how users might actually react.

This design also exhibits a similar problem to the first design in that it does not distinguish exploration and acquisition of previous commands for re-execution in the use of arrow keys for exploration. This is a problem of conflating meta-navigation with command acquisition or alteration.

Flipped Tuples

This idea was also explored with a slight alteration — the swapping of output and input within tuples. Each node is now an output along with the input command that the user previously reacted with.

This puts more emphasis on the state of the environment as a node, as it visibly appears first, and can appear more prominently above the input field as part of the tuple model. The links between the nodes now demonstrate the result of an input command.

It is unclear whether this is more or less intuitive than the first iteration. However, this makes it more difficult to present candidate or undone commands due to the altered meaning of links.

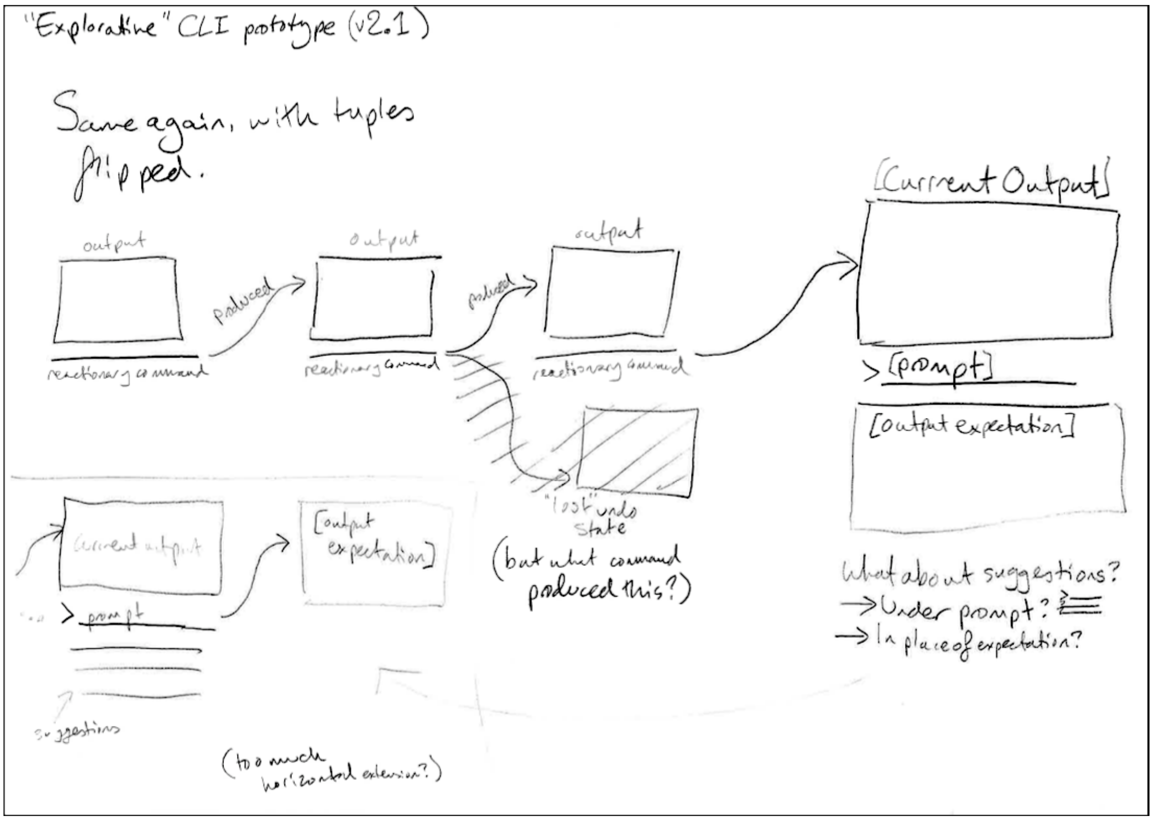


Figure 4.4: V2.1 presents an alternative method of pairing inputs and outputs.

Altered Layout and Control Scheme

In order to make better use of the space, the design was re-imagined again in a layout which promoted better use of screen real-estate with respect to the wide nature of inputs and outputs, whilst maintaining a logical flow from left to right.

Controls were also altered to make the distinction between historic command re-triggering and history search more pronounced. Users must now press tab to switch between history and input, and navigation of the space via arrow keys is broadly the same — however, when focusing historic commands, the user should press enter to auto-fill the command into their input area.

It is unclear whether this arguably more fiddly control scheme provides a benefit of finer control, or will simply frustrate or obstruct users.

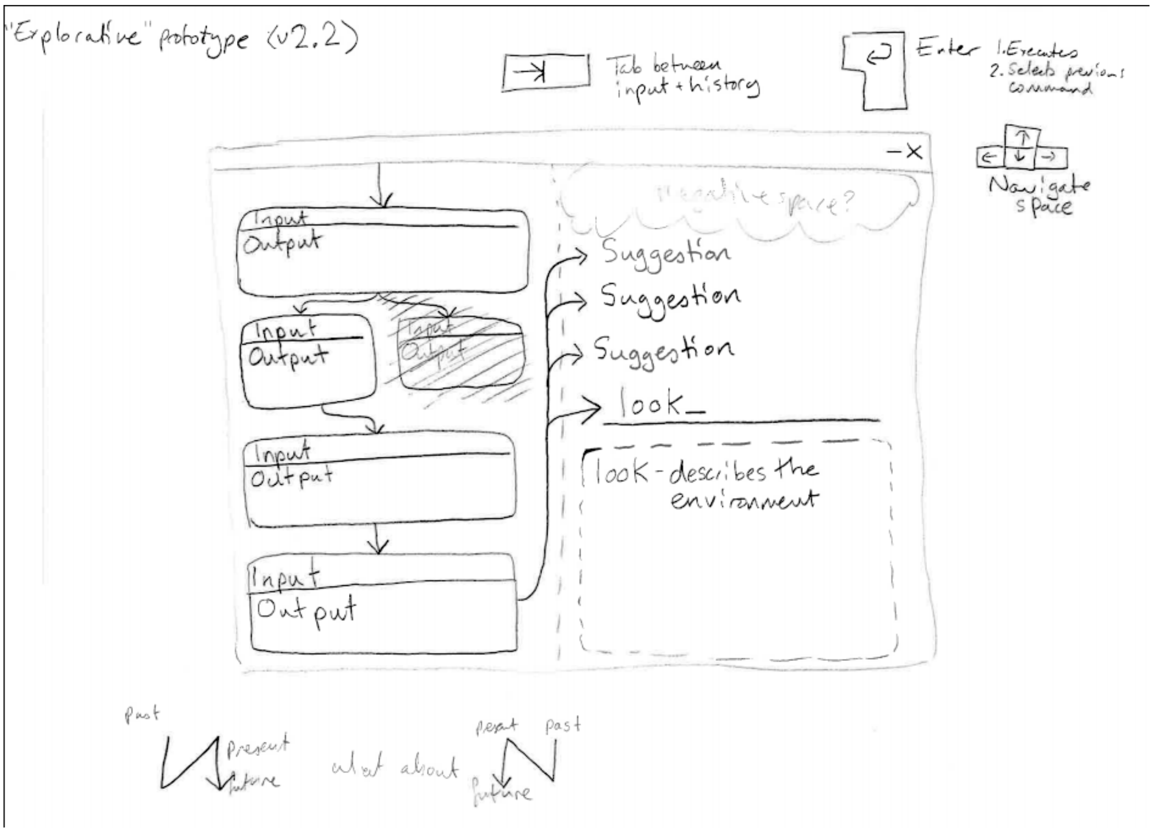


Figure 4.5: V2.2 presents an attempt to create a more meaningful and structured variant of the layout than previous iterations.

User Discussion

Many review participants were wary of the first iteration of the design due to its unfamiliar nature — one user commented that they "wouldn't consider this a command line at all". However, many were still interested in exploring the ideas presented. One such user suggested that this type of scheme would likely be "more natural or nicer for new users", but that their "command line knowledge" would be almost "inapplicable" to such an interface.

Review participants also expressed their confusions with the various proposed control schemes — for instance, one user stated that they would not be sure whether the left arrow key would get a previous command, perform an undo, or scroll through a typed command. Indeed, another stated that their "intuitions about arrow keys" were "messed around" by such a scheme. One participant also stated that they would feel quite natural using a mouse for certain interactions, like undoing graphically to a specific level of history.

Participants stated that the positioning of the history also adds a certain complexity — it was difficult to reach a consensus on what direction would be suitable for the history to flow in.

When presented with the second iteration, one participant stated that they would enjoy being able to toggle between the different representations shown in each iteration, stating that "they would be good in different situations".

One participant also stated, when presented with the third iteration, that they did not like how suggestions were presented, and that they were intrusive. They stated that this was because they were above where the user types, took up a lot of space, and that a mechanism such as auto-fill within the prompt for suggestions would potentially be more suitable.

Participants also stated that the layout seemed strange and cluttered to them — one such user stated that the zig-zagging nature of the flow was particularly strange.

In addition, participants expressed their concerns with the control schemes in this iteration, stating that arrow keys were "probably more intuitive" if the scheme was "better organised".

4.4.4 AI Conversation CLI Prototype (v3.0)

This prototype design exploits the descriptive and conversational nature of command line interaction by modelling the interactive process as a conversation with an AI, but is otherwise largely traditional. These notions of action via conversation are supported by works within Speech Act Theory proposed by Winograd and Flores [1987]. However, artefacts generated on the basis of a theory of performative speech are not without their implementation issues — labelling is a principle challenge highlighted by works such as Wobbrock et al. [2005].

The AI will largely simply carry out requested commands, but may also provide suggestions along with requested output. The AI might also be capable of interpreting erroneously typed commands or correcting them.

This design is intended to emulate the peer-driven learning paths of command line users by providing an always-present peer to gain insight from.

Initial Criticisms

It is clear that this design would likely be unsuitable for most expert users, who are likely to take the view that the design is gimmicky, obtrusive or "hand-holdy". Similarities can be drawn between this proposal and early versions of Microsoft Office's AI assistant, "Clippit"

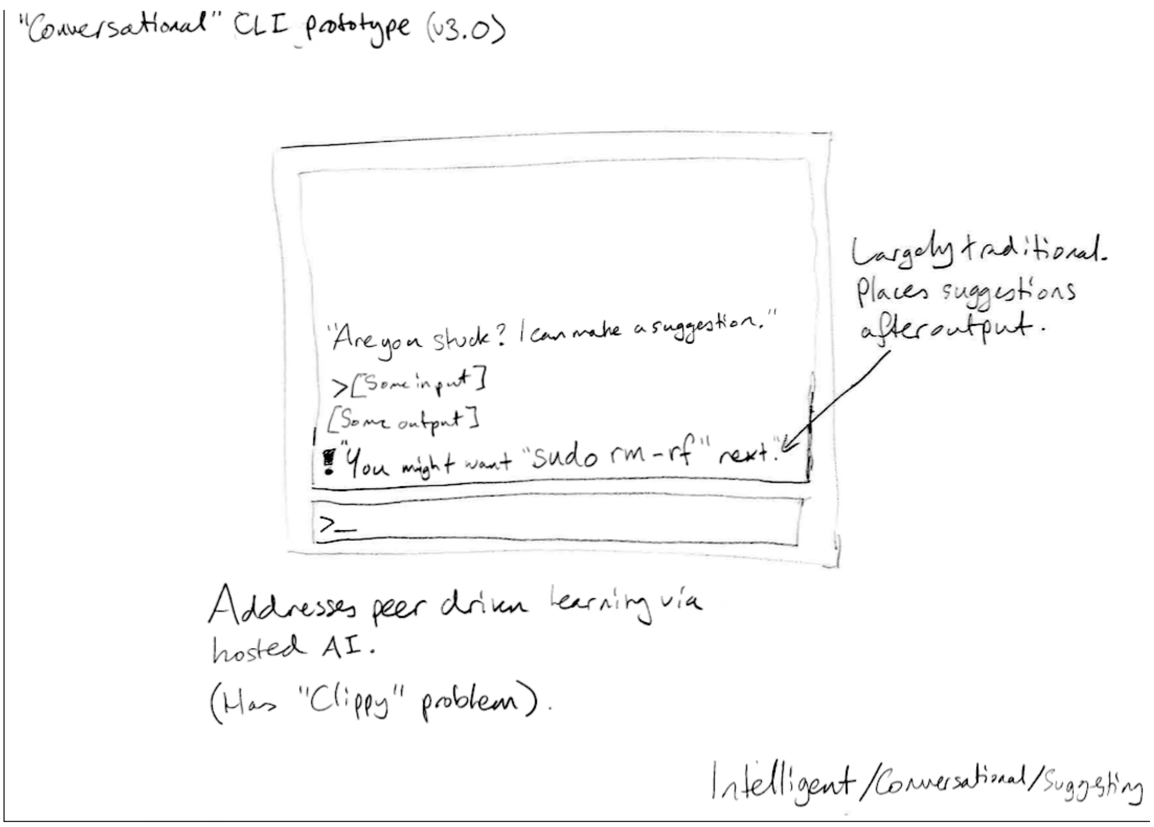

"Conversational" CLI prototype (v3.0)
"Are you stuck? I can make a suggestion."
>[Some input]
[Some output]
! "You might want "sudo rm -rf" next."
>_
Largely traditional. Places suggestions after output.
Addresses peer driven learning via hosted AI.
(Has "Clippy" problem).
Intelligent/Conversational/Suggesting

Figure 4.6: V3.0 presents an interactive conversation with a hosted artificial intelligence.

(better known as "Clippy"), a notorious usability failure [Swartz, 2003].

Another issue is trust: how can a user trust an intelligent artefact with agency to carry out their requests correctly or without alterations? Command line users are likely to want as much control and agency as possible when using what should simply be a tool — an extension of their own will.

User Discussion

Most users saw merit in the motivations and potential of this design, one stating that they would enjoy it if the agent "figured out when I'm lost or don't know what to do", or notified them of what they might have meant when an invalid command is issued. Another stated that in such scenarios, "it'd be easier than googling" a help query.

Review participants also, however, were unsure if such an artefact would feel intrusive, and one user was particularly concerned about their feelings of agency and control over critical tasks:

> "I don't want an extra interpretation layer — if something goes wrong I want it to be my fault"

4.5 Final Design

In accordance with feedback, it would appear that the most suitable choice of design for implementation, and thus further investigation, will be v1.1, with some alterations that allow support of desirable features from other designs. This design is arguably the most suitable because it addresses the specification whilst maintaining some traditional command line intuitions.

The design will largely remain as previously described, but with alterations to support the visibility on lost undo states. This is a response to positive feedback on this feature within the v2.x designs.

This will be visualised in the same way as shown in v2.2, where undo states will be shown in parallel to the resultant state, split in half, and on the right side of the output area. The arrow keys can be used to navigate these alternative states.

The design was also altered to incorporate colour in the borders of input-output tuples to convey the semantic nature of the output — green for a successful command, red for a syntax error, and blue for meta-information.

The principle features of this final iteration can be summarised as such:

- Context Aware Suggestions — command suggestions visually provisioned on the basis of previous interaction, according to a model trained from many users.
- Tree Undo — the provisioning of support for undo, and the hierarchical visualisation of undone states.
- Colour-coded Output — colouration of output frames such that the semantic nature of the content is conveyed.
- Visually Paired Input-Output — visual grouping of inputs and their resultant outputs within individual frames.
- Help Snippets — lines from operational documentation, queried by the currently entered command.

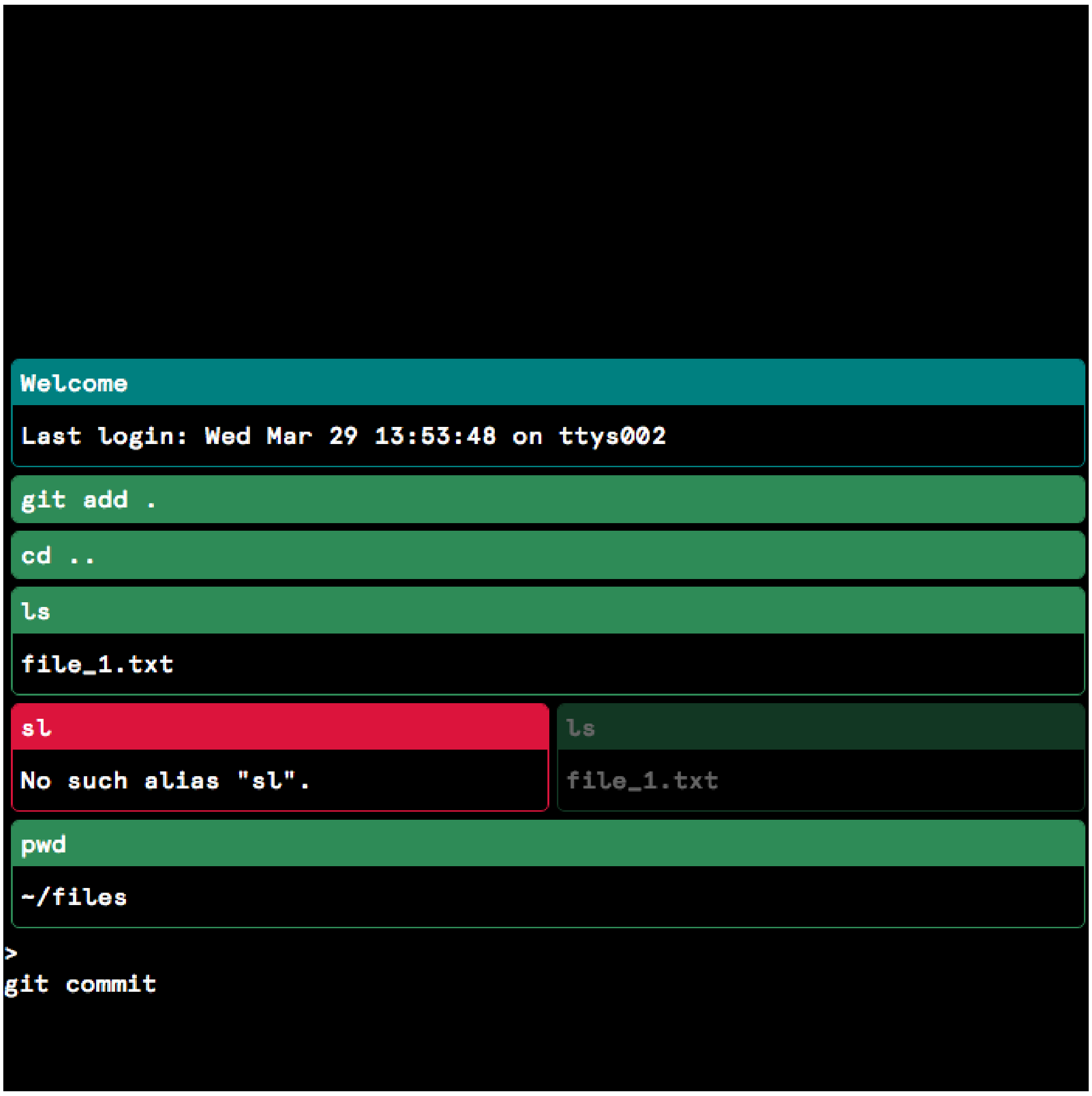
- Syntax Transformation — the transformation of command strings to use more concise aliases upon execution.

Figure 4.7 shows a high-fidelity mock-up of the design created in HTML5.

4.5.1 Claims Analysis

It is now possible to present a structured design rationale of the proposal, and evaluate to what extent its features address the goals set out in the specification, as well as where we expect the design to have a negative impact. To enrich these claims past the initial premises of the specification, the features are also evaluated in the context of Norman's Stages of Action.

The results of this analysis are detailed in table 4.2, showing the main areas of confidence in the benefits of the proposed features.



```
Welcome
Last login: Wed Mar 29 13:53:48 on ttys002
git add .
cd ..
ls
file_1.txt
sl                                  ls
No such alias "sl".                 file_1.txt
pwd
~/files
>
git commit
```

Figure 4.7: A high-fidelity mock-up of the final design, created in HTML5. "`git commit`" is a suggestion which appears as a result of the user previously inputting "`git add .`". The second "`ls`" command is undone, and is thus set aside and its opacity is reduced.

Goal/Intention Implications	**Specification/Execution Implications**	**Perception Implications**
Context Aware Suggestions + Enhances a peer-oriented learning process + Provide users with increased awareness of possible action + Provides a means by which valid syntax can be browsed, filtered by the context of use + Provides user with explicit selection of potentially appropriate actions + Provides user with an opportunity to evaluate more advanced command sequences - May disrupt flow by forcing users to re-evaluate their intentions **Tree Undo** + May encourage users to formulate more risky intentions, which broadens their experience **Colour-coded Output** + Helps a user to identify inappropriate action	**Context Aware Suggestions** + Limits the extent to which operational documentation must be checked when deriving a command - May execute a command without understanding its effect, accidentally or otherwise **Help Snippets** + Limits the extent to which operational documentation checking interrupts execution - May distract from command specification process **Keyboard Control** + Allows selectable features to be accessed rapidly + Allows actions to be executed in a physically low effort manner - Rapid access to features makes slips more likely	**Help Snippets** + Provide a means to appraise the usefulness of particular commands **Syntax Transformation** + Increases visibility of more efficient command aliases - May cause user to disassociate their action from the tuple displayed **Tree Undo** + Provides a comprehensive view on a user's action history **Visually Paired Input-Output** + Clarifies the underlying interaction model of the interface + Allows users to focus on pertinent output **Colour-coded Output** + Assists a user to identify an error state at the point of occurrence

Table 4.2: Claims analysis structured by Norman's Human Action Cycle [Norman, 1988].

4.6 Implementation

The designed interface was subsequently implemented using a boilerplate project (courtesy of `https://github.com/myurasov/Angular-ES6-JSPM-Gulp`) incorporating Angular 2.0 [Google], ES6 [ECMA], JSPM [Bedford] and Gulp [Bublitz] web frameworks, and a back-end service was created in the Spring [Pivotal] J2EE [Oracle] framework. These are connected via a websocket connection. The code written can be found in the digital appendices.

The interface constitutes a drivable viewmodel of all the features detailed. The server is capable of responding to the client interface when a command is submitted, but at this point in the investigation, no domain model has been created. Indeed, more implementation work is to be done on integrating a domain which addresses the aims of the study. Thus, each feature was tested through test suite data.

Figure 4.8 demonstrates the structure of the viewmodel, and the connection between the server and UI systems.

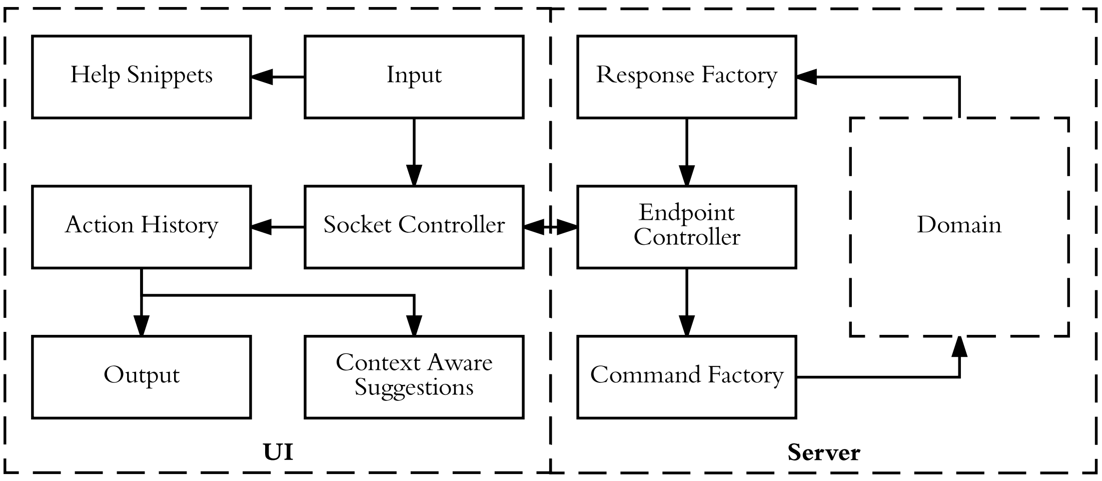


Figure 4.8: High-level architecture diagram of the integrated viewmodel, the server, and how they connect.

4.6.1 Context Aware Suggestions

Suggestions are shown on the basis of a similarity function which takes into account key words from the last output. This similarity function will provide suggestions on the basis of a model, which maps a command suggestion to a set of keywords tested by the similarity measure. The algorithm used for the similarity function is detailed below in pseudocode:

Algorithm 4.1: Context-Aware Suggestions.

```
function compareTwoStrings(situation, suggestion):
  score = 0
  for each splitWords(situation) as a:
    for each splitWords(suggestion) as b:
      if a == b:
        if a in getDisallowedWords()
        and not a in getDesiredWords(situation):
          score += getWeakScore()
        else:
          score += getStrongScore()
  return score

function makeSuggestions(items, command, history):
  lastCommand = lastItemIn(history)
  suggestions = []
  foreach items as item:
    points = compareTwoStrings(item, lastCommand)
    if points > getLowerPointsThreshold():
      appendTo(suggestions, { lastCommand, points })
  return sortBySuitability(suggestions)
```

The basis of the algorithm is in awarding a score to each suggestion in a model for how well it fits certain criterion. This is mostly defined by the rate of equivalence to a selection of words performed by the `getDesiredWords()` on a particular context object. The `splitWords` function provides a list of the words given a body of text or an object.

Some functions provide specific tuning of suggestions. The `getDisallowedWords()` function provides a list of common commands words we don't wish to match on. In particular, functions like `getWeakScore()`, `getStrongScore()` and `getLowerPointsThreshold()` allow precise numerical tuning of scores.

4.6.2 Help Snippets

Help Snippets work on a similar basis to Context Aware Suggestions, in that they are also shown on the basis of the similarity function presented. In this case, the function operates on lines from the operational documentation subject to what tokens have been typed as input. Effectively, the interface displays a sorted list of these lines based on a similarity measure to each token in the input command.

4.6.3 Output Model

The output model is structured as a tree of undo states, via recursive lists, and is updated with each new response from the server. That is, each output object has optional output text, input text, and optionally a list of output objects that came after it but were undone. The interface renders this model as shown in the mockup (figure 4.7).

4.7 Summary

In this section, we have detailed an approach to creating a CLI which addresses the Exploration Problem, and presented a design which addresses several — but not all — facets of this problem. This was achieved largely by introducing features which elaborate on traditional CLI features, such as methods of signification for afforded commands, and methods for promoting conceptualisation.

Chapter 5

Comparative Study

In this chapter, an approach to the appraisal of the features of the usability enhanced CLI detailed in the design section will be presented in the form of an experimental study of two CLI design alternatives — both of which include novel support for user interaction. The pertinent features to be examined are Context Aware Suggestions, Help Snippets and Tree Undo. We shall aim to evaluate these with respect to the facets of the Exploration Problem previously described.

We will begin by presenting the aims of this study, including several qualitative study questions which relate directly to the pertinent features of the enhanced CLI. Next, the design and implementation of an artificial domain which addresses the requirements of the study will be detailed, including the definition of tasks and specification of a control language. We will then detail hypotheses, the experimental design, the methods and the procedure undertaken. The results of the experiments undertaken will then be presented, alongside detailed statistical analyses. Finally, we will discuss the implications of these analyses and any observations taken with respect to the study questions and hypotheses derived, and draw a number of conclusions which build upon the findings of the Literature Review and the Exploratory Study.

5.1 Aims

The aims of this study are to appraise the effectiveness of Context Aware Suggestions, Help Snippets and Tree Undo for completing routine tasks and solving novel problems in a CLI environment. In addition to this, there are a number of study questions which we must aim to answer, relating to the features of the proposed design.

5.1.1 Understandability

In investigating understandability, we shall aim to determine whether a user is able to more easily comprehend or form appropriate mental models of a domain, command syntax, or the interaction model of the interface with enhancing features.

SQ1a. Do Help Snippets promote a user's understanding of commands?

We can likely expect Help Snippets to help a user to more rapidly form an understanding of the functionality of possible commands, as they more readily deliver operational documentation in a cut-down fashion.

5.1.2 Exploration

In investigating exploration, we shall aim to determine whether a user more often speculatively browses for or executes new commands with enhancing features. This is either in the context of broadening one's experience and knowledge of the command space provided, or with the aim to discover a desired command.

SQ2a. Do users explore more with Context Aware Suggestions?

We can expect users to participate more in the exploration of new commands with Context Aware Suggestions, because unlike other popular suggestion mechanisms for command lines such as those from Doras and de Arruda [2013], the user is afforded the rapid execution of commands they may never have executed before. There are several measures by which it is possible to judge the degree to which a user is engaging in exploratory activities in the context of suggestion mechanism use.

One such way is by examining how often a user engages in trial and error, speculatively selecting and executing commands in order to discover the possibility of useful action. The possibility for more rapid action is likely to enhance these processes, but also due to the increased visibility of possible commands, even in the context of potentially incomplete knowledge about the referents concerned.

SQ2b. Do users explore more with Tree Undo?

We can expect users to more often speculatively execute new commands with Tree Undo provisioned. This is based on the assumption that users will feel a heightened sense of security from erroneous or catastrophic domain states, and thus be less likely to shy away from certain types of destructive command which may permanently effect the domain environment.

5.2 Design of Study Environment

In order to address the aims of the study, we can propose the design of an experimental domain, command space and tasks. These can be used to compare different configurations of enhanced CLI features under some pertinent conditions of CLI use.

5.2.1 Domain Design

There are a number of aims which must be fulfilled when designing a domain which can support the experimental aims laid out. These include the following:

- It must propose an underlying model which is readily attainable by any participant.
- It must propose a clear aim to the participant.
- It must permit an engaging experience, such that participants care about and are invested in task success.
- It must be calculatively rational — that is, no real-time change occurs, and only the participant can affect change.
- It must provide analogies for some aspects of real-world CLI use.
- It must support routine tasks and novel problems.

Given these requirements, it is possible and indeed suitable to propose the design of a text adventure game, akin to classic titles such as Zork and Hitchhiker's Guide to the Galaxy.

Arguably, such a domain readily provisions a very clear underlying model through comprehensive grounding in physical analogies, while also being engaging and fun. Through effective scenario design, it is also arguably possible to provide an accurately analogous experience to real-world CLI use, as studies such as Heron [2015] also propose.

One way to provision a clear aim is to propose a room-escape paradigm. Such a format will conceptually place the user in a room from which they need to escape, using the objects and passages that are in the room.

5.2.2 Scenario and Action Design

Given the experimental aims, we can derive some requirements for the design of a set of scenarios to be completed by participants:

- They must provide some reasonable analogy to real-world CLI use.
- They must require the user to perform routine or maintenance tasks — that is, apply commands they use on a frequent basis, possibly to a highly consistent or over-learnt activity.
- They must require the user to solve novel problems — that is, to search for and apply commands that they may never have used before, and apply critical or analytical thinking to particular activities.
- They must be presented in a serial problem solving context — that is, the user will not need to manage multiple goals.
- They must present a problem space with one or more unrecoverable error states — that is, there are some actions which will cause them to fail.

- They must present tasks that make sense to perform iteratively to reduce cognitive or physical effort.

These requirements allow us to speculate on how such scenarios may take form, given the proposed domain environment. For instance, in one scenario, a user may have to obtain a key to unlock a locked door to escape a room. The action of picking up the key forms a routine task, but arriving at a state in which this can be done could require a user to solve a novel problem, such as figuring out how to break open a crate to reveal a key — this could require a command which was priorly never used by the user.

However, in order to derive concrete scenarios, we must first understand and specify what actions are possible in the proposed environment. The principal aim of the design of a set of actions is to provide adequate flexibility to design scenarios. Actions which affect, combine or transform domain objects in a variety of different ways should be specified.

New commands may also evolve out of a process of specifying domain objects for particular scenarios. Therefore, an iterative approach to the design of scenarios and actions will be taken. When these are specified, appropriate operational syntax can be derived to address the space of possible actions.

5.2.3 Syntax Design

In designing syntax to address the space of possible actions, several important design questions must be asked which are pertinent to the aims of the investigation.

In order to fulfil understandability goals of the experiment domain, it is clear that the derived syntax must be highly accessible — that is, participants should be able to grasp the syntax rapidly enough to usefully participate in a short-term study. It is clear that these accessibility goals can be achieved by producing aliases which constitute domain language, and that command strings can be structured like plain-english sentences.

Another interesting facet of CLI interaction is the ability to iterate over a particular set of domain objects — syntax must be provisioned to support this type of interaction.

At this juncture, we might also propose the incorporation of multiple aliases for each referent — some of which may reflect a more efficient action paradigm through truncation or using only the initial character. However, given the aims of the study, it is likely that such inclusion serves only to complicate or interfere with the findings.

Meta-commands, which provide operational functionality at the interface level rather than at the domain level, must also be included in the syntax corpus.

There is something pertinent to be said here about the use of flags in the design of command structure — some sources [Harding, 2016][Norman, 1981] criticise the use of flags, because they fundamentally change the way a particular referent functions, which causes severe overloading of functionality in large CLI-driven systems. They so occur, arguably, to fulfil efficiency aims at the expense of understandability. However, this is a sort of action specification that is particularly interesting to examine, and is provided by many real-world CLI syntax corpora. They are also likely to assist the structuring of command phrases in a plain-english fashion.

5.2.4 Final Design

Given the design aims, facets and processes detailed above, the following syntax corpus (table 5.1) and domain scenarios (table 5.2) were derived.

Command	Description of function
`look`	Displays a list of objects in the room
`look (object)`	Displays information about an object's internal state
`look inventory`	Displays a list of objects that are in the inventory
`use (object)`	Uses a standalone-usable object (e.g. doors)
`use (object) on (target)`	Catch-all for applying the object to the target in some way
`pickup (object)`	Puts an item in the inventory, removing it from the room
`every (query)`	Searches the room for all objects with names matching the query (used in the context of other commands)
`place (object) on (target)`	Places or attaches an object to the target in some manner
`break (object)`	Destroys the object, removing it from the room
`open (object)`	Opens a container of some kind (e.g. crates)
`help`	Shows operational documentation
`undo`	Reverses the effect of the last command
`start`	Moves to the next scenario (can be used to skip a scenario)

Table 5.1: The syntax corpus for the experiment.

Name	Task features	Description	Ideal Solution Sequence

Keydoor	Routine	The room contains a key and a door. The participant must pick up the key and use it on the door, and then use the door to escape.	`pickup key`
			`use key on door`
			`use door`

Clothkey	Routine	The room contains a key obscured by a cloth, and a door. The participant must pick up the cloth, and then the key in order to unlock the door and escape.	`pickup cloth`
			`pickup key`
			`use key on door`
			`use door`

Pailtube	Routine	The room contains a pail full of water, a key with a float in a tube, and a door. The participant must empty the pail into the tube in order to pick up the key and escape via the door.	`use pail on tube`
			`pickup key`
			`use key on door`
			`use door`

Snowglobe	Novel	The room contains a snowglobe which has a key in it, and a door. The participant must break the snowglobe to obtain the key and escape via the door.	`break snowglobe`
			`pickup key`
			`use key on door`
			`use door`

Pressurepad	Novel	The room contains a pressure pad, a weight, and a door. The participant must place the weight on the pressure pad to unlock the door.	`place weight on floorpad`
			`use door`

Weightgate	Novel, Iterable	The room contains a portcullis, some weights, and a chain connected by a series of pulleys to the portcullis. The participant must place the weights on the chain to open the portcullis and escape.	`place every weight on chain`
			`use portcullis`

Watertube	Novel, Iterable	The room contains a tube half filled with water containing a key, some stones, and a door. The participant must use the stones to displace the water inside the tube to reach the key and unlock the door.	`place every stone in tube`
			`pickup key`
			`use key on door`
			`use door`

Ropeweight	Novel, Iterable, Failable	The room contains a portcullis, some weights, and a rope connected by a series of pulleys to the portcullis. The participant must place some weights on the chain to open the portcullis and escape. If the participant places more than a certain number of weights on the rope, the rope will snap, and they become stuck.	`place weight on rope`
			`place weight on rope`
			`place weight on rope`
			`place weight on rope`
			`use portcullis`

Ladderhatch	Routine	The room contains a ladder, and a hatch. The ladder must be used on the hatch in order to climb up and escape the room.	`use ladder on hatch`

Cratekey	Novel	The room contains a crate containing a key and a door. The participant must open or break the crate to reveal the key, and use the key on the door to escape.	`open crate`
			`pickup key`
			`use key on door`
			`use door`

Hairdrier	Routine, Failable	The room contains a hairdrier, a door, and a key encased in a block of ice. The participant must use the hairdrier on the ice block to obtain the key and escape the room via the door. If the participant breaks the ice, the key will snap and they will become stuck.	`use hairdrier on iceblock`
			`pickup key`
			`use key on door`
			`use door`

Cratetrapdoor	Novel	The room contains a crate which is obscuring a trapdoor. The participant must break the crate to reveal the trapdoor, and use the trapdoor to escape.	`break crate`
			`use trapdoor`

Manyice	Routine, Iterable, Failable	The room contains a door, a hairdrier, and several blocks of ice encasing stones, but one encases a key instead. The user must use the hairdrier on the ice block which contains the key, and use it on the door to escape. If the participant breaks the ice block containing the key, the key will snap, and they will become stuck	`use hairdrier on every iceblock`
			`pickup key`
			`use key on door`
			`use door`

Nestedcrates	Novel, Iterable	The room contains a door, and a crate, which in turn contains a smaller crate, which in turn also contains a smaller crate, which contains a key. The user must break each crate until they discover the key, and use it on the door to escape.	`break crate`
			`break crate`
			`break crate`
			`pickup key`
			`use key on door`
			`use door`

Table 5.2: Detail on the scenarios designed, which will form the essential participant activities of the experiment. Novel task features are ones which require not-often used commands, and routine task features are those that require frequently used commands.

These can be taken forward and integrated with the enhanced CLI previously created.

5.2.5 Implementation

The syntax, domain objects and scenarios for the game were integrated into the Spring [Pivotal] application previously detailed. To provide a broad explanation, the domain object classes are defined on the basis of command interfaces, such as `IBreakable` or `IPlaceableTarget`[1] — these provide the basis for the interaction and manipulation of these domain objects via generated command objects. Each scenario is simply a room definition, which creates a `Room` object with a set of domain objects.

The source code for the game can be found in the digital appendices, alongside a demonstration video of use.

Context Aware Suggestions

A very rudimentary model for Context Aware Suggestions was hand-trained — designed and programmed by hand — based on the ideal action sequences of the scenarios detailed. A JSON representation of this model can be found in the appendices.

Help Snippets

Help Snippets uses a list of lines from the output produced the `help` command (see figure 5.1).

Undo Tree Models

While the output model within the UI needs to represent the whole action history as a tree, the internal output model on the server only needs to represent a list of command objects that were previously executed. This is because the server only needs to know the previous command object when an undo command is sent, which gets reversed thrown away. If a command is selected and redone on the UI, the server simply re-executes the command. However, these server-side command objects must hold on to the references of the domain objects they altered for the undo operations to correctly work.

5.3 Experiment Design

In order to investigate the effectiveness of certain enhancing features, two different variations of the enhanced CLI shall be defined.

[1]For example, the `Door` class is an `IStandaloneUsable` and an `IUsableTarget`

```
look - describes the room
look <object> - describes the object
look inventory - shows what objects you are holding
use <object> - performs some action with the object
use <object> on <target> - applies the object to the target in some way
pickup <object> - puts an object in your inventory
every <query> - searches the room for all objects with names matching the
query (used in the context of other commands)
place <object> on <target> - places or attaches an object to the target in
some manner
break <object> - attempts to destroy the object
open <object> - opens a container of some kind
help - shows this list
undo - reverses the effect of the last command
```

Figure 5.1: Output of the `help` command.

The first, "CAS+HS+TU", provides the user with Context Aware Suggestions, Help Snippets and Tree Undo as enhancing features. The second, "TU", provides the user with only Tree Undo as an enhancing feature. These can be provided by integrating two versions of the viewmodel — one as previously described, and another which removes Context Aware Suggestions and Help Snippets — and serving them at two different `localhost` locations.

Various hypotheses, derived on the basis of the broader aims of the study, will be investigated via the differentiation of performance metrics and observations between these two interfaces. Therefore, a 2×2 mixed factorial design was used in order to provide necessary control whilst maximising the data which can be obtained from each participant.

The scenarios presented were split into two sets, which will be respectively completed by each participant with each interface. That is, a participant will perform both task sets, and each task set will be performed with a different interface. Presentation order of task set and interface was controlled to address the impact of practice on user performance.

The task sets were designed such that their presentation order had a low impact, and that they do not provide a fundamentally different experience to one another. This was done by attempting to get the same proportion of routine and novel tasks, the same number of failable tasks, and the same number of iterable tasks. We also wish to increase the difficulty of scenarios over time in order to manage how well a user can cope with scenarios as they learn more about the system, but also try to decrease the proximity of similar tasks such that the solutions to certain problems are not highly primed from a previous scenario.

Task set 1	Task set 2
Clothkey	Ladderhatch
Pailtube	Cratekey
Snowglobe	Hairdrier
Pressurepad	Cratetrapdoor
Weightgate	Manyice
Watertube	Nestedcrates
Ropeweight	

Table 5.3: Scenarios split up into ordered task sets.

	CAS+HS+TU presented first	TU presented first
Task set 1 first	Group A	Group B
Task set 2 first	Group C	Group D

Table 5.4: A mixed-factorial design to control presentation orders of task sets and interface, and interaction effects between interface and task sets.

Table 5.3 shows the designed task sets and table 5.4 demonstrates the group design.

5.3.1 Metrics

A framework for automatically collecting data about a participant's session of use was created. With this, various metrics can be taken to help analyse the performance of users under different scenarios within different interfaces. Table 5.5 details the metrics that the framework was configured to take for each scenario the participant completes.

5.3.2 Observation Classes

In order to accurately report on pertinent aspects of participant performance, it is necessary to define classes of observation. These serve to narrow down the events which the observer must look out for during the experiment, increasing the clarity and salience of the findings.

The classes are detailed in table 5.6.

Metric Id	Type	Description
UNDONE	Frequency	The number of times a participant executes an undo command
INTENT ACT	Duration List	A list of durations between each time a participant presses enter
TIME	Duration	The time between the start of the task and the task's success
SUCCESS	Boolean	True if the participant completed the task, false if the participant skipped the task
HELP	Frequency	The number of times a participant executes a help command
ENTERS	Frequency	The number of times a participant presses enter
CHARS	Frequency	The number of times a participant presses a character key
ARROWS	Frequency	The number of times a participant presses the up or down arrow keys
INVALID	Frequency	The number of times a participant executes an invalid command

Table 5.5: The metrics which are automatically measured for each scenario a participant attempts.

Observation Id	Event Signifiers	Facets to report on
CONFUSION	The participant makes an expression of confusion in reaction to a particular event	What event caused the confusion?
RESOLUTION	The participant makes an expression of realisation or understanding when priorly they were confused	Was their understanding the result of an aspect of the interface?
UNKNOWN INTENT	The participant seems not to know what they want to do next or spends a significant time not performing an action, or attempting random actions	What juncture in the scenario is the participant at?
EXPLORE	The participant attempts to find out how to perform a desired action	What mechanism do they use to do so?
SPECULATE	The participant attempts to use a command they have not used before	What command was it?
DISCOVER	The participant finds a desired command and executes it successfully	What mechanism do they use to do so? What command was it?

Table 5.6: The principle observations that will be reported on during the experiment.

5.3.3 Participants

16 students of technical courses at the University of Bath were recruited as participants of the study, each of whom had at least some prior experience of CLI use.

5.4 Hypotheses

Given these metrics and observation classes, and in the context of the design rationale previously discussed and the study aims, we can derive several hypotheses for the performance and suitability of the features of the enhanced CLI created, structured by some of the broader aims of the investigation.

5.4.1 Task Success

In investigating task success, we shall aim to determine whether a user is able to complete tasks faster or more effectively with enhancing features.

H1a. Users achieve a better task success rate with Context Aware Suggestions

We can expect participants to more often succeed at tasks — that is, they are less likely to give up under time pressure — with Context Aware Suggestions. This is because they deliver potential solutions to sub-problems, which, if noticed by the user, will more often guide them to success.

H1b. Users will complete tasks in a shorter time with Context Aware Suggestions

We can expect users to achieve more rapid success of tasks with Context Aware Suggestions and Help Snippets. This is because they provide a rapid means by which potential solutions can be selected and executed.

5.4.2 Efficiency

In investigating efficiency, we shall aim to determine whether a user requires less physical and cognitive effort to complete a task with enhancing features.

H2a. Users will use fewer keypresses to succeed a task with Context Aware Suggestions

We can expect users to achieve task success in fewer keypresses with Context Aware Suggestions. This is based on the assumption that the number of times a user will have to press an arrow key to select a suggestion will not frequently exceed the characters required to type a desired command.

5.4.3 Discovery

In investigating discovery, we shall aim to determine whether a user more often discovers the commands required to complete a certain task with enhancing features.

H3a. Users will discover commands faster with Context Aware Suggestions

We can expect users to discover a required command faster with Context Aware Suggestions based on the assumption that, by design, they are able to deliver probabilistically suitable commands for a domain context.

5.4.4 Documentation Checking

In investigating documentation checking, we shall aim to determine whether a user checks operational documentation less often with enhancing features.

H4a. Users will check documentation less with Context Aware Suggestions and Help Snippets

We can expect users to check operational documentation less with Context Aware Suggestions, because they promote the visibility of commands that users might otherwise search for within documentation. We can also expect this of Help Snippets, as lines from operational documentation are, by design, made more visible with them.

5.4.5 Security from Error States

In investigating security from error states, we shall aim to determine the extent to which a user is able to recover from or avoid domain error states with enhancing features.

H5a. Users will execute fewer invalid commands with Context Aware Suggestions and Help Snippets

We can expect users to less often execute invalid commands with Context Aware Suggestions, because they are more likely to use commands which are definitively valid from suggestions. We can also expect this of Help Snippets, as they provision users with critical information about correct usage before they execute a command, which may lead them to the alteration or aversion of a potentially erroneous action.

H5b. Users will use Tree Undo to recover from error states

We can expect users to utilise Tree Undo functionalities to reverse a command which resulted in a domain error state, on the basis that they are confident in their understanding of how such functionality would work.

5.4.6 Intent-to-Command Translation

In investigating intent-to-command translation, we shall aim to determine the ease with which a user is able to specify their intent to action as a command with enhancing features.

H6a. Time between the end of a user's last action and the start of the next action will be reduced with Context Aware Suggestions

We can expect users to, overall, take less time to derive their intention, retrieve or derive a required command and then execute this in less time with Context Aware Suggestions. This is because they address the rapid discovery and entry of required commands.

5.5 Procedure

Given the experimental design detailed above, the procedure undertaken is presented below. This procedure was designed in accordance with the University of Bath Computer Science Ethics checklist (see form in appendices).

5.5.1 Environment

The experiment was carried out in a controlled, distraction-free environment. A laptop was provided for participants to use.

5.5.2 Consent

Participants were given a consent form before the experiment began, which detailed what the participant was be expected to do and broadly what the research is for. A copy of this consent form can be found in the appendices, and all signed forms can be found in the digital appendices.

5.5.3 Explanation

The participant was then informed that they will be playing a text adventure game, and that the aim in each scenario was to escape the room using the objects in their environment. Any terms were clarified to the participant at this point if they were unsure of what they were being asked to do.

5.5.4 Practice

Depending upon their group, the participant was presented with a particular variant of the interface, loaded with a particular task set, and was invited to execute a series of commands in a practice scenario (the Keydoor scenario). The commands specifically shown to the user were `help`, `look`, `pickup` and `use`, as these were meant to represent routine operations. Any other command is not explicitly shown.

5.5.5 Scenarios

The user was informed that they would be timed for the rest of the experiment, and that they could skip past any scenario using the `start` command, but that they are encouraged to attempt all of them. The user was then allowed to type start to begin the scenario. At that point onward, observation data and automatic metrics (as detailed previously) were taken for the participant's activities.

When the user finished the scenarios, they were asked if they have any comments. These were informally recorded, and both the interface and task sets were then switched. The process, including the practice session, was then repeated for the other interface variant and task set.

5.5.6 Post-Experiment

After the participant finished the experiment, the researcher,in particular cases, went through some of the observations made with them to clarify particular aspects. The participant was then asked which interface they preferred using, and why. These comments were recorded informally.

5.6 Methods

Methods used to statistically analyse the results below include the following. For each, an indication is provided as to why they are suitable and thus how they shall be used in the analysis of the data:

- Histogram — suitable for the visual analysis of a frequency distribution of bucketed continuous data.
- Averaging — the method for summarising the data gathered for each metric will involve averaging for each scenario and each interface, and then averaging over the scenario averages for each interface. This is to control weighting for varying sample sizes over interface to gain a comparable average.
- Two-way ANOVA — suitable for determining the effect of two predictor variables on a continuous outcome variable (courtesy of `http://vassarstats.net/anova2u.html`).
- Bar charts with indicated error — suitable for providing an indication of relationships and noisiness of continuous samples between groups.
- Poisson Regression Model with Chi-Square Test — suitable for testing the significance of a factor as a predictor for an effect on frequency data (courtesy of `http://stats.idre.ucla.edu/r/dae/poisson-regression/` — modified R script in appendices).

5.7 Results

A pilot was carried out on one participant, which was successful, and thus the results were kept. Minor changes were made to the formalisms of pre-experiment explanations.

Some data was unfortunately lost, or corrupted, and thus had to be discarded from the results. Analysable data remained for 3 participants in each group, except for group B, for which 4 remained. This provided analysable data from 13 participants, or 169 total scenarios.

The raw results of the statistical analyses can be found in the appendices. The data gathered can be found in the digital appendices.

5.7.1 Task Success Rate

Results show that participants successfully completed 83/85 (a success rate of 98.8%) with CAS+HS+TU, and 83/84 (a success rate of 97.6%) with TU. This does not constitute a significant difference.

5.7.2 Task Success Time

A 2x2 analysis of variance (table B.1) was performed on task time, testing presentation order and interface type. The results show that interface has a statistically significant effect ($p < 0.05$), as well as a statistically significant interaction effect between interface and presentation order.

The results also show that, across scenarios, participants achieved, on average, faster task success with CAS+HS+TU in the majority of cases — overall averages were 34.3s (SD 18.6s) for CAS+HS+TU and 46.0s (SD 24.7s) for TU. The data gathered are however very noisy, with very wide standard deviation.

5.7.3 Keypresses

A linear model (table B.2) of number of keypresses given interface type and scenario was generated, and an analysis of deviance of the full model and the model excluding interface was performed. A two degree-of-freedom chi-square test (table B.3) indicates that interface is a statistically significant ($p < 0.05$) predictor of number of keypresses per-scenario. Averages show that significantly fewer keypresses were performed when using CAS+HS+TU — 59 (SD 45) for CAS+HS+TU and 112 (SD 49) for TU.

5.7.4 Number of Commands

A linear model (table B.4) of number of commands given interface type and scenario was generated, and an analysis of deviance of the full model and the model excluding interface was performed. A two degree-of-freedom chi-square test (table B.5) indicates that interface is a statistically significant ($p < 0.05$) predictor of number of commands per-scenario. Averages show that fewer commands were executed when using CAS+HS+TU — 6 (SD 2) with CAS+HS+TU, and 7 (SD 3) with TU.

5.7.5 Documentation Checks

A linear model (table B.6) of number of documentation checks given interface type and scenario was generated, and an analysis of deviance of the full model and the model excluding interface was performed. A two degree-of-freedom chi-square test (table B.7) indicates that interface is a statistically significant ($p < 0.05$) predictor of number of documentation checks per-scenario. Averages show that fewer documentation checks where performed when using CAS+HS+TU — a rate of 0.25 for CAS+HS+TU, and 0.58 for TU.

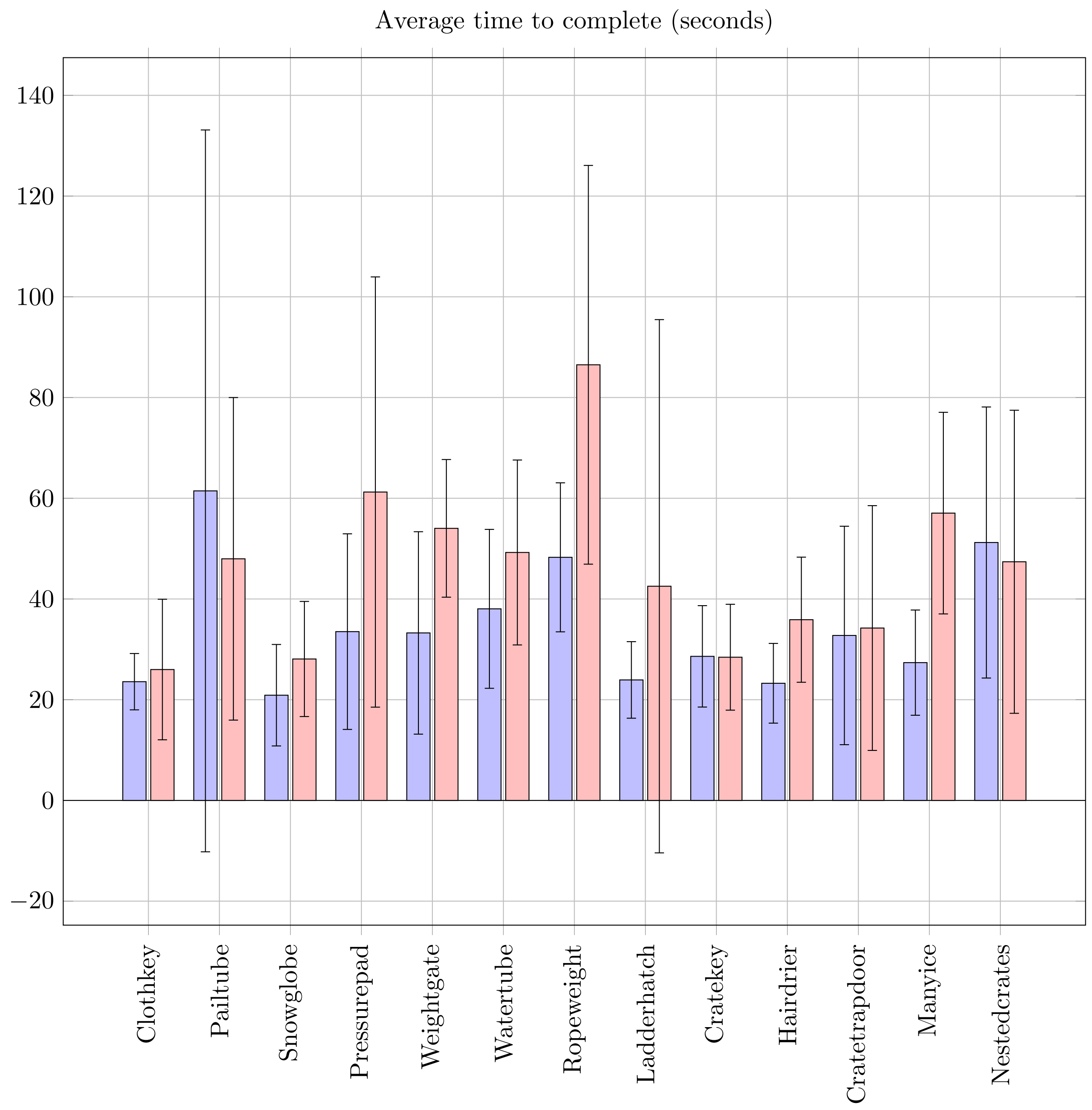


Figure 5.2: Bar graph showing average time to complete for each scenario and each interface (blue shows CAS+HS+TU and red shows TU)

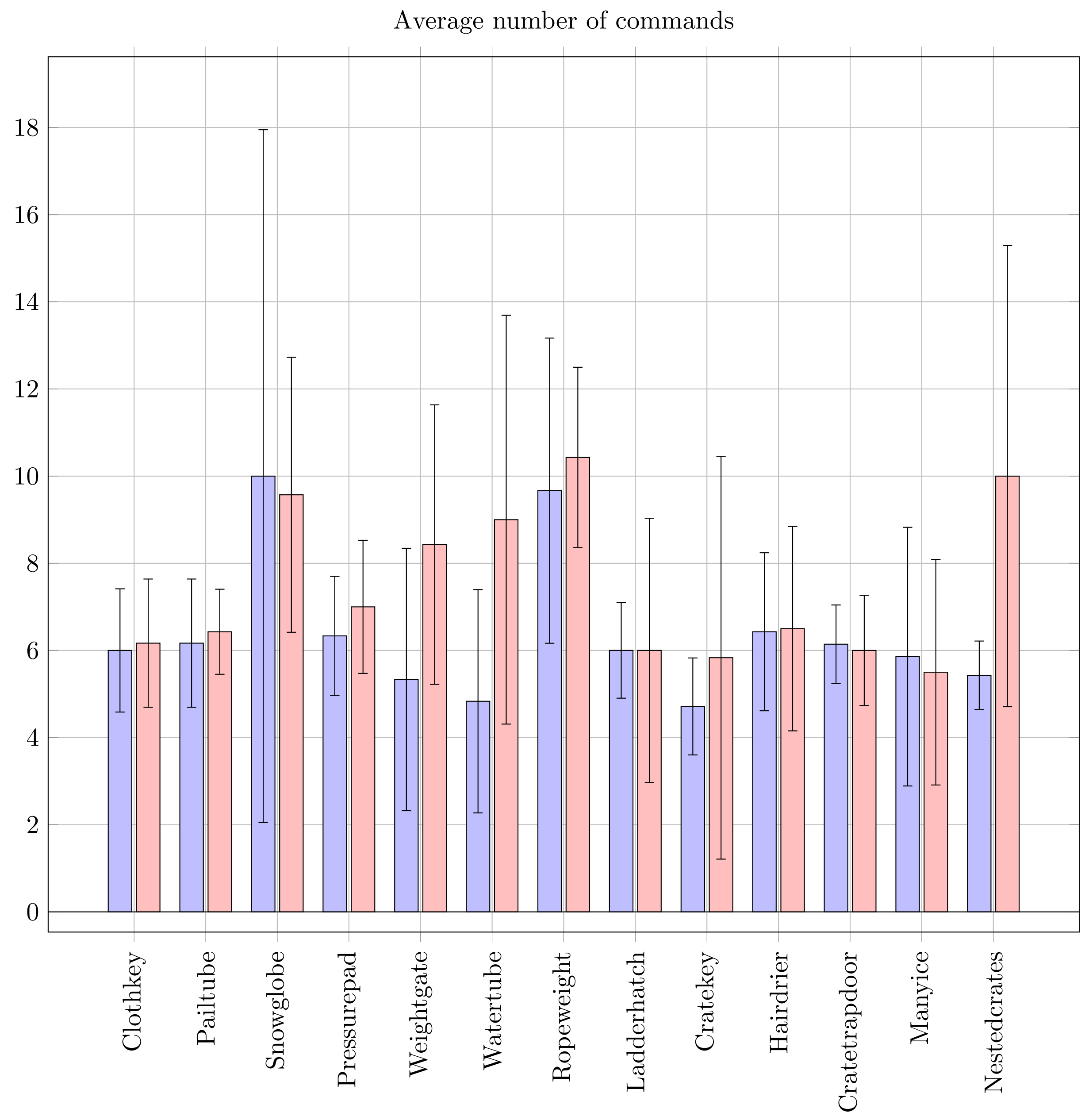


Figure 5.3: Bar graph showing average number of commands executed for each group and each interface (blue shows CAS+HS+TU and red shows TU).

5.7.6 Number of Invalid Commands

A linear model (table B.8) of number of invalid commands executed given interface type and scenario was generated, and an analysis of deviance of the full model and the model excluding interface was performed. A two degree-of-freedom chi-square test (table B.9) indicates that interface is a statistically significant ($p < 0.05$) predictor of number of invalid commands executed per-scenario. Averages show that fewer invalid commands where performed when using CAS+HS+TU — a rate of 0.27 for CAS+HS+TU, and 0.57 for TU.

5.7.7 Time Between Each Command

A 2x2 analysis of variance (table B.10) was performed on all times between commands, testing presentation order and interface type. The results show no statistically significant effects. CAS+HS+TU had an average of 4.8s (SD 3.6s) and TU had an average of 5.9s (SD 5.2s)

5.7.8 Observations

Observations which were consistently exhibited by a significant proportion of participants are described here.

The phrase “many participants” is used here to refer to a proportion of the participants that is greater than half, but not all. The phrase “some participants” is used here to refer to a proportion of the participants that is less than half, but greater than one individual.

Sources of Confusion

Across interfaces, the following sources of confusion were consistently observed:

- Many participants were confused by how the domain worked in certain contexts.
- Many participants were confused by commands which had similar usage or structure, but arbitrary applicability or validity — amongst these were the `use`, `place` and `pickup` commands — this was observed more frequently during TU use than CAS+HS+TU use.
- Many participants were confused when the usage or structure of a certain command did not fall into expectations established by routine tasks — in particular the `pickup` command — this was observed more frequently during CAS+HS+TU use than TU use.

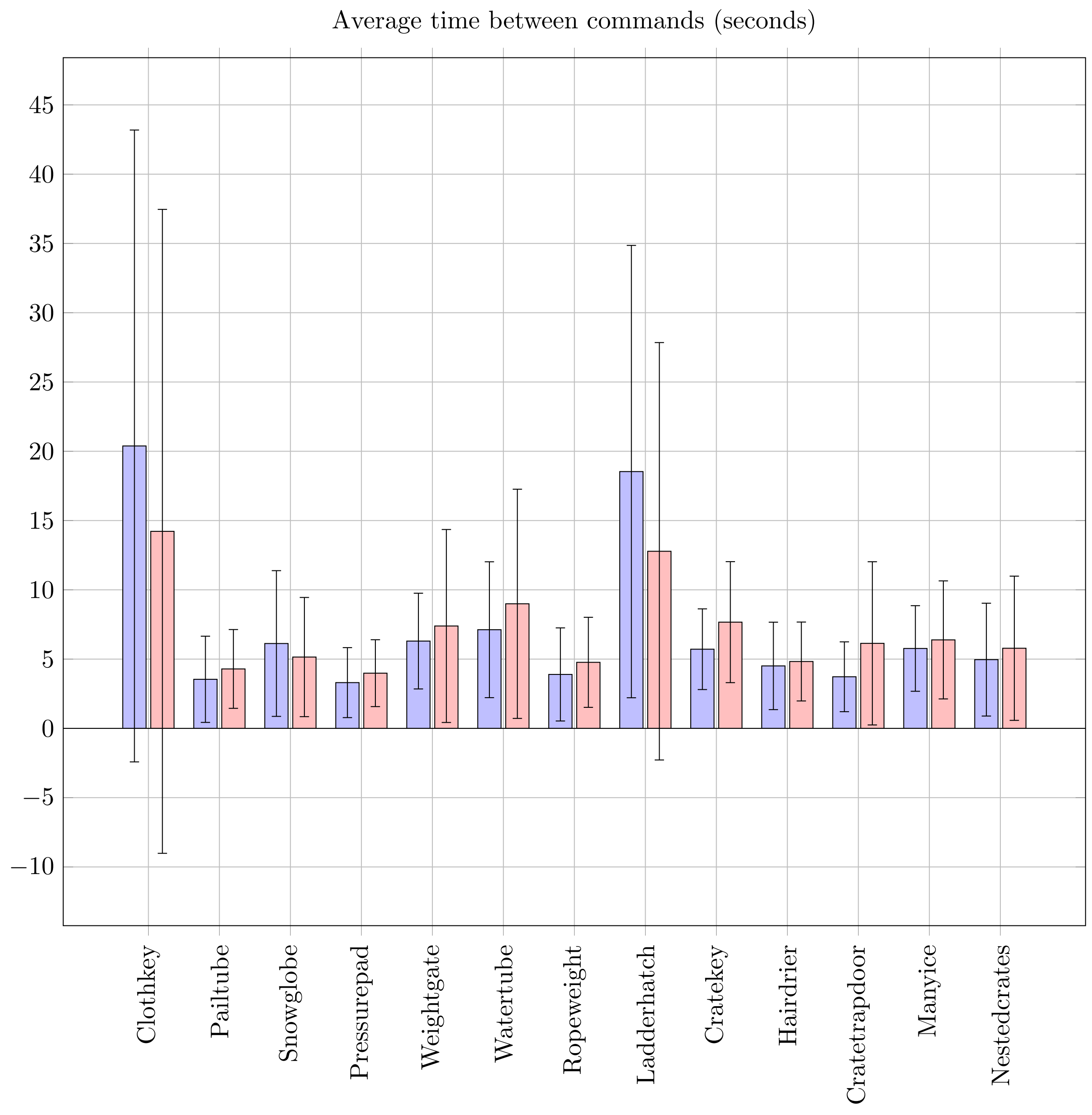


Figure 5.4: Bar graph showing average time between commands for each group and each interface (blue shows CAS+HS+TU and red shows TU)

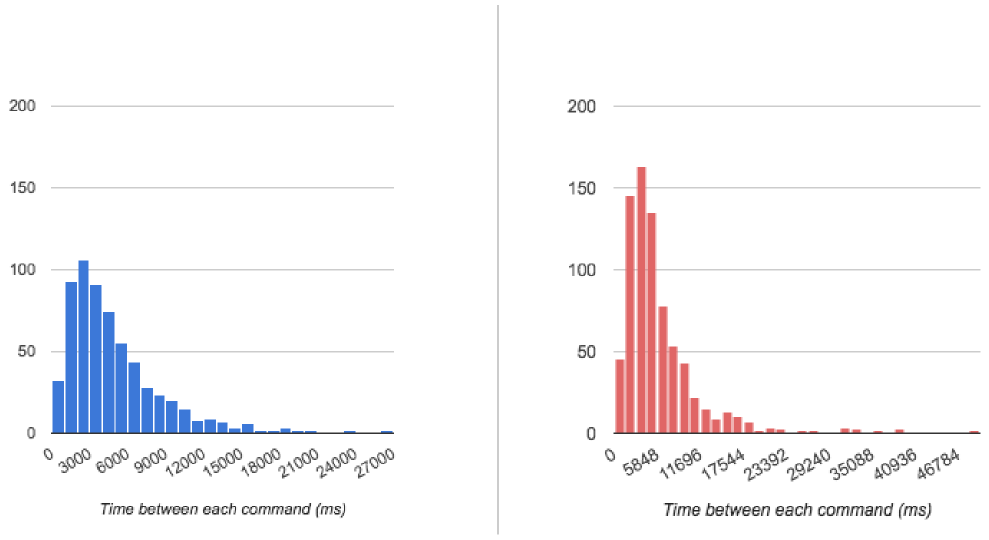


Figure 5.5: Histograms showing the frequency distributions of time (in ms) between each command for CAS+HS+TU (blue) and TU (red), with bucket size 1000

For only CAS+HS+TU, the following sources of confusion were observed:

- Some participants attempted to select Help Snippets as they would Context Aware Suggestions, which confused them.
- Some participants frequently did not examine the output of commands, and thus were not aware of domain changes that occurred, which lead to a state of confusion.
- Some participants, expecting different results, attempted to re-execute commands which were invalid.

Exploratory and Speculative Actions

Across interfaces, the following exploratory and speculative activities were consistently observed:

- Many participants explored possible commands by looking through operational documentation — this was observed more frequently during TU use than CAS+HS+TU use.
- Many participants explored the domain by using `look` on various objects — this was observed more frequently during TU use than CAS+HS+TU use.
- Many participants guessed commands based on their intuitions about the domain.
- Some participants used operational documentation to explore less over time.

For only CAS+HS+TU, the following exploratory and speculative activities were observed:

- Many participants explored what was possible through Context Aware Suggestions.
- Some participants systematically trialled many different commands on one object.

For only TU, it was also observed that some participants performed `undo` on a successful command out of curiosity.

Resolution Events and Discovery Methods

Across interfaces, the following resolution events and methods of discovery were consistently observed:

- Many participants resolved a state of confusion by using `look` on the objects in the room.
- Many participants discovered a required command through the use of operational documentation — commands in particular which were discovered in this way were `place`, `undo`, `open` and `break` — this was observed more frequently during CAS+HS+TU use than TU use.

For only CAS+HS+TU, the following resolution events and methods of discovery were observed:

- Many participants used undo to recover from an error state.
- Many participants discovered a required command through the use of Context Aware Suggestions — commands which were found in this way were `undo`, `break`, `place` and in particular `every` and longer commands incorporating this.

Other Observations and Participant Comments

Across interfaces, many participants executed `help` and `look` at the start of every scenario.

For CAS+HS+TU:

- Many participants used Context Aware Suggestions to, in particular, execute routine activities more rapidly.
- Some participants utilised Context Aware Suggestions to specifically provide rapid action.
- Some participants rarely utilised Context Aware Suggestions.

- Some participants commented that the presence of suggestions made scenarios significantly easier.

For TU, some participants were not able to understand iterative actions (use of `every`) and did not utilise them in any scenario.

5.7.9 Preference

When queried, most participants stated that they preferred CAS+HS+TU. Specifically, 9 participants preferred CAS+HS+TU, 2 preferred TU, and 2 had no strong preference.

CAS+HS+TU

Participants had a number of common overarching reasons why they preferred CAS+HS+TU over TU:

- It provided them with a choice of valid actions, which they stated was easier than deriving their own commands.
- It was easier to discover the solutions to domain problems.
- They did not need to use operational documentation as much.
- It provided them with a good prompt for remembering the syntax, and thus was useful for learning.
- There was no harm in them being there, as it only took up a small region of the screen.

Other reasons that were given in isolation were as follows:

- It made them more willing to try new commands, because they felt safer from error states.
- They had to type less, which was faster for them.
- They had to read and understand less of what was happening.
- It provided a welcome variation on traditional CLI experiences — “Command lines are old and need to be updated”.

TU

Participants had a number of common overarching reasons why they preferred TU over CAS+HS+TU:

- The command suggestions were not always relevant or useful to them, so they did not trust them overall.
- They did not use Context Aware Suggestions.
- They were more accustomed to typing than selecting what they needed.
- They prefer to have more control over what actions they take.
- They felt less engaged with the activity.
- It was uncomfortable given their expertise with other CLIs.
- The extra features were distracting to them.
- Alternating between selecting commands and typing them was cognitively hard for them.

5.8 Discussion

In this section, the hypotheses and study questions will be evaluated with respect to the results obtained.

5.8.1 Evaluation of Hypotheses

H1a. Users achieve a better task success rate with Context Aware Suggestions

Hypothesis 1a. is contradicted by the results, as there was not a significant difference between the average success rates of the two interfaces. Many participants stated that the tasks were very easy, and as such, it is not unreasonable to suggest that the success rate results may have been subject to a ceiling effect.

H1b. Users will complete tasks in a shorter time with Context Aware Suggestions

Hypothesis 1b. is supported by the results, as CAS+HS+TU had a significantly lower average completion time than TU. There was a significant interaction effect between interface and presentation order, and this is shown by the fact that group A in fact had a higher average task completion time overall. This could be due to the particular task set, or a dithering effect caused by the presentation of a novel interface.

The chart (5.2) shown, however, throws uncertainty over claims of statistical significance — these reveal that per-scenario, there is so much noisiness over the samples that their variance consistently overlaps. It is possible that these results were affected in some way by experimental error. However, observations and discussions with participants provide further evidence to support the hypothesis, with many participants stating that they found the use of Context Aware Suggestions a faster alternative to manual typing, and that it was easier to find the solution to a task.

H2a. Users will use fewer keypresses to succeed a task with Context Aware Suggestions

Hypothesis 2a. is supported by the results, as CAS+HS+TU had significantly lower average per-scenario keypresses. Whilst this does seem to indicate that less physical effort is required to solve a problem with Context Aware Suggestions, it is not clear whether overall cognitive effort is reduced.

Many participants seemed divided on this matter — some stated that they preferred typing more, and others stated that they preferred typing less. One participant pointed out that alternating between typing and selecting suggestions was cognitively hard for them, and this alternation was not worth the benefits to them.

H3a. Users will discover commands faster with Context Aware Suggestions

Hypothesis 3a. is supported by the results, as CAS+HS+TU had a significantly faster average success time than TU. This is based on the assumption that faster discovery of a desired command leads to faster task success. However, other factors may have also lead to faster task success times.

Further evidence is given by observations that participants discovered their desired command more often from Context Aware Suggestions than from operational documentation when using CAS+HS+TU. This suggests that Context Aware Suggestions provide a faster discovery method than the operational documentation. This also assumes that participants spend most of their time searching for a solution to the problem.

Novel commands especially were discovered through Context Aware Suggestions, such as commands incorporating `place`, `undo`, `open`, `break` and `every`. Some participants also stated that they found it easier to find the solution to a problem with Context Aware Suggestions, and that they did not need to use the `help` command.

H4a. Users will check documentation less with Context Aware Suggestions and Help Snippets

Hypothesis 4a. is supported by the results, as, on average, `help` was executed significantly less in scenarios completed with CAS+HS+TU than with TU. The observations gathered also broadly support this, as many participants were observed to use Context Aware Suggestions as a command discovery method, which lessened the extent the `help` command needed to be used. Some participants did also state that suggestions removed the need to check documentation as much for them.

H5a. Users will execute fewer invalid commands with Context Aware Suggestions and Help Snippets

Hypothesis 5a. is supported by the results, as fewer invalid commands were executed per-scenario on average with CAS+HS+TU than with TU. It is clear that, presentation of a set of valid commands makes it less likely that participants would speculatively execute an invalid command.

This could either be because they selected one from Context Aware Suggestions, or because they have a better idea of what constitutes a valid command due to exposure of valid commands from Context Aware Suggestions and Help Snippets. One participant stated that "suggestions almost eliminated need for `undo`", because they felt they made fewer mistakes as a result of being suggested valid commands.

H5b. Users will use Tree Undo to recover from error states

Hypothesis 5b. is difficult to support using the results presented. Had metrics been taken on irreversible or domain error states, it might have been possible to correlate this with `undo` use per-scenario. It was observed that `undo` was most often used to undo the Ropegate (table 5.2) irreversible failure state, though not all participants knew to use undo in this scenario, and some simply skipped. In fact, only a slim majority (61.5%) of participants used `undo` at least once. Some participants did comment that they found undo helpful.

H6a. Time between the end of a user's last action and the start of the next action will be reduced with Context Aware Suggestions

Hypothesis 6a. is contradicted by the results, as there is no significant effect of interface on the time between commands. The frequency distributions of time between commands for each interface (figure 5.5), show that shorter times are in fact more frequent for TU than CAS+HS+TU, which had a more spread distribution. It could be suggested that the presentation of Context Aware Suggestions and Help Snippets incited more deliberation time amongst participants.

However, it was observed amongst many participants that routine tasks in particular — such as those constituting the ideal actions in the Keydoor (see table 5.2) scenario — were executed very rapidly with suggestions. The times to read the result of `help` commands are also included in these measures, so it is not clear if it is possible to conclude that suggestions were ineffective at lowering this time. That is, reading operational documentation and reading Context Aware Suggestions may take roughly the same amount of time, but these measures split the time it takes to deliberate the referral of, and the reading of operational documentation.

Given that the `help` command was used much less, and tasks were completed in a shorter amount of time with CAS+HS+TU, one can argue that time of formation of intent to transformative action — that is, not `look` or `help` — may have indeed decreased with suggestions. However, without direct empirical evidence, its difficult to support this claim. There are after all, many facets of a participant's action — formation of intent, specification of intent and execution of intent — that may have been affected here.

5.8.2 Evaluation of Study Questions

SQ1a. Do Help Snippets promote a user's understanding of commands?

There is no quantitative data to support claims that Help Snippets promoted understanding of commands amongst participants. Indeed, it was informally observed that only a small minority of participants found Help Snippets useful. However, some participants did state that Help Snippets were good for understanding command syntax and structure.

SQ2a. Do users explore more with Context Aware Suggestions?

If the assumption is taken that a participant would execute more commands if they explored more, then the results in fact contradict the hypothesis — participants executed significantly fewer commands per-scenario with CAS+HS+TU. It can be argued that the features within CAS+HS+TU werent specifically used to explore a command space, as observations show that the `help` or `look` commands were used just as often as a method for browsing possibilities.

Some participants commented that Context Aware Suggestions made them more willing to try out new operations, but many other participants seemed to utilise Context Aware Suggestions exclusively to provide swift action for routine tasks. One participant even commented that they explored less with suggestions.

SQ2b. Do users explore more with Tree Undo?

There is very little quantitive data to support an answer to this question, particularly in the context of such a small test. In addition, users were not in fact notified that undo existed before the test began. However, some participants did undo successful commands out of curiosity, and one participant did comment that they "wouldn't have speculated so much without undo". One participant touched on the potential issue that most terminal users will expect such an undo functionality not to exist, and that it might be difficult to "untrain" such an expectation.

5.9 Critical Analysis

It is pertinent to provide a critical analysis of the experiment as a whole, in order to assess the applicability of the findings presented.

5.9.1 Applicability of the Designed Domain

It is clear that the designed domain presented differs very greatly from those that are operated on in real-world CLI use, even though an attempt is made to explore the pertinent aspects of such domains.

The Game Factor

Indeed, some participants did question whether a game provided a correct experience to test real-world CLI use. It seems that while a fundamental aim of a work-related activity is effectiveness, the fundamental aim of engagement in games is very distinct. One participant in particular stated that Context Aware Suggestions would be better "in a work context".

Technical Aspects

Not all technical aspects of CLI interaction could be explored in the design of the domain, such as referencing specific objects which were of the same class by a unique name. The interactions between these sorts of aspects may have been important to consider or examine.

Long-term Use

Whilst the experimental tasks carried out only lasted, in total, around 30 minutes for each participant, much real-world CLI use occurs in a work context over much longer periods of time. The experiment fails to explore potential nuances of long-term use, such as learning or customisation, which effect the use of the enhancing features. Indeed, one participant stated that they may have found themselves getting into the habit of using Context Aware Suggestions if they used the tool for a longer period of time.

Command Corpus Size

It is difficult to support claims that the domain presented a vast array of functionality. Indeed, only 13 commands were exposed, when a domain such as the Unix core exposes around 160 different commands [Hamilton, 2014]. This is very likely to have had an effect on the results; indeed more than one participant stated that there were not a lot of commands to remember.

The fact that the entire corpus was extremely easy to access via the `help` command meant that users found the process of discovery often very easy. A better method might have been to provide the corpus through some external documentation method.

Uncharacteristic Guessability

The design process taken to deriving the command corpus was largely domain driven, however many CLI environments, such as Unix, present an generalised, OS-level paradigm in which commands are appropriated to many domains and contexts. Indeed, it would seem pertinent to classify two sorts of command corpora based on their generality with respect to a particular domain. This domain-first style of syntax design certainly had an impact on the guessability of the commands.

This provides some insight as to why many participants found the act of discovery in this domain so easy, when much of real-world CLI discovery is very difficult. Commands were extremely guessable because they had good cues to memory, and were easy to understand from a domain context, due to a natural morphological affordance relationship between domain objects and commands. These factors could have been in some way eliminated to accurately test Context Aware Suggestions as a method for discovery in a less guessable

CLI environment. However, the principles of this project strongly derive from those of HCI, and thus deliberately obscuring mappings between meaning and labels would not make sense. Indeed, it would at best have created a weak "strawman" argument.

Hand-Trained Context Aware Suggestions

It is not clear whether it was reasonable to hand-train the model for Context Aware Suggestions. The commands suggested were sometimes completely nonsensical given the context — for example, `place weight on rope` is suggested when `help` is typed. This is due to the fact that the suggestion engine only analyses the first most recent output. Participants often found this confusing, and it is possible that with more sophisticated machine learning methods, participants would have more faith in Context Aware Suggestions.

5.9.2 Limitations of the Data

The data gathered were largely unsatisfactory for the analytical requirements of the study. Not enough metrics were taken to adequately support the evaluation of the hypothesis.

In addition, samples within groups were very small, leading to extremely noisy data, which makes it very hard to justify claims about underlying phenomena.

Much of the data is also likely to have been affected by experimental error. A big factor in this is the ceiling effect caused by the easiness of most aspects of the scenarios presented. Statistical tests were difficult to perform, as the experiment had a complex design, with different numbers of participants taking different scenario sets with different interfaces — this is due to the fact that there are 4 participants in group B, and 3 in the rest.

It is also difficult to make concrete claims about the performance of different methods of enhancement in isolation based on the data gathered, due to the way they were incorporated in the different interface versions. Had more versions been created and tested which incorporated the enhancements in different ways, it might have been easier to provide a more complete comparative analysis.

5.10 Conclusions

Given the analysis performed, it is possible to draw a number of conclusions which address the broader aims of the study.

5.10.1 Effectiveness of Context Aware Suggestions

It is possible to conclude that Context Aware Suggestions were effective for lessening physical effort of execution amongst participants, and also for rapidly discovering required solutions. However, they were not effective in every case, and indeed not every participant appreciated them. This was often due to the fact that they required more cognitive effort to engage with, and that they had a perceived reductive effect on agency. With more sophisticated machine learning methods and big data, Context Aware Suggestions could become much more effective.

Though they do appear to solve a broad set of CLI problems, there are potentially better solutions to the Exploration Problem. Indeed, suggestions seem to bypass concerns of exploration, as they tend to just provide what is supposedly the “correct” action to take. It is not clear whether this disrupts a user’s intent, but an option which provides more agency or control may be more beneficial.

Another pertinent aspect of the use of Context Aware Suggestions is their impact on engagement that many participants described. Participants described how Context Aware Suggestions detached them from the problem solving process, or that they did not need to know what was going on to complete certain tasks. It is clear that this disengagement can impact the use of Context Aware Suggestions as a tool for learning and understandability.

5.10.2 Effectiveness of Help Snippets

It can be concluded that Help Snippets were not an effective feature in the context of this experiment. Indeed, only a small minority of participants engaged with Help Snippets as a method of documentation checking by typing a few letters from a command. Often, however, they were typically just a source of confusion, as many participants tried to select help items as though they were suggestions.

It seems that Help Snippets were a feature which found an improper place in the interaction flow of the participants; a participant already knew what they wished to do when in the process of executing a command, which is primarily where Help Snippets show. Perhaps if a different paradigm of interaction were encouraged, one which invited users to speculatively search for commands in the prompt, they would be more effective at promoting command understandability.

5.10.3 Effectiveness of Tree Undo

It can be concluded that, while the `undo` command itself was found to be useful by participants in a particular situation, no benefit was seen of the visualised aspect of Tree

Undo. This is likely because this functionality only finds niche usefulness in cases where a participant had to enact certain changes before a sequence of already executed commands — an activity which was not afforded by the scenarios presented.

Chapter 6

Analysis

In this chapter, we shall aim to draw together and summarise the findings from the investigations carried out throughout this dissertation, generate some additional design ideas which address the overarching aims, and provide further analysis with respect to the literature. In doing this, it is possible to propose models which formalise the findings of these works.

We shall first present a formalised view on the Exploration Problem introduced in the Literature Review. From this basis, and on the basis of findings from the Exploratory Study and the Comparative Study, we will present an informed model of action specification for CLI users. Finally, discovery methods and approaches to the Exploration Problem will be presented and evaluated, informed by findings throughout the study.

6.1 Formalising the Exploration Problem

The Exploration Problem is presented here as a formalised extension of the Vocabulary Problem proposed by Furnas et al. [1987], which describes issues of action specification within CLIs.

Given the following conditions:

- An interface whose operation is subject to some structured control language [Norman, 2007], which is in part constituted of a set of labels which correspond to a set of referents.
- An interface which exposes a sufficient number of referents such that it is not trivial to hold comprehensive knowledge over them.
- A user subject to some overarching domain activity which requires them to solve a "novel" problem, which is one which cannot be solved solely on the basis of recognition-primed decisions.
- A domain of activity populated with interactive domain objects.

The Exploration Problem is the problem of, in an efficient manner, searching a combinatorial set of commands validly applied to domain objects, with the intention to discover a set of command strings which can be successfully applied to solve the domain problem. Commands are formally defined here as a set of referent labels, structured in some way by the syntax of the control language, which express a particular intention of a user.

This process of discovery can be defined as having two stages, which could occur in any order, or simultaneously:

- The discovery of the existence of a set of domain referents for solving the problem, and their operational details.

- The discovery of actionable labels and syntactic structure to validly execute the required referents.

This allows the generation of a model of action specification for CLIs (figure 6.1), which characterises the operational states of a user engaging in an Exploration Problem scenario.

6.1.1 Types of Exploratory Activity

The generated model of action specification of CLI users (figure 6.1), delivers a taxonomy of distinct exploratory activities for progressing between states of knowledge. These types of activity are presented in a manner specifically divorced from methods by which a user might discover information — indeed, these can emerge from the utilisation of any sort of artefact — but some exemplar methods of discovery are presented alongside each type.

These activities can be, more or less, accurately described in the context of the findings of this dissertation. To illustrate each activity, a hypothetical example scenario will be described, which features a persona, Susan, whose intention is to move a particular set of files using a Unix Shell. Each of these incorporates key CLI issues identified in the Literature Review, user experiences described in the Exploratory Study, and empirical observations from the Comparative Study.

Type A: Label Speculation — from Unknown to Known Commands

Type A exploration is concerned with the discovery of syntactic structure and labels, but where referents and their technical detail are partially or wholly unknown or not fully understood. This primarily occurs through guessing or speculating about potential labels which make sense from the perspective of the domain of activity, or by browsing documentation which details high-level syntactic structure.

> Susan examines some documentation on wildcards, and determines that she can use these to apply some function to each file she wants to move. She guesses that the function she might want might have the label `mv`, but she is not entirely sure if it exists in this form. She also is not sure how she might specify the path to move the files to, but given other commands she has used, this could be done via a flag or simply by providing another parameter.

This categorisation is supported by observations made on the way participants of the Comparative Study speculated about the existence of a command — some participants systematically trialled command labels on one object (see section 5.7.8).

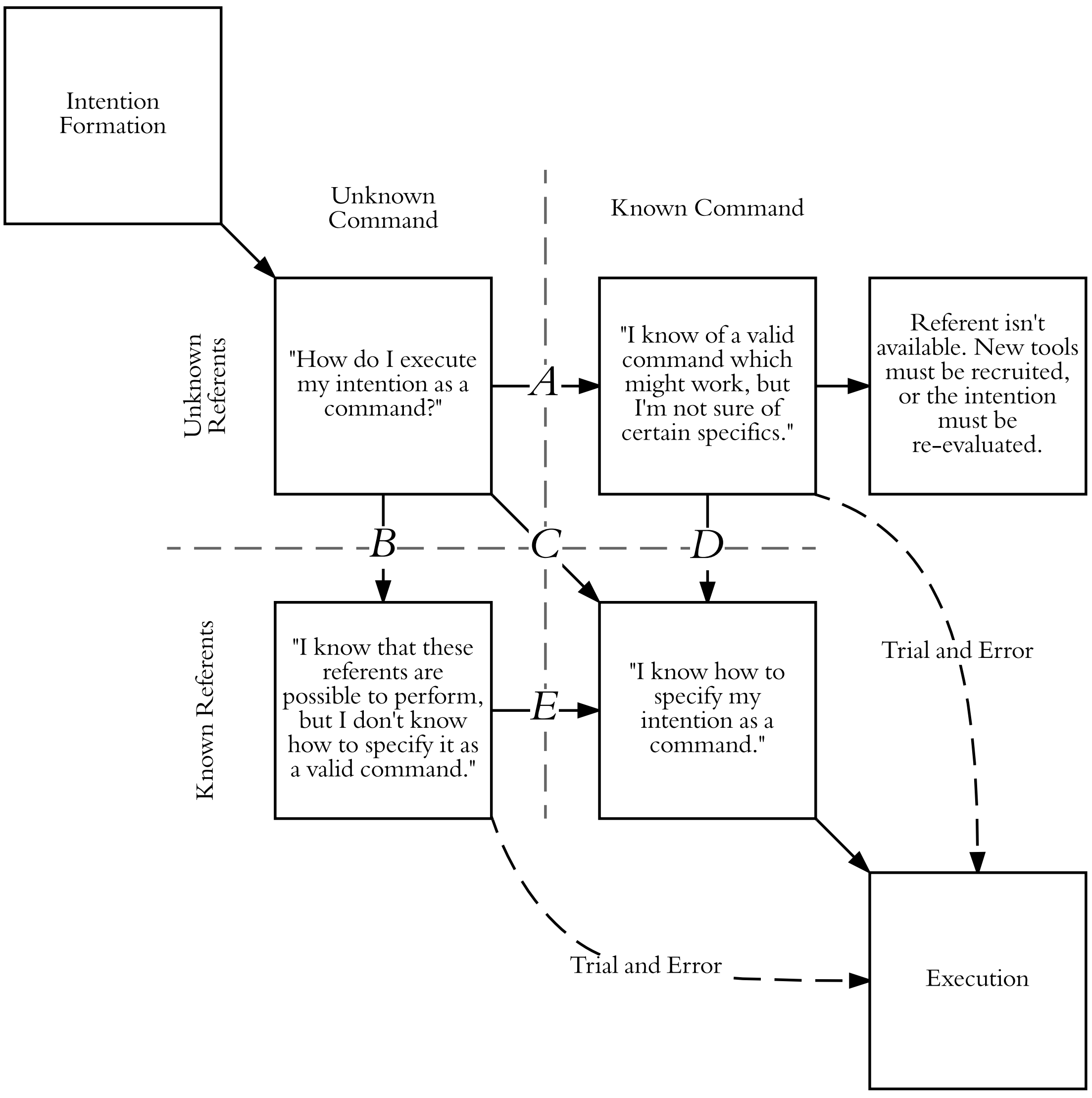

Intention Formation
Unknown Command
Known Command
Unknown Referents
Known Referents
"How do I execute my intention as a command?"
"I know of a valid command which might work, but I'm not sure of certain specifics."
Referent isn't available. New tools must be recruited, or the intention must be re-evaluated.
"I know that these referents are possible to perform, but I don't know how to specify it as a valid command."
"I know how to specify my intention as a command."
A
B
C
D
E
Trial and Error
Trial and Error
Execution

Figure 6.1: A model of action specification for CLI users.

Type B: Referent Browsing — from Unknown to Known Referents

Type B exploration is concerned with the discovery of the required referents, but where the required structure or labels are not immediately clear. This could occur through browsing a list of manual entries, which provide comprehensive lists of referents, but no information on parameter formats or other syntactic structure. This can also occur by understanding the capabilities of the tools available, but not remembering particular labels, parameter formats or connective syntax.

> Susan types `man` and begins to search for a command which will help move her files. She finds one, but it is not clear how this could be applied to a number of files.

This categorisation is supported by observations made on the way participants of the Comparative Study used operational documentation — many participants guessed commands based on their intuitions about the domain (see section 5.7.8).

Type C: Direct Solution Discovery — from Unknown to Known Commands and Referents

Type C exploration allows a user to directly discover exactly how to specify what they want to do as a valid command. This process can occur if documentation is found which wholly describes the required method, or described by a peer interpersonally or on a forum.

> Susan Googles the following query: "move multiple files in Unix". The first result is a StackOverflow (`https://stackoverflow.com/`) page which explains the exact command she requires, and how to apply it to her situation.

This categorisation is supported by discussions had with participants of the Exploratory Study on the use of Google as a method of discovery — Googling for direct answers was found to be a particularly common thread (see section 3.5.2).

Type D: Referent Detail Checking — from Unknown to Known Referents

Type D exploration is the process of checking the details of the referent space — does the referent exist for the speculative method, and if it does, what are the operational details which are required for their execution? This process can occur through referent documentation checking by querying potential labels, or by peer guidance.

> Susan types `man mv` into her terminal. This gives her comprehensive operational detail on how the `mv` function should be used, allowing her to construct a valid command.

It is also possible that this avenue of discovery results in failure because one or more of the referents that the user is searching for are not available, or are not correct for the required application. In this case, different tools may need to be used, the referents possibly imported from a package management system, or the user may re-evaluate their intentions.

This categorisation is supported by observations made on the way participants of the Comparative Study used operational documentation — many participants discovered a required referent through the use of operational documentation (see section 5.7.8). In addition, participants of the Exploratory Study cite the usefulness of help flags as methods for gaining more detail on a candidate referent (see section 3.5.2).

Type E: Syntax Checking — from Unknown to Known Commands

Type E exploration is the process of checking correct labels and connective syntax given the required referents. This might be done by examining required operational documentation, or through peer guidance.

> Susan asks her colleague, Samantha, if the command `move /file*.txt to ../folder` is correct. Samantha sees what Susan is trying to do, and tells her the command she wants is `mv /file*.txt ../folder`.

This categorisation is supported by discussions had with participants of the Comparative Study regarding Help Snippets — some participants stated that short excerpts of documentation were good for understanding command syntax and structure (see section 5.8.2).

Trial Actions

An additional sort of exploratory activity occurs in the process of trial and error, whereby a trial action is taken on the basis of incomplete knowledge. This is based on a hypothesis that some knowledge must be known about the available referents, the required elements of the control language, or both, in order to make a valid guess. These trial actions may not result in success, but may potentially help a user move towards a correct solution.

A trial action can result from a guessed, but valid command in the absence of knowledge about it's referents, but also from a guess about the exact phrasing of a command, given a set of known referents.

This categorisation is supported by observations made on the way participants of the Comparative Study participated in trial and error, and also how participants of the Exploratory Study described their approaches to this process — trial and error was a method of a discovery, but also, in part, a learning tool for them (see sections 3.5.2 and 3.5.2).

6.1.2 The Learning Factor

The extent to which each type of activity described is an exploratory or browsing process depends upon the expertise of the user. That is, a user may not have to search if they already know the answer, and thus they will move more swiftly through these stages of specification the more their activities are knowledge-primed. When a user is able to maintain complete knowledge over the required set of referents, their technicalities and labels, and any other connective syntax, the activity becomes a routine task.

It is the aim of learning in this context to provision a process whereby novel problems within CLIs are transformed into routine tasks. This constitutes a movement from analytical decision processes to recognition-primed decision processes. Arguably, this factor explains the continuum of "innateness" or automaticity of action described by participants of the Exploratory Study, which characterises the variety of actions taken by CLI users. Indeed, the process of learning allows users to progress to situations where they are able to achieve states of flow — fluent translation of intent into commands, no conscious process of planning or browsing. This state of ultimate efficiency is the goal of many CLI users.

It would also seem that many processes by which CLI users perform exploratory activities also directly contribute to learning. As a consequence of this, it may be suggested that certain exploratory activities are actually highly desirable from a learning standpoint. These claims are supported by works such as Bjork and Kroll [2015] on "Desirable Difficulties" and Howes [1994] on exploration as a learning exercise. A question remains as to what kinds of exploratory activity are most conducive to learning efforts, and indeed whether it is desirable to eliminate the need for exploration altogether in the context of these realisations. Indeed, works within NDM (as discussed in section 2.3.2) suggest that exploration is simply a natural part of human problem solving.

6.1.3 Limitations

The model presented is a highly idealised view on the action specification of CLI users and is also limited to application under only the specific assumptions made clear. Particular features which are explicitly not included within this model are any evaluative steps, or backtracking to previous states. Indeed, there may be scenarios which are not encapsulated within this model at all.

In addition, it may not be clear to what extent users undergo planning processes within the activities described by the model. These may indeed occur before, or alongside exploratory activities.

6.2 Approaches to the Exploration Problem

The principal aim in developing solutions to address the Exploration Problem of CLIs is to support efficient methods of discovery which address all types of exploratory action as detailed in the model above. High-level methods and their actualisations that address this, which were explored in this dissertation, are summarised here.

6.2.1 Peer Support

A versatile source of learning and discovery for CLI users is the provision of support from peers. Peer support is particularly effective because observing the approaches of others can frequently help users discover new ways to approach novel problems. Advice from a peer can serve any of the exploratory activities detailed, but only if the user resides in an environment where advice is rapidly accessible. Therefore, useful extensions to this method of discovery are those that increase the availability of peer advice, such as groupware products and public forums.

Artificial intelligence can also play a role in the delivery of peer support, as suggested by works from Klein et al. [2004] on collaborative models of problem solving.

6.2.2 Documentation Checking

Operational documentation checking can provide an accurate and particularly educational method of discovery, but it's effectiveness hinges on several factors. The ability to search documentation is a primary factor — many participants of the Exploratory Study stated that they found manual pages difficult to search, and found them less effective as a result. Format and layout of the represented documentation is another factor which plays an important role, as this affects the extent to which it can be parsed and understood. The speed with which documentation can be accessed is also affected by whether it must be accessed internally, as with manual pages, or externally, as with web-hosted content or guides.

There are a breadth of documentation methods, including manual pages, help flags or online guides, but each of these support a very narrow range of exploratory goals.

6.2.3 Supporting Trial and Error

Trial and error is a particularly natural method of discovery, but its effectiveness is subject to how well it is supported by the control language and the knowledge that the user currently holds about the domain. Artefacts which can improve the effectiveness of trial and error typically improve the guessability of command labels for easier access to referents, such as Wobbrock et al. [2005]'s Unlimited Aliasing method. Correction mechanisms are another way to improve the speed with which CLI users converge on the correct referent label — these include features like "did you mean" suggestions appended to invalid syntax error messages.

In addition, trial and error processes can be supported by providing increased security from error states. This allows a user to experiment with the domain and the control language in an unimpeded manner, and encourage speculative action in critical scenarios. One way this can be achieved is to provision methods of recovery, such as the `oops` command found within AutoDesk [2015].

6.2.4 Supporting Learning

The aim of learning in CLIs, as previously discussed, is to allow a user to knowledge-prime processes of action specification, such that very little or no exploratory activities need be engaged with. Supporting the process of learning is thus a clear way to address the Exploration Problem.

Participants of the Exploratory Study stated that they often learnt through peer support or checking documentation. It was also noted that learning typically occurred on a needs-driven basis — only very occasionally is time set aside for the purpose of learning. Therefore, CLI learning processes are effective if they can happen rapidly and within a practical context.

There is also evidence from the studies carried out that the ability to properly conceptualise about areas of the domain of activity is key to learning processes. It is clear that the extent to which a user engages with a problem effects how well they are able to formulate a mental model. In the context of this, it is possible to propose that certain methods of discovery, particularly those which are of type C, may have a negative impact on learning, because, as methods of discovery, they are not engaging. This hypothesis is supported by work done by Bjork and Kroll [2015] on what they call "desirable difficulties".

The extent to which learning is effective as a process for knowledge-priming exploratory activities is also subject to the memorability of the control language and the referents. That is, a user may have once discovered a method which may prove useful in a current scenario, but has forgotten what label to refer to it by — the exploratory process must be undertaken again to remind them. Participants of the Exploratory Study state that memorability of a control language is affected by a number of factors, including meaningfulness and consistency of labels, acronyms, parameter orders and connective syntax. It is clear that command labels, when known, should be effective cues to memory for a referent, but also that, given a known referent, a user should be able to logically derive or remember the required label.

6.2.5 Conceptualisation Mechanisms

In the analysis of the shortcomings of the Comparative Study carried out, we discover some pertinent sentiments regarding conceptualisation. Participants of the Comparative Study found the tasks easy because there was an emergent quality of the control language presented. The highly domain-oriented labels and essentially plain-english syntactic structure of commands allowed for uncharacteristically effective action specification. Domain objects were characterised by a naturally present mental model of affordance, due to the physical, rather than abstract, nature of the domain.

Participants had a natural tendency to express commands on the basis of domain sense, even when commands were not syntactically correct. It seems that representing an underlying model in a clear or flexible way to a CLI user may allow effective action specification irrespective of their level of ability.

This idea of providing a highly understandable interface onto a domain may be harnessed by helping a user understand commands from the context of the objects which they can be used upon. That is, CLI domain objects could be enriched with representations of signifiers that indicate afforded commands. These afforded commands, of course, change depending upon the context of use.

Mechanisms for enhancing conceptualisation include visual representation and the use of natural language within the design of a control language.

However, these mechanisms often cause increased "visual clutter", and decrease overall efficiency of parsing and expression, due to expanded or redundant command phrases.

6.2.6 Suggestion Mechanisms

Suggestion mechanisms have been thoroughly explored as an approach to address facets of the Exploration Problem. They are mechanisms which provision direct representation of possible commands, and aim to provide rapid support in exploratory activities.

There are many types of suggestion mechanism, including "did you mean?" suggestions within syntax error messages, history-based mechanisms, autocompletion methods such as those within Fish [Doras] and zsh-autosuggest [de Arruda, 2013], and sophisticated realtime suggestions which analyse the context of execution presented in this dissertation.

These mechanisms, especially those that provide suggestions learnt from other users, are very effective at addressing the Exploration Problem, because they directly demonstrate what is possible to a CLI user. However, it has been found that advanced suggestion mechanisms do have several downsides. One such downside is the fact that suggestions often decrease the user's engagement in a problem-solving task, which has a negative effect on learning processes. Another downside is that they remove perceived agency and control from a CLI user, which was described as a fundamental desire by many participants within the Exploratory Study. It is clear that, for suggestion mechanisms to be effective, these deficits and their effects must be somehow be addressed or limited.

6.3 Summary

In this section, the key findings of this dissertation have been formalised and summarised. A formal model of CLI action specification and the Exploration Problem have been proposed, as well as potential ways in which the Exploration Problem may be addressed.

Chapter 7

Conclusions and Future Work

In the context of the work carried out during this dissertation, it is possible to draw a number of conclusions, present a critical analysis of the findings, and propose future works for further expansion of the area explored.

7.1 Conclusions

In conclusion, it is clear that many — but not all — CLI usability issues are caused by the Exploration Problem of CLI action specification. A critical aim in the design of CLIs and their control languages is to address this problem. A model of action specification (figure 6.1) has been presented which classifies different sorts of exploratory activity. These include:

- **Label Speculation** — speculating about known labels and syntax without precise knowledge of referent detail.
- **Referent Browsing** — exploring what is possible to do within a particular domain without precise knowledge of control language details.
- **Direct Solution Discovery** — exploring with the desire to discover an exact command that can express the given intention.
- **Referent Detail Checking** — converging upon details of required referents.
- **Syntax Checking** — converging upon required details of the control language.
- **Trial Actions** — speculatively executing a command on the basis of incomplete knowledge about required referents or the control language.

There are a number of principle design methods for addressing the Exploration Problem:

- **Delivery of peer support** — artefacts which increase the availability or speed to access advice from other expert users.
- **Documentation** — effective representation of operational details of the control language and the domain.
- **Supporting trial and error** — allowing a user to safely and effectively make judicious guesses to discover required details about a command space.
- **Supporting learning** — increasing the extent to which exploratory activities can become knowledge-primed.
- **Conceptualisation mechanisms** — artefacts which increase user awareness of an underlying domain model in a way which compliments the technical aspects of a control language.

- **Suggestion mechanisms** — a vehicle for delivery of fully-fledged commands which are suitable for a given context.

However, each approach presents certain facets and downsides, which must be addressed in their integration.

7.2 Critical Analysis

In many ways, this dissertation has provided a cursory examination on a diverse set of pertinent modern CLI issues, but has often lacked specific focus on the Exploration Problem. Indeed, though these diverse aspects were explored as facets of the Exploration Problem, many features explored are presented in a disambiguated way, and may, in places, lack the necessary literature support. However, this is mostly as a consequence of the fact that CLI usability receives very little academic attention [Heron, 2015].

In addition, the results and findings of the investigations carried out are largely carried by observations of and discussions with a slim subset of CLI users, who are all students of technical courses at the University of Bath. As such, these conclusions may only apply to specific sorts of CLI user experiences. Indeed, much of the works presented are done so under the pretences of traditional CLIs, and in particular, terminal emulators and shells.

7.3 Future Works

There are a number of future works which can be proposed as a result of the investigations taken and the findings of this dissertation.

7.3.1 Technology Review for CLI Integration

A proper technology review of the tools for CLI integration may also be conducted as a factor for encouraging more development of modern CLI artefacts. A variety of methods are available for review, including those for of integration into CLIs for operating systems, and frameworks and extensions such as Hyper [Rauch], Zsh [Falstad], and Fish [Doras].

7.3.2 Real-World Integration of Context Aware Suggestions

A method for the integration of Context Aware Suggestions has been presented on the basis of solving CLI usability problems, but very little analysis has been performed on the feasibility of its integration into real-world CLI domains. Presenting a method for machine learning the context of use of a real-world CLI is a difficult technology problem, but it also introduces serious issues of security and privacy.

An extended investigation into this may provide an examination of effective statistical methods for integration of domain context pattern recognition in the context of a real-world CLI domain such as Unix, and explore the necessary precautions that must be taken from the context of security and privacy. These works will likely expand upon works done within the area of planning theory, such as Doane and Sohn [2000], Allen [1979] and Allen and Perrault [1980].

7.3.3 Expanded Study of Suggestion Mechanisms within CLIs

The comparative study presented here was a largely speculative investigation which lacks a specific focus on a particular approach to the Exploration Problem. An expanded comparative study should place significant focus on the types of suggestion mechanism which can be integrated in a CLI domain. This domain should have characterising aspects of real-world CLI domains, including large syntax corpuses and highly unguessable command spaces. Different methods of delivering suggestions could include:

- “Did you mean” suggestions within error messages.
- Real-time, continuously represented suggestions, as with the Context Aware Suggestions design previously developed.
- Suggestions upon request, via some command, for a particular context.
- Querying domain objects, via a command, to present a list of commands that the object affords.

This study should take an expanded array of metrics, including the recording of key-by-key input timelines, some method of measuring user understanding, and proper quantification of exploratory behaviours. A large sample of users must be studied, potentially under the context of everyday use, to effectively analyse underlying phenomena.

7.3.4 Information Visualisation of Command Spaces

It is clear that there is an information visualisation problem for command spaces which reaches past the scope of this dissertation. Understanding and developing effective methods for visualising a command space may provide better approaches to the Exploration Problem, by enhancing methods such as suggestion mechanisms, conceptualisation mechanisms and documentation.

7.3.5 Harnessing Benefits of CLIs in Mainstream Software Products

This dissertation has mostly presented findings which benefit a minority of technical and expert users, but there is no reason that these cannot be extended into the realm of mainstream products. It is clear that it may be possible to push the boundaries of definitional CLIs to create hybrid interfaces, incorporating visual, spacial or highly interactive paradigms. These should move towards an aim of integrating the benefits of CLIs into modern software systems for the benefit of users of all expertise. Principally, such a study should be geared towards the accessibility of CLI benefits such as efficiency, and rapid access to large sets of functionality.

7.3.6 Relationship of CLI Efficiency and Understandability

At certain junctures of the investigations carried out, certain evidence can be observed that methods for increasing efficiency and power of action within CLIs are at odds with methods for increasing understandability and conceptualisation of a domain. It may be pertinent to explore whether there is indeed a direct relationship between these factors, and that whether supporting one consistently detriments the other.

7.3.7 User Customisation in CLIs

The role of customisation in CLI users' experiences was briefly explored in the Exploratory Study, but it is clear that customisation is a greatly important facet of how a user appropriates their tools for a particular context of work over time. An investigation into such an aspect should aim to understand why CLI users customise their environments, and whether it is truly the result of a fundamentally dissatisfying "base" or "vanilla" experience.

7.3.8 Peer Support through CLI Groupware

Peer support was identified as a principle approach to the Exploration Problem, but very little work has been done on how guidance or advice from experts can be delivered swiftly to CLI users. An investigation which addresses groupware artefacts for CLIs should evaluate direct integration of a CLI environment with public forums such as StackOverflow (`https://stackoverflow.com/`).

7.3.9 CLI Learning Processes

Learning within CLIs is asserted as the process by which exploratory processes of action selection become more knowledge-primed, but it is clear that it plays an expanded role outside of this purpose. Further investigations into learning processes of CLI users should aim to understand methods by which effective learning is achieved — it is clear that in order to achieve this, a long-term empirical study into real-world CLI use must be performed.

7.3.10 CLI Problem Solving Agents

At many conjunctures of this dissertation, artificial intelligent problem solving agents were identified as being a possible solution to the Exploration Problem. These can simulate the support of a peer and provide suggestions judiciously, based on an internal model of a user's intentions. An investigation should be conducted into how to effectively engage users with sophisticated artificial intelligences for assisting problem solving within CLIs, in the context of existing personal assistants such as Apple's Siri.

Agency, Control and Responsibility within CLIs

Many participants touched upon key notions of agency, control, and responsibility within CLIs — that is, the extent to which a user feels that they are able to affect meaningful action within a CLI is very important. An investigation which, in particular, examines CLI problem solving agents in any capacity, should question how the integration of artificial intelligence and machine learning models threatens a user's agency and control, as well as responsibility in the context of institutional and critical action.

7.3.11 Emotional and Aesthetic Design of CLIs

Works such as [Norman, 2004] prescribe a view upon interface design that importance should be placed on emotional factors, and the way look and feel has an effect on this. Indeed, these works demonstrate that emotional states can alter a user's approach to a problem solving exercise, their mindsets broadened if they are relaxed, or focused if they are stressed. An investigation into emotional and aesthetic design of CLIs should evaluate features such as colour, form, animation and typeface, and their emotional effect on users. Perhaps findings can also be made about fundamental emotional facets of CLI interaction.

7.3.12 Undo within CLIs

It is clear from the literature and findings from the studies carried out that there is a lack of, but strong desire for, useful methods of recovering from error states or reversing catastrophically erroneous commands. An investigation should be performed into an expert CLI user's conception of undo with a CLI, as well as other methods of recovery, in the context of works such as [Cass et al., 2006].

7.3.13 Designing Effective Control Languages

The focus of this dissertation was largely upon the form of a CLI interface, rather than the control language which it exposes. This is because the design of a control language presents an incredibly broad and in-depth challenge. An investigation into the design of effective control languages should examine the extent to which it is important observe principles such as the Unix Philosophy, or those derived from an understanding of purely functional programming languages.

7.4 Summary

This dissertation has provided a number of insights into the patterns of use of expert CLI users. In particular, we have defined and examined the Exploration Problem of CLI action specification. Several methods for improving the experience of expert users of CLIs for novel problem solving have been identified and explored, particularly for addressing the Exploration Problem. Particular examination has been exercised on intelligent suggestion mechanisms for commands, and the relative benefits and detriments they bring.

These works have provided a starting point for many potential extended works, and may provide impetus for contemporary use of advanced CLI features within mainstream software systems. Perhaps the accessibility of powerful CLI features can be improved, such that they no longer exclusively exist within the realm of expert use.

Bibliography

Albany University. Multi-attribute Utility (MAU) Models SWOT Analysis (Strengths , Weaknesses , Opportunities , Threats). *Technology*, (1982):1–6, 2003.

J. F. Allen. A plan-based approach to speech act recognition. *Portal.Acm.Org*, 1979. URL `http://portal.acm.org/citation.cfm?id=909217{%}5Cnpapers://28f4590b-a64b-4c0a-9ebc-51d6773cdd9a/Paper/p505`.

J. F. Allen and C. R. Perrault. Analyzing intention in utterances. *Artificial Intelligence*, 15(3):143–178, dec 1980. ISSN 00043702. doi: 10.1016/0004-3702(80)90042-9. URL `http://linkinghub.elsevier.com/retrieve/pii/0004370280900429`.

Apple. Mac Basics: Spotlight helps you find what you're looking for - Apple Support, a. URL `https://support.apple.com/en-gb/HT204014`.

Apple. Command Line Primer, b. URL `https://developer.apple.com/library/content/documentation/OpenSource/Conceptual/ShellScripting/CommandLInePrimer/CommandLine.html`.

AutoDesk. AutoCAD For Mac & Windows — CAD Software — Autodesk. URL `http://www.autodesk.co.uk/products/autocad/overview`.

AutoDesk. OOPS (Command), 2015. URL `https://knowledge.autodesk.com/support/autocad/learn-explore/caas/CloudHelp/cloudhelp/2016/ENU/AutoCAD-Core/files/GUID-0E72CDCD-9ECA-453A-8A04-CA2921740270-htm.html`.

G. Bedford. jspm.io - Frictionless Browser Package Management. URL `http://jspm.io/`.

R. J. Bergeron, G. Huang, R. E. Smith, N. Bharti, J. S. McManis, and A. Butler. Total synthesis and structure revision of petrobactin. *Tetrahedron*, 59(11):2007–2014, 2003. ISSN 00404020. doi: 10.1016/S0040-4020(03)00103-0. URL `http://portal.acm.org/citation.cfm?id=829549`.

R. A. Bjork and J. F. Kroll. Desirable difficulties in vocabulary learning. *American Journal of Psychology*, 128(2):241–252, 2015. ISSN 00029556. doi: 10.5406/amerjpsyc.128.2.0241.

S. Bradley. 3 Design Layouts: Gutenberg Diagram, Z-Pattern, And F-Pattern - Vanseo Design, 2011. URL `http://vanseodesign.com/web-design/3-design-layouts/`.

V. Braun and V. Clarke. Using thematic analysis in psychology. *Qualitative Research in Psychology*, 3(May 2015):77–101, 2006. ISSN 1478-0887. doi: 10.1191/1478088706qp063oa.

B. Bublitz. gulp.js. URL `http://gulpjs.com/`.

J. Cao, Y. Riche, S. Wiedenbeck, M. Burnett, and V. Grigoreanu. End-user mashup programming: through the design lens. *Proceedings of the 28th international conference on Human factors in computing systems - CHI '10*, pages 1009–1018, 2010. ISSN 1605589292. doi: 10.1145/1753326.1753477. URL `http://portal.acm.org/citation.cfm?doid=1753326.1753477`.

S. Carberry. Techniques for plan recognition. *User Modeling and User-Adapted Interaction*, 11(1-2):31–48, 2001. ISSN 09241868. doi: 10.1023/A:1011118925938.

A. G. Cass, C. S. T. Fernandes, and A. Polidore. An empirical evaluation of undo mechanisms. In *Proceedings of the 4th Nordic conference on Human-computer interaction changing roles - NordiCHI '06*, pages 19–27, 2006. ISBN 1595933255. doi: 10.1145/1182475.1182478. URL `http://dl.acm.org/citation.cfm?id=1182475.1182478`.

E. Charniak and R. P. Goldman. A Bayesian model of plan recognition. *Artificial Intelligence*, 64(1):53–79, 1993. ISSN 00043702. doi: 10.1016/0004-3702(93)90060-O.

T. de Arruda. Zsh Autosuggestions, 2013. URL `https://github.com/zsh-users/zsh-autosuggestions`.

S. M. Doane and Y. W. Sohn. ADAPT: a predictive cognitive model of user visual attention and action planning. *User modeling and user-adapted interaction*, 10(1):1–45, 2000. ISSN 0924-1868. URL `http://www.ncbi.nlm.nih.gov/pubmed/12192684`.

L. Doelle. Konsole Terminal - Konsole - Terminal Emulator. URL `https://konsole.kde.org/`.

C. Doras. Fish Shell. URL `https://fishshell.com/`.

I. Durham, D. A. Lamb, and J. B. Saxe. Spelling Correction in User Interfaces. *Communications of the Association for Computing Machinery*, 26(10):764–773, 1983. ISSN 0001-0782. doi: 10.1145/358413.358426.

M. Dyson and M. Haselgrove. The influence of reading speed and line length on the effectiveness of reading from screen. *International Journal of Human-Computer Studies*, 54(4):585–612, 2001. ISSN 10715819. doi: 10.1006/ijhc.2001.0458. URL `http://www.sciencedirect.com/science/article/pii/S1071581901904586`.

ECMA. ECMAScript 6: New Features: Overview and Comparison. URL `http://es6-features.org/`.

J. S. B. T. Evans. Dual-processing accounts of reasoning, judgment, and social cognition. *Annual review of psychology*, 59:255–278, 2008. ISSN 0066-4308. doi: 10.1146/annurev.psych.59.103006.093629.

P. Falstad. Zsh. URL http://www.zsh.org/.

M. Freudenthal. Using DSLs for developing enterprise systems. *Proceedings of the Tenth Workshop on Language Descriptions, Tools and Applications - LDTA '10*, pages 1–7, 2010. doi: 10.1145/1868281.1868292.

G. W. Furnas, T. K. Landauer, L. M. Gomez, and S. T. Dumais. The vocabulary problem in human-system communication. *Communications of the ACM*, 30(11):964–971, 1987. ISSN 00010782. doi: 10.1145/32206.32212.

K. R. Gibson and T. Ingold. *Tools, Language and Cognition in Human Evolution.* 1993. ISBN 0521414741. doi: 10.2307/2804508. URL http://books.google.com/books?hl=en{&}lr={&}id=Cb1HMHirsBQC{&}oi=fnd{&}pg=PR11{&}dq=tim+ingold+human+tool+use{&}ots=nfvbZCt0hd{&}sig=X1VIVVU4sbMVkjee1QRh5-5dq-4.

G. Gigerenzer. *Bounded rationality: The adaptive toolbox.*, volume 79. 2003. ISBN 0262571641. doi: 10.1901/jeab.2003.79-409.

GNU. GNU Bash. URL https://www.gnu.org/software/bash/.

Google. AngularJS Superheroic JavaScript MVW Framework. URL https://angularjs.org/.

I. D. Greenwald. T h e SHARE 709 S y s t e m : Programming and Modification *. pages 128–133, 1958.

P. Groth and Y. Gil. A Scientific Workflow Construction Command Line. *International Conference on Intelligent User Interface 2009 (IUI2009)*, 2009. doi: 10.1145/1502650.1502716.

F. G. Halasz and T. P. Moran. December 1983 Mental Models and Problem Solving in Using a C a l c u l a t o r. *Chi 1983*, (December):212–216, 1983. doi: 10.1145/800045.801613.

B. Hamilton. A Sysadmin's Unixersal Translator (ROSETTA STONE), 2014. URL http://bhami.com/rosetta.html.

T. Harding. Snail Shells, 2016. URL http://www.tomharding.me/2016/12/24/fixing-the-shell/.

M. J. Heron. A Case Study into the Accessibility of Text-parser Based Interaction. In *Proceedings of the 7th ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS '15, pages 74–83, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3646-8. doi: 10.1145/2774225.2774833. URL http://doi.acm.org/10.1145/2774225.2774833.

T. Hewett. Designing with the human memory in mind. *ACM International Conference Proceeding Series*, 111:363–364, 2005. doi: 10.1145/1085777.1085869.

A. Howes. A model of the acquisition of menu knowledge by exploration. *Proceedings of the SIGCHI conference on Human factors in computing systems celebrating interdependence - CHI '94*, pages 445–451, 1994. doi: 10.1145/191666.191820. URL `http://portal.acm.org/citation.cfm?doid=191666.191820`.

E. Hutchins, J. Hollan, and D. Norman. Direct Manipulation Interfaces. *Human-Computer Interaction*, 1(4):311–338, 1985. ISSN 0737-0024. doi: 10.1207/s15327051hci0104_2.

M. Kampe. Guidelines for Command Line Interface Design. URL `http://www.cs.pomona.edu/classes/cs181f/supp/cli.html`.

B. W. Kernighan and J. R. Mashey. The UNIX??? programming environment. *Software: Practice and Experience*, 9(1):1–15, 1979. ISSN 1097024X. doi: 10.1002/spe.4380090102.

M. Kerrisk. Linux Programmer's Manual - Glob. URL `http://man7.org/linux/man-pages/man7/glob.7.html`.

G. Klein. Decision making in action: Models and methods, C.E. (eds). Norwood, NJ: Ablex, 1993, 480 pp. ISBN 089391794X (pb). *Journal of Behavioral Decision Making*, 8(3): 218–219, 1995. ISSN 08943257. doi: 10.1002/bdm.3960080307. URL `http://doi.wiley.com/10.1002/bdm.3960080307`.

G. Klein, D. D. Woods, J. M. Bradshaw, R. R. Hoffman, and P. J. Feltovich. Ten challenges for making automation a "team player" in joint human-agent activity. *IEEE Intelligent Systems*, 19(6):91–95, 2004. ISSN 15411672. doi: 10.1109/MIS.2004.74.

K. Kuuti. Activity Theory as a Potential Framwork for Human-Computer Interaction Research. In *Context and Consciousness: Activity Theory and Human-Computer Interaction*, pages 17–44. 1996. ISBN 9780262140584.

D. M. Lane, H. A. Napier, S. C. Peres, and A. Sandor. Hidden Costs of Graphical User Interfaces: Failure to Make the Transition from Menus and Icon Toolbars to Keyboard Shortcuts. *International Journal of Human-Computer Interaction*, 18(2):133–144, 2005. ISSN 1044-7318. doi: 10.1207/s15327590ijhc1802_1.

N. Marriott. tmux. URL `https://tmux.github.io/`.

Microsoft. Microsoft PowerShell. URL `https://msdn.microsoft.com/en-us/powershell/mt173057.aspx`.

R. C. Miller, V. H. Chou, M. Bernstein, G. Little, M. Van Kleek, D. Karger, and M. C. Schraefel. Inky: A Sloppy Command Line for the Web with Rich Visual Feedback. *Victoria*, pages 131–140, 2008. ISSN 159593975X. doi: 10.1145/1449715.1449737. URL `http://eprints.soton.ac.uk/266861/6/inky-video.mp4`.

B. Mirel. *Interaction Design for Complex Problem Solving.* Elsevier, 2004. ISBN 1558608311.

B. Moolenaar. welcome home : vim online. URL `http://www.vim.org/`.

S. R. Murillo and J. A. Sánchez. Empowering Interfaces for System Administrators: Keeping the Command Line in Mind when Designing GUIs. *Interaccion*, pages 0–3, 2014. doi: 10.1145/2662253.2662300. URL `http://doi.acm.org/10.1145/2662253.2662300`.

a. Newell, J. C. Shaw, and H. a. Simon. General Problem Solving?, 1969. ISSN 0028-0836. URL `http://www.nature.com/doifinder/10.1038/224923a0`.

D. Norman. The next UI breakthrough: command lines. *Interactions*, 14 (3):44–45, 2007. ISSN 1072-5520. doi: 10.1145/1242421.1242449. URL `http://portal.acm.org/citation.cfm?id=1242421.1242449{&}coll=ACM{&}dl=ACM{&}CFID=55881752{&}CFTOKEN=48540854{#}`.

D. A. Norman. The trouble with Unix: The user interface is horrid. *Datamation*, (December): 139–150, 1981.

D. A. Norman. The Psychology of Everyday Things. In *The Psychology of Everyday Things*, pages 1–104. 1988. ISBN 0465067093. doi: 10.2307/1423268.

D. A. Norman. Emotional design. *Ubiquity*, 2004(January):1–1, 2004. ISSN 15427331. doi: 10.1145/985600.966013.

R. C. Omanson, C. S. Miller, E. Young, and D. Schwantes. Comparison of Mouse and Keyboard Efficiency. *Proceedings of the Human Factors and Ergonomics Society*, pages 600–604, 2010. ISSN 1071-1813. doi: 10.1177/154193121005400612.

Oracle. Java Platform, Enterprise Edition (Java EE) — Oracle Technology Network — Oracle. URL `http://www.oracle.com/technetwork/java/javaee/overview/index.html{#}close`.

H. Pashler. Dual-task interference in simple tasks: Data and theory. *Psychological Bulletin*, 116(2):220–244, 1994. ISSN 0033-2909. doi: 10.1037/0033-2909.116.2.220.

S. J. Payne, G. B. Duggan, and H. Neth. Discretionary task interleaving: heuristics for time allocation in cognitive foraging. *Journal of experimental psychology. General*, 136 (3):370–388, 2007. ISSN 0096-3445. doi: 10.1037/0096-3445.136.3.370.

J. L. Peterson. Computer programs for detecting and correcting spelling errors. *Communications of the ACM*, 23(12):676–687, 1980. ISSN 00010782. doi: 10.1145/359038.359041.

Pivotal. Spring. URL `https://spring.io/`.

J. Preece, Y. Rogers, and H. Sharp. Interaction Design: Beyond Human-Computer Interaction. *Design*, 18(1):68–68, 2007. ISSN 00104485. doi: 10.1016/S0010-4485(86)80021-5. URL `http://linkinghub.elsevier.com/retrieve/pii/S0010448586800215`.

G. Rauch. Hyper. URL `https://hyper.is/`.

B. Shneiderman. Direct Manipulation: A Step Beyond Programming Languages. *Computer*, 16(8):57–69, 1983. ISSN 00189162. doi: 10.1109/MC.1983.1654471.

H. A. Simon and A. Newell. Human problem solving: The state of the theory in 1970. *American Psychologist*, 26(2):145–159, 1971. ISSN 0003-066X. doi: 10.1037/h0030806. URL `http://www.jstor.org/stable/2063712?origin=crossref{%}5Cnhttp://content.apa.org/journals/amp/26/2/145`.

D. Stenberg. curl. URL `https://curl.haxx.se/`.

T. Štolfa. The Future of Conversational UI Belongs to Hybrid Interfaces The Layer Medium, 2016. URL `https://medium.com/the-layer/the-future-of-conversational-ui-belongs-to-hybrid-interfaces-8a228de0bdb5{#}.b5o7hkm0m`.

L. Swartz. Why People Hate the Paperclip: Labels, Appearance, Behvavior and Social Responses to User Interface Agents. *Knowledge Creation Diffusion Utilization*, 2003.

E. Thorndike. The Law of Effect Author (s): Edward L . Thorndike Source : The American Journal of Psychology , Vol . 39 , No . 1 / 4 (Dec ., 1927), pp . 212-222 Published by : University of Illinois Press Stable URL : http://www.jstor.org/stable/1415413. 39(1): 212–222, 1927.

TLDP. GNU/Linux Command-Line Tools Summary - The command-line history. URL `http://www.tldp.org/LDP/GNU-Linux-Tools-Summary/html/x1712.htm`.

P. M. Todd and G. Gigerenzer. Putting naturalistic decision making into the adaptive toolbox. *Journal of Behavioral Decision Making*, 14(5):381, 2001. ISSN 1099-0771. doi: 10.1002/bdm.396. URL `http://doi.wiley.com/10.1002/bdm.396`.

L. Torvalds. Git. URL `https://git-scm.com/`.

G. van Rossum. Welcome to Python.org. URL `https://www.python.org/`.

T. Winograd and F. Flores. On understanding computers and cognition: A new foundation for design. A response to the reviews, 1987. ISSN 00043702.

J. O. Wobbrock, H. H. Aung, B. Rothrock, and B. A. Myers. Maximizing the guessability of symbolic input. *CHI'05 extended abstracts . . .*, pages 1869–1872, 2005. doi: 10.1145/1056808.1057043. URL `http://dl.acm.org/citation.cfm?id=1057043`.

C. E. Zsambok and G. Klein. Naturalistic Decision Making. *Human factors*, 50 (3):430, 1996. ISSN 0018-7208. doi: 10.1518/001872008X288385. URL `http://www.amazon.com/Naturalistic-Decision-Making-Expertise-Applications/dp/080581874X/ref=sr{_}1{_}1?s=books{&}ie=UTF8{&}qid=1325860512{&}sr=1-1`.

Appendix A

Code Listings

A.1 Suggestions Model for Experiments

```
[
  {
    "command": "help",
    "history": [
      "invalid"
    ]
  },
  {
    "command": "undo",
    "history": [
      "invalid",
      "snaps",
      "tension"
    ]
  },
  {
    "command": "look inventory",
    "history": [
      "pickup",
      "help",
      "invalid"
    ]
  },
  {
    "command": "look",
    "history": [
      "use",
      "help",
      "welcome",
      "help",
      "invalid"
    ]
  },
  {
    "command": "pickup key",
    "history": [
      "key",
      "door",
      "room",
      "reveals",
      "floats",
      "water",
      "tube",
      "snowglobe",
      "debris",
      "displaces",
      "stone",
      "crate",
      "open",
      "melt",
      "puddle"
    ]
  },
  {
    "command": "use key on door",
    "history": [
      "key",
      "pickup"
    ]
  },
  {
    "command": "use door",
    "history": [
      "unlock",
      "door",
      "key",
      "unlocks",
      "click",
      "heft"
    ]
  },
  {
    "command": "look door",
    "history": [
      "unlock",
      "door",
      "key",
      "room",
      "look"
    ]
  },
  {
    "command": "start",
    "history": [
      "congratulations",
      "escaped",
      "start"
    ]
  },
  {
    "command": "pickup cloth",
    "history": [
      "cloth",
      "look"
    ]
  },
  {
    "command": "use pail on tube",
    "history": [
      "pail",
      "tube",
      "door",
      "look",
      "room"
    ]
  },
  {
    "command": "look pail",
    "history": [
      "pail",
      "tube",
      "door",
      "look",
      "room"
    ]
  },
  {
    "command": "look tube",
    "history": [
      "pail",
      "tube",
      "door",
      "look",
      "room",
      "stone",
      "displaces",
      "insert",
      "water"
    ]
  },
  {
    "command": "look snowglobe",
    "history": [
      "snowglobe",
      "look",
      "room",
      "door"
    ]
  },
  {
    "command": "use snowglobe",
    "history": [
      "snowglobe",
      "look",
      "room",
      "door"
    ]
  },
  {
    "command": "break snowglobe",
    "history": [
      "snowglobe",
      "look",
      "room",
      "door"
    ]
  },
  {
    "command": "place weight on
        floorpad",
    "history": [
      "floorpad",
      "weight",
      "door"
    ]
  },
  {
    "command": "look weight",
    "history": [
      "floorpad",
      "weight",
      "door",
      "portcullis",
      "chain"
    ]
  },
  {
    "command": "look floorpad",
    "history": [
      "floorpad",
      "weight",
      "door"
    ]
  },
  {
    "command": "look portcullis",
    "history": [
      "portcullis",
      "chain",
      "weight",
      "room",
      "look"
    ]
  },
  {
    "command": "look chain",
    "history": [
      "portcullis",
      "chain",
      "weight",
      "room",
      "look"
    ]
  },
  {
    "command": "place weight on chain",
    "history": [
      "portcullis",
      "chain",
      "weight",
      "room",
      "look"
    ]
  },
  {
    "command": "use chain",
    "history": [
      "portcullis",
      "chain",
      "weight",
      "room",
      "look"
    ]
  },
  {
    "command": "place every weight on
        chain",
    "history": [
      "portcullis",
      "chain",
      "weight",
      "room",
      "look"
    ]
  },
  {
    "command": "use portcullis",
    "history": [
      "portcullis",
      "chain",
      "weight",
      "heavy",
      "opened",
      "hook"
    ]
  },
  {
    "command": "look stone",
    "history": [
      "stone",
      "tube",
      "door",
      "look"
    ]
  },
  {
    "command": "use stone on tube",
    "history": [
      "stone",
      "tube",
      "door",
      "look"
    ]
  },
  {
    "command": "use every stone on tube",
    "history": [
      "stone",
      "tube",
      "door",
      "look"
    ]
  },
  {
    "command": "use rope",
    "history": [
      "portcullis",
      "rope",
      "weight",
      "room",
      "look"
    ]
  },
  {
    "command": "look rope",
    "history": [
      "portcullis",
      "rope",
      "weight",
      "room",
      "look"
    ]
  },
  {
    "command": "place weight on rope",
    "history": [
      "portcullis",
      "rope",
      "weight",
      "room",
      "look",
      "place",
      "on"
    ]
  },
  {
    "command": "place every weight on
        rope",
    "history": [
      "portcullis",
      "rope",
      "weight",
      "room",
      "look"
    ]
  },
  {
    "command": "look ladder",
    "history": [
      "ladder",
      "hatch",
      "room"
    ]
  },
  {
    "command": "look hatch",
    "history": [
      "ladder",
      "hatch",
      "room"
    ]
  },
  {
    "command": "use ladder on hatch",
    "history": [
      "ladder",
      "hatch",
      "room"
    ]
  },
  {
    "command": "look crate",
    "history": [
      "crate",
      "door",
      "room"
    ]
  },
  {
    "command": "open crate",
    "history": [
      "crate",
      "door",
      "room"
    ]
  },
  {
    "command": "open every crate",
    "history": [
      "crate",
      "door",
      "room"
    ]
  },
  {
    "command": "break every crate",
    "history": [
      "crate",
      "door",
      "room"
    ]
  },
  {
    "command": "break crate",
    "history": [
      "crate",
      "door",
      "room",
      "nothing"
    ]
  },
  {
    "command": "look iceblock",
    "history": [
      "iceblock",
      "hairdrier",
      "door",
      "room",
      "look"
    ]
  },
  {
    "command": "break iceblock",
    "history": [
      "iceblock",
      "hairdrier",
      "door",
      "room",
      "look"
    ]
  },
  {
    "command": "look hairdrier",
    "history": [
      "iceblock",
      "hairdrier",
      "door",
      "room",
      "look"
    ]
  },
  {
    "command": "use hairdrier on
        iceblock",
    "history": [
      "iceblock",
      "hairdrier",
      "door",
      "room",
      "look"
    ]
  },
  {
    "command": "use hairdrier on every
        iceblock",
    "history": [
      "iceblock",
      "hairdrier",
      "door",
      "room",
      "look"
    ]
  },
  {
    "command": "use trapdoor",
    "history": [
      "stomp",
      "crate",
      "trapdoor",
      "splinters",
      "rubble"
    ]
  }
]
```

A.2 R Script for Statistical Analysis of Frequency Data

Courtesy of the Institute for Digital Research and Education (`http://stats.idre.ucla.edu/r/dae/poisson-regression/`).

```
p <- read.csv("~/Downloads/poisson-keys.csv")
p <- within(p, {
    interface <- factor(interface, levels=1:2, labels=c("CASHTU", "TU"))
    scenario <- factor(scenario, levels=1:13,
        labels=c("Clothkey","Pailtube","Snowglobe","Pressurepad","Weightgate","Water
    id <- factor(id)
  })
summary(p)
summary(m1 <- glm(num_X ~ interface + scenario, family="poisson",
    data=p))
```

```
cov.m1 <- vcovHC(m1, type="HC0")
std.err <- sqrt(diag(cov.m1))
r.est <- cbind(Estimate= coef(m1), "Robust_SE" = std.err,
"Pr(>|z|)" = 2 * pnorm(abs(coef(m1)/std.err), lower.tail=FALSE),
LL = coef(m1) - 1.96 * std.err,
UL = coef(m1) + 1.96 * std.err)
with(m1, cbind(res.deviance = deviance, df = df.residual,
  p = pchisq(deviance, df.residual, lower.tail=FALSE)))
## update m1 model dropping prog
m2 <- update(m1, . ~ . - interface)
## test model differences with chi square test
anova(m2, m1, test="Chisq")



##export model to csv
results_df <-summary(m1 <- glm(num_X ~ interface + scenario,
    family="poisson", data=p))$coefficients
write.csv(results_df, "~/Downloads/myCSV.csv")

##export chisq to csv
results_df <-anova(m2,m1,test="Chisq")
write.csv(results_df, "~/Downloads/myCSV.csv")
```

A.3 Collation Scripts for Experiment Results

```
var fs = require('fs');
var file =
    JSON.parse(fs.readFileSync('red-a.json',
    'utf8'));


var tasks = 8;

var printLine = (key,raw)=>{
  var line = "";
  for (var i = 0; i < tasks; i++) {
    var o = raw[key+i];
    if(typeof o != 'undefined'){
      line = line + o;
    }
    if(i!=tasks-1) line = line + "\t"
  }
  console.log(line);
}

var printIntentActGrid = (raw)=>{
  var dataRemains = true;
  var row = 0;
  while(dataRemains){
    dataRemains = false;
    var line = "";
    for (var i = 0; i < tasks; i++) {
      var list = raw["INTENT_ACT_"+i];
      if(typeof list != 'undefined'){
        var item = list[row];
        if(typeof item != 'undefined'){
          line = line + item;
          dataRemains = true;
        }
      }
      if(i!=tasks-1) line = line + "\t"
    }
    console.log(line);
    row ++;
  }

}

var printTimeDifference = (raw)=>{
  var line = "";
  for (var i = 0; i < tasks; i++) {
    var b = raw["TASK_TIME_BEGIN_"+i];
    var e = raw["TASK_TIME_END_"+i];
    if(typeof b != 'undefined' && typeof
        e != 'undefined'){
      line = line + (e-b);
    }
    if(i!=tasks-1) line = line + "\t"
  }
  console.log(line);
}

console.log("UNDONE");
file.forEach((item)=>{
  printLine("COMMANDS_UNDONE_",item);
});


console.log("HELP");
file.forEach((item)=>{
  printLine("HELP_REFERS_",item);
});

console.log("INTENTACT");
file.forEach((item)=>{
  printIntentActGrid(item);
});

console.log("INVALID");
file.forEach((item)=>{
  printLine("INVALID_SUBMITS_",item);
});

console.log("ARROWS");
file.forEach((item)=>{
  printLine("TASK_ARROWS_",item);
});

console.log("CHARS");
file.forEach((item)=>{
  printLine("TASK_CHARS_",item);
});

console.log("ENTERS");
file.forEach((item)=>{
  printLine("TASK_ENTER_",item);
});

console.log("SUCCEEDS");
file.forEach((item)=>{
  printLine("TASK_SUCCEED_",item);
});

console.log("TIME");
file.forEach((item)=>{
  printTimeDifference(item);
});
```

```
#!/usr/bin/env node

const read = require('read-directory')
const output = {}

const format = object =>
    Object.assign(object, {
  'task time begin': 0,

  'task time end': (object['task time
      end']
                    - object['task time
                        begin']) / 1000,

  'intent act': object['intent act']
                ? object['intent
                    act'].map(x => x /
                    1000)
                : undefined
})

const groups = {
  A: [],
  B: [],
  C: [],
  D: []
}

const groupcodes = [ "A","B","C","D"];

const groupscensfull = {
  A: [1,2,3,4,5,6,7],
  B: [8,9,10,11,12,13],
  C: [1,2,3,4,5,6,7],
  D: [8,9,10,11,12,13]
}
const groupscensred = {
  A: [8,9,10,11,12,13],
  B: [1,2,3,4,5,6,7],
  C: [8,9,10,11,12,13],
  D: [1,2,3,4,5,6,7]
}

read('Subjects', { dirnames: true },
    (err, contents) => {
  Object.keys(contents).forEach(identifier
      => {
    const [ name, part ] =
        identifier.split('/')

    if (!(name in output)) output[name]
        = {}
    output[name][part] = []

    const data =
        JSON.parse(contents[identifier])

    Object
    .keys(data)
    .filter(key =>
        key.match(/\d+_LAST_TS$/))
    .forEach(key => { delete data[key] })

    Object
    .keys(data)
    .map(key => {
      const components = key.split('_')
      const index =
          +components[components.length
          - 1]

      components.pop()
      const yek =
        components
        .map(x => x.toLowerCase())
      .join(' ')

    output[name][part][index] =
      output[name][part][index] || {}

    output[name][part][index][yek] =
        data[key]
  })

  output[name][part].forEach(format)
  output[name][part].shift()
})


Object
.keys(groups)
.forEach(group =>{
  groups[group]
  .forEach(part=>{
    groups[part]=group;
  })
})

var id = 1;
var metric = "invalid submits";

console.log("id,num_X,interface,scenario");

Object
.keys(output)
  .forEach(participant => {
    var i=0;
    var group = groups[participant];
    groupscensfull[group].forEach(scencode=>{
      if((typeof
          output[participant]["FULL"][i]!="undefined")
          output[participant]["FULL"][i][metric]!="unc
        var keys =
            output[participant]["FULL"][i][metric]
        console.log(id+","+keys+",1,"+scencode);
      }
      i++;
    })
    i=0;
    groupscensred[group].forEach(scencode=>{
      if((typeof
          output[participant]["RED"][i]!="undefined")&
          output[participant]["RED"][i][metric]!="unde
        var keys =
            output[participant]["RED"][i][metric]
        console.log(id+","+keys+",2,"+scencode);
      }
      i++;
    })


    id++;
  })
})
```

Appendix B

Statistical Analysis

Source	SS	df	MS	F	P
Presentation Order (r)	9098317.55	1	9098317.55	0.01	0.9205
Interface (c)	6356251720	1	6356251720	8.19	0.0048
r x c	5321080508	1	5321080508	6.86	0.0097
Error	125709327628	162	775983503.9		
Total	137395758174	165			

Table B.1: 2x2 Analysis of variance of task time on presentation order (groups AC vs BD) and interface (CAS+HS+TU vs TU).

	Estimate	Std. Error	z value	Pr(> $\|z\|$)
(Intercept)	3.9449	0.0346	114.1078	0.0000
interfaceTU	0.5768	0.0180	32.1154	0.0000
scenarioPailtube	0.3999	0.0418	9.5558	0.0000
scenarioSnowglobe	0.0289	0.0455	0.6363	0.5246
scenarioPressurepad	0.0978	0.0447	2.1872	0.0287
scenarioWeightgate	0.2822	0.0439	6.4333	0.0000
scenarioWatertube	0.4465	0.0415	10.7704	0.0000
scenarioRopeweight	0.6998	0.0447	15.6557	0.0000
scenarioLadderhatch	-0.3926	0.0517	-7.5972	0.0000
scenarioCratekey	-0.1021	0.0476	-2.1466	0.0318
scenarioHairdrier	-0.2001	0.0489	-4.0953	0.0000
scenarioCratetrapdoor	0.1381	0.0448	3.0860	0.0020
scenarioManyice	0.2222	0.0439	5.0624	0.0000
scenarioNestedcrates	0.3053	0.0442	6.9139	0.0000

Table B.2: Linear model of interface and scenario for keypresses per scenario.

	Resid. Df	Resid. Dev	Df	Deviance	Pr(>Chi)
1	148	4889.205467			
2	147	3814.332501	1	1074.872965	9.55E-236

Table B.3: Chi-squared analysis of deviance on models of keypresses with and without interface as a predictor.

	Estimate	Std. Error	z value	Pr(> $\|z\|$)
(Intercept)	1.823	0.112	16.261	0.000
interfaceTU	0.140	0.057	2.436	0.015
scenarioPailtube	0.370	0.139	2.657	0.008
scenarioSnowglobe	0.023	0.151	0.151	0.880
scenarioPressurepad	-0.096	0.155	-0.621	0.535
scenarioWeightgate	0.215	0.147	1.460	0.144
scenarioWatertube	0.476	0.137	3.485	0.000
scenarioRopeweight	0.713	0.148	4.816	0.000
scenarioLadderhatch	-0.312	0.165	-1.886	0.059
scenarioCratekey	-0.013	0.153	-0.082	0.935
scenarioHairdrier	-0.111	0.157	-0.712	0.477
scenarioCratetrapdoor	-0.001	0.152	-0.005	0.996
scenarioManyice	-0.048	0.154	-0.315	0.753
scenarioNestedcrates	0.459	0.140	3.287	0.001

Table B.4: Linear model of commands executed per scenario for interface and scenario.

	Resid. Df	Resid. Dev	Df	Deviance	Pr(>Chi)
1	148	170.7670673			
2	147	164.8168778	1	5.950189515	0.01471570398

Table B.5: Chi-squared analysis of deviance on models of number of commands per scenario with and without interface as a predictor.

Estimate	Std. Error	z value	Pr(> $\|z\|$)	
(Intercept)	-0.783	0.374	-2.096	0.036
interfaceTU	0.819	0.274	2.986	0.003
scenarioPailtube	-0.357	0.493	-0.724	0.469
scenarioSnowglobe	-0.916	0.592	-1.549	0.121
scenarioPressurepad	-0.357	0.493	-0.724	0.469
scenarioWeightgate	-0.401	0.516	-0.777	0.437
scenarioWatertube	-0.693	0.548	-1.266	0.206
scenarioRopeweight	-0.099	0.548	-0.180	0.857
scenarioLadderhatch	-1.144	0.659	-1.738	0.082
scenarioCratekey	-1.144	0.659	-1.738	0.082
scenarioHairdrier	-1.144	0.659	-1.738	0.082
scenarioCratetrapdoor	-0.297	0.493	-0.602	0.547
scenarioManyice	-1.550	0.775	-2.000	0.045
scenarioNestedcrates	-1.433	0.775	-1.849	0.064

Table B.6: Linear model of documentation checks performed for interface and scenario.

	Resid. Df	Resid. Dev	Df	Deviance	Pr(>Chi)
1	148	123.4167062			
2	147	113.6900884	1	9.726617765	0.001816183906

Table B.7: Chi-squared analysis of deviance on models of documentation checks per scenario with and without interface as a predictor.

Estimate	Std. Error	z value	Pr(> $\|z\|$)	
(Intercept)	-0.217	0.288	-0.752	0.452
interfaceTU	0.590	0.189	3.119	0.002
scenarioPailtube	0.470	0.329	1.428	0.153
scenarioSnowglobe	-1.322	0.563	-2.349	0.019
scenarioPressurepad	0.000	0.365	0.000	1.000
scenarioWeightgate	-0.527	0.438	-1.203	0.229
scenarioWatertube	-0.143	0.379	-0.378	0.706
scenarioRopeweight	-0.161	0.458	-0.353	0.724
scenarioLadderhatch	-2.664	1.033	-2.579	0.010
scenarioCratekey	-1.278	0.563	-2.270	0.023
scenarioHairdrier	-1.971	0.753	-2.618	0.009
scenarioCratetrapdoor	0.332	0.342	0.971	0.332
scenarioManyice	-1.278	0.563	-2.270	0.023
scenarioNestedcrates	-0.611	0.458	-1.334	0.182

Table B.8: Linear model of invalid commands performed for interface and scenario.

	Resid. Df	Resid. Dev	Df	Deviance	Pr(>Chi)
1	148	240.544938623533			
2	147	230.35468026161	1	10.1902583619235	0.00141184532315303

Table B.9: Chi-squared analysis of deviance on models of invalid commands per scenario with and without interface as a predictor.

Source	SS	df	MS	F	P
Presentation Order (r)	14035989.02	1	14035989.02	0.05	0.8231
Interface (c)	197274521.9	1	197274521.9	0.69	0.4063
r x c	25727235.9	1	25727235.9	0.09	0.7642
Error	386316565982	1357	284684278.5		
Total	386553603729	1360			

Table B.10: 2x2 Analysis of variance of times between each command on presentation order (groups AC vs BD) and interface (CAS+HS+TU vs TU).

Appendix C

Study Documents