



**Hochschule für Technik
und Wirtschaft Berlin**

University of Applied Sciences

*Entwicklung und Evaluation von Methoden zur Absenkung der Nutzungsschwelle
von Kommandozeilen-Interfaces*

Abschlussarbeit

zur Erlangung des akademischen Grades:

Bachelor of Science (B.Sc.)

an der

Hochschule für Technik und Wirtschaft (HTW) Berlin
Fachbereich 4: Informatik, Kommunikation und Wirtschaft
Studiengang *Angewandte Informatik*

1. Gutachter: Prof. Dr.-Ing. Johann Habakuk Israel
2. Gutachter: B.Sc. Moritz Wachter

Eingereicht von Jonathan Neidel (573619)

13. Februar 2023

Danksagung

An Endava für die Möglichkeit diese Arbeit im Betrieb zu schreiben.

An Moritz der dies ermöglicht hat und mir helfend zur Seite stand.

An Herr Israel für die Beantwortung aller aufkommenden Fragen.

An Sven für die kurzfristige Hilfe mit dem Abstract.

An Francis und meine Mutter für den moralischen Support.

Abstract

Diese Arbeit beschäftigt sich mit der Verbesserung von Anwendungen mit Kommandozeilen-Interfaces, kurz CLI Apps. Zur Optimierung deren Usability wurden die zwei zugrunde liegenden Probleme aufgezeigt. Es wurden anwendbare Methoden erarbeitet, welche diese Probleme mitigieren sollen.

Die Umsetzung dieser Methoden wurde an einer Beispielanwendung durchgeführt, um Details und Komplikationen in der Implementierung aufzuzeigen. Darüber hinaus wurden Performance und Attraktivität verglichen, indem die Kommandozeilen Anwendung einer grafischen gegenüber gestellt wurde.

Im Vergleich bevorzugten die sechs Teilnehmer die vergleichbare Performance als auch die bessere Nutzbarkeit des Kommandozeilen-Interfaces.

Inhaltsverzeichnis

1. Einleitung	1
1.1. Hintergrund der Arbeit	1
1.2. Problem- und Zielstellung	1
1.3. Aufbau der Arbeit	2
2. Grundlagen	3
2.1. Historischer Kontext	3
2.2. Vorteile der Kommandozeile	3
2.3. Die Terminal Umgebung	4
2.4. Begrifflichkeiten der Kommandozeile	4
2.5. Usability	5
3. Probleme des Kommandozeilen-Interface	6
3.1. Erinnern von Kommandos	6
3.2. Syntax und Semantik	6
3.3. Adressierung der Probleme durch Weiterentwicklungen	7
3.3.1. Innerhalb des Terminals	7
3.3.2. Außerhalb des Terminals	9
4. Entwicklung von Methoden	11
4.1. Methodologie	11
4.2. Flaggen anstatt Argumenten	11
4.3. Unterstützung aller Hilfs- und Versionsflaggen	13
4.4. Relevante Standardwerte	14
4.5. Interaktives Nachfragen bei fehlendem Parameter	15
4.6. Natürliche Sprache	16
4.7. Fehlerkorrektur	16
4.8. Autovervollständigung	17
4.9. Menü-basiertes Interface	17
4.10. Relevante Kommandovorschläge	18
5. Beispiel Anwendung	19
5.1. Anforderungsanalyse	19
5.1.1. Konzept	19
5.1.2. Anforderungen	20
5.2. Technische Aspekte	21
5.2.1. Fundament	21
5.2.2. Libraries	21
5.3. Implementierte Anwendung	23
6. Implementation der Methoden	28
6.1. Flaggen anstatt Argumenten	28

Inhaltsverzeichnis

6.2.	Unterstützung aller Hilfs- und Versionsflaggen	28
6.3.	Relevante Standardwerte	28
6.4.	Interaktives Nachfragen bei fehlendem Parameter	30
6.5.	Natürliche Sprache	30
6.6.	Fehlerkorrektur	31
6.7.	Autovervollständigung	31
6.8.	Menü-basiertes Interface	32
6.9.	Relevante Kommandovorschläge	32
7.	Evaluation	33
7.1.	Design der Studie	33
7.1.1.	Rahmenbedingungen	33
7.1.2.	Aufgabenstellungen	34
7.1.3.	Zu erhebende Daten	35
7.1.4.	Durchführung	36
7.2.	Hypothesen	36
7.3.	Auswertung	36
7.3.1.	Diskussion der Ergebnisse	36
7.3.2.	Auswertung der Hypothesen	40
7.3.3.	Beobachtungen aus der Evaluation	40
8.	Zusammenfassung	41
8.1.	Schlussfolgerungen	41
8.2.	Limitationen	41
8.3.	Ausblick	41
8.3.1.	Entwicklung und Strukturierung von Methoden	41
8.3.2.	Vergleich mit CLI App	41
8.3.3.	Verbesserung der Implementation	41
8.3.4.	Bearbeiten der Problemstellung in anderer Herangehensweise	42
	Quellenverzeichnis	43
9.	Glossar	i
A.	Appendix	ii
A.1.	Quell-Code	ii
A.2.	Rohdaten	ii

1. Einleitung

1.1. Hintergrund der Arbeit

Diese Arbeit setzt sich mit dem Kommandozeilen-Interface auseinander. Laut Raskin 2008 ist es “still one of the most powerful interface paradigms we have for controlling our computers.”. Gleichzeitig ist es aber auch als traditionelles Interface Teil der Computergeschichte (Nielsen 1993a). Damit einher gehen Probleme und Schwächen.

1.2. Problem- und Zielstellung

Das Kommandozeilen-Interface ist heutzutage ein wenig verbreitetes Anwendungs-Interface. Besonders in der allgemeinen Bevölkerung, aber selbst unter Entwicklern werden für die meisten Anwendungsfälle grafische Apps bevorzugt.

Das sich daraus ergebende Problem ist: Wie kann das Kommandozeilen-Interface Entwicklern oder der Allgemeinheit näher gebracht werden? Das Teilproblem im Fokus dieser Arbeit sind die Probleme in der Nutzbarkeit die ein Kommandozeilen-Interface mit sich bringen kann.

Ziel ist es implementierbare Methoden zum Verbessern der Usability von Anwendungen mit Kommandozeilen-Interface zu entwickeln. Die Anwendung dieser sollte die Nutzungsschwelle von Kommandozeilen Apps abzusenken und an die Usability von alternativen, meist grafischen Anwendungen angleichen. Um dies zu demonstrieren sollen die entwickelten Methoden in einem Anwendungsbeispiel implementiert und in einer Evaluation mit einer grafischen Alternative verglichen werden.

1.3. Aufbau der Arbeit

1. Einleitung

Definieren der Ziele und Struktur.

2. Grundlagen

Beschreiben grundlegenden Wissens.

3. Probleme des Kommandozeilen-Interface

Zusammenfassen von Problemen, welche das Nutzen von Kommandozeilen-Interfaces erschweren.

4. Entwicklung von Methoden

Formulieren von Methoden um die zuvor geschilderten Probleme adressieren.

5. Beispiel Anwendung

Definieren und Bauen einer CLI App als Beispiel zur Anwendung der Methoden.

6. Implementation der Methoden

Beschreiben von Details und Schwierigkeiten bei der Implementierung der Methoden.

7. Evaluation

Durchführen einer vergleichenden Studie zwischen der implementierten CLI App und einer funktionell gleichen¹ GUI Webapp.

8. Zusammenfassung

Ziehen einer Schlußfolgerung.

¹Im Kontext der Studie wird nur die von beiden Anwendungen überlappende Funktionalität getestet und verglichen.

2. Grundlagen

2.1. Historischer Kontext

Die Kommandozeile ist ein Produkt der Evolution von Computern. Die ersten Formen der Interaktion in Echtzeit kamen zusammen mit dem ‘teletypewriter’ (TTY), einem Schreibmaschinen-ähnlichem Gerät. Erstmals konnten Menschen ihre Befehle eingeben und die direkt die Resultate sehen. Das Papier und die mechanischen TTY’s wurden schließlich durch text-darstellende Displays und elektronische Tastaturen ersetzt. Diese Interaktionsform mit Tastatur und Textausgabe ist als **Kommandozeilen-Interface (CLI)** bekannt geworden (Nagarajan 2018, 35f). Und funktioniert wie folgt:

1. Das System fordert den Nutzer auf einen Programmaufruf zu schreiben. (‘Read’)
2. Das System führt den Befehl aus und zeigt das Resultat. (‘Eval’ und ‘Print’)
3. Diese Sequenz wiederholt sich nun unendlich. (‘Loop’)

Ein System das dieses Vorgehen implementiert, wird auch als REPL (‘Read-eval-print loop’) Umgebung bezeichnet.

Um die Limitation der Kommandozeile¹ zu adressieren, wurde unter anderem das grafische User Interface (**GUI**) entwickelt (Nielsen 1993a). Welches von dem text-basierten CLI zu dem heute allgegenwärtigen Fenster, ‘Icons’, Menüs und die Maus mit sich brachte²

2.2. Vorteile der Kommandozeile

Das CLI bietet Experten viel Flexibilität konstruieren eines (komplexen) Operation (Norman 1983). Ein Beispiel für diese Flexibilität wäre folgendes. Es sollen alle PDF Dateien aus dem Jahr 2021 gelöscht werden, diese haben etwa Namen wie: `20210819-Rechnung.pdf`. In der Kommandozeile kann dies mit `rm 2021*.pdf` erreicht werden. Im grafischen Interface müssten die Dateien markiert und gelöscht werden. Mit einer steigender Anzahl von Dateien steigt auch der Aufwand und die Chance einen Fehler zu machen. In der Shell funktioniert `rm` auch problemlos bei einer Million Dateien.

Ein weiter Vorteil für Experten liegt in dem Erstellen von Skripten. Mit Hilfe dieser können Operationen automatisiert werden. Wenn z.B. eine der Rechnungen aus dem letzten Paragraphen heruntergeladen wurde, kann diese mittels Skript umbenannt, in den richtigen Ordner verschoben und der To-do-liste ein Eintrag à la ‘Rechnung xy begleichen’ hinzugefügt werden. In einer graphischen Umgebung ist diese Art von präziser Automatisierung nicht so einfach möglich.

Experten mit viel Wissen und Erfahrung mit dem System wissen meist exakt welche Operation angewandt werden muss. In der Kommandozeile kann dies einfach und direkt

¹Im Kapitel 3 wird noch tiefer auf die Probleme eingegangen.

²Mehr zu den Weiterentwicklung des CLI in Kapitel 3.3.

2.3. DIE TERMINAL UMGEBUNG

eingetippt werden. Während selbst Experten in grafischen und Menü-basierten Interfaces sich immer durch die gleichen Menüs und Untermenüs klicken müssen.

Als Text-basiertes Interface kommt der Kommandozeile die Klarheit von Worten zugute. Abstrakte Konzepte wie Marxismus lassen sich nur schwer bis gar nicht in Bildern beschreiben (Raskin 2008), sind aber als Wort sehr umgänglich.

2.3. Die Terminal Umgebung

Die moderne Kommandozeile existiert nicht mehr in Isolation. Um auf diese in einem grafischen Fenstersystem zugreifen zu können, wird ein **Terminal Emulator** benötigt. Wie der Name schon preisgibt: es soll ein traditionelles text-basiertes Terminal nachgebildet werden. Als grafische Anwendung besteht aber der Vorteil die Maus zum Klicken oder Markieren verwenden zu können. In diesem Terminal läuft eine Shell. So nennt man das Programm, welches die Befehle entgegennimmt und deren Resultate ermittelt und ausgibt. Am weitesten verbreitet sind Unix Shells wie `bash` oder `zsh`. Die Sprache der Shell gesprochen ist ‘Shell Script’. Dessen Funktionsumfang variiert je nach Shell-Dialekt. Die meisten Unix Shells implementieren aber den POSIX Standard und teilen deshalb eine gewisse Grundfunktionalität. DOS und andere Windows Shells sind hier außen vor. Diese teilen weder Funktionalität noch Core Utilities mit den Unix Shells. Die `coreutils` sind Grundlegende Werkzeuge zur Datei- und Textmanipulation (*GNU core utilities* 2023), wie z.B. `ls`, `cat` oder `rm`.

Neben der Shell gibt es noch andere Kommandozeilen Umgebungen. So gibt es für viele Programmiersprachen eine Kommandozeile (z.B. für Node.js, Python oder Scala). In diesen können dann nach gleichem REPL Schema Befehle in der Programmiersprache interaktiv geschrieben und ausgeführt werden.

2.4. Begrifflichkeiten der Kommandozeile

Für ein Verständnis des CLI sind ein paar Begriffe zu definieren. Ein Shell Befehl kann auf seine Einzelteile herunter gebrochen werden:

```
1 $ cd Downloads
2 $ git commit -m "Initial commit"
```

Das `$` zu Beginn bezeichnet, dass nachfolgend Shell Code kommt. In der ersten Zeile ist `cd` das Kommando, was verwendet werden soll, gefolgt von einem Argument, etwas das dem Kommando übergeben wird, in diesem Falle der Name eines Ordners (names `Downloads`.) Die Reihenfolge ist dabei bei mehreren Argumenten entscheidend. In der zweiten Zeile wird dem `git` Kommando das Subkommando `commit` übergeben (Nagaranjan (2018), Prasad u. a. (2022).)

Nicht alle Kommandozeilen Apps haben Subkommandos. Dickey (2018) definiert zwei verschiedene Arten von CLI Anwendungen: “single and multi-command”. Also klassische ‘UNIX-Style’ Werkzeuge wie `cd`, `cp` oder `grep`. Und jene Apps mit Subkommandos wie `npm` oder `git`, die meist moderner und komplexer sind.

Zurück zu obigen `git` Befehl: Dem `commit` Subkommando wird die `-m` Flagge und mit ihr ein Parameter (die Nachricht `Initial commit`) übergeben. Die `-m` Flagge ist

2.5. USABILITY

die Kurzversion der langen `--message` Flagge. Kurze Flaggen bestehen immer aus Bindestrich und einem Buchstaben. Lange Flaggen starten mit zwei Bindestrichen gefolgt von einem oder mehr Wörtern (meist sind mehrere Wörter auch durch Bindestriche verbunden, z.B. `--non-interactive`.) Funktionell sind beide Flaggenvarianten identisch. Flaggen die keinen Parameter erfordern, werden als Boolean Flagge bezeichnet, deren An- oder Abwesenheit meist etwas an- oder abschaltet. Die Reihenfolge der Flaggen untereinander ist irrelevant (Nagarajan (2018), Prasad u. a. (2022).)

2.5. Usability

Nielsen (1993b, S. 26) definiert Usability als eine Kombination aus mehreren Attributen:

1. **Erlernbarkeit:** Das System sollte einfach erlernbar sein, sodass Nutzer schnell produktiv damit arbeiten können.
2. **Effizienz:** Sobald der Nutzer das System erlernt hat sollte ein hohes Maß an Produktivität möglich sein.
3. **Erinnerbarkeit:** Das System sollte einfach zu erinnern sein. Nach einem Zeitraum des nicht Nutzens sollte das System nicht erneut erlernt werden müssen.
4. **Fehler:** Das System sollte eine geringe Fehlerquote haben. Das heißt der Nutzer macht wenig Fehler und kann bei deren Auftreten gut mit ihnen umgehen.
5. **Befriedigung:** Das System sollte angenehm zu nutzen sein. Die Nutzer sind subjektiv befriedigt.

3. Probleme des Kommandozeilen-Interface

Die Usability Schwierigkeiten des Kommandozeilen-Interfaces lassen sich auf zwei fundamentale Probleme zurückführen.

3.1. Erinnern von Kommandos

‘Command Recall’ beschreibt die Problematik, dass ein Nutzer zum Verwenden eines Programmes in der Kommandozeile immer dessen Namen kennen muss. Dies gilt für Kommandos, deren Subkommandos und Flaggen, aber auch für die Reihenfolge von Argumenten (Raskin 2008).

Problem 1: Erinnern von Kommandos, Subkommandos, Flaggen und Argumenten.

Dieses fundamentale Problem bedeutet, dass sich die Nutzer immer erst mit einem Kommando auseinander setzen müssen, bevor sie dieses verwenden können. Auch spielt das Vergessen mit der Zeit eine große Rolle. Raskin (2008) weist auch auf teilweise geringe Einprägsamkeit von Befehle wie `tar -xzf FILE`¹ hin.



Abbildung 3.1.: Die Shell bietet nichts an. Ohne ein Kommando zu kennen, passiert nichts.

Gentner und Nielsen (1996) beschreiben auch den Fakt, dass es keinen einfachen Weg zum Auffinden von Kommandos gibt. Der Nutzer muss also den Namen des Programmes kennen um dieses zu nutzen oder auf dessen Hilfsseite oder ‘man page’ zugreifen zu können.

3.2. Syntax und Semantik

“Commands and associated parameters must be typed, maintaining the correct semantic content and syntactic form.” (Westerman 1997, S. 184)

Problem 2: Einhalten der richtigen Syntax and Semantik.

Die Kommandozeile gilt als sehr starr und wenig tolerant gegenüber imperfekter Syntax (Gentner und Nielsen 1996). Diese Syntaxfehler können vielerorts und sehr einfach auftreten. Probleme entstehen etwa bei:

¹Entpacken einer ‘gzipped tar’ Datei (`tar.gz`).

3.3. ADRESSIERUNG DER PROBLEME DURCH WEITERENTWICKLUNGEN

- Rechtschreibfehlern in Kommando, Subkommando, Flaggen oder übergebenen Dateien
- Durch das Weglassen benötigter Argumente bei Flaggen oder (Sub-) Kommandos
- Missachtung der Reihenfolge (von Argumenten oder Subkommandos, Flagge in Beziehung zu Subkommando oder Argument)
- Missachtung der Shell Regeln (etwa durch Weglassen von Anführungszeichen bei Argument mit Leerzeichen)

Vor allem, wenn das Wissen um die impliziten Regeln der Shell fehlen, können schnell komplizierte Fehler auftreten. Je nach Qualität der Fehlermeldung und Erfahrung reißen die Syntaxfehler den Nutzer mehr oder weniger aus dem Flow.

```
1 $ git comit -m "Add commit"
2 git: 'comit' is not a git command. See 'git --help'.
3
4 The most similar command is
5     commit
```

Listing 3.1.: Fehlerhafter Subkommando Name bei `git`. Mit hilfreicher Fehlermeldung.

Semantische Probleme können auch auftreten. Meist sind diese Folge eines Logikfehlers oder dem Durcheinanderbringen der Reihenfolge.

```
1 $ git add remote origin git@github.com:jneidel/oraclett.git
```

Listing 3.2.: Ein syntaktisch valider `git` Befehl, der aber nicht tut, was gemeint war.

Der Semantische Fehler im obigen Listing 3.2 liegt im durcheinanderbringen der `add` und `remote` Subkommandos. `git remote` ist zum Verwalten von ‘tracked repositories’. `git add` markiert eine Datei für den nächsten ‘commit’. Je nach Kontext haben `add` und `remote` eine andere Bedeutung. `git remote add` fügt eine neue ‘repository’ hinzu während `git add remote` aber eine Datei mit dem Namen `./remote` für den nächsten ‘commit’ markiert.

Durch eine kleine Veränderung der Reihenfolge entsteht eine völlig andere Bedeutung.

3.3. Adressierung der Probleme durch Weiterentwicklungen

Die Lösungen zu den Problemen der Kommandozeile wurden vielfach in der Schöpfung neuer Interface Typen gesucht, ob innerhalb des Terminals oder außerhalb der Text-basierten Welt.

3.3.1. Innerhalb des Terminals

Obwohl diese Evolutionen in der Kommandozeilenumgebung blieben, brachen sie trotzdem mit der klassischen CLI Tradition. Die Unterscheidungen zwischen CLI und dem menu-basierten bzw. interaktivem Interface sind dabei trotzdem nicht glasklar, sondern eher verschwommen (Paap und Roske-Hofstrand 1988). Auch weil Aspekte des CLI (wie Hilfsseiten/man pages, Flaggen, etc.) weiterhin Teil dieser Applikationen sind. Auch weil

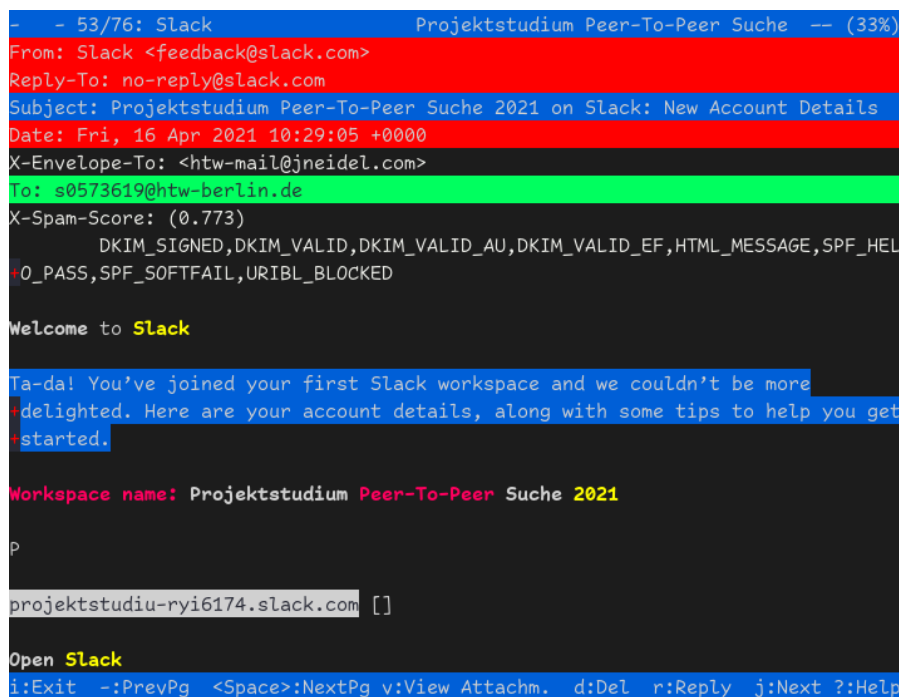
3.3. ADRESSIERUNG DER PROBLEME DURCH WEITERENTWICKLUNGEN

sich die verschiedenen Interfaces ergänzen und zusammen besser sein können als in ihrer Reinform.

Bland u. a. (2007) beispielsweise implementieren neben dem reinen nicht-interaktiven CLI Workflow noch einen menü-basierten Modus in welchem der Nutzer mit “menus similar to those of the GUI” durch den Prozess geführt wird.

Menü-basiertes Interface

Das Menü oder auch ‘Text-based user interface’ (TUI) ähnelt dem GUI insofern, als dass dem Nutzer die Optionen visuell präsentiert werden. Anders als beim GUI werden dafür aber nur Textelemente verwendet. Mausinteraktion wird von modernen TUI’s aber unterstützt. Da zur Implementation historisch oft die ‘curses’ Bibliothek verwendet wurde, ist es auch als ‘curses’ Interface bekannt.



```
- 53/76: Slack      Projektstudium Peer-To-Peer Suche -- (33%)
From: Slack <feedback@slack.com>
Reply-To: no-reply@slack.com
Subject: Projektstudium Peer-To-Peer Suche 2021 on Slack: New Account Details
Date: Fri, 16 Apr 2021 10:29:05 +0000
X-Envelope-To: <htw-mail@jneidel.com>
To: s0573619@htw-berlin.de
X-Spam-Score: (0.773)
DKIM_SIGNED,DKIM_VALID,DKIM_VALID_AU,DKIM_VALID_EF,HTML_MESSAGE,SPF_HELO_PASS,SPF_SOFTFAIL,URIBL_BLOCKED

Welcome to Slack

Ta-da! You've joined your first Slack workspace and we couldn't be more
delighted. Here are your account details, along with some tips to help you get
started.

Workspace name: Projektstudium Peer-To-Peer Suche 2021

projektstudiu-ryi6174.slack.com []

Open Slack
i:Exit  -:PrevPg  <Space>:NextPg v:View Attachm.  d:Del  r:Reply  j:Next ?:Help
```

Abbildung 3.2.: Beispiel eines TUI: Emails im Terminal mit neomutt.

Laut Paap und Roske-Hofstrand (1988) wandelt das Menü das Problem des ‘Command Recall’ in ‘Command Recognition’, ein Wiedererkennen, um. Auch werden die Anfälligkeit für Syntaxfehler reduziert, weil der Nutzer vor illegalen Optionen abgeschirmt wird (Kantorowitz und Sudarsky 1989).

Im einem vergleichenden Experiment mit der Kommandozeile stellte Westerman (1997) für manche Nutzergruppen unsignifikant bessere Performance fest. Auch wurde bei freier Wahl das Menü über alle Nutzergruppen hinweg doppelt so häufig zur Nutzung ausgewählt.

Verglichen mit dem GUI sind TUI’s trotzdem für Anfänger trotzdem schlechter (in Performance und Meinung). Bei Experten liegen beide gleichauf (Chen und Zhang 2007).

3.3. ADRESSIERUNG DER PROBLEME DURCH WEITERENTWICKLUNGEN

Interaktive CLI

```
Hi, welcome to Node Pizza
? Is this for delivery? Yes
? What's your phone number? 030 1234567
? What size do you need? Medium
? How many do you need? 2
? What about the toppings? Pepperoni and cheese
? You also get a free 2L beverage
1) Pepsi
2) 7up
3) Coke
Answer: 
```

Abbildung 3.3.: Ein Interaktives CLI leitet durch eine Pizza Bestellung.

Das Interaktive CLI basiert auf einem “question and answer model” (Spolsky 2001, S. 42). Dem Nutzer werden kontinuierlich Fragen gestellt auf die er antwortet (siehe Abbildung 3.3.) Durch die hilfreichen Fragen entfällt der Bedarf sich an Kommandos erinnern zu müssen (vgl. ‘Command Recall’.)

Je nach Bedarf kann der Nutzer nach einfachem Text-Input oder Zahlen gefragt werden. Auch komplexeres, wie das Auswählen aus einer Liste oder ‘multiple-choice’ Fragen sind möglich (vgl. Abbildung 3.4.)

```
? Select toppings (Press <space> to select,
= The Meats =
o Pepperoni
● Ham
>o Ground Meat
o Bacon
= The Cheeses =
● Mozzarella
(Move up and down to reveal more choices)
```

Abbildung 3.4.: Der Belag einer Pizza wird interaktiv zusammengestellt.

3.3.2. Außerhalb des Terminals

Historisch wurde die grafische Benutzeroberfläche als Lösung für die Probleme des CLI gesehen (vgl. Kap. 2.1.) So wurden dem Nutzer die möglichen Optionen grafisch präsentiert, anstatt diese nachschauen zu müssen. (vgl. ‘Command Recall’.)

Es vollzog sich damit ein Wechsel von der funktionsorientierten Kommandozeile hin zum objektorientierten GUI (Nielsen 1993a). Die Funktionsorientiertheit drückt sich etwa durch die in ‘single-command’ CLI Apps häufig vertretene Verb-Nomen Struktur aus. Beispielsweise in `rm FILE` oder `cat FILE`. In der objektorientierten Welt gehen alle Aktionen vom Objekt aus. So wird eine Datei in den Papierkorb gezogen (vgl. `rm FILE`) oder per Doppelklick angezeigt (vgl. `cat FILE`). Semantische Probleme (wie mit `mv` die Reihenfolge durcheinander zu bringen) sind damit passé. Und von der Textform befreit, sind es auch die in der Kommandozeile prävalenten Syntaxfehler.

Aber auch das grafische User Interface hat seine Schwächen. Und diese werden vor allem ‘at scale’ sichtbar. Die Massen des Internets lassen sich nicht grafisch darstellen

3.3. ADRESSIERUNG DER PROBLEME DURCH WEITERENTWICKLUNGEN

und auch ein volles Email Postfach ist mit nur grafischen Werkzeugen wenig durchsichtig. Norman (2007) beschreibt wie Suchmaschinen hier einen Ausweg bieten. Und nennt sie ‘answer engine’. Eine flexiblere, robustere und getarnte Kommandozeile die mit Rechtschreibfehlern und Synonymen umgehen kann.

4. Entwicklung von Methoden

4.1. Methodologie

Nach Betrachtung der Probleme sollten in Literaturrecherche Lösungsansätze erarbeitet werden, welche zusammen mit einer Erklärung als Methoden formuliert wurden.

Auf der Abstraktionsskala von Softwarekonzepten befindet sich die Methode in der Mitte zwischen abstrakt und konkret.

```
1 abstrakt <-----> konkret
2 Prinzip <---> Methode <---> Werkzeug
```

Listing 4.1.: *Abstraktionsskala von Softwarekonzepten.*

Das abstrakte und allgemeine Prinzip (Balzert 2009) wäre zu philosophischer Natur gewesen um im nächsten Schritt angewandt zu werden. Werkzeuge nach Balzert (2009) sind zu konkret und real. Es nicht Ziel für jeden Lösungsansatz ein direkt anwendbares Werkzeug zu kreieren.

Eine Methode nach Balzert (2009) ist eine begründete Vorgehensweise zur Erreichung festgelegter Ziele. Sie ist nicht deskriptiv wie ein Werkzeug, und nicht philosophisch wie ein Prinzip, sondern gibt eine Handlungsanweisung die dem Entwickler Spielraum in der Implementierung lässt.

Die zu formulierenden Methoden sollten sich daher: auf Probleme die durch sie gelöst/gemindert werden beziehen. Und begründet darstellen, warum dieser Effekt eintreten wird.

Nachfolgend nun die erarbeiteten Methoden.

4.2. Flaggen anstatt Argumenten

Sobald ein (Sub-) Kommando mehr als ein Argument annimmt, bietet sich die Gelegenheit, die für Argumente relevante Reihenfolge durcheinander zu bringen (vgl. Syntax/Semantik.) Dickey (2018) empfiehlt deshalb bei zwei oder mehr Argumenten anstatt dieser Flaggen zu verwenden. Das ist etwas mehr Aufwand zum Schreiben, eliminiert aber die Reihenfolge als Fehlerfaktor. Außerdem können Flaggen auch mittels Autovervollständigung vorgeschlagen werden (vgl. Methode 7).

Methode 1: Verwende Flaggen, wenn mehr als ein Argument benötigt wird.

Dies trifft vor allem zu wenn ein (Sub-) Kommando gleichzeitig Parameter über Flaggen und Argumente entgegennimmt.

```
1 # Argument und Flaggen Kombination
2 $ oraclett project add INTPD999DXD --taskDetail "01 - Career development"
```


4.2. FLAGGEN ANSTATT ARGUMENTEN

```
3
4 # Nur Flaggen
5 $ oraclett project add --project INTPD999DXD --taskDetail "01 - Career development"
```

Listing 4.2.: Argument ersetzt durch Flaggen: vorher und nachher

Außen vor sind Kommandos welche eine variable Anzahl von Argumenten annehmen. Beispielsweise `rm` dem mehrere Dateien zum Löschen übergeben werden können. Hier handelt es sich aber nicht um Argumente mit verschiedenen Bedeutungen (Dickey (2018).)

Ein weiter Vorteil ist das Erstellen von Aliassen zu vereinfachen. Das sind Abkürzungen, die ein Nutzer sich innerhalb seiner Shell definieren kann. Etwa um ein Kommando abzukürzen: `alias v="nvim"` macht `nvim` mit dem Kommando `v` nutzbar.

Man stelle sich vor eine CLI App nimmt eine Anzahl von Stunden und ein Datum:

```
1 $ app HOURS DATE
```

Der Nutzer möchte einen Alias mit dem für den heutigen Tag eine Anzahl von Stunden übergeben wird. Gewünschte Nutzung des Alias sieht so aus:

```
1 $ apptoday 8
```

Um diesen Alias zu ermöglichen muss aber eine kompliziertere¹ Shell Funktion definiert werden, weil das Argument an die richtige Stelle übergeben werden muss:

```
1 apptoday() {
2   app $1 today
3 }
```

Würde `app` anstatt der Argumente Flaggen annehmen:

```
1 $ app --hour HOURS --date DATE
```

Dann wäre der Alias viel leichter und flexibler zu definieren:

```
1 $ alias apptoday="app --date today --hour"
```

Wenn die Stunde anstatt des Tages durch den Alias festgeschrieben werden soll, ist das mit der Flaggen-basierten Struktur auch kein Problem. Der Nutzer ist unabhängig von der durch den Entwickler definierten Reihenfolge.

¹Neulinge kommen mit der Alias Syntax meist früher in Kontakt. Neben unvertrauter Syntax spielt mit hinein das Shell Argumente (`$1`) verwendet werden müssen und das die Möglichkeit entfällt dynamisch weitere Flaggen zu übergeben.

4.3. Unterstützung aller Hilfs- und Versionsflaggen

Der Großteil aller Apps bietet eine Hilfs- und eine Versionsseite. Hilfsseiten beschreiben ähnlich wie eine ‘man page’² die Subkommandos, Flaggen und Syntax. Meist sind diese auch relativ und beziehen sich auf das aktuelle Subkommando. Versionsseiten geben die Version der App³ wieder.

```

1 $ mullvad status --help
2 mullvad-status
3 View the state of the VPN tunnel
4
5 USAGE:
6     mullvad status [OPTIONS] [SUBCOMMAND]
7
8 OPTIONS:
9     --debug          Enables debug output
10    -h, --help        Print help information
11    -l, --location    Prints the current location and IP. Based on GeoIP lookups
12    -v               Enables verbose output
13
14 SUBCOMMANDS:
15    listen           Listen for VPN tunnel state changes

```

Listing 4.3.: Als Beispiel: Die gekürzte, relative Hilfsseite von `mullvad status`.

Beim Aufrufen dieser Seiten haben Nutzer ihre Präferenzen wenn es darum geht welche Flagge sie verwenden um diese angezeigt zu bekommen.

```

1 $ app -h
2 $ app --help
3 $ app help
4
5 $ app -v
6 $ app -V
7 $ app --version
8 $ app -version
9 $ app version

```

Listing 4.4.: Gängige Varianten der Hilfs- und Versionsflaggen.

Laut Dickey (2018) sollten möglichst alle Varianten in von einer Anwendung unterstützt werden. So kann vermieden werden, dass der Nutzer nicht das angezeigt bekommt, wonach er sucht. Das Problem des ‘Command Recall’ wird vermieden, weil der Nutzer durch Probieren seiner präferierten Variante direkt zur Lösung kommt.

Methode 2: Unterstütze alle gängigen Formen der Hilfs- und Versionsflaggen.

In einer unrepresentativen Stichprobe (siehe Tabelle) wurde zusammengestellt, welche Flaggen in 15 Anwendungen funktionieren bzw. nicht funktionieren. Dabei stellte sich heraus, dass `-h` und `--help` praktisch überall funktionieren. Mit `--version` wird in den

²Einer über das `man` Kommando verfügbaren Hilfsseite.

³Sowie ggf. eine Liste von optionalen Features die mit einkompiliert wurden.

4.4. RELEVANTE STANDARDWERTE

Kommando	-h	--help	help	-v	-V	--version	-version	version
git	☑	☑	☑	☑	☐	☑	☐	☑
node	☑	☑	☐	☑	☐	☑	☐	☐
mullvad	☑	☑	☐	☐	☐	☐	☐	☑
grep	☐	☑	☐	☐	☑	☑	☐	☐
zathura	☑	☑	☐	☑	☐	☑	☐	☐
tsp	☑	☐	☐	☐	☑	☐	☐	☐
pubs	☑	☑	☐	☑	☐	☑	☐	☐
ffmpeg	☑	☑	☐	☐	☐	☐	☑	☐
systemctl	☑	☑	☐	☐	☐	☑	☐	☐
pacman	☑	☑	☐	☐	☑	☑	☐	☐
make	☑	☑	☐	☑	☐	☑	☑	☐
synctex	☑	☑	☑	☑	☑	☑	☑	☑
curl	☑	☑	☐	☐	☑	☑	☐	☐
ansible	☑	☑	☑	☐	☐	☑	☐	☐
nvim	☑	☑	☐	☑	☐	☑	☑	☐
Summe	14	14	3	7	5	12	4	3

Tabelle 4.1.: Unterstützte Varianten von Hilfs- und Versionsflaggen bei einer Stichprobe von 15 Kommandos.

meisten Fällen auch das gewünschte Ergebnis angezeigt. Aber wenn `--version` nicht zum Ziel führt ist durchprobieren angesagt.

Am wenigsten verbreitet sind `help` und `version`, welche aber auch etwas außen vor sind, weil das erste Argument bei vielen ‘single-command’ Kommandos eine ‘Input’ Datei bezeichnet. Aber bei ‘multi-command’ Apps sollten auch diese Varianten unterstützt werden.

4.4. Relevante Standardwerte

Relevante ‘Defaults’ sind nicht Kommandozeilen-spezifisch, können dort aber wichtiger sein als in grafischen Anwendungen.

Dem Nutzer werden die Möglichkeiten eben nicht zur Auswahl gestellt und dieser klickt die gewünschte Option an. Sondern ist das Auflisten der Möglichkeiten i.d.R. ein separater Schritt, losgelöst von der eigentlich beabsichtigten Aktion. Beispielsweise muss um sich mit `cd` in einen Ordner hineinzubewegen der Name dessen zuerst in Erfahrung gebracht werden (etwa mit `ls -d`.)

Es greift wieder das fundamentale Problem des ‘Command Recall’. Den Schritt des Auflistens der Möglichkeiten (ähnlich wie das Auflisten von möglichen Kommandos) ist ein extra Schritt der nur nötig ist weil der Nutzer sich nicht an die Option erinnern kann die er verwenden möchte.

Methode 3: Biete relevante Standardwerte an.

Dem Nutzer relevante Defaults anzubieten erlaubt es diesem weniger Argumente an die CLI App übergeben zu müssen. Es gibt weniger Fehlermeldungen die den Nutzer

4.5. INTERAKTIVES NACHFRAGEN BEI FEHLENDEM PARAMETER

zurückweisen, weil dieser etwas vergessen hat.

Es ist aber auch elementar die Nutzung dieser Standardwerte zu kommunizieren. Damit der Nutzer von ihnen nicht überrascht wird.

4.5. Interaktives Nachfragen bei fehlendem Parameter

Man stelle sich folgendes Szenario vor. Eine CLI App hat ein Subkommando welches einen Parameter erfordert. Normalerweise erfolgt bei nicht-Übergabe dieses Parameters eine Fehlermeldung welche darauf hinweist.

```
1 $ mullvad relay set location
2 error: The following required arguments were not provided:
3     <country>
4
5 USAGE:
6     mullvad relay set location <country> [ARGS]
7
8 For more information try --help
```

Listing 4.5.: Fehlermeldung bei fehlendem Parameter in mullvad's CLI.

Die Anwendung weiß aber bereits was der Nutzer tun möchte und könnte anstatt der Fehlermeldung dem Nutzer mit einer Eingabeaufforderung (interaktivem Nachfragen, vgl. Kapitel 3.3.1) nach dem fehlenden Parameter fragen (Dickey 2018). Oder, je nach Kontext, diesem sogar eine Liste von Vorschlägen geben, um daraus auswählen. Dem Nutzer wird erspart sich damit auseinander zu setzen welche Parameter erfordert sind (vgl. 'Command Recall')

Methode 4: Frage bei fehlendem Parameter interaktiv nach.

```
1 $ mullvad relay set location
2 Enter location country code:
```

Listing 4.6.: Beispiel der Eingabeaufforderung zur Verbesserung von Listing 4.5.

Fairerweise ist zu bemerken das die mullvad CLI App dies auch, an anderer Stelle, genau so handhabt. Bei `mullvad account login` wird nach der fehlenden Accountnummer gefragt.

Mit einer Fehlermeldung anstatt des interaktivem Nachfragens könnte der Nutzer leichter auf die ggf. komplexere Verwendung des Kommandos hingewiesen werden. So nimmt das mullvad Kommando aus dem ersten Beispiel neben der Landeskennung auch noch optional die ein Stadt- und 'Hostkennung'. Was aber aus der Fehlermeldung auch nicht klar hervorgeht. Wenn unter Berücksichtigung dieser Methode implementiert wird kann dieser Einwand aber auch mitigiert werden. Entweder durch das Vereinfachen oder Umgestalten der Kommandos. Oder mit Hinweisen auf die Hilfsseite, welche die Parameter vollumfänglich erklärt.

4.6. Natürliche Sprache

Raskin (2008) beschreibt die Kapazitäten linguistischer Kommandozeilen, welche Normans Konzept der ‘answer engines’ (vgl. Kapitel 3.3.2) ähneln. Diese haben wie der Name schon hergibt einen Fokus auf menschlicher Sprache⁴. Ein schönes, genanntes Beispiel von Google Calendar für die Stärke solcher, auf natürlicher Sprache basierender Interfaces: “Sunday dinner at 7:30 p.m. with Asa Jasa.” Es werden keine Kommandos verwendet. Das gewünschte wird einfach in natürlicher Sprache beschrieben. Syntax ist nur insofern relevant, als das die menschliche Sprache ihre eigene Syntax hat.

“Recently, there has been a growing movement that sees today’s command line as a human-first text-based UI, rather than a machine-first scripting platform (Prasad u. a. 2022)” - (Schröder und Cito 2021)

Die Syntax- und Semantikeinschränkungen des Kommandozeilen-Interfaces lassen eine alleinig ‘human language’-gestützte Anwendung nicht zu. Zumindest nicht der Reinform des CLI. Als ‘human-first’ User Interface sollte trotzdem zumindest eine eingeschränkte Form natürlicher Spracheingabe unterstützt werden (Seneviratne 2008).

Methode 5: Unterstütze Eingaben in natürlicher Sprache.

4.7. Fehlerkorrektur

Diese Methode zeigt eine gewisse Nähe zur Unterstützung der natürlichen Sprache (Kapitel 4.6). So hob Raskin (2008) für die linguistische Kommandozeile die Verträglichkeit von Rechtschreibfehlern hervor.

Fehlertoleranz im Bezug auf Rechtschreibung würde auch das Problem der zu strikten Syntax zum Teil adressieren. Fehlermeldungen und die damit einhergehende Frustration minimieren. Prasad u. a. (2022) empfehlen ganz konkret: “If the user did something wrong and you can guess what they meant, suggest it.” Im Kontrast dazu soll nach der ‘Do what I mean’ Philosophie der Fehler direkt und ohne Nutzerfeedback ausgebessert werden (Raymond 2023). Was natürlich schneller ist und den Nutzungsflow nicht unnötig unterbricht. Prasad u. a. (2022) weisen aber auch darauf hin das eine fehlerhafte Eingabe neben Rechtschreibfehlern eben auch aus logischen Fehler resultieren kann. Dann kann ‘Do what I mean’ zu sehr unerwartetem, gefährlichem Verhalten führen.

Methode 6: Unterstütze (suggestive) Fehlerkorrektur.

```

1 heroku pss
2 > Warning: pss is not a heroku command.
3 Did you mean ps? [Y/n]:

```

Listing 4.7.: Korrekturvorschlag eines Rechtschreibfehlers nach Prasad u. a. (2022).

⁴Die Linguistik ist das Studium der menschlichen Sprache.

4.8. Autovervollständigung

‘Auto-completion’ in der Kommandozeile beschreibt das beim Drücken der ‘TAB’ Taste etwas ergänzt/vervollständigt wird.

In der Shell funktioniert dies etwa für Kommandos und Dateipfade:

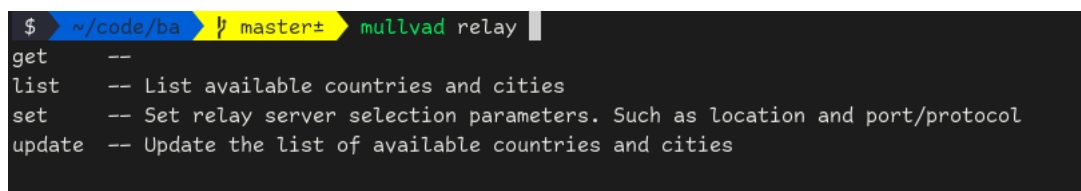
```

1 $ wg # Nutzer drückt TAB
2 $ wget
3
4 $ cat ./note # Nutzer drückt TAB
5 $ cat ./notes.md

```

Listing 4.8.: Autovervollständigung in der Shell.

Innerhalb einer CLI App kann die Vervollständigung, neben Dateipfaden, auch Subkommandos und Flaggen umfassen.



```

$ ~/code/ba master+ mullvad relay
get      --
list     -- List available countries and cities
set      -- Set relay server selection parameters. Such as location and port/protocol
update   -- Update the list of available countries and cities

```

Abbildung 4.1.: mullvad’s CLI listet nach dem drücken von ‘TAB’ mögliche Subkommandos.

‘Auto-completion’ adressierte beide Probleme welche das CLI mit sich bringt. Durch das Vorschlagen und Vervollständigen wird das Erinnern erleichtert. Und dadurch das nur valide Kommandos vorgeschlagen werden ist das einhalten von Syntax- und Semantikregeln erleichtert. Auch profitiert der Nutzer von Kontext-bezogenen Beschreibungen.

Methode 7: Nutze Autovervollständigung zum Vorschlagen von Flaggen und Subkommandos (Seneviratne 2008).

In Dutta (o.D.) geht der Author sogar noch einen Schritt weiter und empfiehlt das automatische Vorschlagen aller Flaggen und Argumente für ein Kommando, sodass der Nutzer sich nicht mal anschauen muss was für das Kommando benötigt wird. So wie vom Author beschrieben ist dies aber nicht in der Shell möglich, sondern müsste in einer eigenen Kommandozeilen Umgebung umgesetzt werden.

4.9. Menü-basiertes Interface

Im Kapitel 3.3.1 wurde schon darauf eingegangen wie ein TUI die Probleme der Kommandozeile adressieren würde. Auch wurde eine klare Präferenz für ein solches Interface unter den Nutzern festgestellt (Westerman 1997). Auch Spolsky (2001) nennt das Menü-basierte Interface als einen Weg Usability zu erhöhen.

Methode 8: Biete ein Menü-Interface an.

4.10. Relevante Kommandovorschläge

Die Essenz von ‘Command Recall’ ist das Nutzer nicht wissen welche (Sub-) Kommandos ihnen zur Verfügung stehen. Kommandos existieren aber nicht im Vakuum. Im Zusammenspiel unterliegen sie meist einer gewissen Reihenfolge bzw. einem ‘Nutzungsflow’. Bevor in einen Ordner gewechselt werden kann (`cd`) muss dieser erst existieren (`mkdir`) oder der Nutzer muss um dieses wissen (`ls`). Dutta (o. D.) schlägt nun vor sich diesen ‘Flow’ zu nutze zu machen um den Nutzer über mögliche nächste Schritte (in Form anderer (Sub-) Kommandos) zu informieren. Für den gegebenen ‘Flow’ könnte also `mkdir` mit einer Nachricht à la `Also look at - cd, ls` versehen werden. Diese Vorschläge können als Erinnerung dienen oder den Nutzer dazu Auffordern sich die Vorschläge einmal näher anzuschauen. Bei ‘multi-command’ CLI Apps mit vielen Subkommandos, wo der Nutzer nicht weiß wo er anfangen soll, sind diese Vorschläge besonders wertvoll.

Methode 9: Schlage dem Nutzer andere relevante Subkommandos vor (Seneviratne 2008).

5. Beispiel Anwendung

5.1. Anforderungsanalyse

Die zu bauende Anwendung wurde so konzipiert das diese weder zu trivial noch zu komplex ist. Nicht zu trivial damit das Interface auch wirklich etwas leisten muss und nicht zu simpel sein kann. Aber auch nicht zu komplex um die Umsetzbarkeit innerhalb des Zeitrahmens zu gewährleisten.

5.1.1. Konzept

Bei Endava¹ werden die Arbeitsstunden über eine interne Webapp (ein Oracle System) festgehalten. Diese Webapp wird im Laufe der Arbeit als **Oracle** bezeichnet².

INTPD999DXD - People Development DXD trainings timecards 01 - Career development Regular	
	40.00 Hours
	Mon,Jan 02; 8.00 Hours
	Tue,Jan 03; 8.00 Hours
	Wed,Jan 04; 8.00 Hours
	Thu,Jan 05; 8.00 Hours
	Fri,Jan 06; 8.00 Hours
Comments	
Bachelorarbeit	

Abbildung 5.1.: Idealtypische Oracle ‘Timecard’.

Am Ende der Arbeitswoche werden die Arbeitsstunden der vergangen Woche in einer sogenannten ‘Timecard’ festgehalten. Dabei wird angegeben an welchen Wochtage wieviele Stunden an welchen Projekten gearbeitet wurde. Wenn nur an einem einzigen Projekt mit gleichbleibenden 40 Stundenwoche gearbeitet wird ist dieser Prozess trivial. Wenn man aber als Werkstudent an mehreren Projekten mit schwankenden Stundenzahlen arbeitet wird es schnell komplizierter. Und wenn dazu noch dokumentiert werden soll woran gearbeitet wurde noch umso mehr. Dies heißt praktisch das die ‘Timecard’ täglich in Oracle aktualisiert werden muss.

¹Der Firma in welcher diese Bachelorarbeit geschrieben wurde.

²Unter diesem Namen ist das System auch firmenintern bekannt.

5.1. ANFORDERUNGSANALYSE

DXDDV003 - Soon at Endava 03 - Concurrent Engineering Regular	13.00 Hours
	Mon, Feb 06; 4.00 Hours
	Tue, Feb 07; 2.00 Hours
	Thu, Feb 09; 4.00 Hours
	Fri, Feb 10; 3.00 Hours
Comments	
Mon: Bug fixes; Tue: internal Discussion; Thu: Implement mailer; Fri: Review PRs	
INTPD999DXD - People Development DXD trainings timecards 03 - Discipline Weeks Regular	8.00 Hours
	Wed, Feb 08; 8.00 Hours
Comments	
Wed: Dev Week preparations	

Abbildung 5.2.: *Eine weniger idealtypische ‘Timecard’.*

Um eine Alternative zur Webapp zu schaffen wurde eine Kommandozeilen App entworfen, welche praktisch die gleichen Kapazitäten anbietet. Es können täglich die Arbeitszeiten sowie Notizen dazu woran gearbeitet wurde festgehalten werden. Um eine sinnvolle Alternative darzustellen sind auch die Kapazitäten zum Verwalten von Projekten und dem Übertragen der Daten zu Oracle notwendig³. Ohne Projektverwaltung würde die Zuordnung erschwert. Und ohne explizit unterstütztes Übertragen wäre die Anwendung weit weniger hilfreich.

5.1.2. Anforderungen

Die Anforderungen wurden in Form von User Stories festgehalten. Erhoben wurden diese durch den Author in Analyse der eigenen Nutzungsmuster mit Oracle, abgeglichen in Rücksprache mit anderen Mitarbeitern.

1. Projekte:

- Als Nutzer möchte ich ein Projekt hinzufügen.
- Als Nutzer möchte ich einem Projekt einen oder mehrere Task Details anfügen.
- Als Nutzer möchte ich die Projekte sowie deren Task Details aufgelistet sehen.
- Als Nutzer möchte ich Projekte und Task Details anpassen oder löschen können.

2. Arbeitszeit:

- Als Nutzer möchte ich meine Arbeitsstunden an einem Tag für eine Projekt-Task Detail Kombination loggen.
- Als Nutzer möchte ich die geloggten Arbeitsstunden einer gegebenen Woche einsehen.

³Eine mögliche, direkte Übertragung an Oracle selbst würde den Rahmen der Arbeit sprengen, ohne dem Thema dienlich zu sein. Weshalb davon abgesehen wurde.

5.2. TECHNISCHE ASPEKTE

- c) Als Nutzer möchte ich geloggte Arbeitsstunden anpassen oder löschen können.

3. Arbeitsnotizen:

- a) Als Nutzer möchte ich festhalten woran ich gearbeitet habe.
- b) Als Nutzer möchte ich meine Notizen sehen können.
- c) Als Nutzer möchte ich meine Notizen bearbeiten oder löschen können.

4. Timecard:

- a) Als Nutzer möchte ich die Daten für die Timecard zu einer gegebene Arbeitswoche sehen, um diese nach Oracle zu übertragen.

Diese User Stories sollen einen Eindruck über die Anforderungen der App und deren Komplexität vermitteln. Das Bauen der App steht aber nicht im Vordergrund, weshalb hier u.a. die Akzeptanzkriterien der User Stories, Implementationsdetails wie Datenpersistierung, etc. nicht weiter beschrieben werden.

5.2. Technische Aspekte

5.2.1. Fundament

Das Fundament der App stellen die zugrundelegenden technologischen Aspekte dar, welche nicht oder nur indirekt die Nutzbarkeit, und damit den Fokus der Arbeit betreffen.

Node.js wurde als Programmiersprache gewählt, weil der Author damit vertraut war, eine Vielzahl von Libraries zum Bauen von CLI Apps existieren und der Paketveröffentlichungsprozess mit npm relativ simpel ist. Weiterhin wurden noch Typescript - für Typisierung - und ESLint - für gleichmäßigen Code Style - verwendet.

5.2.2. Libraries

Ein Überblick über die Libraries welche für das Bauen des Interfaces und die Umsetzung der Methoden relevant waren.

oclif

“oclif is an open source framework for building a command line interface (CLI) in Node.js. Create CLIs with a few flags or advanced CLIs that have subcommands.” - *oclif: The Open CLI Framework* (2023)

Mit oclif können problemlos ‘multi-command’ App gebaut werden. Die Hierarchie der Subkommandos wird über die Dateistruktur angegeben:

```
1  src/commands
2  └─ project
3     └─ add.ts
4     └─ edit.ts
5     └─ list.ts
6     └─ remove.ts
```

5.2. TECHNISCHE ASPEKTE

Listing 5.1.: Subkommandohierarchie am Beispiel des `project` Subkommandos.

In einer dieser Dateien können nun ohne viel Boilerplate die Flaggen, Argumente und Beschreibung und spezifiziert werden. Danach wird nur noch die eigene Businesslogik angeschlossen (siehe Abbildung 5.3 für ein Implementationsbeispiel.) `oclif` übernimmt dabei das Subkommando- und Flaggenparsing, generiert die Hilfseite, uvm.

```
1 import { Command, Flags } from "@oclif/core";
2
3 import { listProjects } from "../controller/project";
4
5
6 export default class List extends Command {
7   static description = "List all projects.";
8
9   static flags = {
10    full: Flags.boolean( {
11      char: "f",
12      description: "Show the full list of task details",
13      aliases: [ "all" ],
14    } ),
15  };
16
17   async run(): Promise<void> {
18     const { flags } = await this.parse( List );
19
20     listProjects( { full: flags.full } );
21   }
22 }
```

Abbildung 5.3.: Implementation des `project list` Subkommandos.

Der Feinschliff wird noch per Konfiguration in der `package.json` erreicht. `oclif` kümmert sich dabei um die Kleinigkeiten. Zum Beispiel wird sichergestellt das der Text der Hilfsseiten richtig formatiert wird. Im unterstehenden Listing wird demonstriert wie `oclif` bei eingeschränkter Breite des Terminals den Text so einrückt, dass immer noch klar zu erkennen ist was Beschreibung und was Flagge ist. Das Gegenbeispiel dazu entsteht wenn einfach nur der Text auf den Bildschirm geschrieben wird, ohne die Terminalbreite zu berücksichtigen.

```
1 # in oclif
2 FLAGS
3 -d, --date=<value> [default: today] A date to specify the day OR
4                      the week (can be human-readable)
5
6 # ohne Framework
7 FLAGS
8 -d, --date=<value> [default: today] A date to specify the day OR
9 the week (can be human-readable)
```

Listing 5.2.: `oclif`'s Handhabung von Zeilenumbrüchen in einem schmalen Terminal im Vergleich zu einer Implementierung ohne Framework

5.3. IMPLEMENTIERTE ANWENDUNG

Neben oclif wurde noch andere Frameworks/Libraries in Erwägung gezogen. U.a. commander.js, yargs, voral und meow.

Inquirer.js

Inquirer ist eine Ansammlung von interaktiven Fragetypen (vgl. Kapitel 3.3.1). Neben einfachen Fragen, wie Text einzugeben oder einen Eintrag aus einer Liste auswählen, gibt es auch ‘multiple-choice’ (siehe Abb. 3.4) Fragen, das Bearbeiten von Text im präferierten Texteditor, uvm.

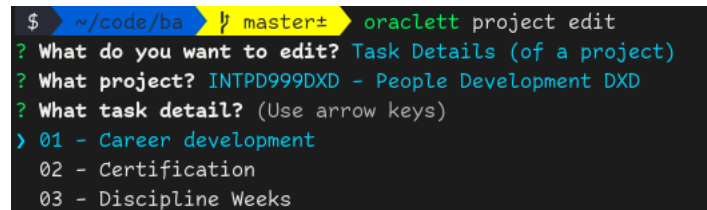


Abbildung 5.4.: Ein Reihe von Fragen zum Herausfinden was der Nutzer bearbeiten möchte.

```
1 const { whatToEdit } = await inquirer.prompt( [ {
2   type   : "list",
3   name   : "whatToEdit",
4   message: "What do you want to edit?",
5   choices: [
6     {
7       name : "Project Name",
8       value: "project",
9     },
10    {
11      name : "Task Details (of a project)",
12      value: "taskDetail",
13    },
14  ],
15 } ] );
16
17 // wird zu:
18
19 ? What do you want to edit? (Use arrow keys)
20 > Project Name
21   Task Details (of a project)
```

Listing 5.3.: Implementation für das Auswählen zwischen zwei Optionen.

5.3. Implementierte Anwendung

Es folgt eine grobe Vorstellung der implementierten App. Es geht darum ein Verständnis für den Kontext schaffen in welchem die Methoden implementiert wurden. Diese werden dann im nächsten Schritt (im nächsten Kapitel) vorgestellt.

‘oracle’ wurde als Name für die App gewählt. Der Name wurde in Anlehnung an das interne Werkzeug zum Festhalten der Arbeitszeiten benannt (bekannt als ‘Oracle’)

5.3. IMPLEMENTIERTE ANWENDUNG

und steht für ‘Oracle time tracker’. Der Name sollte 1) kurz und prägnant als auch 2) mit Autovervollständigung leicht zu vervollständigen sein. Mit **oracle** beginnend ist die Assoziation mit dem intern Tool hergestellt und kann beim drücken von TAB dann zu **oraclett** vervollständigt wird. Andere in Erwägung gezogene Varianten waren **oracle-time-tracker** oder **ott**. Diese waren aber zu lang bzw. kryptisch.

```
1 $ oraclett --help
2 Oracle time tracker
3
4 VERSION
5   oraclett/0.1.1 linux-x64 node-v19.5.0
6
7 USAGE
8   $ oraclett [COMMAND]
9
10 TOPICS
11   hour      Log and manage hours.
12   note      Write down and manage notes.
13   project   Manage projects.
14   timecard  Generate a report for filling out timecards.
15
16 COMMANDS
17   help      Display help for oraclett.
18   timecard  Generate a report for filling out timecards.
```

Listing 5.4.: *Das Hilfsmenu der CLI App.*

```
1 $ oraclett hour --help
2 Log and manage hours.
3
4 USAGE
5   $ oraclett hour COMMAND
6
7 COMMANDS
8   hour add      Log working hours.
9   hour edit     Edit the logged hours interactively.
10  hour list     List all logged hours.
11  hour remove    Remove logged hours interactively.
```

Listing 5.5.: *Hilfsseite des hour Subkommandos*

Die Subkommandos sind nach dem ‘noun verb’ Prinzip (Prasad u. a. 2022) strukturiert. Die Nomen sind dem ersten Listing 5.4 zu entnehmen: **hour**, **note**, **project** und **timecard**. Auf eines Nomen, im obenstehenden Listing 5.5 etwa die Stunde (**hour**), folgt ein Verb wie **add**, **edit**, **remove** oder **list**. Diese Verben sind über alle Kommandos hinweg einheitlich⁴. Zum Bearbeiten eines Projektes wird ebenso das Verb **edit** verwendet wie beim Bearbeiten einer Notiz.

Der vorgesehene Nutzungsflow wäre wie folgt. Wenn man einem neuen Projekt zugewiesen wird fügt man dieses **oraclett** mit **project add** hinzu.

⁴Außer bei **timecard**, welches keine Subkommandos hat.

5.3. IMPLEMENTIERTE ANWENDUNG

```
$ ~/code/ba | master± oraclett project add
? What is the project code you want to add? DXDDV003 - Soon at Endava
? Add a task detail: 02 - Planning
? Do you want to add another task detail to the project code? No
Successfully updated project
DXDDV003 (Soon at Endava)
* 02 (Planning)
```

Abbildung 5.5.: Interaktives Hinzufügen von einem Projekt.

```
$ ~/code/ba | master± oraclett project list
INTPD999DXD (People Development DXD)
* 01 (Career development)
* 02 (Certification)
* 03 (Discipline Weeks)
* .. (--full to see all)
INTET999DXD (Germany DU Bench Time)
* 01 (Bench)
```

Abbildung 5.6.: Auflisten der im System vorhandenen Projekte.

Im normalen Arbeitsalltag würden dann die gearbeiteten Stunden mit `hour add` festhalten werden.

```
1 $ oraclett hour add --help
2   Log working hours.
3
4 USAGE
5   $ oraclett hour add [-t <value>] [-p <value>] [-d <value>] [-H
6     <value>]
7
8 FLAGS
9   -H, --hours=<value>      The number of hours to log. (1h: 1, 30min: 0.5,
10                          etc.)
11  -d, --date=<value>       [default: today] The date for which to log (can be
12                          human-readable)
13  -p, --project=<value>    A project code (it it's short version)
14  -t, --taskDetail=<value> The task details (in it's short version, e.g. 01)
15
16 DESCRIPTION
17   Log working hours.
18
19   Passing no arguments will start an interactive session.
20
21   This command will add to existing hours. If run twice the hours will be
22   logged doubly.
23
24 EXAMPLES
25   $ oraclett hour add
26   $ oraclett hour add -H 3
27   $ oraclett hour add -H 3 -p INTPD999DXD -t 01
28   $ oraclett hour add -H 3 -p INTPD999DXD -t 01 --date yesterday
```

Listing 5.6.: Hilfsmenü von von `hour add`.

5.3. IMPLEMENTIERTE ANWENDUNG

```
$ ~/code/ba master± oraclett hour add -p DXDDV003 -t 02 -H 4
Successfully added hours

Hours logged for this week (Feb06 - Feb12):
Project: Task Details Mon Tue Wed Thu Fri Sat Sun Total
-----
DXDDV003: 02                                     4      4
Sums                                           4      4
$ ~/code/ba master± oraclett hour add -p DXDDV003
? How many hours? 4
? What task detail? (Use arrow keys)
> 02 - Planning
  03 - Concurrent Engineering
```

Abbildung 5.7.: Loggen von Stunden. Beim Weglassen von Flaggen werden die Werte interaktiv erfragt.

Auch Notizen, dazu woran gearbeitet wurde, können unkompliziert mit `note add` niedergeschrieben werden.

```
$ ~/code/ba master± oraclett note add -p DXDDV003 -t 02 -n "My note" -d today
Successfully added note

Notes for this week (Feb06 - Feb12):
Sat:
  DXDDV003 - Soon at Endava: 02 - Planning
    My note
$ ~/code/ba master± oraclett note add -p DXDDV003
? What task detail? 02 - Planning
? What date? today
? Note: Another note
```

Abbildung 5.8.: Hinzufügen von Notizen. Beim Weglassen von Flaggen werden die Werte auch hier interaktiv erfragt.

Zum Abschluss der Arbeitswoche würde mit `timecard` ein Report erstellt, dessen Daten dann in das interne Oracle System übertragen werden.

```
$ ~/code/ba master± oraclett timecard -I
Timecard for this week (Feb06 - Feb12):

Project Code: DXDDV003 - Soon at Endava
Task Details: 02 - Planning
Date Groups:
  Dates:      Sat
  Quantity:  8
Comments:
  Sat: My note, Another note
```

Abbildung 5.9.: Ein die ganze Woche zusammenfassender Report. Die Struktur ist darauf optimiert in die Web-Oberfläche von Oracle übertragen zu werden.

5.3. IMPLEMENTIERTE ANWENDUNG

Zum Verwalten oder Ausbessern der Daten im System gibt es jetzt noch die **edit** und **remove** Subkommandos. Diese stehen falls benötigt zur Verfügung, sind aber nicht Teil des normalen Workflows.

```
$ ~/code/ba master± oraclett note remove
Using DXDDV003 - Soon at Endava
Using 02 - Planning
? Execute deletion of this note?
Note: My note, Another note (Y/n) █
```

Abbildung 5.10.: Das Löschen einer Notiz

Die Farben für verschiedene Daten sind über alle **list** Subkommandos einheitlich gehalten. So sind die Referenzcode von Projekte und ‘Task Details’ Grün bzw. Blau⁵. Notizen sind in Gelb markiert⁶ und die Anzahl von Stunden sowie Wochentage werden mit Magenta hervorgehoben⁷.

```
$ ~/code/ba master± oraclett note list
Notes for this week (Feb06 - Feb12):
Sat:
  DXDDV003 - Soon at Endava: 02 - Planning
  My note, Another note
```

Abbildung 5.11.: Die Auflistung der Notizen einer gewählten Woche.

```
$ ~/code/ba master± oraclett hour list
Hours logged for this week (Feb06 - Feb12):
Project: Task Details      Mon Tue Wed Thu Fri Sat Sun Total
-----
DXDDV003 - Soon at Endava: 02 - Planning      8      8
Sums                      8      8
```

Abbildung 5.12.: Die Auflistung aller geloggtten Stunden für die aktuelle Woche.

Die Anwendung ist über npm⁸ als Paket verfügbar: `npm install -g oraclett`. Und funktioniert auch problemlos unter Windows.

⁵Siehe Abbildungen 5.5, 5.6, 5.8, 5.9, 5.11 und 5.12.

⁶Siehe Abbildungen 5.8, 5.9 und 5.11.

⁷Siehe Abbildungen 5.7, 5.8, 5.9, 5.11 und 5.12.

⁸Den ‘Node Package Manager’.

6. Implementation der Methoden

Die im Kapitel 4 erarbeiteten Methoden wurden in der, im vorherigen Kapitel vorgestellten, ‘Time tracking‘ Anwendung implementiert. Dieses Kapitel hebt hervor wie die Methoden implementiert werden können und welche Probleme ggf. zu beachten sind.

6.1. Flaggen anstatt Argumenten

Wie schon im Hilfsmenü von `hour add` zu sehen war (vgl. Listing 5.7) werden alle Werte als Flaggen erwartet.

Neben dem Wegfallen der Reihenfolge sind die Flaggen auch elementar für das Zusammenspiel mit anderen Methoden. Wenn der Nutzer von Defaultwerten Gebrauch machen will oder nach fehlenden Werte interaktiv gefragt werden will, kann die spezifische Flagge einfach weg gelassen werden. Mit Argumenten wäre dies so nicht möglich gewesen.

6.2. Unterstützung aller Hilfs- und Versionsflaggen

Bezüglich Methode 2 wurden alle gängigen Varianten unterstützt.

Mit `oclif` war dies sogar sehr einfach (`--help` und `--version` sind standardmäßig unterstützt):

```
1  ..
2  "oclif": {
3    "additionalHelpFlags": [
4      "-h",
5      "help"
6    ],
7    "additionalVersionFlags": [
8      "-v",
9      "-V",
10     "-version"
11     "version"
12   ],
13   ..
```

Listing 6.1.: Konfiguration von Hilfs- und Versionsflaggen für `oclif` in der `package.json`.

6.3. Relevante Standardwerte

Methode 3 wurde auf zwei Wegen implementiert. Einmal exakt wie vorgesehen und einmal als Weiterführung im Sinne der Methode.

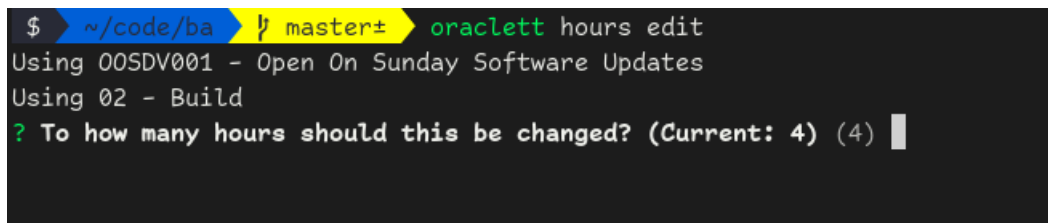
6.3. RELEVANTE STANDARDWERTE

```
1 -d, --date=<value> [default: this week] A date to specify the week
2                      (can be human-readable)
```

Listing 6.2.: Eine mit Default versehene `--date` Flagge.

Der Überstehende Auszug aus dem Hilfsmenü von `hour list` kommuniziert das die `--date` Flagge, sollte diese nicht übergeben werden, den Wert `this week` annimmt. Wird das Subkommando `hour list` ohne Flaggen aufgerufen, werden die Stunden der aktuellen Woche angezeigt. Natürlich kann auch ein anderer Zeitraum angegeben werden, aber die aktuelle Woche wird für die meisten Nutzer am relevantesten sein.

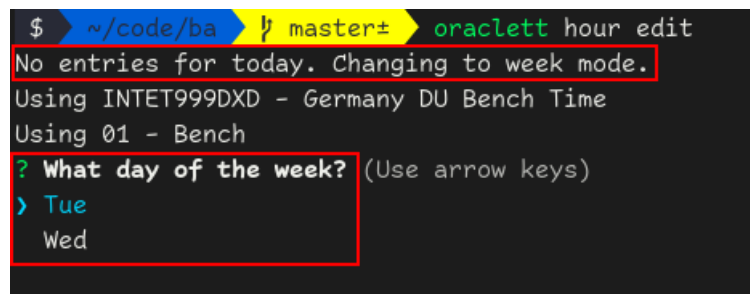
In manchen Fällen besteht nur eine Auswahlmöglichkeit. Diese wird dann automatisch ausgewählt ohne den Nutzer interaktiv danach zu fragen. Natürlich muss dieser darüber auch informiert werden. Am Beispiel von `hour edit` in der Abbildung 6.1 gestaltet sich das wie folgt: Der Standardwert für das Datum ist `today`. In diesem Falle wurden ‘heute’ nur Stunden für eine einzige Projekt-‘Task Detail’ Kombination festgehalten und diese wird deshalb direkt ausgewählt.



```
$ ~/code/ba | master± oraclett hours edit
Using OOSDV001 - Open On Sunday Software Updates
Using 02 - Build
? To how many hours should this be changed? (Current: 4) (4) |
```

Abbildung 6.1.: Wenn nur eine Option zur Auswahl steht wird diese automatisch verwendet.

Und wenn am gewählten Tag keine Stunden (oder Notizen) festgehalten wurden, wird in den Wochenmodus gewechselt. Dieser erlaubt, wie auch in Abb. 6.2 zu sehen, das Auswählen eines Wochentages für den Stunden vorhanden sind.



```
$ ~/code/ba | master± oraclett hour edit
No entries for today. Changing to week mode.
Using INTET999DXD - Germany DU Bench Time
Using 01 - Bench
? What day of the week? (Use arrow keys)
> Tue
  Wed
```

Abbildung 6.2.: Wechseln vom Tagesmodus in den Wochenmodus, welcher aus Wochentagen mit vorliegenden Daten auswählen lässt.

Außerdem wurden eine Auswahl an häufigen geteilten Projekten eingebettet. Der Nutzer hat immer noch die Möglichkeit diese zu entfernen, aber wahrscheinlich werden diese von allen Anwendern früher oder später einmal benötigt. Darunter sind u.a. Projekt Codes zum absolvieren von Trainings, internen Events und der Bank (keinem kommerziellen Projekt zugeordnet).

6.4. Interaktives Nachfragen bei fehlendem Parameter

Wie schon in Abbildung 5.7 und 5.8 zu sehen, wurde Methode 4 konsequent umgesetzt. So werden, wie auf den Abbildungen zu sehen, die fehlenden Angaben erfragt. Wie schon angesprochen ist das so aber nur im Zusammenspiel mit Methode 1 (vgl. Kap. 6.1) möglich.

Es wurde insbesondere vom Auswählen aus einer Liste von Werten Gebrauch gemacht (wie in Abb. 5.7 zu sehen), da es sehr intuitiv ist.

6.5. Natürliche Sprache

Die Methode 5 wurde primär durch das Verarbeiten von Datumsangaben in natürlicher Sprache implementiert. So kann anstatt starrer Formate wie MM/DD/YY einfach **Monday** geschrieben werden, wenn der Montag dieser Woche gemeint ist.

```
1 mon, last monday, monday two weeks ago, today, yesterday, tomorrow, aug 14
```

Listing 6.3.: *Unterstützte Varianten der Datumsangabe.*

Eingaben wie **Monday** oder **yesterday** sind keine absoluten Datumsangaben, sondern steht immer im Kontext dazu wann der Befehl ausgeführt wird. Da aber der Fokus der Anwendung auf der aktuellen bzw. der vergangenen Woche liegt besteht darin kein Problem. In der Evaluation zeigten sich die Nutzer zufrieden damit auf diesem Wege Daten zu beschreiben. Absolute Datumsangaben werden aber natürlich auch unterstützt. Umgesetzt wurde das Ganze mit der Library `sugar.js` (keine Empfehlung¹).

Auch sollten Synonyme für Subkommandos “erkannt” werden. Angedacht war das Einrichten von alternativen Namen bzw. Aliasen. So sollten Subkommandos wie **hour** auch als **hours** erkannt werden. Oder **remove** als **delete**, etc. Leider wurde dies von `oclif` nur inkonsequent und verwirrend unterstützt. So können Flaggen ganz einfach mit einem Alias versehen werden. Auch Kommandos haben diese Option, dabei würde aber folgendes heraus kommen:

```
1 COMMANDS
2 hour add      Log working hours.
3 hour edit     Edit the logged hours interactively.
4 hour list     List all logged hours.
5 hour log      Log working hours.
6 hour remove   Remove logged hours interactively.
```

Listing 6.4.: *Auszug aus dem Hilfsmenü von hour. Subkommandos add und log teilen die gleiche Beschreibung.*

Kommandos die als Alias markiert sind tauchen immer in den Hilfsseiten auf. Dadurch wird die Liste weniger übersichtlich und hilfreich. Der Nutzer weiß nicht was der Unterschied ist. Es gibt aber von `oclif` aus keinen Weg diese zu verstecken. Weshalb sich gegen diese potenziell frustrationssparenden Aliase entschieden wurde.

¹Keine Updates seit 4 Jahren. Auch sind im späteren Teil des Bauens ein paar ‘Edgecase’ Fehler aufgefallen.

6.6. Fehlerkorrektur

```
$ ~/code/ba ⚡ master± oraclett hours list
> Warning: hours list is not a oraclett command.
Did you mean hour list? [y/n]:
```

Abbildung 6.3.: Beispiel der ‘Meintest du ..’ Fehlerkorrektur.

Um dem Nutzer bei Tippfehler trotzdem einen schnellen Weg zum Ausführen des richtigen Befehls zu geben, wurde im Bezug auf Methode 6 eine suggestive Fehlerkorrektur eingebaut (siehe Abb. 6.3.) Diese kommt mitgeliefert als Plugin für oclif.

Synonyme wie die im letzten Kapitel (6.5) besprochenen `hour` ↔ `hours` die sich sehr ähnlich sind können durch auch die Fehlerkorrektur verbessert werden (siehe Abb. 6.3.) Weiter entfernte Synonyme wie `remove` ↔ `delete` funktionieren leider nicht. Da dem System wahrscheinlich auch keine Synonymdatenbank unterliegt.

6.7. Autovervollständigung

Die Methode 7 und die damit verbundene Autovervollständigung sind technisch gesehen implementiert, praktisch aber nicht nutzbar:

```
$ ~/code/ba ⚡ master± oraclett
autocomplete -- display autocomplete installation instructions
help -- Display help for <%= config.bin %>.
hour:list -- List all logged hours.
note:list -- List all notes.
note:remove hour:remove note:edit hour:edit -- If a day is specified, you will edit that days hours.
```

Abbildung 6.4.: Problematisches Autovervollständigungsmenü der App

Wie auf der Abbildung 6.4 zu sehen ist existieren einige Probleme. Am offensichtlichsten sind die Doppelpunkte (‘:’) zum Trennen der Subkommandos anstatt von Leerzeichen. Das ist eine von oclif vorgegebene Konvention und obwohl der Rest der Anwendung Leerzeichen nutzt (die auch viel weiter verbreitet sind), sind die Doppelpunkte an dieser Stelle nicht konfigurierbar. In der letzten Zeile wird als Beschreibung etwas von “If a day is specified [..]” geschrieben. Es wurde unerklärlicherweise nicht die Zusammenfassung verwendet sondern ein Schnipsel aus dem Hilfsmenü. Welcher an dieser Stelle nur verwirrend für den Nutzer ist. Und zuletzt wird in der Beschreibung der ersten Zeile der Beginn des Satzes nicht großgeschrieben und in der zweiten Zeile der Template String nicht evaluiert. Alle dieser Dinge sind nicht konfigurierbar. Für eine ordentliche Vervollständigung müsste diese also selbst gebaut werden. oclif erschwert dies aber für externe Libraries wie omelette. Und die Completion Dateien selbst in Shell zu schreiben, ohne den Zugriff auf die Quellcode Dateien welche die Struktur und Inhalte vorgeben, wäre sehr fehleranfällig.

Die geringe Qualität dieser Autovervollständigung war beim initialen Testen des Frameworks und dem Lesen der Dokumentation noch nicht ersichtlich. Und nach der Implementation war es zu spät zu einem anderen Framework zu wechseln², bei welchem mög-

²Mögliche Alternativen wurden im Kapitel 5.2.2 aufgeführt.

licherweise eine bessere Autocompletion möglich gewesen wäre.

Um Verwirrung bei den Endnutzern zu vermeiden wurde die Funktionalität wieder aus der App entfernt.

6.8. Menü-basiertes Interface

Bei Methode 8 wurde sich aufgrund des zeitlichen Rahmens gegen eine Implementierung entschieden. Das Interface wäre von dem primären Kommandozeilen-Interface gänzlich abgetrennt. Die Implementation und das gewährleisten einer einheitlichen Nutzungserfahrung die über beide Interfaces hinweg würde einen, im Vergleich zum Rest der App, unverhältnismäßigen Aufwand mit sich bringen. Die Anforderungen hätten aber durchaus als Menü abgebildet werden können.

6.9. Relevante Kommandovorschläge

Methode 9 hat in der App keine Anwendung gefunden. Wie auch in der Erklärung wiedergegeben spricht Dutta (o.D.) diese Empfehlung in Hinsicht auf CLI Apps mit sehr vielen (Sub-) Kommandos aus. Die implementierte Anwendung umfasst nur eine sehr limitierte Anzahl von Subkommandos. Aufgrund dieser bereits gegebenen Übersichtlichkeit wurde sich gegen die direkten Vorschläge entschieden.

In den Fehlermeldungen wurde der Geiste der Methode aber angewandt. So verweist ein fehlendes Projekt etwa auf das Hinzufügen und Auflisten von Projekten hin (siehe nachstehende Abbildung.)

```
$ ~/code/ba | master± oraclett hour add -H3 -p "not-a-project"
Error: The project does not exist.

To check existing: project list
To add a new one : project add
```

Abbildung 6.5.: *Vorschläge zum Hinzufügen und Auflisten von Projekten bei Angabe eines invalides Projektnamens.*

7. Evaluation

Im Rahmen der Studie beziehen sich GUI oder GUI App immer auf die Oracle Webapp. Gleichmaßen steht es um CLI oder CLI App, mit welchem immer die im Rahmen der Arbeit gebaute `oraclett` Anwendung gemeint.

7.1. Design der Studie

Es wurde ein Usability-Test durchgeführt. Laut Hegner 2003, S. 36 werden dabei “[...] konkrete Nutzungssituationen durch repräsentative Endnutzer simuliert, um die Bedienbarkeit eines Produkts oder Prototypen zu überprüfen.” Es wurde die implementierte CLI App der existierenden GUI Webapp gegenüber gestellt. Die Teilnehmer haben in beiden Apps Aufgabenstellungen erfüllt und dann die Attraktivität der Applikation bewertet. Das Ziel war es zu sehen welchen Effekt die entwickelten und implementierten Methoden auf die Nutzbarkeit des CLI hatten. Ob die, der Kommandozeile inhärente, Nutzungsschwelle abgesenkt werden konnte. Die existierende GUI Webapp diente dazu als Maßstab gegen welchen es die Usability und Performance des CLI zu vergleichen galt.

7.1.1. Rahmenbedingungen

Die Teilnehmer der Studie waren alle Mitarbeiter der Firma Endava. Dies war eine Voraussetzung um die Zugriffsrechte für die Nutzung der internen Oracle Platform, mit welcher verglichen werden sollte, zu gewährleisten. Die Teilnehmer waren mehr oder weniger mit dem zum Vergleich stehenden GUI System vertraut¹. Und hatten dieses schon für die zu testenden Tätigkeiten verwendet. Das Verständnis der Anwendungsdomäne der Apps war also gegeben. Alle Teilnehmer waren Software Entwickler oder Entwicklungsnah. Expertise mit der Kommandozeile war zwischen “wird nur benutzt wenn es nicht anders geht” und “wird regelmäßig verwendet”. Insgesamt wurden 6 Personen getestet. Laut Spolsky (2001, S. 11) ist “[...] five or six users [...] all you need.” Durchgeführt wurde der Test via Videoanruf mit geteiltem Bildschirm. Die Teilnehmer arbeiteten das erste Mal mit der CLI App. Um Erlernbarkeit zu testen und trotzdem vergleichbare Performancemetriken zu erhalten wurde den Teilnehmern eine Einführung in die Anwendung gegeben. In welcher alle Funktionalität gezeigt und beschrieben wurde. Inklusive einem ‘Refresher’ zum Auffinden von Hilfe und Shell Basics wie `CTRL+c`² und Anführungszeichen um gruppieren von Text³. Alles um zu gewährleisten das die Teilnehmer, ohne das Eingreifen des Testleiters, die gegebenen Aufgaben lösen können.

¹Von mehreren Jahren bei Endava bis zu wenigen Wochen.

²Unterbricht die interaktive Eingabe.

³Text mit Leerzeichen muss in der Shell gruppiert werden: `-d "next monday"` ↔ `-d next monday`. Der zweite Aufruf führt zu einem Fehler, weil die App denkt `monday` wäre als Argument übergeben.

7.1. DESIGN DER STUDIE

7.1.2. Aufgabenstellungen

Es sollten die gleichen Aufgaben in beiden System erledigt werden. Um die beiden Apps vergleichbar zu machen, wurde nur die Schnittmenge der Funktionalitäten beim Erstellen der Aufgaben berücksichtigt. Die zu testende Funktionalität wurde dementsprechend auf das Verwalten von Stunden und Notizen beschränkt. In Hinsicht auf die Erlernbarkeit wurde auf Anraten von Prof. Israel eine Menge von 20 Aufgaben gewählt.

Die Studie wurde auf Englisch durchgeführt. Um die sprachlichen Feinheiten nicht zu vertuschen, wurden die Aufgaben hier im originalen Wortlaut wiedergegeben. Die Teilnehmer hatten zu jeder Aufgabe auch noch Informationen dazu, auf welche Projekte-‘Task-Details’ Kombination die Aufgabe Bezug nimmt.

1. Add 8 hours for today on the Bench.
2. Describe that you worked on “Self-Learning” during that time today.
3. Add another 8 hours for tomorrow on the Bench.
4. Change the 8 hours for tomorrow to 6.
5. Remove the 6 hours for tomorrow.
6. Log 2 hours on the Bench for Friday this week.
7. Add 6 hours on “People Development - Certification” for Friday this week.
8. Describe that you did a “Node.js Certificate” in the hours you just logged.
9. Change today's 8 hours on the bench to 2.
10. And delete your note (“Self-Learning”) for today.
11. Add 4 hours of Dev Discipline for today.
12. Add 4 hours of Testing Discipline for today.
13. Add 3 hours of both Testing Discipline and Dev Discipline for Monday of this week.
14. Log 6 hours on the bench for Monday, Tuesday and Wednesday of next week.
15. Add a note for Monday of next week that you “Played around with Golang”.
16. Change the just added note to “Played around with Rust”.
17. Change the hours for next weeks Tuesday and Wednesday to 8.
18. Delete the hours logged for next weeks Wednesday.
19. Show and tell Jonathan how many hours you worked in total for this week.
20. Show and tell Jonathan how many hours you logged for the Bench next week.

Die Häufigkeit der einzelnen Subkommandos basiert darauf wie häufig diese circa in realer Nutzung auch auftreten würden.

Die Auflistung nach Häufigkeit:

7.1. DESIGN DER STUDIE

- 8x: hour add
- 3x: note add
- 3x: hour edit
- 2x: hour remove
- 2x: hour list
- 1x: note edit
- 1x: note remove

Um das Spielfeld etwas zu ebnen, wurde nicht immer der gleiche Wortlaut für Verben wie ‘remove’ verwendet. Sondern auch das synonyme ‘delete’. Die CLI App verwendet aber das Verb **remove** als Subkommando. Und dies führte dazu das Teilnehmer u.a. versuchten **hour delete** zu nutzen. Andere Beispiele für diese Synonyme sind ‘describe’ ↔ ‘add’, ‘change’ ↔ ‘edit’ und ‘log’ ↔ ‘add’. Die Sätze meinen immer noch dasselbe, fordern die Teilnehmer bei der Verwendung des CLI aber etwas mehr heraus.

Aufgrund der unterschiedlichen Einschränkungen der beiden User Interfaces waren manche Aufgaben mit dem einen oder anderen theoretisch einfacher zu erledigen. In der Auswertung wird auf diese Effekte bei spezifischen Aufgaben genauer eingegangen.

7.1.3. Zu erhebende Daten

Performance

Um die Effizienz zu messen wurde festgehalten wie lange ein Benutzer benötigt um eine gegebene Aufgabe zu erfüllen. Dazu wurden die Sitzungen aufgezeichnet und danach mit Zeitmarken versehen. Der Zeitstempel zum Start wurde nach dem Lesen der Aufgabe (bei erster Interaktion mit dem Interface) gesetzt und der Zeitstempel zum Ende nach Erfüllung der Aufgabe.

Attraktivität

Um die subjektive Befriedigung der Nutzer zu messen wurde eine Attraktivitätsumfrage nach *AttrakDiff* (2023) durchgeführt. Es wurde nach dem Durcharbeiten der Aufgaben für ein Interface eine Einschätzung über dessen Attraktivität eingeholt. Wie auf Abbildung 7.1 zu sehen bewerteten die Teilnehmer dabei das Interface bezüglich mehrerer entgegengesetzter Adjektivpaare.

einfach*	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	kompliziert
hässlich*	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	schön
praktisch*	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	unpraktisch
stilvoll*	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	stillos
voraussagbar*	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	unberechenbar
minderwertig*	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	wertvoll
phantasielos*	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	kreativ
gut*	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	schlecht
verwirrend*	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	übersichtlich
lahm*	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	fesselnd

Abbildung 7.1.: Beurteilungsbogen der kurzen *AttrakDiff* Umfrage.

7.2. HYPOTHESEN

7.1.4. Durchführung

Nach einer generellen Einführung und Installation der App auf der Maschine des Teilnehmers begann der Teilnehmer entweder mit der GUI Webapp oder der CLI Applikation. 50% der Teilnehmer begannen mit der einen, die andere Hälfte mit der anderen. Vor der Verwendung der CLI App gab es die erläuterte Einführung. Mit dem Hinweis darauf, dass der Versuchsleiter keine Hilfestellung leisten würde, ging es mit der Bearbeitung der Aufgaben los. Die Checkliste der Aufgaben waren eine nach der anderen zu bewältigen. Nach Erledigung der Aufgaben für ein Interface folgte die Attraktivitätsumfrage, gefolgt von der Bearbeitung derselben Aufgaben für das andere Interface, sowie erneuter Umfrage.

7.2. Hypothesen

Mit den erhobenen Daten sollten folgende Hypothesen adressiert werden:

Hypothese 1: Das CLI hat im Durchschnitt eine ähnliche Performance wie das GUI.

Die Probleme der Kommandozeile werden dazu führen das Effizienz eingebüßt wird. Diese Probleme zu mitigieren und minimieren war aber Zweck der Methoden. Weshalb unter Anwendung der Methoden zumindest eine ähnliche Performance wie die einer grafischen Anwendung erwartet wird.

Nun waren die Nutzer schon mehr mit der GUI App vertraut. Gleiche Performance bei ungleichen Startbedingungen würde sogar auf einen Performanceüberschuss bei gleichen Startbedingungen hindeuten (i.e. wenn die Teilnehmer mit beiden Anwendungen gleich vertraut sind.)

Hypothese 2: Die Performance des CLI verbessert sich im Verlaufe der Aufgaben.

Von einer erlernbaren Anwendung wäre zu erwarten das sich deren Performance steigert wenn man den Teilnehmer genug Zeit gibt diese zu erlernen.

Zu betrachten ist die Performance ähnlicher Aufgabestellungen.

Hypothese 3: Das CLI wird dem GUI subjektiv vorgezogen.

Bei Westerman (1997) wurde das CLI bei freier Wahl des Interface deutlich weniger gewählt. Die Bevorzugung des CLI gegenüber dem GUI unter den Teilnehmern würde also den Effekt der Methoden auf die Usability aufzeigen.

7.3. Auswertung

7.3.1. Diskussion der Ergebnisse

Über alle Diagramme hinweg ist Blau auf die Farbe der CLI App bzw. Orange die der GUI App.

7.3. AUSWERTUNG

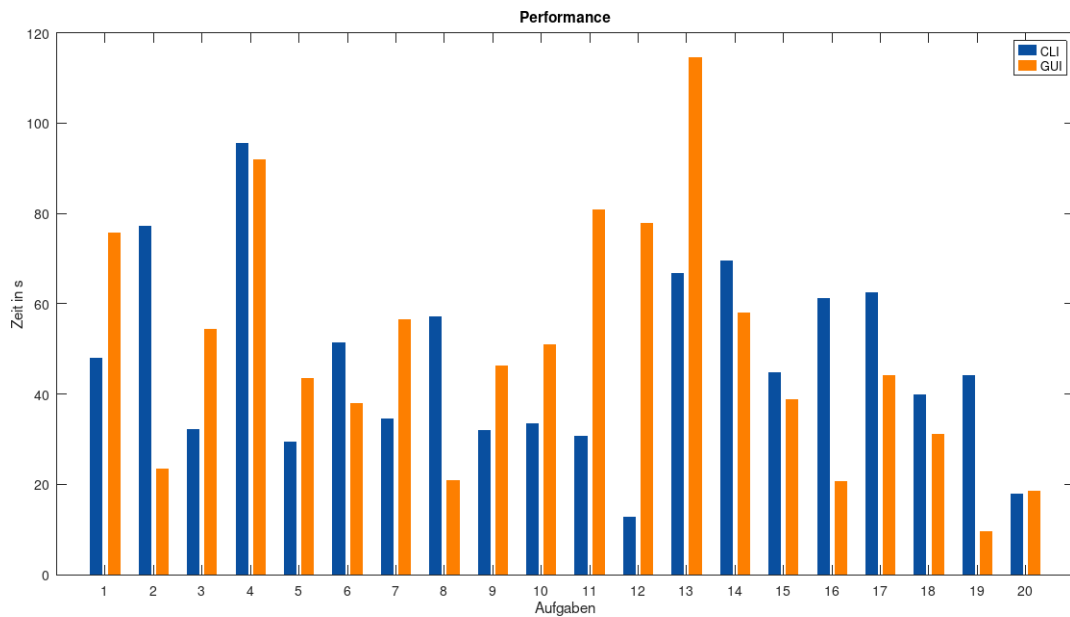


Abbildung 7.2.: Performance beider Interfaces bei der Erfüllung gleicher Aufgaben.

Die Performance der verschiedenen Aufgaben schwanken stark zwischen den Aufgaben bei gleichem Interface (vgl. Abbildung 7.2.) Das begründet sich darin das die Aufgaben innerhalb der gleichen Anwendung unterschiedlich schwer zu lösen waren.

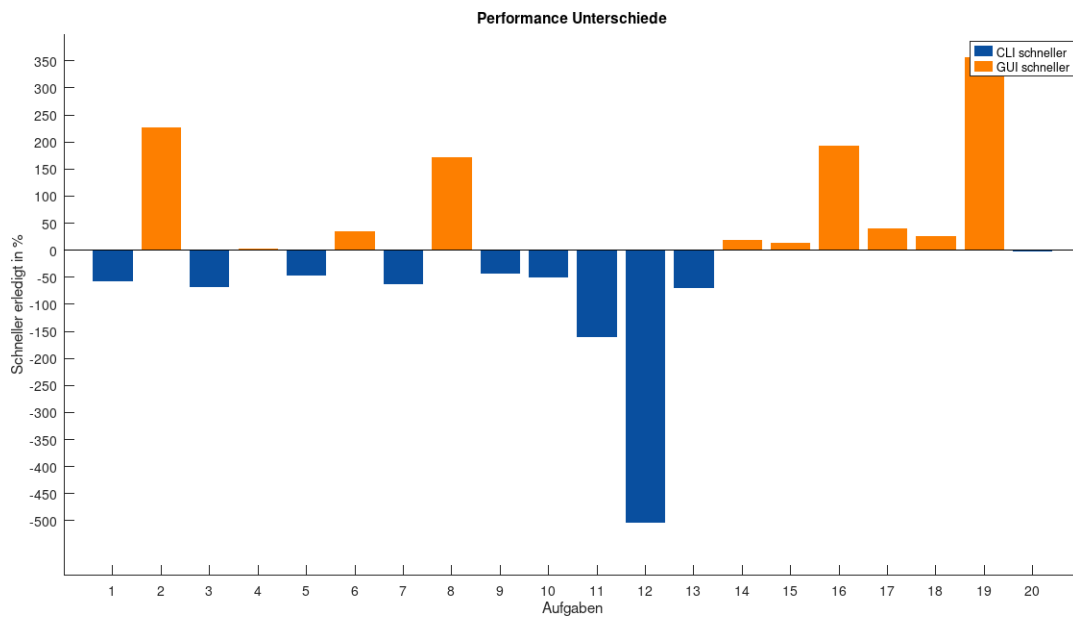


Abbildung 7.3.: Performance Unterschiede zwischen den Anwendungen.

Auch die Performance Differenz zwischen den beiden Anwendungen schwankt (vgl. Abbildung 7.3.) Was seine Gründe darin hat das die Aufgabe durch ein Interface effizienter erledigt werden konnte. Für das Kommandozeilen Interface spielt noch mit hinein das bei neuen Aufgabenstellungen (solche die ein neues Subkommando oder eine neue Flagge erfordern) dem Nutzer nicht klar ist, wie vorzugehen ist. Und dort Zeit für das

7.3. AUSWERTUNG

Lesen von Hilfsmenüs aufgewendet werden muss. Was aber in der Natur des CLI liegt.

Das Nachschauen im Hilfsmenü ist einer der Faktoren weshalb das CLI beispielsweise bei der zweiten Aufgabe die dreifache Zeit zur Erfüllung brauchte. Es stellte die erste Verwendung von `note add` dar. Hinzu kommt, dass das Hinzufügen von Notizen im GUI relativ unkompliziert ist. Verglichen mit der zweiten Verwendung in Aufgabe 8 bleibt die Performance des GUI fast gleich während das CLI zulegt (etwa 30%⁴ schneller als zuvor.)

Ebenso sind auch die Differenzen der Aufgaben 19 und 20 zu erklären. Das erste bzw. zweite Mal das `hour list` verwendet wurde. Von 355% schnellerer GUI Performance zum gleichauf sein in der nächsten Aufgabe.

Die extremste Differenz ist bei Aufgabe 12 entstanden. Bei den Aufgaben 11 und 12 werden vier Stunden für dasselbe Projekt auf verschiedenen ‘Task Details’ festgehalten. Im GUI bleibt der Aufwand fast gleich während er sich für das CLI halbiert und im Vergleich zum GUI zu 500% schneller Performance führt. Bei optimaler Verwendung kann im Kommandozeilen-Interface der gleiche Befehl für beide Aufgaben wiederverwendet werden⁵.

Aufgabe 16 stellt den letzten der Extremfälle dar und ist das einzige Auftritt des `edit` Subkommandos⁶.

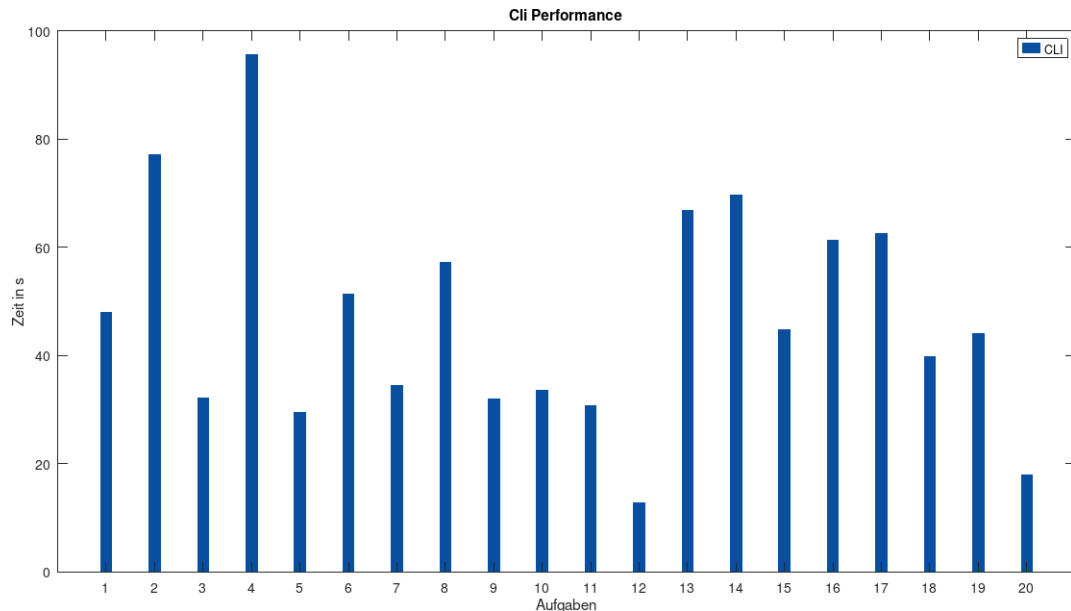


Abbildung 7.4.: Isolierte Performance der CLI App.

⁴Im Vergleich zur GUI Performance. Von 225% schneller Erledigung zu 160%.

⁵Der optimale Befehl ist: `hour add -H 4 -d today -p INTDS999DXD` und es wird nur das richtige Task Detail aus der Liste ausgewählt. Alternativ wäre auch das Spezifizieren von `-t 08` bzw. `-t 12` ähnlich schnell.

⁶Wie in Kap. 5.3 erläutert hat `orcalett` eine ‘noun verb’ Struktur. Sowohl die Substantive `project`, `hour` und `note` bieten jeweils ein `edit` Subkommando an.

7.3. AUSWERTUNG

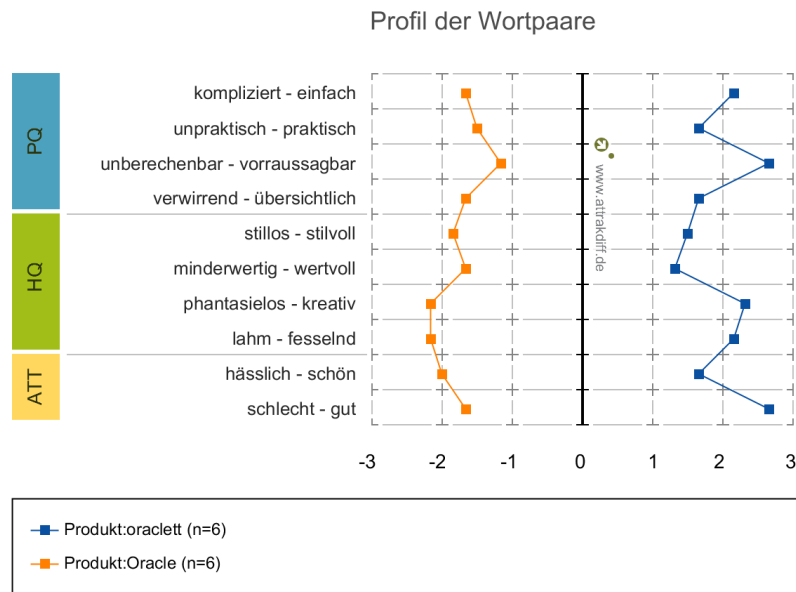


Abbildung 7.5.: Ergebnisse der Attraktivitätsumfrage darüber wo die Anwendungen für die einzelnen Adjektivepaare landeten.

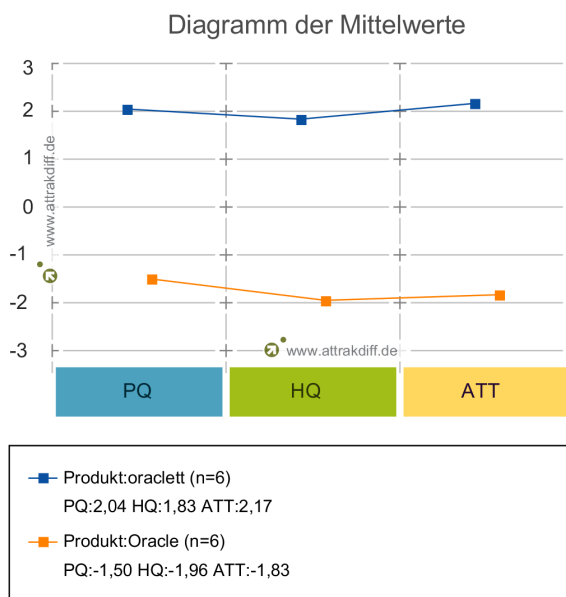


Abbildung 7.6.: Mittelwerte über die von AttrakDiff (2023) definierten Kategorien.

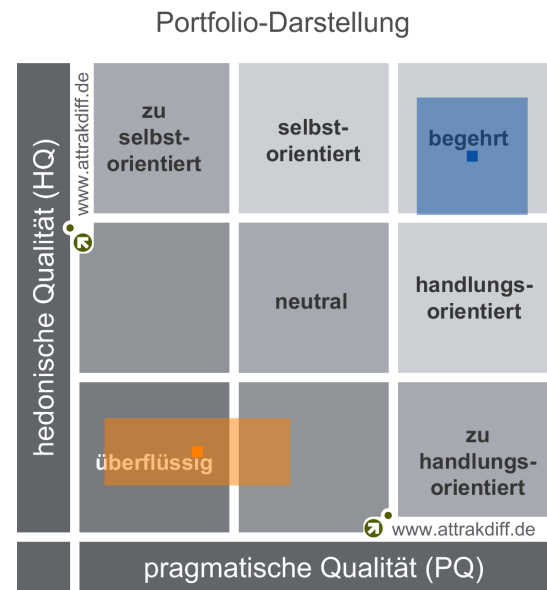


Abbildung 7.7.: Einordnung der Ergebnisse in die Matrix nach hedonischer und pragmatischer Qualität.

Wie in Abbildung 7.6 zu sehen ist, sind die Anwendungen in allen Kategorien über 3.5 Punkte auseinander. Auf einer Skala die nur 7 Punkte breit ist, ist das eine Menge.

In Abbildung 7.7 "spiegelt das Konfidenz-Rechteck [..] wieder, wie "einig" sich die Nutzer bei der Beurteilung des Produkts sind. Je größer das Konfidenz-Rechteck ist, desto

7.3. AUSWERTUNG

unterschiedlicher wird das Produkt bewertet.” - *AttrakDiff* (2023) via esurvey.uid.com.

7.3.2. Auswertung der Hypothesen

Hypothese 1

Hypothese 1: Das CLI hat im Durchschnitt eine ähnliche Performance wie das GUI.

Der Performance Durchschnitt über alle Aufgaben aufsummiert liegt bei $-2.76s$, das CLI ist also um 3 Sekunden schneller. Wie in Abbildung 7.3 zu sehen ähnelt sich die Performance beider Anwendungen. Keine der beiden ist konsistent schneller als die andere. Es wurden nur bei sechs Aufgaben mehr als 100% Performance Differenz festgestellt, also das ein Interface mehr als doppelt so schnell als das andere. Auf diese Extremfälle wurde auch bereits einleitend eingegangen.

Hypothese 2

Hypothese 2: Die Performance des CLI verbessert sich im Verlaufe der Aufgaben.

Bei Betrachtung der Performance des CLI (Abb. 7.4) hinsichtlich der Erlernbarkeit sind die Aufgabenstellungen zu beachten. So werden die Aufgaben mit der Zeit schwieriger: Es werden Projekte mit mehr als einem Task Detail verwendet (ab 7.), das Datum wandert zu anderen Wochentagen (ab 6.) und in die nächste Woche (ab 14.). Es werden zuvor einzelne Aufgaben gebündelt (ab 13.), etwa das hinzufügen mehrerer Stunden an verschiedenen Tagen als Teil einer Aufgabe. Auch werden kontinuierlich neue Subkommando Kombinationen gefordert⁷.

Trotz all dieser erschwerenden Faktoren ist ein fallender Trend zu erkennen.

Hypothese 3

Hypothese 3: Das CLI wird dem GUI subjektiv vorgezogen.

Der subjektive Vergleich ist ziemlich eindeutig. Die Abbildung 7.7 beschreibt die CLI App als begehrt und Oracle als überflüssig. Auch auf Abbildung 7.5 liegt das CLI in allen Punkten signifikant vorne.

7.3.3. Beobachtungen aus der Evaluation

In der Nutzung beider Interfaces traten bei manchen Teilnehmern Momente auf in denen sie nicht wussten wie vorzugehen war. In der CLI App sind die Teilnehmer als Reaktion darauf zur Hilfsseite zurückgekehrt und haben versucht sich neu zu orientieren und den Fehler in ihrem letzten Befehl zu finden. Dabei wurde sich nicht an den Versuchsleiter gewendet. Den Teilnehmern war bewusst, dass sie dieses Problem allein lösen können. Bei Aussetzern von Oracle (aus unerfindlichen Gründen) herrschte immer Verwirrung. Das Interface teilte nichts mit und tat einfach seinen Dienst nicht. An dieser Stelle wurde immer die Hilfe des Versuchsleiters gesucht.

⁷Siehe die Auflistung der Anzahl von Aufgaben pro Subkommando in Kapitel 7.1.2.

8. Zusammenfassung

Nach Analyse der grundlegenden Probleme der Kommandozeile, wurden neun Methoden zur Verbesserung der Usability für Kommandozeile-Interfaces erarbeitet. Diese wurden in einer CLI App implementiert und mit einer grafischen Anwendung mit gleichem Funktionsumfang verglichen. Die Teilnehmer bevorzugten das CLI über das GUI, bei gleichauf liegender Performance.

8.1. Schlussfolgerungen

Es wurde gezeigt, dass in der Kommandozeile, unter Anwendung der erarbeiteten Methoden, Usability erreicht werden kann die es mit grafischen Anwendungen aufnimmt.

8.2. Limitationen

Es war im Rahmen der Arbeit nicht möglich den Effekt einzelner Methoden isoliert festzustellen. Es konnte also nur eine Aussage über alle implementierten Methoden zusammen getroffen werden.

8.3. Ausblick

8.3.1. Entwicklung und Strukturierung von Methoden

Die Methoden könnten in stärker strukturierter Form und außerhalb des Kontextes dieser Arbeit für sich stehend formuliert werden. Zusammen mit einer englischen Übersetzung wären sie dadurch dann leichter referenzier- und implementierbar. Auch könnten noch weitere Methoden entwickelt werden.

8.3.2. Vergleich mit CLI App

Anstatt des Vergleiches mit einer GUI App, würde der Vergleich mit einer baugleichen CLI App, welche die Methoden nicht implementiert, eine stärkere Aussagekraft bezüglich des Effektes der Methoden haben. Auch wäre es damit möglich nur eine Methode auf einmal zu testen. Mit identischen Apps A und B, aber B hat zusätzlich die zu untersuchende Methode implementiert.

8.3.3. Verbesserung der Implementation

In der Evaluation wurden noch einige Schwächen der Implementation entdeckt, welche noch auszubessern sind. Auch die Versäumnissen etwa wie bei der Autovervollständigung (vgl. Kap. 6.7), dem Menü Interface (vgl. Kap. 6.8) oder die fehlenden Synonyme (vgl. Kap. 6.5) bieten noch Verbesserungsbedarf.

8.3. AUSBLICK

Eine Anbindung an Oracle selbst war im Rahmen dieser Arbeit nicht vorgesehen, weil der Fokus auf den Methoden lag. Als zukünftige Verbesserung der App wäre dies aufgrund der Zeitersparnis aber sinnvoll.

8.3.4. Bearbeiten der Problemstellung in anderer Herangehensweise

Die zu Beginn gestellte, offene Problemstellung war: Wie Entwicklern oder der Allgemeinheit das CLI näher gebracht werden kann. Neben dem Verbessern der Nutzbarkeit von CLI Applikationen, sind noch andere Herangehensweisen möglich. Etwa das Terminal-Ökosystem zugänglicher und verständlicher zu machen. Oder die Zugänglichkeit der Werkzeuge für nicht technisch-fokussierte Personen verbessern. Vieles ist möglich.

Quellenverzeichnis

- AttrakDiff* (2023). User Interface Design GmbH. URL: <https://www.attrakdiff.de> (besucht am 08.02.2023).
- Balzert, Helmut (2009). *Lehrbuch der Softwaretechnik: Basiskonzepte und Requirements Engineering*. Spektrum Akademischer Verlag. DOI: 10.1007/978-3-8274-2247-7. URL: <https://doi.org/10.1007%2F978-3-8274-2247-7>.
- Bland, Wesley u. a. (2007). „Design and implementation of a menu based oscar command line interface“. In: *21st International Symposium on High Performance Computing Systems and Applications (HPCS'07)*. IEEE, S. 25–25.
- Chen, Jung-Wei und Jiajie Zhang (Okt. 2007). „Comparing Text-based and Graphic User Interfaces for novice and expert users.“ In: *AMIA ... Annual Symposium proceedings. AMIA Symposium vol. 2007*, S. 125–129.
- Dickey, Jeff (2018). *12 Factor CLI Apps*. Heroku. URL: <https://medium.com/@jdxcode/12-factor-cli-apps-dd3c227a0e46> (besucht am 28.12.2022).
- Dutta, Samrat (o. D.). „Usability improvements for products that mandate use of command-line interface: Best Practices“. In: ().
- Gentner, Don und Jakob Nielsen (Aug. 1996). „The Anti-Mac interface“. In: *Communications of the ACM* 39.8, S. 70–82. DOI: 10.1145/232014.232032. URL: <https://doi.org/10.1145%2F232014.232032>.
- GNU core utilities* (2023). GNU. URL: <https://www.gnu.org/software/coreutils/> (besucht am 31.01.2023).
- Hegner, Marcus (2003). „Methoden zur Evaluation von Software“. In:
- Kantorowitz, Eliezer und Oded Sudarsky (Nov. 1989). „The adaptable user interface“. In: *Communications of the ACM* 32.11, S. 1352–1358. DOI: 10.1145/68814.68820. URL: <https://doi.org/10.1145%2F68814.68820>.
- Nagarajan, Vivek (2018). *Command Line Fundamentals*. Packt Publishing. ISBN: 9781-789807769.
- Nielsen, Jakob (1993a). *Noncommand User Interfaces*. Nielsen Norman Group. URL: <https://www.nngroup.com/articles/noncommand/> (besucht am 05.12.2022).
- (1993b). *Usability Engineering*. 1st. Morgan Kaufmann. ISBN: 0125184069; 9780125184069.
- Norman, Donald A. (1983). „Design principles for human-computer interfaces“. In: *Proceedings of the SIGCHI conference on Human Factors in Computing Systems - CHI 83*. ACM Press. DOI: 10.1145/800045.801571. URL: <https://doi.org/10.1145%2F800045.801571>.
- (Mai 2007). „The Next UI Breakthrough: Command Lines“. In: *Interactions* 14.3, S. 44–45. DOI: 10.1145/1242421.1242449. URL: <https://doi.org/10.1145%2F1242421.1242449>.
- oclif: The Open CLI Framework* (2023). Salesforce. URL: <https://oclif.io> (besucht am 02.01.2023).
- Paap, Kenneth R. und Renate J. Roske-Hofstrand (1988). „Design of Menus“. In: *Handbook of Human-Computer Interaction*. Elsevier, S. 205–235. DOI: 10.1016/b978-0-

- 444-70536-5.50015-4. URL: <https://doi.org/10.1016%2Fb978-0-444-70536-5.50015-4>.
- Prasad, Aanand u. a. (2022). *Command line interface guidelines*. URL: <https://clig.dev> (besucht am 23. 12. 2022).
- Raskin, Aza (Jan. 2008). „The linguistic command line“. In: *Interactions* 15.1, S. 19–22. DOI: 10.1145/1330526.1330535. URL: <https://doi.org/10.1145%2F1330526.1330535>.
- Raymond, Eric S. (2023). *DWIM*. URL: <http://www.catb.org/~esr/jargon/html/D/DWIM.html> (besucht am 02.02.2023).
- Schröder, Michael und Jürgen Cito (Dez. 2021). „An empirical investigation of command-line customization“. In: *Empirical Software Engineering* 27.2. DOI: 10.1007/s10664-021-10036-y. URL: <https://doi.org/10.1007%2Fs10664-021-10036-y>.
- Seneviratne, Nilanthi (2008). *New command line interfaces*.
- Spolsky, Joel (2001). *User Interface Design for Programmers*. 1. Aufl. Apress. ISBN: 9781893115941.
- Westerman, S. J. (Juni 1997). „Individual Differences in the Use of Command Line and Menu Computer Interfaces“. In: *International Journal of Human-Computer Interaction* 9.2, S. 183–198. DOI: 10.1207/s15327590ijhc0902_6. URL: https://doi.org/10.1207%2Fs15327590ijhc0902_6.

9. Glossar

CLI: ‘Command-line Interface’, Kommandozeilen-Interface. Siehe Kapitel 2.1.

Flagge: Siehe Kapitel 2.4.

GUI: ‘Grafical User Interface’, grafisches Interface. Siehe Kapitel 3.3.2.

Interaktiv: Siehe Kapitel 3.3.1.

Kommando: Siehe Kapitel 2.4.

Menü: Siehe Kapitel 3.3.1.

Oracle: Im Kontext der Arbeit ist die interne Plattform zum Festhalten von Arbeitszeiten gemeint.

Task Details: Zum Loggen von Zeit in Oracle muss ein Projekt und eine Tätigkeit innerhalb des Projektes (die ‘Task Details’) angegeben werden.

Unix: Ein frühes Betriebssystem, welches u.a. Standard für POSIX definierte. Mehr in Kap. 2.3.

Usability: Siehe Kapitel 2.5.

A. Appendix

A.1. Quell-Code

Der Quelle-Code für die Beispielanwendung ist auf GitHub zu finden:

<https://github.com/jneidel/oraclett>

A.2. Rohdaten

Nachfolgend die in der Evaluation erhobenen Performancedaten. Diese wurden mit Octave verarbeitet und dargestellt.

```
1  #!/bin/octave
2
3  a_is_cli_first = 1;
4  a = [
5      [ 1115.5625, 1193.9375 ],
6      [ 1240.4375, 1288.9375 ],
7      [ 1322.625, 1340.5 ],
8      [ 1350, 1566.4375 ],
9      [ 1587.3125, 1618 ],
10     [ 1641.3125, 1665.0625 ],
11     [ 1684, 1708.9375 ],
12     [ 1727, 1759.75 ],
13     [ 1787.0625, 1800.125 ],
14     [ 1824.0625, 1845.6875 ],
15     [ 1887.25, 1907.0625 ],
16     [ 1915.3125, 1929.75 ],
17     [ 2050.3125, 2090.1875 ],
18     [ 2107.1875, 2178.3125 ],
19     [ 2219.625, 2259.75 ],
20     [ 2270.25, 2291.75 ],
21     [ 2431.375, 2465.3125 ],
22     [ 2474.9375, 2486.8125 ],
23     [ 2496.75, 2706.375 ],
24     [ 2772.9375, 2784.375 ],
25     [ 3295.5625, 3374.1875 ],
26     [ 3388.5, 3404.9375 ],
27     [ 3439.5, 3482.0625 ],
28     [ 3491.6875, 3586.8125 ],
29     [ 3597.75, 3656.1875 ],
30     [ 3663.5625, 3698.1875 ],
31     [ 3727, 3797.25 ],
32     [ 3805.0625, 3831.625 ],
33     [ 3835.5625, 3938.125 ],
34     [ 3942.25, 4042.75 ],
35     [ 4067.5625, 4154.5 ],
36     [ 4167.5, 4215 ],
37     [ 4227.0625, 4296.375 ],
```

A.2. ROHDATEN

```
38 [ 4373.75, 4411.375 ],
39 [ 4422.25, 4480.125 ],
40 [ 4485.1875, 4498.0625 ],
41 [ 4507, 4579.25 ],
42 [ 4589.125, 4646.4375 ],
43 [ 4651.1875, 4655.6875 ],
44 [ 4664.625, 4675.5625 ],
45 ];
46
47 b_is_cli_first = 1;
48 b = [
49 [ 24.83, 50.61 ],
50 [ 91.76, 193.67 ],
51 [ 210.08, 244.37 ],
52 [ 258.88, 338.75 ],
53 [ 348.67, 377.09 ],
54 [ 455.14, 497.14 ],
55 [ 535.125, 554.0625 ],
56 [ 589.5625, 643.375 ],
57 [ 701.4375, 732.75 ],
58 [ 825.5, 928.625 ],
59 [ 955.0625, 997.375 ],
60 [ 1049.4375, 1053.375 ],
61 [ 1191.4375, 1297.125 ],
62 [ 1323.5625, 1379.9375 ],
63 [ 1408.375, 1469.375 ],
64 [ 1499.0625, 1609.0625 ],
65 [ 1629.25, 1721.625 ],
66 [ 1740.0625, 1872.75 ],
67 [ 1890.8125, 1794.4375 ],
68 [ 1923.5, 1973.25 ],
69 [ 2083.25, 2134.375 ],
70 [ 2348.75, 2367.75 ],
71 [ 2398.5, 2454.8125 ],
72 [ 2466.6875, 2506.9375 ],
73 [ 2540.875, 2607.125 ],
74 [ 2619.4375, 2658.9375 ],
75 [ 2688.5, 2719.375 ],
76 [ 2734.375, 2748.1875 ],
77 [ 2756.5625, 2808.125 ],
78 [ 2814.9375, 2837.9375 ],
79 [ 2853.5, 2896 ],
80 [ 2906.4375, 2991.5625 ],
81 [ 3026.6875, 3227.375 ],
82 [ 3249.5, 3293.6875 ],
83 [ 3309.1875, 3322 ],
84 [ 3342.0625, 3368.75 ],
85 [ 3387.3125, 3442.375 ],
86 [ 3457.25, 3497.3125 ],
87 [ 3503.375, 3507.375 ],
88 [ 3508.3125, 3544.4375 ],
89 ];
90
91 c_is_cli_first = 1;
92 c = [
93 [ 205.625, 266.5 ],
94 [ 286.9375, 405.25 ],
95 [ 428.6875, 454.6875 ],
96 [ 463.75, 589.625 ],
```

A.2. ROHDATEN

```
97 [ 607.375, 637 ],
98 [ 644, 667.625 ],
99 [ 684.3125, 702.25 ],
100 [ 713, 808.375 ],
101 [ 818.1875, 859.125 ],
102 [ 872.8125, 888.25 ],
103 [ 926.625, 967.375 ],
104 [ 971.0625, 988.5 ],
105 [ 999.3125, 1109.1875 ],
106 [ 1130, 1219.1875 ],
107 [ 1235.4375, 1269.0625 ],
108 [ 1275.875, 1359.5625 ],
109 [ 1389.1875, 1509.25 ],
110 [ 1545, 1584.875 ],
111 [ 1609.25, 1615 ],
112 [ 1616.875, 1624.25 ],
113 [ 1949.6875, 2049.5 ],
114 [ 2066.9375, 2085.5625 ],
115 [ 2096.0625, 2130.75 ],
116 [ 2137.5625, 2163 ],
117 [ 2202.625, 2216.5 ],
118 [ 2225.75, 2274.75 ],
119 [ 2282.625, 2318.875 ],
120 [ 2334.0625, 2348.9375 ],
121 [ 2356.75, 2402.4375 ],
122 [ 2409.9375, 2444.375 ],
123 [ 2452.625, 2618.75 ],
124 [ 2637.25, 2696.5 ],
125 [ 2716.75, 2798.4375 ],
126 [ 2813.25, 2853.8125 ],
127 [ 2858.4375, 2889.25 ],
128 [ 2894.6875, 2911.0625 ],
129 [ 2917.1875, 2945.5625 ],
130 [ 2958.9375, 2993.375 ],
131 [ 2998.1875, 3029.375 ],
132 [ 3035.0625, 3042.1875 ],
133 ];
134
135 d_is_cli_first = 0;
136 d = [
137 [ 443.4375, 510.5 ],
138 [ 525.75, 549.9375 ],
139 [ 718.0625, 826.625 ],
140 [ 835.4375, 898.3125 ],
141 [ 904.25, 931.6875 ],
142 [ 941.9375, 974.75 ],
143 [ 986.5625, 1041.125 ],
144 [ 1046.75, 1077 ],
145 [ 1084.125, 1111 ],
146 [ 1113.3125, 1213.8125 ],
147 [ 1254.75, 1306.3125 ],
148 [ 1320.5, 1375.9375 ],
149 [ 1480.3125, 1611.5625 ],
150 [ 1621.875, 1709.5625 ],
151 [ 1718.3125, 1786.375 ],
152 [ 1789.3125, 1805.5625 ],
153 [ 1833, 1847.8125 ],
154 [ 1856, 1865.9375 ],
155 [ 1873.625, 1880.0625 ],
```

A.2. ROHDATEN

```
156 [ 1883.5, 1897.8125 ],
157 [ 2339.6875, 2372.5625 ],
158 [ 2381.8125, 2491.75 ],
159 [ 2499.4375, 2533.1875 ],
160 [ 2542.6875, 2573.8125 ],
161 [ 2602.1875, 2663.8125 ],
162 [ 2673.5625, 2710.0625 ],
163 [ 2720.0625, 2821.5625 ],
164 [ 2831.5625, 2856.75 ],
165 [ 2873.125, 2924.375 ],
166 [ 2941.0625, 2961 ],
167 [ 2970.375, 3004.8125 ],
168 [ 3016.375, 3022 ],
169 [ 3044.125, 3076.5625 ],
170 [ 3085.0625, 3178.1875 ],
171 [ 3185.0625, 3236.25 ],
172 [ 3242.65625, 3319 ],
173 [ 3392.25, 3425.1875 ],
174 [ 3431.59375, 3450.4375 ],
175 [ 3456.625, 3545.5625 ],
176 [ 3548.90625, 3563.84375 ],
177 ];
178
179 e_is_cli_first = 0;
180 e = [
181 [ 242.4375, 343.25 ],
182 [ 404.8125, 444.125 ],
183 [ 468.125, 538.75 ],
184 [ 551.875, 835 ],
185 [ 848.6875, 926.3125 ],
186 [ 944.9375, 993.8125 ],
187 [ 1001.75, 1080 ],
188 [ 1086.75, 1107.3125 ],
189 [ 1114.125, 1149.1875 ],
190 [ 1155.3125, 1183.3125 ],
191 [ 1188.1875, 1282.375 ],
192 [ 1291.4375, 1430.1875 ],
193 [ 1439.9375, 1549.9375 ],
194 [ 1562.0625, 1658.25 ],
195 [ 1693.875, 1734.4375 ],
196 [ 1743.5625, 1771.75 ],
197 [ 1787.875, 1842.625 ],
198 [ 1848.4375, 1879.125 ],
199 [ 1889.8125, 1895.125 ],
200 [ 1904.375, 1930.875 ],
201 [ 2741.125, 2807.9375 ],
202 [ 2820.625, 2885.75 ],
203 [ 2894.25, 2961.6875 ],
204 [ 2974.25, 3060.3125 ],
205 [ 3079.875, 3098.0625 ],
206 [ 3117.5, 3270.5 ],
207 [ 3283.25, 3312.5 ],
208 [ 3339.3125, 3441.1875 ],
209 [ 3457.1875, 3499.875 ],
210 [ 3512.8125, 3537.3125 ],
211 [ 3721.625, 3746.5 ],
212 [ 3751.75, 3775.125 ],
213 [ 3790.625, 3839.4375 ],
214 [ 3861.25, 3929 ],
```

A.2. ROHDATEN

```
215 [ 3975.375, 4009.875 ],
216 [ 4015, 4074.1875 ],
217 [ 4089.75, 4137.125 ],
218 [ 4144.625, 4168.0625 ],
219 [ 4177.5, 4194.4375 ],
220 [ 4211.1875, 4228.875 ],
221 ];
222
223 f_is_cli_first = 0;
224 f = [
225 [ 422.5, 480.3125 ],
226 [ 492.25, 515.9375 ],
227 [ 568.5625, 583.25 ],
228 [ 587.4375, 633.5625 ],
229 [ 644.0625, 663 ],
230 [ 668, 692.25 ],
231 [ 705.0625, 774.5 ],
232 [ 786.375, 806.5625 ],
233 [ 813.9375, 831 ],
234 [ 836.25, 856.625 ],
235 [ 862.1875, 906.8125 ],
236 [ 912.6875, 995 ],
237 [ 1005.1875, 1100.875 ],
238 [ 1107.8125, 1150.5625 ],
239 [ 1156.5, 1180.6875 ],
240 [ 1190.0625, 1215.1875 ],
241 [ 1231.5625, 1272.375 ],
242 [ 1282.8125, 1298.1875 ],
243 [ 1302, 1308.5 ],
244 [ 1312.0625, 1329.375 ],
245 [ 1999.3125, 2023.3125 ],
246 [ 2115.625, 2135.5 ],
247 [ 2250.4375, 2265.375 ],
248 [ 2294.25, 2329.125 ],
249 [ 2347.25, 2356.3125 ],
250 [ 2401.25, 2431.6875 ],
251 [ 2446.875, 2462.0625 ],
252 [ 2492.375, 2527.4375 ],
253 [ 2533.5625, 2546.9375 ],
254 [ 2551.6875, 2569.125 ],
255 [ 2584.25, 2607.3125 ],
256 [ 2611.875, 2624.625 ],
257 [ 2638.3125, 2703.75 ],
258 [ 2732.9375, 2774.125 ],
259 [ 2836.4375, 2885.1875 ],
260 [ 2887.125, 2904.625 ],
261 [ 2913.5, 2962.53125 ],
262 [ 2982.96875, 2995.90625 ],
263 [ 3036.3125, 3077.125 ],
264 [ 3077.71875, 3085.09375 ],
265 ];
266
267 # cli = data{1};
268 # gui = data{2};
269 function res = putCliFirst(arr, is_cli_first)
270     if ( length(arr) ≠ 40 )
271         "error: len ≠ 40"
272     endif
273
```

A.2. ROHDATEN

```

274 starts = arr(:,1);
275 ends = arr(:,2);
276 diffs = ends - starts;
277
278 if ( is_cli_first == 1 )
279     res = {
280         diffs(1:20),
281         diffs(21:40)
282     };
283 else
284     res = {
285         diffs(21:40),
286         diffs(1:20)
287     };
288 endif
289 endfunction
290 function res = cli(data)
291     res = data{1};
292 endfunction
293 function res = gui(data)
294     res = data{2};
295 endfunction
296
297 a = putCliFirst(a, a_is_cli_first);
298 b = putCliFirst(b, b_is_cli_first);
299 c = putCliFirst(c, c_is_cli_first);
300 d = putCliFirst(d, d_is_cli_first);
301 e = putCliFirst(e, e_is_cli_first);
302 f = putCliFirst(f, f_is_cli_first);
303 avgCli = (cli(a) + cli(b) + cli(c) + cli(d) + cli(e) + cli(f))/6;
304 avgGui = (gui(a) + gui(b) + gui(c) + gui(d) + gui(e) + gui(f))/6;
305
306 function p = combinedPerformance(data)
307     p = bar(data, "grouped", "edgecolor", "white");
308     set(p(1), "facecolor", "#094F9F");
309     set(p(2), "facecolor", "#FD7F00");
310     axis([ 0 21 ]);
311     legend({"CLI", "GUI"});
312     set(gca, 'xtick', 1:1:20);
313     ylabel("Zeit in s");
314     xlabel("Aufgaben");
315     title("Performance");
316     set(gca, "fontsize", 14);
317 endfunction
318
319 averageTaskTimes = [
320     avgCli, avgGui;
321 ];
322 figure(1);
323 p = combinedPerformance(averageTaskTimes);
324
325 function p = cliPerformance(data)
326     p = bar(data, "grouped", "edgecolor", "white", "facecolor", "#094F9F", "barwidth", .27);
327     axis([ 0 21 ]);
328     legend({"CLI"});
329     set(gca, 'xtick', 1:1:20);
330     ylabel("Zeit in s");
331     xlabel("Aufgaben");
332     title("Cli Performance");

```


A.2. ROHDATEN

```
333     set(gca, "fontsize", 14);
334 endfunction
335
336 figure(2);
337 p = cliPerformance(avgCli);
338
339 function b = percentDiffBar(x)
340     hold on
341     for i = 1 : length(x)
342         if (x(i) < 0)
343             b = bar( i, x(i), 'facecolor', "#094F9F", "barwidth", .42, "edgecolor", "white" );
344         else
345             b = bar( i, x(i), 'facecolor', "#FD7F00", "barwidth", .42, "edgecolor", "white" );
346         endif
347     endfor
348     hold off;
349
350     axis([ 0 21 ]);
351     legend( "CLI schneller", "GUI schneller" );
352     ylabel("Schneller erledigt in %");
353     xlabel("Aufgaben");
354     title("Performance Unterschiede");
355     set(gca, "fontsize", 14);
356     set(gca, 'xtick', 1:1:20);
357     set(gca, 'ytick', -500:50:350);
358 endfunction
359
360 function diff = diff(data)
361     diff = cli(data) - gui(data);
362 endfunction
363 avgTimeDifferences = \
364     (diff(a) + diff(b) + diff(c) + diff(d) + diff(e) + diff(f))/6;
365 meanAvgTimeDifference = \
366     sprintf("%.2f", sum(avgTimeDifferences)/length(avgTimeDifferences))
367
368 minimums = min(avgCli, avgGui);
369 differences = avgCli - avgGui;
370 percentDifference = differences.*100./minimums;
371
372 figure(3);
373 p = percentDiffBar(percentDifference);
374
375 waitfor(p);
```

A.2. ROHDATEN

Teilnehmer	Bearbeitung	einfach - hässlich	praktisch	stilvoll - schön	voraussage	minderwertig	phantasie	gut - schlecht	verwirrend	lahm - fesselnd	
		ein_kom	hae_sch	pra_unp	sti_sti	vor_unb	min_wer	pha_kre	gut_sch	ver_ueb	lah_fes
c4d00c8e	110	2	1	1	1	4	2	1	1	2	1
48026be4	111	4	4	5	3	5	3	2	4	5	2
f0af7652	111	3	2	4	3	5	2	2	4	3	2
f96d70eb	119	2	1	1	2	1	2	1	1	1	1
7b943829	70	2	2	3	2	1	2	4	3	2	1
c2a34427	128	1	2	1	2	1	3	1	1	1	4

Abbildung A.1.: Daten der Attraktivitätsumfrage: GUI

Teilnehmer	Bearbeitung	einfach - hässlich	praktisch	stilvoll - schön	voraussage	minderwertig	phantasie	gut - schlecht	verwirrend	lahm - fesselnd	
		ein_kom	hae_sch	pra_unp	sti_sti	vor_unb	min_wer	pha_kre	gut_sch	ver_ueb	lah_fes
a4751c88	54	6	6	6	6	7	6	7	7	6	6
2f41629f	95	7	7	7	7	7	7	6	7	7	7
ca1f0db0	120	5	4	2	4	6	5	6	6	6	5
4d957323	72	5	4	5	5	6	2	6	6	6	5
dd37490f	326	7	7	7	7	7	6	7	7	2	7
a3ad4ad8	91	7	6	7	4	7	6	6	7	7	7

Abbildung A.2.: Daten der Attraktivitätsumfrage: CLI

Eidesstattliche Versicherung

Hiermit versichere ich an Eides statt durch meine Unterschrift, dass ich die vorstehende Arbeit selbstständig und ohne fremde Hilfe angefertigt und alle Stellen, die ich wörtlich oder annähernd wörtlich aus Veröffentlichungen entnommen habe, als solche kenntlich gemacht habe, mich auch keiner anderen als der angegebenen Literatur oder sonstiger Hilfsmittel bedient habe. Die Arbeit hat in dieser oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen.

13.02.23, Berlin,

Datum, Ort, Unterschrift