



# HANGMAN

Elias Knoll, Silvan Horn, Antonija Pavic, Alessia Meier

## Inhaltsverzeichnis Hangman-Projekt

Abbildungsverzeichnis .....	3
Tabellenverzeichnis .....	3
Versionierung.....	3
Kontaktdaten .....	4
Einleitung .....	5
Vorgehensweise .....	5
Planung.....	5
Entwicklung.....	6
Testen .....	6
Lastenheft .....	7
Zielsetzung.....	7
Benutzeroberfläche .....	7
Spielregeln .....	8
Pflichtenheft .....	9
Rahmenbedingungen .....	9
Ziel des Spiels .....	9
Funktionale Anforderungen .....	9
Nichtfunktionale Anforderungen.....	10
Technische Umsetzung .....	10
Umsetzung funktionaler Anforderungen .....	10
Umsetzung nichtfunktionaler Anforderungen .....	14
Technische Spezifikationen .....	15
Pipeline .....	16
Testing .....	16
Formatting.....	16
Deployment.....	16
Testing.....	17
Testverfahren.....	17
Testaufbau .....	17
Testfälle .....	19
Testdrehbuch .....	21

Testprotokoll.....	23
Testbericht .....	27
Fazit .....	27
Funktionsdefinition .....	28
Struktogramm .....	29
Funktionsbeschrieb.....	30
main .....	30
processing_guess .....	30
Initialising.....	30
alphabet_block.....	30
display_game_frame .....	30
endings_order.....	31
display_startup.....	31
gallows .....	31
wrong 1-6 .....	31
mistakesN .....	31
init_wordfile.....	31
choose_word.....	32
Variabel Beschreibung.....	33
Beispielausgabe.....	34
Quellcode.....	34
KVP .....	34
GUI .....	35
Grössere Auswahl der Wörter .....	35
Erstellen von .EXE-Datei .....	35
Fazit .....	36

## Abbildungsverzeichnis

Abbildung 1: Aufgabenverteilung .....	6
Abbildung 2: Darstellung Hangman-Figur .....	7
Abbildung 3: Code-Snippet Wortauswahl.....	11
Abbildung 4: Code-Snippet Eingabe und Buchstabenprüfung.....	12
Abbildung 5: Code-Snippet Versuchszähler.....	13
Abbildung 6: Darstellung Hangman ABC .....	14
Abbildung 7: Modul endings.sh.....	14
Abbildung 8: run-tests.sh .....	18
Abbildung 9: All Tests passed .....	18
Abbildung 10: Beispieldaten.....	34

## Tabellenverzeichnis

Tabelle 1: Versionierung .....	3
Tabelle 2: Kontaktdaten .....	4
Tabelle 3: Technische Spezifikationen .....	15
Tabelle 4: Testfälle .....	20
Tabelle 5: Testdrehbuch .....	22
Tabelle 6: Testprotokoll .....	26
Tabelle 7: Funktionsdefinition.....	28

## Versionierung

Diese Versionierung zeigt auf, von wem wann welche Änderungen an dieser Dokumentation vorgenommen wurden.

Version	Ersteller	Datum	Kommentar
0.1	Silvan Horn	12.05.2025	Erster Entwurf
0.2	Silvan Horn	12.05.2025	wichtige Themen und Kapitel hinzugefügt
0.3	Alessia Meier	12.05.2025	Lastenheft hinzugefügt
0.4	Elias Knoll	19.05.2025	Hinzufügen Testing
0.5	Silvan Horn	26.05.2025	Testing überarbeitet
1.0	Silvan Horn	08.06.2025	Dokumentation finalisieren

Tabelle 1: Versionierung

## Kontaktdaten

In der nachfolgenden Tabelle sind die Kontaktdaten des Entwicklerteams ersichtlich.

Name	Vorname	E-Mail
Pavic	Antonija	<a href="mailto:Antonija.pavic@stud.bzbs.ch">Antonija.pavic@stud.bzbs.ch</a>
Meier	Alessia	<a href="mailto:Alessia.meier@stud.bzbs.ch">Alessia.meier@stud.bzbs.ch</a>
Knoll	Elias	<a href="mailto:Elias.Knoll@stud.bzbs.ch">Elias.Knoll@stud.bzbs.ch</a>
Horn	Silvan	<a href="mailto:Silvan.horn@stud.bzbs.ch">Silvan.horn@stud.bzbs.ch</a>

Tabelle 2: Kontaktdaten

## Einleitung

Im Rahmen des vorliegenden Projektes wurde ein Hangman-Game als Konsolen-Anwendung in Bash realisiert.

Ziel des Projektes war es, grundlegende Konzepte der Shell praktisch anzuwenden und ein funktionsfähiges Programm zu entwickeln.

Die Umsetzung erfolgte testgetrieben (**TDD test-driven-development**). Das heisst, dass wir bevor wir mit der eigentlichen Implementierung der Funktionen begannen, bereits die Testfälle definierten und anschliessend die Tests schrieben.

Dadurch konnten Anforderungen frühzeitig geklärt werden und Fehler bereits im Entwicklungsprozess erkannt und behoben werden. Dies vermindert das Auftreten von unerwarteten Fehlern (**Bugs**).

Die Entwicklung des Projektes erfolgte funktionsorientiert. Das bedeutet, dass wir für die unterschiedlichen Funktionalitäten jeweils unterschiedliche Funktionen definierten. Wir achteten bei der Entwicklung darauf, möglichst wenige globale Variablen zu verwenden und stattdessen Variablen als Parameter zu übergeben.

Wenn möglich haben wir diese als Referenz übergeben.

Diese Herangehensweise verbesserten nicht nur die Lesbarkeit des Codes, sondern waren auch wichtig für die Testbarkeit.

## Vorgehensweise

In diesem Kapitel ist beschrieben, wie wir bei der Arbeit an diesem Projekt vorgingen.

### Planung

Als wir den Auftrag bekamen, ein Projekt in Bash zu entwickeln, mussten wir uns zuerst einige grundlegende Gedanken machen.

Unter anderem standen wir vor der Frage, was für ein Projekt wir entwickeln wollten. Ein weiterer wichtiger Punkt, den wir zu lösen hatten, war es, die Aufgaben zu verteilen.

Nach einigen Gesprächen wurde klar, dass wir ein simples Game entwickeln wollten. Schnell fiel daraufhin die Entscheidung auf ein Hangman Game.

Für die Aufgabenverteilung haben wir eine Excel-Liste erstellt, die uns dabei half, immer im Blick zu haben, welche Aufgabe von wem erledigt wird. Sie half auch, zu sehen, wann die Aufgaben erledigt wurden.

Was	Bemerkungen	Verantwortlich	Fertiggestellt am	Kontrolle	Besprochen am	Abgelegt auf
Lastenheft		Alessia & Antonia	12.05.2025	Silvan		
Pflichtenheft		Silvan	12.05.2025	Alessia & Antonia		
Struktogramm/PAP		Silvan	12.05.2025	Elias	12.05.2025	<a href="#">bash-hangman/struktogramm at main · KnollElias/bash-hangman · GitHub</a>
Testplan/Testdokumentation		Elias	28.04.2025	Alessia		<a href="https://github.com/KnollElias/bash-hangman/tree/main/hangman/tests">https://github.com/KnollElias/bash-hangman/tree/main/hangman/tests</a>
funktionsorientierter Projektaufbau		Elias	19.05.2025	Silvan		
Code		Elias	19.05.2025	Silvan & Alessia		
Dokumentation		Alessia				
präsentation		Antonija				

Abbildung 1: Aufgabenverteilung

Ein weiterer wichtiger Punkt war, dass wir bereits am Anfang definierten, dass wir sämtliche Dokumente auf das GitHub-Repository hochladen.

Dadurch konnten wir Missverständnisse vermindern und die Effizienz steigern.

## Entwicklung

Die Anwendung wurde in **Bash** (Bourne Again Shell) **geschrieben**. Bash ist die freie und erweiterte Version der Bourne-Shell, die zu den Betriebssystemen GNU/Linux gehört. Somit **kann** dieses Programm **auf fast jedem Gerät ausgeführt werden**, solange ein **Shell-Interpreter** vorhanden ist.

Die **Versionsverwaltung** erfolgte über Github. Details dazu folgen im weiteren Verlauf dieser Projektdokumentation.

## Testen

Für das Testen an diesem Projekt bekamen wir die Vorgabe, nach **Test-Driven-Development** zu entwickeln. Bei dieser Entwicklungsstrategie werden während der Entwicklung unit-tests geschrieben.

Diese Unit-Tests wurden umgesetzt, bevor mit der tatsächlichen Entwicklungsarbeit in Form der Implementierung der Funktionen begonnen wurde.

Auf diese Weise konnte sichergestellt werden, dass sich das Projekt wie vorgesehen entwickelte.

Die unit-tests werden vor jedem Deployment ausgeführt.

Der Code muss dann so lange optimiert werden, bis alle Tests bestanden sind.

Wenn ein Test neu den Status „bestanden“ erreicht, ist das Ziel der Entwickler, das betreffende Code-snippet so weit zu refaktorieren, dass es den Test immer noch besteht, aber möglichst leichtgewichtig ist.

Neben dem kontinuierlichen Testen im Rahmen des TDD wurden zum Abschluss des Projektes einige manuelle Tests durchgeführt, etwa zur Prüfung der Darstellung und der Basisfunktionen.

Auf einen systematischen manuellen Test wurde jedoch bewusst verzichtet, weil das Hangman-Game einen klar definierten Ablauf hat, der durch TDD und gezielte Checks bereits ausreichend abgedeckt ist.

Die genaue Teststrategie ist in dem Kapitel Testing beschrieben.

## Lastenheft

In dem nachfolgenden Lastenheft sind die Anforderungen der Stakeholder an das Projekt beschrieben.

### Zielsetzung

Das Hangman-Spiel soll es ermöglichen, ein geheimes Wort zu erraten. Die Versuche sind begrenzt durch Striche, welche schlussendlich einen «Hangman» bilden. Wenn der Hangman komplett und das Wort noch nicht erraten ist, hat der Benutzer verloren. Das Wort kann durch Raten der einzelnen Buchstaben aufgedeckt werden.

Wenn der Benutzer das Wort errät, bevor der Hangman komplett ist, hat der Benutzer gewonnen. Der Hangman besteht aus sechs Strichen, somit darf der Benutzer bis zu sechs falschen Buchstaben raten, bevor das Spiel verloren ist.

### Benutzeroberfläche

Die Benutzeroberfläche sollte aus der Hangman-Figur, der Eingabeaufforderung, dem maskierten Wort, sowie einem Alphabet-Block bestehen.

Der Alphabet-Block sollte eine Darstellung sämtlicher Buchstaben im Alphabet sein. In diesem sollten die richtig und falsch geratenen Buchstaben durch Farben erkenntlich gemacht werden.

Die Figur sollte der Darstellung im nebenstehenden Bild entsprechen. Der Benutzer sollte Striche anstelle von den Buchstaben sehen. Bei jedem richtig geratenen Buchstaben sollte der Buchstabe an der richtigen Stelle erscheinen. Bei falschen Eingaben sollte ein Teil des Hangmans erscheinen.

Nachdem das richtige Wort erraten wurde, sollte einer der folgenden Texte erscheinen, abhängig von der Anzahl Fehlern, die der Benutzer gemacht hat:

0. Perfekt! 🤘 Du hast das Wort ohne Fehler erraten.
1. Wegen dieses einen Fehlers hast du es leider nicht fehlerfrei geschafft.
2. Das kannst du mit weniger Fehlern!
3. Geschafft, aber mit vielen Fehlern. Übung macht den Meister.
4. Wenn du nochmals so viele Fehler machst, wird es aber knapp. 😥
5. Kopf hoch, das war nicht das Ende.
6. Bei 6 Fehlern hat der Benutzer verloren

Abbildung 2: Darstellung Hangman-Figur



## Spielregeln

Der Benutzer sollte vom Spiel ein zufälliges Wort bekommen. Der Benutzer versucht das Wort zu erraten und nennt einen Buchstaben pro Runde. Wenn der Buchstabe im Wort vorkommt, sollten alle Stellen mit diesem Buchstaben aufgedeckt werden. Ist der Buchstabe falsch, sollte ein Teil vom Hangman ergänzt werden. Nach sechs Fehlversuchen sollte der Benutzer das Spiel verloren haben.

Das Ziel dieses Spiels ist es, das Wort zu erraten bevor der Hangman komplett ist.

## Pflichtenheft

In dem nachfolgenden Pflichtenheft ist ersichtlich, wie die im Lastenheft beschriebenen Anforderungen umgesetzt werden.

### Rahmenbedingungen

Im Folgenden sind die Rahmenbedingungen für dieses Projekt detailliert beschrieben. Diese orientieren sich hauptsächlich am Lastenheft.

### Ziel des Spiels

Der Spieler sollte möglichst schnell das richtige Wort erraten.

### Funktionale Anforderungen

Nachfolgend sind funktionale Anforderungen an dieses Projekt beschrieben.

#### *Wortauswahl*

Die Wortauswahl erfolgt zufällig aus einer Wörterliste.

Der Benutzer bekommt von der Wortauswahl nichts mit.

#### *Buchstabenprüfung*

Die eingegebenen Buchstaben werden von dem Script überprüft.

Daraufhin erfolgt eine entsprechende Reaktion.

#### *Eingabe*

Der Benutzer kann pro Runde einen Buchstaben raten.

#### *Versuchszähler*

Das Script zählt die Anzahl der Versuche, die der Benutzer benötigt.

Der Benutzer darf höchstens sechs Versuche brauchen.

#### *Anpassbare Wortauswahl*

Der Benutzer kann im Script eine Option auswählen, die eine erweiterte Wortauswahl ermöglicht.

Wenn diese Option ausgewählt wird, kann der Benutzer eine Datei auswählen, welche anschliessend als Wordlist fungiert.

Nachdem der Benutzer die Datei ausgewählt hat, kann er auswählen, ob seine Wörterliste die Standard Wörterliste ersetzen oder ergänzen soll.

#### *Highscore-System*

In einem separaten .txt-file wird der Highscore jeweils mit Namen und Wert gespeichert.

Dazu wird jeweils, nachdem der Spieler die Runde beendet hat, der Name abgefragt.

Anschliessend wird überprüft, ob der besagte Benutzer schon in der Datei enthalten ist, wenn das der Fall ist, wird überprüft, ob die aktuelle Versuchszahl niedriger ist.

Wenn dies der Fall ist, wird der Highscore der jeweiligen Person ersetzt.  
Damit ein solches System funktionieren kann, ist der Versuchszähler erforderlich.

Von den oben genannten Anforderungen werden die Wortauswahl, die Buchstabenprüfung sowie die Eingabe als verpflichtende Ziele angesehen.

Die Anforderungen an den Versuchszähler, die anpassbare Wortauswahl und das Highscore-System werden als optionale Anforderungen betrachtet.

## Nichtfunktionale Anforderungen

### *Intuitive Benutzeroberfläche*

Die Benutzeroberfläche sollte ansprechend sein und dem Benutzer dabei helfen, das Spiel zu verstehen.

### *Motivierende Benutzerinteraktion*

Wenn ein Spiel fertig ist, wird entsprechend der Anzahl der Fehler ein Satz ausgegeben.

### *Stabilität*

Das Spiel sollte stabil, flüssig und ohne Fehler laufen.

### *Case-Insensitiveness*

Bei den Eingaben werden sowohl Grossbuchstaben als auch Kleinbuchstaben akzeptiert.

Von den oben genannten Anforderungen werden die intuitive Benutzeroberfläche, die motivierenden Benutzerinteraktionen und die Stabilität als verpflichtende Anforderungen angesehen.

Die Anforderung der Case-Insensitiveness wird als optional angesehen.

## Technische Umsetzung

In diesem Abschnitt wird beschrieben, wie die vorhergehend beschriebenen Anforderungen technisch umgesetzt wurden.

## Umsetzung funktionaler Anforderungen

Im folgenden Abschnitt wird die Umsetzung der beschriebenen funktionalen Anforderungen dargestellt.

### *Wortauswahl*

Beim Start des Programmes wird sichergestellt, dass eine Datei mit dem Namen wordlist\_de.txt vorhanden ist. Diese wird erstellt, falls sie noch nicht existiert. Aus dieser Liste wird dann ein zufälliges Wort gelesen.

Zuerst wird die Liste gelesen, von welcher anschliessend ein zufälliges Wort ausgewählt wird.

Auf dem untenstehenden Bild ist die Umsetzung anhand eines Code-Snippets ersichtlich.

```
choose_word() {  
    init_wordfile  
    local w=""  
    # so lange wiederholen, bis wir wirklich ein nicht-leeres Wort haben  
    while [[ -z "$w" ]]; do  
        w=$(shuf -n1 "$wordfile" | tr -d '\r\n')  
    done  
    # einmalig anhängen, falls neu (nice-to-have)  
    grep -qxF "$w" "$wordfile" || echo "$w" >> "$wordfile"  
    echo "$w"  
}
```

Abbildung 3: Code-Snippet Wortauswahl

### *Eingabe*

In Abbildung 4 ist ersichtlich, wie die Benutzereingaben eingelesen werden.

Der gesamte Block befindet sich in einer Endlosschleife. Das bedeutet, dass er so lange ausgeführt wird, bis durch den break-Befehl die Schleife verlassen wird.

Mit read lesen wir Benutzereingaben in das Script ein.

Mit dem Parameter -r stellen wir sicher, dass wir keine Probleme mit Escape-Zeichen bekommen.

Mit dem Parameter -p können wir den read-Befehl mit einer Anweisung ausgeben.

Es wird genau ein Zeichen, ohne auf ein Enter-Drücken zu warten, eingelesen.

Dieses wird der Variablen ltr zugewiesen. Dieser Name bedeutet letter.

Wenn read einen Fehler zurückgeben sollte, wird ein Zeilenumbruch gemacht und der aktuelle Schleifendurchlauf übersprungen. Es wird eine neue Schleifendurchlauf gestartet. Das bedeutet für den Benutzer, dass er nochmals einen Buchstaben eingeben muss.

### Buchstabenprüfung

In der Abbildung 4 ist ebenfalls ersichtlich, wie die Eingaben geprüft werden.

Nachdem die Eingabe erfolgt ist, wird geprüft, ob es sich um einen Buchstaben handelt.

Wenn das nicht erfüllt ist, wird eine Fehlermeldung ausgegeben, sowie die aktuelle Iteration beendet.

Das bedeutet für den Benutzer, dass er einen neuen Buchstaben eingeben muss.

Wenn dieser Schritt erfolgreich war, wird der Buchstabe lower-case gemacht.

Damit wird das Ziel der Case-Insensitiveness erfüllt.

Anschliessend wird die Funktion process\_guess aufgerufen. Was diese macht, ist im Abschnitt Versuchszähler ersichtlich.

Anschliessend wird die Schleife mit break verlassen.

```
while true; do
    read -rp "Rate einen Buchstaben (a-z): " -n1 ltr || {
        echo
        continue
    }
    echo
    [[ "$ltr" =~ [A-Za-z] ]] || {
        echo "Bitte gib einen Buchstaben ein."
        continue
    }
    ltr=${ltr,,}

    process_guess guessed_ok guessed_bad $ltr wrongstate

    break
done
done
```

Abbildung 4: Code-Snippet Eingabe und Buchstabenprüfung

### Versuchszähler

```
[[" ${ok_guess[*]} ${bad_guess[*]} " == *$ltr* ]] && user_error="schon geraten" && echo $user_error && return  
if [[ "$secret" == *$ltr* ]]; then  
| ok_guess+=("$ltr")  
else  
| bad_guess+=("$ltr")  
| ((wrong_count++))  
fi
```

Abbildung 5: Code-Snippet Versuchszähler

In dem obenstehenden Bild ist ersichtlich, wie die Versuchszählung implementiert wurde.

Zuerst wird geprüft, ob der geratene Buchstaben in der \$ltr Variabel in einem der beiden Arrays gespeichert ist.

Wenn dies der Fall ist, wird die Funktion verlassen und die Arrays sowie die Wrong\_Count Variabel, welche die Fehlversuche zählt, werden nicht weiter befüllt.

Wenn die beschriebene Bedingung allerdings nicht zutrifft, wird zuerst geprüft, ob der geratene Buchstaben in der Variabel \$secret, welche das Geheimwort enthält, ist.

Wenn das zutrifft, war der Versuch erfolgreich und der Array ok\_guess wird um den aktuellen Buchstaben ergänzt.

Wenn diese Bedingung nicht zutrifft, wird der aktuelle Buchstaben in den Array bad\_guess hinzugefügt.

## Umsetzung nichtfunktionaler Anforderungen

### *Intuitive Benutzeroberfläche*

Diese Anforderung wird durch ein aufgeräumtes, minimalistisches und selbsterklärendes Design erfüllt. Dieses ist auf dem nachfolgenden Bild ersichtlich.

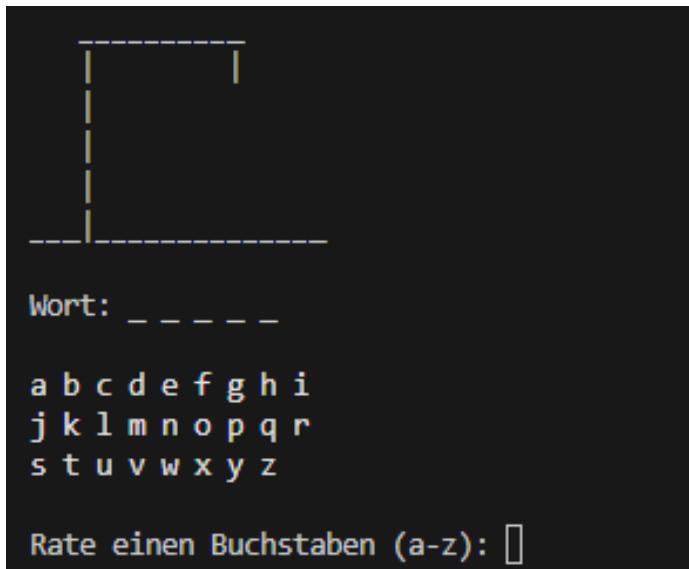


Abbildung 6: Darstellung Hangman ABC

### *Motivierende Benutzerinteraktion*

Um dem Benutzer bei Gewinnen des Spiels eine individuelle Rückmeldung, basierend auf der Anzahl der gemachten Fehler zu geben, haben wir ein Modul mit den folgenden Funktionen erstellt.

Dieses ist in der nachfolgenden Abbildung ersichtlich.

```
mistakes0() {
    output="Perfekt! 🌟 Du hast das Wort ohne Fehler erraten."
}
mistakes1() {
    output="Wegen dieses einen Fehlers hast du es leider nicht fehlerfrei geschafft."
}
mistakes2() {
    output="Das kannst du mit weniger Fehlern!"
}
mistakes3() {
    output="Geschafft, aber mit vielen Fehlern. Übung macht den Meister."
}
mistakes4() {
    output="Wenn du nochmal so viele Fehler machst, wird es aber knapp. 😞"
}
mistakes5() {
    output="Kopf hoch, das war nicht das Ende."
}
```

Abbildung 7: Modul endings.sh

Die `mistakesx`-Funktionen werden beim Gewinnen mit einem Case-Statement aufgerufen.

### *Stabilität*

Die vielen **detaillierten Tests** sorgen für eine hohe Funktionsabdeckung und vermindern das Risiko für unvorhergesehene Fehler.

## Technische Spezifikationen

Entwicklungsumgebung	Visual Studio Code / vi
Script-Sprache	Bash
Benutzeroberfläche	CLI (command line interface)
Ausführung	Lokal / GitHub Pipeline
Versionskontrolle	Git

Tabelle 3: Technische Spezifikationen

## Pipeline

In diesem Kapitel wird detailliert beschrieben, wie wir die Continuous-Integration Pipeline in Form von Github Actions bei diesem Projekt für unsere Zwecke einsetzen konnten.

Bei sämtlichen Pushes auf die Main-Branch wird automatisch das workflow-File ausgeführt.

## Testing

Sämtliche Tests, die wir im Rahmen der TDD-Entwicklungsmethode in diesem Projekt durchführten, werden in der Pipeline ausgeführt.

Der Vorteil davon war, dass wir im main-Branch nur funktionierenden, getesteten Code hatten.

Ein weiterer Vorteil der Tests in der Pipeline war, dass eine universale Umgebung verwendet wurde, so konnten Fehler aufgrund lokaler Probleme eliminiert werden.

## Formatting

Zur automatisierten Formatierung des Quellcode in der Pipeline nutzen wir das Tool shfmt. Es wird bei jeden Push ausgeführt und sorgt dafür, dass definierte Stilregeln eingehalten werden.

Dadurch können wir eine bessere Lesbarkeit und allgemein stärkere Code-Qualität gewährleisten.

## Deployment

In der Pipeline haben wir ein Feature installiert, welches automatisch aus den pushes eine .exe-Datei erstellt.

Diese Funktionalität funktioniert zum aktuellen Stand noch nicht wie gewünscht.

Wir haben uns aber entschieden, dass wir uns auf die Haupt-Entwicklungsaufgaben, die dieses Projekt mit sich bringt, konzentrieren wollen.

Aus diesem Grund haben wir dieses Feature nicht mehr weiterentwickelt.

## Testing

In dem nachfolgenden Kapitel der Testprozess dieses Projektes detailliert beschrieben.

### Testverfahren

Es wurde nach dem TDD-Verfahren gearbeitet.

Das Testing wird hauptsächlich nach dem TDD-Verfahren gemacht.

TDD ist Englisch und steht für Test getriebene Entwicklung.

In der testgetriebenen Entwicklung werden die Tests als zentrale Funktion der Entwicklung angesehen. Sie werden nicht erst im Nachhinein durchgeführt, sondern fortlaufend während des Entwicklungsprozesses.

Daneben finden allerdings auch manuelle Testmethoden Verwendung in diesem Projekt. Diese wurden hauptsächlich dazu verwendet, graphische Darstellungen zu überprüfen.

### Testaufbau

Für die Tests gibt es in dem Projekt-Repository einen eigenen Ordner. Dieser heisst tests. Darin werden die Unit-Tests von dem run-tests.sh Skript aufgerufen.

Dieses iteriert durch den tests-Ordner.

Wenn ein Test-script erfolgreich ausgeführt werden konnten, wurde dieser jeweilige Test als erfolgreich verbucht. Als erfolgreich bezeichnet man eine Script-Ausführung immer dann, wenn sie 0 als Exit-Code zurückgibt. Wenn alle Test-scripts erfolgreich ausgeführt werden konnten, wurde dies anhand einer Ausgabe auf der Kommandozeile vermerkt.

In der Abbildung 8 ist das run-tests.sh Script ersichtlich.

In der Abbildung 9 ist die Ausgabe auf der Kommandozeile ersichtlich, wenn alle Tests erfolgreich ausgeführt wurden.

Damit man in den Test-scripts nicht bei sämtlichen unvorhergesehenen Ereignissen manuell ein Exit 1 einbauen musste, hat man in den besagten Test-scripts den Befehl set – euo pipefail verwendet.

Das sorgt dafür, dass Bei jedem Fehler, der in dem Script auftritt, das Script direkt beendet wird.

Eine weitere hilfreiche Funktionalität, welche in den Testscripts Verwendung fand, war der source Befehl. Damit kann man im aktuellen Kontext ein anderes Script ausführen.

So kann man Variablen der «externen Scripts» im Test-script verwenden und manipulieren.

```
#!/usr/bin/env bash
set -euo pipefail
cd "$(dirname "$0")"

failed=0
for t in test_*.sh; do
    printf '==> %s\n' "$t"
    if ! bash "$t"; then
        echo "FAIL"
        failed=1
    fi
done
[[ $failed -eq 0 ]] && echo "All tests passed"
exit $failed
```

Abbildung 8: run-tests.sh

Bei einer Erfolgreichen Ausführung der Tests erschien dieses Resultat auf dem Terminal:

```
$ bash run-tests.sh
==> test_alphabet.sh
==> test_block_multiple_letters.sh
==> test_drawings.sh
==> test_endings_order.sh
==> test_mask_word.sh
==> test_mistake_counter.sh
==> test_startup.sh
All tests passed
```

Abbildung 9: All Tests passed

## Testfälle

In diesem Abschnitt sind sämtliche Testfälle, welche entweder mit Unit-Tests oder manuell ausgeführt werden, aufgeführt.

Nummer	Beschreibung	Erwartet	Manuell / automatisch	Bei automatischen Tests: Dazugehöriger unit-test	optional
TC001	Wird das Alphabet beim Starten des Scriptes richtig dargestellt?	Korrekte Darstellung gemäss Abbildung 6	Automatisch	Test_alphabet.sh	Nein
TC002 <sup>1</sup>	Kann der Benutzer keine Manipulationen der Darstellung vornehmen?	Benutzer kann die Darstellung nicht manipulieren	Manuell	-	Ja
TC003	Wird die Figur korrekt dargestellt?	Korrekt Darstellung gemäss Abbildung 6	Automatisch	Test_drawings.sh	Nein
TC004	Wird das Wort korrekt dargestellt?	Bei bereits korrekt geratenen Buchstaben erscheinen diese an der Stelle. An den anderen Stellen sind weiterhin die Platzhalter (-).	Automatisch	Test_mask_word.sh	Nein
TC005	Wird der Startbildschirm korrekt dargestellt?	Es sollte «Willkommen zu Hangman» Und «Das Wort hat x Buchstaben» stehen.	automatisch	Test_startup.sh	Nein

---

<sup>1</sup> Der Testfall zur Manipulationssicherheit wurde aufgenommen, obwohl keine explizite Anforderung dazu bestand. Aus diesem Grund wird dieser als optional klassifiziert.

TC006	Werden gültige und ungültige Eingaben korrekt erkannt und weiterverarbeitet?	Eingaben werden überprüft, ob sie im Wort vorkommen. Wenn eine Eingabe schonmal geraten wurde, sollte ein Hinweis kommen. Es sollten Eingaben gemacht werden, die keine Buchstaben sind, diese sollten mit einer Fehlermeldung blockiert werden.	Manuell	-	Nein
TC007	Stimmt die Reihenfolge der motivierenden Sätze?	Reihenfolge gemäss Pflichtenheft	Automatisch	Test_endings_order.sh	Nein
TC008	Werden wiederholte Eingaben abgefangen?	Wiederholte Eingaben sollten unterbunden werden.	Automatisch	Test_block_multiple_tries.sh	Nein
TC009	Werden die Fehlerversuche korrekt gezählt?	Fehler sollten korrekt gezählt werden. Wenn mehrmals derselbe Fehler gemacht wird, sollte sich der Fehler-Zähler nicht weiter erhöhen.	automatisch	Test_mistake_count.sh	Nein

Tabelle 4: Testfälle

## Testdrehbuch

Testfall	Bezeichnung	Beschreibung	Testschritte
TC001	Alphabet-check	automatisch	<ol style="list-style-type: none"> <li>1. Funktion, welche für Darstellung des ABC verantwortlich ist, wird aufgerufen.</li> <li>2. Es wird überprüft, ob das ABC auf drei Zeilen dargestellt ist.</li> <li>3. Es wird geprüft, ob das auch zutrifft, wenn bereits einige Buchstaben geraten wurden.</li> </ol>
TC002	Antimanipulations-check	manuell	<ol style="list-style-type: none"> <li>1. Script starten</li> <li>2. Prüfen, ob man die Darstellung durch Eingeben von Buchstaben verunstalten kann.</li> </ol>
TC003	Darstellungs-check	automatisch	<ol style="list-style-type: none"> <li>1. Funktion, die für die Darstellung verantwortlich ist, wird aufgerufen.</li> <li>2. Es wird geprüft, ob die Formen den Erwartungen entsprechen.</li> </ol>
TC004	Wort-Darstellung	automatisch	<ol style="list-style-type: none"> <li>1. Funktion, die für die Wortdarstellung verantwortlich ist, wird aufgerufen.</li> <li>2. Prüfen, ob die Wörter gemäss Erwartungen dargestellt werden.</li> <li>3. Prüfen, ob die Maskierung (-) von unbekannten Buchstaben funktioniert.</li> </ol>
TC005	Start-Bildschirm	automatisiert	<ol style="list-style-type: none"> <li>1. Display_startup Funktion wird aufgerufen.</li> <li>2. Es wird geprüft, ob die gewünschten Ausgaben</li> </ol>

			zurückgegeben werden.
TC006	Verarbeitung der Benutzereingaben	Manuell	<ol style="list-style-type: none"> <li>1. Script starten.</li> <li>2. Prüfen, ob die eingegebenen Buchstaben korrekt angezeigt werden.</li> <li>3. Sind es die richtigen?</li> <li>4. Stimmt der Umgang mit Fehlern? → Werden diese rot markiert?</li> <li>5. Werden richtige Eingaben grün markiert?</li> </ol>
TC007	Reihenfolge der Sätze	automatisch	<ol style="list-style-type: none"> <li>1. Die Funktion, welche für die Sätze nach Abschluss des Spieles verantwortlich ist, wird mit unterschiedlichen Anzahlen von Fehlern aufgerufen.</li> <li>2. Entsprechen die Sätze den Erwartungen?</li> </ol>
TC008	Blockieren wiederholter Fehler	Automatisch	<ol style="list-style-type: none"> <li>1. Die Funktion, die für die Buchstabenprüfung zuständig ist, wird aufgerufen.</li> <li>2. Es werden mehrmals die selben Buchstaben gegeben.</li> <li>3. Werden diese in den arrays guessed_ok oder guessed_bad hineingeschrieben?</li> </ol>
TC009	Fehlerversuche-Zählung	automatisch	<ol style="list-style-type: none"> <li>1. Die Funktion, die Fehler zählt, wird aufgerufen.</li> <li>2. Fehlerversuche werden durch falsche Daten simuliert.</li> <li>3. Stimmt die Anzahl der Fehlerversuche in der entsprechenden Variabel?</li> </ol>

Tabelle 5: Testdrehbuch

## Testprotokoll

In dem nachfolgenden Protokoll ist ersichtlich, wie die Testfälle, welche vorhergehend beschrieben wurden, ausgefallen sind.

Was eventuell auffällt ist, dass jeder Testfall drei Mal aufgelistet ist.

Dies war eine bewusste Entscheidung, damit man Tests, die beim ersten Versuch nicht gelingen zu einem späteren Zeitpunkt, nach Verbesserungen noch weiter prüfen konnte. Des Weiteren sind diese mehrfachen Tests gut, um den Zufälligkeitsfaktor zu reduzieren.

Testfall	Durchlauf	Ergebnis	erfüllt / nicht erfüllt	Datum
TC001	1	Die Anzahl der Zeilen des ABC betrug erwartungsgemäss 3.	Erfüllt <input checked="" type="checkbox"/>	26.05.2025
TC001	2	Die Anzahl der Zeilen des ABC betrug erwartungsgemäss 3.	Erfüllt <input checked="" type="checkbox"/>	05.06.2025
TC001	3	Die Anzahl der Zeilen des ABC betrug erwartungsgemäss 3.	Erfüllt <input checked="" type="checkbox"/>	05.06.2025
TC002 <sup>2</sup>	1	Der Benutzer kann Manipulationen vornehmen, diese wirken sich allerdings nicht auf den Betrieb oder die Funktionalität des Scriptes aus.	Nicht erfüllt <input type="checkbox"/>	26.05.2025
TC002	2	Der Benutzer kann Manipulationen vornehmen, die sich nicht auf die Funktionalität auswirken.	Nicht erfüllt <input type="checkbox"/>	07.06.2025
TC002	3	Der Benutzer kann Manipulationen	Nicht erfüllt <input type="checkbox"/>	07.06.2025

<sup>2</sup> Der Testfall TC002 wurde als nicht erfüllt protokolliert. Da jedoch keine konkreten Anforderungen zur Unveränderbarkeit der Darstellung bestanden, wird dieser Test als optional gewertet und hat keinen Einfluss auf die Gesamterfüllung der Projektanforderungen.

		vornehmen, die sich nicht auf die Funktionalität auswirken.		
TC003	1	Die Figur ist gemäss Erwartungen vorhanden.	Erfüllt <input checked="" type="checkbox"/>	29.05.2025
TC003	2	Die Figur ist gemäss Erwartungen vorhanden.	Erfüllt <input checked="" type="checkbox"/>	05.05.2025
TC003	3	Die Figur ist gemäss Erwartungen vorhanden.	Erfüllt <input checked="" type="checkbox"/>	05.06.2025
TC004	1	Die Wörter werden gemäss Erwartungen dargestellt.	Erfüllt <input checked="" type="checkbox"/>	27.05.2025
TC004	2	Die Wörter werden gemäss Erwartungen dargestellt.	Erfüllt <input checked="" type="checkbox"/>	05.06.2025
TC004	3	Die Wörter werden gemäss Erwartungen dargestellt.	Erfüllt <input checked="" type="checkbox"/>	05.06.2025
TC005	1	Der Startbildschirm wird gemäss Erwartung dargestellt	Erfüllt <input checked="" type="checkbox"/>	27.05.2025
TC005	2	Der Startbildschirm wird gemäss Erwartung dargestellt	Erfüllt <input checked="" type="checkbox"/>	05.06.2025
TC005	3	Der Startbildschirm wird gemäss Erwartung dargestellt	Erfüllt <input checked="" type="checkbox"/>	05.06.2025
TC006	1	Die Eingaben werden korrekt verarbeitet. Es gibt keine Fehler.	Erfüllt <input checked="" type="checkbox"/>	29.05.2025
TC006	2	Die Eingaben werden korrekt verarbeitet.	Erfüllt <input checked="" type="checkbox"/>	05.06.2025

		Es gibt keine Fehler.		
TC006	3	Die Eingaben werden korrekt verarbeitet. Es gibt keine Fehler.	Erfüllt <input checked="" type="checkbox"/>	05.06.2025
TC007	1	Die Reihenfolge der Sätze stimmt mit den Erwartungen aus dem Pflichtenheft überein.	Erfüllt. <input checked="" type="checkbox"/>	31.05.2025
TC007	2	Die Reihenfolge der Sätze stimmt mit den Erwartungen aus dem Pflichtenheft überein.	Erfüllt <input checked="" type="checkbox"/>	05.06.2025
TC007	3	Die Reihenfolge der Sätze stimmt mit den Erwartungen aus dem Pflichtenheft überein.	Erfüllt <input checked="" type="checkbox"/>	05.06.2025
TC008	1	In einem ersten manuellen Test konnte festgestellt werden, dass die erwartete Funktionalität gewährleistet werden kann. Später wird aber noch ein automatisierter Test entwickelt.	Erfüllt. <input checked="" type="checkbox"/>	01.06.2025
TC008	2	Der Test konnte automatisiert werden. Dabei konnte festgestellt werden, dass Fehlversuche korrekt abgeblockt werden. Die Arrays werden nicht weiter befüllt	Erfüllt <input checked="" type="checkbox"/>	04.06.2025

		bei Wiederholungen. Zusätzlich ist eine Fehlermeldung ersichtlich.		
TC008	3	Fehlversuche werden korrekt abgeblockt.	Erfüllt <input checked="" type="checkbox"/>	05.06.2025
TC009	1	Fehler-Count wird richtig geführt. Fraglich momentan ist, ob der Fehler-Count sich erhöhen sollte, bei mehrmals demselben Fehler. Momentan wird bei jedem Fehler der Fehler-Zähler inkrementiert.	Erfüllt mit Anmerkung: Das Verhalten Bei mehrfach auftretenden, identischen Fehlern muss genauer definiert werden. <input checked="" type="checkbox"/>	29.05.2025
TC009	2	Wir haben entschieden, dass die Erwartung an diesen Testfall ist, dass der Fehler-Zähler nur dann inkrementiert wird, wenn ein neuer Fehler vorfällt. Nach Änderungen am Code haben wir den Test erneut durchgeführt und konnten feststellen, dass der die neuen Erwartungen erfüllt sind.	Erfüllt <input checked="" type="checkbox"/>	02.06.2025
TC009	3	Der Test verlief erwartungsgemäss.	Erfüllt <input checked="" type="checkbox"/>	05.06.2025

Tabelle 6: Testprotokoll

## Testbericht

In dem nachfolgenden Testbericht ist beschrieben, wie die vorliegenden Tests, welche dazu eingesetzt wurden, das Hang-Man Projekt auf seine Funktionalität, Benutzerfreundlichkeit und Stabilität zu testen, ausgefallen sind.

Grössenteils verliefen die Tests nach den Erwartungen.

Die meisten Funktionalitäten arbeiteten wie erwartet. Allerdings konnten im Rahmen des Test-Prozesses auch einige Unklarheiten und Fehler aufgedeckt werden.

Beispiele hierfür sind etwa die Unklarheit, ob für wiederholte Fehler der Fehlerzähler weiter hochzählen sollte oder nicht.

Bei dieser Unklarheit mussten wir uns abklären und haben entschieden, dass wir keinen zusätzlichen Fehler werten wollten. Dementsprechend haben wir das auch implementiert und den entsprechenden Test angepasst.

Eine weitere Unklarheit, welche wir durch den Test-Prozess aufdecken konnten, war der Umgang mit Manipulationen der Oberfläche durch Eingaben des Benutzers.

Wir prüften, ob es Massnahmen gibt, die solch ein Verhalten unterbinden konnten.

Diese Idee ist dann allerdings wieder in den Hintergrund gerückt, als wir uns auf andere Aspekte des Projektes konzentrierten.

Damit wir die Anforderungen systematisch testen konnten, setzten wir vor allem auf Unit-Tests. Diese konnten automatisch ausgeführt werden. Das erleichterte den Test-Prozess massgeblich.

Die Unit-Tests prüfen unter anderem die Reaktion auf wiederholte Eingaben oder die Reihenfolge der motivierenden Sätze nach dem Spielende.

Auch manuelle Tests haben in dem Test-Prozess Verwendung gefunden.

Diese waren unter anderem für den bereits erwähnten Test, ob Manipulation der Darstellung vom Spieler möglich sei, verwendet worden.

Dieser Test wurde nicht erfüllt. Daraufhin machten wir uns Gedanken, ob und wie wir uns diesem Problem widmen wollten.

Wir kamen zu dem Schluss, dass wir uns erst um dieses Problem kümmern würden, wenn die sonstige Entwicklung abgeschlossen ist.

## Fazit

Insgesamt verliefen die Tests wie erwartet. Sie halfen uns, das Verhalten des Programms zu verfeinern und machten uns auch auf kleinere Probleme und Unregelmässigkeiten aufmerksam.

Abschliessend lässt sich sagen, dass die vorliegende Applikation nach der Durchführung der Tests stabil läuft und die definierten Funktionalitäten korrekt umsetzt.

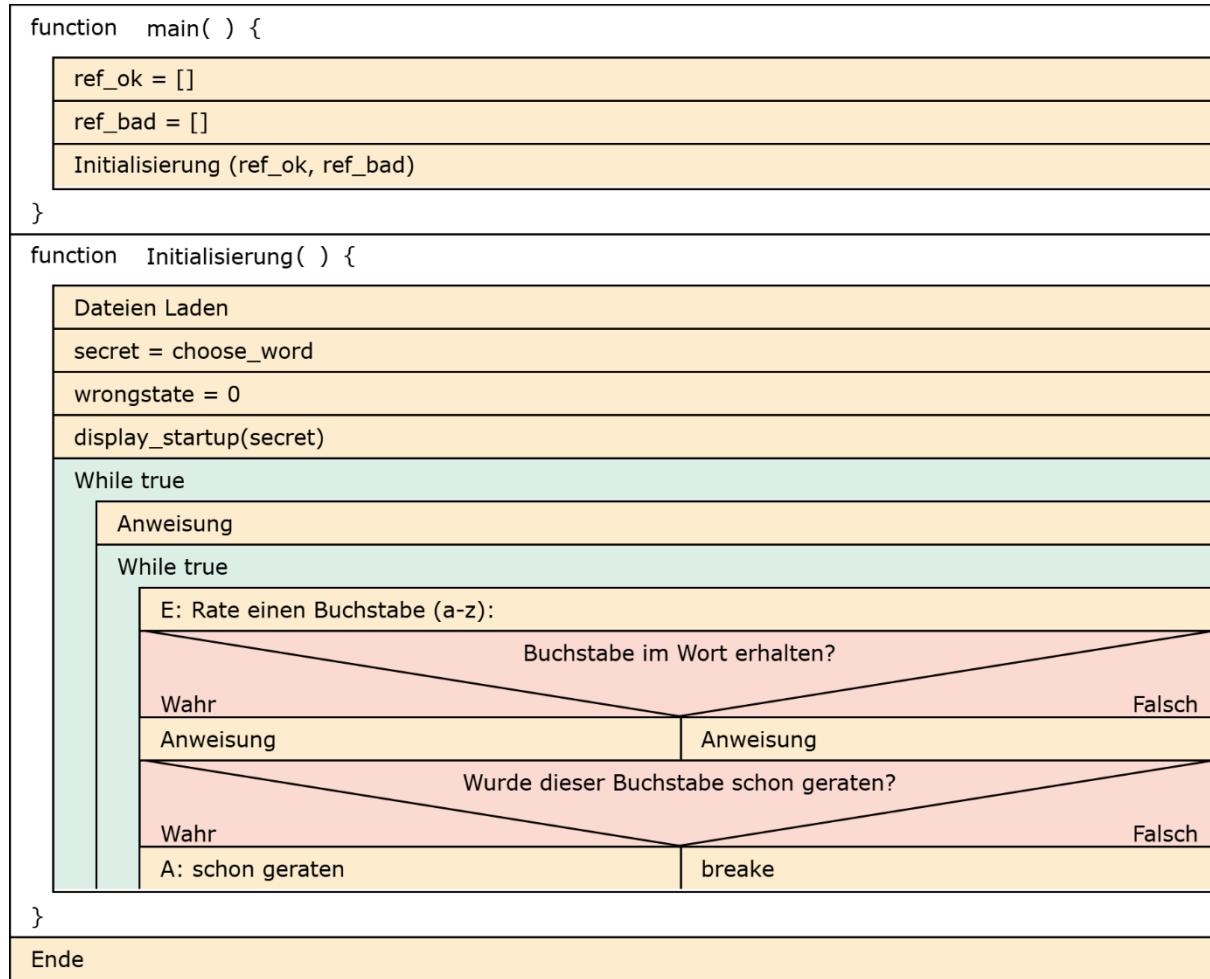
## Funktionsdefinition

Im Folgenden werden die verwendeten Funktionen definiert.

Funktion	Modul	Beschreibung
main	main	Führt <b>Initialisierung</b> aus und <b>lädt Abhängigkeiten</b>
Process_guess	main	Prüft Input und wertet Richtigkeit aus
initialising	main	Zeigt <b>TUI</b> an und verarbeitet Eingaben
alphabet_block	alphabet	Rendert Alphabet Block zur Übersicht der erratenen Buchstaben
display_game_frame	display	Zeigt Ergebnis oder Aktueller Spielstand
endings_order	display	Zeigt das entsprechende Ergebnis an
display_startup	display	Zeigt Willkommens-nachricht
gallows	drawings	Zeigt leeren Mast
wrong 1-6	drawings	Zeigt aktuellen Hangman am Mast
display / mask_word	drawings	Sing utility-functions
mistakesN	endings	Geben Endtexte zurück
init_wordfile	wordlist	Erstellt eine Wörterliste als .txt Dokument
choose_word	wordlist	Wählt ein zufälliges Wort aus

Tabelle 7: Funktionsdefinition

## Struktogramm



## Funktionsbeschrieb

Im Folgenden werden die verwendeten Funktionen beschrieben.

### main

- Ermittelt Absoluten Pfad und lädt dementsprechend die Module
- Ruft die Erstellung eines Zufälligen Wortes auf
- Erstellt Variablen für erfolgreich und fehlerhaft geratene Buchstaben
- Ruft Initialisierungs-Funktion auf
- Rückgabewert: Exit-Status Shell-Script

### processing\_guess

- Überprüft, ob der Buchstaben schon geraten wurde
- Fügt den erratenen Buchstaben den aktuellen Game-State Variablen hinzu
- Eingabe: ok\_guess (array), bad\_guess (array), ltr (char), wrong\_count (int)
- Rückgabewert: Mutiert Arrays & wrong\_count; echo Fehlermeldung bei Duplikat

### Initialising

- Zeigt den aktuellen Game-Frame an
- Liest Eingaben
- Ruft die Verarbeitung des aktuellen erratenen Buchstabens mit grundlegendem Error-Handling zur Erkennung ungültiger Eingaben auf
- Eingabe: wrong (int), ok\_word (array), bad\_word (array), \_secret (string)
- Rückgabewert: Spiel-Loop, ruft andere Funktionen

### alphabet\_block

- Stellt das Alphabet als Block dar, indem alle korrekt erratenen Buchstaben Grün und alle falsch erratenen Buchstaben Rot sind.
- Eingabe: ok\_arr (nameref-Array), bad\_arr (nameref-Array)
- Rückgabewert: Druckt farbigen Alphabet-Block

### display\_game\_frame

- Leert den angezeigten Terminal Inhalt
- Zeigt den Aktuellen Status des Hangmans und des Mastens an.
- Erkennt wenn der Benutzer verloren hat, gibt dementsprechende Nachricht aus und beendet den Prozess mit einem Statuscode 1
- Ruft die Anzeige des Alphabet-Bocks auf
- Zeigt Wort mit den aktuell erratenen Buchstaben an
- Eingabe: ok\_name (array), bad\_name (array), \_wrong (int), \_secret (string)
- Rückgabewert: return 1 wenn Spiel verloren, sonst 0; diverse Prints

## endings\_order

- Beim Gewinnen wird die Gewinnernachricht angezeigt
- Eingabe: wrong (int), mask (string)

## display\_startup

- Liest das Lösungswort aus, um eine Begrüßungs-nachricht mit der Länge des Wortes anzuzeigen
- Wartet 3.3 Sekunden, damit der Benutzer die Nachricht lesen kann
- Eingabe: \_secret (string)
- Rückgabewert: Aus keine Variable, aber gibt Text aus

## gallows

- Zeigt einen Leeren Masten an, der aufgerufen wird, wenn der Benutzer noch keinen Flaschen Buchstaben geraten hat

## wrong 1-6

- Bei jedem Fehler wird in einem Switch Statement eine höhere Funktion aufgerufen
- Die Funktionen geben ein Masten mit dem aktuellen Stand des Hangmans auf der CLI aus.

## display

- Baut das Ausgabewort aus **secret** + Array **guessed**
- Rückgabewert: echo: Maskiertes Wort (z. B. \_ a \_\_)

## mask\_word

- Wie display, nutzt aber **guessed\_ok** (korrekte Buchstaben).
- Rückgabewert: echo: Maskiertes Wort

## mistakesN

- Gibt die Nachricht zurück, die ausgegeben wird, wenn der Nutzer das Spiel gewinnt

## init\_wordfile

- Wenn keine Wortliste vorhanden ist, wird eine Textdatei erstellt, und hardgedeckte Wörter werden in das Dokument geschrieben  
  
Wenn schon eine Wörter Liste vorhanden ist, oder der Benutzer die Wörterliste anpasst, bleibt die Datei unverändert
- Rückgabewert: Erstellt/füllt \$wordfile falls leer

## choose\_word

- Ruft die Erstellung der Wörterliste auf
- Wählt und gibt ein nicht leeres zufälliges Wort aus
- Rückgabewert: echo: zufälliges Wort (string)

## Variabel Beschreibung

In der folgenden Tabelle sind die wichtigsten Variablen, die in diesem Projekt verwendet wurden, definiert und beschrieben.

Wichtig zu erwähnen ist dabei, dass innerhalb der Funktionen häufig Referenzen auf Variablen verwenden werden.

Dadurch kommt es vor, dass es beispielsweise eine Variabel gibt, die `$_secret` heisst.

Der Grund dafür ist, dass Referenzen nicht denselben Namen wie der Ursprung haben dürfen.

Variablename	Zweck	Datentyp	Wird vom Programm manipuliert
<code>\$secret</code>	Speicherung des Geheimwortes	String	Nein
<code>\$wrongstate</code>	Speicherung der Anzahl Fehlversuche	String	Ja
<code> \${guessed_ok[@]}</code>	Speicherung aller richtig geratenen Buchstaben	Array	Ja
<code> \${guessed_bad[@]}</code>	Speicherung aller falsch geratenen Buchstaben	Array	Ja
<code>\$ltr</code>	Speicherung des aktuell geratenen Buchstabens	String	Nein
<code>\$mask</code>	Speicherung des Geheimwort, dargestellt mit Unterstrichen als Platzhalter für nicht geratene Buchstaben	String	Ja
<code>\$ending_output</code>	Speicherung des motivierenden Satzes, welche abhängig der Anzahl Fehler nach erfolgreichem Abschluss des Spiels ausgegeben wird.	String	Nein
<code>\$cleaned</code>	Speicherung des Inhaltes von <code>\$mask</code> ohne Leerzeichen.	String	Ja

## Beispielausgabe

Im untenstehenden Bild ist ersichtlich, wie die Ausführung des Spiels aussehen kann.

Es ist ersichtlich, dass der Benutzer bereits viermal geraten hat, drei Versuche waren dabei erfolglos und werden im Alphabet-Block rot dargestellt.

Der Versuch mit dem Buchstaben «e» war erfolgreich, demzufolge wurde er grün dargestellt im Alphabet-Block.

Es ist auch zu erkennen, dass das erfolgreich geratene «e» an allen Stellen, in denen es im Wort vorkommt, dargestellt wird.

Der Benutzer wird jetzt erneut aufgefordert, einen Buchstaben zu raten, weil das Wort noch nicht erraten ist und die Anzahl der Fehlversuche sechs nicht überschreitet.

```
Wort: _ _ _ _ e _ e
a b c d e f g h i
j k l m n o p q r
s t u v w x y z
Rate einen Buchstaben (a-z):
```

Abbildung 10: Beispielausgabe

## Quellcode

Der komplette Quellcode im Projekt-Repository abgelegt.

Dieses ist unter dem folgenden Link erreichbar:

<https://github.com/KnollElias/bash-hangman>

Der Quellcode ist in mehrere Module unterteilt.

Das führt zu einer besseren Übersichtlichkeit und Verständlichkeit.

Des Weiteren führt diese Massnahme zu einer verbesserten Testbarkeit.

## KVP

Im Rahmen des kontinuierlichen Verbesserungsprozesses sind nachfolgend einige Punkte aufgelistet, die in Zukunft noch verbessert werden könnten.

## GUI

Durch die Implementierung einer grafischen Benutzeroberfläche könnte man die Benutzererfahrung deutlich steigern.

Eine solche könnte beispielsweise durch ein Tool wie Dialog realisiert werden.

Dialog ist nicht nativ in Linux enthalten, sondern muss zuerst mit dem Package-manager installiert werden.

## Grössere Auswahl der Wörter

Eine grössere Auswahl der Wörter würde ebenfalls eine verbesserte user-experience zur Folge haben.

Um das zu realisieren, könnte man grössere Wörterlisten einbauen. Beispielsweise steht dabei die Wörterliste der deutschen Sprache zur Verfügung. Diese umfasst sämtliche 600'000 Wörter der deutschen Sprache. Gegen diese Liste spricht allerdings, dass in dieser Liste teilweise sehr unbekannte Wörter enthalten sind.

Alternativ besteht die Möglichkeit, eine KI-generierte Liste zu verwenden.

Ein mögliches Problem einer solchen könnte sein, dass nicht existente Wörter verwendet werden.

## Erstellen von .EXE-Datei

An dieser Massnahme haben wir schon praktisch gearbeitet; allerdings ist diese wieder in den Hintergrund gerückt, als wir eine Priorisierung der Aufgaben vornahmen.

Es wäre möglich, in der Pipeline durch ein Plugin automatisiert aus dem push eine .exe Datei zu erstellen.

Diese kann dann als Artefakt zur Verfügung gestellt werden.

Der Vorteil einer solchen Datei ist, dass sie einfacher in Windows-Umgebungen ausgeführt werden kann als ein .sh-Datei, welche immer eine Unix-Shell benötigt.

## Fazit

Abschliessend lässt sich sagen, dass das vorliegende Projekt erfolgreich umgesetzt werden konnte.

Wir konnten das Hauptziel, selbstständig und testgetrieben ein Projekt in bash zu entwickeln, erreichen.

In diesem Projekt konnten wir die Themen, welche wir im Unterricht bereits eher theoretisch erlernten, erstmals in einem grösseren selbstständigen Projekt umsetzen. Das macht dieses Projekt zu einer sehr wertvollen und lehrreichen Erfahrung. Wir lernten dabei viel über die praktische Arbeit mit Bash-Scripten.

Eine grössere Herausforderung war das automatisierte Testing.

Besonders das Erstellen von komplexen Scripts für die Sicherstellung der Funktionalität war teilweise sehr anspruchsvoll, jedoch denken wir, dass wir dabei viel lernen konnten.

Während dieses Projektes konnten wir wertvolle Erfahrungen in der Teamarbeit sammeln. Wir nahmen schon früh im Projekt eine grobe Aufgaben-Verteilung vor. Das half uns dabei, Unklarheiten zu vermindern.

Während der Arbeit an dem Projekt besprachen wir regelmässig, welche Aufgaben anstehen und ob Probleme auftauchten.

Wir konnten bei diesem Projekt sehen, wie ein Projekt mit mehreren Leuten organisiert werden kann.

Auch die Einbindung der Continuous-Integration Pipeline war eine neue Herausforderung, an welche wir uns in diesem Projekt heranwagten.

Die Pipeline vereinfachte insbesondere das automatisierte Testen mittels unit-tests deutlich. Ansonsten setzten wir diese auch für das automatisierte Formatieren von Quellcode-Dateien ein.

Allerdings war es am Anfang ein gewisser Knackpunkt, bis die entsprechenden Dateien richtig konfiguriert waren und fehlerfrei funktionierten.