Software Verification Plan

Übersicht

Projekt: Projekt Episko
Inkrement: 8
Autor: Ben Oeckl
Datum: 14.02.2025
Zuletzt geändert:
von: Ben Oeckl
am: 14.02.2025
Version: 1
Prüfer:
Letzte Freigabe:
durch:
am:

Changelog

Datum	Verfasser	Kurzbeschreibung
14.02.2025	Ben Oeckl	Initiales Erstellen und Verfassen
30.03.2025	Simon Blum	Anpassung an Projekt

Distribution List

- Simon Blum simon21.blum@gmail.com
- Ben Oeckl ben@oeckl.com
- Maximilian Rodler maximilianreinerrodler@gmail.com
- Paul Stöckle paul.stoeckle@t-online.de

Software Verification Plan

Einleitung und Zielsetzung

Die Verifikation der Software ist notwendig, um eine zuverlässige Funktion in allen Anforderungsbereichen Sicherzustellen und Fehler zu minimieren um Kosten (in Form von Aufwand) gering zu halten.

Im Kontext des Projektes existieren keine kritischen sicherheitsbedingten Anforderungen. Primär geht es beim Testen und Verifizieren dabei, eine reibungslose und zuverlässige Benutzerfahrung zu garantieren.

Anwendungsbereich

Getestet werden sollen alle Funktionen der Anwendung. Darunter fällt die Erstellung, Betrachtung und Verwaltung von Softwareprojekten anhand eigens dafür eingeführten Manifest-Dateien.

Verifikationsstrategie

Zur Verifikation der Software wird auf verschiedenen Ansätzen gebaut.

Unittests Um das Verhalten einzelner Funktionen im Code zu überprüfen werden Unittests verfasst. Vorhandene Unittests werden im Rahmen unserer CI/CD Pipeline automatisch ausgeführt und verhindern somit das Hinzufügen von fehlerhaftem Code auf dem `main` Branch der Anwendung.

Test Umgebungen: - *(automatisiert)* GitHub Actions Umgebung | `Ubuntu 24.04` Betriebssystem - *(manuell)* x86_64 Rechner | NixOS / Debian / Mint

Ziel Metriken: - **C0** (Region) Coverage: **60%** - **C1** (Branch) Coverage: **60%**

Die genannten Prozentwerte wurden so gewählt, dass eine flächendeckende Verifikation der Anwendung sichergestellt werden kann, sie aber im Kontext des Zeitrahmens trotzdem erreichbar sind.

Hierbei ist zudem anzumerken, dass Metriken für Unittests sich primär auf das Backend, somit den Rust Code beziehen, da sich hier die gesamte Logik der Anwendung befindet.

Automatisierte Last- und Integrationstests Um das Gesamtverhalten der Anwendung, beziehungsweise der einzelnen Module zu verifizieren werden Last- und Integrationstests erstellt.

Testfall-Definition: Im Rahmen des Testes sollen 2000 Metadaten Objekte erstellt, ihre Existenz verifiziert und diese abschließend wieder gelesen werden.

Umgebungen: Durch die modulare Architektur, welche das Backends in mehrere Crates aufteilt gibt es 3 verschiedene Umgebungen wo dieser Test durchgeführt wird.

1. Als Integrationstests im Bezug auf `episko_lib`

- Hierbei wird in dem Test direkt die Bibliothek des Projektes verwendet, jedoch wird dieser als seperate Crate kompiliert und ausgeführt

2. Als Integrationstest im Bezug auf `episko_gui_backend`

- Hierbei werden in dem Test die Schnittstellen für das Frontend verwendet. Somit stellt dieser einen Platzhalter für jenes dar.

3. Seperat von jeglichem Rust Code durch Aufruf der `episko_cli`

- Hierbei wird durch ein seperates Programm das Command Line Interface wiederholt aufgerufen um geforderten Projekte zu erstellen.

Auch diese Tests werden im Zuge der CI/CD Pipeline automatisert ausgeführt.

Manuelle Last- und Integrationstests Um das Verhalten der GUI Anwendung im oben gennanten Testfall zu prüfen, wird die geforderte Anzahl an Projekten generiert.

Folgend wird geprüft, wie die Anwendung sich in diversen Situation verhält. Diese beinhalten:

- Start der Anwendung
- Ansicht Gesamtprojektübersicht
- Ansicht Projektdetails
- Ansicht Statistiken
- Erstellen neuer Projekte
- Bearbeiten existierender Projekte

Hierbei ist jedoch zu beachten, dass aufgrund systemspezifischer Leistungsunterschiede nur subjektive Urteile getroffen werden können und dass zu ziehende Folgen auch aufgrund der anomal hohen Anzahl vorhandener Projekte jeweils seperat abgewogen werden müssen.

Rollen & Verantwortlichkeiten

- Unit-Tests werden von den Entwicklern des betroffenen Codes erstellt.
- Automatiserte Last- und Integrationstests werden von der zu Beginn hierfür bestimmtem Person erstellt.
- Manuelle Tests werden vor dem `Mergen` eines Branches von für diesen verantwortlichen Person durchgeführt.

Zeitplan & Meilensteine

- Unit-Test werden während den Entwicklungsphasen erstellt.
- Unit-Tests werden beim Bauen automatisch ausgeführt (CI-Pipeline)
- Oberflächentests werden während der Entwicklung und an der fertigen Anwendung durchgeführt.
- Lasttests werden durchgeführt wenn alle, zu einem Ablauf gehörenden, Funktionen implementiert sind.

Dokumentation & Nachverfolgbarkeit

- Auffälligkeiten bei Oberflächentests werden bei Reviews dokumentiert.
- Unit-Tests werden durch die Implementierung selbst Dokumentiert.
- Die Ergebnisse von Lasttests werden Dokumentiert.

Testdurchführung

Anbei finden sich Anleitungen zum ausführen einzelner Testgruppen.

Unittests und automatisierte Last-/Integrationstests Umgebung 1+2
Sowohl Unittests als auch die automatiserten Lasttests definiert für Umgebung 1 und 2 können gemeinsam über Rust/Cargo ausgeführt werden.

Hierbei wird die Verwendung von `cargo-nextest` empfohlen, jedoch kann auch `cargo test` verwendet werden. Hierbei werden einige Tests der `episko_cli` hier übersprungen. Mehr Infos diesbezüglich sind an den relevanten Stellen im Quellcode dokumentiert.

```
# Im root Verzeichnis des Projektes
cargo nextest run --workspace
# Alternativ
cargo test --workspace
```

Generierung Coverage Report Zur Einsicht von C0 und C1 Coverage kann mithilfe von `cargo-llvm-cov` ein Bericht erstellt werden.

```
# Generierung eines Berichts mit Konsolenausgabe
cargo +nightly llvm-cov --all-features --workspace --branch

# (Alternativ) Generierung eines html basierten Berichts
cargo +nightly llvm-cov --all-features --workspace --branch --html

# Öffnen des Berichts (beispielhaft mit Firefox)
firefox target/llvm-cov/html/index.html
```

Automatisierter Last-/Integrationstest Umgebung 3 Der dritte automatisierte Test der Kategorie verwendet die Programmiersprache `go` zur Generation zufälliger Daten und Aufrufen der `episko_cli` Anwendung.

```
go run util/data-generation/main.go -test -count 2000 #INOP
```

Manuelle Last-/Integrationstest Zur Durchführung der manuellen Tests kann die gewünchte Anzahl der Daten auch mithilfe des zuvor genannten `go` Programms generiert werden. Diese werden automatich zum Cache hinzugefügt und mittels der Config bei Start automatisch geladen.

```
go run util/data-generation/main.go -base util/test-data -count 2000
```