Testkonzept

Version 1.0

Hauke Preisig

Dokumentenhistorie

Datum	Version	Beschreibung	Autor
15.04.2021	Version 1.0	Erstellung Testkonzept	Hauke Preisig

Inhaltsangabe

Testkonzept

Einführung

Das Testkonzept für die Entwicklung von dem Multi-User-Dungeon „MUD CAKE“ wird im Folgenden beschrieben. Zunächst wird kurz auf die verwendeten Technologien des Produktes eingegangen. Danach wird festgelegt welche Funktionen voraussichtlich durch Tests abgedeckt werden und welche nicht. Um die Funktionen und Eigenschaften möglichst effizient zu testen werden verschiedene Testarten verwendet. Zum einen die Funktionstests, welche sich wieder in Unit Tests, Integrationstests und UI-Tests untergliedern, aber alle die Funktion der Anwendung überprüfen. Neben diesen gibt es die Performancetests, welche die Performanz der Anwendung überprüfen.

Testgegenstand

Die Anwendung teilt sich in Frontend, Backend und eine Datenbankanbindung auf. Für das Frontend wird Angular benutzt. Angular folgt dem Prinzip eines MVVM Design Pattern und Komponenten, die mit Angular erstellt werden, können aufgrund eines eingebauten Unittests Modul einfach getestet werden.
Das Backend besteht aus einem Python Anwendungsserver, hier werden die Tests mithilfe des Python-Testframeworks „PyUnit“ erstellt.

Zu testende Eigenschaften

Funktionstests:
/TF10/ Geschäftsprozess: Dungeons erstellen
/TF20/ Geschäftsprozess: Map bearbeiten
/TF30/ Geschäftsprozess: Räume konfigurieren
/TF40/ Geschäftsprozess: Item anlegen
/TF50/ Geschäftsprozess: NPCs anlegen
/TF60/ Geschäftsprozess: Rassen erstellen
/TF70/ Geschäftsprozess: Klassen erstellen
/TF80/ Geschäftsprozess: Dungeon speichern und laden
/TF90/ Geschäftsprozess: Dungeon kopieren
/TF100/ Geschäftsprozess: Dungeon/Spiel löschen
/TF110/ Geschäftsprozess: Charakterkonfiguration
/TF120/ Geschäftsprozess: Aktionen ausführen
/TF130/ Geschäftsprozess: Aktionen beantworten
/TF140/ Geschäftsprozess: Inventar eines Spielers
/TF150/ Geschäftsprozess: Charakter Lebenspunkte fallen auf null
/TF160/ Geschäftsprozess: Dungeon filtern

/TF170/ Geschäftsprozess: Beitritt der Spieler verwalten
/TF180/ Geschäftsprozess: Erstmaliges Beitreten eines Spiels
/TF290/ Geschäftsprozess: Spiel verlassen.
/TF200/ Geschäftsprozess: Wiedereintritt als Spieler in Spiel
/TF210/ Geschäftsprozess: Dungeon Master verlässt das Spiel
/TF220/ Geschäftsprozess: Wiedereintritt in Spiel als Dungeon Master
/TF230/ Geschäftsprozess: Chatten als Spieler
/TF240/ Geschäftsprozess: Chatten als Dungeon Master
/TF250/ Geschäftsprozess: Map aufdecken
/TF260/ Geschäftsprozess: Responsive Design
/TF270/ Geschäftsprozess: Exceptionhandling bei Fehlerhaften Eingaben

Performance Tests:
/TF280/ Geschäftsprozess: Reaktion auf Benutzeraktion unter 4s
/TF290/ Geschäftsprozess: Chat Übertragung unter 3s

Nicht zu testende Eigenschaften

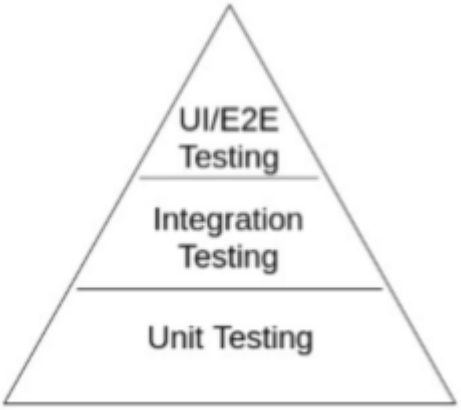
/TF300/ Geschäftsprozess: Account Registrierung
/TF310/ Geschäftsprozess: Account Einloggen
/TF320/ Geschäftsprozess: Account Zurücksetzen
/TF330/ Geschäftsprozess: Account löschen
/TF340/ Geschäftsprozess: Dungeon veröffentlichen
/TF350/ Geschäftsprozess: Reaktionen der Oberfläche innerhalb von 3 Sekunden
/TF360/ Geschäftsprozess: Benutzer können maximal 3 Dungeons gleichzeitig speichern
/TF370/ Geschäftsprozess: maximale Konfigurationsebene unter 10 Klicks

Teststrategien

Um möglichst effizient viele Fehler frühzeitig zu Entdecken und da es unmöglich ist den ganzen Code mit Tests abzudecken wird versucht das Risiko eines unentdeckten Fehlers zu minimieren, indem zuerst die Fehleranfälligen Code-Abschnitte getestet werden. Die zu testenden Eigenschaften werden nach ihrer Relevanz in der Qualitätssicherung ausgewählt und abgearbeitet.
Der Fokus in unserem Testkonzept liegt auf den Funktionstest. Neben diesen werden auch wenige Performance Tests angelegt. Diese werden hauptsächlich Laufzeit Tests sein, welche sicherstellen, dass das Spiel benutzerfreundlich und ansprechend ist.
Stress und Speichertests werden voraussichtlich vernachlässigt.

Funktionstests

Ein Funktionstest prüft eine Funktionseinheit gegen deren funktionale Anforderungen. Dies wird sowohl durch das Sollverhalten im positiven Sinne geprüft, als auch durch das Nachweisen, dass es auch bei fehlerhaften Eingaben nicht zu inakzeptablen Reaktionen kommen kann. Funktionstests gibt es auf unterschiedlichen Ebenen, die unterste Ebene bilden die Unit-Tests, welche die Funktionalitäten von einzelnen Methoden oder Klassen prüfen. Die mittlere Ebene bilden die Integrationstests, diese prüfen die Schnittstellen von Modulen und deren Interaktionen, auf Fehlverhalten. Im Idealfall haben die verschiedenen Module Unit Tests, die deren interne Funktionalität sicherstellen.



Die oberste ebene bilden die UI-Tests. Diese überprüfen, wie die Anwendung mit Benutzeraktionen umgeht, die über Maus und Tastatur ausgeführt werden, sowie ob die visuellen Elemente korrekt angezeigt werden.

Unit Tests:
Testplanung:
Ein Entwickler muss nachdem er ein Produktfunktion implementiert hat planen, welche Methoden wie getestet werden müssen, damit eine möglichst große Codeabdeckung gewährleistet ist. Beim Schreiben von Unit Tests muss der Entwickler die Testprinzipien von „FIRST“ (Fast, Independent, Repeatable, Self-validating, Timely) einhalten
Testspezifikation:
Die Testfallspezifikation dokumentiert die zu benutzenden Eingabewerte und erwarteten Ausgabewerte. Dies wird bei Unit Tests im Namen der Testmethoden wie folgt festgehalten:
„NameDerMethode_InputReturnsErwartetesVerhalten“
Testdurchführung:
Alle Unit Tests werden automatisch bei jedem Commit ausgeführt.

Testprotokollierung:
Die Ergebnisse von allen Tests werden nach jedem Commit auf einem Bitbucket Account protokolliert.
Testauswertung:
Alle Unit Tests werden automatisch nach jedem Commit ausgewertet.
Testende:
Nachdem alle Unit Test ausgewertet wurden ist der Testprozess abgeschlossen. Wenn alle Tests erfolgreich waren wird der Commit mit dem Main Branch gemerched. Sobald ein Test fehlschlägt muss der Entwickler entweder seinen Code anpassen oder die Tests aktualisieren.

Integrationstests:
Testplanung:
Bei Integrationstest wird die „testzielorientierte Strategie“ mit der der Top-Down-Methode kombiniert.
Testspezifikation:
Die Testfallspezifikation dokumentiert die zu benutzenden Eingabewerte und erwarteten Ausgabewerte. Dies wird bei UI Tests im Namen der Testmethoden wie folgt festgehalten:
„NameDerMethode_InputReturnsErwartetesVerhalten“
Testprotokollierung:
Die Ergebnisse von allen Tests werden nach jedem Commit auf einem Bitbucket Account protokolliert.
Testauswertung:
Alle Integrationstests werden automatisch nach jedem Commit ausgewertet.
Testende:
Nachdem alle Integrationstest ausgewertet wurden ist der Testprozess abgeschlossen. Wenn alle Tests erfolgreich waren wird der Commit mit dem Main Branch gemerched. Sobald ein Test fehlschlägt muss der Entwickler entweder seinen Code anpassen oder die Tests aktualisieren, falls er keins von beidem kann muss er einen neuen Bug für den fehlgeschlagenen Test anlegen.

UI-Tests:
Testplanung:
Für jeden UI-Test wird eine feste Reihenfolge an Benutzereingaben festgelegt und das Sollverhalten der Anwendung definiert.
Testprotokollierung:
Der Tester vergleicht das in der Testplanung festgelegte Sollverhalten mit dem tatsächlichen Verhalten der Anwendung und notiert gegebenenfalls Abweichungen.
Testauswertung:
Wenn das Ist- und Sollverhalten nicht übereinstimmen ist der Test fehlgeschlagen.
Testende:
Der Test ist abgeschlossen, wenn das Ist- und Sollverhalten übereinstimmen oder bei einem fehlgeschlagenen Test ein neuer Bug angelegt wurde.

Performancetests

Performancetests testen die Geschwindigkeit, Reaktionszeit, Stabilität, Zuverlässigkeit, Skalierbarkeit und Ressourcennutzung der Softwareanwendung unter einer bestimmten Last. Im Rahmen dieses Projekts wird der Focus bei den Performancetests auf der Geschwindigkeit und Reaktionszeit liegen. Deshalb kann man diese ähnlich wie die Unit Tests implementieren.

Testplanung:
Die Testplanung beruht auf den Performance Anforderungen der Anwendung.
Testspezifikation:
Die Testfallspezifikation dokumentiert die zu testenden Anwendungseigenschaften und deren Sollwert. Dies wird bei Performancetests im Namen der Testmethoden wie folgt festgehalten:
„NameDerEigenschaft_Sollwert“
Testdurchführung:
Alle Performance Tests werden automatisch bei jedem Commit ausgeführt.
Testprotokollierung:
Die Ergebnisse von allen Tests werden nach jedem Commit auf einem Bitbucket Account protokolliert.
Testauswertung:
Alle Performancetests werden automatisch nach jedem Commit ausgewertet.
Testende:
Nachdem alle Performancetests ausgewertet wurden ist der Testprozess abgeschlossen. Wenn alle Tests erfolgreich waren wird der Commit mit dem Main Branch gemerched. Sobald ein Test nicht den Anforderungen entspricht muss der Entwickler entweder seinen Code anpassen.

Testumgebungsvoraussetzungen

Bei Jenkins handelt es sich um ein Softwaresystem zur kontinuierlichen Integration von Softwarekomponenten zu einem Anwendungsprogramm. Hierbei werden automatisierte Tests ausgeführt, sobald man in die entsprechende Versionsverwaltung eincheckt. Ziel ist es die Softwarequalität zu steigern. Der Jenkins-Server wurde auf einem RaspberryPi installiert um vom Game- sowie Client-Server unabhängig zu sein.

Maven

Maven ist das Build-Tool, welches der Jenkins-Server zum Bauen der Applikation verwenden kann. Es stellt viele Standards bereit, sodass nur minimale Konfigurationen notwendig sind, um den Lebenszyklus vom Build über die Tests zum Deploy einer Software abzubilden.

Git

Bei Git handelt es sich um ein Versionsverwaltungssystem. Es ist mit dem Jenkins-Server Verbunden damit dieser deployed sobald eine Codeänderung gepushed wurde.

JDK 8/11 (Java Development Kit)

Da Jenkins in Java geschrieben wurde, wird das Java Development Kit 8 oder 11 benötigt. Bei Java handelt es sich um eine objektorientierte Programmiersprache, welche für Jenkins benötigt wird. Die JDK beinhaltet neben dem Java Runtime Environment, den Compiler und viele weiteren Entwicklungstool für Java.

Zuständigkeiten

Generell erfolgt das Testen des Codes nach dem „Code First“-Prinzip. D.h. der Test wird erst geschrieben, nachdem die Komponente entwickelt wurde. Dieses Vorgehen passt besser zum Erfahrungsstand des Teams, da beim Entwickeln einer Komponente oft noch nicht alle integrierten Spezifikationen finalisiert wurden und es somit sehr schwierig ist vorab einen Test zu schreiben für eine Komponente, die noch nicht existiert.

Die Unit-Tests werden vom jeweiligen Code-Erzeuger getestet. Dies hat den Vorteil, dass der Entwickler sich nicht unnötig lange in den Code einarbeiten muss, bevor er einen entsprechenden Test überhaupt erst definieren kann. Bei den Integrationstest kommt noch zusätzlich der Testbeauftragte für eine bessere Koordination der Integrationstest dazu. Die UI-/Systemtest werden auf Grund ihrer geringen Komplexität wöchentlich vom Testbeauftragten durchgeführt.