Testkonzept

HOR-TINF2021

Gruppe

Tobias Hahn
Lars Holweger
Fabian Unger
Timo Zink
Frank Sadoine
Fabian Eilber

Betreuer

Prof. Dr. Phil. Antonius van Hoof

Dokumentverlauf

Version	Beschreibung/Änderung	Autor	Datum
0.5	Erstellung des Dokuments und erster Entwurf	Frank Sadoine	01.04.2023
0.7	Weitere Ausarbeitung des Dokuments	Frank Sadoine	05.04.2023
1.0	Einarbeiten des Feedbacks vom Betreuer	Frank Sadoine	07.04.2023
1.1	Vervollständigung des Dokuments	Timo Zink, Tobias Hahn	09.04.2023

Inhaltsverzeichnis

Dokumentverlauf 1
Inhaltsverzeichnis 1
0. Ziel des Dokuments 2
1. Testarten 2
1.1. Komponenten – Test 3
1.2. Integration – Test 3
1.3. User-Interface – Test 3
1.4. Gebrauchstauglichkeit und Akzeptanz – Test 3
2. Werkzeuge 4
2.1. Selenium 4
2.2. JBehave 4
2.3. JUnit 4
2.4. Jasmine 5
2.5. Code Coverage 5
3. Testkonzept 6
3.1. Allgemein 6
3.2. Komponenten – Tests 7
3.3. Integration – Tests 8
3.4. User-Interface-Tests 9
3.5. Gebrauchstauglichkeit und Akzeptanztest 9

0. Ziel des Dokuments

Ziel des Projektes ist die Entwicklung eines Event-Management-Systems. Zum Erreichen dieses Zieles arbeiten verschiedene Programmierer an verschiedenen Komponenten des Projektes. Bei einem Projekt dieser Größe muss damit gerechnet werden, dass Entwickler, also Menschen, Fehler bei der Implementierung oder Spezifikation machen. Deswegen ist es essenziell die Software während der Entwicklung zu testen, um die Funktionalität zu bewerten. Testing dient dazu festzustellen, ob die entwickelte Software die spezifizierten Anforderungen erfüllt oder nicht. Durch Tests werden Mängel identifiziert und sichergestellt, dass das Produkt einwandfrei funktioniert. Im folgenden Dokument wird das geplante Testkonzept unseres Produkts genauer erläutert.

1. Testarten

- **Black Box Testing**: Bei diesen Tests wird lediglich darauf geachtet was als Input eingegeben wird und was als Output rauskommt. Der innere Aufbau ist dabei nicht bekannt.
- **White Box Testing**: Die internen Funktionalitäten der Software sind bekannt, folglich konkrete Codeeinheiten einer Software, werden getestet

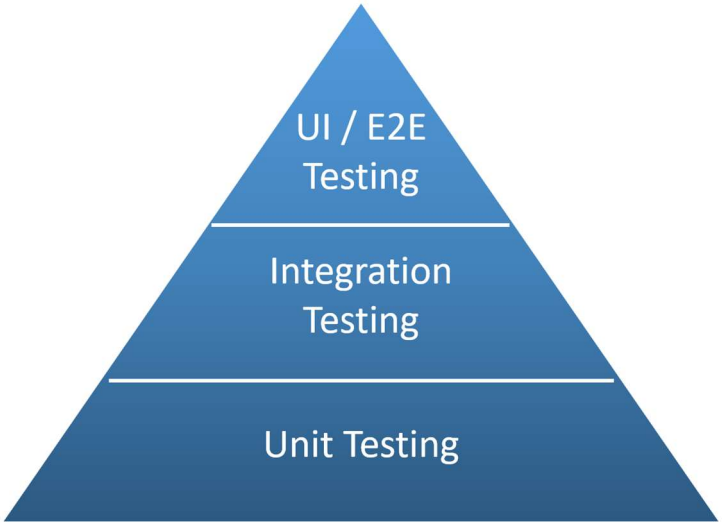

Abbildung 1: Hierarchie der Testarten

In der Abbildung 1 werden die elementaren Testarten dargestellt. Unit Testing ist dem White Box Testing zuzuordnen, während das UI-Testing dem Black Box Testing zugeordnet wird. Integrationstest können beiden zugeordnet werden, sind in der Regel eher White Box Tests. Durch den pyramidenartigen Aufbau lässt sich erkennen das die verschiedenen Testarten aufeinander aufbauen. Im Folgenden werden die verschiedenen Testarten kurz allgemein definiert.

1.1. Komponenten – Test

Komponententests (engl. Unit Test) prüfen, ob die einzelnen Komponenten der Anwendung korrekt funktionieren. Hierbei werden typischerweise die einzelnen Codeeinheiten der Anwendung isoliert getestet, um sicherzustellen, dass sie in sich selbst korrekt funktionieren. Komponenten können Klassen oder einzelne Methoden sein, jedoch nie das gesamte Projekt.

Um Komponententests in das Testkonzept zu integrieren, sollten entsprechende Testfälle entwickelt werden, die auf den einzelnen Codeeinheiten der Anwendung basieren. Diese Testfälle sollten automatisiert und dokumentiert werden, um sicherzustellen, dass die einzelnen Komponenten der Anwendung in sich selbst korrekt funktionieren und so Fehler frühzeitig erkannt werden können.

Komponententests stellen die unterste und somit breiteste Stufe der Pyramide da. Dies bedeutet, dass sie am meisten verwendet werden. Das Ziel ist es eine möglichst hohe Testabdeckung der einzelnen Komponenten zu erreichen.

1.2. Integration – Test

Integrationstests prüfen, ob die verschiedenen Komponenten der Anwendung korrekt miteinander interagieren. Hierbei werden typischerweise Schnittstellen zwischen den verschiedenen Komponenten getestet, um sicherzustellen, dass Daten korrekt ausgetauscht und verarbeitet werden. Integrationstests bieten somit ein umfassendes Testbild des gesamten Projekts. Der Aufwand für Integrationstests ist jedoch hoch, da das gesamte Projekt kompiliert und bei Verwendung einer Datenbank auch diese neu generiert und eingebunden werden muss. Diese Maßnahmen gewährleisten ein neutrales Testergebnis. Die Laufzeit der Tests ist in der Regel länger als bei Komponententests und kann Minuten dauern Integrationstests können automatisiert werden, um den Testprozess zu beschleunigen und zuverlässiger zu machen.

1.3. User-Interface – Test

User-Interface-Tests prüfen, ob die Benutzeroberfläche der Anwendung korrekt funktioniert. Hierbei werden typischerweise die verschiedenen Benutzerinteraktionen getestet, um sicherzustellen, dass die Anwendung auf Benutzereingaben korrekt reagiert. Jeder Klick auf eine Schaltfläche oder ein Element sollte die erwartete Reaktion auslösen, wie beispielsweise die Weiterleitung auf eine neue Seite. Obwohl die Implementierung einzelner Tests schnell umgesetzt werden kann, wird es sehr umfangreich, wenn man möglichst alle Elemente des User-Interface testen möchte. Daher werden nur die wichtigsten Szenarien getestet. Es gibt auch Überschneidungen mit Gebrauchstauglichkeitstests, bei denen der Nutzer das User-Interface testet. Daher muss überlegt werden, wo der Implementierungsaufwand am wichtigsten ist und was manuell vom Nutzer übernommen werden kann.

1.4. Gebrauchstauglichkeit und Akzeptanz – Test

Diese Testarten ordnen sich im oberen Drittel der Pyramide ein. Sie werden manuell durchgeführt.

Bei der **Gebrauchstauglichkeit** (Usability) ist das Entwickler-Team involviert und prüft typischerweise im Rahmen von UI-Tests Aspekte wie konsistentes Design, korrekte Rechtschreibung und Übersichtlichkeit, die in den vorherigen Teststufen nicht automatisiert überprüft werden konnten. Es kann auch vorkommen, dass bestimmte Funktionalitäten und UI-Elemente überprüft werden, was zu Überschneidungen mit UI-Tests führen kann.

Akzeptanztests prüfen, ob die Anwendung die Erwartungen des Kunden erfüllt. Hierbei werden typischerweise Tests durchgeführt, die auf den spezifischen Use Cases des Kunden basieren, um sicherzustellen, dass die Anwendung alle funktionalen und nichtfunktionalen Anforderungen des Kunden erfüllt. Speziell bei agilen Vorgehensweisen („Zusammenarbeit mit dem Kunden mehr als Vertragsverhandlung“ – Agiles Manifest) hat diese Testart eine hohe Wichtigkeit.

2. Werkzeuge

2.1. Selenium

Selenium ist ein Framework für die automatisierte Durchführung von Tests von Web-Anwendungen. Es basiert auf dem Selenium WebDriver und unterstützt alle gängigen Browser wie Chrome, Firefox, Edge, Safari und Opera. Die Selenium-Schnittstelle ermöglicht es, das Verhalten eines Browsers vorzugeben und zu steuern, was es Benutzern ermöglicht, Daten aus einer Web-Anwendung auszulesen, Klicks und Eingaben zu emulieren, sowie die Navigation zu einer bestimmten URL durchzuführen. Durch diese Funktionen können Benutzer testen, ob das User-Interface den Anforderungen entspricht. Selenium ist nicht nur mit JUnit-Testprojekten kompatibel, sondern auch mit anderen Testframeworks. Das Framework ist besonders in agilen Entwicklungsumgebungen beliebt, da es Entwicklern und Testern die Möglichkeit gibt, Feedback in Echtzeit zu erhalten.

2.2. JBehave

JBehave ist ein Open-Source-Framework für die akzeptanzgetriebene Entwicklung von Java-Anwendungen. Es basiert auf dem Konzept von „Stories“ und „Scenarios“ und bietet eine strukturierte Möglichkeit, Tests auf der Ebene der Geschäftsprozesse zu schreiben. JBehave ermöglicht auch die Erstellung von automatisierten Tests auf der Ebene der Benutzeroberfläche und der Backend-Services. Zudem unterstützt JBehave auch die Erstellung von Test-Reports und ermöglicht Entwicklern, die Testergebnisse in einer leicht verständlichen Form zu präsentieren.

2.3. JUnit

JUnit ist ein Open-Source-Framework für die Entwicklung von Unit-Tests in der Programmiersprache Java. Es hilft Entwicklern schnell und einfach automatisierte Tests für ihre Anwendungen zu erstellen und auszuführen. JUnit bietet eine Reihe von Assertion-Methoden, mit denen Entwickler prüfen können, ob bestimmte Bedingungen erfüllt sind oder nicht. Es ermöglicht auch das Erstellen von Testsuiten, um mehrere Tests gleichzeitig auszuführen.

```java
import static org.junit.jupiter.api.Assertions.assertEquals;

import example.util.Calculator;

import org.junit.jupiter.api.Test;

class MyFirstJUnitJupiterTests {

    private final Calculator calculator = new Calculator();

    @Test
    void addition() {
        assertEquals(2, calculator.add(1, 1));
    }

}
```

Abbildung 2: Codebeispiel für JUnit

2.4. Jasmine

Jasmine ist ein Open-Source-Framework für die automatisierte Testentwicklung von JavaScript-Anwendungen. Es bietet eine breite Palette an Funktionen und Methoden zur Erstellung von Tests. Jasmine basiert auf dem Behavior Driven Development Ansatz und ermöglicht Entwicklern, Tests auf eine verständliche und menschenlesbare Weise zu schreiben. Das Framework bietet eine Vielzahl von Assertion-Methoden, die es Entwicklern ermöglichen, die erwarteten Ergebnisse zu definieren und sicherzustellen, dass sie mit den tatsächlichen Ergebnissen übereinstimmen.

```javascript
describe("A suite is just a function", function() {
  let a;

  it("and so is a spec", function() {
    a = true;

    expect(a).toBe(true);
  });
});
```

Abbildung 3: Codebeispiel für Jasmine

2.5. Code Coverage

Die Code Coverage gibt prozentual an, wie viel des Gesamtcodes durch Tests abgedeckt ist. Als Code Coverage Tool wird die integrierte Code Coverage in Intellij verwendet.

3. Testkonzept

In Kapitel 1 Testarten wurden verschiedene Testarten vorgestellt. Dieses Kapitel beschreibt, wie diese Testarten im Projekt umgesetzt werden. Die Gliederung erfolgt in den einzelnen Testarten. Dabei sollen jeweils verschiedenen Punkte definiert werden:

- Wer kontrolliert die Einhaltung des Konzeptes?
- Wer implementiert die Tests?
- Woher kommen die Testdaten?
- Wie findet die Umsetzung statt?
- Was wird getestet?
- Welche vorher definierten Werkzeuge werden benötigt?

3.1. Allgemein

Als allgemeines Konzept wird **Behavior Driven Development (BDD)** verwendet. In diesem Verfahren werden eigentlich bereits in der Anforderungsanalyse Aufgaben, Ziele und Ergebnisse in einer speziellen Textform festgehalten. Dies muss für dieses Projekt erst nachgeholt werden. Dazu wird stets eine besondere Textform verwendet, welche auf „Wenn, Dann“-Sätzen basiert. Es handelt sich um normale Sprache, die eine definierte Form bekommt. Sie kann jedoch weiterhin auch von Nicht-Entwicklern, und somit Kunden, normal gelesen und verstanden werden.

Um dies umzusetzen, wird JBehave verwendet (siehe 2.2 JBehave). In einem JBehave Testprojekt werden schon vor der Implementierung in Gherkin-Sprache wichtige Szenarien definiert, welche getestet werden sollen. Ein Szenario kann aus den Elementen „Gegeben“, „Wenn“, „Dann“ bestehen, wobei diese auch mit „Und“ verkettet werden können. Im Folgenden ein Beispiel.

Szenario: Nutzer meldet sich an

- **Gegeben** ist, dass der Nutzer die Website geöffnet hat
- **Und** er nicht angemeldet ist
- **Wenn** der Nutzer auf Anmelden klickt
- **Und** richtige Zugangsdaten eingibt
- **Dann** ist der Nutzer erfolgreich eingeloggt

Nach der Implementierung muss der konkrete Test implementiert werden. Gegebenheiten und Ziele sind bereits klar definiert. Nun kann der Entwickler in der Schrittdefinition die einzelnen Schritte, also Zeilen, als Test implementieren. Dazu werden dann beispielsweise JUnit oder Selenium verwendet. Für die Erstellung der wichtigsten Testszenarien anhand des Pflichtenheftes ist Tobias Hahn verantwortlich. Im Allgemeinen werden die Tests automatisiert und bei jedem Build ausgeführt.

3.2. Komponenten – Tests

Die Erstellung von Komponenten Tests obliegt dem jeweiligen Entwickler, da dieser das beste Verständnis für die Implementierung und Schnittstellen zu anderen Komponenten hat. Es bedeutet jedoch nicht, dass jede Komponente einen solchen Test benötigt. Generell wird eine möglichst hohe Testabdeckung im Projekt angestrebt, doch es kann vorkommen, dass bestimmte Komponenten nicht testbar sind. Sollte dennoch ein Test in JBehave definiert sein, muss der Entwickler mit dem Testverantwortlichen Frank Sadoine klären, ob die Komponente automatisiert, testbar ist. Falls dies nicht möglich oder nur mit unverhältnismäßigem Aufwand verbunden ist, wird das Szenario dem Gebrauchstauglichkeitstests zugeordnet, die später definiert werden. Es ist keinesfalls zulässig, einen Test einfach zu verwerfen.

Die Umsetzung der Test wird mit dem vorgestellten Werkzeugen JBehave und JUnit realisiert. Eine konkrete Beispielimplementierung ist in Abbildung 4 zu sehen. Der erste Abschnitt stellt die Definiton in JBehave-Form dar.

```
Scenario: User logs in
Given that the user has opened the website
And he is not logged in
When the user clicks on Log In
And enters correct login data
Then the user is successfully logged in
```

Abbildung 4: Definition eines Testes mit JBehave

Der zweite Abschnitt zeigt die Realisierung mithilfe von JUnit auf.

```java
public class UserloginTest {

    no usages
    @Test
    public void userCanLogin() {
        // Arrange
        Website website = new Website();
        User user = new User();
        website.userLogout(user); //

        // Act
        website.loginSiteVisit();
        website.clickLoginButton();
        website.giveuserDate(user, "right Username", "right Password");

        // Assert
        Assert.assertTrue(website.isUserLoggedIn(nuser));
    }
}
```

Abbildung 5: Ausschnitt aus einem JUnit Test

Die Tests werden nach dem Arrange/Act/Assert (AAA) Muster erstellt. Dieses Muster teilt den Testvorgang in drei Schritten auf:

- Im ersten Schritt **Arrange** werden benötigte Objekte initialisiert und Variablen gesetzt. Diese werden im weiteren Verlauf des Tests verwenden.
- Im zweiten Schritt **Act** wird der eigentliche Test durchgeführt. Dabei werden die Methoden mit den zuvor in Arrange definierten Parameter aufgerufen
- Im dritten Schritt **Assert** werden die Ergebnisse der in Act aufgerufenen Methoden ausgewertet. Hier entscheidet sich, insofern es nicht zuvor einen Fehler gab, ob der Test erfolgreich ist oder nicht.

Da im Projekt JBehave verwendet wird, müssen die einzelnen Schritte (Gegeben, Wenn, Dann) als eigene Methoden implementiert werden. Letztendlich zwingt dies dazu, dass das AAA-Muster zu verwenden.

Die Methoden werden gemäß der jeweiligen Zeile eines Szenarios in CamelCase benannt. Bezogen auf das Beispiel im vorherigen Abschnitt würde der Methodenname also "BenutzerKlicktAnmelden" lauten. Da jedoch in Englisch codiert wird, muss die Definition in JBehave auch auf Englisch erfolgen.

Die für einen Test benötigten Daten werden in einer Testdatenbank mit vorgefertigten Datensätzen gehalten. Kontrolliert wird die allgemeine Testabdeckung der Komponenten von Frank Sadoine, dem Verantwortlichen für Tests.

3.3. Integration – Tests

Integrationstests werden, wie auch Komponententests, mithilfe von JUnit implementiert. Aufgrund der in Kapitel 1.2 Integration – Test genannten Gründe gibt es im Vergleich zu anderen Testarten eher wenige Integrationstests, die jedoch eine hohe Testabdeckung haben.

Integrationstests werden nach dem Bottom-Up-Prinzip implementiert, das heißt, sobald die einzelnen Komponententests erfolgreich abgeschlossen wurden, kann ein Test für eine bestimmte Komponentengruppe oder das gesamte Projekt erstellt und durchgeführt werden.

Ein Beispiel-Szenario in unserem Projekt ist die Überprüfung der Verfügbarkeit der Anwendung. Hierbei wird bei der Testausführung eine neue Version mit eigener Datenbank generiert. Bei erfolgreicher Generierung wird der Statuscode der Website abgefragt. Wenn die Rückmeldung der Anwendung in Ordnung ist (HTTP-Statuscode: 200), kann davon ausgegangen werden, dass die Website erreichbar ist und der Benutzer darauf zugreifen kann.

Integrationstests werden aufgrund ihrer geringen Anzahl direkt vom Testbeauftragten erstellt. Gegebenenfalls kann auch ein qualifiziertes Teammitglied mit der Implementierung beauftragt werden. Die Durchführung der Integrationstests erfolgt nach erfolgreicher Durchführung der Komponententests.

Aufgrund der hohen Komplexität erfordert diese Art des Testens eine detaillierte Dokumentation. In dieser soll kurz beschrieben werden, was getestet wird, was damit sichergestellt wird und welche Anforderungen erfüllt werden müssen, um den Test durchführen zu können. Diese Dokumentation kann auch in einem über Microsoft Teams synchronisierten Dokument stichpunktartig erfolgen. Nach Erstellung eines neuen Tests muss dieses stets aktualisiert werden.

3.4. User-Interface-Tests

User-Interface-Tests werden, wenn möglich mit Selenium und Jasmine automatisiert, ansonsten wöchentlich manuell ausgeführt. Dafür ist der Testverantwortliche Frank Sadoine zuständig. Die Ergebnisse werden nur mit ihrem Ausführungszeitraum und eventuellen Fehlern dokumentiert. Es werden alle Knöpfe und Navigationselemente der Benutzeroberfläche getestet, dabei sollte aber kein unrealistischer Zeitaufwand entstehen. Deshalb liegt der Fokus auf Fällen, in dem ein Fehler dem Benutzer bei einem Gebrauchstauglichkeitstests nicht sofort auffliegen würde. Beispielsweise kann getestet werden, ob der Benutzer nach einem Klick auf seinem eigenen Event auch tatsächlich bei seinem Event landet. Für die Implementierung werden zu testende HTML-Elemente eine ID zugewiesen. Diese ID kann dann in der Implementierung der Tests abgefragt werden, um somit die richtigen Elemente zu finden und Aktionen auszulösen.

3.5. Gebrauchstauglichkeit und Akzeptanztest

Akzeptanztests: sind nur optional. Letztendlich wurde mit dem Pflichtenheft ein Vertrag ausgehandelt, nachdem die Anwendung nun entwickelt wird. Merkt der Kunde, dass er vergessen hat weitere Anforderungen zu spezifizieren oder umzuändern, ist es in der Theorie zu spät. Auf Nachfrage kann ihm trotzdem ein Dokument zur Dokumentation der Durchführung eines Akzeptanztestes zugesendet werden. Dieses beinhaltet außerdem einen Leitfaden zum Vorgehen. Da sowieso mindestens einmal pro Woche ein neuer Release auf der

Website erfolgt, kann dieser getestet werden. Sieht der Kunde seiner Ansicht nach falsch oder gar nicht umgesetzt Punkte, kann er diese im Dokument vermerken und diese zurücksenden. So sind die Entwickler in der Lage darauf zu reagieren. Ob dies geschieht, hängt vom jeweiligen Fall ab, da das Grundgerüst mit dem Pflichtenheft fest definiert ist. Ziel ist jedoch eine hohe Kundenzufriedenheit, weshalb die Option besteht.

Sollten vom Kunden Akzeptanztests durchgeführt werden, ist Tobias Hahn für die Kommunikation verantwortlich.

Gebrauchstauglichkeitstests werden durch das gleiche Dokument wie die Akzeptanztests dokumentiert. Dessen Aufbau wird am Ende des Kapitels kurz aufgezeigt. Jedoch werden sie von den Entwicklern selbst durchgeführt. Jedes Team-Mitglied ist dazu angehalten mindestens alle zwei, optional jede Woche einen Gebrauchstauglichkeitstest durchzuführen und zu dokumentieren. Überprüft werden folgende Punkte.

- Konsistentes Farbschema
- Konsistente Formauswahl (Button usw.)
- Konsistente Skalierung bezüglich Schrift und Formelementen
- Übersichtlichkeit
- Intuition
- Lesbarkeit
- Vollständigkeit abgeschlossener Features
- Korrekte Rechtschreibung und Grammatik
- Funktionalität in der Praxis
- (Performance)

Die meisten dieser Punkte sind deshalb sehr wichtig, da sie nicht automatisierbar sind. Der Entwickler selbst, in der Rolle eines Spielers, muss testen. Diese Art des Testens ist somit am nächsten an der Realität und deshalb sehr wichtig. Es werden nicht nur einzelne Komponenten oder deren Zusammenspiel, sondern das Zusammenspiel von allem getestet. Getestet wird jeweils der aktuelle MasterBranch.