Quality-Assurance-Document

Testplan & Definition-of-Done

-

von ahoiSoftware

Version 1.4

Inhaltsverzeichnis

1. Änderungshistorie

Version	Datum	Änderung	Geändert von
1.0	07.11.2018	Initial	Artem Burchanow
1.1	13.12.2018	Gemäß dem Entschluss des Weekly-Meetings vom 04.12.2018 wird der Entwicklungsprozess kein Test-Driven-Development mehr vorsehen	Artem Burchanow
1.2	16.01.2019	Gemäß dem	Artem Burchanow

		Entschluss des Weekly-Meetings vom 08.01.2019 sind Reviews bei Tasks nicht mehr nötig	
1.3	18.01.2019	Ergänzungen für den Umgang mit Issues	Artem Burchanow
1.4	24.01.2019	Berücksichtigung von Android-Studio und Coverage-Tools und Gradle	Artem Burchanow

2. Anlagen

Auf folgende gesondert angehängte Dokumente wird in dem QA-Dokument verwiesen:

- Code-Style-Guideline (Code-Style-Guideline.pdf)
- Git-Guideline (Git-Guideline.pdf)
- Gestaltungsguideline (Gestaltungsguideline.pdf)
- Scrum-Guideline (Scrum-Guideline.pdf)

3. Definition-of-Done

3.1 Ziel der Definition-of-Done

In der Definition-of-Done wird festgelegt, wann Ziele einzelner Kategorien als erreicht beziehungsweise als abgeschlossen gelten. Diese Ziele sind als Richtlinien anzusehen und sind im gesamten Projektteam akzeptiert. Grundsätzlich wird durch die Definition-of-Done ein Standard an Qualität für das Softwareprojekt geboten.
Dieses Dokument ist nicht als statisch anzusehen. Über die Scrum-Retroperspektive können Erfahrungswerte auch im Nachhinein noch mit in das Dokument einfließen. Eine Änderung der Definition-of-Done behält sich nur der Qualitätsmanager[1] vor. Die Änderungsanfragen sind im akzeptierten Kommunikationsmedium Slack im Channel #DoD-Anpassungen zu sammeln. In der wöchentlichen Retroperspektive wird über die Änderungen abgestimmt und bei Bedarf das Dokument durch den Qualitätsmanager angepasst. Die Änderungen sind in der Änderungshistorie festzuhalten.

3.2 Abschluss von Tasks

Erst wenn alle folgenden 6 Punkte erfolgreich umgesetzt sind, so wird der Status des Tasks in Taiga[2] auf „Done“ gesetzt.

1. Der Code ist kompilierbar

Der Code ist mithilfe von Maven vollständig und fehlerfrei kompilierbar.

[1] Qualtitätsmanager ist Artem Burchanow
[2] Taiga ist ein Projektmanagement-Tool

2. Der Code wurde getestet

Wann ein Test erfolgreich ist, hängt von dem verwendeten Testverfahren ab. Für ein erfolgreiches Bestehen der Unit-Tests ist es ausreichend, wenn alle geschriebenen Tests bestanden sind. Komponenten, die Teil von Integrationstests sind müssen vorher alle anwendbaren Unit-Tests sowie, falls benötigt, alle anwendbaren manuellen Tests bestehen. Anschließend wird getestet, ob diese Komponenten im Verbund fehlerfrei funktionieren und die gewünschten Resultate erzielen. Ist dies der Fall, gelten die Integrationstests als erfolgreich. Bei Tests von Netzwerkkomponenten wird überprüft, ob eine fehlerfreie Netzwerkkommunikation gewährleistet ist.
Ein manueller Test gilt als erfolgreich, sofern das Resultat den Erwartungen entspricht und keine anderen Durchführungen Resultate erzeugen, die der Erwartung entsprechen (Vgl. Manuelle Tests). Dabei ist anzumerken, dass Benutzeroberflächen manuell getestet werden und somit in den Test-Coverage-Reports nicht auftauchen. Interfaces müssen nicht getestet werden, da sie keinerlei Funktionalität bieten.

3. Die Code-Guideline ist eingehalten

Der Code-Style entspricht den in der Code-Style-Guideline festgehaltenen Richtlinien (siehe Anlagen).

4. Der Code ist vollständig dokumentiert

Im Laufe des Projektes wird JavaDoc als Dokumentation im Quellcode verwendet. Es gelten die im Testplan und der Code-Style-Guideline festgehaltenen Richtlinien (siehe Anlagen).
Die Dokumentation ist vollständig, wenn sie den aktuellen Stand der Entwicklung wiederspiegelt. Hierbei sind auch fundamentale Änderungen an der Software zu beachten. Diese Änderungen müssen auch mit in das Benutzerhandbuch einfließen.
Falls notwendig müssen Änderungen auch mit in das DevOps-Dokument einfließen.

5. Der Code wurde in GIT commited

Sind die ersten 4 Punkte abgeschlossen, so muss der Code gemäß der Git-Guideline (siehe Anlagen) commited und gepusht werden.

6. GUI Elemente entsprechen den Gestaltungsrichtlinien

Sämtliche GUI-Elemente haben den Gestaltungsrichtlinien (siehe Anlagen) zu genügen. Ausnahmen sind in begründeten Fällen mit dem Team abgesprochen und akzeptiert worden.

3.3 Abschluss von Issues

Falls in einem bereits fertiggestellten Teil der Software Fehler oder Unstimmigkeiten auftreten, so werden diese als Issue in Taiga festgehalten und mit einer Category und Priority versehen.
Issues werden eigenständig vom Entwickler erstellt und können im Anschluss optional einer User-Story zugeordnet werden. Sie gelten dann als abgeschlossen, wenn sie die gleichen Kriterien wie Tasks erfüllen (Vgl. Abschluss von Tasks). Ihr Status wird dann auf „Done" gesetzt.

3.4 Abschluss von User-Stories

User-Stories wurden zu Beginn des Projekts vom Product-Owner[3] akzeptiert. Am Anfang des Sprints wurden nach der Retroperspektive des letzten Sprints im Team User-Stories zur Bearbeitung ausgewählt und die Zuständigkeiten eingeteilt.
Eine User Story ist als abgeschlossen anzusehen, wenn alle ihr zugeordneten Tasks die oben genannten Kriterien (Vgl. Abschluss von Tasks) erfüllen und ihr Status in Taiga entsprechend auf „Done" gesetzt wurde. Dies hat zur Folge, dass auch die übergeordnete User-Story auf „Done" gesetzt werden kann und somit als fertiggestellt gilt. Zuletzt muss noch der Product-Owner im Sprint-Review über die Finalisierung entscheiden. Zur Sicherstellung der Funktionalität und Einhaltung der

[3] Product-Owner ist Lukas Gehring

Code-Style-Guideline wird ein Code-Review durchgeführt (Vgl. Code-Review). Die erfolgreiche Durchführung des Code-Reviews wird zudem in Taiga mit dem Statusübergang von „in Review“ nach „Done“ markiert. Dabei ist es möglich, dass zusammenhängende Funktionalitäten in einem Review überprüft werden, obwohl es einzelne User Stories sind.

3.5 Abschluss von Sprints

Ein Sprint gilt als erfolgreich beendet, wenn die ihm im Projektmanagement-Tool Taiga zugewiesenen User-Stories als abgeschlossen markiert sind (Vgl. Abschluss von User-Stories). Die Überprüfung, ob ein Sprint erfolgreich abgeschlossen ist, findet in Zusammenarbeit mit dem Product-Owner[3] im wöchentlichen Sprint-Review statt.
Sollte es vorkommen, dass ein Sprint nicht vollständig abgeschlossen ist, so fließen die nicht abgeschlossenen User-Stories (Vgl. Abschluss von User-Stories) wieder zurück in das Product-Backlog. Diese User-Stories werden dann erneut in den nächsten Sprints eingeteilt und abgearbeitet.

3.6 Abschluss von Epics

Epics werden durch die Ihnen zugeteilten User-Stories definiert und gelten als abgeschlossen, wenn alle zugehörigen User-Stories abgeschlossen sind (Vgl. Abschluss von User-Stories).

3.7 Abschluss von Releases

Releases sind entsprechend den Git-Guidelines zu erstellen. Der Git-Tag des Releases wird im Anschluss per E-Mail an den Auftraggeber übermittelt, um ihn über das Release zu informieren. Releases werden erzeugt, wenn größere, lauffähige Module der Software gemäß den oben genannten Kriterien (Vgl. Abschluss von User-Stories) fertiggestellt sind. Damit ein Release vollständig ist, dürfen keine kritischen Bugs offen sein. Außerdem müssen alle Hotfixes gemäß der Git-Guideline (siehe Anlagen) gemerged sein. Zudem muss ein Review über das Release im Projektteam stattgefunden haben.

4. Testplan

4.1 Ziel des Testplanes

In diesem Dokument wird grundlegend festgelegt, wie und in welchem Umfang getestet wird. Dabei wird zudem auch auf die zu verwendeten Tools und die damit einhergehende Integration in den Testplan eingegangen. Funktion des Testplans ist es, dem Team eine Vorlage zu geben, in der genau beschrieben ist, wie mit Tests umgegangen werden muss. Der Qualitätsmanager[4] behält sich vor den Testplan in Abstimmung mit dem Projektteam auch im Entwicklungsprozess zu ändern, da durch die durchgeführte Retroperspektive eines Sprints zukünftige Erfahrungen mit in den Testplan einfließen können. Das Dokument ist somit nicht statisch.

4.2 Verwendete Teststrategie

Im Folgenden wird die Teststrategie und die in der Entwicklung verwendeten Tools genauer erläutert. Zum Einsatz kommen dabei Maven in Verbindung mit GitLab CI/CD. Des Weiteren wird auf Unit-Tests und manuelle Tests der GUI in Kombination mit Code-Reviews gesetzt.

4.3 Integration der verwendeten Tools

Zur Absicherung der Qualität der zu entwickelnden Software werden die Tools Maven, Gradle und GitLab CI/CD verwendet. Dabei agieren Maven, Gradle und EclEmma direkt in der IDE[5] des Entwicklers. Im Gegensatz dazu wirkt GitLab CI/CD erst beim Commit in GitLab.

[4] Qualitätsmanager ist Artem Burchanow

[5] Integrierte Entwicklungsumgebung

4.3.1 Maven und Gradle

Maven dient in Eclipse als Hilfsmittel, um eine Einteilung des Projektes in Module zu ermöglichen und erlaubt eine einfache Einbindung von Plugins, Bibliotheken oder anderen Projekten. Mehrere Module oder auch ein ganzes Projekt können mittels JUnit getestet werden. Dies kann auch direkt vor dem Buildprozess stattfinden, um fehlerhafte Builds zu vermeiden.
Ein Modul bietet sogar die Möglichkeiten dieses einzeln und unabhängig oder aber im Verbund mit anderen Modulen und Abhängigkeiten zu testen. Abgesehen davon kann ein Maven-Modul mehrmals, auch in anderen Maven-Modulen, verwendet werden.
Gradle hingegen unterstützt den Entwickler in Android-Studio bei automatisierten Builds und JUnit-Tests. Gradle hat dieselbe Intention wie Maven.

4.3.2 Gitlab CI/CD

GitLab CI/CD agiert in Kooperation mit Maven zur Qualitätssicherung der Commits. Dabei kontrolliert es einen Commit durch eine festgelegte Reihenfolge von Tests und/oder anderen Befehlen aus der Datei „.gitlab-ci.yml". Dies soll dafür sorgen, dass Commits möglichst fehlerfrei bleiben und bestehende Fehler aufgedeckt werden. Ausgeführt werden diese Tests auf einem sogenannten „Runner", der die Commits überprüft. Dieser Client kann sowohl auf der Maschine des Entwicklers, als auch auf einem Server laufen.

4.3.3 EclEmma

EclEmma ist ein Code-Coverage-Tool für Eclipse und Android-Studio, welches besonders bei der Entwicklung dem Programmierer eine Übersicht über die Testabdeckung des Codes während der Entwicklung liefern kann. Da EclEmma direkt in Eclipse und Android-Studio integriert ist, bietet es die Möglichkeit eine Analyse zu erstellen indem das Programm im „Coverage Mode" gestartet wird. Dies ermöglicht die Benutzung einer Oberfläche, welche eine Auflistung aller Java Dateien und deren prozentuale Abdeckung beinhaltet. Außerdem kann diese Abdeckung auch direkt im Programm-Code mittels Einfärbung dargestellt werden.

4.4 Manuelle Tests

Werden Tests von Komponenten durchgeführt, welche beispielsweise nicht auf Funktionalität, sondern auf Nutzbarkeit abzielen, wie es bei Tests von Oberflächen oft der Fall ist, kommen manuelle Tests zum Einsatz.
Ein manueller Test besteht aus drei Teil-Tests, von denen der erste auf die beabsichtigte Funktionalität/Nutzbarkeit gemäß der Erwartung prüft. Der zweite testet auf unbedachte Nutzung anderer Elemente, falscher Eingaben oder gezielten „Problem Eingaben" (z.B. Texteingaben in einem Zahlenfeld). Der letzte Test wird als DAU[6] durchgeführt und überprüft, ob ein Resultat auf einem nicht vorgesehenen Weg erzielt werden kann.
Dokumentiert wird ein manueller Test mittels der Vorlage „manueller Test", startend mit der Benennung des Ablaufes, also welche Schritte bei einem Vorgang durchgeführt werden sollten. Es folgt die Festlegung des zu erwartenden Resultates in der entsprechenden Vorlage für manuelle Tests.
Danach werden die Tests durchgeführt, beginnend damit, dass das Resultat der Durchführung der generellen Handlung notiert wird.
Im nächsten Teil werden alle Testdurchläufe aus dem zweiten Test aufgelistet, die das erwartete Resultat geliefert haben, wobei jeder Eintrag eine Nummer zugewiesen bekommt. Das heißt, dass eine Möglichkeit einer Durchführung gefunden wurde, die zu der Erwartung geführt hat.
Ein einfaches Beispiel ist folgendes: Die Bestätigung des Buttons „Login" führt bei korrekter Eingabe des Benutzernamens und Server-IP in die Lobby.

[6] dümmster anzunehmender User

Auch der DAU-Test wird in dieser Form als nummerierte Aufzählung festgehalten. Schlussendlich gibt es noch die Möglichkeit Auffälligkeiten, die bei den Tests aufgetreten sind, unter diesem Punkt zusammenzufassen.
Oberflächen werden durch manuelle Tests gemäß dem oben genannten Schema überprüft.

4.5 Unit-Tests

Unit Tests sind möglichst kleine unabhängige Codefragmente, welche dazu dienen, einzelne Methoden auf ihre Funktion zu prüfen. Zum Testen werden Testfunktionen verwendet, sodass die zu testende Methode möglichst entkoppelt zum Rest des Projektes getestet werden kann. Wichtig ist dabei, dass die Testfunktionen möglichst einfach zu halten sind und das zu erwartende Ergebnis der zu testenden Methode bei Eingabe der Testfunktion vollständig vorher bekannt ist. Dies dient dazu fehlerhafte Ergebnisse einfach, schnell und zuverlässig zu erkennen. Zudem ist dadurch gewährleistet, dass der Fehler vollständig auf den zu testenden Code zurückzuführen ist.
Ziel für Unit-Tests ist es eine möglichst hohe Abdeckung der Codebase[7] zu erzielen. Aus diesem Grund ist das Ziel jede nicht triviale Methode mittels Unit Tests abzudecken, die maschinell getestet werden kann. Trivial sind dabei Methoden, die nur aus einem einzelnen Befehl bestehen (z.B.: Getter & Setter) oder nicht essentiell für die korrekte Ausführung des Programms sind (z.B. Debug- oder Logging-Methoden).
Die Unittests werden dabei vor dem zu testenden Code geschrieben und ähnlich wie der Code selbst einem anderen Entwickler zum Review vorgelegt. Nur wenn das Review erfolgreich war, beginnt die Implementation des zu testenden Codes (Vgl. **Fehler! Verweisquelle konnte nicht gefunden werden.**). Andernfalls müssen fehlerhafte und/oder vergessene Testfälle korrigiert und ergänzt werden, sodass eine vollständige Test-Coverage für nicht triviale Methoden erreicht wird. Entwickler führen keine Reviews an ihrem eigenen Code durch.
Unit-Tests werden wiederholt, falls der zu testende Code geändert wurde. Ein einzelner Unit-Test sollte, bei nicht zeitabhängigen Methoden, dabei eine maximale Laufzeit von 60 Sekunden aufweisen. Wird diese Zeit überschritten, gilt der Test als nicht bestanden. Zeitabhängige Methoden sind dabei Methoden, welche zeitgesteuerte und nicht leistungsabhängige Laufzeiten besitzen. Jede Unit-Test-Datei besteht aus mehreren Tests, welche jeweils genau eine Testklasse umfassen. Der erste Teil jeder Testmethode besteht dabei immer aus dem Setup der Testumgebung, also der Konfiguration, auf der die zu testende Methode ausgeführt werden soll. Der zweite Teil umfasst die Testabfragen, die direkt auf die zu testende Methode angewandt werden.
Ein Gegenbeispiel für eine sinnvolle Anwendung von Unit Tests ist folgender Pseudo-Code:

```
public void testSomething() {
//setup test
assert_true(foo());
}

public boolean foo() {
    boolean x = false;
    if (bar()) {
    x = true;
    }
    else {
    x = baz(); //<- this method should be tested
    }
    return !(x && bar())||bar();
}

public boolean bar() {
//do something
}

public boolean baz() {
//newly implemented Code that need to be tested
}
```

[7] Der Begriff Codebase im Bereich der Softwaretechnik bezeichnet die Gesamtheit der zu einem Projekt gehörenden Quelltextdateien. Vgl. https://de.wikipedia.org/wiki/Codebasis abgerufen am 10.11.2018

4.6 Testen der Komponenten

Durch Mavens Möglichkeiten im Projektaufbau können auch ganze Komponenten, wie die Netzwerk- oder Interface-Komponente, mit JUnit Tests überprüft werden. Nachteil dabei ist jedoch der große Aufwand für einen Setup-Aufbau.

4.7 Security und Bugs

Falls während der Implementierung oder beim Testen Sicherheitslücken oder Bugs[8] auftauchen, werden diese in Taiga als Issues festgehalten und demjenigen Projektmitglied zugewiesen, der für die Funktionalität verantwortlich ist. Falls die Zuständigkeit nicht direkt geklärt werden kann, wird Rücksprache mit dem Projektleiter oder Gruppenleiter gehalten. Optional kann die Zuständigkeit aber auch im entsprechendem Slack-Channel geklärt werden.
Security-Issues werden mit höchster Priorität versehen und unter Severities als Critical markiert, während Bugs, je nach Relevanz des Fehlers, mit niedrigerer Priorität eingestellt werden. Dies stellt sicher, dass Security-Issues vor regulären Bugs abgearbeitet werden. Issues können von jedem Projektmitglied eingestellt werden.
Issues werden nach der in der Definition-of-Done festgelegten Richtline abgeschlossen (Vgl. Abschluss von Issues).

4.8 Code-Review

Nachdem ein Entwickler ein Modul oder eine Funktion in der Implementierung abgeschlossen hat, wird ein zweiter, beliebiger Entwickler zur Hand genommen, der den Code nach genannten Kriterien und eventuellen Fehlern überprüft (Vgl. Abschluss von Tasks).
Pair-Programming wird besonders für kritische Infrastrukturen angewendet, wo sich ein Team von mindestens zwei Entwicklern mit der Implementation beschäftigt. Auch bei Pair-Programming wird ein Pair-Review durchgeführt.

[8] ein Bug ist ein unerwartetes auftauchendes Fehlverhalten der Software