UNIVERSITÄT
LEIPZIG

Softwaretechnikpraktikum

# Qualitätssicherungskonzept


**Gruppe:** nw19a

**Mitglieder:** Thomas Pause, Sabine Lorius, Arik Korte, Martin George, Josephine Lange, Esther Prause, Anh Kiet Nguyen, Bärbel Hanle

**Verantwortlich:** Arik Korte & Thomas Pause

**Betreuer:** Dr. Nicolas Wieseke

**Tutor:** Martin Frühauf



**Abgabedatum:** 16.12.2019


**Stand:** 16. Dezember 2019

# Inhaltsverzeichnis

# 1 Dokumentationskonzept

Eine gute Dokumentation erleichtert es Softwareentwicklern, sich schnell in den Code einzuarbeiten, auch wenn sie bislang nicht am Projekt beteiligt waren. Im Rahmen eines größeren Softwareprojekts ist dieser Aspekt besonders wichtig, insbesondere wenn das Produkt nach Abschluss des Projekts weiterentwickelt werden soll.
In unserem Projekt wollen wir Dokumentation auf drei Ebenen mit unterschiedlichem Fokus erstellen: die Entwurfsbeschreibung mit Blick auf das Gesamtprodukt, die quelltextnahe, strukturierte Dokumentation, welche von außen auf die einzelnen Klassen schaut, und die interne Dokumentation, die in den einzelnen Klassen Details der Implementierung erklärt.

## 1.1 Entwurfsbeschreibung

Zu jedem Release ist eine Entwurfsbeschreibung abzugeben[1]. Sie liefert einen Überblick über die Struktur des Softwareprodukts und dient als Einstiegspunkt für einen Entwickler, der anfängt sich mit dem Projekt zu beschäftigen. Deshalb werden in diesem Dokument auch alle wichtigen Modellierungs-, Struktur- und Designentscheidungen begründet.

## 1.2 Quelltextnahe, strukturierte Dokumentation

Um im Sinne der objektorientierten Programmierung Klassen wiederverwenden zu können, werden diese in der quelltextnahen, strukturierten Dokumentation beschrieben. Dazu gehört neben der allgemeinen Beschreibung der Klasse eine Erklärung sämtlicher Methoden und Konstanten, die von ihr zur Verfügung gestellt werden. Wir verwenden KDoc[2], um diese Dokumentation automatisch zu generieren.

## 1.3 Interne Dokumentation

Bei der internen Dokumentation werden erklärungsbedürftige Schritte durch Kommentare im Code erläutert. Sie hilft dem Entwickler also, sich in einer Klasse zurechtzufinden. Durch sprechende Namen für Variablen und Methoden sowie durch einen einfachen, gut strukturierten Code soll diese Art der Dokumentation weitgehend überflüssig gemacht werden.

# 2 Coding Standard

## 2.1 Code Conventions

Als Standard nutzen wir den *Kotlin Style Guide*[3], welcher Bestandteil des Kotlin-Plugins für Android Studio ist.
Um diesen zu aktivieren, muss in den Einstellungen von Android Studio folgendes geändert werden:
Bei **File | Settings | Editor | Code Style | Kotlin** wird auf **Set from...** in der rechten oberen Ecke geklickt und dann **Predefined style | Kotlin style guide** ausgewählt. Für die automatische Überprüfung der Formatierung muss noch bei **File | Settings | Editor | Inspections | Kotlin | Style issues | File is not formatted according to project settings** ein Häkchen gesetzt werden.

## 2.2 Linting

Unter dem Begriff *Linting* wird das automatisierte Durchführen von statischen Code-Analysen zusammengefasst. Der Begriff leitet sich von Lint ab, dem ersten für diese Aufgabe entwickelten Werkzeug.

Zweck dieser Praktik ist es, Schwachstellen im Quellcode aufzuspüren, also auf Bugs, stilistische und programmatische Fehler sowie sog. *verdächtige Konstrukte* zu prüfen.

# 3 Testkonzept

Die Definition, Planung und Durchführung von Softwaretests ist für die Erfüllung der Anforderungen an das Produkt unerlässlich. Daher stellt das richtige Testkonzept ein Qualitätsmaß dar.
Allerdings dienen Tests nicht dazu zu beweisen, dass keine Fehler oder Schwachstellen in der Software vorhanden sind. Sie überprüfen lediglich die gewählten Testfälle.
Ziel ist es, mit möglichst geringem Zeit- und Ressourcenaufwand eine möglichst hohe *Test-Coverage* zu erhalten. Wir orientieren uns hierbei an der aus dem Tutorium bekannten *Testpyramide*, die auf die effiziente Verteilung der Tests ausgelegt ist.

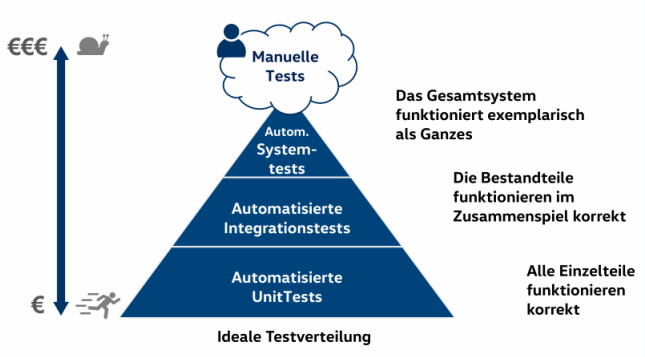


Abbildung 1: Die Testpyramide

## 3.1 Komponententests

Komponenten- oder Unit-Tests überprüfen die Funktion einzelner Methoden. Sie werden vom jeweiligen Entwickler selbst geschrieben. Damit der Ergebnisbereich eines Komponententests überschaubar bleibt und insgesamt eine gute Testabdeckung erreicht wird, soll die Komplexität von Methoden reduziert und auf einzelne Methoden mit möglichst einfacher Funktionalität verteilt werden.
Da wir mit der IDE Android Studio in der Sprache Kotlin programmieren, nutzen wir das für Java und Kotlin entwickelte Framework *JUnit 5*[4], welches in dieser Umgebung den Standard definiert.
Pro geschriebener Klasse soll es eine Testklasse geben, pro Funktion einen Test. Wir unterteilen die Unit-Tests jeweils in die Teile *Arrange*, *Act* und *Assert*. Arrange instanziiert ein Objekt der zu testenden Klasse, welches auch global für alle zu testenden Fälle verwendet werden kann. Act führt die zu testende Funktion auf diesem Objekt aus und Assert überprüft das erhaltene Ergebnis bzw. vergleicht das tatsächliche Verhalten mit dem erwarteten. Zur besseren Lesbarkeit im automatisch generierten Testbericht nutzen wir die Möglichkeit von Kotlin, Bezeichner mit Leerzeichen als Funktionsnamen zu verwenden. Dabei ist die Bezeichnung nach dem folgenden Muster zu wählen:

> ‘Test <funXY> to assert <expected result>‘

Wenn in Unit-Tests Schnittstellen geprüft werden, muss die andere Seite der Schnittstelle durch *Mocking* simuliert werden, um lediglich die Funktionalität der einzelnen Seite zu testen. Wir verwenden hierfür die JUnit-Erweiterung *Mockito*[5].

### 3.2 Integrations- und Systemtests

Integrationstests überprüfen die Zusammenarbeit von unabhängig voneinander entwickelten Modulen. Dabei wird unter anderem der Datenfluss bzw. -austausch getestet. Die Erstellung und Durchführung dieser Tests ist Aufgabe der Qualitätsmanager (siehe 4.5) und soll perspektivisch in die Continuous Integration ausgelagert werden. Wir nutzen dafür die Möglichkeiten von *JUnit 5* und *Gradle*[6].
Unter Systemtests versteht man Ende-zu-Ende-Tests durch das gesamte System. Dabei soll die Usersicht eingenommen und über die Benutzerschnittstelle (das User Interface) ein möglichst alle Alternativen abdeckender Weg durch die Anwendung gegangen werden. Wir möchten dies mithilfe des Frameworks *Espresso*[7] automatisieren.
Dadurch können wir weitestgehend auf umfangreiche manuelle Tests verzichten. Dennoch sollen ergänzend auch manuelle Systemtests der Anwendung durchgeführt werden.

### 3.3 GitLab Continuous Integration

Die in GitLab integrierte CI ermöglicht es, bereits beim Hochladen von Änderungen bestimmte Test-, Build-, Debug- und Lint-Jobs durchzuführen. Dies erfolgt über ein Docker-Image, welches ein lauffähiges Linux-System mit den Voraussetzungen zur Ausführung von Android-Anwendungen beinhaltet. In diesem sind auf (momentan) 3 Stages verschiedene Jobs definiert worden, die auf den verschiedenen Branches unterschiedliche Aufgaben durchführen. Sie nutzen den für Android Studio als Standard vorgesehenen Gradle Builder.
Das Teammitglied, das den Code gepusht hat, wird per E-Mail benachrichtigt, wenn ein Job fehlgeschlagen und der Build damit nicht erfolgreich verlaufen ist.
Auf die Einrichtung von *Pre-Commit Hooks*, welche Commits bereits lokal zurückweisen, verzichten wir bewusst. Eine Erweiterung des CI-Systems um beispielsweise (halb-)automatisierte Integrations- und Systemtests sowie automatisch generierte Testberichte streben wir jedoch an.

## 4 Organisatorische Festlegungen

### 4.1 Kommunikation

Der organisatorische und fachliche Austausch im Entwicklerteam erfolgt über die Softwarelösung Microsoft Teams, in der man neben persönlichen und Gruppenchats sowie (Video-)Telefonaten auch Kanäle für bestimmte Themen definieren kann. Innerhalb dieser lassen sich Dateien, Links, ein Wiki sowie beispielsweise ein Kalender integrieren. So wird die Kommunikation stets übersichtlich und jeder Einzelne gut informiert gehalten.

### 4.2 Vorausschauende und transparente Planung

Es ist Aufgabe der Projektleitung, möglichst frühzeitig eine globale Planung des Projekts über die gesamte Laufzeit aufzustellen, bei der die unterschiedlichen zeitlichen Beschränkungen aller Teammitglieder berücksichtigt werden und auf Ausfälle flexibel reagiert werden kann. Dies garantiert Transparenz und Terminsicherheit für das Team. Für dieses Projekt wurde bereits eine zeitliche Durchstrukturierung bis zum Abschluss erarbeitet, welche im Releaseplan für den Auftraggeber schriftlich fixiert wird. Zudem wurden die zum 1. Release zu erreichenden Teilziele und daraus resultierenden Aufgaben dem Team bereits *en détail* zur Verfügung gestellt, so dass hinsichtlich der Anforderungen bis zum Jahresende keine Unsicherheiten entstehen können. Darüber hinaus ist zwischen dem 2. und 3. Release eine *flexible Gleitphase* vorgesehen, in der sich jeder individuell Zeit für die Prüfungsvorbereitung sowie nach Wunsch für Urlaub nehmen kann.

Mit diesen Maßnahmen soll die Motivation des Teams und damit die resultierende Qualität der Software hoch gehalten werden.

## 4.3 Teamtreffen

Für den stetigen Austausch und die Verteilung von Aufgaben finden jede Woche mittwochs ab 14 Uhr Team-Meetings statt. Ab 17 Uhr sind dann auch Auftraggeber und Tutor mit vor Ort, außerdem werden Reviews zu diesen Terminen durchgeführt. Die Treffen werden vorbereitet (in Form einer Agenda, die von den Projektleitern und der moderierenden Person entworfen wird), protokolliert und nachbereitet (u.a. durch die Übertragung der besprochenen Aufgaben in Git-Issues). Die Rolle der Moderation und des Protokollführenden rotiert jede Woche, um für eine hohe mentale Involviertheit in das Projekt zu sorgen.
Zudem haben wir eine Mediator-Rolle an Esther Prause vergeben, die beim Umgang mit internen Ungereimtheiten, Missverständnissen und Streitigkeiten zum Tragen kommt. Für Unterstützung und Beratung bei mentalem Stress, Demotivation und anderen individuellen psychischen Herausforderungen das Projekt betreffend haben wir eine Rolle als Stresscoach (Sabine Lorius) definiert.
Des Weiteren nutzen wir Motivationsmethoden wie den *Check-In*, um die Effizienz unserer Meetings zu maximieren und als Gruppe eine gemeinsame Diskussionsbasis zu behalten.

## 4.4 Arbeit mit dem GitLab-Repo

### 4.4.1 Git-Verantwortlicher

Für die Verwaltung des GitLab-Repos ist der Git-Verantwortliche Thomas Pause zuständig. Er ist auch direkter Ansprechpartner bei Fragen rund um dieses Konzept. So können wir als Team ein sortiertes und durchdachtes Versionierungs-Werkzeug nutzen, das sich entsprechend den Anforderungen erweitert lässt. Für die wachsenden Anforderungen wird eine zweite Person aus dem Team nominiert, den Git-Verantwortlichen bei Bedarf zu unterstützen.

### 4.4.2 Issues und Vorgaben zu Commits

Anfallende Aufgaben werden vom Git-Verantwortlichen in Issues umgesetzt, mit einem Meilenstein verknüpft und mit einem Label sowie einem Ablaufdatum versehen.
Meilensteine sind Abgabetermine, zu denen bestimmte Arbeitspakete final zur Bewertung übermittelt werden. Bis zu diesem Zeitpunkt müssen alle in den verknüpften Issues definierten Aufgaben erledigt sein.
Bei jedem Commit soll auf das zugehörige Issue im Repo verwiesen werden, damit der Arbeitsablauf nachvollziehbar bleibt. Dies geschieht mit dem Schlüsselwort *see* gefolgt von # und der Nummer des Issues, auf das sich der Commit bezieht.
Um einheitliche Commits zu erhalten, definieren wir folgende Formatvorgabe für Commit-Messages:
“Add, Update, Delete these things to this file (details), see #NoOfIssue".

### 4.4.3 Der Lebenszyklus eines GitLab-Issues

Um Arbeiten gut nachvollziehbar und planbar zu machen, wird der folgende exemplarische Lebenszyklus eines Issues definiert. Dabei stehen die Farben und Bezeichnungen für bereits eingerichtete Label, die flexibel vergeben werden können.
Nach der Erstellung eines Issues durch den Git-Verantwortlichen wird neben dem Meilenstein, einer Beschreibung der zu erledigenden Aufgabe sowie einem Ablaufdatum das *ToDo*-Label vergeben.
Wenn die Aufgabe bearbeitet wird, setzt der jeweilige Bearbeiter sich als Assignee und das Label auf

*Doing*. Ist die Bearbeitung der Aufgabe abgeschlossen, wird das Label auf *Review* geändert. Das Review wird immer von einer anderen Person und einem Qualitätsmanager (siehe 4.4.4 und 4.5) durchgeführt. Dabei wird entschieden, ob Nacharbeiten nötig sind (Label *ReDo* und Wiederholung des Zyklus) oder ob die Aufgabe zufriedenstellend gelöst ist und das *Done*-Label vergeben werden kann.
Für Anweisungen, was im *ReDo* zu erledigen ist, verwenden wir die Möglichkeit von GitLab, Kommentare im letzten Commit direkt im Code zu hinterlassen.
Nach einem weiteren Code-Review mit dem Team wird dann das Issue rechtzeitig vor Ablauf des betreffenden Meilensteins vom Git-Verantwortlichen geschlossen.
Zur zusätzlichen Gewichtung der Dringlichkeit könnte man noch die Weight-Funktion nutzen und zur Definition von Abhängigkeiten wäre das Arbeiten mit Major- bzw. Minor-Versionsnummern an den Issues möglich (so dass bspw. Task 1.2 vor Task 1.3 erledigt sein muss). Dies wird allerdings nur bei Bedarf und in Absprache mit dem Team umgesetzt, falls es benötigt wird.
Dieses Vorgehen ermöglicht uns bestmögliche Kontrolle des Entwicklungsprozesses bei maximaler Flexibilität in der Umsetzung der Teilaufgaben. So soll der Kreativprozess gefördert und trotzdem eine gleichbleibend hohe Qualität gewährleistet werden.

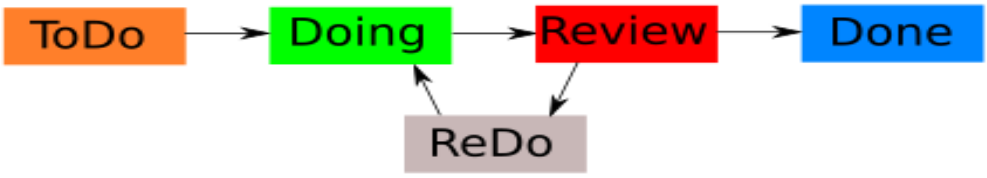


Abbildung 2: Der Lebenszyklus eines GitLab-Issues

### 4.4.4 Git-Workflow

Um einen möglichst störungsfreien Entwicklungsablauf sicherzustellen, haben wir uns für den *Gitflow-Workflow*[8] entschieden, eine logische Erweiterung des zentralisierten Git-Workflows. Dafür wird vom Master-Branch ein Develop-Branch abgezweigt, auf dem die versionierte Entwicklung des Softwareprodukts durchgeführt wird. Nach Abschluss des Vorprojekts sollen zusätzlich Feature-Branches erstellt werden, deren Eltern-Branch der Develop-Branch ist. In diesen können dann einzelne Erweiterungen der Software erstellt und getestet werden. Nach einem positiven Code-Review durch mindestens ein nicht an der Entwicklung des Features beteiligtes Teammitglied wird die getestete Erweiterung auf den Develop-Branch gemerged und der nicht mehr benötigte Feature-Branch gelöscht.
Die Zusammenführung mit dem Master-Branch erfolgt nur für stabile Releases, die dem CI-Konzept entsprechend geprüft wurden (siehe 3.3) und wird nur direkt vor Releases durchgeführt. Dafür wird das komplette Team in den Reviewprozess integriert. Somit bleibt der Master-Branch aufgeräumt und es befinden sich dort nur lauffähige Versionen des Produktes.
Abbildung 3 zeigt eine Übersicht des gewählten Workflow-Prinzips.

### 4.4.5 GitLab Issue-Board

Für die Entwicklung unserer App wählen wir einen agilen Ansatz, SCRUM. Dabei soll in fünf *Sprints* das Produkt inkrementell entwickelt werden. Das bedeutet, dass während jeder dieser Entwicklungsphasen ein stabiles, getestetes und dokumentiertes Produkt mit wachsendem Funktionsumfang bereitgestellt wird.
Ein wichtiges Instrument ist hierbei das *Item-Backlog*, eine ToDo-Liste, die vor jedem Sprint erstellt

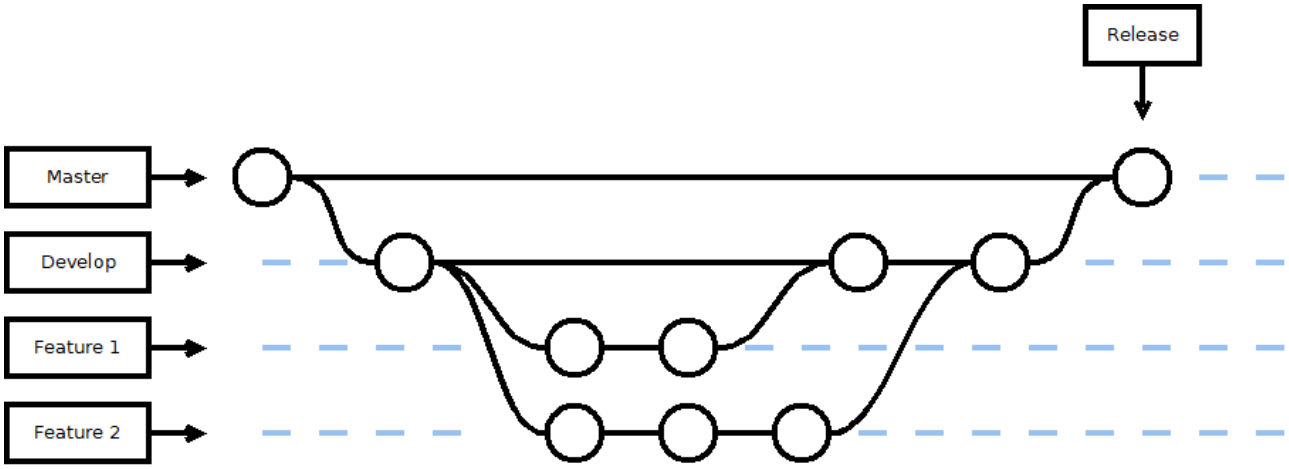


Abbildung 3: Das Branching-Modell des Gitflow-Workflows

wird. Wir nutzen hierfür die Idee eines *Kanban-Boards*, welches übersichtlich den Arbeitsstand aufzeigt und uns hilft, interne Code-Reviews planen zu können. Als Mittel nutzen wir die Möglichkeiten des Issue-Boards auf GitLab, welches in Kombination mit oben erwähnten Issue-Labels eine Plattform für die Planung und Durchführung der Sprints bietet. Vorbereitet, geleitet und nachbearbeitet werden diese durch einen Qualitätsmanager.
Abbildung 4 zeigt ein beispielhaftes Issue-Board, welches auf unserem internen Test-Repo angelegt wurde.

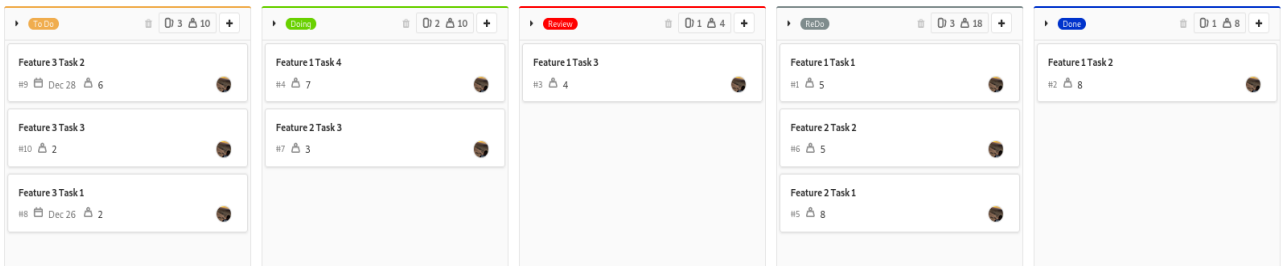


Abbildung 4: Beispiel zur Nutzung von GitLab-Board als Kanban-Board

## 4.5 Qualitätsmanager

Um die Einhaltung der in diesem Dokument definierten Standards zu gewährleisten, setzen wir ein *Qualitätsmanagement* ein. Von den Qualitätsmanagern sind dabei zwei große Themenbereiche abzudecken.
Erstens ist dies die Git-Verwaltung und SCRUM-Umsetzung bei der Entwicklung, die Teilnahme an inkrementellen Code-Reviews sowie die Planung der Code-Reviews mit dem Team. Zweitens gehören die Definition und Durchführung von Integrations- und Systemtests und die Überwachung der Test-Coverage zu den Aufgaben. Dies übernehmen die Verantwortlichen dieses Qualitätssicherungskonzeptes, Arik Korte und Thomas Pause.
So wird den Qualitätsanforderungen dauerhaft Rechnung getragen, das Team hat Ansprechpartner für diese Themen und der kreative Entwicklungsprozess wird geringstmöglich beeinträchtigt.

# 5 Glossar

**Check-In** Unter Check-In im Kontext von Gruppen-Meetings versteht man, dass zu Beginn alle Team-Mitglieder die Chance haben kurz zu erläutern, wie es ihnen geht bzw. was sie beschäftigt, damit sie den Kopf frei bekommen und während der Besprechung sich alle anderen darüber im Klaren sind.

**Kanban-Board** Ein Kanban-Board ist eine Methode um Arbeitsprozesse zu visualisieren und zu planen. Dafür werden einzelne Aufgaben in Spalten sortiert, wobei jede Spalte für ein bestimmtes Stadium im Arbeitsprozess steht. Während eine Aufgabe verschiedene Stadien durchläuft, wandert auch ihr Eintrag auf dem Kanban-Board von Spalte zu Spalte.

**Test-Coverage** Test-Coverage bezeichnet das Verhältnis von durch Tests validierten Eigenschaften einer Software zu dem, was durch Tests validiert werden kann.

**Testpyramide** Die Testpyramide soll visualisieren wie sich die Anzahl von Testfällen staffeln soll: Die Basis bildet eine Vielzahl von Komponententests auf denen einige Integrationstests aufbauen, so dass schließlich nur noch wenige aufwändigere Systemtests notwendig sind.

**Mocking** Mocking bedeutet, dass Instanzen von Klassen nicht vollständig generiert werden, sondern nur bestimmte Teile davon simuliert werden. Dies ist hilfreich, wenn Schnittstellen zwischen verschiedenen Modulen getestet werden und ausgeschlossen werden soll, dass eventuelle Fehler durch unabhängigen Code innerhalb der entsprechenden Module auftreten.

**Pre-Commit Hook** Git-Hooks sind Skripte, welche von Git vor bestimmten Ereignissen ausgeführt werden. In diesem Falle wird ein Commit durch ein bestimmtes Skript überprüft und eventuell abgelehnt, bevor er von Git akzeptiert wird.

**Verdächtige Konstrukte** Hierunter fallen valide Konstrukte, welche aber vermutlich nicht repräsentieren, was der Implementierer beabsichtigt hat, wie z.B Verwendung von uninitialisierten Variablen, oder einer Zuweisungen wo ein Vergleich erwartet wurde.

## Quellenverzeichnis

[1] *Siehe Handreichung zum Praktikum.*

[2] *KDoc-Referenz.* URL: https://kotlinlang.org/docs/reference/kotlin-doc.html (besucht am 13. 12. 2019).

[3] *Kotlin Style Guide.* URL: https://kotlinlang.org/docs/reference/coding-conventions.html (besucht am 13. 12. 2019).

[4] *JUnit 5.* URL: https://junit.org/junit5/ (besucht am 13. 12. 2019).

[5] *Mockito.* URL: https://www.baeldung.com/mockito-junit-5-extension (besucht am 13. 12. 2019).

[6] *Gradle.* URL: https://docs.gradle.org/current/userguide/userguide.html (besucht am 13. 12. 2019).

[7] *Espresso.* URL: https://developer.android.com/training/testing/espresso/ (besucht am 13. 12. 2019).

[8] *Gitflow Workflow.* URL: https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow (besucht am 14. 12. 2019).