BERUFSAKADEMIE SACHSEN
BA SACHSEN
UNIVERSITY OF COOPERATIVE EDUCATION

Staatliche Studienakademie Dresden

– Studienrichtung Informationstechnik –

Möglichkeiten und Grenzen für die automatisierte Qualitätssicherung und Bereitstellung von Software am Beispiel einer iOS-Anwendung

Abschlussarbeit zur Erlangung
des staatlichen Abschlusses

Bachelor of Engineering (B.Eng.)

vorgelegt am 15. Juli 2022 von

Kai Ciesielski

Matrikelnummer: 3003857

Ausbildender Praxispartner:	Fraunhofer IVI Zeunerstraße 38, 01069 Dresden
Begutachtung Praxispartner:	Sebastian Koch, B.Sc.
Begutachtung BA Sachsen:	Dipl.-Softwaretechn. Thomas Nindel

Autorenreferat

Ciesielski, Kai: Möglichkeiten und Grenzen für die automatisierte Qualitätssicherung und Bereitstellung von Software am Beispiel einer iOS-Anwendung, Fraunhofer IVI, Berufsakademie Sachsen, Staatliche Studienakademie Dresden, Studiengang Informationstechnologie, Studienrichtung Informationstechnik, Bachelorthesis 2022.

57 Seiten, 22 Literaturquellen, 5 Anlagen (inkl. CD mit Quellcode)

Zur Verbesserung der Effizienz des Entwicklungsprozesses soll die Qualitätssicherung und Bereitstellung einer nativen iOS-Anwendung automatisiert werden. Dazu werden u.a. aus ITIL und dem BSI-Grundschutz-Kompendium allgemeine qualitätssichernde Maßnahmen zusammengetragen und deren Eignung für eine Automatisierung erläutert. Anhand eines Experteninterviews werden Anforderungen bestimmt, die als Grundlage für die Definition von Kriterien dienen. Diese werden schließlich zur Auswahl von Werkzeugen und zur Evaluierung einer exemplarischen Umsetzung herangezogen, wodurch anschließend eine fundierte Empfehlung ausgesprochen werden kann.
Allgemein sollten dynamische und statische Tests automatisiert durchgeführt werden. Dynamische Tests umfassen dabei u.a. Unit-, User-Interface-, End-to-End- und Regressions-Tests. Statische Tests umfassen die Prüfung der Einhaltung von Programmierrichtlinien, Schwachstellenanalysen auf Basis von Schwachstellendatenbanken, Secret-Detection, Dependency-Scanning und weiterer Codeanalysen, bspw. zur Beurteilung der Wartbarkeit oder sonstigen Qualitätsmerkmalen. Darüber hinaus sollten automatisiert Metriken erfasst und diese für ein Quality Gate genutzt werden, das zur Unterstützung von Code-Reviews dient. Analyseergebnisse und Metriken liefern allerdings keinen Beweis dafür, dass die Software frei von Sicherheitsmängeln ist.
In Verbindung mit GitLab stellt GitLab CI/CD die geeignetste Lösung für die Automatisierung dar. Für dynamische Tests eignet sich in Bezug auf die konkrete iOS-Anwendung das Testframework XCTest zum Erstellen und das Werkzeug Xcodebuild zum Ausführen. Für statische Tests stellt die Kombination aus SonarQube mit dem OWASP Dependency-Check-Plugin, SwiftLint, Mobsfscan und Gitleaks die geeignetste Kombination für den konkreten Anwendungsfall dar. Für die Bereitstellung empfiehlt sich Xcodebuild zum Exportieren der iOS-Anwendung, AsciiDoctor für Versionshinweise, 7-Zip für das Paketieren, Komprimieren und Verschlüsseln eines Release-Paketes sowie ein Python-Skript. Das Skript übernimmt die Erstellung des Release-Paketes und kommuniziert mit der unternehmenseigenen Nextcloud-Instanz. Die Auswahl der Werkzeuge basiert dabei auf der Erfüllung und Bewertung bestimmter Kriterien, wie bspw. dem Vorhandensein benötigter Funktionalitäten, Effizienz, Zuverlässigkeit oder Bedienbarkeit.
Die Automatisierung ist vor allem dann geeignet, wenn das Projekt eine gewisse Laufzeit aufweist, damit sich der anfänglich hohe Aufwand für die Umsetzung der Automatisierung amortisiert. Weiterhin sollte die Software als Gegenstand der Automatisierung einem gewissen Reifegrad entsprechen und über eine hohe Testabdeckung verfügen.

STUDIEREN IM MARKT

Name	**Kai Ciesielski**
Studiengang	**Informationstechnologie – Informationstechnik**
Seminargruppe	**3IT19 - 2**

Auftrag zur Anfertigung der Bachelorthesis

Hiermit erteilen wir Ihnen den Auftrag, folgendes Thema zu bearbeiten:

Möglichkeiten und Grenzen für die automatisierte Qualitätssicherung und Bereitstellung von Software am Beispiel einer iOS-Anwendung

1. Gutachter:	Sebastian Koch, B.Sc.
2. Gutachter:	Dipl.-Softwaretechn. Thomas Nindel

Dresden, 30.04.2022

Leiter des Studienganges Informationstechnologie

Inhaltsverzeichnis

Definitionen

Abbildungen

Tabellen

Quellcode

Gender Erklärung

In dieser Arbeit wird aus Gründen der besseren Lesbarkeit das generische Maskulinum verwendet. Weibliche und anderweitige Geschlechteridentitäten werden dabei ausdrücklich mitgemeint, soweit es für die Aussage erforderlich ist.

Teil I

Thesis

Kapitel 1

Einleitung

1.1 Motivation

Zur Verbesserung der Effizienz des Entwicklungsprozesses soll die Qualitätssicherung und Bereitstellung einer nativen iOS-Anwendung automatisiert werden. Automatisierung kann Kosten einsparen, menschliche Fehler bei repetitiven Aufgaben reduzieren und die Mitarbeitererfahrung verbessern. [Ebe21, S.268f] Das erklärt, warum Automatisierung heute in vielen Bereichen eingesetzt wird. In der Softwareentwicklung sind bspw. eben diese Prozesse der Qualitätssicherung und Bereitstellung dafür prädestiniert. Hier lassen sich standardisierte, werkzeug-unterstützte und sich wiederholende Arbeitsschritte finden, die sich für eine Automatisierung eignen.
Bei der Qualitätssicherung wird u.a. geprüft, ob die Software bestimmten Qualitätsanforderungen genügt. Dabei werden für jede Anforderung Tests und Ergebnisse festgelegt, die durchgeführt und erreicht werden müssen. Sind diese einmal definiert, ist deren Durchführung und Prüfung eine gleichbleibende und sich wiederholende Tätigkeit. Ähnlich verhält es sich bei dem Auslieferungsprozess einer Software. Immer gleiche Arbeitsschritte sorgen dafür, dass die Software den Endkunden zur Verfügung gestellt wird.

Für die generelle Automatisierung eines Prozesses wird eine Instanz benötigt, die die einzelnen Arbeitsschritte steuert, Ereignisse protokolliert und diese zur Verfügung stellt. Hierfür gibt es bereits existierende Automatisierungswerkzeuge, wie Jenkins[1] oder Fastlane[2]. Für eine automatisierte Qualitätssicherung und Bereitstellung gibt es ebenso eine Vielzahl an Maßnahmen und zugehörigen unterstützenden Werkzeugen, die unterschiedliche Anwendungsbereiche abdecken, bspw. zur Durchführung von UI-Tests oder für eine automatische statische Codeanalyse.
Für die Umsetzung der Automatisierung sollten infrage kommende Werkzeuge auf Basis der eigenen Anforderungen und Ziele evaluiert werden. [GM16, S.24] [Wie21, S.64] Besonders für die Qualitätssicherung sollten bereits im Vorfeld wichtige Testfälle und Fehlerarten identifiziert werden, die ein geeignetes Werkzeug als solche erkennt. [Wie21, S.60f] Andernfalls liefert es nicht den gewünschten Effekt. Vielmehr verursacht es aufgrund des falschen Gefühls der Sicherheit und des unentdeckten Fehlers weit mehr Kosten, als es einsparen sollte. Denn je später Fehler entdeckt werden, desto teurer ist deren Beseitigung.[3]

Zusammenfassend lässt sich sagen, dass sich ein Prozess nicht einfach nebenbei automatisieren lässt. Besonders für eine automatisierte Qualitätssicherung bedarf es einer gründlichen Analyse des Gegenstandes der Automatisierung und einer spezifischen Entscheidung über Zeitpunkt, Umfang und Methode. Dabei sind die jeweiligen Entwicklungsmodalitäten, die Eignung der Software an sich und dazugehörige (Qualitäts-)Anforderungen mit zu beachten. Für eine optimale Unterstützung sollten Werkzeuge sorgfältig ausgewählt und entsprechend evaluiert werden.

[1] Vgl. Jenkins: https://www.jenkins.io (Zugegriffen am 13. Juli 2022)
[2] Vgl. Fastlane: https://fastlane.tools (Zugegriffen am 13. Juli 2022)
[3] Verdeutlicht wird dieser Effekt durch die sogenannte 10er-Regel. Weitere Informationen sind unter folgendem Link zu finden: https://olev.de/0/10er-regl.htm (Zugegriffen am 13. Juli 2022)

1.2 Ziele der Arbeit

Das Ziel der Arbeit ist die Automatisierung ausgewählter Arbeitsschritte zur Qualitätssicherung und Bereitstellung einer iOS-Anwendung. Dazu zählt bspw. die Auswahl von Testumgebungen und der Prozess zur Bereitstellung der Anwendung für den Kunden. Zusätzlich sollen weitere mögliche Maßnahmen zur automatisierten Sicherstellung der Softwarequalität definiert werden. Im Rahmen der Arbeit sollen folgende Forschungsfragen beantwortet werden:

- Welche Maßnahmen können allgemein in der Automatisierung ergriffen werden, um die Softwarequalität einer iOS-Anwendung zu sichern?
- Welche konkreten Möglichkeiten und unterstützende Werkzeuge gibt es für eine automatisierte Qualitätssicherung und Bereitstellung einer iOS-Anwendung in Verbindung mit GitLab?
- Unter welchen Kriterien sollten in Frage kommende Möglichkeiten und Werkzeuge evaluiert werden?
- Ab wann ist Automatisierung (aus Sicht des Entwicklungsprozesses) effektiv und effizient? Unter welchen Voraussetzung kann eine automatisierte Sicherstellung der Softwarequalität als zuverlässig betrachtet werden?

Die Definition qualitätssichernder Maßnahmen bezieht sich dabei besonders auf jene, die sich in der Implementierungs- und Integrationsphase der Software anwenden lassen. Dabei werden insbesondere analytische Maßnahmen betrachtet, die Entwicklern nützliches Feedback liefern können. Auf generelle Vorgehensweisen zur Qualitätssicherung in der Anforderungsanalyse, im Design und Entwurf oder nach Veröffentlichung der Software soll nicht näher eingegangen werden.
Unter Bereitstellung wird im Rahmen der Arbeit das Verfahren verstanden, durch das die Möglichkeit besteht, die Anwendung den jeweiligen Kunden zur Verfügung zu stellen. Sie umfasst nicht die tatsächliche und sofortige Veröffentlichung der Software für den Kunden.

1.3 Vorgehen und Gliederung

Die Arbeit ist wie folgt gegliedert: Zunächst wird grundlegend geklärt, was iOS-Anwendungen sind und welche Besonderheiten deren Entwicklung mit sich bringt. Dazu wird auf iOS, die Entwicklungsumgebung Xcode, Configuration-Files und den Code-Signing-Prozess eingegangen.
Um die Frage nach qualitätssichernden Maßnahmen beantworten zu können, wird in der Folge ein einheitliches Bild von Softwarequalität und Qualitätssicherung erarbeitet. Das beinhaltet auch die Vorstellung wesentlicher Metriken, die zur Messung von Qualitätsmerkmalen dienen. Anschließend werden allgemein empfohlene Maßnahmen zur Qualitätssicherung herausgearbeitet und dargestellt. Danach wird auf Automatisierung in der Softwareentwicklung eingegangen. Das umfasst u.a. die Nennung dafür nötiger Voraussetzungen und grundlegende technische Aspekte zu deren Umsetzung. Es wird aufgezeigt, welche Arbeitsschritte im Rahmen der Qualitätssicherung und Bereitstellung automatisiert werden können und wie Automatisierung im Ganzen in den Entwicklungsprozess eingebunden werden kann. Darauf folgt die Vorstellung existierender Werkzeuge, die eine Automatisierung überhaupt erst möglich machen. Insbesondere werden unterstützende Werkzeuge für die automatisierte Durchführung von Tests vorgestellt.
Für die konkrete Umsetzung der Automatisierung sowie zur Auswahl geeigneter Werkzeuge wird ein Experteninterview vorbereitet und durchgeführt, dass zur Identifizierung der Anforderungen dient. Diese werden anschließend für die Definition wichtiger Kriterien verwendet, anhand derer infrage kommende Werkzeuge evaluiert werden sollen. Dafür wird ein Konzept für eine mögliche Umsetzung der automatisierten Qualitätssicherung und Bereitstellung erarbeitet. Darauffolgend werden geeignet erscheinende Werkzeuge für eine exemplarische Umsetzung des Konzeptes herangezogen. Mit Hilfe einer Beispielanwendung werden anschließend die verschiedenen Werkzeuge unter Beachtung der Kriterien evaluiert. Abschließend soll eine Empfehlung für die Umsetzung einer automatisierten Qualitätssicherung und Bereitstellung einer iOS-Anwendung ausgesprochen werden. Zum Schluss wird eine Zusammenfassung formuliert und es werden Aussichten genannt.

Kapitel 2

Grundlagen

2.1 Ausgewählte Grundlagen für die Entwicklung einer Anwendung für iOS

2.1.1 iOS und Xcode Konzepte

iOS bezeichnet das Betriebssystem von iPhones und iPads. Es wird herausgegeben und entwickelt von Apple. Native Anwendungen für diese Systeme werden iOS-Anwendungen genannt. Diese werden heute üblicherweise mittels der Programmiersprache Swift entwickelt. Weiterhin benötigt es für die Entwicklung, vor allem für die Veröffentlichung, von iOS-Anwendungen Hard- und Software von Apple. D.h. einen Mac und die Entwicklungsumgebung (IDE) Xcode. Mit deren Hilfe kann eine iOS-Anwendung gebaut, ausgeführt, getestet, analysiert sowie archiviert und exportiert werden.

Zur Unterstützung der Entwicklung setzt Xcode verschiedene Konzepte[4] um, wie in Abbildung B.0.1 dargestellt. Zu Beginn der Entwicklung wird als erstes ein neues Xcode Project angelegt. Ein Project beinhaltet dabei alle Dateien, Ressourcen und Informationen die nötig sind, um ein oder mehrere Softwareprodukte zu bauen. Ein einzelnes Softwareprodukt wird durch ein Target definiert. Es enthält bestimmte Anweisungen und Einstellungen, um aus zugehörigen Ressourcen ein fertiges Softwareprodukt zu erzeugen. Die Einstellungen in ihrer Gesamtheit werden als Build-Configuration bezeichnet. Eine darin enthaltene Einstellmöglichkeit wird Build-Setting[5] genannt und spezifiziert, wie der Buildprozess ausgeführt werden soll. Standardmäßig gibt es vorkonfigurierte Debug- und Release-Configurations auf Project-Ebene, die für jedes Target übernommen und ggf. einzeln angepasst werden können.
In einem Project kann immer nur ein Target aktiv sein, was durch das jeweilige ausgewählte Xcode Scheme definiert wird. Ein Scheme bestimmt, was bei Build, Test usw. geschieht. D.h. es verweist auf vorhandene Targets und die jeweils anzuwendende Build-Configuration. Prinzipiell können beliebig viele Schemes angelegt werden.
Wenn ein Project weitere Ressourcen, wie externe Bibliotheken oder andere Projects, enthält, so werden diese in einem Xcode Workspace gebündelt. Externe Bibliotheken können bspw. über den Dependency-Manager CocoaPods[6] eingebunden werden. Mittlerweile gibt es auch den direkt in Xcode integrierten Swift Package Manager[7].

[4] Vgl. Xcode Concepts: https://developer.apple.com/library/archive/featuredarticles/XcodeConcepts(Zugegriffen am 13. Juli 2022)
[5] Eine Liste mit allen Build-Settings ist unter https://developer.apple.com/documentation/xcode/build-settings-reference zu finden (Zugegriffen am 13. Juli 2022)
[6] Vgl. CocoaPods: https://cocoapods.org (Zugegriffen am 13. Juli 2022)
[7] Vgl. Swift Package Manager: https://www.swift.org/package-manager/ (Zugegriffen am 13. Juli 2022)

2.1.2 Konfiguration mit Build-Configuration-Files

Die in Unterabschnitt 2.1.1 erwähnten Build-Settings lassen sich direkt in Xcode bearbeiten.[8] Eine weitere Möglichkeit, um Build-Settings effektiv und einfach, auch außerhalb von Xcode, zu verwalten, bieten Build-Configuration-Files[9]. Mittels einfacher Schlüssel-Wert-Paare lassen sich vordefinierte Build-Settings überschreiben oder erweitern und vieles mehr. Weiterhin können Konstanten als Umgebungsvariablen für verschiedene Configurations definiert werden, bspw. um jeweils eine Backend-URL für Test- und Produktivumgebung zu nutzen. So können nicht wartungsfreundliche Anweisungen im Quellcode, wie in Quellcode 2.1.1, vermieden werden. Außerdem sollten ohnehin Konfigurationen vom Quellcode getrennt sein.[10]

Quellcode 2.1.1: Unterscheidung der Backend-URL im Quellcode mittels Compilerdirektiven

```
1 #if DEBUG
2 let backendBaseURL = URL(string:
      "https://https://testbackend.example.de")!
3 #else
4 let backendBaseURL = URL(string: "https://backend.example.de")!
5 #endif
```

Stattdessen könnte in jeweils einem Build-Configuration-File für Debug und Release eine Umgebungsvariable mit der entsprechende URL definiert sein. Dazu benötigt es eine Zeile der Form: `BACKEND_BASE_URL = https://backend.example.de`. Der Zugriff im Quellcode würde vereinfacht wie folgt aussehen.

Quellcode 2.1.2: Backend-URL als Umgebungsvariable

```
1 let backendBaseURL = Bundle.main.object(forInfoDictionaryKey:
      "BACKEND_BASE_URL") as! String
```

2.1.3 Code Signing und Provisioning Profiles

iOS-Anwendungen können über den iOS App Store oder außerhalb dessen auf iOS-Geräten installiert werden.[11] Damit diese auch ausgeführt werden können, müssen bestimmte Schritte während der Bereitstellung einer iOS-Anwendung beachtet werden. So muss der gesamte ausführbare Code mit einem Zertifikat des Entwicklers, das von Apple ausgestellt wird, signiert werden.[12] Das tut Apple, um sicherzustellen, dass alle Anwendungen aus einer bekannten und genehmigten Quelle stammen und nicht manipuliert wurden.[13] Für ein Zertifikat wird ein Apple Developer Account und die Mitgliedschaft in einem Entwicklerprogramm[14] von Apple benötigt. Zur Auswahl steht u.a. das Apple Developer Enterprise Program, das speziell für Unternehmen gedacht ist, um proprietäre oder intern genutzte Anwendungen zu entwickeln und bereitzustellen. Mit Hilfe des Developer Accounts können dann die jeweilige Anwendung bei Apple registriert und entsprechende Zertifikate für die Entwicklung und den Vertrieb erstellt werden.

[8] Vgl. „Configuring the Build Settings of a Target“: https://developer.apple.com/documentation/xcode/configuring-the-build-settings-of-a-target(Zugegriffen am 13. Juli 2022)
[9] Vgl. „Adding a Build Configuration File to Your Project.“: https://developer.apple.com/documentation/xcode/adding-a-build-configuration-file-to-your-project (Zugegriffen am 13. Juli 2022)
[10] Vgl. „Store config in the environment“: https://12factor.net/config (Zugegriffen am 13. Juli 2022)
[11] Vgl. „Preparing Your App for Distribution“: https://developer.apple.com/documentation/xcode/preparing-your-app-for-distribution (Zugegriffen am 13. Juli 2022)
[12] Vgl. „Codesignierung“: https://developer.apple.com/de/support/code-signing/ (Zugegriffen am 13. Juli 2022)
[13] Vgl. „App code signing process in iOS and iPadOS“: https://support.apple.com/de-de/guide/security/sec7c917bf14/web (Zugegriffen am 13. Juli 2022)
[14] Vgl. Apples Entwicklerprogramme: https://developer.apple.com/programs/ (Zugegriffen am 13. Juli 2022)

Weiterhin wird geprüft, ob die Anwendung überhaupt auf dem jeweiligen Gerät ausgeführt werden darf und, wenn ja, welche Berechtigungen die Anwendung besitzt. Dies wird mit sogenannten Provisioning Profiles[15] (dt: Bereitstellungsprofilen) gelöst. Diese werden ebenfalls im Developer Portal erstellt und von Apple signiert. Ein Provisioning Profile ist dabei ein Systemprofil, dass zum Starten von Anwendungen und Nutzen bestimmter Services auf einem oder mehreren Geräten gebraucht wird. Technisch gesehen handelt es sich um eine Datei mit einer Liste von Eigenschaften, die als *Property-List* oder *plist* bezeichnet wird. Ein Provisioning Profile enthält die folgenden Informationen:

- Verweis auf Zertifikate, die für das Signieren der Anwendung genutzt werden dürfen.
- Berechtigungen der Anwendungen für die Nutzung bestimmter Services.
- App-ID für Identifikation der Anwendung.
- Liste mit Geräten, die für die Ausführung der Anwendung berechtigt sind.
- Ablaufdatum des Profils. (meist nach einem Jahr)

Abhängig von dem ausgewählten Entwicklerprogramm gibt es unterschiedliche Auslieferungsmethoden[16] mit jeweils eigenen Provisioning Profiles und zu verwendenden Zertifikaten:

- **Development** für die Verteilung der Anwendung an Mitglieder des Entwicklungsteams mit eingetragenen Geräten.
- **App Store Connect** für den Vertrieb im App Store. Ausführung auf jedem Gerät möglich.
- **Ad Hoc** für die Installation der Anwendung auf eingetragenen Geräten.
- **Enterprise** für den Vertrieb im Unternehmen. Ausführung auf jedem Gerät möglich.

Bevor eine Anwendung auf einem Gerät ausgeführt werden kann, wird die Signatur sowie das Profil der Anwendung überprüft. Der Benutzer muss zusätzlich explizit bestätigen, dass er der Institution vertraut, die die Anwendung signiert hat.[17] Wenn alles korrekt ist, dann kann die Anwendung normal verwendet werden, ansonsten nicht. Eine Ausnahme von der Prüfung und der manuellen Vertrauensbestätigung gibt es bei Anwendungen, die aus dem App Store geladen wurden. Bei dem Verfahren zur Bereitstellung der Anwendung für den App Store wird diese erneut durch Apple signiert, wodurch jedes Gerät automatisch der Anwendung vertraut.[15]

Provisioning Profiles müssen jedes Mal manuell erneuert und heruntergeladen werden, wenn ein neues Gerät hinzugefügt wird, ein Zertifikat oder das Profil selbst abläuft. Das ist zeitaufwendig und immer dasselbe. Bei großen Entwicklerteams hat meist jedes Teammitglied ein eigenes Zertifikat, wodurch das Problem noch verstärkt wird. Eine Lösung dafür stellt der Codesigning-Guide bereit, welcher ein Best-Practice-Leitfaden zur Verwaltung von Zertifikaten und Provisioning Profiles in Entwicklungsteams sein soll.[18] In diesem wird vorgeschlagen, nur eine Code-Signatur-Identität für ein Team zu besitzen und diese über ein privates Git-Repository zu teilen und zu verwalten.

[15] Vgl. „TN3125: Inside Code Signing: Provisioning Profiles“: https://developer.apple.com/documentation/technotes/tn3125-inside-code-signing-provisioning-profiles (Zugegriffen am 13. Juli 2022)
[16] Vgl. „Distribution methods“: https://help.apple.com/xcode/mac/current/#/dev31de635e5 (Zugegriffen am 13. Juli 2022)
[17] Vgl. „Manually trust a developer on iOS “: https://help.apple.com/xcode/mac/current/#/dev96a12fb84 (Zugegriffen am 13. Juli 2022)
[18] Vgl. „A new approach to code signing“: https://codesigning.guide (Zugegriffen am 13. Juli 2022)

2.2 Qualitätssicherung in der Softwareentwicklung

2.2.1 Qualität von Software

Softwaresysteme sind immateriell, beliebig komplex und werden in den unterschiedlichsten Bereichen eingesetzt. Das führt dazu, dass der Begriff der Softwarequalität vielfältig ausgelegt und verwendet werden kann. Der ISO/IEC/IEEE-Standard 24765 definiert den generellen Begriff der Softwarequalität wie folgt:

Definition 2.2.1 (Softwarequalität):

„degree to which a software product satisfies stated and implied needs when used under specified conditions“ [ISO17, S.424]

Softwarequalität beschreibt also die Fähigkeit eines Systems, die Anforderungen von Kunden oder Benutzern zu erfüllen. Für die Beurteilung dieser werden je nach Anforderung verschiedene Qualitätsmerkmale herangezogen, die quantitativ oder qualitativ mit menschlichen oder automatisierten Mitteln gemessen werden können. Die wesentlichsten Merkmale wurden dabei im ISO/IEC-Standard 25010 in Kategorien zusammengefasst und der Produkt- oder Nutzungsqualität zugeordnet, wie in Abbildung 2.2.2 dargestellt. [ISO11, S.3f]

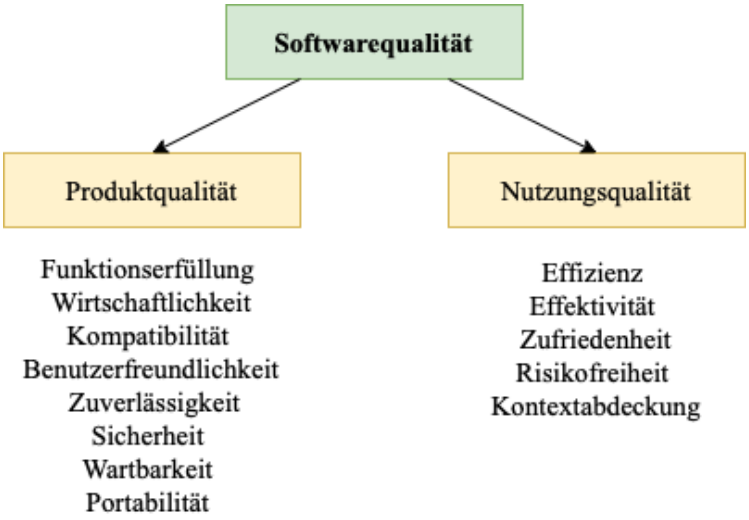

Abbildung 2.2.2: Kategorien der Produkt- und Nutzungsqualität von Software nach ISO/IEC 25010 (eigene Darstellung)

Die **Produktqualität** betrifft laut Standard das Softwareprodukt als solches und beinhaltet folgende Kategorien und Merkmale: [ISO11, S.4]

- Die **Funktionserfüllung** (Effektivität) beschreibt gegebene Funktionalitäten, d.h. Vollständigkeit, Korrektheit und Angemessenheit der Funktionen.
- Mit **Wirtschaftlichkeit** (Effizienz) ist das Zeitverhalten, die Ressourcennutzung und Leistungsfähigkeit des Softwaresystems gemeint.
- **Kompatibilität** beschreibt die Koexistenz und Interoperabilität mit anderen Systemen.
- **Benutzerfreundlichkeit** umfasst eine intuitive Bedienbarkeit, eine zielführende Bedienung, ästhetisches Aussehen, Fehlerbehandlung und vieles mehr.
- **Zuverlässigkeit** ist die Wahrscheinlichkeit, mit der das System die Anforderungen erfüllt. Das umfasst die Verfügbarkeit, Fehlertoleranz und Wiederherstellungsmöglichkeiten.

- **Sicherheit** beinhaltet die Merkmale Vertraulichkeit, Integrität, Zurechenbarkeit und Authentizität.
- Unter **Wartbarkeit** ist die Modularität, Wiederverwendbarkeit einzelner Komponenten, Erweiterbarkeit, Änderbarkeit und Testbarkeit zu verstehen.
- **Portabilität** umfasst Merkmale zur Installation, Übertragung auf andere Systeme und Ersetzbarkeit durch andere Systeme.

Mittels der **Nutzungsqualität** werden laut Standard jene Auswirkungen beschrieben, die das Softwareprodukt auf alle Beteiligten hat. Sie beinhaltet folgende Kategorien und Merkmale: [ISO11, S.3]

- **Effizienz** setzt die aufgewendeten Ressourcen für die Zielerreichung in Bezug zur eigentlichen Zielerreichung.
- Die **Effektivität** beschreibt, wie zielführend die Software arbeitet.
- Merkmale von **Zufriedenheit** sind die Nützlichkeit und persönliche Einstellung (Freunde, Vertrauen, ...) gegenüber der Software.
- **Risikofreiheit** drückt sich durch ein vermindertes Risiko in wirtschaftlichen, gesundheitlichen oder sicherheitstechnischen Bereichen aus.
- **Kontextabdeckung** umfasst die Flexibilität und Kontextvollständigkeit.

Je nach Einsatzbereich und Anforderungen können die aufgeführten wesentlichen Merkmale mehr oder weniger stark ausgeprägt sein. Außerdem können auch Neue hinzukommen.
Im Allgemeinen haben verschiedene Faktoren Einfluss auf die Softwarequalität. Dazu gehören der Entwicklungsprozess, verwendete Technologien, beteiligte Personen sowie finanzielle und zeitliche Rahmenbedingungen. [BK21, S.63] Darüber hinaus können sich bestimmte Qualitätsmerkmale auch gegenseitig beeinflussen. [Hof13, S.11f]

2.2.2 Metriken

Die in Unterabschnitt 2.2.1 vorgestellten Qualitätsmerkmale dienen vor allem als Unterstützung zur Identifikation von geforderten Eigenschaften. Für die Sicherstellung deren Erreichung müssen quantitative Messungen durchgeführt werden. Denn nur wer etwas misst, kann schließlich auch vergleichen, verfolgen und damit verbessern. Eine pauschale Aussage wie „das System ist zuverlässig“ ist bspw. wenig aussagekräftig. Zur Bekräftigung solcher Aussagen werden Kennzahlen und Maßfunktionen, sogenannte Metriken, herangezogen.

Definition 2.2.2 (Metrik):

„**1.** quantitative measure of the degree to which an item possesses a given quality attribute **2.** function whose inputs are software data and whose output is a single numerical value that can be interpreted as the degree to which the software possesses a given quality attribute“ [ISO17, S.363]

Heute existiert eine Vielzahl an Metriken für die unterschiedlichsten Anwendungszwecke. So lassen sich bspw. Umfang, Komplexität, Effizienz oder Wartbarkeit durch Metriken erfassen. Einige der bekanntesten werden im Folgenden kurz vorgestellt:

- **Lines of Code** (LoC): Als klassisches Umfangsmaß dienend werden hierbei schlicht alle Code-Zeilen aufaddiert. Je nach Programmierstil und -sprache unterscheiden sich die Zahlen jedoch erheblich voneinander, was sie nur unzureichend vergleichbar macht. Nach Hoffmann könne sie jedoch als eine Eingangsgröße für komplexere Metriken miteinbezogen werden. So zum Beispiel zur Berechnung des Code-Umfangs, für den ein Entwickler die Verantwortung trägt. Daraus könne dann abgeleitet werden, wie viel Zeit dieser für Wartung und Pflege der Codebasis benötige, die bspw. nicht für Neuentwicklungen zur Verfügung stünde. [Hof13, S.249ff]
- **Zyklomatische Komplexität** (McCabe-Metrik): Diese Metrik dient zur Messung der Komplexität von Software, speziell von Funktionen. Zugrunde liegt die Annahme, dass mit steigender Komplexität das Verständnis von Menschen abnimmt. Dies wirkt sich unmittelbar

auf die Wart- und Testbarkeit der Software aus. Die zyklomatische Komplexität kann dabei mittels Verzweigungen (also if-else- oder switch-case-Strukturen) oder Knoten und Kanten (im Kontrollflussgraph) gemessen werden. Eine Zahl kleiner 10 entspräche dabei nach McCabe selbst einer annehmbaren Komplexität. Andernfalls müsse die Funktion zerlegt werden. [Hof13, S.261]
Einfache aber vielfältige switch-case-Strukturen können die Zahl allerdings schnell in die Höhe treiben, obwohl der Quellcode einfach zu verstehen wäre. So lasse die Zahl für Broy lediglich eine Aussage über den Testaufwand zu, da die Anzahl der möglichen Pfade bestimmt wird. [BK21, S.68f]

- **Halstead-Metriken**: Die Halstead-Metriken umfassen mehrere aufeinander aufbauende Metriken zur Vermessung des Umfangs, Aufwandes oder Schwierigkeit (Komplexität) einer Software. Als Grundlage dienen die Anzahl der unterschiedlichen verwendeten Operatoren (Kontrollstrukturen und Schlüsselwörter der Programmiersprache) und Operanten (Variablen- und Funktionsbezeichner, Konstanten) sowie deren jeweilige Gesamtanzahl. Aufgrund ihrer Einfachheit werden die Metriken heute häufig in Werkzeugen zur statischen Codeanalyse eingesetzt. [Hof13, S.251ff]
- **Testabdeckung**: Um sicherzustellen, dass alle wesentlichen Bestandteile einer Software getestet wurden, wird die Testabdeckung gemessen. Für diesen Zweck gibt es verschiedene Überdeckungsmetriken. Die Funktionsabdeckung gibt bspw. das Verhältnis der durch Testfälle ausgeführten Funktionen zu allen existierenden Funktionen an. Weitere Überdeckungsmetriken sind die Anweisungs-, Zweig-, Pfad- und Bedingungsüberdeckung. [BK21, S.73f]
- **MTBF**: Als ein Maß für die Zuverlässigkeit wird die „Mean Time Between Failure“ herangezogen. Im einfachsten Fall ist sie der Quotient aus gesamter Betriebszeit und Anzahl der Ausfälle. Es existieren noch weitere Formeln, die auch Reparaturzeiten mit berücksichtigen, die entstehen, wenn Fehler behoben werden. [Lig09, S.263]

Metriken können den Entwicklungsprozess und die Qualitätssicherung sinnvoll unterstützen. So können bereits frühzeitig suboptimale Entwicklungen erkannt und ggf. Gegenmaßnahmen ergriffen werden. Broy und Liggesmeyer weisen allerdings darauf hin, dass die bloße Erfassung von Metriken des Erfassens und Messens wegen eher kontraproduktiv sei. Die Messwerte lieferten an sich keine hinreichende Aussage, vielmehr müssten diese systematisch ausgewertet und interpretiert werden. [BK21, S.61ff] [Lig09, S.266f] So ist bspw. ein hoher Anteil an Kommentarzeilen ein Indiz für gute Dokumentation aber es kann nicht gemessen werden, ob die Kommentare auch wirklich sinnvoll und hilfreich sind. Vor der Verwendung einer Metrik sollte nach Broy immer deren Objektivität, Genauigkeit, Aussagekraft, Vergleichbarkeit und Nützlichkeit geprüft werden. Die Verwendung solle dabei immer einem konkreten Zweck dienen, da sonst mehr Aufwand als Nutzen entstünde. [BK21, ebenda] D.h., dass die Auswahl nützlicher Metriken immer von den konkreten Anforderungen abhängt. Für die Identifikation von Metriken empfiehlt sich das *Goal-Question-Metric*-Verfahren (GQM) [KB22, S.766ff].

2.2.3 Maßnahmen für die Qualitätssicherung

Um sicherzustellen, dass festgelegte Qualitätsmerkmale die gewünschten Ausprägungsgrade erreichen, sollte eine Qualitätssicherung durchgeführt werden.

Definition 2.2.3 (Qualitätssicherung):

„**1.** part of quality management focused on providing confidence that quality requirements will be fulfilled **2.** all the planned and systematic activities implemented within the quality system, and demonstrated as needed, to provide adequate confidence that an entity will fulfill requirements for quality“ [ISO17, S.360f]

Qualitätssicherung umfasst laut Definition also alle geplanten und systematischen Tätigkeiten, die dafür sorgen, dass bestimmte Qualitätsanforderungen erreicht werden. Letztendlich beginnt Qualitätssicherung dementsprechend schon bei der Wahl des Vorgehens für die Softwareentwicklung. Denn auch die Qualität des Entwicklungsprozesses und dessen Rahmenbedingungen können sich auf

die Produkt- und Nutzungsqualität des Softwaresystems auswirken. [ISO11, S.27] [HKL+20, S.10] Während der Implementierung und Integration von Software besitzt das **Testen** einen großen Stellenwert. Grundsätzlich wird dabei laut ISO/IEC/IEEE 29119-2-Standard zwischen statischen und dynamischen Tests unterschieden. [ISO21, S.21] Bei statischen Tests wird die Software anhand einer Reihe von (Qualitäts-)Kriterien untersucht, ohne diese auszuführen. Das trifft bei Code-Reviews oder statischen Codeanalysen zu. Dynamische Tests hingegen setzen eine Ausführung der Software voraus. Hierbei wird zwischen Testebenen (bspw. Unit-, Integration-, System-Tests) und Testarten (bspw. UI-, Leistungs-, Sicherheits-Tests) unterschieden. Je nach Projekt, Team und Anforderungen kommen die unterschiedlichsten Methoden in verschiedenen Kombinationen und Variationen vor.

Generell empfiehlt der ISO/IEC/IEEE 29119-2-Standard die Erarbeitung einer Teststrategie auf Basis der eigenen Anforderungen und identifizierten Risiken. Diese sollte in der Regel statische und dynamische Tests beinhalten. [ISO21, S.21] ITIL 4[19] verweist in seinen Practices auf funktionale und nicht-funktionale Tests, die ausgeführt werden sollten, um sicherzustellen, dass eine entwickelte Lösung die gewünschten Funktionen aufweise. [Ebe21, S.396] Das Bundesamt für Sicherheit in der Informationstechnik (BSI) identifiziert in seinen IT-Grundschutz-Bausteinen[20] für Softwareentwicklung und Softwaretests allgemeine Gefährdungen und gibt praktische Empfehlungen zum generellen Vorgehen in diesen Bereichen. Darunter, dass so früh wie möglich entwicklungsbegleitende Tests durchgeführt werden sollten. [BSI22, CON.8.A7] Weiterhin hat Bitkom[21] einen Leitfaden „Zur Sicherheit softwarebasierter Produkte“ veröffentlicht, der einen allgemeinen Überblick zur Entwicklung und dem Betrieb softwarebasierter Produkte geben soll. [HKL+20] Die von den verschiedenen Institutionen vorgeschlagenen Maßnahmen werden im Folgenden zusammengefasst dargestellt.

Es wird allgemein empfohlen dynamische Tests durchzuführen. [HKL+20, S.30] [ISO21, S.21] Dazu sollte in erster Linie eine eigene Testumgebung vorhanden sein, die getrennt von der Produktivumgebung und deren Daten sei. [BSI22, OPS.1.1.6.A13] Zu den üblichen durchzuführenden Tests gehören **Unit-, Integration- und System-Tests**. [Ebe21, S.396] Unit-Tests testen dabei einzelne kleine Komponenten einer Software. Das kann eine Funktion oder auch eine Klasse sein. Integrationstests sollen sicherstellen, dass eine Gruppe zusammengehöriger Komponenten oder Module fehlerfrei zusammenarbeitet. Systemtests testen, dem Namen entsprechend, ein ganzes Softwaresystem. Eine wichtige Testart sind UI-Tests, bei denen geprüft wird, ob sich die grafische Benutzeroberfläche wie gewünscht verhält. Zur Abnahme werden schließlich Akzeptanztests oder auch Ende-zu-Ende-Tests (E2E-Tests) durchgeführt, um die Software ganzheitliche zu testen. Wenn es sich um eine Client-Server-Anwendung handelt, dann wird dabei auch deren Zusammenspiel geprüft.
Um sicherzustellen, dass bei Weiterentwicklungen oder Veränderungen bereits vorhandene Funktionalitäten nicht beeinträchtigt worden sind, sollten **Regressionstests** durchgeführt werden. [BSI22, OPS.1.1.6.A12] [Ebe21, S.396] Das bedeutet, dass bspw. die aufgeführten Komponenten- und Integrationstests nicht einfach verworfen werden, wenn die Entwicklung der Funktionalität abgeschlossen ist, sondern, dass diese Tests behalten und regelmäßig mit durchgeführt werden.
Bei Anwendungen mit erhöhtem Schutzbedarf sollten darüber hinaus noch sogenannte **Penetrationstests** durchgeführt werden.[22] [BSI22, OPS.1.1.6.A14] [Ebe21, S.396] Damit lässt sich prüfen, inwieweit eingesetzte Sicherheitsmaßnahmen verschiedenen Bedrohungen standhalten.
Für die Überprüfung der Schnelligkeit und Kapazität unter Last sollten Performance- und Kapazitätstests durchgeführt werden. [Ebe21, S.396] Das ist vor allem bei Softwaresystemen von Bedeutung, die bestimmte Reaktionszeiten aufweisen oder hohe Nutzerzahlen aushalten müssen.

[19] „ITIL stellt einen umfassenden und allgemein verfügbaren Best-Practice-Leitfaden für das IT-Service-Management dar. Die dort niedergeschriebenen Erfahrungen und Empfehlungen haben sich als ITIL Best Practices mittlerweile zum De-facto-Standard entwickelt und bewährt.“ [Ebe21, S.4]

[20] Das BSI veröffentlicht mit dem IT-Grundschutz-Kompendium jährlich Empfehlungen zu dem Thema Informationssicherheit in den verschiedensten Bereichen. Das zentrale Element sind die Grundschutz-Bausteine, die gewisse Anwendungsgebiete abdecken. Darunter sind auch Bausteine für die Softwareentwicklung und Softwaretests.

[21] Bitkom ist ein Branchenverband deutscher Unternehmen, die vor allem in den Bereichen Software und IT-Services sowie Telekommunikations- und Internetdiensten tätig sind. Weitere Informationen: https://www.bitkom.org. (Zugegriffen am 13. Juli 2022)

[22] Für die Durchführung von Penetrationstests empfiehlt sich das Durchführungskonzept für Penetrationstests des BSI, siehe https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Publikationen/Studien/Penetrationstest/penetrationstest.pdf?__blob=publicationFile&v=3. (Zugegriffen am 13. Juli 2022)

Im Bitkom-Leitfaden wird abschließend noch auf das sogenannte **Fuzzing oder Fuzz-Testing** als effektive Methode hingewiesen. [HKL+20, S.30ff] Dabei werden viele zufällige Testeingaben generiert, mit denen die Software getestet wird, um Fehler zu erzeugen. [WC13, S.3ff]

Neben den dynamischen Tests sollten auch statische Tests bzw. Codeanalysen durchgeführt werden. Für die Sicherstellung der Wartbarkeit sollte eine allgemeine verbindliche **Programmierrichtlinie** erstellt werden. [BSI22, CON.8.A11] [HKL+20, S.30ff] [Ebe21, S.396] Diese sollte Namenskonventionen beinhalten sowie bestimmte Stile und Elemente vorschreiben, die (nicht) verwendet werden dürfen. Allgemein akzeptierte Swift Style Guidelines sind beispielsweise die von Google[23] oder Ray Wenderlich[24]. Deren Einhaltung sollte mit Hilfe einer automatischen statischen Codeanalyse überprüft werden. [HKL+20, S.31]
In Bezug auf die Sicherheit sollten mit Hilfe geeigneter Werkzeuge **Schwachstellenanalysen** durchgeführt werden, die auf Schwachstellendatenbanken (CWE[25], CVE[26], OWASP[27]) basieren. [HKL+20, S.30ff] [Ebe21, S.396] Daneben sollten sich Passwörter und Zugangsdaten (Secrets) nicht in ausgeliefertem Quellcode oder Konfigurationsdateien befinden. Spätestens im Rahmen des Auslieferungsprozesses sollte dementsprechend eine **Secret-Detection** durchgeführt werden. [BSI22, CON.8.A5] Bei der Verwendung von externen Komponenten (Bibliotheken) sollte eine ähnliche Qualitätssicherung durchgeführt werden, wie für selbst entwickelte Software. [HKL+20, S.10] Sie müssen aus vertrauenswürdigen Quellen stammen und deren Integrität geprüft werden. Darüber hinaus sollte sichergestellt sein, dass keine veralteten Versionen genutzt werden und durch die Nutzung nicht gegen Lizenzbestimmungen verstoßen wird. Sie müssten außerdem auf Schwachstellen und potentielle Konflikte hin untersucht werden. [BSI22, CON.8.A20] Die aufgeführten Maßnahmen werden unter dem Begriff **Dependency-Scanning** zusammengefasst.
Abschließend sollte noch geprüft werden, ob der Quellcode ausreichend kommentiert wurde, sodass andere Entwickler mit Hilfe der **Kommentare** den Quellcode verstehen und dessen Funktionalität nachvollziehen können. [BSI22, CON.8.A12]

Zusätzlich zu den Testmethoden und -verfahren sollten Ergebnisse ausführlich dokumentiert und entsprechend ausgewertet werden. [BSI22, OPS.1.1.6.A3] Hierzu sollten **Metriken**, bspw. zur Komplexitätsmessung und Testabdeckung, verwendet werden [HKL+20], um Fortschritte sichtbar und die Qualität messbar zu machen. Wenn alle Tests erfolgreich waren und die Ergebnisse mit den vorher festgelegten Erwartungen übereinstimmen, dürfe die Software freigegeben werden. [BSI22, OPS.1.1.6.A4] Dieser Übergang werde auch als **Quality-Gate** bezeichnet, das bspw. durch ein manuelles Review realisiert werden könne. [McC09, S.467]

Zusammenfassend lässt sich sagen, dass eine Vielzahl verschiedener Maßnahmen existiert. Einige von ihnen, wie Unit- und Regressionstests, die Festlegung von Programmierrichtlinien sowie eine statische Codeanalyse werden prinzipiell immer empfohlen. Der generelle Einsatz sollte allerdings immer in Übereinstimmung mit den eignen Anforderungen und Zielen erfolgen. Es lässt sich nicht die eine Methode finden, die alle Fehler aufdeckt. Nach McConnell sei es vielmehr die Kombination aus mehreren Maßnahmen und Techniken. [McC09, S.469ff]

[23] Vgl. Google Swift Style Guide: https://google.github.io/swift/ (Zugegriffen am 13. Juli 2022)
[24] Vgl. Ray Wenderlich's Swift Style Guide: https://github.com/raywenderlich/swift-style-guide (Zugegriffen am 13. Juli 2022)
[25] Vgl. Common Weakness Enumeration: https://cwe.mitre.org (Zugegriffen am 13. Juli 2022)
[26] Vgl. Common Vulnerabilities and Exposures: https://cve.mitre.org (Zugegriffen am 13. Juli 2022)
[27] Vgl. OWASP Top Ten: https://owasp.org/www-project-top-ten/ (Zugegriffen am 13. Juli 2022)

2.3 Automatisierung in der Softwareentwicklung

2.3.1 Automatisierung und deren Voraussetzungen

Wie bereits in der Einleitung erarbeitet, ist Automatisierung das Mittel zur Wahl, um menschlichen Fehler bei repetitiven Tätigkeiten zu vermeiden. Automatisierung bezeichnet dabei ganz generell einen Prozess, der automatisch und ohne menschliches Zutun abläuft. [ISO17, S.37]
ITIL 4 führt das *Optimieren und Automatisieren* als eines seiner universellen und beständigen Grundprinzipien auf, die eine Institution in allen Situation leiten soll. Darin heißt es, dass jegliche Arten der Verschwendung[28] vermieden werden sollten. Dazu sollte dort, wo es möglich ist, Technologie und Automation als Unterstützung eingesetzt werden. Manuelle Aktivitäten sollten, soweit möglich, ausschließlich auf wertschöpfende Arbeiten reduziert werden. [Ebe21, S.225]
Bevor jedoch automatisiert werden kann, sollten folgende Empfehlungen beachtet werden: [Ebe21, S.268f]

- Zu automatisierende Prozesse sollten so weit wie möglich vereinfacht und optimiert worden sein. Geeignet seien besonders standardisierte und sich wiederholende Tätigkeiten.
- Es sollten geeignete ergebnis- und wertorientierte Metriken identifiziert und definiert worden sein. Anhand dieser sollte die Effizienz der Automatisierung bewertet werden.
- Alle anderen Grundprinzipien sollten bereits vorher angewandt worden sein. D.h. u.a., dass bereits existierende Technologien als Einsatzmöglichkeit herangezogen werden sollten, um Kosten zu sparen. Technologien sollten dabei jedoch niemals unreflektiert und um ihrer selbst willen eingesetzt werden. Es bedürfe einer genauen Analyse und Evaluierung. Abschließend solle die Automatisierung schrittweise eingeführt sowie auf Einfachheit und Praktikabilität geachtet werden. Das führe zu einem schnelleren und sichtbaren Fortschritt.

Automatisierung erfordere weiterhin ein gewisses Know-How, sowohl von Entwicklern als auch von den Unternehmen, die diese einführen wollen. Besonderen Wert solle auf die Menschen und deren Akzeptanz bei der Einführung gelegt werden. Es brauche Menschenkenntnisse und andere Softskills, um Menschen von jener Automatisierung zu überzeugen, die ganze Arbeitsplätze ablösen könne. [GM16, S.24f]

2.3.2 Prinzipielle technische Umsetzung

Für die Automatisierung wird grundsätzlich eine Instanz benötigt, die definierte Anweisungen selbstständig in einer bestimmten Reihenfolge ausführt, Ereignisse protokolliert und entsprechend Feedback bereitstellt. Generell kann dies lokal auf einem System oder auf einem Server geschehen. Das hängt von dem jeweiligen Anwendungszweck der Automatisierung ab.
Im Prinzip könnte also ein einfaches Skript ausreichen, dass sequentiell die einzelnen Befehle ausführt und Rückmeldung in die Kommandozeile schreibt. Zur besseren Bedienung sowie mehr oder spezifischeren Anwendungs- und Konfigurationsmöglichkeiten wurde dieses Prinzip weiterentwickelt und als Grundlage für Automatisierungslösungen verwendet. Diese umfassen bspw. grafische Benutzeroberflächen zur einfachen Konfiguration oder Netzwerkfähigkeiten, um Befehle auch auf entfernten Systemen ausführen zu können. Zusätzlich werden Artefakte, also Dateien usw., die bei der Ausführung von Befehlen entstanden sind, gesichert und zugänglich gemacht. Teilweise werden auch verwaltungstechnische Aspekte, wie eine Benutzerverwaltung, und vieles mehr, integriert.
Die einzelnen Arbeitsschritte und Regeln können meist über eine grafische Benutzeroberfläche oder textbasiert mittels unterschiedlicher Sprachen (bspw. YAML, Kotlin, Groovy) definiert werden. Die Datei dient als Vorlage für die Erzeugung einer Instanz mit konkreten Arbeitsschritten und Abfolgen. Diese wird als *Pipeline* bezeichnet. In der Dokumentation von Jenkins[29], einem führenden Automatisierungsserver, wird die Definition einer Pipeline via textbasierte Datei empfohlen, die mit in ein Versionsverwaltungssystem aufgenommen werden kann. Das ermögliche Entwicklerteams diese leicht mittels einer IDE zu adaptieren und schließlich durch ein Review zu kontrollieren. Außerdem können komplexe Abläufe mit vielen Arbeitsschritten und Regeln nur schwer mittels grafischer

[28] Gemeint sind hierbei die sieben Arten der Verschwendung nach Lean. [Ebe21, S.150f]
[29] Vgl. „What is Jenkins Pipeline?“: https://www.jenkins.io/doc/book/pipeline/ (Zugegriffen am 13. Juli 2022)

Benutzeroberfläche umgesetzt werden. Das Prinzip nennt sich auch *Pipeline-as-Code*. Zur besseren Übersicht werden ähnliche oder zusammenhängende Arbeitsschritte einer Pipeline meist in sogenannten Stages gruppiert.
Um festzustellen, ob ein Arbeitsschritt korrekt ausgeführt wurde und alle Ergebnisse den Erwartungen entsprechen, werden die Exit-Codes der einzelnen Befehle verwendet. Bei Null ist alles in Ordnung und die Automatisierung fährt fort. Alle weiteren Codes entsprechen je nach Befehl bestimmten Fehlermeldungen, was zu einem Abbruch der Automatisierung führen kann, wenn gewünscht.

2.3.3 Möglichkeiten und Grenzen für Automatisierung im Entwicklungsprozess

Wie aus Unterabschnitt 2.3.1 hervorgeht, sind besonders standardisierte und sich wiederholende Tätigkeiten für eine Automatisierung geeignet. Im folgenden Kapitel soll herausgearbeitet werden, welche konkreten Arbeitsschritte das im Bereich der Qualitätssicherung und Bereitstellung sind. Außerdem wird aufgeführt, wie und unter welchen Bedingungen sich eine solche Automatisierung in den generellen Entwicklungsprozess integrieren lässt.

Bei dem **Build-Prozess** einer iOS-Anwendung muss immer explizit auswählt werden, welches Scheme oder Target für welche Plattform gebaut werden soll. Außerdem ist noch die Auswahl eines Provisioning Profiles nötig. Das Vorgehen ist dabei immer und für jedes Scheme oder Target gleich und eignet sich somit für eine Automatisierung.
Gleichbleibende Abläufe lassen sich auch bei **dynamischen und statischen Tests** finden: Programmierrichtlinien bleiben in der Regel die meiste Zeit unverändert. Die Überprüfung deren Einhaltung in einem manuellen Review ist ein stures Vorgehen nach Checkliste. Ob Variablen mindestens drei Zeichen lang sind, kann dabei ebenso gut durch ein Werkzeuge mit definierten Regeln automatisch überprüft werden. Gleiches gilt für die Suche nach bekannten Schwachstellen in der eigenen Software und verwendeten externen Komponenten. Zeitintensiv wäre auch die manuelle Kontrolle, ob eine veraltete Version einer externen Bibliothek genutzt wird, ob sich noch Secrets im Quellcode befinden oder ob Lizenzbestimmungen externer Komponenten verletzt werden. Automatisch arbeitende Werkzeuge können hier Abhilfe schaffen. Gleiches gilt auch für das Fuzzing. Des Weiteren müssen für dynamische Tests die Testumgebung (inkl. Testdaten) und Betriebssystemversion eingestellt sowie die entsprechenden Tests durchgeführt werden. Bei mehreren unterstützten Geräten und Versionen ein langer, sich wiederholender und gleichbleibender Prozess. Wenn bei jeder Änderung Regressionstests durchgeführt werden sollen, dann erhöht sich der manuelle Aufwand nochmals. Besonders bei Regressionstests sei eine Automatisierung daher effizient, wie Garousi darstellt. [GM16, S.23]
Abschließend sollten auch **Penetrationstests** durchgeführt werden. Diese können jedoch nicht einfach automatisiert werden. Dafür bedarf es intensiver Vorbereitung und Wissen. Hierzu kann mit externen Partnern zusammengearbeitet werden, die sich auf solche Tests spezialisiert haben.

Auch **Metriken** werden nach einem immer gleichen Prinzip gemessen, das sich mit Hilfe verschiedener Werkzeuge automatisieren lässt. Die einzelnen Metriken könnten in ihrer Gesamtheit für ein automatisches **Quality Gate** genutzt werden. Wie in Unterabschnitt 2.2.2 beschrieben, liefern Metriken allerdings keine hinreichende Aussage, sondern nur erste Indizien.
Des Weiteren sind auch Ergebnisse einer statischen Codeanalyse kein Beweis dafür, dass der Code frei von Sicherheitsmängeln ist, wie Wiegenstein in dem Artikel „Kaffeesatzleserei: Ein kritischer Blick auf statische Codeanalyse-Verfahren“ beschreibt. Sie würden sich sehr gut eignen, um unsauberen Code oder Regelverstöße zu erkennen. Darüber hinaus, besonders in Bezug auf die Erkennung von Schwachstellen und Sicherheitslücken, seien die Ergebnisse der Werkzeuge allerdings mit Vorsicht zu genießen. Dies sei darin begründet, dass „Anwender sich sichere, schnelle, präzise und korrekte Ergebnisse wünschen, was Anbieter dazu verleitet, weniger Ergebnisse anzuzeigen, um falsche Alarme zu vermeiden“. Anhand eigener Beispiele begründet Wiegenstein weiter, dass „es schon einfache Vorgehensweisen erlauben, absichtlich eingefügten, schadhaften Code vor der statischen Analyse zu verbergen“. Es zieht das Fazit, dass diese Werkzeuge „nicht dazu entwickelt wurden, um vorsätzlich implementierte oder gar getarnte Sicherheitsprobleme zu erkennen“. Als einzige Alternative sieht Wiegenstein manuelle Code-Reviews, denn nur diese „profitieren von der

analytischen Denkweise und der Intuition erfahren Auditorinnen und Auditoren“. [Wie, S.96ff]
Die Entscheidung, ob die Software bestimmten Qualitätsanforderungen genügt, sollte dementsprechend nicht einzig und allein auf Metriken und Automatismen beruhen. Je nach Art und Umfang der Änderung/Erweiterung sollte im Einzelfall manuell entschieden werden. Die Ergebnisse eines automatischen Quality Gates bzw. die dafür als Grundlage genutzten Metriken und Ergebnisse von Codeanalysen liefern dafür einen ersten Hinweis.

Abschließend lassen sich im **Bereitstellungsprozess** einer Software Möglichkeiten für Automatisierung finden. Dieser erfolgt standardisiert mit genau definierten Arbeitsschritten. Im Falle von iOS wird die Anwendung mittels Xcode archiviert, exportiert und schließlich auf bestimmtem Wege verteilt - entweder in den App Store hochgeladen oder anderweitig.

Wie soeben herausgearbeitet gibt es viele Möglichkeiten für eine voneinander unabhängige Automatisierung der einzelnen Arbeitsschritte. Da Build, Test und Bereitstellung aber aufeinanderfolgen, gibt es auch Möglichkeiten diese in einem gesamten Prozess zu automatisieren. Ein oder mehrere definierte Auslöser (Trigger), bspw. ein Push von Änderungen in ein zentrales Repository eines Versionsverwaltungssystem, sorgen dafür, dass die Software **voll- oder teilautomatisch** gebaut, getestet und schließlich bereitgestellt oder sogar in einer Produktivumgebung installiert wird. Falls der Build oder ein Test fehlschlägt, wird die Pipeline abgebrochen und die Ursache muss beseitigt werden.
Auf diesem Prinzip basieren die heute weit verbreiteten Konzepte von Continuous Integration (CI) sowie Continuous Delivery und Continuous Deployment (beides abgekürzt mit CD). M. Fowler beschreibt CI als ein Vorgehen, bei dem Teammitglieder ihre Arbeit regelmäßig, mindestens jedoch einmal am Tag, in genau einen Hauptbranch integrieren sollten. Genauso regelmäßig sollten auch die Arbeiten andere Mitarbeiter in die eigene Arbeit integriert werden. Dabei werde jede Integration durch automatisches Bauen und Testen überprüft. Schlage das Bauen oder ein Test fehl, so werde unmittelbar der jeweilige Entwickler benachrichtigt. Bei CI/CD gehe es vor allem darum schnell Feedback zu bekommen, um Fehler frühestmöglich erkennen und beheben zu können.[30] Continuous Delivery setzt CI voraus und bezeichnet nach Fowler ein Vorgehen, bei dem Software immer in einem auslieferbaren Zustand gehalten werde, sodass jederzeit die Möglichkeit zum Ausliefern bestehe. Diese müsse dabei allerdings manuell gestartet werden. Bei Continuous Deployment werde die Auslieferung vollautomatisch durchgeführt.[31] Eine schematische Darstellung, wie eine CI/CD-Pipeline aufgebaut sein kann, wird in Abbildung B.0.2 dargestellt.

Dank technischer Unterstützung kann die Automatisierung in vielen Bereichen der Softwareentwicklung eingesetzt werden. Doch nur, weil es funktionieren würde, heißt das nicht, dass es auch getan werden sollte. Garousi und Mäntylä stellen fest, dass sich die Software an sich generell dafür eignen müsse. D.h., dass sie sich bereits in einem stabilen Zustand und nicht in der Anfangsphase der Entwicklung befinden sollte. Sonst würden die Kosten für Testanpassungen die eigentlichen Vorteile zunichte machen. Dies treffe besonders auf UI-Tests zu. Die Software müsse einen bestimmten Reifegrad erreicht haben, da sonst Designänderungen zu erhöhtem Aufwand führen, der aufgebracht werden müsse, um Tests anzupassen. Weiterhin solle die Software eine lange Lebensdauer aufweisen. [GM16, S.22f] Für kurz laufende Projekte ergibt eine Automatisierung dementsprechend nur wenig Sinn, da sich der anfängliche hohe Aufwand erst nach einer gewissen Zeit amortisiert. [GM16, S.25]
Einiges könne nach McConnell auch gar nicht erst automatisiert werden. Entwickler müssten sich auch immer mit der realen Welt auseinandersetzen, mit Menschen, die nicht immer logisch denken, nicht immer wissen, was sie wollen, mit Vorschriften und Gesetzen, die außerhalb der IT-Welt entstehen aber Einfluss auf diese haben. Sie müssten außerdem über Abläufe und Algorithmen nachdenken. Das alles brauche Menschen, die analysieren, Lösungen für spezielle Probleme finden, um eine Brücke zwischen der realen Welt und der IT zu bauen. So etwas könne nicht automatisiert werden. [McC09, S.723]

[30] Vgl. „Continuous Integration“: https://martinfowler.com/articles/continuousIntegration.html (Zugegriffen am 13. Juli 2022)
[31] Vgl. „Continuous Delivery“: https://martinfowler.com/bliki/ContinuousDelivery.html (Zugegriffen am 13. Juli 2022)

2.3.4 Auswahl der richtigen Werkzeuge als Unterstützung

Prinzipiell kann jedes Werkzeug, das sich mit Hilfe von Befehlen über die Kommandozeile ausführen lässt, auch von einer Automatisierung genutzt werden. Von großer Bedeutung ist jedoch, ob und wie das Werkzeug Rückmeldung gibt. Ohne eine Auskunft, dass es seine Arbeit verrichtet hat und welche Ergebnisse dabei erreicht wurden, kann auch die Automatisierungslösung keine weiteren Auskünfte geben oder Schritte ableiten. Aus diesem Grund werden sich die in Unterabschnitt 2.3.2 erwähnten Exit-Codes zu Nutze gemacht. Wichtig dabei ist, dass in den Exit-Code auch die Ergebnisse des (Test-)Werkzeuges mit einfließen und nicht nur, dass das Werkzeug an sich korrekt, ohne internen Fehler, gearbeitet hat. Sonst könnte die Automatisierung annehmen, dass keine Fehler gefunden wurden, obwohl nur die Programmabarbeitung gemeint war. Ein Testautomatisierungswerkzeug kann bspw. alle Tests durchführen und hat somit korrekt gearbeitet. Ob ein Test fehlgeschlagen ist, wird daraus aber nicht klar. Dies sollte sich im Exit-Code widerspiegeln, damit die Automatisierung Fehler auch als solche erkennen und melden kann.

Für eine Testautomatisierung spielt u.a. auch die richtige Auswahl eines Testautomatisierungswerkzeuges eine wesentliche Rolle. Garousi und Mäntylä sind durch eine umfassende Literaturrecherche auf folgende wichtige Erkenntnisse gestoßen, die in eine diesbezügliche Entscheidungsfindung mit einfließen sollten. Nach den Autoren sei es wichtig, dass das Testwerkzeug generell für die Software geeignet ist. Es müsse verstanden werden, wie Testwerkzeug und Software miteinander interagieren. Weiterhin sollte das Geschäftsmodell des Werkzeugherstellers hinterfragt und geprüft werden, sodass zukünftige Entwicklungen und daraus resultierende Risiken abgeschätzt werden könnten. Besonders empfehle sich die Nutzung von Open-Source-Werkzeugen, da diese geringere Kosten, weniger Risiken und eine bessere Zukunftsfähigkeit aufweisen. [GM16, S.24] Diese Kriterien können dabei auch allgemein auf allgemeine Automatisierungswerkzeuge angewandt werden.
Garousi, Mäntylä und Raulamo-Jurvanen untersuchten durch eine weitere Literaturrecherche explizit, welche Kriterien von Praktikern für die Auswahl eines geeigneten Testautomatisierungswerkzeuges genutzt werden. Sie führen die folgenden, zu berücksichtigenden, Kriterien auf. [RJMG17, S.8]

- Entwicklungsteam und -umgebung
  - Erfahrungen und Fähigkeiten des Entwicklungsteams
  - Eignung für Arbeitsumgebung (Kompatibilität mit anderen Werkzeugen)
  - Erfüllung der Testanforderungen *(am häufigsten genannt)*
- Externe Merkmale des Werkzeuges
  - Herstellerbewertung
  - Support
  - Kosten / Lizenzgebühren *(am häufigsten genannt)*
- Technische Merkmale des Werkzeuges
  - Funktionen zum Aufzeichnen und Wiedergeben
  - Unterstütze Skriptsprachen
  - Vielseitigkeit / Anpassbarkeit / Wartbarkeit
  - Benutzerfreundlichkeit / Verwendbarkeit (Usability) *(am häufigsten genannt)*
  - Einbindung von Testdaten
  - Reportingfunktionen (Testberichte) *(am zweithäufigsten genannt)*
  - Andere Merkmale (bspw. nur lokale, selbst-gehostete Werkzeuge)

Die genannten Kriterien lassen sich auch in einem Heise iX-Artikel von Gegendorfer, Grötz und Zax über Testautomatisierungswerkzeuge für mobile Anwendungen wiederfinden. Besonders bei Open-Source-Werkzeugen sei es wichtig, dass die Entwicklung stabil ist und aktiv vorangetrieben wird. Das biete den Vorteil, dass neueste technologische Entwicklungen schnell eingebunden und genutzt werden können. [GGZ, S.110]

2.4 Werkzeuge für die automatisierte Qualitätssicherung und Bereitstellung einer iOS-Anwendung

2.4.1 Überblick

Für eine Automatisierung gibt es viele Systeme, Plattformen und Werkzeuge. Diese sorgen für eine korrekte Abarbeitung der Pipeline, empfangen Feedback und verwalten entstandene Artefakte. Für die einzelnen Arbeitsschritte können wiederum andere unterstützende Werkzeuge von der Automatisierung genutzt werden, um bspw. dynamische oder statische Tests durchzuführen.
In den folgenden Kapiteln sollen allgemeine Automatisierungswerkzeuge, Testautomatisierungswerkzeuge und weitere unterstützende Werkzeuge vorgestellt werden, die für eine automatisierte Qualitätssicherung und Bereitstellung einer iOS-Anwendung infrage kommen könnten. Die Liste vorhandener Werkzeuge am Markt ist groß, wie die Heise iX-Artikel „Komplettpaket für Code: Systeme für Continuous Integration und Continuous Delivery im Vergleich“ [GKFLM] oder „Quelltext im Fokus: Werkzeuge zur statischen Codeanalyse“ [Wie21] verdeutlichen. Eine vollumfängliche Aufzählung und Vorstellung aller Werkzeuge ist weder möglich noch zielführend und würde den Rahmen der Arbeit übersteigen. Deswegen wird sich im Folgenden auf Werkzeuge konzentriert, die sich bereits in ihrem jeweiligen Bereich etabliert haben und explizit für eine Automatisierung sowie für iOS-Projekte (inkl. Xcode) geeignet sind. Wenn möglich, soll vor allem Open-Source-Software betrachtet werden, die aktiv weiterentwickelt wird.

Hinweis: Viele Werkzeuge enthalten CI/CD in ihren Namen, was möglicherweise etwas irreführend ist, wenn die Definitionen und Erläuterungen von M. Fowler bekannt sind. Denn technisch gesehen handelte es sich dabei lediglich um eine Automatisierungslösung, die als Grundlage für CI/CD genutzt werden kann aber nicht muss. Automatisierung geht auch ohne das CI betrieben wird. Diese Diskrepanz könnte damit erklärt werden, dass in der Praxis verschiedene Definition und Vorstellungen von CI existieren.

2.4.2 Allgemeine Automatisierungswerkzeuge

Xcode Server und Xcode Cloud

Xcode Server[32] ist eine Automatisierungslösung von Apple. Das zentrale Element sind sogenannte Bots, die für ein Projekt über Xcode auf einem dezidierten Xcode Server erstellt werden. Diese dienen dazu, um automatisiert (nach jedem Commit, Ausführungsplan, manuell) Codeänderungen aus einem Repository zu laden und anschließend definierte Aktionen auszuführen. Diese umfassen das Bauen, Analysieren, Testen und, wenn aktiviert, das Archivieren und Exportieren. Für die automatische Durchführung von dynamischen Tests werden definierte Simulatoren und physischen Geräte verwendet. Die Ergebnisse werden dann wieder zurück an die Xcode IDE geliefert. Statistiken und Artefakte können über ein Webinterface eingesehen und heruntergeladen werden.

Xcode Cloud[33] ist eine Automatisierungsplattform von Apple in der Cloud. Apple selbst bezeichnet es als CI/CD-Service mit dessen Hilfe sich die Praktiken von Continuous Integration und Continuous Delivery umsetzen lassen. Es dient dazu, automatisch Anwendungen zu bauen, zu testen und zu verteilen (via TestFlight[34] und App Store Connect[35]). Die Nutzung ist kostenpflichtig.

Jenkins

Jenkins[1] ist der wohl bekannteste Automatisierungsserver für das Bauen, Deployen und das generelle Automatisieren von Entwicklungsprozessen. Die Software ist Open-Source, basiert auf Java und

[32] Vgl. „About Continuous Integration in Xcode“: https://developer.apple.com/library/archive/documentation/IDEs/Conceptual/xcode_guide-continuous_integration/index.html (Zugegriffen am 13. Juli 2022)
[33] Vgl. Xcode Cloud: https://developer.apple.com/xcode-cloud/ (Zugegriffen am 13. Juli 2022)
[34] Vgl. Beta Testing Made Simple with TestFlight: https://developer.apple.com/testflight/ (Zugegriffen am 13. Juli 2022)
[35] Vgl. App Store Connect: https://developer.apple.com/app-store-connect/ (Zugegriffen am 13. Juli 2022)

lässt sich mittels einer Vielzahl von Plugins[36] für nahezu jeden Anwendungszweck konfigurieren und nutzen. Jenkins muss selbst-gehostet werden und kann auf den bekanntesten Plattformen (Windows, macOS, Linux) installiert werden.
Pipelines können über das Webinterface oder als Code in einem *Jenkinsfile*[37] definiert werden. Für das *Jenkinsfile* gibt es *Declarative Pipelines* oder *Scripted Pipelines* die mittels einer domänenspezifischen Sprache beschrieben werden. *Scripted Pipelines* basieren auf einer limitierten Syntax von Groovy und können komplexere Logiken und Regeln enthalten. Eine deklarative Pipeline ist der einfachere, zu bevorzugende Weg und kann wie folgt aussehen.

Quellcode 2.4.1: Jenkinsfile (deklarative Pipeline)

```
pipeline {
    agent any
    stages {
        stage('Build') {
            steps {
                echo "This step builds something."
            }
        }
    }
}
```

Die einzelnen *Steps* werden dabei auf sogenannten *Nodes* ausgeführt, die bspw. ein macOS- oder Linux-System sein können. Die einzelnen Befehle an sich werden auf dem System von einem *Agent* ausgeführt und gesteuert. Diese sind mit dem Jenkins Controller verknüpft, der die generelle Befehlsausführung steuert und überwacht.[38]

GitLab CI/CD

GitLab CI/CD[39] ist ein Werkzeug, dass in GitLab[40] integriert ist und alle Arbeitsschritte von Build über Test bis Deployment automatisieren kann. GitLab selbst ist eine vollumfängliche Softwareentwicklungsplattform auf Open-Source-Basis mit einem Webinterface. Es umfasst eine integrierte Versionskontrolle, Aufgabenverwaltung, Code-Review-Funktionalitäten und vieles mehr. Dabei kann es selbst gehostet oder als Service genutzt werden.
Neben der einfachen Automatisierung bietet GitLab CI/CD noch viele weitere unterstützende Features an. Über Templates können Jobs zur Qualitätssicherung[41], Secret-Detection[42] und Schwachstellenanalyse bzw. Security-Tests[43] eingebunden werden. Zusätzlich bietet GitLab ein Dependency-Scanning[44] an. Testberichte werden in Merge-Requests (als Kommentar oder über spezielle Widgets) zur Verfügung gestellt, damit das Feedback zur richtigen Zeit, am richtigen Ort und der richtigen Person zur Verfügung steht. Für Deployments stehen verschiedene Mechanismen bereit, um bestimmte Funktionen oder Änderungen für ausgewählte Nutzergruppen zugänglich zu machen.

[36] Vgl. Jenkins-Plugins: https://plugins.jenkins.io (Zugegriffen am 13. Juli 2022)
[37] Vgl. Using a Jenkinsfile: https://www.jenkins.io/doc/book/pipeline/jenkinsfile/ (Zugegriffen am 13. Juli 2022)
[38] Vgl. Managing Nodes: https://www.jenkins.io/doc/book/managing/nodes/ (Zugegriffen am 13. Juli 2022)
[39] Vgl. „Continuous Integration and Delivery“: https://about.gitlab.com/features/continuous-integration/ (Zugegriffen am 13. Juli 2022)
[40] Vgl. GitLab: https://about.gitlab.com (Zugegriffen am 13. Juli 2022)
[41] Vgl. Code Quality: https://docs.gitlab.com/ee/ci/testing/code_quality.html (Zugegriffen am 13. Juli 2022)
[42] Vgl. Secret Detection: https://docs.gitlab.com/ee/user/application_security/secret_detection/ (Zugegriffen am 13. Juli 2022)
[43] Vgl. Static Application Security Testing: https://docs.gitlab.com/ee/user/application_security/sast/ (Zugegriffen am 13. Juli 2022)
[44] Vgl. Dependency Scanning: https://docs.gitlab.com/ee/user/application_security/dependency_scanning/ (Zugegriffen am 13. Juli 2022)

Pipelines inkl. Trigger werden als Code in YAML-Syntax in einer `.gitlab-ci.yml`-Datei definiert, die GitLab automatisch erkennt und nutzt. Ein Beispiel ist in Quellcode 2.4.2 dargestellt.

Quellcode 2.4.2: gitlab-ci.yml-File

```
stages:
    - build
    - test
    - deliver

build_job:
    stage: build
    script:
        - echo "This job builds something."
```

Die einzelnen Arbeitsschritte in der Pipeline werden als *Jobs* bezeichnet. Die darin enthaltenen Skripte bzw. Befehle werden mittels sogenannten GitLab-Runnern ausgeführt. Diese stellen das Äquivalent zu den Agents in Jenkins dar.
Generell können Pipelines für verschiedene Szenarien verwendet werden: Für einen einfachen Commit, Merge-Request oder als Merge-Result-Pipeline, um zu prüfen, wie ein Merge verlaufen würde. Weiterhin können verschiedene vordefinierte Templates, bspw. für die Qualitätssicherung, eingebunden werden.

Einen großen Vorteil bietet GitLab CI/CD, da es direkt in GitLab integriert ist und damit viele Bereiche abgedeckt werden. Das minimiert Lizenzkosten und sorgt für eine steile Lernkurve. Grundlegend steht GitLab CI/CD in allen Editionen frei zur Verfügung aber es gibt einige Extrafunktionen, die kostenpflichtig sind[45] Darunter das Dependency-Scanning, License-Compliance oder Performance-Tests. Außerdem ist die Darstellung einiger Testberichte, wie die der Security-Tests, ebenfalls erst in einer kostenpflichtigen Edition verfügbar.

Fastlane

Fastlane[2] ist ein Open-Source-Werkzeug speziell für den mobilen Bereich und unterstützt vor allem iOS aber auch Android. Es ist keine ganze Plattform mit Webinterface sondern speziell für die Kommandozeile gedacht, um lokal eine Automatisierung bestimmter Prozesse bereitzustellen, wenn der Nutzer diese benötigt. Dafür werden sogenannten *Lanes* definiert, in denen nacheinander verschiedene *Actions*[46] ausgeführt werden. Diese ermöglichen bspw. ein automatisches Bauen und Testen der Anwendung. Die einzelnen Lanes werden dabei in einem *Fastfile* definiert, wie in Quellcode 2.4.3 beispielhaft dargestellt. Die jeweiligen Actions haben ihre Konfigurationen entweder in gesonderten Dateien oder können direkt als Parameter bei dem Aufruf in der Lane mit übergeben werden.

Quellcode 2.4.3: Lane in einem Fastfile

```
lane :release do
    build_app
    run_tests
    increment_version_number
    upload_to_app_store
end
```

[45] Einen Überblick über GitLab CI/CD, Konzepte, Konfigurationsmöglichkeiten und Feature ist hier zu finden: https://docs.gitlab.com/ee/ci/. (Zugegriffen am 13. Juli 2022)

[46] Eine Liste mit allen Actions ist hier zu finden: https://docs.fastlane.tools/actions/ (Zugegriffen am 13. Juli 2022)

Die Automatisierung bzw. Abarbeitung der Lane wird in der Kommandozeile mit `fastlane release` im jeweiligen Projektordner mit Fastfile gestartet. Die große Anzahl an bereits vorhanden Actions und die Möglichkeit, dass auch eigene Actions erstellt werden können machen das Werkzeug sehr vielseitig einsetzbar. Für eine automatisierte Ausführung einer Lane bei einem Push in ein Repository muss der jeweilige Befehl allerdings immer in eine Pipeline eines Automatisierungssystemes wie Jenkins eingebunden werden.

Weitere Automatisierungswerkzeuge

Neben Jenkins und GitLab CI/CD gibt es noch viele weitere Werkzeuge unterschiedlichster Hersteller, die eine Automatisierung und die Umsetzung von CI/CD-Konzepten möglich machen. Darunter viele proprietäre und cloud-basierte Werkzeuge. Alle haben gemeinsam, dass bestimmte Arbeitsschritt automatisiert werden können. Unterschiede gibt es vor allem im Bereich der grafischen Benutzeroberfläche und somit der Bedienung oder bei der Integration mit anderen Werkzeugen. Jeder Hersteller integriert dabei seine eigenen Werkzeuge bestmöglich, um eine durchgängige und geeignete Unterstützung für den gesamten Entwicklungsprozess zu gewährleisten. Werkzeuge von bekannten Herstellern sind bspw. TeamCity von JetBrains[47] und Bamboo von Atlassian[48]. Eine Übersicht inkl. Vergleich der bekanntesten Automatisierungs- bzw. CI/CD-Systeme ist dem Heise iX-Artikel „Komplettpaket für Code - Systeme für Continuous Integration und Continuous Delivery im Vergleich“ zu entnehmen [GKFLM].

2.4.3 Werkzeuge für dynamische Tests

XCTest

XCTest[49] ist ein von Apple entwickeltes Framework für Unit-, Integration-, UI- und Performance-Tests. Die Tests werden über spezielle Test-Targets organisiert, lassen sich in der Programmiersprache Swift schreiben und integrieren sich nahtlos in den Testworkflow von Xcode. Für die Erstellung von UI-Tests bietet Xcode eine Recording-Funktion an, mit deren Hilfe UI-Tests schnell und unkompliziert erstellt werden können.

EarlGrey 2.0

EarlGrey 2.0[50] ist ein Open-Source-Testframework, das auf XCTest aufsetzt und nativ mit Xcode zusammenarbeitet. Das Framework besitzt umfangreiche Möglichkeiten, um besser auf Netzwerkaktivitäten, Animationen usw. zu warten. Es überwacht direkt die UI und macht einen Pixel-per-Pixel-Vergleich von Screenshots, um sicherzustellen, dass UI-Element auch wirklich vorhanden sind. Das hat zur Folge, dass Benutzer mit genau diesen Elementen interagieren, mit denen auch EarlGrey interagiert. Für die Nutzung werden allerdings spezielle externe Bibliotheken benötigt.

Appium

Appium[51] ist eine Open-Source-Framework zur Testautomatisierung für mobile Anwendungen. Es unterstützt iOS, Android und Windows-Anwendungen sowohl in Simulatoren als auch auf physischen Geräten. Appium an sich ist ein einfacher HTTP-Server, der HTTP-Requests von einem Testskript bekommt und diese über einen speziellen Agent auf einem Simulator oder Gerät ausführt. Der Agent nimmt befehle des Appium-Servers entgegen und gibt diese an das plattformspezifische Testframework (bspw. XCTest) weiter. Ergebnisse werden wieder via HTTP an den Appium-Server übertragen. Appium abstrahiert somit die plattformspezifischen Testframeworks und macht es somit möglich, dass Tests plattformübergreifend in verschiedenen Programmiersprachen, wie Python oder Java, geschrieben werden können. Für Nutzung sind spezielle Bibliotheken erforderlich.

[47] Vgl. TeamCity: https://www.jetbrains.com/de-de/teamcity/ (Zugegriffen am 13. Juli 2022)
[48] Vgl. Bamboo: https://www.atlassian.com/de/software/bamboo (Zugegriffen am 13. Juli 2022)
[49] Vgl. XCTest: https://developer.apple.com/documentation/xctest (Zugegriffen am 13. Juli 2022)
[50] Vgl. EarlGrey 2.0: https://github.com/google/EarlGrey/tree/earlgrey2 (Zugegriffen am 13. Juli 2022)
[51] Vgl. Appium: https://appium.io (Zugegriffen am 13. Juli 2022)

Weitere Werkzeuge

Neben den vorgestellten Testframeworks existieren auch ganze Systeme und Services, die sich dieser Thematik widmen. Ein gute Übersicht liefert der Heise iX-Artikel „Es teste, wer sich ewig bindet - Testautomatisierungswerkzeuge für mobile Applikationen im Überblick“ [GGZ].

Ein nützliches Werkzeug ist noch Bluepill[52]. Es dient als Unterstützung, um iOS-Tests parallel auf mehreren Simulatoren auszuführen. Die Ausführung erfolgt dabei ohne Benutzung einer grafischen Benutzeroberfläche, um Speicherplatz einzusparen. Weiterhin kann es Simulatoren automatisch neu starten, wenn sie abgestürzt sind und Testreports erzeugen.

2.4.4 Werkzeuge für statische Tests

Xcode Analyze

Xcode kann mittels des eingebauten Clang Static Analyzers[53] automatisiert eine statische Codeanalyse durchführen, um Endlosschleifen, ungenutzten Code, Sicherheitsprobleme und logische Fehler aufzudecken. Die Analyse kann direkt in Xcode als Aktion ausgeführt werden und Entwicklern somit schnell Feedback liefern. [54] Allerdings lassen sich damit nur Fehler in jener Quellcode finden, der mit C, C++ oder Objective-C geschrieben wurde. Somit ist diese Funktionalität für Swift-Projekte unbrauchbar.

Code Climate CLI

Code Climate ist ein Unternehmen, dass sich u.a. auf die Qualitätssicherung von Software spezialisiert hat. Es stellt Entwicklern Werkzeuge zur Verfügung, um die Wartbarkeit und Sicherheit ihrer Software zu überprüfen und sicherzustellen. Neben deren kostenpflichtiger Code-Climate-Analysis-Plattform stellen sie auch ein Open-Source-Command-Line-Interface (Code Climate CLI)[55] bereit, welches lokal genutzt werden kann. Das Werkzeug basiert auf den Code Climate Engines, die über Docker Images bereitgestellt werden, und führt eine statische Codeanalyse durch - sogenannte Maintainability-Checks[56]. Hierfür werden u.a. zyklomatische Komplexität, kognitive Komplexität, Anzahl der Codezeilen pro Datei und Methode, Code-Duplizierung sowie Anzahl an Argumenten und Rückgabewerten pro Methode erfasst und ausgewertet. Über Plugins können weitere Checks und Analysen von anderen unterstützten Werkzeugen durchgeführt werden, darunter auch SwiftLint.

SwiftLint

SwiftLint[57] ist ein Werkzeug, dass die Einhaltung von allgemein anerkannten Swift-Programmierrichtlinien und -stilen auf Basis des Swift-Style-Guides von Ray Wenderlich[24] prüft. Es warnt bei Verstößen und kann bei Bedarf einige Stile durchsetzen indem ganze Dateien formatiert werden.
Das Prinzip basiert auf einer Menge von definierten Regeln[58], die es einzuhalten gilt. Das Werkzeug hat integrierte Standardregeln. Diese umfassen bspw. wie viele Zeilen ein Datei haben darf oder wie lang und komplex eine Methode werden darf. Außerdem können auch eigene, auf regulären Ausdrücken basierende, Regeln definiert werden.

[52] Vgl. Bluepill: https://github.com/MobileNativeFoundation/bluepill (Zugegriffen am 13. Juli 2022)
[53] Vgl. Clang Static Analyzer: https://clang-analyzer.llvm.org/index.html (Zugegriffen am 13. Juli 2022)
[54] Vgl. „Analyze your code for potential flaws“: https://help.apple.com/xcode/mac/11.4/#/devb7babe820 (Zugegriffen am 13. Juli 2022)
[55] Vgl. Code Climate CLI: https://docs.codeclimate.com/docs/command-line-interface und https://github.com/codeclimate/codeclimate (Zugegriffen am 13. Juli 2022)
[56] Vgl. Maintainability: https://docs.codeclimate.com/docs/maintainability (Zugegriffen am 13. Juli 2022)
[57] Vgl. SwiftLint: https://github.com/realm/SwiftLint (Zugegriffen am 13. Juli 2022)
[58] Eine Liste aller verfügbarer Regeln ist unter folgendem Link zu finden: https://realm.github.io/SwiftLint/rule-directory.html. (Zugegriffen am 13. Juli 2022)

SonarQube

SonarQube[59] ist ein Werkzeug von SonarSource, das Bugs, Schwachstellen und Code Smells[60] anhand definierter Regeln[61] aufspüren kann. Es berechnet zusätzlich die technische Schuld[62]. Dazu wird entdeckten Fehlern eine geschätzte Dauer zu deren Behebung zugeordnet und die Zeiten aufaddiert. So kann die Qualität von vorhandenem oder neu hinzugekommenen Code analysiert und überwacht werden. Dabei wird eine Vielzahl an Programmiersprachen unterstützt, darunter auch Swift.
Einen großen Stellenwert nimmt dabei das integrierte Quality Gate ein. Für dieses werden verschiedene Metriken[63] erhoben, die Interpretationen hinsichtlich Wartbarkeit, Sicherheit, Zuverlässigkeit usw. zulassen. Den konkreten Ausprägungen der Merkmale werden Buchstaben von A bis E zugeordnet, die der Bewertung dienen. A ist dabei das Beste, E das Schlechteste. Da die Qualität, d.h. die Ausprägungen bestimmter Merkmale von den eigenen Anforderungen abhängen, können eigene Quality Gates erstellt werden. SonarSource empfiehlt in den meisten Fällen allerdings die Nutzung des eigenen „sonar way“-Quality-Gates. Dies ist standardmäßig voreingestellt und aktiviert. Die darin verwendeten Bedingungen sind Abbildung B.0.3 zu entnehmen. Ein mögliches Ergebnis kann Abbildung B.0.5 entnommen werden.
Damit SonarQube alle relevanten Informationen für die Bewertung und somit das Quality Gate zur Verfügung stehen, müssen vor der SonarQube-Analyse sämtliche Arbeitsschritte ausgeführt und entsprechend protokolliert werden. Das betrifft bspw. die Durchführung von Unit-Tests und die Erzeugung von Testberichten. Die entstandenen Berichte können anschließend von SonarQube gescannt und deren Inhalte, bspw. die Testabdeckung, mit verarbeitet werden. Das Resultat kann dann über die Weboberfläche von SonarQube eingesehen werden, wie in Abbildung B.0.4 dargestellt.
Zur Verfolgung der Entwicklung erstellt SonarQube Statistiken in die alle Ergebnisse jedes einzelnen Scans mit einfließen. So kann bspw. eingesehen werden wie sich die Testabdeckung oder gefundene Probleme über die Zeit verändern.
SonarQube ist im Kern eine Open-Source-Software, die mit proprietären Funktionalitäten erweitert werden kann. Die Unterstützung für Swift erfolgt bspw. erst ab der kostenpflichtigen Developer Edition.

Mobsfscan

Das Werkzeug Mobsfscan[64] dient der statischen Codeanalyse von mobilen Anwendungen. Es unterstützt iOS, somit auch Swift, und ist auf die Durchführung von Sicherheitsscans spezialisiert, um Schwachstellen zu identifizieren. Dafür nutzt es die Analyse-Regeln des Mobile Security Frameworks[65] (MobSF). MobSF ist eine Open-Source-Software und wird für Penetrationstests, Malwareanalyse und Schwachstellenerkennung im mobilen Bereich (iOS und Android) eingesetzt. Dafür greift es auf Schwachstellendatenbanken von OWASP[27] und CWE[25] zurück.

Gitleaks

Gitleaks[66] ist ein Open-Source-Werkzeug, um in Git-Repositories oder sonstigen Dateien und Ordnern nach Passwörtern, API-Keys, Tokens usw. zu suchen. Für deren Aufdeckung werden reguläre Ausdrücke verwendet. Standardmäßig besitzt das Werkzeug viele verschiedene Secret-Typen und bietet außerdem die Möglichkeit, eigene Regeln mittels regulären Ausdrücken definieren zu können.

[59] Vgl. SonarQube: https://www.sonarqube.org (Zugegriffen am 13. Juli 2022)
[60] Vgl. „Code Smell“: https://martinfowler.com/bliki/CodeSmell.html (Zugegriffen am 13. Juli 2022)
[61] Vgl. „SonarSource static code analysis“: https://rules.sonarsource.com (Zugegriffen am 13. Juli 2022)
[62] Mit technischer Schuld werden die potentiellen Folgen schlechter technischer Umsetzung von Software bezeichnet. Weitere Informationen: https://martinfowler.com/bliki/TechnicalDebt.html. (Zugegriffen am 13. Juli 2022)
[63] Vgl. Metric Definitions: https://docs.sonarqube.org/latest/user-guide/metric-definitions/ (Zugegriffen am 13. Juli 2022)
[64] Vgl. Mobsfscan: https://github.com/MobSF/mobsfscan (Zugegriffen am 13. Juli 2022)
[65] Vgl. Mobile Security Frameowkr: https://github.com/MobSF/Mobile-Security-Framework-MobSF (Zugegriffen am 13. Juli 2022)
[66] Vgl. Gitleaks: https://gitleaks.io (Zugegriffen am 13. Juli 2022)

OWASP Dependency-Check

OWASP Dependency-Check[67] ist ein Werkzeug, das versucht, bekannte Schwachstellen zu erkennen, die in den Abhängigkeiten (Bibliotheken) eines Projekts enthalten sind. Dazu wird festgestellt, ob es einen CPE-Bezeichner (Common Platform Enumeration) für eine bestimmte Abhängigkeit gibt. Dieser identifiziert das Softwarepaket und wird genutzt, um in der NVD[68] nach zugehörigen CVE-Einträgen zu suchen. Das Werkzeug untersucht auch Abhängigkeiten, die durch den Swift Package Manager oder CocoaPods eingebunden wurden. Beide Funktionalitäten sind allerdings als experimentell gekennzeichnet.[69]

Weitere Werkzeuge

Für das Auffinden von bekannten Schwachstellen in externen Komponenten gibt es weiterhin den *SwiftDependencyChecker*[70]. Dieser sucht ebenfalls mittels gefundenen CPEs in der NVD. Unterstützt werden hierbei der Swift Package Manager, CocoaPods und Carthage. Das Werkzeug kann direkt in Xcode integriert werden.

Um externe Komponenten nach neuen Versionen zu prüfen gibt es verschiedene Möglichkeiten und Werkzeuge. *CocoaPods* bietet selbst die Funktion über den Befehl `pod outdated` neue Versionen eingebundener Bibliotheken aufzulisten. Für den Swift Package Manager und Xcode kann *swift-outdated*[71] verwendet werden und listet ebenfalls auf, für welche Bibliotheken neue Versionen verfügbar sind.

Neben den (Teil)-Open-Source-Werkzeugen existiert ebenfalls eine Vielzahl an proprietären und cloud-basierten Werkzeugen für statische Tests aller Art. Einige Beispiele und Hersteller sind Codacy[72], Checkmarx[73], Synk[74] und viele mehr. Eine umfangreiche Übersicht liefert der Heise-Artikel „Quelltext im Fokus - Werkzeuge zur automatischen Codeanalyse“ von A. Wiegenstein. [Wie21]

2.4.5 Weitere unterstützende Werkzeuge

Apple Command Line Tools

Die Command-Line-Tools (CLTs) sind eine Sammlung verschiedener Werkzeuge, die für die Entwicklung von macOS- und iOS-Anwendungen in der Kommandozeile eingesetzt werden können. Darunter befindet sich auch das Werkzeug *Xcodebuild*. Mit dessen Hilfe kann ein beliebiges Xcode-Projekt über die Kommandozeile gebaut, analysiert, getestet, archiviert und exportiert werden.[75] Diesen sogenannten Build-Actions können verschiedene Parameter und Konfigurationen mitgegeben werden.[76] Statusmeldungen und Ergebnisse können direkt der Kommandozeile entnommen werden, wenn der Befehl ausgeführt wird.

---

[67] Vgl. OWASP Dependency-Check: https://owasp.org/www-project-dependency-check/ (Zugegriffen am 13. Juli 2022)

[68] Vgl. National Vulnerability Database: https://nvd.nist.gov (Zugegriffen am 13. Juli 2022)

[69] Vgl. File Type Analyzers: https://jeremylong.github.io/DependencyCheck/analyzers/index.html (Zugegriffen am 13. Juli 2022)

[70] Vgl. SwiftDependencyChecker: https://github.com/kristiinara/SwiftDependencyChecker (Zugegriffen am 13. Juli 2022)

[71] Vgl. swift-outdated: https://github.com/kiliankoe/swift-outdated (Zugegriffen am 13. Juli 2022)

[72] Vgl. Codacy: https://www.codacy.com (Zugegriffen am 13.06.2022)

[73] Vgl. Checkmarx: https://checkmarx.com (Zugegriffen am 13. Juli 2022)

[74] Vgl. Synk: https://snyk.io (Zugegriffen am 13. Juli 2022)

[75] Vgl. „Building from the Command Line with Xcode FAQ“: https://developer.apple.com/library/archive/technotes/tn2339/_index.html (Zugegriffen am 13. Juli 2022)

[76] Einen ausführlichen Überblick zur Anwendung liefert die Manpage des Werkzeuges. Aufrufbar in der Kommandozeile mit dem Befehl: `man xcodebuild`. Xcode oder die CLTs müssen dafür auf dem Mac installiert sein.

AsciiDoc für technische Dokumentationen

AsciiDoc[77] ist eine Auszeichnungssprache, die speziell für die Erstellung technischer Dokumentationen entwickelt wurde. Anwendung findet sie bspw. für Entwickler- und Kundendokumentationen oder für Versionshinweise, welche den Kunden bei einer Bereitstellung der Software mit zur Verfügung gestellt werden.
AsciiDoc-Dateien sind im Prinzip einfache Textdateien, die mit jedem beliebigen Texteditor erstellt und bearbeitet werden können. Das ermöglicht ein direktes Schreiben solcher Dokumentation in einer IDE und eine einfache Einbindung in ein Versionskontrollsystem. Somit können AsciiDoc-Dateien auch leicht mit in einen Review-Prozess integriert werden.
Mittels des Open-Source-Werkzeuges Asciidoctor[78] können AsciiDoc-Dateien einfach in andere Formate, wie PDF oder HTML konvertiert werden.

[77] „Vgl. AsciiDoc Language Documentation“: https://docs.asciidoctor.org/asciidoc/latest/ (Zugegriffen am 13. Juli 2022)
[78] Vgl. Asciidoctor: https://asciidoctor.org (Zugegriffen am 13. Juli 2022)

Kapitel 3

Anforderungsanalyse

3.1 Planung und Durchführung eines Experteninterviews

Wie bereits in der Einleitung und den Grundlagen erarbeitet, spielen die eigenen Anforderungen, sowohl zur Bestimmung von Maßnahmen für die Qualitätssicherung als auch zur generellen Automatisierung, eine entscheidende Rolle. So sollten wichtige Qualitätsmerkmale und dazugehörige Tests bekannt sein, damit unterstützende Werkzeuge sorgfältig evaluiert werden können. Auch sollte der zu automatisierende Prozess klar definiert und abgegrenzt sein, um diesen zielgerichtet automatisieren zu können. Um die erforderlichen Informationen herauszufinden muss mit direkten Beteiligten gesprochen werden. Aus diesem Grund soll ein Experteninterview zum Zweck der Anforderungsanalyse durchgeführt werden.
Ein Experteninterview biete nach C. Rupp die Vorteile, dass relevante Informationen (neue Anforderungen oder implizite Anforderungen) aus erster Hand gewonnen werden können. Durch gezieltes Nachfragen könne flexibel reagiert, Unklarheiten beseitigt und ausgewählte Bereiche vertieft werden. So erhöhe die persönliche Präsenz des Interviewers die Wahrscheinlichkeit, dass Fragen auch wirklich in seinem Sinne korrekt und vollständig beantwortet werden. Von Nachteil sei, dass die Effektivität des Interviews von der Erfahrung des Interviewers und der Haltung des Interviewten abhingen. Außerdem wirken zusätzlich zur Formulierung der Fragestellung auch Mimik, Gestik und Tonfall des Interviewers auf den Interviewten ein. Allgemein bedürfe ein Experteninterview sehr viel Aufwand, da der Interviewer selbst in gewisser Weise zu einem Experten werden müsse, um entsprechend Fragen stellen und auf Antworten reagieren zu können. [SOP14, S.106ff].

Für ein strukturiertes und zielführendes Vorgehen im Interview wurde ein Interviewleitfaden erarbeitet. Dieser enthält wichtige Leitfragen, eine Erwartungshaltung sowie einige Hinweise zu vertiefenden Fragen. Der Interviewleitfaden ist dem Anhang zu entnehmen, siehe Abschnitt A.1.
Für die Effektivität des Interviews ist die richtige Auswahl eines Experten wichtig. Hierfür wurde der am Fraunhofer IVI technische Angestellte und leitende iOS-Entwickler Herr Sebatian Koch befragt. Aufgrund seiner langjährigen Erfahrung, tiefgreifender Kenntnisse des Entwicklungsprozesses und seiner Position ist er als Experte für das Interview geeignet. Das entstandene Protokoll ist dem Anhang zu entnehmen, siehe Abschnitt A.2.

3.2 Kontext, Stakeholder und Ziele

Auf Grundlage des Experteninterviews, siehe Abschnitt A.2, werden im Folgenden der genaue Kontext, wichtige Stakeholder und Ziele erläutert.
Der **Kontext** geht aus der Antwort auf Frage F2 hervor. Generell soll die Qualitätssicherung und Bereitstellung einer iOS-Anwendung automatisiert werden. Die vier Jahre alte native iOS-Anwendung ist dabei Teil einer Multiplattform-Anwendung und dient als Client auf iOS-Geräten. Programmiert wird die Anwendung mittels der Programmiersprache Swift in Xcode oder AppCode. Externe Bibliotheken werden über CocoaPods und den Swift Package Manager eingebunden.
Grundlegend wird bei der Entwicklung nach Scrum vorgegangen und die Entwicklungsplattform

GitLab als Unterstützung genutzt. Der Quellcode wird mit Hilfe unterschiedlicher Branches verwaltet. So existiert einen Haupt-Branch *(master)* von dem ein dauerhaft existierender Entwicklungs-Branch (development) abzweigt. Für neue Funktionen oder Verbesserungen jeglicher Art wird das sogenannte *Feature-Branching* eingesetzt.[79] Vor jeder Zusammenführung (Merge) eines Feature-Branches in den Entwicklungs-Branch erfolgt ein manuelles Code-Review durch einen anderen Entwickler. Für einen Release der neuen oder verbesserten Funktionalitäten wird der Entwicklungs-Branch in den Haupt-Branch gemergt und die entstandene Codebasis als Ausgangspunkt verwendet. Tests werden von den jeweiligen Entwicklern selbst geschrieben und verwaltet.
Die wichtigsten **Stakeholder** sind Entwickler, Projektverantwortliche sowie Auftraggeber und Nutzer (F3). Für Entwickler zähle eine Fehlerminimierung, effizientere Arbeitsweise und schnelles Feedback. Projektverantwortliche seien an der Qualitätssicherung, effektiver und effizienter Ressourcennutzung, Risikominimierung und einfacher Umsetzung von individuellen Kundenwünschen interessiert. Auftraggebern und Nutzern seien wenige Fehler und schnelle Release-Zeiten wichtig.
Um die Interessen der verschiedenen Stakeholder zu genügen, sollen mittels der Automatisierung folgende **Ziele** erreicht werden (F1). Zum einen soll die Softwarequalität kontinuierlich gesichert und messbar gemacht werden. Die dafür eingesetzte Automatisierung soll zur Minimierung von menschlichen Fehlern bei repetitiven Aufgaben und zur Steigerung der Effizienz dienen. Zum anderen soll dadurch ein qualitativeres und sturkturierteres Vorgehen aller Teammitglieder erreicht werden. Für die Erreichung der gesetzten Ziele sollen Bausteine und Werkzeuge für die automatisierte Qualitätssicherung während der Implementierung und Integration der Anwendung gefunden und exemplarisch umgesetzt werden. Ein vollautomatisches Deployment, also die automatische Installation der Anwendung bei den Kunden, kann und soll aufgrund spezieller Gegebenheiten nicht umgesetzt werden. (F4)
Die Ziele gelten als erreicht, wenn definierte manuelle Arbeitsschritte automatisiert ausgeführt werden und eine Empfehlung für die Etablierung weiterer automatischer qualitätssichernder Maßnahmen existiert. (F5) Als Beweis für die Machbarkeit soll die Automatisierung exemplarisch umgesetzt werden.

3.3 Auswahl geeigneter Maßnahmen zur automatisierten Qualitätssicherung

3.3.1 Statische und dynamische Tests

Im folgenden Kapitel sollen Bausteine definiert werden, die für eine automatisierte Qualitätssicherung im Rahmen des Kontextes infrage kommen.
Aus den Antworten auf die Fragen F7 und F8, siehe Abschnitt A.2, geht bereits hervor, dass **Unit-, Integration- und Regressions-Tests** durchgeführt werden sollen. Hinzu sollen generell auch **statische Tests** kommen, die u.a. duplizierten Code, ungenutzte Deklarationen und Verstöße gegen allgemeine Programmierrichtlinien erkennen (F9). Weiterhin sollten, wenn möglich, auch **UI- und E2E-Tests** durchgeführt werden, siehe F10. Diese Maßnahmen entsprechen dabei den allgemein Empfohlenen, wie in Unterabschnitt 2.2.3 dargestellt. Da die iOS-Anwendung bereits vier Jahre alt ist (F3), kann davon ausgegangen werden, dass sich die grafische Benutzeroberfläche (bei den Hauptfunktionen) nur geringfügig ändert. Somit ist auch das Risiko niedrig, UI- oder E2E-Tests häufig anpassen zu müssen.
Zusätzlich zu den geforderten Maßnahmen werden in Unterabschnitt 2.2.3 weitere empfohlen, die sich auch für die konkrete iOS-Anwendung anbieten. Das umfasst die **Schwachstellenanalyse**, die **Secret-Detection** und das **Dependency-Scanning**, da externe Komponenten verwendet werden (F2). Eine weitere empfohlene Maßnahme ist das **Fuzzing**. Das sei nach Wishnick allerdings für iOS-Anwendungen schwierig umsetzbar, da herkömmliche Simulatoren und reale Geräte einen limitierenden Faktor darstellen. [WC13, S.7f] Eine dahingehende umfassende Untersuchung ist nicht Ziel der Arbeit und bietet sich somit für zukünftige Untersuchungen an.

[79] Ein Feature-Branch zweigt dabei von einem definierten Haupt-Branch ab, dient der Implementierung eines neuen Features und wird wieder mit dem Haupt-Branch zusammengeführt. Weitere Informationen: https://martinfowler.com/articles/branching-patterns.html#feature-branching (Zugegriffen am 04.07.2022).

3.3.2 Metriken und Quality Gate

Neben dem Testen stellt die Erfassung von **Metriken** eine weitere wichtige Maßnahme dar, die ergriffen werden sollte. Wie in Unterabschnitt 2.2.2 erläutert, gibt es viele unterschiedliche Metriken für die unterschiedlichsten Anwendungsgebiete. Aus der Antwort auf Frage F6 geht hervor, dass keine speziellen Metriken festgelegt worden sind und im Rahmen der Arbeit Vorschläge gemacht werden sollen. Eine umfassende Darstellung der Identifikation geeigneter Metriken, deren Berechnung, Effektivität, Interpretationen und daraus abzuleitende Maßnahmen würde an dieser Stelle den Rahmen der Arbeit übersteigen. Vielmehr bietet es den Stoff für eine weitere Arbeit, die sich intensiv mit dieser Thematik befasst. Auf einige Metriken soll an dieser Stelle dennoch eingegangen werden, da sie essentiell für die Qualitätssicherung sind. Denn nur wer misst, kann schließlich Fort- oder Rückschritte sichtbar machen und Verbesserungspotentiale erkennen.
Als Grundlage für die Identifikation geeigneter Metriken dienen einige Qualitätsmerkmale der iOS-Anwendung, die als besonders wichtig erachtet werden. Diese sind Sicherheit, Funktionserfüllung und Wartbarkeit (F6). Für die Sicherheit könnte sich die Anzahl an gefundenen Schwachstellen eignen. Zur Überwachung der Funktionserfüllung kann eine Metrik zur Testabdeckung herangezogen werden, wie die Funktions- oder Anweisungsabdeckung. Komplexitätsmetriken (bspw. zyklomatische Komplexität) und die Anzahl an Verstößen gegen festgelegt Programmierrichtlinien könnten eine Schluss über die Wartbarkeit zulassen. Jeder ermittelte Wert sollte dabei hinterfragt und sorgfältig interpretiert werden. Gefundene Schwachstellen und Fehler sollten beseitigt, Tests geschrieben oder ggf. ein Refactoring durchgeführt werden.
Um die Effizienz der Automatisierung an sich beurteilen zu können, kann die Dauer gemessen werden, die eine Pipeline zum Abarbeiten der einzelnen Arbeitsschritte benötigt. Um die Dauer zu verkürzen, können bspw. voneinander unabhängige Aufgaben parallelisiert oder Tests hinsichtlich Performance verbessert werden, sofern möglich.
Die einzelnen Metriken könnten für eine automatisches **Quality Gate** genutzt werden, wie in Unterabschnitt 2.3.3 vorgestellt. Die Durchführung von Code-Reviews (F2) stellen dabei bereits ein etabliertes manuelles Quality Gate dar. Die gemessenen Werte könnten an dieser Stelle verwendet werden, um festzustellen, ob ein manuelles Code-Review überhaupt notwendig ist oder allein ein automatische Quality Gate genügt. Das könnte bspw. der Fall sein, wenn keine Schwachstellen gefunden worden sind, die Testabdeckung einem bestimmten Wert entspricht (bspw. größer als 80%), keine Komplexitätswarnungen entstanden sind, keine Verstöße gegen Programmierrichtlinien festgestellt worden sind und der Umfang einer Änderungen nur wenige Code-Zeilen umfasst. Aufgrund der Ausführungen in Unterabschnitt 2.3.3 wird dies allerdings nicht empfohlen. Vielmehr sollten die Metriken und Ergebnisse eines automatischen Quality Gates als Unterstützung in manuellen Code-Reviews dienen.

3.4 Möglichkeiten zur Einbindung der Automatisierung in den Entwicklungsprozess

Sowohl der Build-Prozess als auch die in Unterabschnitt 3.3.1 aufgeführten Tests können automatisiert werden, wie bereits grundlegend in Unterabschnitt 2.3.3 erläutert. Auch die in Unterabschnitt 3.3.2 genannten Metriken können automatisch berechnet und anschließend ausgewertet werden. Das für die Bereitstellung der iOS-Anwendung notwendige Archivieren und Exportieren, die Erzeugung einer PDF-Datei mit Versionshinweisen sowie das anschließende Paketieren und Verschlüsseln eines Release-Paketes stellt ebenfalls kein Problem dar, da es immer die gleichen werkzeug-gestützten Arbeitsschritte sind. Das betrifft auch den Upload des Release-Paketes in die unternehmenseigene Nextcloud, die Erstellung einer Freigabe sowie die Erzeugung einer PDF-Datei mit den jeweiligen Freigabeinformationen.

Zunächst ist festzustellen, dass die geforderten Prozesse automatisiert werden können. Angefangen bei dem Build-Prozess, über das Testen, bis hin zur Bereitstellung. Da diese Prozesse aufeinanderfolgen besteht die Möglichkeit, dass alles vollautomatisch hintereinander abläuft, ausgelöst durch einen Push in das zentrale Git-Repository oder manuell durch einen Entwickler. Doch nur, weil es geht, sollte es nicht gleich getan werden. Es ist bspw. wenig sinnvoll für jeden Push in einen Feature-

Branch eine Pipeline zu erstellen, die für eine automatische Bereitstellung sorgt. Das Feature könnte schlichtweg noch nicht fertig sein und der Push lediglich zur Sicherung des Arbeitsstandes dienen. In Anlehnung an CI ist es allerdings wiederum sinnvoll, dass bei jedem Push in einen Feature-Branch statische und dynamische Tests durchgeführt werden, um dem Entwickler schnellstmöglich Feedback liefern zu können. Jene Tests sollten auch durchgeführt werden, wenn ein Push in *development* oder *master* erfolgt ist. Das sorgt für eine automatisierte Qualitätssicherung von *master* bis auf Feature-Branch-Ebene. In der Pipeline für Feature-Branches und *development* bietet sich außerdem die Definition eines abschließenden manuell zu startenden Arbeitsschrittes an, der bei Bedarf die Anwendung zu Testzwecken automatisiert exportiert. Die entsprechende `*.ipa`-Datei könnte somit schnell und einfach einen aktuellen Entwicklungsstand widerspiegeln und auf Testgeräten installiert werden. Erst bei einem Push in den *master* sollte die Anwendung vollautomatisch und vollumfänglich bereitgestellt werden - vorausgesetzt, dass vorher alle Tests erfolgreich durchgelaufen sind.
Je nach Branch unterscheiden sich also die einzelnen auszuführenden Arbeitsschritte in einer Pipeline. Außerdem sollte die Möglichkeit bestehen, dass bestimmte Arbeitsschritte erst nach einer manuellen Bestätigung ausgeführt werden.

Aufgrund der vorangegangenen Feststellungen liegt es nahe, nach dem Konzept von CI/CD vorzugehen. Wie in Unterabschnitt 2.3.3 erläutert, müsse dabei jeder Entwickler seine Arbeit mindestens einmal am Tag in genau einen Hauptbranch integrieren. Durch die Anwendung von Feature-Branching und Code-Reviews wird die Umsetzung des Konzeptes jedoch erschwert bis unmöglich. Humble und Farley haben es mit der folgenden Erläuterung bzgl. Feature-Branching auf den Punkt gebracht: „If different members of the team are working on separate branches, then by definition they're not continuously integrating. Perhaps the most important practice that makes continuous integration possible is that everybody checks in to mainline at least once a day. So if you merge your branch to (not just from) mainline once a day, you're OK. If you're not doing that, you're not doing continuous integration." [HF11, S.390f] Zu dem damit verbunden erhöhten Merge-Aufwand durch das momentane Branching-Modell kommt noch hinzu, dass jede Änderungen, die in *development* gemergt werden soll, ein Code-Review von einem anderen Entwickler benötige (F3). Fowler führt auf, dass dies wichtige Zeit koste und die Integration blockiere. Aus diesen Gründen sei Feature-Branching und Code-Review eine bessere Kombination als CI und Code-Review.[80] Daraus folgt, dass auch CD nicht zur Anwendung kommen kann, da diese Konzepte auf CI basieren. Für ein Vorgehen nach CI müsste das aktuelle Vorgehen bei der Entwicklung überdacht werden. Ursprünglich ist CI aus den Praktiken von XP bekannt.

[80] Vgl. „Patterns for Managing Source Code Branches - Pre-Integration Review": https://martinfowler.com/articles/branching-patterns.html#reviewed-commits (Zugegriffen am 21.07.2022)

3.5 Anforderungsdefinition

Basierend auf dem Experteninterview, siehe Abschnitt A.2, und den Ausführungen der vorherigen Kapitel ergeben sich die in Tabelle 3.5.1 aufgeführten Anforderungen an ein Automatisierungssystem.

Ref.	Anforderung
	Das System **muss** fähig sein, ...
AM1	eine Pipeline bei einem Push in das zentrale Git-Repository (GitLab) zu starten.
AM2	den Quellcode aus dem Repository zu laden.
AM3	definierte Targets zu bauen.
AM4	vorhandene Unit-Tests durchzuführen.
AM5	Tests in definierten Simulatoren auszuführen.
AM6	duplizierten Code zu erkennen.
AM7	ungenutzte Deklarationen zu erkennen.
AM8	Klassen mit mehr als 400 Zeilen zu erkennen.
AM9	(Release-)Builds mit einer spezifischen Signatur zu erzeugen.
AM10	eine PDF-Datei mit Versionshinweisen zu erzeugen.
AM11	Versionshinweise und Release-Build als Release-Paket zu paketieren.
AM12	ein Release-Paket als ZIP-Archiv zu komprimieren.
AM13	das Release-Paket in die Nextcloud hochzuladen.
AM14	Ergebnisse und Artefakte zentral über ein Webinterface zur Verfügung zu stellen.
AM15	den Zugang zu Ergebnissen und Artefakten mit Nutzername und Passwort zu beschränken.
AM16	die Pipeline manuell über ein Webinterface zu starten.
AM17	die Abarbeitung bestimmter Arbeitsschritte in der Pipeline manuell zu bestätigen.
AM18	verschiedene Arbeitsschritte je nach Branch (Master, Development, Feature-Branch) auszuführen.
	Das System **sollte** fähig sein, ...
AS1	fehlende Completion-Aufrufe zu erkennen.
AS2	Singletons zu erkennen.
AS3	Verstöße gegen allgemeine Programmierrichtlinien zu erkennen.
AS4	bekannter Schwachstellen zu erkennen.
AS5	Secret im Quellcode zu erkennen.
AS6	ein Dependency-Scanning durchzuführen.
AS7	UI-Tests durchzuführen.
AS8	E2E-Tests unter Nutzung eines Testbackends durchzuführen.
AS9	Tests auf einem physischen Endgerät auszuführen.
AS10	die Testabdeckung (Funktionsüberdeckung) zu ermitteln (Metrik).
AS11	die (zyklomatische) Komplexität zu ermitteln (Metrik).
AS12	die Anzahl an Verstößen gegen Programmierrichtlinien zu ermitteln (Metrik).
AS13	die Anzahl an bekannten gefundenen Schwachstellen zu ermitteln (Metrik).
AS14	die Lines of Code bei Änderungen zu ermitteln (Metrik).
AS15	die Ergebnisse eines automatischen Quality Gates einzubinden.
AS16	Release-Pakete mittels Passwort und AES-256 zu verschlüsseln.
AS17	eine Nextcloud-Freigabe pro hochgeladenem Release-Paket zu erstellen.
AS18	eine PDF-Datei mit Freigabelink und -passwort sowie Passwort des verschlüsselten Release-Paketes zu erzeugen.

Tabelle 3.5.1: Anforderungsdefinition

3.6 Kriterien zur Auswahl von Werkzeugen

Zur Bewertung und Auswahl von unterstützenden Werkzeugen wurden auf Grundlage der Anforderungen, siehe Tabelle 3.5.1 und der Literaturrecherche, siehe Unterabschnitt 2.2.1 sowie Unterabschnitt 2.3.4, die folgenden Kriterien definiert. Diese dienen vor allem zur Überprüfung der Abdeckung definierter Anforderungen und deren Erfüllungsgrad.
Die Kriterien wurden in Ausschluss- und Bewertungskriterien unterteilt. Ausschlusskriterien dienen dem sofortigen Ausschluss des Werkzeuges, wenn durch dessen Nutzung eine festgelegte Anforderung, die erfüllt werden muss, nicht erfüllt werden kann. Bewertungskriterien dienen zur Feststellung des Erfüllungsgrads und bedürfen einer Messung, eines Tests oder einer Einschätzung zur Bewertung. Dafür sollen Noten von 1 (hoher Erfüllungsgrad) bis 5 (geringer Erfüllungsgrad) verwendet werden. Um in jedem Fall das Werkzeug mit dem größten Mehrwert zu erhalten, wurden die Bewertungskriterien entsprechend ihrer Nutzbringung von 1 (eher niedriger Nutzen) bis 3 (hoher Nutzen) gewichtet.

Alle Werkzeuge sollten generell den in Tabelle 3.6.1 aufgeführten Kriterien genügen.

Kategorie	**Kriterium**	**Gewichtung**
	Ausschlusskriterien	
Rahmenbedingungen	Lizenzkosten	-
	Cloud-Service	-
	Onlineverbindung zu Hersteller	-
	Bewertungskriterien	
Effizienz	Geringer Ressourcenverbrauch (Zusätzliche Hardware, Wartungskosten, usw.)	3
Benutzbarkeit	Übersichtliche / informative Benutzeroberfläche	2
	Vollständige und verständliche Dokumentation	3
	Zielführende / intuitive Bedienung	2
Zuverlässigkeit	Geringe Gefahr durch Herstellerabhängigkeit	2
	Gute Herstellerbewertung/große Nutzerbasis	1
Effektivität	Übereinstimmung mit geforderten Funktionalitäten	3
Wartbarkeit	Leicht anpassbar und änderbar	2
Kompatibilität	Eignung für Umgebung/Werkzeugkette	3

Tabelle 3.6.1: Generelle Kriterien für unterstützende Werkzeuge

Speziell für ein Automatisierungswerkzeug ergeben sich zusätzlich die in Tabelle 3.6.2 dargestellten Kriterien.

Kategorie	Kriterium	Gewichtung
	Ausschlusskriterien	
Funktionalität	keine Unterstützung für Befehlsausführung auf macOS	-
	keine Unterstützung für Git	-
	keine Unterstützung für Pipeline-Trigger durch GitLab	-
	keine Unterstützung für Branch-Bedingungen	-
	keine Unterstützung für manuell zu bestätigende Arbeitsschritte während Ausführung der Pipeline	-
	kein Artefaktesupport (Caching und Bereitstellung)	-
	kein Pipeline-as-Code-Ansatz	-
	kein Webinterface für Visualisierung und Steuerung	-
	keine Zugangskontrolle mittels Nutzername und Passwort	-
	Bewertungskriterien	
Funktionalität	Einfache manuelle Bestätigung für entsprechende Arbeitsschritte	1
	Einfacher Zugang zu entstandenen Artefakten	3
	Einfache Steuerung der Pipeline	2
	Nützliche Erweiterungsmöglichkeiten durch andere vorhandene Funktionalitäten	1

Tabelle 3.6.2: Kriterien für ein Automatisierungswerkzeug

Um UI- und E2E-Tests effizient und effektiv durchführen zu können, wird ein Werkzeug benötigt, das zusätzlich den in Tabelle 3.6.3 aufgeführten Kriterien entspricht. Die Tests könnten dabei auch durch andere Programmiersprachen geschrieben werden, siehe Appium in Abschnitt 2.4.3. Da es aber kein explizites Testteam gibt, Tests von jeweiligen Entwicklern selbst geschrieben werden und für Unit-Tests bereits Swift verwendet wird, soll an dieser Stelle der Einfachheit halber auch Swift verwendet werden.

Kategorie	Kriterium	Gewichtung
	Ausschlusskriterien	
Rahmenbedingungen	keine Unterstützung für iOS	-
	keine Unterstützung für Swift	-
Funktionalität	kein Recording zur Erstellung von Testfällen	-

Tabelle 3.6.3: Kriterien für ein UI- und E2E-Testautomatisierungswerkzeug

Die Definition von Kriterien für genau ein Werkzeug zur Durchführung von statischen Tests gestaltet sich schwierig, da die infrage kommenden Werkzeuge, siehe Unterabschnitt 2.4.4, jeweils auf bestimmte Anwendungsfälle spezialisiert sind. Vielmehr ist es eine Kombination aus möglichen Werkzeugen, die dabei die in Tabelle 3.6.4 dargestellten Kriterien bestmöglich abdecken sollten. Wenn bestimmte Funktionalitäten vorhanden sind, so sollen deren Gewichtungen aufaddiert und zur Bewertung herangezogen werden.

Kategorie	**Kriterium**	**Gewichtung**
	Ausschlusskriterien	
Rahmenbedingungen	keine Unterstützung für iOS	-
	keine Unterstützung für Swift	-
	Testergebnisse beeinflussen nicht den Exit-Code	-
	Bewertungskriterien (Erfüllung)	
Funktionalität	besitzt die Möglichkeit zur Nutzung eines Quality Gates	3
	stellt Testabdeckung dar	2
	erkennt bekannte Schwachstellen	3
	erkennt Secrets im Quellcode	2
	führt Dependency-Scans durch	2
	erkennt Verstöße gegen allgemeine Programmierrichtlinien	1
	erkennt duplizierten Code	2
	erkennt ungenutzte Deklarationen	2
	erkennt Klassen mit mehr als 400 LoC	2
	erkennt fehlende Completion-Aufrufe	2
	erkennt Singletons	2
	misst Lines of Code	1
	misst zyklomatische Komplexität	2
	misst Anzahl an vorhandenen Schwachstellen	1
	misst Anzahl an Verstößen gegen allgemeine Programmierrichtlinien	1
	Möglichkeit zur Definition eigener Regeln	2
	besitzt umfangreiches Reporting	3

Tabelle 3.6.4: Kriterien für Werkzeuge zur Durchführung von statischen Tests

Für Werkzeuge, die der Durchführung von Unit-Tests oder der Bereitstellung dienen, gelten ebenfalls die allgemeinen Kriterien. Zusätzlich müssen an dieser Stelle die Kriterien erfüllt sein, dass mit Hilfe der ausgewählten Werkzeuge Unit-Tests automatisch durchgeführt werden und die Testabdeckung gemessen wird. Für die Bereitstellung ergeben sich die Kriterien, dass die iOS-Anwendung automatisch archiviert und exportiert wird, Versionshinweise als PDF-Datei erzeugt werden, ein Release-Paket paketiert, zu einem ZIP-Archiv komprimiert und mittels Passwort unter Verwendung von AES-265 verschlüsselt wird. Hinzu kommt noch, dass automatisch mit der Nextcloud kommuniziert wird, um Release-Pakete hochzuladen und Freigaben zu erstellen. Als letztes folgt das Kriterium, dass eine Bereitstellungsinformation als PDF-Datei erzeugt wird.

3.7 Vorauswahl geeigneter Werkzeuge anhand der definierten Kriterien

Wie Abschnitt 2.4 erkennen lässt, existieren viele Werkzeuge, die als Unterstützung infrage kommen. Im folgenden Kapitel soll anhand der definierten Kriterien eine erste Vorauswahl getroffen werden, um die am geeignetsten erscheinenden Werkzeuge für eine konkrete Umsetzung zur Erfüllung der Anforderungen heranzuziehen.

Angefangen bei den Automatisierungswerkzeugen gibt es speziell für iOS-Anwendungen den Xcode Server. Dieser unterstützt allerdings nicht das *Pipeline-as-Code*-Prinzip und es können auch keine Branch-Bedingungen definiert werden. Darüber hinaus ist dieser kaum anpassbar und kann nicht für andere Projekte, bspw. für Android, genutzt werden. Die enthaltene statische Codeanalyse mit Xcode Analyze kann zudem nicht für Swift-Projekte genutzt werden. Ähnlich verhält es sich mit Xcode Cloud. Hinzu kommt hier, dass es ein Cloud-Service und obendrein noch kostenpflichtig ist. Beide Werkzeuge kommen somit nicht für eine Umsetzung infrage.
Für die allgemeine Automatisierung gibt es weiterhin GitLab CI/CD, Jenkins, Fastlane, Atlassian Bamboo, TeamCity und viele mehr, wie in Unterabschnitt 2.4.2 vorgestellt. Bei einer Auswahl ist besonders darauf zu achten, dass das Automatisierungswerkzeug mit GitLab als Versionsverwaltungssystem interagieren kann, um bei einem Push überhaupt automatisiert eine Pipeline triggern zu können und Quellcode zu laden. GitLab CI/CD als Teil von GitLab erfüllt dabei keine Ausschlusskriterium und sollte dementsprechend für eine Umsetzung in Betracht gezogen werden.
GitLab selbst führt Jenkins als externen Dienst für Continuous Integration auf, der vollumfänglich integriert werden kann. Auch Jenkins erfüllt keine Ausschlusskriterien, was eine weitere Untersuchung rechtfertigt.[81]
Laut GitLab können außerdem die eigenen GitLab-Projekte in andere Anwendungen integriert werden.[82] Für CI/CD-Pipelines werden neben Jenkins noch die Systeme Bamboo[48], Buildkite[83], Drone[84] und TeamCity[47] genannt. Bamboo von Atlassian ist kostenpflichtig und steht nur als Cloud-Service oder für Großunternehmen mit eigenem Rechenzentrum zur Verfügung. Buildkite ist ebenfalls eine proprietäre Lösung, deren ausführende Agents zwar auf der eigenen Infrastruktur laufen aber die ganze Konfiguration und Visualisierung über eine Cloud-Plattform abgehandelt wird. Drone ist ebenfalls kostenpflichtig. Für TeamCity existiert eine freie Version, die selbst-gehostet werden kann. Allerdings stehen dabei maximal drei Build Agents[85] zur Verfügung, die Befehle ausführen. Für weitere Agents ist eine kostenpflichtige Lizenz erforderlich. Speziell bei iOS wird bereits ein Build Agent für macOS benötigt. Außerdem kann ein Agent nur einen Prozess zur selben Zeit steuern. Wenn unterschiedliche Pipelines unterschiedlicher Teams und Projekte auf ein und denselben Agent zugreifen, so kann es zu unnötigen Warteschlagen kommen. Diese mindern die Effizienz der Automatisierung und sorgen für ein langsameres Feedback. Daraus resultiert, dass sich die freie Version von TeamCity für ein kleines Team mit nur einer Anwendung eignet, aber nicht für die Automatisierung einer Client-Server-Anwendung mit mehrere Clients und Teams für unterschiedliche Plattformen. Dafür bräuchte es mehr Agents, was zu Lizenzkosten führt.
Als weiteres Automatisierungswerkzeug kommt noch Fastlane in Betracht, das viele Schritte vereinfacht und Prozesse automatisiert. Es ist allerdings keine zentrale Plattform mit Nutzer- und Artefakteverwaltung sondern läuft lokal auf dem installierten System in der Kommandozeile. Es muss dementsprechend von einer anderen Automatisierungslösung wie GitLab CI/CD oder Jenkins eingebunden werden. Die Schritte, die dabei mittels Fastlane automatisiert werden, können allerdings genauso gut von der übergeordneten Automatisierung übernommen werden, wodurch die Konfigurationsdateien von Fastlane erspart bleiben. Das Werkzeug kommt dementsprechend nicht als zentrale Automatisierungslösung infrage aber ist dennoch für bestimmte Angelegenheiten interessant. So

[81] Vgl. GitLab integrations - Continuous integration: https://docs.gitlab.com/ee/integration/#continuous-integration (Zugegriffen am 23.06.2022).
[82] Vgl. Project integrations - Available integrations: https://docs.gitlab.com/ee/user/project/integrations/#available-integrations (Zugegriffen am 13. Juli 2022)
[83] Vgl. Buildkite: https://buildkite.com (Zugegriffen am 13. Juli 2022)
[84] Vgl. Drone: https://www.drone.io (Zugegriffen am 13. Juli 2022)
[85] Vgl. TeamCity Build Agent: https://www.jetbrains.com/help/teamcity/build-agent.html (Zugegriffen am 13. Juli 2022)

könnten die Lanes nicht nur von der Automatisierung, sondern vom Entwickler lokal in der Kommandozeile genutzt werden, ohne aktuelle Änderungen pushen zu müssen, bspw. um einen Testdurchlauf vor einem Push durchzuführen.

Für dynamische Tests, insbesondere UI-Tests, stehen XCTest, EarlGrey, Appium und diverse Testplattformen zur Verfügung. XCTest integriert sich nahtlos in Xcode, Tests können mit Swift geschrieben werden und in Verbindung mit Xcode gibt es eine Recording-Funktion. EarlGrey baut auf XCTest auf und bringt einige Vorteile mit sich, benötigt werden allerdings externe Bibliotheken, die einer Kontrolle bedürfen, und Zeit für die Einarbeitung in dessen Möglichkeiten. Eine Recording-Funktion steht nicht zur Verfügung. Für Appium wird der Appium-Server benötigt, der Ressourcen bindet, um diesen zu konfigurieren und zu warten. Testfälle können nicht in Swift geschrieben werden sondern erfordern Java, Python oder eine andere Sprache. Außerdem muss sich auch hier explizit in Appium eingearbeitet werden. Für die Verwendung sind spezielle externe Bibliotheken im Projekt erforderlich. Eine Recording-Funktion ist ebenfalls nicht vorhanden. Testplattformen bieten eine Vielzahl an Möglichkeiten, oft auch das Recording von Testfällen. Die Einrichtung und das Testen der Systeme erfordert allerdings viel Zeit. Hinzu kommt der erhöhte Wartungsaufwand und Einarbeitungszeit für Entwickler.
Da XCTest ohnehin bereits verwendet wird, soll sich im Rahmen der Arbeit in erster Linie auf XCTest in Verbindung mit Xcode und Xcodebuild konzentriert werden. So wird auch ein erhöhter Aufwand zum Testen, Einarbeiten und Warten anderer Systeme vermieden. Auf Appium und möglich Testplattformen soll am Ende der Arbeit aber nochmal kurz eingegangen werden.

In Bezug auf statische Tests lässt sich kein Werkzeug finden, dass alle Kriterien und damit Anforderungen abdeckt. Die Hersteller haben sich meist auf jeweilige Bereiche, wie Wartbarkeit oder Schwachstellenanalyse, spezialisiert. Eine Kombination verschiedener Werkzeuge ist dabei in allen Fällen sinnvoll, um die verschiedensten Bereiche abzudecken. Die Vielversprechendsten für eine Automatisierung stellen SwiftLint, Mobsfscan, SonarQube und GitLeaks dar, da jeweils explizit eine CI-Unterstützung genannt wurde. SonarQube steht dabei unternehmensintern bereits in der Developer Edition zur Verfügung, kann benutzt werden und würde keine neuen Lizenzkosten verursachen. GitLab CI/CD erlaubt dank diverser Templates die Möglichkeit Secrets zu erkennen, die Wartbarkeit zu analysieren und Dependencies zu scannen. Das Dependency-Scanning ist jedoch kostenpflichtig und unterstützt außerdem keine Swift- bzw. iOS-Projekte. Für die Secret-Detection wird in GitLab Gitleaks verwendet und für die Analyse der Wartbarkeit Code Climate CLI. An sich weist Code Climate CLI keine explizite CI-Unterstützung auf. Die Analyseergebnisse wirken sich nicht auf den Exit-Code aus, wodurch sich in allen Fällen immer der erzeugte Report angesehen werden muss. Ein automatischer Abbruch der Pipeline wird dadurch nicht möglich. Da dessen Ergebnisse aber über Widgets im Merge Request in GitLab angezeigt werden können, soll das Werkzeug dennoch als Möglichkeit in Betracht gezogen werden. Aufgrund einer leichteren Umsetzung und zur besseren Vergleichbarkeit sollen die Werkzeuge allerdings unabhängig von den Templates eingebunden und evaluiert werden. Herausgefundene Ergebnisse gelten jedoch gleichermaßen auch für die Einbindung der Werkzeuge über die entsprechenden Templates.
OWASP Dependency-Check weist ebenfalls keine explizite CI-Unterstützung auf, könnte aber über ein SonarQube-Plugin eingebunden werden, wodurch es wieder interessant wird.
SwiftDependencyChecker, swift-outdated oder CocoaPods (mit der outdated-Funktion) können keine Reports erzeugen und schreiben ihre Ergebnisse lediglich in die Kommandozeile. Auch hierbei wirken sich die Ergebnisse nicht auf den Exit-Code aus, wodurch immer die Logs der Pipeline bzw. des Arbeitsschrittes überprüft werden müssten. Ein automatischer Abbruch der Pipeline wird dadurch auch hier nicht möglich. SwiftDependencyChecker und swift-outdated können allerdings direkt in Xcode integriert werden, sodass veraltetet oder schwachstellenbehaftete externe Komponenten schnell erkannt werden können. Aus diesem Grund sollten die Werkzeuge eher in Xcode als in eine zentrale Pipeline integriert werden.
Alle weiteren Werkzeuge sind vor allem cloud-basiert und/oder proprietär, was sie im Rahmen dieser Arbeit als Möglichkeit ausschließt.

Für die Bereitstellung muss die iOS-Anwendung in erster Linie archiviert und exportiert werden. Dies funktioniert mit Xcodebuild oder Fastlane, wobei Fastlane auch wieder auf Xcodebuild zurückgreift. Da Xcodebuild ohnehin durch Xcode vorhanden ist, soll dieses für die Umsetzung herangezogen werden.
Für Versionshinweise bietet sich AsciiDoc an, wie in Abschnitt 2.4.5 dargestellt. Durch das bereitgestellte CLI kann der Konvertierungsprozess einfach in eine Automatisierung eingebunden werden.
Zum Paketieren, Komprimieren und Verschlüsseln des Release-Pakets soll das Open-Source-Werkzeug *7-Zip*[86] verwendet werden. Das herkömmliche Zip-Werkzeug unterstützt kein AES-256 und dessen Verschlüsselung kann zudem leicht gebrochen werden, wie W. Drehling darstellt. [Dre22]

Zusammenfassend sollen den Ausführungen entsprechend folgende Werkzeuge für die Konzeption und Umsetzung herangezogen werden:

- GitLab CI/CD und Jenkins als Möglichkeiten für die allgemeine Automatisierung
- XCTest mit Xcode für Unit-, Integration-, UI- und E2E-Tests
- Xcodebuild zum Bauen, Testen, Archivieren und Exportieren der iOS-Anwendung
- SwiftLint, Mobsfscan, SonarQube (inkl. OWASP Dependency-Check-Plugin), Code Climate CLI, GitLeaks für statische Tests
- AsciiDoc und AsciiDoctor für Versionshinweise
- 7-Zip für das Paketieren und Verschlüsseln eines Release-Paketes

Fastlane vereinfacht einige Prozesse und bietet zusätzliche Möglichkeiten. Die grundsätzliche Automatisierung lässt sich jedoch auch ohne dieses Werkzeug umsetzten. Aus diesem Grund soll an geeigneten Stellen lediglich darauf verweisen werden. Eine Umsetzung erfolgt jedoch nicht und kann als mögliche Erweiterung angesehen werden.

[86] Vgl. 7-Zip: https://www.7-zip.de (Zugegriffen am 13. Juli 2022)

Kapitel 4

Konzeption für eine automatisierte Qualitätssicherung und Bereitstellung einer iOS-Anwendung

Die Pipeline gibt vor, wann und in welcher Reihenfolge bestimmte Arbeitsschritte ausgeführt werden sollen und wie mit entstandenen Artefakten verblieben wird. Wie in Unterabschnitt 2.3.2 beschreiben ist die Pipeline als Skript der empfohlene Weg und soll auch für die Umsetzung gewählt werden. Da sich die Syntax zur Definition bei GitLab CI/CD und Jenkins sowie die einzusetzenden Werkzeuge unterscheiden, folgt die Konzeption in abstrakter Weise.

Zunächst soll eine Reihenfolge der Arbeitsschritte festgelegt werden, denn diese spielt eine wichtige Rolle. Die Software sollte bspw. nicht vor den Tests bereitgestellt und freigegeben werden. Die Anwendung wird implementiert, getestet, integriert, getestet und schließlich, wenn alle vorherigen Schritte in Ordnung waren, freigegeben und bereitgestellt.
Generell ist ein schnelles Feedback zu fehlgeschlagenen Tests förderlich, um diese schnell beheben zu können. Dafür wird sich innerhalb der Pipeline das *fail-fast*- oder *run-until-failure*-Prinzip zu Nutze gemacht (anhand Exit-Codes). Daraus resultiert, dass alle Tests, die den größten Nutzen bringen, zuerst durchzuführen sind. Dabei ist zu beachten, dass dynamische Tests länger brauchen als statische Tests. Aus diesem Grund sollen zuerst statische Test durchgeführt werden. Wenn möglich, ist eine parallele Ausführung der Tests noch besser. Zudem sollen alle Tests Artefakte in Form von Testreports erzeugen, damit eingesehen werden kann, wo welcher Fehler vorliegt.
Neben der Pipeline sind auch deren Trigger wichtig. Wie in Abschnitt 3.4 dargestellt soll eine Pipeline bei jedem Push in das Repository ausgeführt werden, wobei der Branch zunächst keine Rolle spielt. Dieser unterscheidet nur, welche Arbeitsschritte ausgeführt werden sollen, d.h. sämtliche Test bei Feature-Branches und *development* aber kein automatischer Export mit Bereitstellung der Anwendung. Es wird lediglich die Möglichkeit zum Exportieren durch einen manuell anzustoßenden Arbeitsschritt geboten. Bei einem Push in *master* erfolgen schließlich alles Tests und das automatische Exportieren und Bereitstellen.
Nach den Ausführungen und Anforderungen ergibt sich die folgende abstrakte Version für eine Pipeline, wie in Abbildung 4.0.1 dargestellt. Alle entstehenden Artefakte sollen dabei über ein Webinterface der Automatisierungsplattform abrufbar sein.

Als Grundlage für die Automatisierung wird ein macOS-System und ein physisches Gerät, bspw. ein iPhone, mit iOS benötigt. Auf dem macOS-System sollen neben Xcode auch alle, in Abschnitt 3.7 genannten, ausgewählten Werkzeuge lokal installiert sein. Installationshinweise sind in Abschnitt D.1 zu finden. Eine Ausnahme stellen GitLab und SonarQube dar, da diese bereits über das Netzwerk zur Verfügung stehen.
Alle Werkzeuge sollen für die Automatisierung über jeweils eigene Shell-Skripte, die den Startbefehl beinhalten, aufgerufen werden. Somit können die Skripte sowohl von GitLab CI/CD als auch von Jenkins genutzt werden.

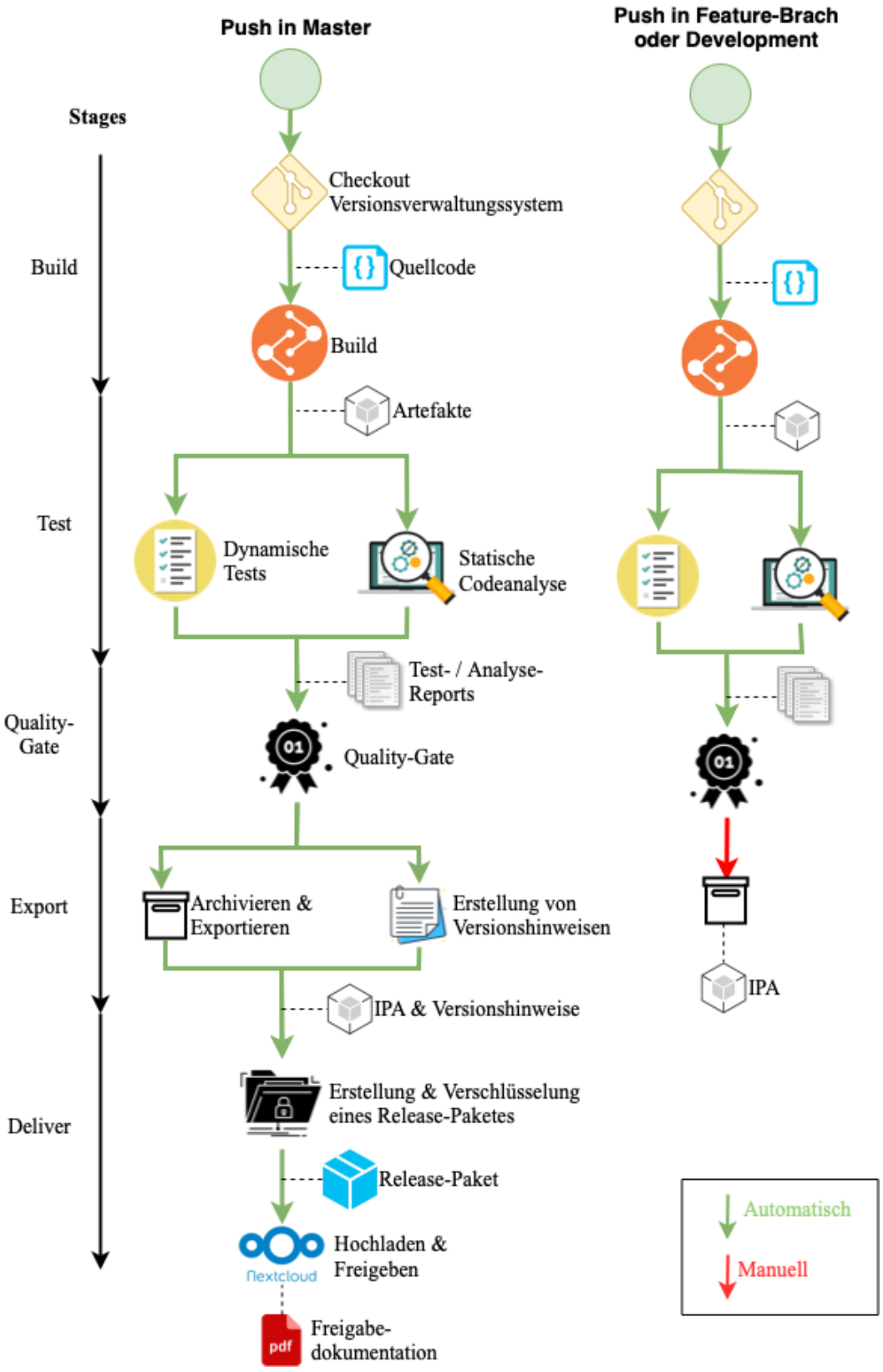

Abbildung 4.0.1: Konzeption der Pipeline (eigene Darstellung)

Die Unit-Tests sollen mittels Xcodebuild ausgeführt werden, UI- und E2E-Tests exemplarisch ebenso. Als Test-Backend soll dafür lokal mittels Docker, Python und Flask[87] eine einfache REST-API bereitgestellt werden.

Werkzeuge für statische Tests sollen den Anforderungen entsprechend konfiguriert werden, wenn nötig und möglich. In SwiftLint sollten dementsprechend ggf. Regeln angepasst oder eigene geschrieben werden, um die geforderten Testfälle abzudecken. In SonarQube muss ein Projekt angelegt und der Aufruf des Scanners passend konfiguriert werden. Des Weiteren soll das Ergebnis des vorhandenen Quality Gates in SonarQube in die Pipeline eingebunden werden. Jedoch nur als Hinweis und nicht als potentiellen Fehlergrund.

Für den Export der Anwendung mittels Xcodebuild wird davon ausgegangen, dass sich alle notwendigen Zertifikate, privaten Schlüssel und Provisioning Profiles auf dem, für die Automatisierung verwendeten, macOS-System befinden. Um diese nach Ablauf oder bei Änderungen automatisch zu aktualisieren empfiehlt sich das Vorgehen nach dem Codesining-Guide, wie in Unterabschnitt 2.1.3 erwähnt. Diese Umsetzung ist jedoch nicht gefordert und kann als Aussicht genannt werden. Hilfsweise kann auch das ohnehin installierte Xcode zusätzlich als Server fungieren und einen einfachen Bot besitzen, der lediglich zum manuellen Aktualisieren der Zertifikate und Profile genutzt wird.
An dieser Stelle könnte Fastlane eingesetzt werden, um schnell und unkompliziert alle gewünschten Targets (auch lokal ohne GitLab oder Jenkins) zu exportieren.

Das Release-Paket soll die entstandene `*.ipa`-Datei vom Export und entsprechende Versionshinweise beinhalten. Zur Erstellung dieser soll eine `release-notes.adoc`-Datei im Projekt dienen. Aus dieser soll mittels AsciiDoctor dann eine PDF-Datei generiert werden.
Mit Hilfe eines Skripts sollen zufällige Passwörter für die Verschlüsselung und Freigabe des Release-Paketes erzeugt, das Release-Paket mittels 7-Zip erstellt und verschlüsselt, mit der Nextcloud API[88] (Webdav für Dateiupload und OCS Share API für Erstellung der Freigabe) kommuniziert und eine PDF-Datei mit allen relevanten Informationen (Passwort für Release-Paket und Freigabe, Freigabelink, Ablaufdatum der Freigabe) erzeugt werden. Außerdem sollen mögliche Fehler zum Exit-Code Eins mit entsprechenden Logs führen. Das Skript soll in Python geschrieben werden, da diese Sprache sehr weit verbreitet, einfach zu verstehen und leicht zu erlernen ist.

Abbildung 4.0.2: Beispielanwendung Todo-App (eigene Darstellung)

Für die Umsetzung und Evaluierung wurde mit Xcode ein Beispielprojekt mit dem Namen *DemoAppForPipeline* angelegt. Es stellt eine einfach Todo-Anwendung dar, wie in Abbildung 4.0.2 zu sehen. Die Beispielanwendung ermöglicht gezielte und unkomplizierte Tests sowie prototypische Implementierungen. Daraus gewonnenen Ergebnisse lassen sich ebenso auf die konkrete iOS-Anwendungen übertragen, die mit Swift und Xcode entwickelt werden. Der gesamte Quellcode ist der beigelegten CD zu entnehmen, siehe Anhang E.

[87] Vgl. Flask: https://flask.palletsprojects.com/en/2.1.x/ (Zugegriffen am 13. Juli 2022)
[88] Vgl. Clients and Client APIs: https://docs.nextcloud.com/server/latest/developer_manual/client_apis/index.html (Zugegriffen am 13. Juli 2022)

Kapitel 5

Exemplarische Umsetzung der Automatisierung mittels ausgewählter Werkzeuge

5.1 Automatisierte Qualitätssicherung

5.1.1 Dynamische Tests

Unit-Tests mit XCTest

Unit-Tests werden in einem gesonderten Test-Target organisiert. Als Beispiel wurde ein *DemoForPipelineTests*-Target erzeugt, dass folgenden Beispieltest enthält.

Quellcode 5.1.1: Unit-Test-Beispiel

```
1 import XCTest
2 @testable import DemoAppForPipeline
3
4 class DemoAppForPipelineTests: XCTestCase {
5     func testExample() throws {
6         // This is an example of a functional test case.
7     }
8 }
```

Die Annotation `@testable` sorgt dafür, dass alle Tests den selben Zugriff auf den Quellcode des Targets *DemoForPipeline* haben, wie der Rest der Anwendung. Jede Testmethode benötigt zusätzlich den Prefix „test“, damit Xcode diese als Test erkennt und aufrufen kann.
Mithilfe des folgenden Xcodebuild-Befehls können die gesamten Tests des Targets schließlich gestartet werden. (AM4) Testergebnisse sind der Kommandozeile zu entnehmen. Zusätzlich wird eine spezielle `*.xcresult`-Datei mit den Testergebnissen erzeugt, die sich mit Xcode öffnen lässt.

```
# xcodebuild clean test \
    -scheme DemoAppForPipeline \
    -enableCodeCoverage YES \
    -destination 'platform=iOS Simulator,name=iPhone 13' \
    -only-testing DemoAppForPipelineTests \
    -derivedDataPath build/ \
    | xcbeautify && exit ${PIPESTATUS[0]}
```

Über die Option `-only-testing` können auch einzelne Testklassen oder -cases ausgeführt werden. `xcbeautify`[89] sorgt dafür, dass der äußerst ausführliche und teils unverständliche Output des Xcodebuild-Befehls in eine verständlichere und übersichtlichere Form umgewandelt wird.
Mittels `-destination 'platform=iOS,name=PHYSICAL_DEVICE_NAME'` können neben Simulatoren auch physische Geräte zur Durchführung der Tests ausgewählt werden. (AM5, AS9) Dazu muss das physische Gerät eine Internetverbindung haben und darf nicht gesperrt oder durch einen Code gesichert sein.

Die konzeptionellen Phasen *Build* und *Test* werden an dieser Stelle zusammengefasst, da Xcodebuild mit dem Testbefehl die Anwendung automatisch baut und danach testet. (AM3)

UI- und E2E-Tests mit XCTest

Um UI- und E2E-Tests durchführen zu können wird ein UI-Test-Target benötigt: *DemoAppForPipelineUITests*. Mit Hilfe von Xcode und der Funktion zum Aufnehmen eines Tests kann dann bspw. der in Quellcode D.3.1 dargestellte E2E-Test erstellt werden. Für die Aufnahme oder Ausführung der Tests muss das Test-Backend mit Testdaten vorhanden sein. Dieses kann mittels Docker und Docker Compose einfach gestartet werden: (AS7, AS8)

```
# docker-compose -f backend/docker-compose.yml up -d
```

Dies beinhaltet zwei miteinander verknüpfte Dienste: Eine Datenbank und eine Python REST-API, die lokal auf dem jeweiligen Host zur Verfügung stehen. Die Backend-URL zum Testen wird dabei mittels der in Unterabschnitt 2.1.2 vorgestellten Build-Configuration-Files in die Anwendung übergeben. Mögliche Testdaten können beim Starten des Datenbank-Containers via SQL-Dumb in die Datenbank übertragen werden.Der Befehl zum Starten der Tests ähnelt dem zum Starten der Unit-Tests, siehe Abschnitt 5.1.1 lediglich der Name des Targets ist anders.
Für UI-Tests ist der Ablauf prinzipiell ähnlich, nur das dabei nicht zwingend ein Backend mit Testdaten zur Verfügung stehen muss. Testdaten und einzelnen Komponenten müssten durch funktionale Platzhalter (Mock-Objekte) repräsentiert werden. Dafür benötigt es eine passende Architektur im protokollorientierten Design.[90]

5.1.2 Statische Tests und Quality Gate

SwiftLint

In der Konfigurationsdatei von SwiftLint, siehe Quellcode D.2.1, wurden die folgenden Regeln angepasst und definiert. Um Klassen zu erkennen, die mehr als 400 Code-Zeilen enthalten wurde die Regel „type_body_length“ angepasst. (AM8) Weiterhin wurden die Regeln „unused_declaration“ und „unused_import“ eingebunden. (AM7) Zusätzlich wurde einfache Regeln für die Erkennung von Singletons ergänzt. (AS2) Eine Regel für fehlende Completion-Aufrufe gibt es nicht. Deren Definition mit regulären Ausdrücken bedarf einer tieferen Beschäftigung und wurde deshalb als zukünftige Erweiterung eingestuft. (AS1)

Durch die Ausführung des folgenden Befehls analysiert das Werkzeug den spezifizierten Quellcode auf Verstöße gegen allgemeine Programmierrichtlinien (AS3) sowie gegen die eigenen Regeln und schreibt die Ergebnisse in eine definierte Datei. Die Option `--reporter` bestimmt das Ausgabeformat. Neben HTML gibt es auch SonarQube, Xcode und weitere. Mittels `--config` wird die zu verwendende Konfigurations-Datei bestimmt.

```
# swiftlint lint \
    --config .swiftlint.yml \
    --reporter html > swiftlint-report.html
```

[89] Vgl. Xcbeautify: https://github.com/tuist/xcbeautify (Zugegriffen am 13. Juli 2022)
[90] Vgl. Protocols: https://docs.swift.org/swift-book/LanguageGuide/Protocols.html (Zugegriffen am 13. Juli 2022)

SonarQube

Zuerst muss in SonarQube selbst ein eigenes Projekt angelegt werden, in das Scanner ihre Ergebnisse bzgl. Schwachstellen (AS4), Verstöße gegen Programmierrichtlinien (AS3) usw. publizieren.[91] Der Scan wird lokal schließlich wie folgt gestartet:

```
# sonar-scanner \
    -Dsonar.host.url=${SONAR_HOST_URL} \
    -Dsonar.login=${SONAR_LOGIN}  \
    -Dsonar.projectKey=${SONAR_PROJECT_KEY} \
    -Dsonar.sources=DemoAppForPipeline/ \
    -Dsonar.branch.name=${BRANCH} \
    -Dsonar.coverageReportPaths=generic-unittest-coverage.xml \
    -Dsonar.qualitygate.wait=true
```

Zur Zuordnung der Analyseergebnisse in SonarQube wird der Branch-Name verwendet. Der Parameter `-Dsonar.qualitygate.wait` bestimmt, ob der Analyseschritt auf das Ergebnis des Quality Gates („sonar way“) warten soll oder nicht. Wenn ja, dann wirkt sich das Ergebnis entsprechend auf den Exit-Code aus, wodurch die Pipeline wie gewünscht reagieren kann. Das Ergebnis des Quality Gate soll im konkreten Fall der Unterstützung dienen, weshalb das Ergebnis zwar relevant ist aber nicht zum Abbruch der Pipeline führen soll. Um dies möglich zu machen, sollte das verwendete Automatisierungssystem an dieser Stelle einen konkreten Hinweis geben, dass etwas nicht stimmt, wie beispielhaft in Abbildung B.0.6 gezeigt (orangefarbener Kreis mit Ausrufezeichen). Weitere Parameter und deren Beschreibungen sind der Dokumentation zu entnehmen.[92]
Um die Testabdeckung in SonarQube anzuzeigen, muss ein Skript[93] verwendet werden, das aus dem XCResult der vorher ausgeführten Tests ein verständliches Format für SonarQube generiert. Die erzeugte Datei wird dann mittels des Parameters `sonar.coverageReportPaths` importiert. (AS10)
Zur Überprüfung von externen Komponenten kann das OWASP Dependency-Check Plugin[94] eingebunden und schließlich mittels verschiedener Parameter beim Scan-Aufruf verwendet werden. (AS6)
Aufgrund der Tatsache, dass dafür Konfigurationsarbeit an der bestehenden SonarQube-Instanz nötig wird, die nicht selbst durchgeführt werden kann, soll im Rahmen der Arbeit erst einmal darauf verzichtet werden.
Die Ergebnisse der gesamten Analyse und der Importe werden anschließend im Webinterface dargestellt, wie in Abbildung B.0.4 zu sehen. Zur Erkennung der Testfälle werden die Standardregeln von SonarQube verwendet. Eigene Regeln können nur mittels SwiftLint definiert werden, dessen Report über eine Schnittstelle importiert werden kann.

Code Climate CLI

Code Climate CLI bedarf keiner weiteren Konfiguration. Die Standard-Checks zu Maintainability, wie bereits in Abschnitt 2.4.4 vorgestellt, werden mittels des folgenden Befehls durchgeführt. Für alle Checks gelten die Standardparameter.[95] (AM6)

```
# codeclimate analyze -f html DemoAppForPipeline/ >
   codeclimate-report.html
```

[91] Vgl. Analyzing a Project: https://docs.sonarqube.org/latest/setup/get-started-2-minutes/ (Zugegriffen am 13. Juli 2022)
[92] Vgl. Analysis Parameters: https://docs.sonarqube.org/latest/analysis/analysis-parameters/ (Zugegriffen am 13. Juli 2022)
[93] Vgl. Skript: https://github.com/SonarSource/sonar-scanning-examples/blob/master/swift-coverage/swift-coverage-example/xccov-to-sonarqube-generic.sh (Zugegriffen am 13. Juli 2022)
[94] Vgl. „Dependency-Check Plugin for SonarQube 8.x and 9.x“: https://github.com/dependency-check/dependency-check-sonar-plugin (Zugegriffen am 13. Juli 2022)
[95] Vgl. Default Checks: https://docs.codeclimate.com/docs/advanced-configuration#default-checks (Zugegriffen am 13. Juli 2022)

Mobsfscan

Mobsfscan bedarf keiner Konfiguration und wird wie folgt ausgeführt. Für den Report kann auch ein mit SonarQube kompatibles Format gewählt werden. (AS4)

```
# mobsfscan --html -o mobsfscan-report.html DemoAppForPipeline/
```

Gitleaks

Gitleaks wird über den folgenden Aufruf mit einer Standardkonfiguration gestartet. Eigene Regeln für die Erkennung können in einer `gitleaks.toml`-Datei spezifiziert werden. Um den Umfang der Arbeit im Rahmen zu halten, wurde darauf verzichtet. (AS5)

```
# gitleaks detect \
    --report-format="json" \
    --report-path="gitleaks-report.json" \
    --source="DemoAppForPipeline"
```

5.2 Automatisierte Bereitstellung

Archivieren und Exportieren der iOS-Anwendung

Für das Archivieren und Exportieren wird Xcodebuild mit den folgenden Befehlen verwendet, die nacheinander ausgeführt werden müssen. In der `exportOptions.plist`-Datei wird festgelegt, welches Provisioning Profile, Zertifikat und welche Methode verwendet werden soll. Ein Beispiel ist Quellcode D.2.2 zu entnehmen. Die erstellte `DemoAppForPipeline.ipa`-Datei ist dann Zusammen mit detaillierten Exporthinweisen in dem export-Ordner zu finden.(AM9)

```
# xcodebuild clean archive \
    -scheme DemoAppForPipeline \
    -configuration Release  \
    -archivePath archive/app.xcarchive

# xcodebuild -exportArchive \
    -archivePath archive/app.xcarchive \
    -exportPath export/ \
    -exportOptionsPlist pipeline/exportOptions.plist
```

Erzeugung einer PDF-Datei mit Versionshinweisen

Als Grundlage für die Erzeugung der Versionshinweise dient die `release-notes.adoc`-Datei. Diese nutzt das AsciiDoc-Format und kann wie in Quellcode 5.2.1 dargestellt aussehen. Mit Hilfe des Befehls für AsciiDoctor: `# asciidoctor-pdf release-notes.adoc` wird die adoc-Datei in eine PDF-Datei konvertiert. Das Resultat ist in Abbildung 5.2.1 zu sehen. (AM10)

Quellcode 5.2.1: Release-Notes im AsciiDoc-Format

```
1 = *Versionshinweise*
2
3 == Release 1.0
4 Release-Datum: 01.06.2022
5
6 * Beispielfunktion 1 hinzugefuegt
7 * Beispielfunktion 2 hinzugefuegt
```

Abbildung 5.2.1: Release-Notes als PDF-Datei (eigene Darstellung)

Paketieren, Komprimieren und Verschlüsseln des Release-Paketes

Exportdateien und Release-Notes werden anschließend mit Hilfe eines Python-Skriptes weiterverarbeitet. Dabei erfolgt zuerst die Generierung eines zufälligen Passwortes für die Verschlüsselung des Release-Paketes. Dafür wird die Bibliothek *random-password-generator*[96] benutzt. Das Paketieren, Komprimieren und Verschlüsseln wird wie folgt durchgeführt: (AM12, AS16)

Quellcode 5.2.2: Paketierung, Komprimierung und Verschlüsselung des Release-Paketes

```
1 subprocess.run(['7z', 'a', '-tzip', '-pPASSWORD',
      'release-package.zip', 'export/', 'release-notes.pdf',
      '-mem=AES256'])
```

Nextcloud: Upload und Freigabe

Als nächstes wird das komprimierte und verschlüsselte Release-Paket mittels der Nextcloud API in die Nextcloud hochgeladen und dort freigegeben. (AM13, AS17) Für die Authentifizierung wird ein Nutzername und (App-)Passwort benötigt. Die Erstellung eines Ordners wird mittels Python und der Bibliothek *requests*[97] wie folgt ausgeführt.[98]

Quellcode 5.2.3: Nextcloud-Upload

```
1     createFolderRequestURL =
          '{}/remote.php/dav/files/{}/{}'.format(NEXTCLOUD_URL,
          NEXTCLOUD_USER, UPLOAD_FOLDER)
2
3     createFolderResponse = requests.request('MKCOL',
          createFolderRequestURL, auth=(NEXTCLOUD_USER,
          NEXTCLOUD_APP_PASSWORD))
```

[96] Vgl. Random password Generator: https://pypi.org/project/random-password-generator/ (Zugegriffen am 13. Juli 2022)
[97] Vgl. Requests: https://pypi.org/project/requests/ (Zugegriffen am 13. Juli 2022)
[98] Jeder nicht existierende Ordner im Upload-Pfad muss mit jeweils einem Request erstellt werden. Vorhandene Ordner werden nicht überschrieben und führen zu einer Fehlermeldung.

Für die Freigabe sind folgende Anweisungen nötig. Eine Liste aller Freigabeoptionen ist der API-Dokumentation zu entnehmen.

Quellcode 5.2.4: Erstellung einer Nextcloud-Freigabe

```
1  expireDate = (date.today() +
       timedelta(days=SHARE_DURATION)).strftime('%Y-%m-%d')
2
3  headers = {
4      'OCS-APIRequest': 'true',
5      'Content-Type': 'application/x-www-form-urlencoded'
6  }
7
8  shareOptions =
       'path={}/{}&shareType=3&publicUpload=false&password={}
       &permissions=1&expireDate={}'.format(UPLOAD_FOLDER,
       RELEASE_PACKAGE_FILE, sharingPassword, expireDate)
9
10 createShareRequestURL =
       '{}/ocs/v2.php/apps/files_sharing/api/v1/shares'
       .format(NEXTCLOUD_URL)
11
12 createShareResponse = requests.post(createShareRequestURL,
       headers=headers, data=shareOptions, auth=(NEXTCLOUD_USER,
       NEXTCLOUD_APP_PASSWORD))
```

Erzeugung einer PDF-Datei mit Bereitstellungsinformationen

Als letztes wird mit Hilfe der Bibliothek *PyFPDF*[99] eine PDF-Datei erzeugt, die den Namen der freigegebenen Datei, Passwort für Verschlüsselung, Freigabelink, Freigabepasswort und Ablaufdatum (standardmäßig auf sieben Tage ab Freigabeerstellung gesetzt) der Freigabe enthält. Das Resultat ist in Abbildung 5.2.2 zu sehen. (AS18)

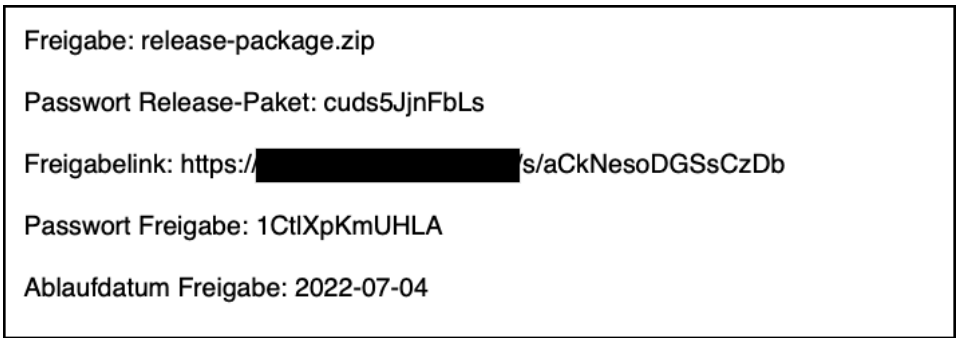

Abbildung 5.2.2: Bereitstellungsinformation(eigene Darstellung)

[99] Vgl. FPDF for Python: https://pyfpdf.readthedocs.io/en/latest/ (Zugegriffen am 13. Juli 2022)

5.3 Automatisierung mit GitLab CI/CD

Auf dem macOS-System muss zuerst GitLab-Runner installiert werden.[100] Anschließend wurde für das konkrete GitLab-Projekt ein Runner (Executor: Shell) auf dem Mac registriert.[101]. Dieser führt Befehle in der Kommandozeile aus und kann über die Tags `macOS, shell` genutzt werden. Zur parallelisierten Abarbeitung von Jobs muss die Konfigurationsdatei des Runners angepasst werden.

Um GitLab CI/CD zu nutzen, muss sich lediglich eine `.gitlab-ci.yml`-Datei im Projektordner befinden. In dieser wurden zunächst alle Stages definiert und generelle Regeln bzw. Bedingungen festgelegt, wann die Pipeline gestartet werden soll. (AM1)

Quellcode 5.3.1: Stages und Workflow in gitlab-ci.yml

```
stages:
    - test
    - qualitygate
    - export
    - deliver

# runs either branch pipelines or merge request pipelines
# Rules are evaluated in order until the first match.
workflow:
    rules:
        - if: $CI_PIPELINE_SOURCE == "merge_request_event" # run merge request pipeline only
        - if: $CI_COMMIT_BRANCH && $CI_OPEN_MERGE_REQUESTS # prevents duplicate pipelines
          when: never
        - if: $CI_COMMIT_BRANCH # run branch pipeline only
```

Die Regeln werden der Reihe nach geprüft. Bei der ersten Übereinstimmung wird die Pipeline gestartet. Nachfolgende Regeln werden nicht mehr geprüft. Zeichenketten mit einem vorangestellten Doller-Zeichen ($) sind Umgebungsvariablen und werden von GitLab bereitgestellt. Über die Projekteinstellungen in GitLab können auch eigene Umgebungsvariablen angelegt werden.

Als repräsentatives Beispiel für die Definition eines Job wurde jener zum Archivieren und Exportieren der Anwendung gewählt, wie in Quellcode 5.3.2. Alle anderen sind nach dem gleichen Prinzip aufgebaut. Zu erwähnen ist, dass für jeden Job standardmäßig ein `git fetch` durchgeführt wird und alle Jobs einer Stage parallel ausgeführt werden, sofern möglich. Beides kann allerdings `gitlab-ci.yml`-Datei angepasst werden. (AM2)
Die Regeln des Jobs bestimmen, ob und wie dieser der Pipeline hinzugefügt werden soll, bspw. nur bei bestimmten Branches. (AM18) Sie werden genauso abgearbeitet wie Workflow-Rules. Wenn die Regel `manual` enthält, so wartet die Pipeline zum Fortfahren auf eine manuelle Bestätigung durch eine berechtigte Person, wie in Abbildung B.0.6 (Zahnrad und Play-Button) dargestellt. (AM17)
Im Skript-Abschnitt wird ein Skript aufgerufen, dass die entsprechenden bereits vorgestellten Befehle zum Archivieren und Exportieren der iOS-Anwendung enthält. Weitere Skript- oder Befehlsaufrufe können nach selbem Prinzip darunter geschrieben werden. Zusätzlich kann noch ein `before_script` und `after_script`-Abschnitt definiert werden, wenn gewünscht. In diesen wurden bspw. die Aufrufe zum Hoch- und Hinunterfahren der Docker-Container des Test-Backends für E2E-Tests platziert. Skripte, die einen Exit-Code ungleich Null liefern führen zum Fehler. Soll der Fehler ignoriert werden, so kann dies explizit mit der Option `allow_failure: true` im Job ergänzt werden, was zu einem Ausrufezeichen als Status führt, wie in Abbildung B.0.6 zu sehen.

[100] Vgl. Install GitLab Runner on macOS: https://docs.gitlab.com/runner/install/osx.html (Zugegriffen am 13. Juli 2022)
[101] Vgl. Registering runners: https://docs.gitlab.com/runner/register/ (Zugegriffen am 13. Juli 2022)

Quellcode 5.3.2: Job in gitlab-ci.yml

```
export_app:
    rules:
        # automatic execution
        - if: $CI_COMMIT_BRANCH == "master"
        # manual execution for all other branches
        - if: $CI_COMMIT_BRANCH
        when: manual
    stage: export
    tags:
        - macOS
        - shell
    script:
        - pipeline/scripts/archive.sh
        - pipeline/scripts/export.sh
    artifacts:
        name: "export-files"
        paths:
            - export/
```

Durch die Ausführung des Befehls entstandene Log-Meldungen und Artefakte können über das Webinterface eingesehen und heruntergeladen werden, siehe Abbildung B.0.7. Die Artefakte werden dabei durch den `artifacts`-Abschnitt gesichert. Jobs können auch auf benötigte Artefakte vorangegangen Jobs zugreifen indem sie den Namen des jeweiligen Jobs in ihrem `dependencies`-Abschnitt aufführen.
Die gesamte Pipeline, ihr Status, erzeugte Artefakte, zugehörige Optionen (Abbrechen, Neustarten) und weitere Informationen sind umfangreich und übersichtlich im Webinterface von GitLab dargestellt. Für den Zugang werden Nutzername und Passwort benötigt. Hier besteht auch die Möglichkeit manuell eine Pipeline zu starten. (AM14, AM15, AM16)

5.4 Automatisierung mit Jenkins

Damit Jenkins Befehle auf einem macOS-System ausführen kann muss zuerst ein dezidierter Knoten mit Agent angelegt werden.[38] Für den Agent wird eine Installation von Java auf dem Zielsystem vorausgesetzt. Idealerweise dieselbe Version, die auch der Jenkins-Server nutzt. Dem Knoten wurde das Label `macOS` zugewiesen, der all seine möglichen Agents logisch gruppiert.
Als nächstes muss ein neues Multibranch-Pipeline-Element[102] in Jenkins angelegt und GitLab entsprechend konfiguriert[103] werden. Als Trigger wurden in GitLab die Push-Events aktiviert. (AM1) Als Grundlage für die Pipeline wurde in Jenkins das *Jenkinsfile* ausgewählt.

Im Gegensatz zu GitLab CI/CD führt Jenkins keine Schritte o.ä. selbstständig aus. Es muss alles explizit im Jenkinsfile definiert werden. Ein Ausschnitt wurde in abgebildet. Aufbau der anderen Stages folgt auch hier dem gleichen Prinzip.
Die einzelnen Aufgaben in den Stages werden standardmäßig nacheinander ausgeführt. Auch hier wird die Pipeline abgebrochen, sobald ein Exit-Code ungleich Null auftritt. Zur Fehlerbehandlung bietet Jenkins `catchError`[104]. Eine parallele Ausführung kann, wie in der Test-Stage angedeutet, definiert werden. Mittels des `when`-Konstrukts können bestimmte Bedingungen überprüft werden,

[102] Vgl. „Creating a Multibranch Pipeline“: https://www.jenkins.io/doc/book/pipeline/multibranch/ (Zugegriffen am 13. Juli 2022)
[103] Vgl. Jenkins integration: https://docs.gitlab.com/ee/integration/jenkins.html (Zugegriffen am 13. Juli 2022)
[104] Vgl. Pipeline: Basic Steps - catchError: Catch error and set build result to failure: https://www.jenkins.io/doc/pipeline/steps/workflow-basic-steps/ (Zugegriffen am 13. Juli 2022)

die darüber entschieden, ob und wie die Aufgaben der Stage ausgeführt werden. (AM18) Mit Hilfe von `input` wartet die Pipeline auf eine Eingabe über das Webinterface. Im einfachsten Fall ist das nur ein Klick auf eine Schaltfläche, damit die Pipeline fortfahren kann, wie in Abbildung B.0.8 dargestellt. (AM17) Möglich sind aber auch textuelle Eingaben, um folgende Arbeitsschritte parametrisieren zu können.
Die Anweisung `archiveArtifacts` sichert definierte Artefakte und stellt sie über das Webinterface zur Verfügung, wie in Abbildung B.0.9 dargestellt. Auch hier ist der Zugang mittels Nutzername und Passwort gesichert. (AM14, AM15)

Genauso wie bei GitLab kann die gesamte Pipeline und alles weitere über das Webinterface eingesehen, kontrolliert und gesteuert werden. (AM16) Neben der standardmäßigen Jenkins-Ansicht kann auch das weitaus elegantere Blue Ocean[105] genutzt werden, wie in Abbildung B.0.8 dargestellt.

[105] Vgl. „Create a Pipeline in Blue Ocean“: https://www.jenkins.io/doc/tutorials/create-a-pipeline-in-blue-ocean/ (Zugegriffen am 13. Juli 2022)

Quellcode 5.4.1: Jenkinsfile

```
pipeline {
    agent { label 'macOS' }

    stages {
        stage('Git Checkout') {
            steps {
                cleanWs() // delete all files and folders in
                    current workspace
                checkout scm // new checkout (AM2)
            }
        }

        stage('Test') {
            parallel {
                stage('Unit-Tests') { ... }
                stage('UI-Tests') { ... }
                ...
            }
        }

        stage('Manual-export') {
            when {
                beforeInput true
                not { branch 'main' }
            }
            input {
                message "Exportieren?"
                ok "ja"
            }
            steps {
                sh "pipeline/scripts/archive.sh"
                sh "pipeline/scripts/export.sh"
                archiveArtifacts artifacts: 'export/'
            }
        }

        stage('Auto-export') {
            when {
                branch 'main'
            }
            steps {
                sh "pipeline/scripts/archive.sh"
                sh "pipeline/scripts/export.sh"
                archiveArtifacts artifacts: 'export/'
            }
        }
    }
}
```

Kapitel 6

Evaluierung

6.1 Ziele und Vorgehen

Zur Entscheidungsfindung für oder gegen ein ausgewähltes Werkzeug sowie zur Überprüfung, ob und inwieweit die in Abschnitt 3.6 definierten Kriterien erfüllt werden, soll auf Grundlage der exemplarischen Umsetzung eine Evaluierung durchgeführt werden. Da die Automatisierung mittels GitLab CI/CD und Jenkins prinzipiell alle vorgestellten weiteren Werkzeuge nutzen kann, erfolgt die Evaluierung getrennt nach Werkzeugen für dynamische und statische Tests. Danach folgt die Evaluierung der Automatisierungswerkzeuge und schließlich die Betrachtung der Bereitstellung mit den restlichen eingesetzten Werkzeugen.

6.2 Werkzeuge für dynamische Tests

Zur Umsetzung der dynamischen Tests wurden lediglich XCTest für das Erstellen und Xcodebuild für das Ausführen herangezogen. Ein Vergleich mit anderen Werkzeugen ist somit nicht möglich, wodurch die Evaluierung kurz in textueller Form erfolgt, ohne Noten zu vergeben.
Die Umsetzung hat gezeigt, dass sowohl Unit- als auch UI- und E2E-Tests leicht mit XCTest und Xcode erstellt werden können. UI- oder E2E-Tests können dabei einfach über die Recording-Funktion von Xcode aufgenommen werden. Die Durchführung der Tests kann effektiv über Xcodebuild gestartet und vielseitig parametrisiert werden, um bspw. die Testabdeckung mit zu erfassen oder nur einzelne Tests zu starten. XCTest und Xcodebuild erfüllen dabei kein Ausschlusskriterium und arbeiten effizient, da sie standardmäßig durch das ohnehin benötigte Xcode vorhanden sind und sich nahtlos in Apples Werkzeugkette integrieren. Bei fehlgeschlagenen Tests ist der Exit-Code von Xcodebuild nicht Null, wodurch die Automatisierung in der Pipeline entsprechend darauf reagieren kann. Weiterhin verfügt vor allem Xcodebuild über eine ausführliche Dokumentation mit vielen Beispielen. XCTest muss hier teilweise Abstriche machen, da die Dokumentation seitens Apple manchmal etwas zu kurz ausfällt. Durch die Verwendung von XCTest für Unit-Tests besteht allerdings schon viel Erfahrung, was das Defizit teilweise ausgleicht. Als weiterführende Literatur oder ergänzende Dokumentation empfiehlt sich an dieser Stelle das Buch „Testing Swift“ von Paul Hudson.[106]
Als letztes ist noch anzumerken, dass die Tests standardmäßig innerhalb einer sogenannten „Host Application“ durchgeführt werden, wobei für jeden einzelnen Test die ganze Anwendung gestartet wird. Während das bei UI- und E2E-Tests nicht anders geht, kostet diese Vorgehensweise für Unit-Tests viel Zeit, da immer der gesamte Startprozess durchlaufen wird.

[106] Vgl. „Testing Swift“: https://www.hackingwithswift.com/store/testing-swift (Zugegriffen am 13. Juli 2022)

6.3 Werkzeuge für statische Tests

Zur Durchführung von statischen Tests wurden mehrere mögliche Werkzeuge für eine Umsetzung herangezogen. Als Hilfe bei der Evaluierung wurde die Beispielanwendung um eine Testklasse ergänzt, sodass unterschiedliche Testfälle abgedeckt und deren Ergebnisse bewertet werden können. Ein Ausschnitt der Testklasse ist Quellcode D.3.2 zu entnehmen.
Die Testklasse umfasst mehrere Variablen, Konstanten und Methoden, um ungenutzte Deklarationen, duplizierten Code und komplexe Methoden zu erkennen. Zur Erkennung einer hohen Komplexität wurde eine Methode mit vielen verschachtelten if-else- und switch-case-Anweisungen implementiert. Eben diese wurde zudem dupliziert, um zu prüfen, inwieweit Codeduplikate erkannt werden. Zusätzlich wurde die Testklasse anhand einer Methode so erweitert, dass sie (inklusive Leerzeilen und Kommentaren) über 400 Zeilen Code enthält. Außerdem existiert eine Implementation einer Methode mit fehlendem Completion-Aufruf sowie die eines Singletons.
Testfälle für Secrets, Schwachstellen oder allgemeine Programmierrichtlinien wurden nicht extra erstellt, da dessen umfangreiche Ausarbeitung sonst den Rahmen der Arbeit übersteigen würde. Es wird davon ausgegangen, dass die Werkzeuge grundsätzliche korrekt funktionieren.

Für SwiftLint, siehe Tabelle C.0.1, Code Climate CLI, siehe Tabelle C.0.2, und SonarQube, siehe Tabelle C.0.3, wurden die Bewertungsergebnisse inklusive Begründungen jeweils in einer eigenen Tabelle zusammengefasst, da alle drei mehrere Kriterien abdecken. Eine Zusammenfassung und Gegenüberstellung ist Tabelle 6.3.1 zu entnehmen. Für Mobsfscan, Gitleaks und OWASP Dependency-Check wurden keine gesonderten Tabellen erstellt, da sie jeweils nur auf eine Aufgabe spezialisiert sind. Die Bewertung sollen im Rahmen der folgenden Gesamtbetrachtung zu den Werkzeugen der statischen Tests mit stattfinden.

Kriterium	**SwiftLint**	**Code Climate CLI**	**SonarQube**
Quality Gate vorhanden	×	×	✓
stellt Testabdeckung dar	×	×	✓
erkennt bekannte Schwachstellen	×	×	✓+
erkennt Secrets im Quellcode	✓	✓*	✓*
führt Dependency-Scans durch	×	×	✓*
erkennt Verstöße gegen allgemeine Programmierrichtlinien	✓	✓*	✓+
erkennt duplizierten Code	×	✓+	✓
erkennt ungenutzte Deklarationen	×	×	×
erkennt Klassen mit mehr als 400 LoC	✓	✓*	✓*
erkennt fehlende Completion-Aufrufe	k.A.	k.A.	k.A.
erkennt Singletons	(✓)	(✓*)	(✓*)
misst Lines of Code	✓+	✓	✓+
misst zyklomatische Komplexität	✓+	✓	✓+
misst Anzahl an vorhandenen Schwachstellen	×	×	✓
misst Anzahl an Verstößen gegen allgemeine Programmierrichtlinien	✓	×	✓
Möglichkeit zur Definition eigener Regeln	✓+	✓*	✓*
umfangreiches Reporting	✓+	✓	✓+
aufaddierte Gewichtungen erfüllter Kriterien	**14 (+2)**	**15 (+2)**	**28 (+2)**

Legende:
✓... Kriterium erfüllt, (✓)... Kriterium teilweise erfüllt, ×... Kriterium nicht erfüllt,
✓*... Kriterium erst durch Einbindung eines externen Werkzeuges erfüllt (meist SwiftLint),
✓+... Kriterium wird besser erfüllt als bei anderen Werkzeugen (siehe einzelne Tabellen)

Tabelle 6.3.1: Gegenüberstellung von Werkzeugen zu statischen Tests

SwiftLint ist ein Werkzeug mit vielen Möglichkeiten. Dank integrierter und konfigurierbarer Regeln sowie der Möglichkeit zur Erstellung eigener Regeln kann es viele Kriterien abdecken. Des Weiteren erfüllt SwiftLint in akzeptablen Maße die in Tabelle 3.6.1 aufgeführten allgemeinen Kriterien. Dementsprechend wird es von vielen genutzt und geschätzt (16200 Stars bei GitHub) sowie dessen Entwicklung aktiv vorangetrieben.[57] Es lässt sich über die Kommandozeile leicht bedienen, über eine Konfigurationsdatei einfach konfigurieren und ist in verständlichem, ausreichendem Maße dokumentiert. Außerdem weist es eine explizite CI-Kompatibilität auf.
Code Climate CLI wird von GitLab genutzt. Es erkennt zuverlässig duplizierten Code und schätzt die Wartbarkeit ein. Darüber hinaus können verschiedene Plugins, darunter SwiftLint, eingebunden werden, wodurch auch jene Funktionalitäten von SwiftLint mit zur Verfügung stehen. Es lässt sich leicht bedienen und ähnlich wie SwiftLint konfigurieren. Allerdings weist es keine CI-Kompatibilität auf. Ergebnisse wirken sich nicht auf den Exit-Code aus, wodurch entstandene Reports immer kontrolliert werden müssen und eine Pipeline nicht automatisch abgebrochen werden kann, um frühestmöglich Feedback bereitzustellen.
Wie aus Tabelle 6.3.1 hervorgeht, deckt SonarQube die meisten Kriterien ab. Dabei fällt auf, dass vor allem durch die Unterstützung externer Werkzeuge viele Funktionalitäten integriert werden können, wodurch die Erfüllung einiger Kriterien überhaupt erst möglich ist. Dank dessen umfangreichen Importfunktionen und Plugins kann bspw. das gesamte Potential von SwiftLint und OWASP Dependency-Checker genutzt werden. Auch eine Testabdeckung kann mit Hilfe von SonarQube importiert werden. Alle Ergebnisse werden dabei übersichtlich und strukturiert in einem Webinterface dargestellt. SonarQube bietet weiterhin eine einfache Bedienung, sehr viele Informationen, wird aktiv weiterentwickelt und geschätzt (7000 Stars bei GitHub). Darüber hinaus kann es über viele verschiedene Projekte hinweg für eine Qualitätssicherung genutzt werden und weist eine umfangreiche Dokumentation auf.
Mobsfscan dient der Schwachstellenanalyse. Es betrachtet im Gegensatz zu SonarQube nicht nur den Quellcode sondern auch dessen Kontext, also den einer nativen iOS-Anwendung. Während SonarQube keine (möglichen) Schwachstellen gefunden hat, weist der Mobsfscan-Report als Information gleich auf mehrere Möglichkeiten hin, wie in Abbildung B.0.10 dargestellt. Neben einem HTML-Format kann der Report auch in einem von SonarQube lesbaren Format erzeugt und entsprechend in SonarQube importiert werden. Beim Testen des Imports trat allerdings ein Fehler auf, der darauf zurückzuführen ist, dass der von Mobsfscan erzeugte Report nicht alle nötigen Parameter enthält, die SonarQube erwartet.
Gitleaks umfasst eine Vielzahl an Standardregeln für die Erkennung von API-Keys, wie bspw. für Twitch, GitHub und viele mehr. Des Weiteren ermöglicht es auch die Definition eigener Regeln und kann darüber hinaus nicht nur für Swift-Projekte eingesetzt werden. Es lässt sich also leicht wiederverwenden, entspricht sonst auch allen geforderten Kriterien, wird aktiv weiterentwickelt und hat über 10000 Stars bei GitHub.

6.4 Allgemeine Automatisierungswerkzeuge

Die Umsetzung hat gezeigt, dass sowohl GitLab CI/CD als auch Jenkins kein Ausschlusskriterium erfüllen und grundlegend allen Anforderungen entsprechen. Das macht sie anhand der aufgestellten Kriterien gut miteinander vergleichbar. Um eine endgültige Entscheidung treffen zu können, wurde der Zielerfüllungsgrad der Kriterien gemäß Abschnitt 3.6 bewertet. Die detaillierten Tabellen für GitLab CI/CD, siehe Tabelle C.0.4, und Jenkins, siehe Tabelle C.0.5, sind dem Anhang zu entnehmen. Eine Zusammenfassung der Ergebnisse ist in Tabelle 6.4.1 dargestellt.
Beide Systeme haben sich in ihren Bereichen etabliert, sind lange auf dem Markt und werden aktiv weiterentwickelt. Beide können Nutzer von einem Verzeichnisdienst via LDAP einbinden, besitzen ein Webinterface und können lokal auf einem Server installiert werden. Weiterhin lassen sich in Beiden verschiedenartige Projekte verwalten und automatisieren - von iOS und Android bis hin zu Webanwendungen oder sonstigem.
Unterschiede lassen sich besonders bei der Effizienz finden. So muss Jenkins erst auf einem Server installiert werden, damit das System verwendet werden kann. Das erfordert neben Hardwareressourcen zusätzlich auch Administrations- und Wartungsaufwand zur Pflege und Aktualisierung. GitLab steht bereits zur Verfügung, wodurch kein extra Server benötigt wird. Des Weiteren ist die GitLab

Dokumentation sehr umfangreich und enthält viele Beispiele wohingegen bei Jenkins die Dokumentation durch die vielen Plugins etwas undurchsichtig wirkt. Darüber hinaus interagiert GitLab CI/CD sehr gut mit den restlichen GitLab-Funktionalitäten, bspw. bei der automatischen Erstellung von Releases[107]. Ein weiterer hervorzuhebender Punkt ist die Gefahr der Herstellerabhängigkeit. GitLab entscheidet selbst darüber, welche Funktionalitäten kostenfrei zur Verfügung stehen und welche nicht. Wenn GitLab sich dazu entscheiden sollte, eine Funktionalität kostenpflichtig zu machen, dann muss gezahlt oder eine Alternative gesucht werden, was wiederum Aufwand bedeutet. Jenkins ist vollständig Open-Source und birgt keine solche Gefahr. Abschließend sei noch erwähnt, dass GitLab und dessen Webinterface sowie Handhabung dem Entwicklungsteam bekannt sind. Es könnte alles in einem System getan werden, wodurch nicht zwischen verschiedenen Systemen gewechselt werden müsste. Jenkins ist in erster Linie neu und erfordert zumindest am Anfang eine kleine Schulung zu dessen Möglichkeiten und Bedienung.

Kriterium	**Gewichtung**	**GitLab CI/CD**		**Jenkins**	
		Note	$\prod$	Note	$\prod$
Geringer Ressourcenverbrauch	3	1	3	3	9
Übersichtliche / informative Benutzeroberfläche	2	1	2	1	2
Vollständige und verständliche Dokumentation	3	1	3	2	6
Zielführende / intuitive Bedienung	2	1	2	2	4
Geringe Gefahr durch Herstellerabhängigkeit	1	3	3	1	1
Gute Herstellerbewertung/große Nutzerbasis	1	1	1	1	1
Übereinstimmung mit geforderten Funktionalitäten	3	1	3	1	3
Leicht anpassbar und änderbar	2	1	2	1	2
Eignung für Umgebung/Werkzeugkette	3	1	3	2	6
Einfache manuelle Bestätigung für entsprechende Arbeitsschritte	1	1	1	1	1
Einfacher Zugang zu entstandenen Artefakten	3	1	3	1	3
Einfache Steuerung der Pipeline	2	1	2	1	2
Nützliche Erweiterungsmöglichkeiten durch andere vorhandene Funktionalitäten	1	1	1	3	3
			1,07		**1,62**

Tabelle 6.4.1: Gegenüberstellung GitLab CI/CD und Jenkins

6.5 Werkzeuge zur Unterstützung der Bereitstellung

Zur Umsetzung der automatisierten Bereitstellung wurden Werkzeuge herangezogen, die den generellen Kriterien entsprechen und die weiteren Definierten erfüllen.
Für das Archivieren und Exportieren der iOS-Anwendung wurde das Xcodebuild-Werkzeug verwendet, welches direkt von Apple stammt, durch Xcode bereits installiert ist und einfach zu benutzen ist, wie schon in Abschnitt 6.2 erläutert.
Zum Erstellen der Versionshinweise wurde dem Projekt eine `release-notes.adoc`-Datei hinzugefügt, die mit Hilfe von AsciiDocotor in eine PDF-Datei konvertiert werden kann. Sowohl das AsciiDoc-Format zum Verfassen der Versionshinweise als auch AsciiDoctor als Werkzeug für die Konvertierung sind umfassend dokumentiert, integrieren sich problemlos in den Entwicklungs- und Automatisierungsprozess und erfüllen somit vollumfänglich alle Kriterien.
Zum Paketieren, Komprimieren und Verschlüsseln des Release-Paketes, der anschließenden Interaktion mit Nextcloud und zur Generierung von Bereitstellungsinformationen wurde ein Python-Skript erstellt. Dieses verwendet das Open-Source-Werkzeug 7-Zip, welches das Paketieren, Komprimieren zu einem ZIP-Archiv und Verschlüsseln des Release-Paketes mit Passwort unter Verwendung von AES-256 ermöglicht. Auch dieses ist umfassend dokumentiert, weit verbreitet und empfohlen,

[107] Vgl. Releases: https://docs.gitlab.com/ee/user/project/releases/ (Zugegriffen am 13. Juli 2022)

wodurch es insgesamt alle nötigen Kriterien erfüllt.
Mit Hilfe von weiteren Python-Bibliotheken wird mit der Nextcloud-API kommuniziert, um das Release-Paket hochzuladen und eine entsprechende Freigabe zu erstellen. Schließlich wird eine PDF-Datei mit den jeweiligen Bereitstellungsinformationen generiert. Mittels Python konnte dieser Prozess auf einfachem und effizienten Weg automatisiert umgesetzt werden. Python selbst ist dabei einfach zu lesen und zu verstehen. Des Weiteren verfügt es über eine sehr große Community, in der sich ggf. immer Hilfe geholt werden kann.

Kapitel 7

Ergebnisse

Anhand der exemplarischen Umsetzung und der anschließenden Evaluierung konnte gezeigt werden, dass die geforderte Arbeitsschritte effizient und effektiv automatisiert werden können. Dabei gibt es nicht genau ein Werkzeug, das alle Kriterien erfüllt. Es ist die Kombination mehrerer Werkzeuge zu einem gesamten System, das eine automatisierte Qualitätssicherung und Bereitstellung möglich machen.

Vorhandene Unit-Tests können effektiv und effizient mit Xcodebuild auf verschiedensten Simulatoren und physischen Geräten ausgeführt werden. Das Starten der Host Application bei jedem Unit-Test ist allerdings wenig effizient, da diese für kleine Komponenten nicht gebraucht wird. Um die Ausführungszeit zu verkürzen bietet es sich an, entsprechende Unit-Tests von der Host Application zu trennen.[108]
Weiterhin sollten auch UI- und E2E-Tests erstellt und im Rahmen der Automatisierung durchgeführt werden. Diese sind sehr nützlich, da sie im Endeffekt genau das Testen, was der Endanwender schließlich bekommt und nutzt. Dazu empfiehlt sich in erster Linie XCTest, da dieses bereits für Unit-Tests verwendet wird und alle weiteren Kriterien entsprechend erfüllt. Ein mögliches Test-Backend kann dafür einfach via Docker und entsprechenden Containern lokal auf dem Testsystem bereitgestellt werden. Anzumerken ist, dass Appium oder diverse Testplattformen unter bestimmten Bedingungen jedoch zukunftstauglicher sind, bspw., wenn sich Android- und iOS-Client von der UI her sehr ähneln oder nahezu identisch sind. Tests könnten dann in abstrakter Form einmal plattformübergreifend und unabhängig von der plattformspezifischen Programmiersprache erstellt werden. Dadurch können auch Personen eingesetzt werden, die nicht explizit mit den plattformspezifischen Testframeworks und Eigenarten vertraut sind. Somit minimiert sich der extra Aufwand, der sich speziell für Android und iOS ergibt. Der Aufwand für mögliche Spezialfälle, die sich nur mit plattformspezifischen Testframeworks abbilden lassen, entfällt dabei allerdings nicht. Dennoch sollte diese Variante mit in Betracht gezogen werden, wenn es plattformübergreifende Clients mit gleicher Funktionalität und UI gibt.

In Bezug auf statische Tests hat die Evaluierung gezeigt, dass die Kombination von SonarQube mit dem OWASP Dependency-Check-Plugin, SwiftLint und auch Mobsfscan, trotz des Fehlers, die geforderten Kriterien bestmöglich abdecken, wie in Tabelle 6.3.1 zu sehen. Aufgrund der Wiederverwendbarkeit empfiehlt sich zur Secret-Detection weiterhin auch Gitleaks. Code Climate CLI empfiehlt sich lediglich als mögliche Alternative, wenn SonarQube nicht zur Verfügung steht.
Mit Hilfe der genannten Werkzeuge lassen sich nun effizient duplizierte Codeteile, Klassen mit mehr als 400 LoC, Verstöße gegen allgemeine Programmierrichtlinien, bekannte Schwachstellen und mögliche Secrets finden. Kein Werkzeug erkennt jedoch ungenutzte Deklarationen. Für Singletons lassen sich in erster Linie nur Indizien finden und zur Erkennung von fehlenden Completion-Aufrufen muss sich nochmal tiefgreifender mit regulären Ausdrücken und dessen Möglichkeiten auseinander gesetzt werden, um eine endgültige Aussage darüber treffen zu können. Auch veraltete

[108] Vgl. „Stop holding your XCTests hostage“: https://medium.com/@alexander.apriamashvili/stop-holding-your-xctests-hostage-751d5b6ceb66 (Zugegriffen am 13. Juli 2022)

eingebundene externe Komponenten lassen sich nicht über die Pipeline feststellen. Abhilfe schafft hier allerdings swift-outdated als Xcode-Integration.
Durch die Werkzeuge, insbesondere SonarQube, lassen sich weiterhin die zyklomatische Komplexität, LoC, Anzahl an Verstößen gegen Programmierrichtlinien sowie die Anzahl an gefundenen Schwachstellen bestimmen. Zusätzlich kann in SonarQube auch eine Testabdeckung integriert werden. Alle Werte können für ein Quality Gate einfließen, dessen Ergebnis aber lediglich als Unterstützung für Code-Reviews genutzt werden sollte, wie in Unterabschnitt 3.3.2 und Unterabschnitt 2.3.3 ausgeführt.

Für die automatisierte Bereitstellung der iOS-Anwendung empfiehlt sich die Kombination von Xcodebuild, AsciiDoctor und einem Python-Skript. Mit Xcodebuild lassen sich problemlos Release-Builds mit einer spezifischen Signatur erstellen. Zu Beachten ist dabei jedoch, dass sich alle nötigen Provisioning Profiles, Zertifikate und zugehörige private Schlüssel auf dem macOS-System befinden, das für die Automatisierung genutzt wird. Diese müssen vor allem gültig sein. Aufgrund von Sicherheitsvorkehrungen seitens Apple verlieren die Profile und Zertifikate meist nach einem Jahr ihre Gültigkeit. Wie diese aktualisiert werden können, wurde bereits in der Konzeption, siehe Kapitel 4, erläutert.
Mittels AsciiDotor lässt sich aus der vorhandenen AsciiDoc-Datei für Versionshinweise eine PDF-Datei erzeugen. Das Python-Skript ermöglicht schließlich das Paketieren, Komprimieren und Verschlüsseln eines Release-Paketes mittels 7-Zip. Ebenso regelt es die Interaktion mit Nextcloud zum Hochladen und Freigeben eines Release-Paketes. Abschließend generiert es entsprechende Bereitstellungsinformationen.

Bezüglich der allgemeinen Automatisierung ist festzustellen, dass GitLab CI/CD am geeignetsten ist, wie Tabelle 6.4.1 zeigt. Es erfüllt alle Kriterien und deckt somit alle Anforderungen ab. Ausschlaggebend ist an dieser Stelle vor allem der Fakt, dass GitLab als Entwicklungsplattform bereits vorhanden ist und umfassend genutzt wird. Weiterhin bringt es eine Vielzahl weiterer Funktionalitäten mit sich, die direkt von GitLab CI/CD genutzt werden können.

Insgesamt ist festzustellen, dass mittels der konkreten Umsetzungen der ausgewählten Werkzeuge alle manuellen Arbeitsschritte automatisch ausgeführt werden können. Besonders die Bereitstellung lässt sich dadurch effizient abwickeln und ist weniger fehleranfällig. Für die automatisierte Qualitätssicherung wurde u.a. aufgezeigt, wie UI- und E2E-Tests erstellt und eingebunden werden können. Hinzu kamen weiterhin eine Schwachstellenanalyse und eine Secret-Detection sowie die Erfassung von Metriken und die Einbindung eines Quality Gates. Zusätzlich kann die Einhaltung allgemeiner Programmierrichtlinien überprüft werden.
Dynamische und statische Tests werden dabei für jeden einzelnen Push in das zentrale Repository ausgeführt, wodurch Entwickler schnell Feedback erhalten und implizit das eigene Vorgehen und Qualitätsbewusstsein verbessert wird. Für Reviewer stehen außerdem detaillierte Berichte von SonarQube usw. bereit. Darüber hinaus müssen sie sich keine Gedanken über nicht eingehaltene Programmierrichtlinien machen und können sich mehr auf design-technische, semantische und logische Angelegenheiten fokussieren.
Den Ausführungen entsprechend wurden alle Erfolgskriterien erfüllt und die Ziele aus Abschnitt 3.2 erreicht. Mit Ausnahme der Erkennung von Singletons und nicht genutzten Deklarationen konnten alle Anforderungen beachtet und umgesetzt werden.

Die Umsetzung hat gezeigt, dass eine Automatisierung der geforderten Prozesse möglich ist. Darüber hinaus konnte eine Empfehlung für weitere automatisierte qualitätssichernde Maßnahmen ausgesprochen und deren Umsetzung gezeigt werden. Für die konkrete iOS-Anwendung sind die in Abschnitt 3.3 aufgeführten Maßnahmen zu nennen. Darunter Unit- und UI-Tests sowie umfangreiche Schwachstellenanalysen. Wie die Automatisierung generell in den Entwicklungsprozess eingebunden werden kann, wurde in Abschnitt 3.4 dargestellt. Hier empfiehlt sich sowohl eine voll- als auch teilautomatische Abarbeitung der Prozesse. CI kommt aufgrund der Entwicklungsmodalitäten vorerst nicht in Betracht.

Vielversprechende Werkzeuge zur Umsetzung wurden in Abschnitt 3.7 identifiziert. Am geeignetsten ist die Automatisierung mit GitLab CI/CD, die oben genannte Werkzeuge einbindet und steuert. Zur Auswahl und Evaluierung der Werkzeuge wurden die in Abschnitt 3.6 definierten Kriterien verwendet. Neben wichtigen Funktionalitäten wurde u.a. darauf geachtet, dass keine neuen Lizenzkosten anfallen und die Werkzeuge lokal laufen, ohne eine Verbindung zum Hersteller aufzubauen. Des Weiterhin wurden allgemein die Effizienz, Benutzbarkeit, Zuverlässigkeit, Effektivität, Wartbarkeit und Kompatibilität bewertet.

Generell sollte eine Automatisierung vor allem bei repetitiven und werkzeug-gestützten Tätigkeiten eingesetzt werden, um menschliche Fehler zu vermeiden. Im Rahmen der Arbeit wurde sich dahingehend besonders auf den Qualitätssicherungs- und Bereitstellungsprozess konzentriert, die sich sehr dafür eignen. Insbesondere bei der Durchführung von Tests lässt sich mit einer Automatisierung viel Optimierungspotential im Hinblick auf Effizienz finden. Grundlegend muss sich aber auch das generelle Projekt und die Software an sich für die Automatisierung eignen, wie in Unterabschnitt 2.3.3 dargestellt.
Was bleibt, ist die Frage, unter welchen Voraussetzungen eine automatisierte Sicherstellung der Softwarequalität bzw. die Automatisierung generell als zuverlässig betrachtet werden kann?
Zuverlässigkeit ist dabei ein weitläufiger Begriff, der viele verschiedene Kriterien umfassen kann. Für die Beantwortung der Frage soll darunter die Fähigkeit verstanden werden, mit der Tests korrekt durchgeführt, Fehler gemeldet und konkrete Verstöße gegen Programmierrichtlinien, Schwachstellen, usw. richtig erkannt werden. Im Hinblick auf Unit-, UI- oder E2E-Tests, die einmal erstellt und nur noch durchgeführt werden müssen, ist dabei kein großes Risiko zu erwarten. Die Automatisierung führt die Tests immer gleich durch und meldet Fehler höchst zuverlässig. Dazu sollten in erster Linie überhaupt Tests vorhanden sein und eine eigene Testumgebung genutzt werden. Diese sollte für jeden Testdurchlauf gleich sein und definierte Testdaten enthalten. Zur Beurteilung der Qualität ist neben der korrekten Testausführung auch immer die Testabdeckung wichtig. Vorhandene Tests sollten einen möglichst großen Teil der gesamten Codebasis bzw. der Benutzeroberfläche abdecken, um eine optimale Aussage treffen zu können. Des Weiteren sollten die Tests bei Änderungen immer mit angepasst werden.
Werkzeuge für statische Tests basieren auf mehr oder weniger komplexen Regeln, die auf den gesamten Quellcode angewendet werden. Dabei können formale Fehler oder unsauberer Code, bspw. Codeduplikate, leicht erkannt und gemeldet werden. Mögliche Schwachstellen, die bewusst oder unbewusst durch Entwickler eingebaut werden, die aktiv von Angreifern gesucht und ausgenutzt werden können, lassen sich jedoch nur in den wenigsten Fällen durch solche Werkzeuge aufspüren, wie in Unterabschnitt 2.3.3 dargestellt. Sie bieten demzufolge keine Garantie dafür, dass der Quellcode frei von Sicherheitsmängeln ist. Auch die automatisierte Erfassung und Auswertung sämtlicher Metriken liefert keinen konkreten Beweis für die Qualität einer Software. Sie liefern lediglich erste Hinweise. Statische Tests werden dementsprechend nie hundertprozentig zuverlässig sein. Damit kann eine automatisierte Qualitätssicherung auch nie als komplett zuverlässig angesehen werden. Sie erleichtert und unterstützt Entwickler aber in vielen Belangen.

Kapitel 8

Zusammenfassung und Ausblick

8.1 Zusammenfassung

Zur Verbesserung der Effizienz des Entwicklungsprozesses sollte die Qualitätssicherung und Bereitstellung einer nativen iOS-Anwendung automatisiert werden. Als Grundlage zum Verständnis wurde dafür zuerst auf iOS-Anwendungen und deren Entwicklung sowie auf Softwarequalität und Metriken zu deren Messung eingegangen. Aus ITIL, dem BSI-Grundschutz-Kompendium und dem Bitkom-Leitfaden „Zur Sicherheit softwarebasierter Produkte“ wurden allgemeine qualitätssichernde Maßnahmen zusammengetragen und deren Eignung für eine Automatisierung erörtert. Weiterhin wurde sich grundlegend mit Automatisierung in der Softwareentwicklung, deren Möglichkeiten und Grenzen, auseinandergesetzt und bekannte Werkzeuge zur Unterstützung vorgestellt. Anhand eines Experteninterviews wurden Anforderungen bestimmt, die als Grundlage für die Definition von Kriterien dienten. Diese wurden zur Auswahl von Werkzeugen und zur Evaluierung einer exemplarischen Umsetzung herangezogen, wodurch anschließend eine fundierte Empfehlung ausgesprochen werden konnte.
Allgemein sollten dynamische und statische Tests automatisiert durchgeführt werden. Dynamische Tests umfassen dabei u.a. Unit-, UI-, E2E- und Regressions-Tests. Statische Tests umfassen die Prüfung der Einhaltung von Programmierrichtlinien, Schwachstellenanalysen auf Basis von Schwachstellendatenbanken, Secret-Detection, Dependency-Scanning und weiterer Codeanalysen, bspw. zur Beurteilung der Wartbarkeit oder sonstigen Qualitätsmerkmalen. Darüber hinaus sollten automatisiert Metriken erfasst und diese für ein Quality Gate genutzt werden, das zur Unterstützung von Code-Reviews dient. An dieser Stelle ist zu betonen, dass der Einsatz von Werkzeugen zu statischen Tests oder die Erfassung von Metriken keinen Beweis dafür liefern, dass die Software frei von Sicherheitsmängel ist. Es handelt sich lediglich um erster Indizien.
In Verbindung mit GitLab stellt GitLab CI/CD die geeignetste Lösung für die Automatisierung dar. Für dynamische Tests eignet sich in Bezug die konkrete iOS-Anwendung das Testframework XCTest zum Erstellen und das Werkzeug Xcodebuild zum Ausführen. Für statische Tests stellt die Kombination aus SonarQube mit dem OWASP Dependency-Check-Plugin, SwiftLint, Mobsfscan und Gitleaks die geeignetste Kombination dar. Für die Bereitstellung empfiehlt sich Xcodebuild, AsciiDoctor, 7-Zip und ein Python-Skript. Das Skript übernimmt die Erstellung eines Release-Paketes und kommuniziert mit der unternehmenseigenen Nextcloud-Instanz. Die Auswahl der Werkzeuge basierte dabei auf der Erfüllung und Bewertung bestimmter Kriterien, wie bspw. dem Vorhandensein benötigter Funktionalitäten, Effizienz, Zuverlässigkeit oder Bedienbarkeit.
Die Automatisierung ist vor allem dann geeignet, wenn es sich um repetitive und werkzeug-gestützte Tätigkeiten handelt, wie sie beim Testen und Bereitstellen von Software zu finden sind. Das Projekt muss dazu jedoch eine gewisse Laufzeit aufweisen, damit sich der anfänglich hohe Aufwand für die Umsetzung der Automatisierung amortisiert. Weiterhin sollte die Software an sich, insbesondere für UI-Tests, einem gewissen Reifegrad entsprechen und über eine hohe Testabdeckung verfügen.

8.2 Ausblick

Im Rahmen der Arbeit wurde grundlegend gezeigt, welche Maßnahmen für eine automatisierte Qualitätssicherung infrage kommen könnten. Dabei wurde auch auf das sogenannte Fuzzing eingegangen, bei dem automatisiert eine Vielzahl an Testeingaben für die zu testende Anwendung generiert werden, die Fehler aufdecken sollen. Insbesondere für mobile Anwendungen lässt sich das allerdings schwer umsetzen, weil schlecht mit den jeweiligen Anwendungen interagiert werden kann, siehe Unterabschnitt 3.3.1. Für eine endgültige Aussage zu dieser Thematik bedarf es vermutlich einer tiefgreifenden Beschäftigung, welche sich im Anschluss an die Arbeit anbietet.
Zur Erhöhung der Sicherheit und Flexibilität bei der Automatisierung bietet sich als Optimierung die Nutzung von Docker-Containern an, die alle Abhängigkeiten, Werkzeuge usw. enthalten.[109] Neben sicherheitsrelevanten Aspekten liefert diese Vorgehensweise die Vorteile, dass verwendete Softwarepakete und ihre Versionen sowie jegliche Konfiguration in der Pipeline hinterlegt werden können und keine bzw. nur wenige manuelle Installationen und Konfigurationen auf dem macOS-System notwendig sind. Es muss lediglich eine Docker-Instanz vorhanden sein, auf die der GitLab Runner zugreifen kann. Das Bauen, Testen, Archivieren und Exportieren einer iOS-Anwendung kann allerdings nicht in Container ausgelagert werden und erfordert immer ein macOS-System mit Xcode.
Für eine vollständige Automatisierung der Bereitstellung, ohne das abgelaufene Profile oder Zertifikate manuell erneuert werden müssen, bietet es sich weiterhin noch an, die Vorgehensweisen des Codesigning Guides[18] zu übernehmen. Hier muss allerdings erst geprüft werden, inwieweit dessen Umsetzung im konkreten Fall möglich und sinnvoll ist.
Zum Abschluss sei noch einmal wiederholt, dass eine automatisierte Qualitätssicherung und der Einsatz von entsprechenden Testwerkzeugen keine Garantie dafür liefern, dass eine Software frei von Fehlern oder Sicherheitsmängeln ist. Weitere Arbeiten könnten an dieser Stelle anknüpfen und prüfen, ob und inwieweit Künstliche Intelligenz zur automatisierten Qualitätssicherung eingesetzt werden kann, um die Zuverlässigkeit in dieser Hinsicht zu verbessern.

[109] Vgl. Run your CI/CD jobs in Docker containers: https://docs.gitlab.com/ee/ci/docker/using_docker_images.html#requirements-and-limitations (Zugegriffen am 13. Juli 2022)

Quellenverzeichnis

[BK21] Broy, Manfred ; Kuhrmann, Marco: *Einführung in die Softwaretechnik*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2021. http://dx.doi.org/10.1007/978-3-662-50263-1. http://dx.doi.org/10.1007/978-3-662-50263-1. – ISBN 978–3–662–50262–4
(zitiert auf Seiten 8 und 9)

[BSI22] *IT-Grundschutz-Kompendium*. Edition 2022. Köln : Reguvis, 01.02.2022 https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Grundschutz/Kompendium/IT_Grundschutz_Kompendium_Edition2022.html. – ISBN 978–3–8462–0906–6
(zitiert auf Seiten 10 und 11)

[Dre22] Drehling, Wilhelm: Angriff nach Lehrbuch: ContiLeaks: Zip-Dateien per Known-Plaintext-Attacke knacken. In: *c't* (2022), No. 9, 166–168. https://www.heise.de/select/ct/2022/9/2206312041007921251
(zitiert auf Seite 34)

[Ebe21] Ebel, Nadin: *Basiswissen ITIL 4: Grundlagen und Know-how für das IT Service Management und die ITIL-4-Foundation-Prüfung*. Heidelberg : dpunkt.verlag, 2021 https://learning.oreilly.com/library/view/-/9781098128661/?ar. – ISBN 978–3–86490–710–4
(zitiert auf Seiten 2, 10, 11 und 12)

[GGZ] Gegendorfer, Christa ; Grötz, Rudolf ; Zax, Matthias: Es teste, wer sich ewig bindet: Testautomatisierungswerkzeuge für mobile Applikationen im Überblick. In: *iX* 2020, No. 9, 102–111. https://www.heise.de/select/ix/2020/9/2020608380237828404
(zitiert auf Seiten 15 und 20)

[GKFLM] Grunwald, Renke ; Kölpin-Freese, Lars ; Limburg, Arne ; Müller, Hendrik: Komplettpaket für Code: Systeme für Continuous Integration und Continuous Delivery im Vergleich. 2021, No. 12, 74–83. https://www.heise.de/select/ix/2021/12/2107510285116693424
(zitiert auf Seiten 16 und 19)

[GM16] Garousi, Vahid ; Mäntylä, Mika V.: *When and what to automate in software testing? A multi-vocal literature review*. 2016. http://dx.doi.org/10.1016/j.infsof.2016.04.015. http://dx.doi.org/10.1016/j.infsof.2016.04.015
(zitiert auf Seiten 2, 12, 13, 14 und 15)

[HF11] Humble, Jez ; Farley, David G.: *Continuous delivery: Reliable software releases through build, test, and deployment automation*. Upper Saddle River, NJ and Toronto and London : Addison-Wesley, 2011 (A Martin Fowler signature book). – ISBN 978–0–321–60191–9
(zitiert auf Seite 27)

[HKL+20] Hessel, Stefan ; Kriesel, Thomas ; Loring, Jacob ; Luckhaus, Stefan ; Mörl, Ramon ; Radomski, Sabine ; Spengler, Roland ; Theilen, Ines ; Termer, Frank: *Zur Sicherheit softwarebasierter Produkte: Status Quo, Ausblick und FAQ zu Entwicklung und Betrieb softwarebasierter Produkte*. Berlin, 2020 https://www.bitkom.org/Bitkom/Publikationen/Bitkom-Leitfaden-zur-Sicherheit-softwarebasierter-Produkte
(zitiert auf Seiten 10 und 11)

[Hof13] Hoffmann, Dirk W.: *Software-Qualität*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2013. http://dx.doi.org/10.1007/978-3-642-35700-8. http://dx.doi.org/10.1007/978-3-642-35700-8. – ISBN 978–3–642–35699–5
(zitiert auf Seiten 8 und 9)

[ISO11] *ISO/IEC 25010:2011: Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models*. 2011
(zitiert auf Seiten 7, 8 und 10)

[ISO17] *ISO/IEC/IEEE 24765:2017: Systems and software engineering — Vocabulary*. 2017
(zitiert auf Seiten 7, 8, 9 und 12)

[ISO21] *ISO/IEC/IEEE 29119-2:2021: Software and systems engineering - Software testing - Part 2: Test processes*. 2021
(zitiert auf Seite 10)

[KB22] Krypczyk, Veikko ; Bochkor, Olena: *Handbuch für Softwareentwickler*. 2., aktualisierte und überarbeitete Auflage. Bonn : Rheinwerk Verlag, 2022 (Rheinwerk Computing). https://ebookcentral.proquest.com/lib/kxp/detail.action?docID=6829117. – ISBN 978–3–8362–7977–2
(zitiert auf Seite 9)

[Lig09] Liggesmeyer, Peter: *Software-Qualität: Testen, Analysieren und Verifizieren von Software*. 2. Aufl. 2009. Heidelberg : Spektrum Akademischer Verlag, 2009 http://nbn-resolving.org/urn:nbn:de:bsz:31-epflicht-1619634. – ISBN 978–3–8274–2056–5
(zitiert auf Seite 9)

[McC09] McConnell, Steve: *Code Complete*. 2nd ed. Sebastopol : O'Reilly Media Inc, 2009 http://gbv.eblib.com/patron/FullRecord.aspx?p=488632. – ISBN 0–7356–1967–0
(zitiert auf Seiten 11 und 14)

[RJMG17] Raulamo-Jurvanen, Päivi ; Mäntylä, Mika ; Garousi, Vahid: Choosing the Right Test Automation Tool. In: Mendes, Emilia (Hrsg.) ; Counsell, Steve (Hrsg.) ; Petersen, Kai (Hrsg.): *Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering*. New York, NY, USA : ACM, 06152017. – ISBN 9781450348041, pp. 21–30
(zitiert auf Seite 15)

[SOP14] *Requirements-Engineering und -Management: Aus der Praxis von klassisch bis agil*. 6., aktualisierte und erw. Aufl. München : Hanser, 2014. – ISBN 978–3–446–43893–4
(zitiert auf Seite 24)

[WC13] Wishnick, Aaron ; Chow, Ming: *Fuzzing an iOS Application*. 2013 (Computer Science, Introduction to Computer). https://www.cs.tufts.edu/comp/116/archive/fall2013/awishnick.pdf
(zitiert auf Seiten 11 und 25)

[Wie] Wiegenstein, Andreas: Kaffeesatzleserei: Ein kritischer Blick auf statische Codeanalyse-Verfahren. In: *iX special* , No. 16, pp. 96–101
(zitiert auf Seite 14)

[Wie21] Wiegenstein, Andreas: Quelltext im Fokus: Werkzeuge zur automatischen Codeanalyse. In: *iX special* 2021 (2021), No. 14, pp. 58–65
(zitiert auf Seiten 2, 16 und 22)

Hilfsmittel und Software

Zur Erstellung der vorliegenden Abschlussarbeit wurden die folgenden Hilfsmittel und Software verwendet:

- macOS Monterey 12.4
- Python 3.8.9
- Homebrew 3.5.4
- SwiftLint v0.47.1
- SonarQube 9.5 Developer Edition
- Mobsfscan v0.1.0
- Gitleaks v8.8.12
- Code Climate CLI 0.85.29
- 7-Zip 17.04
- Nextcloud 24.0.2
- GitLab 15.1.0 Community Edition und GitLab Runner 15.1.0
- Jenkins v2.332.3
- Xcode (inkl. Xcodebuild) 13.4
- Visual Studio Code 1.69.0
- Ranger Desktop 1.3.0
- Asciidoctor 2.0.17
- TeXstudio 4.2.3
- Citavi Web
- Duden Rechtschreibprüfung online
- diagrams.net
- Google Scholar
- Connected Papers
- Sächsische Landesbibliothek – Staats- und Universitätsbibliothek Dresden

Teil II

Anhang

Anhang A

Anforderungsanalyse

A.1 Interviewleitfaden

Einstieg

- Begrüßung und Dank für die Zeit
- kurze Themenbeschreibung: Automatisierung der Qualitätssicherung und Bereitstellung einer iOS-Anwendung (speziell: iOS-Client von MobiKat)
- kurze Beschreibung des Ablaufs, der Dauer und zur Erstellung des Protokolls

Einstiegsfragen
(dienen der Bestimmung von Zielen, Grenzen, Kontext und beteiligten Personen)

Frage: Für welchen Zweck soll die Automatisierung eingesetzt werden?

- Erwartung: Aufzählung von übergeordnete Leitzielen und dadurch erreichten Verbesserungen / Vorteilen
- Grund: zur Bestimmung des Erfolgs / der Zielerreichung

Frage: In welchem Kontext soll die Automatisierung eingesetzt werden?

- Erwartungen: Erläuterung des Kontextes in dem Automatisierung eingesetzt werden soll (Anwendung, Entwicklungsprozess, Vorgehen, Programmiersprache, Testerstellung)
- Grund: für kontextrelevante Entscheidungen

Frage: Welche Personen sind mit welchen Interessen beteiligt?

- Erwartung: Aufzählung der Stakeholder und deren groben Interessen
- Grund: für Einbezug verschiedener Sichtweisen

Frage: Was soll herausgearbeitet / getan werden und was nicht?

- Erwartung: kurze Beschreibung der zu erledigenden Aufgaben
- Grund: Abgrenzung der Aufgaben

Frage: Was sind Erfolgskriterien?

- Erwartung: Aufzählung von Kriterien, die nötig sind, um Ziele zu erreichen
- Grund: effiziente und effektive Zielerreichung

Schlüsselfragen
(dienen der Bestimmung von konkreten Anforderungen)

Frage: Welche generellen Prozesse sollen automatisiert werden?
- Erwartung: Erarbeitung eines ersten generellen Überblickes über zu automatisierende Prozesse
- Rückfragen: ggf. nähere Erläuterungen zu Prozessen

Frage: Welche Metriken müssen/sollen verwendet werden?
- Erwartung: Aufzählung einzelner Metriken, die verwendet werden müssen oder sollen
- Rückfragen:
  - Welchem Zweck dienen die genannten Metriken?
  - keine genannten Metriken: Auf welche Qualitätsmerkmale / -kategorien wird bei der iOS-Anwendung besonders Wert gelegt?

Frage: Welche Funktionalitäten und Anforderungen müssen vorhanden und erfüllt sein?
- Erwartung: Aufzählung einzelner Funktionalitäten und Anforderungen, die definitiv vorhanden und erfüllt sein müssen
- Rückfragen: ggf. Zerlegung komplexer Funktionalitäten, vertiefende Fragen

Frage: Welche Fehler müssen oder sollten durch statische Tests erkannt werden?
- Erwartung: Aufzählung konkreter Fehler (bspw. keine Passwörter im Quellcode oder Ähnliches)
- Rückfragen: ggf. Notwendigkeit klären oder Erwartung, was bei Entdeckung geschehen soll

Frage: Welche Funktionalitäten und Anforderungen sollten vorhanden und erfüllt sein?
- Erwartung: Aufzählung einzelner Funktionalitäten und Anforderungen, die vorhanden und erfüllt sein sollten aber nicht müssen
- Rückfragen: ggf. Zerlegung komplexer Funktionalitäten, vertiefende Fragen

Frage: Gibt es bereits existierende Lösungen, die abgelöst werden sollen?
- Erwartung: Aufzählung von bereits verwendeten Lösungen für eine (teil)automatisierte Qualitätssicherung und Bereitstellung einer iOS-Anwendung
- Rückfragen: Warum soll die jeweilige Lösung abgelöst werden?

Frage: Gibt es Systeme, mit denen interagiert werden muss?
- Erwartung: Aufzählung von einzelnen Systemen, die an einzelnen Arbeitsschritten beteiligt sind (bspw. Versionsverwaltungssystem, ...)
- Rückfragen: Wozu und wie werden diese genutzt?

Frage: Gibt es sonstige Nebenbedingungen oder Anmerkungen?
- Erwartung: Nennung von sonstigen Bedingungen oder Anmerkungen

Zusammenfassung und Verabschiedung
- Wiederholung und Zusammenfassung des Gesagten auf Grundlage des Protokolls, ggf. Berichtigung
- Information über Auswertung und Verwendung
- Dank für Zeit und Verabschiedung

A.2 Protokoll Experteninterview

Ort und Zeit

- Büroraum, Fraunhofer IVI
- 10.05.2022, 10:00 Uhr bis 11:30 Uhr

Teilnehmer

- Herr Sebastian Koch (Interviewter, Technischer Angestellter, leitender iOS-Entwickler)
- Herr Kai Ciesielski (Interviewer, BA-Student)

Fragen und Antworten

F1: *Für welchen Zweck soll die Automatisierung genutzt werden?*

- kontinuierliches Sichern der Softwarequalität
- Minimieren von menschlichen Fehlern bei repetitiven Aufgaben
- Messbare Qualitätssicherung
- Steigerung der Effizienz (Minimieren von Zeitaufwänden)
- Qualitativeres und strukturierteres Vorgehen aller Teammitglieder

F2: *In welchem Kontext soll die Automatisierung eingesetzt werden?*

- iOS-Anwendungsentwicklung (nativer iOS-Client als Teil einer Multiplattform-Anwendung)
- iOS-Anwendung ist vier Jahre alt
- Anwendungsentwicklung mittels der Programmiersprache Swift, Xcode und AppCode (IDE)
- Externe Bibliotheken werden via CocoaPods und Swift Package Manager eingebunden
- Vorgehen nach Scrum mit Feature-Branching und Code-Reviews
- Existierende Branches: Master, Development, Feature-Branches
- Feature-Branches werden in Development gemergt und bei Release wird von Development in Master gemergt
- Erstellung von Tests ist Aufgabe des jeweiligen Entwicklers
- Unit-Tests werden mittels XCTest erstellt

F3: *Welche Personen sind mit welchen Interessen beteiligt?*

- Entwickler: effizientere Arbeitsweise, hohe interne Softwarequalität, schnelles Feedback
- Projektverantwortliche: Sicherung der Qualität, effektive und effiziente Ressourcennutzung, geringeres Risiko bei Verteilung und Veröffentlichung der Anwendung, einfache Bereitstellung individueller Kundenwünsche ohne größeren Arbeitsaufwand
- Auftraggeber und Nutzer: schnelle Release-Zeiten, hohe Produkt- und Nutzungsqualität

F4: *Was soll herausgearbeitet / getan werden und was nicht?*

- Finden von Bausteinen und Werkzeugen für eine automatisierte Qualitätssicherung während der Implementierung und Integration der Software
- Umsetzung einer automatisierten Bereitstellung (Vollautomatisiertes Deployment kann und soll aufgrund spezieller Gegebenheiten nicht umgesetzt werden)

F5: *Was sind Erfolgskriterien?*

- Automatisierung definierter manueller Arbeitsschritte (siehe F7)
- Empfehlung für Etablierung weiterer automatischer qualitätssichernder Maßnahmen
- Exemplarische Umsetzung der Automatisierung als Machbarkeitsbeweis

F6: *In Bezug auf die messbare Qualitätssicherung: Welche Metriken müssen/sollen verwendet werden?*

- keine festgelegt; wichtige Metriken sollen vorgeschlagen werden
- Qualitätsmerkmale der iOS-Anwendung, auf die besonders Wert gelegt wird: Sicherheit, Funktionserfüllung, Wartbarkeit

F7: *Welche generellen Prozesse sollen automatisiert werden?*

- Auswahl bzw. Festlegung von Testsystemen und -umgebungen
- Durchführung von Tests (Unit-, Integration- und Regressions-Tests)
- Archivierung und Export der iOS-Anwendung (Erzeugung von IPAs)
- Erstellung von Versionshinweisen (Release-Notes)
- Erstellung von komprimierten und verschlüsselten Release-Paketen (enthält IPA, PDF-Datei mit Versionshinweisen und ggf. weitere Dateien)
- Upload von Release-Paketen in unternehmenseigene Nextcloud
- Generierung von Nextcloud-Freigaben

F8: *Welche Funktionalitäten und Anforderungen müssen vorhanden und erfüllt sein?*

- Automatisiertes Bauen von definierten Targets
- Automatisierte Durchführung von Unit-Tests
- Durchführung von Tests im Simulator
- Automatisierte Durchführung von statischen Tests
- Automatisierte Erzeugung von (Release-)Builds mit spezifischer Signatur
- Automatisierte Erzeugung einer PDF-Datei mit Versionshinweisen
- Automatisierte Erzeugung von komprimierten Release-Paketen (ZIP) pro definiertem Target
- Automatisierter Upload in die Nextcloud
- Zugriff auf Ergebnisse und Artefakte durch Teammitglieder via Webinterface
- Zugangskontrolle mittels Nutzername und Passwort
- Manuelles Starten des automatisierten Prozesses auf einfache Art und Weise
- Einfache Bedienbarkeit und Wartbarkeit
- Ausführliche Dokumentation von unterstützenden Werkzeugen

F9: *Welche Fehler müssen oder sollten durch statische Tests erkannt werden?*

- Duplizierter Code muss erkannt werden
- Ungenutzte Variablen und Methoden müssen erkannt werden
- Zu große Klassen (mehr als 400 Zeilen) müssen erkannt werden
- Fehlende Completion-Aufrufe sollten erkannt werden
- Singleton sollte erkannt werden
- Erkennung von Verstöße gegen allgemeine Programmierrichtlinien der Swift Community

F10: *Welche Funktionalitäten und Anforderungen sollten vorhanden und erfüllt sein?*

- Automatisierte Durchführung von UI- und E2E-Tests (d.h. inklusive Einbindung eines Test-Backends und Testdaten)
- Automatisierte Durchführung von Tests zusätzlich auf einem physischen Endgerät (iPad und iPhone)
- Automatisierte Verschlüsselung von Release-Paketen mittels Passwort und AES-256
- Automatisierte Erstellung einer Freigabe in Nextcloud pro definiertem Target
- Automatisierte Generierung einer PDF-Datei mit Freigabelink und -passwort sowie Passwort des verschlüsselten Release-Paketes

F11: *Gibt es bereits existierende Lösungen, die abgelöst werden sollen?*

- Xcode Server inklusive zugehöriger Bots
- Begründung: Xcode Server und Bots stellen einen zusätzlichen Aufwand speziell für dieses iOS-Projekt dar. Eine zentralere und einheitliche Lösung wäre wünschenswert.

F12: *Gibt es Systeme, mit denen interagiert werden muss?*

- GitLab als Entwicklungsplattform inkl. Versionsverwaltungssystem in der freien und selbst-gehosteten Version
- Selbt-gehostete Nextcloud für die Verteilung der Anwendung
- macOS mit Xcode für das Erstellen und Exportieren der iOS-Anwendung

F13: *Gibt es sonstige Nebenbedingungen oder Anmerkungen?*

- Cloud-Dienste dürfen nicht verwendet werden
- Neue Lizenzkosten dürfen nicht entstehen
- Selbt-gehostetes SonarQube steht als Werkzeug für eine statische Codeanalyse zur Verfügung und kann genutzt werden. Lizenz: Developer Edition

Mit meiner folgenden Unterschrift bestätige ich, dass ich das Protokoll gelesen habe und die relevanten Interview-Inhalte korrekt erfasst worden sind.

Sebastian Koch
Interviewter

Kai Ciesielski
Interviewer, Protokollant

Anhang B

Abbildungen

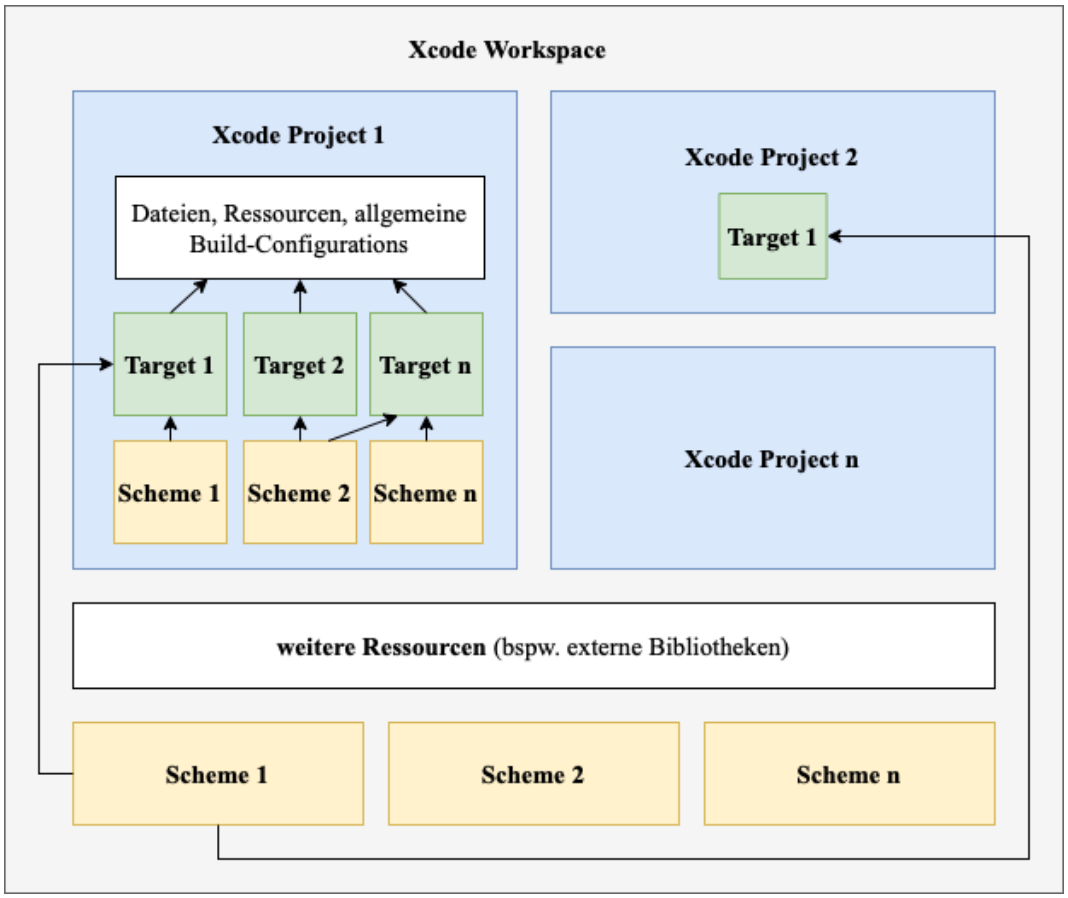

Abbildung B.0.1: Xcode Konzepte (eigene Darstellung)

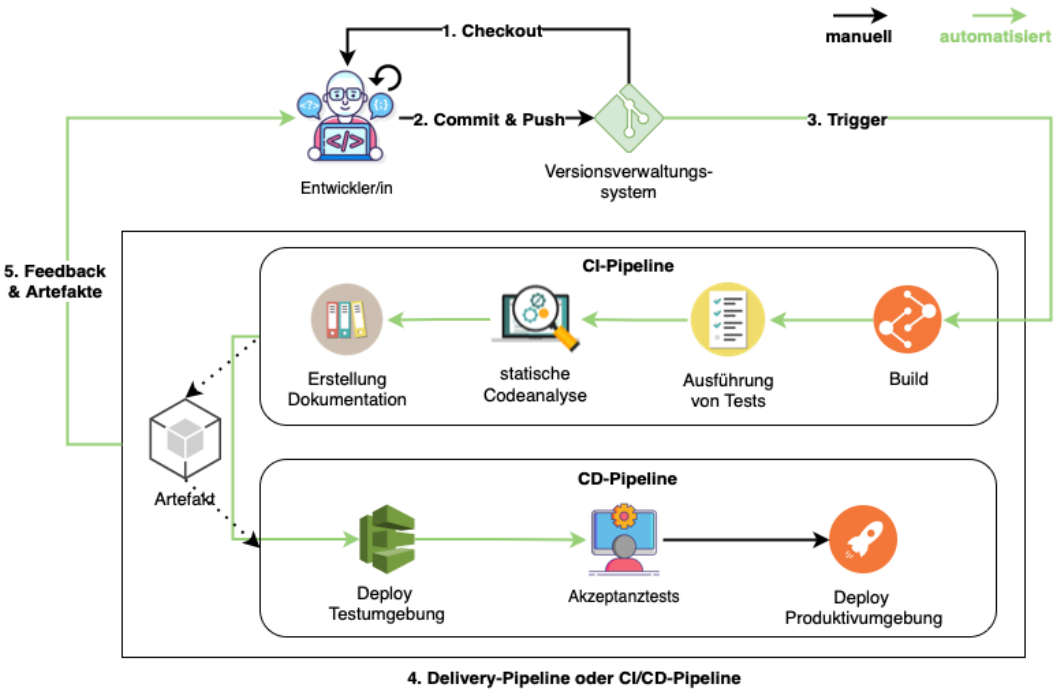

Abbildung B.0.2: Continuous Delivery (eigene Darstellung)

Sonar way BUILT-IN

Conditions

Conditions on New Code

Conditions on New Code apply to all branches and to Pull Requests.

Metric	Operator	Value
Coverage	is less than	80.0%
Duplicated Lines (%)	is greater than	3.0%
Maintainability Rating	is worse than	A
Reliability Rating	is worse than	A
Security Hotspots Reviewed	is less than	100%
Security Rating	is worse than	A

Abbildung B.0.3: Bedingungen des Quality-Gates „sonar way“ (Bildschirmaufnahme SonarQube)

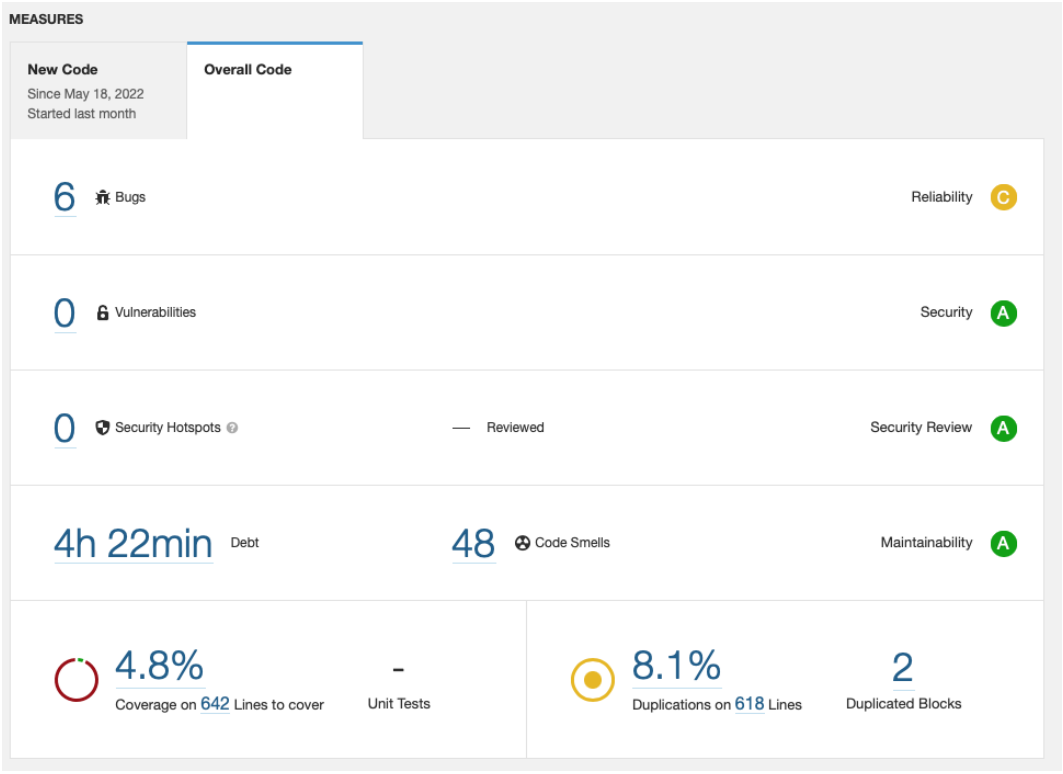

Abbildung B.0.4: SonarQube Analyseübersicht (Bildschirmaufnahme SonarQube)

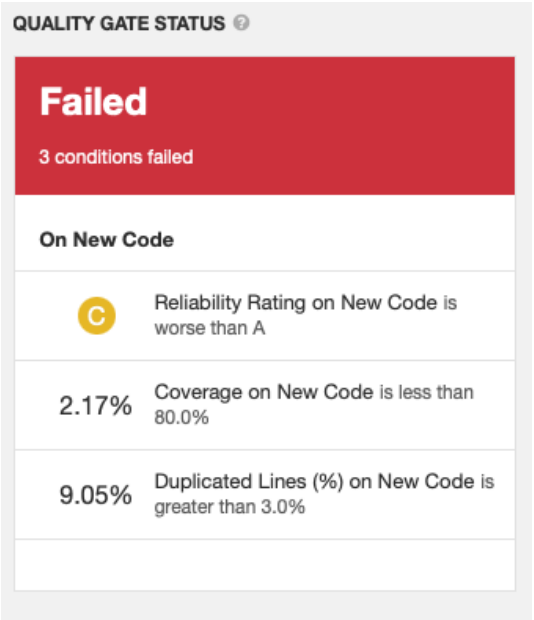

Abbildung B.0.5: Status des Quality Gates in SonarQube(Bildschirmaufnahme SonarQube)

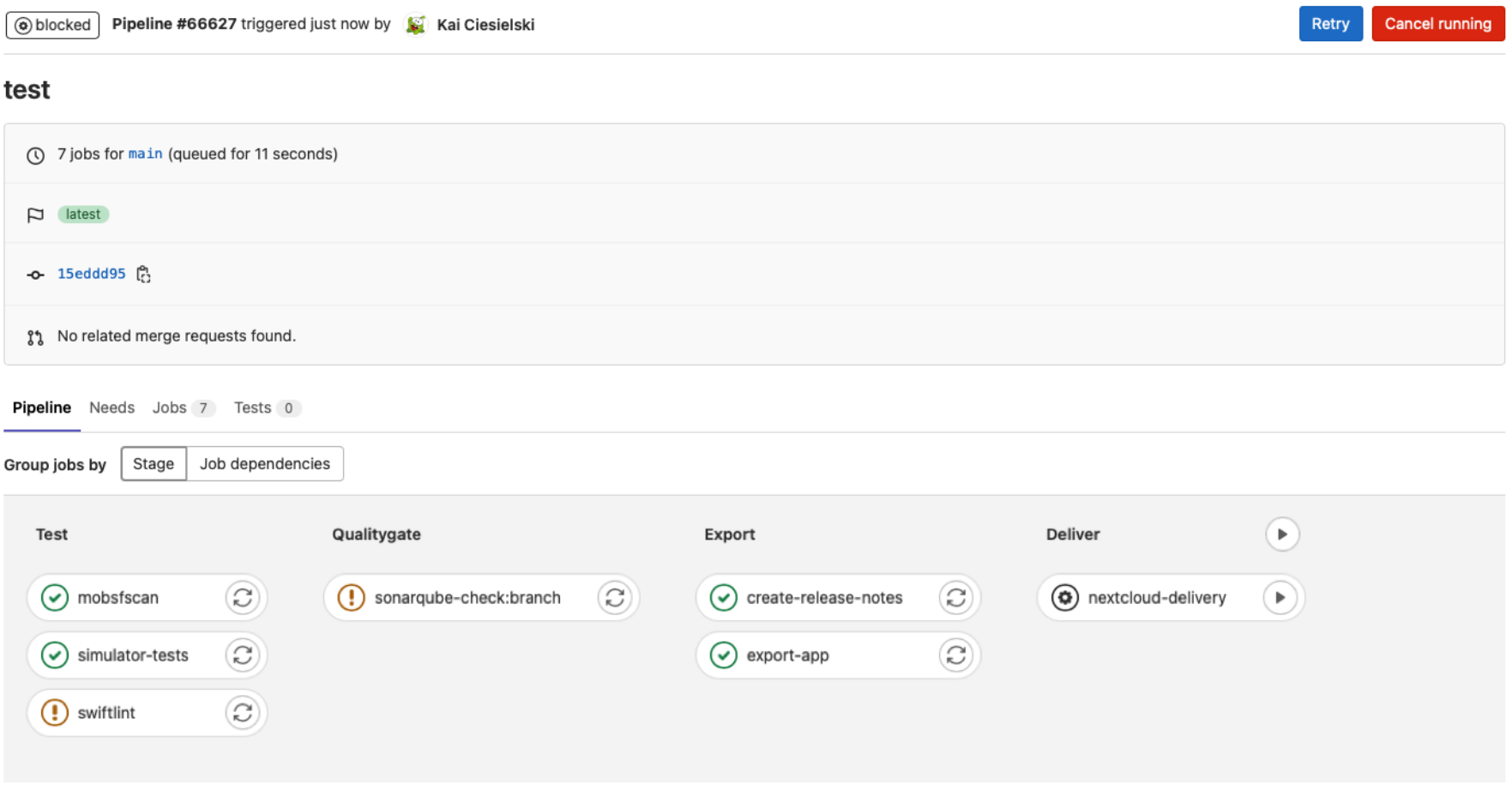

Abbildung B.0.6: Pipelineübersicht in GitLab (Bildschirmaufnahme GitLab)

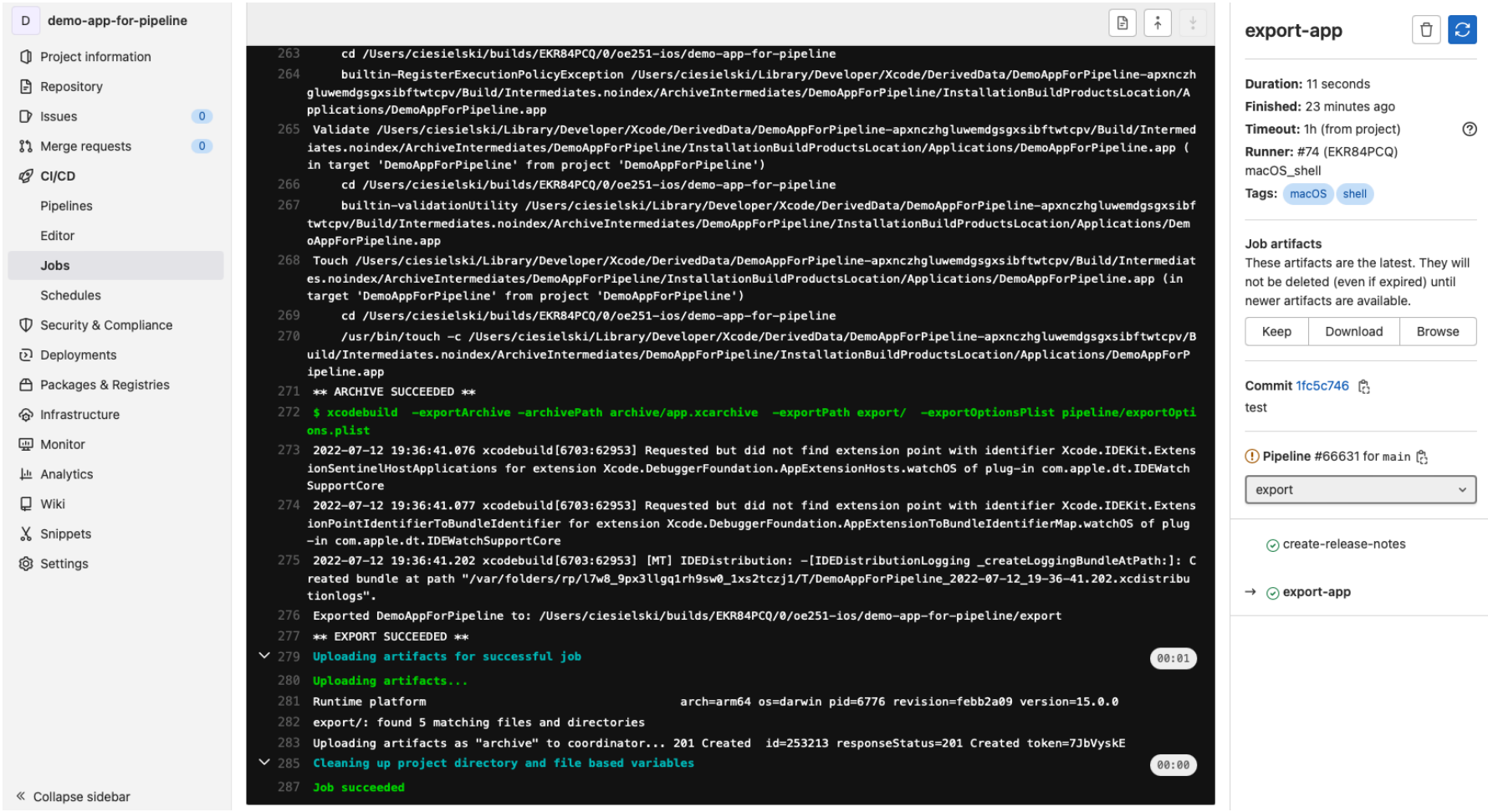

Abbildung B.0.7: Detailansicht eines Jobs in GitLab (Bildschirmaufnahme GitLab)

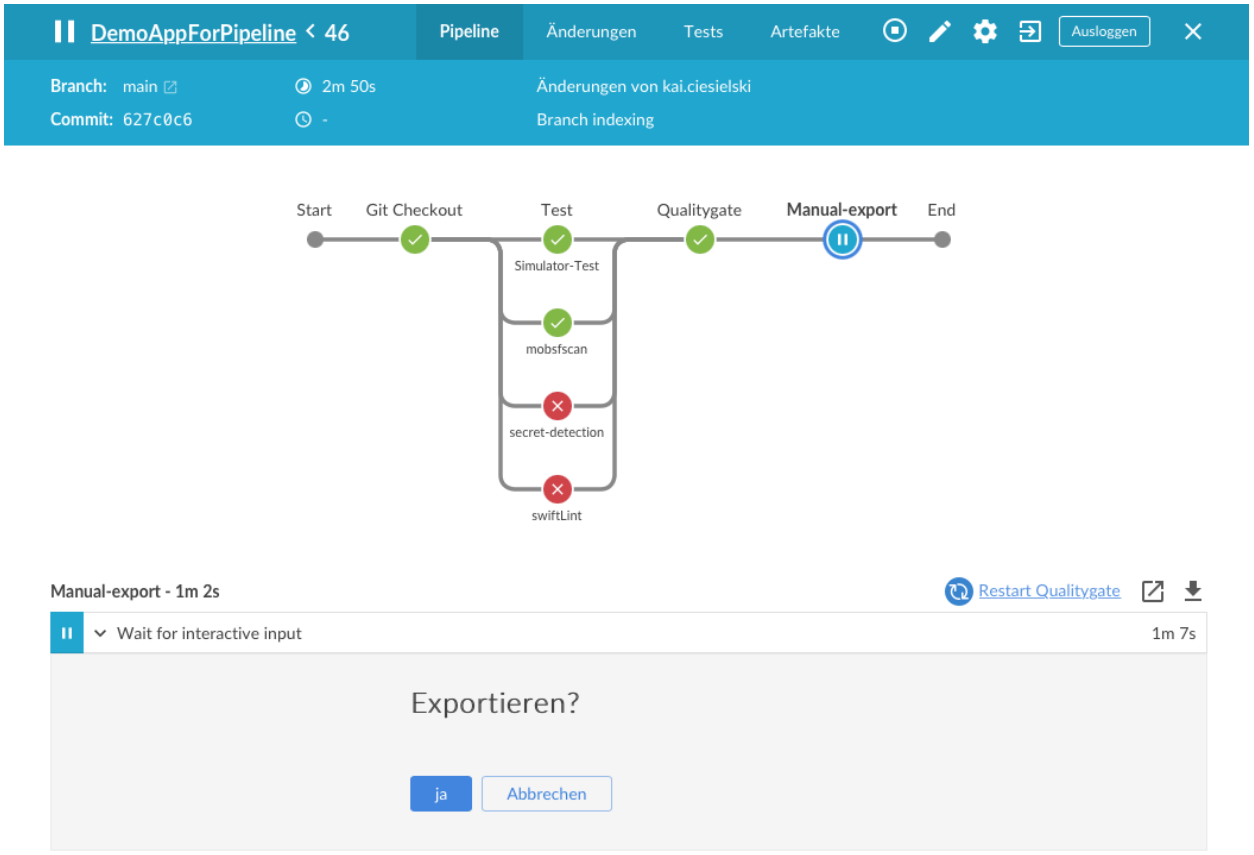

Abbildung B.0.8: Pipelineübersicht in Jenkins Blue Ocean (Bildschirmaufnahme Jenkins)

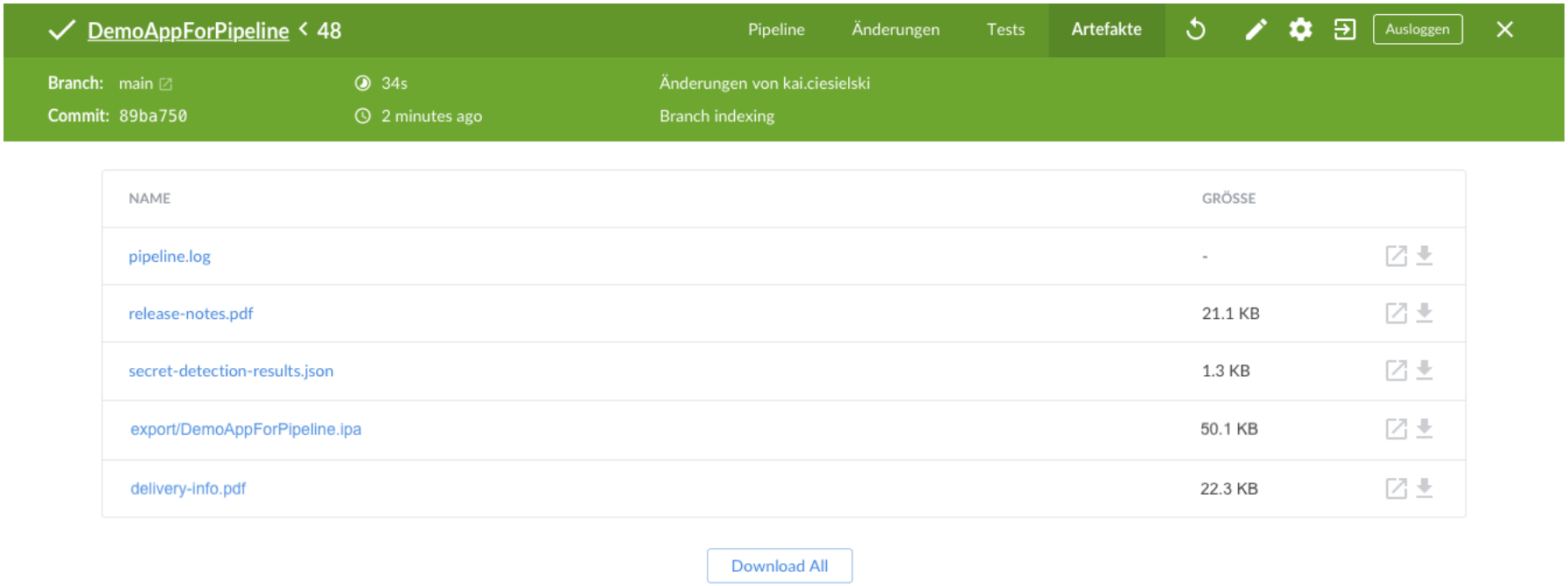

Abbildung B.0.9: Artefakteübersicht in Jenkins Blue Ocean (Bildschirmaufnahme Jenkins)

RULE ID	ios_keyboard_cache
CWE	CWE-919: Weaknesses in Mobile Applications
MASVS	MSTG-STORAGE-5
OWASP-MOBILE	M2: Insecure Data Storage
REFERENCE	https://github.com/MobSF/owasp-mstg/blob/master/Document/0x06d-Testing-Data-Storage.md#finding-sensitive-data-in-the-keyboard-cache-mstg-storage-5
DESCRIPTION	This app does not disable Keyboard cache. It must be disabled for all sensitive data inputs.
SEVERITY	INFO
RULE ID	ios_detect_reversing
OWASP-MOBILE	M9: Reverse Engineering
MASVS	MSTG-RESILIENCE-4
CWE	CWE-919: Weaknesses in Mobile Applications
REFERENCE	https://github.com/MobSF/owasp-mstg/blob/master/Document/0x06j-Testing-Resiliency-Against-Reverse-Engineering.md#ios-anti-reversing-defenses
DESCRIPTION	This app does not have Reverse engineering detection capabilities.
SEVERITY	INFO
RULE ID	ios_custom_keyboard_disabled
CWE	CWE-919: Weaknesses in Mobile Applications
MASVS	MSTG-PLATFORM-11
OWASP-MOBILE	M1: Improper Platform Usage
REFERENCE	https://github.com/MobSF/owasp-mstg/blob/master/Document/0x06h-Testing-Platform-Interaction.md#app-extensions
DESCRIPTION	This app does not have custom keyboards disabled.
SEVERITY	INFO

Abbildung B.0.10: Ausschnitt Mobsfscan-Report (Bildschirmaufnahme Mobsfscan-Report)

Violations

Serial No.	File	Location	Severity	Message
1	DemoAppForPipeline/Configuration.swift	34:16	Error	Force tries should be avoided.
2	DemoAppForPipeline/SceneDelegate.swift	19:19	Warning	Prefer `!= nil` over `let _ =`
3	DemoAppForPipeline/TodoCommunicationHandler.swift	14:5	Warning	Maybe a singleton?
4	DemoAppForPipeline/TodoCommunicationHandler.swift	59:20	Error	Force tries should be avoided.
5	DemoAppForPipeline/Testclass.swift	18:5	Warning	Function should have complexity 10 or less: currently complexity equals 11
6	DemoAppForPipeline/Testclass.swift	56:5	Warning	Function should have complexity 10 or less: currently complexity equals 11
7	DemoAppForPipeline/Testclass.swift	409:0	Warning	File should contain 400 lines or less: currently contains 409
8	DemoAppForPipeline/Testclass.swift	99:5	Error	Function body should span 40 lines or less excluding comments and whitespace: currently spans 307 lines
9	DemoAppForPipeline/Testclass.swift	18:76	Error	Variable name should be between 3 and 40 characters long: 'i'
10	DemoAppForPipeline/Testclass.swift	56:86	Error	Variable name should be between 3 and 40 characters long: 'i'
11	DemoAppForPipeline/Testclass.swift	10:1	Warning	Type body should span 300 lines or less excluding comments and whitespace: currently spans 359 lines

Summary

Total files with violations	4
Total warnings	6
Total errors	5

Abbildung B.0.11: SwiftLint-Report (Bildschirmaufnahme SwiftLint-Report)

Abbildung B.0.12: Ausschnitt gefundener Issues vom SonarQube (Bildschirmaufnahme SonarQube)

Similar blocks of code found in 2 locations. Consider refactoring.

```
50
51
52  func duplicatedComplexMethod(testA: Bool, testB: Bool, testC: Bool, testD: Bool, i:
53
54      if testA && testB {
55
56          if testC || testD {
57
58              if testA && testC {
59                  print("Test")
60              } else if !testC && testD {
61                  print("Test")
62              } else {
63                  print("Test")
64              }
```

▼ Other instances

Testclass.swift

```
12  let unusedString: String = "I'm useless."
13
14  func complexMethod(testA: Bool, testB: Bool, testC: Bool, testD: Bool, i: Int) {
15
16      if testA && testB {
17
18          if testC || testD {
19
20              if testA && testC {
21                  print("Test")
22              } else if !testC && testD {
23                  print("Test")
24              } else {
25                  print("Test")
26              }
```

▼ Details

Duplicated Code

Duplicated code can lead to software that is hard to understand and difficult to change. The Don't Repeat Yourself (DRY) principle states:

> Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.

When you violate DRY, bugs and maintenance problems are sure to follow. Duplicated code has a tendency to both continue to replicate and also to diverge (leaving bugs as two similar implementations differ in subtle ways).

Abbildung B.0.13: Ausschnitt gefundener Issues von Code Climate CLI (Bildschirmaufnahme Code-Climate-CLI-Report)

Anhang C

Tabellen

Kriterium	Erfüllung	Ergänzung/Bemerkung
Quality Gate vorhanden	×	Funktionalität nicht vorhanden.
stellt Testabdeckung dar	×	Funktionalität nicht vorhanden.
erkennt bekannte Schwachstellen	×	Funktionalität nicht vorhanden.
erkennt Secrets im Quellcode	✓	Definition eigener Regeln mittels regulären Ausdrücken möglich.
führt Dependency-Scans durch	×	Funktionalität nicht vorhanden.
erkennt Verstöße gegen allgemeine Programmierrichtlinien	✓	SwiftLint umfasst über 200 eigene Regeln, die zur Überprüfung von allgemeinen Programmierrichtlinien und -stilen verwendet (teilweise auch konfiguriert) werden können. (Beispiel: Abbildung B.0.11)
erkennt duplizierten Code	×	Funktionalität nicht vorhanden.
erkennt ungenutzte Deklarationen	×	Regel vorhanden aber hat Testfälle nicht erkannt.
erkennt Klassen mit mehr als 400 LoC	✓	LoC kann für Klasse überprüft werden, siehe Abbildung B.0.11. Regel kann entsprechend konfiguriert werden.
erkennt fehlende Completion-Aufrufe	k.A.	Regeldefinition möglich aber nicht umgesetzt und evaluiert.
erkennt Singletons	(✓)	Zeichenkette „private init“ kann mittels einer eigenen Regel gefunden werden, siehe Abbildung B.0.11. Liefert lediglich einen Indiz für die Verwendung eines Singletons.
misst Lines of Code	✓	LoC kann für eine Datei, Klasse, Methode, Closure bestimmt und überprüft werden. Jeweilige Regeln können konfiguriert werden.
misst zyklomatische Komplexität	✓	Regel kann konfiguriert werden. Warnung erscheint standardmäßig bei 10, Fehler bei 20. Switch-Case-Anweisungen können ignoriert werden. (siehe Abbildung B.0.11)
misst Anzahl an vorhandenen Schwachstellen	×	Funktionalität nicht vorhanden.
misst Anzahl an Verstößen gegen allgemeine Programmierrichtlinien	✓	Testreport (html) erzeugt eine entsprechende Zusammenfassung, siehe Abbildung B.0.11.
Möglichkeit zur Definition eigener Regeln	✓	Eigene Regeln können mittels regulärer Ausdrücke definiert werden.
umfangreiches Reporting	✓	Report (html) enthält Nummerierung, Ort, Schwere (Warnung, Error) und Meldung zu gefundenen Verstößen. Zusätzlich wird Anzahl der Verstöße angezeigt. Für den Report stehen folgende Formate zur Verfügung: xcode, json, csv, checkstyle, codeclimate, junit, html, emoji, sonarqube, markdown und github-actions-logging.

Tabelle C.0.1: Evaluierung SwiftLint

Kriterium	Erfüllung	Ergänzung/Bemerkung
Quality Gate vorhanden	×	Funktionalität nicht vorhanden.
stellt Testabdeckung dar	×	Funktionalität nicht vorhanden.
erkennt bekannte Schwachstellen	×	Funktionalität nicht vorhanden.
erkennt Secrets im Quellcode	✓*	Via SwiftLint-Plugin, ansonsten nicht.
führt Dependency-Scans durch	×	Funktionalität nicht vorhanden.
erkennt Verstöße gegen allgemeine Programmierrichtlinien	✓*	Führt standardmäßig nur Maintainability-Checks durch. Mit Hilfe des SwiftLint-Plugins können Programmierrichtlinien allerdings überprüft werden.
erkennt duplizierten Code	✓	Der duplizierte Code des Testfalls wurde erkannt, siehe Abbildung B.0.13. Zusätzlich wird eine Erläuterung gegeben und es werden Verbesserungshinweise aufgeführt.
erkennt ungenutzte Deklarationen	×	Keine Regel vorhanden. Auch keine Erkennung via SwiftLint.
erkennt Klassen mit mehr als 400 LoC	✓*	Erkennt standardmäßig nur LoC von Dateien oder Methoden. Für Klasse via SwiftLint möglich.
erkennt fehlende Completion-Aufrufe	k.A.	Regeldefinition via SwiftLint möglich aber nicht umgesetzt und evaluiert.
erkennt Singletons	(✓*)	Erkennung standardmäßig nicht möglich. Mit Hilfe von SwiftLint können allerdings erste Indizien geliefert werden.
misst Lines of Code	✓	LoC kann für eine Datei und Methode überprüft werden. Schwellwert für Warnung kann konfiguriert werden. Zusätzlich via SwiftLint möglich.
misst zyklomatische Komplexität	✓	Schwellwert für Warnung kann konfiguriert werden. Standard: 5. Zusätzlich via SwiftLint möglich.
misst Anzahl an vorhandenen Schwachstellen	×	Funktionalität nicht vorhanden.
misst Anzahl an Verstößen gegen allgemeine Programmierrichtlinien	×	Nur Auflistung von Verstößen.
Möglichkeit zur Definition eigener Regeln	✓*	Via SwiftLint-Plugin, ansonsten nicht.
umfangreiches Reporting	✓	Report (html) enthält Liste mit gefundenen Issues. Zusätzlich sind teilweise Verbesserungshinweise hinterlegt. Für den Report stehen folgende Formate zur Verfügung: json, text, und html.

Tabelle C.0.2: Evaluierung Code Climate CLI

Kriterium	Erfüllung	Ergänzung/Bemerkung
Quality Gate vorhanden	✓	Build-in Quality Gate „sonar-way“ vorhanden. Eigene können definiert werden. (Beispiel des Quality Gates Status, siehe Abbildung B.0.5)
stellt Testabdeckung dar	✓	Externer Report kann importiert werden, sodass Testabdeckung in Übersicht angezeigt wird, siehe Abbildung B.0.4 Tests müssen vorher dementsprechend durchgeführt wurden sein.
erkennt bekannte Schwachstellen	✓	Schwachstellenanalyse auf Basis der OWASP Top 10, SANS Top 25 und CWE. Zusätzlich noch auf Basis eigener Daten von SonarSource. Außerdem kann externer Report von Mobsfscan importiert werden.
erkennt Secrets im Quellcode	✓*	Funktionalität standardmäßig nicht vorhanden. Möglichkeit besteht über External-Issues-Report (bspw. von SwiftLint).
führt Dependency-Scans durch	✓*	Funktionalität standardmäßig nicht vorhanden. Möglichkeit über Plugin von OWASP Dependency-Check besteht.
erkennt Verstöße gegen allgemeine Programmierrichtlinien	✓	125 eigene Regeln vorhanden. Weitere können mit Hilfe von externen Werkzeugen überprüft werden, deren Report importiert wird (bspw. von SwiftLint).
erkennt duplizierten Code	✓	Der duplizierte Code des Testfalls wurde erkannt. Zusätzlich wird eine Zeitschätzung zu Refactoring angegeben, siehe Abbildung B.0.12.
erkennt ungenutzte Deklarationen	×	Keine eigene Regel vorhanden. Auch keine Erkennung via SwiftLint.
erkennt Klassen mit mehr als 400 LoC	✓*	Erkennt nur LoC von Dateien oder Methoden. Für Klasse via SwiftLint möglich. (siehe Abbildung B.0.12)
erkennt fehlende Completion-Aufrufe	k.A.	Regeldefinition via SwiftLint möglich aber nicht umgesetzt und evaluiert.
erkennt Singletons	(✓*)	Erkennung standardmäßig nicht möglich. Mit Hilfe von SwiftLint können allerdings erste Indizien geliefert werden.
misst Lines of Code	✓	LoC wird für ein ganzes Projekt, eine Datei, Methode oder für gepushte Änderungen erfasst. Zusätzlich via SwiftLint möglich.
misst zyklomatische Komplexität	✓	Standard: 10. Nicht konfigurierbar. Zusätzlich via SwiftLint möglich.
misst Anzahl an vorhandenen Schwachstellen	✓	Wird in Übersicht angezeigt, siehe Abbildung B.0.4.
misst Anzahl an Verstößen gegen allgemeine Programmierrichtlinien	✓	Wird in Übersicht unter „Code Smells“ angezeigt, siehe Abbildung B.0.4.
Möglichkeit zur Definition eigener Regeln	✓*	Definition eigener Regeln über verschiedene Wege möglich. Für Swift allerdings nur über externe Werkzeuge wie SwiftLint.
umfangreiches Reporting	✓	Über Webinterface stehen ausführliche Reports, Ansichten und Zusammenfassungen zur Verfügung.

Tabelle C.0.3: Evaluierung SonarQube

Kriterium	Note	Begründung
Geringer Ressourcenverbrauch	1	GitLab-Instanz bereits vorhanden; kein weiterer Installations- und Wartungsaufwand als ohnehin; keine Schulungen bzgl. generellen Umgangs mit GitLab notwendig
Übersichtliche / informative Benutzeroberfläche	1	Übersicht aller Pipelines, Übersicht einzelne Pipeline inkl. Jobs, Übersicht einzelner Job vorhanden; Status, Dauer, Zeitstempel, Triggerer, Commit, Branch und Verwaltungsoptionen übersichtlich, logisch und klar dargestellt (siehe Abbildung B.0.6)
Vollständige und verständliche Dokumentation	1	sehr umfangreiche Dokumentation mit vielen Beispielen; klare Strukturierung; leicht verständlich in englischer Sprache
Zielführende / intuitive Bedienung	1	Menüpunkt in Seitenleiste mit CI/CD inkl. Untermenüs zu Pipeline, Editor für Bearbeitung der .gitlab-ci.yml-Datei, Jobs und Schedules für schnelle Zielerreichung; Verwendung von Signalfarben und Icons für Schaltflächen; keine versteckten Funktionalitäten, alles offensichtlich (siehe Abbildung B.0.6 und Abbildung B.0.7)
Geringe Gefahr durch Herstellerabhängigkeit	3	Vollständig abhängig von GitLab und deren Lizenzmodell
Übereinstimmung mit geforderten Funktionalitäten	1	weist alle geforderten Funktionalitäten auf und deckt diese vollständig und korrekt ab
Leicht anpassbar und änderbar	1	GitLab CI/CD kann für viele verschiedene Projekte genutzt werden; Pipelines können via Include- und Exclude-Mechanismen von Pipelines anderer Projekte eingebunden werden; Eltern-Kind-Pipelines; Jobs können einfach mit einem Punkt vor dem Namen deaktiviert werden; Umgebungsvariablen (bspw. Secrets) können sicher über Webinterface definiert werden
Eignung für Umgebung/Werkzeugkette	1	Optimale Interoperabilität mit GitLab; .gitlab-ci.yml-Datei über eingebauten Editor bearbeitbar; GitLab-Runner für macOS mit Shell- und Docker-Executor; sendet Feedback per Mail
Einfache manuelle Bestätigung für entsprechende Arbeitsschritte	1	Einfach und schnell über entsprechende Schaltflächen mit erkennbaren Icons (siehe Abbildung B.0.6 - Zahnrad, Play-Button)
Einfacher Zugang zu entstandenen Artefakten	1	werden in Übersichten entsprechend angezeigt und können heruntergeladen werden (siehe bspw. Abbildung B.0.7)
Einfache Steuerung der Pipeline	1	Pipeline kann über Schaltfläche gestartet werden; Pipeline oder nur bestimmte Jobs können über Schaltflächen abgebrochen oder erneut gestartet werden (siehe Abbildung B.0.6)
Nützliche Erweiterungsmöglichkeiten durch andere vorhandene Funktionalitäten	1	Verwendung von Templates; Merge-Result-Pipelines; Anzeige und Nutzung (Mergefreigabe) des Pipelinestatus in Merge Request; Erstellung von Releases; Tag-Pipelines und vieles mehr

Tabelle C.0.4: Evaluierung GitLab CI/CD

Kriterium	Note	Begründung
Geringer Ressourcenverbrauch	3	Jenkins muss erst auf einem Server installiert und entsprechend konfiguriert werden; erhöhter Ressourcen-, Installations- und Wartungsaufwand; Schulungen bzgl. generellen Umgangs notwendig
Übersichtliche / informative Benutzeroberfläche	1	Jenkinsansicht eher altbacken und unübersichtlich; über Blue-Ocean-Plugin sehr übersichtlich und informativ mit Branch, Commit, Dauer, Zeitstempel, Triggerer (siehe Abbildung B.0.8)
Vollständige und verständliche Dokumentation	2	Dokumentation von Jenkins selbst übersichtlich und mit Beispielen hinterlegt; Dokumentation bei Plugins eher durchwachsen
Zielführende / intuitive Bedienung	2	Plugins müssen sich teilweise erst zusammengesucht werden für gewisse Funktionalitäten; Bedienelemente teilweise versteckt; Blue-Ocean-Ansicht allerdings sehr intuitiv
Geringe Gefahr durch Herstellerabhängigkeit	1	Vollständig Open Source, kein Lizenzmodell
Übereinstimmung mit geforderten Funktionalitäten	1	weist alle geforderten Funktionalitäten auf und deckt diese vollständig und korrekt ab
Leicht anpassbar und änderbar	1	Jenkins kann für viele verschiedene Projekte genutzt werden; Vielzahl an Plugins; Projekte können auf Basis anderer Projekte erzeugt werden
Eignung für Umgebung/Werkzeugkette	2	Gute Interoperabilität mit GitLab, was reine Automatisierung angeht (Unterschiedliche Trigger, Pipelinestatus kann GitLab gemeldet werden)
Einfache manuelle Bestätigung für entsprechende Arbeitsschritte	1	Einfach und schnell über entsprechende Schaltflächen mit Messages (ggf. sogar parametrisierbar) (siehe Abbildung B.0.8)
Einfacher Zugang zu entstandenen Artefakten	1	extra Übersicht in Blue-Ocean (siehe Abbildung B.0.9)
Einfache Steuerung der Pipeline	1	Pipeline kann über Schaltfläche gestartet werden; Pipeline oder fehlgeschlagene Schritte können über Schaltflächen erneut gestartet werden (siehe Abbildung B.0.8)
Nützliche Erweiterungsmöglichkeiten durch andere vorhandene Funktionalitäten	3	diverse Erweiterungsmöglichkeiten über Plugins aber Jenkins ist und bleibt eine Automatisierungsserver, keine Entwicklungsplattform

Tabelle C.0.5: Evaluierung Jenkins

Anhang D

Konfigurationen und Quellcode

D.1 Installationshinweise zu verwendeten Werkzeugen

Werkzeuge und Installationshinweise:

- **Homebrew**[110]: Zur Installation vieler Werkzeuge auf macOS-Systemen eignet sich der Paketmanager Homebrew. Installationshinweise zu diesem sind auf der Website zu finden.
- **Xcode** kann über den Mac App Store oder Apples Entwicklerportal geladen und installiert werden.
- **GitLab** kann mit Hilfe dessen Dokumentation einfach geladen, installiert und konfiguriert werden.[111]
- **Jenkins** kann ebenfalls mit Hilfe dessen Dokumentation einfach geladen, installiert und konfiguriert werden.[112]
- **SonarQube**: Für die Verwendung von SonarQube ist eine Installation des SonarQube Servers erforderlich.[113] Weiterhin wird ein SonarScanner[114] benötigt. Für beides existiert eine umfangreiche Dokumentation.
- **SwiftLint**: Installation über Homebrew: `brew install swiftlint`
- **Code Climate CLI** setzt eine Installation von Docker[115] voraus kann auf macOS via Homebrew installiert werden:
  `brew tap codeclimate/formulae` und `brew install codeclimate`
- **Gitleaks**: Installation via Homebrew: `brew install gitleaks`
- **Mobsfscan**: Installation mittels Pythons Paketverwaltungsprogramm *pip*[116]:
  `pip3 install mobsfscan`

[110] Vgl. Homebrew: https://brew.sh/index_de (Zugegriffen am 13. Juli 2022)
[111] Vgl. „Install self-managed GitLab“: https://about.gitlab.com/install (Zugegriffen am 13. Juli 2022)
[112] Vgl. „Installing Jenkins“: https://www.jenkins.io/doc/book/installing/ (Zugegriffen am 13. Juli 2022)
[113] Vgl. Install Sonar Server: https://docs.sonarqube.org/latest/setup/install-server/ (Zugegriffen am 13. Juli 2022)
[114] Vgl. SonarScanner: https://docs.sonarqube.org/latest/analysis/scan/sonarscanner/ (Zugegriffen am 13. Juli 2022)
[115] Verwendet wurde Ranger Desktop: https://rancherdesktop.io (Zugegriffen am 13. Juli 2022)
[116] Vgl. pip: https://pypi.org/project/pip/ (Zugegriffen am 13. Juli 2022)

D.2 Konfigurationen

Quellcode D.2.1: Konfiguration von SwiftLint

```
opt_in_rules:
    - unused_declaration
    - unused_import

included:
    - DemoAppForPipeline

type_body_length:
    warning: 300
    error: 400

reporter: "html"

custom_rules:
    singleton:
        regex: "private init"
        message: "Maybe a singleton?"
        severity: warning
```

Quellcode D.2.2: ExportOptions.plist-Datei

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
    "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
    <key>compileBitcode</key>
    <true/>
    <key>destination</key>
    <string>export</string>
    <key>method</key>
    <string>ad-hoc</string>
    <key>provisioningProfiles</key>
    <dict>
        <key>de.test.demoappforpipeline</key>
        <string>DemoAppForPipeline Ad Hoc</string>
    </dict>
    <key>signingCertificate</key>
    <string>Apple Distribution</string>
    <key>signingStyle</key>
    <string>manual</string>
    <key>stripSwiftSymbols</key>
    <true/>
    <key>teamID</key>
    <string>DNGAJRUFNS</string>
</dict>
</plist>
```

D.3 Quellcode

Quellcode D.3.1: UI/E2E-Test-Beispiel mit XCTest

```
func testE2ETestExample() {
    let app = XCUIApplication()
    app.launch()

    let createTodoButton =
    app.navigationBars["Todo's"].buttons["Add"]
    createTodoButton.tap()

    // title for todo
    let titleTextField = XCUIApplication().alerts["Todo
    hinzufuegen"].scrollViews.otherElements.
    collectionViews.textFields["Todo eintragen"]
    titleTextField.typeText("E2E-Test")

    // create todo
    app.alerts["Todo hinzufuegen"].scrollViews.
    otherElements.buttons["Todo erstellen"].tap()

    // check if todos exists
    XCTAssertTrue(app.tables.cells.containing(.staticText,
    identifier: "E2E-Test").element.waitForExistence(timeout:
    1))

    // delete todo
    app.tables.cells.containing(.staticText,
    identifier:"E2E-Test").children(matching:
    .other).element(boundBy: 0).swipeLeft()
    app.tables.buttons["Delete"].tap()

    // check if todo is deleted
    XCTAssertFalse(app.tables.cells.containing(.staticText,
    identifier: "E2E-Test").element.waitForExistence(timeout:
    1))
}
```

Quellcode D.3.2: Testklasse für statische Tests

```
class Testclass {

    let unusedString: String = "I'm useless."

    func complexMethod(testA: Bool, testB: Bool, testC: Bool, testD: Bool, i: Int) {
        if testA && testB {
            if testC || testD {
                if testA && testC {
                    print("Test")
                } else if !testC && testD {
                    print("Test")
                } else {
                    print("Test")
                }
            } else {
                print("Test")
            }
        } else if testA && !testB && testC {
            if testD {
                print("Test")
            } else {
                print("Test")
            }
        } else {
            switch i {
                case 1: break
                case 2: break
                case 3: break
                case 4: break
                default: break
            }
        }
    }

    func duplicatedComplexMethod(testA: Bool, testB: Bool, testC: Bool, testD: Bool, i: Int) {
        // ...
    }

    func completionNotCalled(testCompletion: ((Bool) -> Void)) {
        return
    }

    func longMethod() {
        print("Test")
        // ... to inflate the class to over 400 lines
        print("Test")
    }
}
```

Anhang E

CD mit Quellcode

Eidesstattliche Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die im Literaturverzeichnis angegebenen Quellen sowie die explizit aufgeführten Hilfsmittel verwendet habe.

Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder noch nicht veröffentlichten Quellen entnommen sind, sind als solche kenntlich gemacht.

Die Zeichnungen oder Abbildungen in dieser Arbeit sind von mir selbst erstellt worden, ansonsten sind sie mit einem entsprechenden Quellennachweis versehen.

Diese Ausarbeitung ist in gleicher oder ähnlicher Form noch bei keiner anderen Prüfungsbehörde eingereicht worden.

Dresden, 15. Juli 2022

Kai Ciesielski