Transferleistung Theorie/Praxis* □

Matrikelnummer:	
Freigegebenes Thema:	
Studiengang, Zenturie:	

** Studierende, die unter den Anwendungsbereich der PVO bis 03.02.2015 fallen, fertigen Transferleistungen weiterhin in der Form von Praxisberichten an und der Begriff hält Einzug in das Abschlusszeugnis. Ab dem Jahrgang 2016 hat der Begriff vollumfängliche Gültigkeit. In der Kommunikation hält der Begriff Transferleistungen ab sofort Einzug.*

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einführung

Die Qualitätsüberprüfung von Software ist ein wichtiges Thema und kann teilweise mehr als 25 % des Budgets und der Zeit ausmachen [9, 1, 8]. Dazu werden in der Softwareentwicklung verschiedene Methoden und Strategien eingesetzt. Alle Methoden versuchen zu verifizieren, dass das zu testende System einwandfrei funktioniert. Die optimale Methode würde mithilfe eines allgemeinen Beweises zeigen, dass das System in jedem Fall korrekt arbeitet. Da diese Art von Verifikation sehr aufwendig, kompliziert und teuer ist, wird die Korrektheit meist nicht bewiesen. Stattdessen wird das System bestimmten Eingaben ausgesetzt und die Ausgabe überprüft. Weil es aber bereits bei kleinen Systemen nicht möglich ist alle Eingaben zu überprüfen[1], kann immer nur ein kleiner Teil der möglichen Testfälle ausgewählt werden. Ein Testfall ist dabei ein bestimmter Vorgang, der die Grundlage für die Durchführung eines Tests bildet. Er sollte mindestens folgendes definieren[10]:

- Die Eingaben, die während des Testes gemacht werden sollen, und
- die Ausgaben, die während, beziehungsweise das Ergebnis, das nach dem Test erwartet wird.

Da verschiedene Methoden unterschiedlich viel Mehrwert bringen, ist für eine erfolgreiche Qualitätssicherstellung eine geeignete Wahl der Testmethoden notwendig. Diese Arbeit wird darstellen, ob und wie UI Tests[2] die Qualität für iOS Apps verbessern kann. Dazu werden zunächst die Testmethoden „Manuelles Testen“, „Unit Tests“ und „UI Tests“[3] einzeln anhand festgelegter Kriterien analysiert. Anschließend wird von der Analyse ausgehend ein Vergleich der Testmethoden durchgeführt, um die Fähigkeit von UI Tests die Qualität von iOS Apps zu verbessern einordnen und abschließend eine Handlungsempfehlung aussprechen zu können.

2 Analyse der Tests

Im Folgenden werden die drei Testmethoden „Manuelles Testen“, „Unit Tests“ und „UI Tests“ anhand von festgelegten Kriterien analysiert. Ausgewählt wurden die beiden Vergleichsmethoden, da sie zusammen die drei Standardmethoden zum Vorbeugen von Fehlern für iOS Apps sind und von Apple supported werden[4]. Nicht betrachtet werden andere Methoden wie Crashre-

[1] Bereits eine Addition von zwei 64 Bit Zahlen ergeben sich $2^{64+64} \approx 3.4 \cdot 10^{38}$ verschiedene Testfälle.

[2] Bei der iOS App Entwicklung ist der Name UI Test der geläufigste. Ein anderer Name für diese Art von Tests ist „Automated Graphical User Interface (GUI) testing“. Im weiteren Verlauf wird UI Tests verwendet.

[3] Auf die Auswahl der Testmethoden wird in Abschnitt 2 weiter eingegangen.

[4] Die Relevanz von Apple für die Auswahl der Testmethoden begründet sich mit Apple als Hersteller der Hardware, des Betriebssystems und vieler Entwicklertools, unter anderem der Testingtools und der UI-Library.

ports oder Rückmeldung von Usern, da diese keine Fehler vor dem Release finden können und damit etwas anders betrachtet werden müssen.

Zunächst wird jeweils beschrieben, wie die Tests auf iOS eingesetzt werden, um eine Grundlage für die weitere Analyse zu schaffen. Zur Analyse werden drei Kriterien eingesetzt. Zuerst wird untersucht, was die Tests abdecken können. Dies ist relevant, da nicht jede Testmethode dieselben Aspekte einer Software beziehungsweise einer App testen kann. Das zu testende System wird im Folgenden als System Under Test (SUT) bezeichnet. Daraufhin wird untersucht, wie zuverlässig die Testmethoden Fehler im SUT erkennen können. Dabei spielen Fehler, die nicht erkannt werden sollen keine Rolle. Abschließend wird jeweils analysiert, wie viel Aufwand in die Erstellung, Wartung und Durchführung investiert werden muss.

2.1 Manuelles Tests

Unter manuellem Testen versteht man das Bedienen und Beobachten der Software durch einen Menschen. Dieser entscheidet auch, ob die Software den Testfall bestanden hat.

iOS Manuelles Testen funktioniert auf iOS wie auf anderen Plattformen auch. Die App wird entweder manuell oder automatisiert zum Beispiel durch ein Continuous Integration System gebaut und anschließend auf einem physischen iOS Gerät oder in einem Simulator installiert. Hier kann durch eine Person die App bedient und beobachtet werden. Diese Testmethode kann sowohl von den Entwicklern während der Entwicklungsphase als auch von den entsprechenden Verantwortlichen zur Abnahme der Änderungen eingesetzt werden.

Testabdeckung Mit manuellem Testen interagiert der Tester so mit der App, wie auch ein Endnutzer mit der App interagieren wird. Das bedeutet, dass sowohl die Eingabe als auch die Ausgabe genau so funktionieren, wie der Endnutzer die App verwenden wird. Zusätzlich können weitere Tools eingesetzt werden, um mehr Informationen über die App zu erhalten.

Ausgabe Es wird jeweils wahrgenommen, was auf dem Display ausgegeben wird oder das Gerät in Form von Tönen oder Vibration von sich gibt. Das bedeutet, es können Fehler in der Ausgabe erkannt werden. Dazu kann auch das Erkennen unpassender Farben oder langsame Animationen gehören.

Eingabe Zu den Eingaben bei iOS Apps gehört das einfache Klicken auf den Bildschirm, die Durchführung komplexer Gesten[5] oder die Sensorik des Gerätes[6]. Der Tester hat hier im Allgemeinen dieselben Möglichkeiten wie der Nutzer.

Weitere Tools Mit weitere Tools kann zum Beispiel die Internetkommunikation oder Logdateien mitgelesen werden, um weiter zu verifizieren, dass sich die App richtig verhält. Es ist auch möglich die App einer Testumgebung auszusetzen, um bestimmtes Verhalten zu erzwingen. Auch kann die App für die Tests speziell verändert werden, um den Tester mehr Eingriffsmöglichkeiten zu geben. Dies erhöht jedoch den Einrichtungs- und Wartungsaufwand.

Exploratives Testen Durch die menschliche Komponente ist beim manuellen Testen auch das explorative Testen möglich. Das bedeutet, dass der Tester ohne vorher definierte Testfälle die App untersucht und so auf andere Fehler stoßen kann.

Zuverlässigkeit Die Zuverlässigkeit ist stark vom Tester und der investierten Zeit abhängig. Da jeder Tester unterschiedlich aufmerksam die App beobachtet, können unterschiedlich viele Fehler entdeckt werden. Die Tests sind nicht immer reproduzierbar, da die Durchführung nicht immer identisch ist.

Aufwand Um manuelle Tests durchzuführen ist grundsätzlich keine Vorbereitung und Wartung notwendig. Es ist jedoch sinnvoll vorher Testfälle zu definieren, um dem Testen eine Struktur zu geben (Exploratives Testen ist eine Ausnahme). Wenn dies gemacht wird ist der Aufwand aber immer noch relativ gering.

Ein großer Nachteil am manuellen Testens ist die Notwendigkeit, dass eine Person jeden Testfall einzeln durchführen muss. Damit ist der Aufwand für die Durchführung erheblich. Da sich eine App auf verschiedenen Geräten unterschiedlich verhält, sollten (alle) Änderungen immer auf allen oder möglichst vielen Geräten getestet werden. Der Aufwand ist dadurch vor allem von zwei Faktoren abhängig: Die Anzahl der Testfälle, die mit der Komplexität der App zunehmen, und die Anzahl der Geräte, auf denen getestet werden soll.

Aufgrund des geschlossenen Betriebssystems lässt sich die Anzahl der verschiedenen Geräte relativ leicht bestimmen. Angenommen wird im Folgenden eine App die auf den Betriebssystemen iOS 10, iOS 11 sowie auf der aktuellen Beta iOS 12 installiert werden soll, da damit 95 % aller aktiven iOS Geräte abgedeckt werden (Abbildung 1). Unter dieser Annahme gibt es 34 verschiedene Modelle auf denen die App installiert werden kann [5, 6]. In den meisten

[5] Komplexe Gesten, die über das Scrollen und Zoomen hinausgehen, können nur auf einem physischen Gerät getestet werden.

[6] Auch hier gibt es Einschränkungen im Simulator.

Auflösung	ppi	Geräte
640 x 1136	326ppi	iPhone SE, iPhone 5, iPhone 5s, iPhone 5c, iPod Touch 5th generation
750 x 1334	326ppi	iPhone 6, iPhone 6s, iPhone 7, iPhone 8
1242 x 2208	401ppi	iPhone 6 Plus, iPhone 6s Plus, iPhone 7 Plus, iPhone 8 Plus
1125 x 2436	458ppi	iPhone X
1536 x 2048	326ppi	iPad mini 2, 3, 4
1536 x 2048	264ppi	iPad 4th generation, iPad 5th generation, iPad Air, iPad Air 2, iPad Pro 9.7 inch
1668 x 2224	264ppi	iPad Pro 10.5 inch
2048 x 2732	264ppi	iPad Pro 12.9 inch
272 x 340	326ppi	Apple Watch 38mm
312 x 390	326ppi	Apple Watch 42mm

Tabelle 1: Geräte nach Bildschirmauflösung

Fällen ist es nicht notwendig auf allen Geräten zu testen. Es ist jedoch sinnvoll zumindest alle Testfälle auf allen Bildschirmen zu testen, um sicherzustellen, dass die UI den eigenen Ansprüchen entspricht. Dabei bleiben wie in Tabelle 1 dargestellt immer noch zehn Geräteklassen übrig, auf denen die App getestet werden muss. Je nach App und Anforderungen kommen hier noch Geräte dazu oder können entfallen, da es neben dem Bildschirm noch weitere Unterschiede in der Hardware gibt. Da sich Apps auch auf verschiedenen iOS Versionen unter Umständen unterschiedlich verhalten, kann es zusätzlich noch nötig sein, die App auf den verschiedenen Versionen zu testen. Durch die Vielzahl der möglichen Faktoren kann das manuelle Testen sehr zeitaufwendig sein.

Der Aufwand des manuellen Testens wächst mit der Komplexität der App schnell an, ist aber auch bei kleinen Projekten bereits groß. Das kann dazu führt, dass ein Projekt, das nur manuelle Testmethoden verwendet, auf Dauer zu wenig getestet wird. Deshalb verwenden die meisten (87 %) mindestens eine Form des automatisierten Testens[12].

2.2 Unit Tests

Unit Test sind eine Art von Test, bei der einzelne Module (Units) aus dem Gesamtprozess raus gelöst getestet wird. Auf diese Weise kann verifiziert werden, dass jedes einzelne Modul für sich korrekt arbeitet. Um dies zu erreichen wird das Modul nicht im Kontext der App, sondern in einem Testkontext aufgerufen. Das bedeutet, dass sowohl alle aufrufenden als auch aufgerufenen Module durch speziell für die Unit Tests erstellte ersetzt wird.

iOS Unit Tests auf iOS werden häufig mit XCTest (Xcode Test)[7] oder einem Wrapper um XCTest erstellt, da Xcode die Ergebnisse anzeigen kann.

Es werden einzelne Testfälle definiert und implementiert, die anschließend beliebig häufig ohne weitere Eingriffe automatisiert durchgeführt werden können. Wie die Tests im einzelnen Aussehen ist stark von dem verwendeten Paradigmen abhängig. Abbildung 2 zeigt ein Beispiel für einen Unit Test, der mit XCTest erstellt wurde und die Bedingungen für einen Testfall erfüllt.

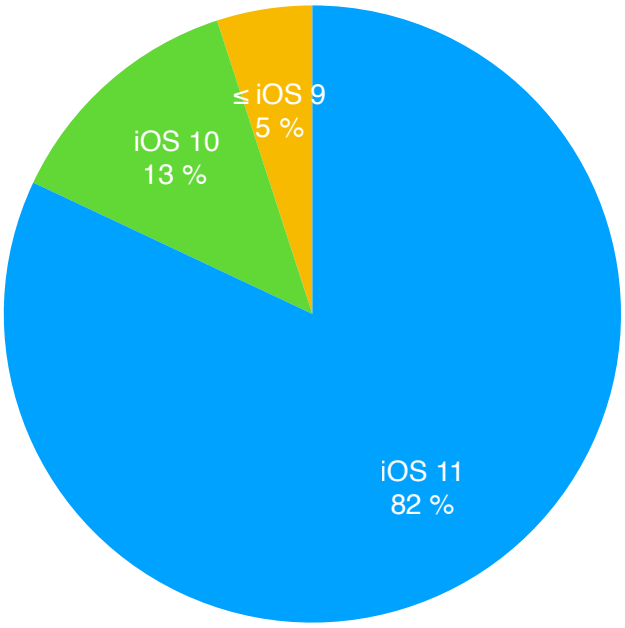

Abbildung 1: Apple kann mithilfe des App-Stores ermitteln, welche iOS Versionen aktiv verwendet werden. Zum Zeitpunkt der Veröffentlichung dieser Daten (31. Mai 2018) war iOS 11 die jüngste Version[3].

Testabdeckung Unit Tests können die Funktionalität einzelner Logikmodule isoliert testen. Dabei werden vor allem Durch diese Isolation können Fehler auf genau ein Modul lokalisiert werden.

Zu beachten ist das Attribut `@testable` vor dem `import` Statement (Zeile 7) im Beispielcode (Abbildung 2). Dies ermöglicht auch den Zugriff auf `internal`[8] Entities[9], sodass noch kleinere Einheiten getestet und Fehler einfacher lokalisiert werden können[2].

Zuverlässigkeit Da Unit Tests wie alle Testmethoden, nicht alle möglichen Ein- und Ausgaben testen können, muss immer eine Auswahl getroffen werden. Eine gute Auswahl sollte möglichst alle Grenzfälle abdecken und nicht nur Erfolgsfälle, sondern auch Fehlerfälle testen. Ein Maß für die Auswahl ist Code Coverage. Code Coverage gibt an, welche Teile des Programmcodes während des Tests ausgeführt wurden. Auf diese Weise kann man feststellen, ob

[7]XCTest ist ein Testing framework von Apple für ihre Plattformen iOS, macOS, tvOS und watchOS und der Nachfolger von OCUnit[4]. Das Framework wird seit Xcode 5 von Apple ausgeliefert und unterstützt in der ersten Version iOS 7 und OS X v10.8[7].

[8]In Swift gibt es fünf Access Control Levels. Normalerweise können Module (Libraries, Executables, Tests, etc.) nur auf `open` und `public` Entities eines anderen Moduls zugreifen. `internal` Entities sind hingegen nur innerhalb desselben Moduls erreichbar.

[9]Entities sind in Swift Klassen und ähnliches (`class`, `struct`, `enum`, `typealias`) sowie Funktionen und Properties (`func`, `init`, `subscript`, `let`, `var`).

alles Programmteile mindestens einmal getestet wurde. Bei einer guten Auswahl der Testfälle sind Unit Tests sehr zuverlässig und können bereits kleine Abweichungen von dem erwarteten Ergebnis finden.

Aufwand Da Unit Tests vollständig automatisierte Tests sind, entsteht für die Durchführung keinen Aufwand. Verwendet man ein Continuous Integration System werden die Tests automatisch gestartet und man wird im Fehlerfall informiert. Sehr schnell laufende Tests können noch häufiger während der Entwicklungsphase gestartet werden.

Die Wartung von Unit Tests sollte im Normalfall gering sein, da eine funktionierende API auch bei internen Änderungen bei denselben Eingaben dieselben Ausgaben liefern sollte. Erstellte Tests sollten also nur im Fall von Bugs angepasst werden müssen.

Das Erstellen von Unit Tests kann sehr unterschiedlich aufwendig sein. Werden die Tests aber zusammen mit der Entwicklung neuer Module implementiert hält sich der Aufwand in Grenzen, da der gedankliche Aufwand für die Funktionsweise sowohl für die Implementierung als auch für die Tests genutzt werden kann.

2.3 UI Tests

Mit UI Tests kann getestet werden, ob die Benutzeroberfläche den Spezifikationen, auch nachdem Änderungen durchgeführt wurden, entspricht.

iOS Auch die UI Tests können auf iOS mit Apples XCTest Framework entwickelt werden. Dabei laufen die Tests als eine eigenständiger, von der getesteten App unabhängigem Prozess. Interagieren können die Tests nur über die Accessibility API[10] mit der App. Das bedeutet, dass die Tests *keinen* Zugriff auf die Funktionen der App haben und auch nicht das Ergebnis so überprüft werden kann, wie es bei Unit Tests der Fall ist. Stattdessen werden alle Eingaben und Ausgaben über die Accessibility API gemacht. Das führt dazu, dass die Tests weniger Informationen über die App erhalten können. Andererseits können die UI Tests aber Informationen erhalten, die ähnlich zu denen sind, die ein Endnutzer bei der Verwendung der App zu Verfügung hat. Außerdem kann man über die Accessibility nicht nur die eigene App, sondern das ganze Betriebssystem und andere Apps bedienen. Dadurch wird ermöglicht, dass Extensions und andere Integrationen in das Betriebssystem getestet werden können.

[10]Die Accessibility API ist eine API von Apple, um dem System die Möglichkeit zu geben, mehr Informationen über die UI zu erhalten oder mit ihr zu interagieren, sodass auch Nutzer die Probleme bei der Bedienung haben (zum Beispiel aufgrund von eingeschränktem Sehvermögen) in der Lage sind das System zu bedienen. Über die Accessibility API wird dem System Informationen wie UI Element Typen, Beschreibung, Größe und Position gegeben sowie die Möglichkeit zu tippen oder zu scrollen.

Es gibt drei Generationen von automatisierten Tests für Benutzeroberflächen die nach dee „Capture & Replay“ Methode funktionieren[11]:

1. Das Speichern und Abspielen von Koordinaten.
2. Mithilfe von Zugriff auf UI Komponenten.
3. Mithilfe von Bilderkennung.

UI Tests mit XCTest fallen dabei in die zweite Generation[11] und laufen damit deutlich stabiler als Frameworks der ersten Genration und haben noch nicht die Komplexität von Bilderkennung[11].

Um UI Tests zu erstellen, gibt es für iOS zwei Wege. Entweder schreibt man den UI Test Code oder man lässt ihn von Xcode generieren, indem man die App bedient und alle Eingaben aufgezeichnet werden. Das Generieren von UI Tests bringt aber viele Probleme mit sich. Dazu gehört

1. Der so entstehende UI Test Code kann sehr unleserlich sein, sodass eine Wartung nur aufwendig möglich ist.
2. Es werden keine richtigen Tests erzeugt, sondern nur die Interaktion aufgenommen. Solche Tests überprüfen nicht die Folgen, die eine Interaktion hat. Damit erfüllen sie nicht das zweite Kriterium eines Testfalls[12].
3. Die generierten Tests können dynamischen Inhalten nicht erkennen. Beispielsweise wird ein Tippen auf ein Button mit dem heutigen Datum auch an anderen Tagen versuchen einen Button mit dem Datum der Aufzeichnung zu finden und fehlschlagen.
4. Zwischen den einzelnen Interaktionen werden Pausen nicht aufgezeichnet, sodass die Tests bei Animationen oder Netzwerklatenz fehlschlagen können.

Ein fertiger UI Test ist in Abbildung 3 dargestellt. Wie derselbe Test durch die automatisch generierten Tests aussehen kann, ist in Abbildung 4 dargestellt.

[11]Es ist jedoch weiterhin möglich mithilfe von Koordinaten die App zu bedienen. Außerdem stellt XCTest Methoden zu Verfügung, um Screenshots zu erstellen, sodass auch UI Tests der dritten Generation denkbar wären.

[12]Die Tests haben indirekt Ausgabeüberprüfungen, da sie fehlschlagen, wenn zum Beispiel ein Button nicht da ist, der getippt werden soll. Für richtige Tests sollte aber auch überprüft werden, ob alle anderen wichtigen UI Änderungen erfolgreich waren.

Testabdeckung Wie in Abschnitt iOS beschrieben interagieren die Tests durch die Accessibility API mit der Benutzeroberfläche der App. Es wird also, ähnlich zu den manuellen Tests, getestet, was auf dem Bildschirm angezeigt wird. Da beim Testen die ganze App gestartet wird, kann getestet werden, ob alle Einzelkomponenten zusammen funktionieren und das Durchführen von komplexen Workflows möglich ist. Bei Änderungen kann so verfolgt werden, welche Teile der App beeinflusst werden und gegebenenfalls noch angepasst werden müssen.

Zuverlässigkeit Es ist relativ einfach UI Tests zu schreiben, die fälschlicherweise fehlschlagen. Beispiele für häufige Fehlerquellen:

- **Schlecht gewählte Timeouts:** Viele Apps machen asynchrone Netzwerkanfragen oder haben Animationen, sodass die Tests auf die App warten muss. Sind die Timeouts zu kurz gewählt können die Tests fehlschlagen, obwohl alles wie gewollt funktioniert. Sind die Timeouts zu lang gewählt, führt dies im Fehlerfall zu unnötig langen Tests.
- **Dynamische Inhalte:** Wenn dynamische Inhalte in der App angezeigt werden, muss darauf geachtet werden, was als richtig erkannt werden soll. Kommen unerwartete Inhalte, die aber richtig sind, schlägt der Test fehl.
- **Lokalisation:** Selbst bei einsprachigen Apps können Texte anders aussehen, weil zum Beispiel ein anderes Datumsformat im System eingestellt ist.
- **iPad Tests:** Auf iPads sind zum Beispiel bei der Verwendung eines `UISplitViewControllers`[13] mehr UI Elemente gleichzeitig sichtbar. Deshalb entfällt unter Umständen ein weiterer Zurückbutton oder mehrer UI Elemente haben dieselbe Signatur.

Wenn jedoch einen UI Test erstellt wurde, der die oben genannten Probleme nicht hat, kann dieser sehr zuverlässig überprüfen, ob die gewünschte Funktionalität korrekt arbeitet. Selbst bei Tests mit den Problemen gibt es vor allem falsch positive Ergebnisse, echte Fehler lassen die Tests weiterhin fehlschlagen. Bei UI Tests werden also mehr Fehler angezeigt, als eigentlich gefunden wurden.

Werden UI Tests regelmäßig ausgeführt können auch sporadisch auftretende Fehler gefunden werden, die bei einzelnen Tests einfacher übersehen werden.

[13] Ein Standardelement der UI Library, das auf iPads zwei Views nebeneinander und auf iPhones nacheinander anzeigt.

Aufwand Da UI Tests vollständig automatisierte Tests sind, gibt es für die Durchführung keinen Aufwand. Die Durchführung kann aber unter Umständen lange dauern, da für jeden Test die ganze App gestartet werden muss.

Dafür ist der Aufwand zum Erstellen und Warten größer. Dies kann vor allem auf zwei Faktoren zurückgeführt werden. Zunächst ist es relativ aufwendig einen einzelnen Test zu schreiben. Aufgrund der zuvor genannten Probleme ist es nicht möglich sinnvolle UI Tests nur mithilfe der Aufnahme Funktion zu erstellen. Stattdessen müssen die Tests manuell erstellt oder überarbeitet werden. Aufgrund der Komplexität von Benutzeroberflächen kann dies Zeitintensiv sein.

Der zweite Faktor ist die Anzahl der UI Tests, die entstehen. Bereits bei kleinen Apps kann es viele UI Aktionen geben. Bei vielen dieser Aktionen ist die Reihenfolge von entscheidender Bedeutung für die Ausgabe, sodass manche Aktionen mehrfach getestet werden müssen, wobei jeweils andere Vorbedingungen eingerichtet werden müssen.

Da die UI Tests der zweiten Generation der Capture und Replay Tests angehören, schlagen die Tests jedoch noch nicht fehl, wenn kleinere Layout Änderungen gemacht wurden, sodass die Tests in solchen Fällen nicht angepasst werden müssen.

3 Vergleich

	Manuell	Unit Tests	UI Tests
SUT	UI	API/Logik	UI (Accessibility API)
Aufwand	Hoch	Gering	Mäßig
Einrichtung	Gering	Mäßig	Hoch
Wartung	Gering	Gering	Mäßig
Durchführung	Hoch	Keiner	Keiner
Zuverlässigkeit	Mäßig	Hoch	Hoch

Tabelle 2: Vergleich der Tests

3.1 UI Tests im Vergleich zum manuellem Testen

UI Tests haben einen ähnlichen Einsatzbereich wie das manuelle Testen, da beide versuchen Fehler in der grafischen Benutzeroberfläche zu finden. Dennoch gibt es einige wichtige Unterschiede.

Die Stärke von UI Tests liegt in der vergleichsweise zuverlässigen Sicherstellung der Durchführbarkeit aller Workflows. Außerdem kann mithilfe von UI Tests aufgrund des nicht vorhandenen Auf-

wands bei der Durchführung regelmäßig und häufiger getestet werden. Dies ermöglicht das frühe Erkennen von Fehlern.

Die Stärke vom manuellem Testen liegt in der menschlichen Komponente. Diese Art der Tests sollen eine Benutzeroberfläche testen, die für die Bedienung durch Menschen gedacht ist. Vor allem Aspekte wie Animationen lassen sich durch UI Tests nicht abbilden, sind aber für eine vernünftige Benutzeroberfläche einer App wichtig.

UI Tests sind deshalb in der Lage Aufgaben vom manuellem Testen zu übernehmen, um hier die Qualitätssicherung zu vereinfachen und dadurch zu verbessern.

3.2 UI Tests im Vergleich zu Unit Tests

UI und Unit Tests verfolgen verschiedene Ziele. Deshalb unterscheiden sich die Testmethoden in vielen Bereichen. Während UI Tests Fehler in der Benutzeroberfläche aufdecken können, versuchen Unit Tests die korrekte Funktionalität von der Logik der App zu testen. Unit Tests eignen sich besonders, wenn komplexe Logik getestet werden soll, da dafür die restliche App keine Bedeutung hat. UI Tests hingegen starten die ganze App und können gut überprüfen, ob die Grundfunktionalität da ist.

Aufgrund der großen Unterschiede können UI Tests eine Ergänzung zu Unit Tests sein, um einen weiteren Teil der App automatisiert zu testen und dadurch die Qualitätssicherung zu verbessern.

4 Zusammenfassung

In keinem Fall können UI Tests die beiden anderen Testmethoden ersetzen, können aber in manchen Fällen eine sinnvolle Ergänzung sein.

Wenn bei einer App die Benutzeroberfläche einfach gestaltet ist oder sich die UI kaum verändern wird, lohnen sich UI Tests besonders, da nach dem einmaligem Erstellen eines Tests fast aufwandsfrei Fehler gefunden werden können, die beim Bearbeiten anderer Teile (zum Beispiel Logik oder Backend) entstehen. Bei Apps, die eine lange Lebensdauer haben werden, kann der Einsatz von UI Tests sinnvoll sein, da das manuelle Testen bestimmter Abläufe auf Dauer sehr aufwendig sein kann. Besonders eignen sich hier Features wie Onboarding, Hilfe, Impressum und ähnliches. Dies sind Abläufe, die sich relativ selten ändern sollen, aber gegebenenfalls größere Auswirkungen haben, wenn diese nicht funktionieren.

Wenn eine App eine sich ständig ändernde Benutzeroberfläche hat, weil sie zum Beispiel noch am Anfang der Entwicklungsphase steht und gegebenenfalls noch nicht klar ist, wie die fertige Benutzeroberfläche aussehen wird, lohnen sich UI Tests weniger, weil der Aufwand für

die Erstellung und Wartung der Tests sehr hoch ist. Auch bei sehr kleinen Apps oder Apps mit einer kurzen Lebensdauer ist der Einsatz von UI Tests nicht unbedingt sinnvoll, da diese Apps meist überschaubar sind und die Erstellung von aufwendigeren UI Tests sich nicht rentiert.

Ob der Einsatz von UI Tests die Qualität der Apps verbessern kann, hängt damit sehr stark von der App selbst ab. Viele Faktoren spielen eine Rolle, sodass es bei manchen Apps sinnvoller ist die Zeit nicht in UI Tests zu investieren, sondern beim manuellen Testen und dem Einsatz von Unit Tests zu bleiben beziehungsweise damit zu beginnen um auf diese Weise die Qualität sicher zu stellen. Andere Apps können hingegen vom Einsatz von UI Tests auf Dauer profitieren.

5 Anhang

5.1 Codebeispiele

Unit Tests

```swift
// MyAppTests.swift

// Zunächst muss das Testframework importiert werden.
import XCTests

// Außerdem muss die App importiert werden.
@testable import MyApp

// Alle Testclassen müssen von der Klasse XCTestCase erben.
class MyAppExampleTestCase: XCTestCase {

    // Die einzelnen Testfälle müssen mit test beginnen.
    func testMyClass() {

        // In den einzelnen Testfällen müssen die zu
           testenden Module erstellt werden.
        let myClass = MyClass()

        // Alle Abhängigkeiten werden durch spezielle Module
            ersetzt.
        myClass.magicNumberGenerator = TestNumberGenerator(
           number: 42)

        // Zuletzt muss das Ergebnis überprüft werden.
        XCTAssertEqual(myClass.magicNumber, 42,
            "Expected magic number to equal 42.")
    }
}
```

Abbildung 2: Beispiel für einen Unit Test in Swift. In diesem Unit Test wird zunächst ein Objekt der Klasse `MyClass` erstellt (Zeile 16). Dies stellt eine Eingabe des Tests dar. Anschließend wir überprüft, ob die Property `magicNumber` gleich 42 ist (Zeile 22). Dies stellt die Überprüfung der Ausgabe dar. Damit werden alle Bedingungen für einen vollständigen Testfall erfüllt.

UI Tests

```swift
// MyAppUITests.swift

// Zunächst muss das Testframework importiert werden.
// Die App muss hier nicht importiert werden.
import XCTests

// Auch alle UI Test Klassen müssen von der Klasse XCTestCase erben.
class MyAppExampleManualUITestCase: XCTestCase {

    // Die einzelnen Testfälle müssen mit test beginnen.
    func testLoginWithMissingPassword() {

        // Zu Beginn des Tests wird die App gestartet.
        let app = XCUIApplication()
        app.launch()

        // Dann kann mit der App interagiert werden.
        let usernameTextField = app.textFields["username"]
        usernameTextField.tap()
        usernameTextField.type("Benutzername")

        app.buttons["login"].tap()

        // Auch hier muss zuletzt das Ergebnis überprüft werden.
        XCTAssertEqual(
            app.staticTexts["status"].label,
            "Das Passwort darf nicht leer sein.",
            "Expected empty password error message.")
    }
}
```

Abbildung 3: Beispiel für einen manuell erstellten UI Test in Swift mit `XCTest`

In diesem Testfall wird getestet, wie sich die App beim Login verhält, wenn man einen Benutzernamen, aber kein Passwort eingibt und versucht sich anzumelden.

```swift
// MyAppUITests2.swift

// Zunächst muss das Testframework importiert werden.
import XCTests

class MyAppExampleAutomaticUITestCase: XCTestCase {
    // So könnte ein UI Test aussehen, der mit der Aufnahme Funktion von Xcode erstellt wurde.
    func testLoginWithMissingPassword() {
        let app = XCUIApplication()
        app.launch()

        // Unaussagekräftige Anweisungen
        let textField = app.otherElements.element(boundBy: 1).scrollViews.element.textField["username"]
        textField.tap()
        textField.type("Benutzername")

        // Unnötige Zuweisungen
        let app2 = app

        app2.otherElements.element(boundBy: 1).scrollViews.element.buttons["Login"].tap()

        // Fehlende Überprüfung des Ergebnisses.
    }
}
```

Abbildung 4: Beispiel für einen automatisch erstellten UI Tests in Swift mit `XCTest`

Wie in Abbildung 3 wird auch hier der Login Prozess getestet. Dieses mal wurde der Code jedoch von Xcodes Aufnahme Funktion aufgenommen.
Die Kommentare sind nicht Teil des automatisch erstellten Tests.

Literatur

[1] Emil Alégroth, Robert Feldt und Pirjo Kolström. "Maintenance of automated test suites in industry: An empirical study on Visual GUI Testing". In: *Information and Software Technology* 73 (2016), S. 66–80. ISSN: 09505849. DOI: 10.1016/j.infsof.2016.01.012. arXiv: 1602.01226.

[2] Apple Inc. *Access Levels for Unit Test Targets — The Swift Programming Language (Swift 4.2)*. 2018. URL: https://docs.swift.org/swift-book/LanguageGuide/AccessControl.html%7B%5C#%7DID519 (besucht am 23. 07. 2018).

[3] Apple Inc. *App Store - Support - Apple Developer*. 2018. URL: https://developer.apple.com/support/app-store/ (besucht am 23. 07. 2018).

[4] Apple Inc. *Appendix B: Transitioning from OCUnit to XCTest*. 2017. URL: https://developer.apple.com/library/archive/documentation/DeveloperTools/Conceptual/testing%7B%5C_%7Dwith%7B%5C_%7Dxcode/chapters/A2-transitioning%7B%5C_%7Docunit%7B%5C_%7Dto%7B%5C_%7Dxctest.html%7B%5C#%7D//apple%7B%5C_%7Dref/doc/uid/TP40014132-CH10-SW1 (besucht am 23. 07. 2018).

[5] Apple Inc. *iOS 10 Product Page*. 2017. URL: http://web.archive.org/web/20170603042345/https://www.apple.com/ios/ios-10 (besucht am 23. 07. 2018).

[6] Apple Inc. *iOS 11 Product Page*. 2018. URL: https://www.apple.com/ios/ios-11/ (besucht am 23. 07. 2018).

[7] Apple Inc. *Testing Basics*. 2017. URL: https://developer.apple.com/library/content/documentation/DeveloperTools/Conceptual/testing%7B%5C_%7Dwith%7B%5C_%7Dxcode/chapters/03-testing%7B%5C_%7Dbasics.html (besucht am 23. 07. 2018).

[8] Application Developers Alliance. "Developer Insights Report – A Global Survey of Today's Developers". In: August (2015). URL: insights.abnamro.nl.

[9] CapGemini. *World Quality Report 2017–18*. Techn. Ber. 2017. URL: https://www.sogeti.com/globalassets/global/downloads/testing/wqr-2017-2018/wqr%7B%5C_%7D2017%7B%5C_%7Dv9%7B%5C_%7Dsecure.pdf.

[10] Cem Kaner. "What Is a Good Test Case?" In: *Software Testing Analysis & Review Conference (STAR East)* (2003), S. 1–16. URL: http://www.kaner.com/pdfs/GoodTest.pdf.

[11] Antonia Kresse und Peter M. Kruse. "Development and maintenance efforts testing graphical user interfaces: a comparison". In: *Proceedings of the 7th International Workshop on Automating Test Case Design, Selection, and Evaluation - A-TEST 2016* (2016), S. 52–58. DOI: 10.1145/2994291.2994299. URL: http://dl.acm.org/citation.cfm?doid=2994291.2994299.

[12] SauceLabs. *Testing Trends in 2017: a Survey of Software Professionals*. Techn. Ber. January. 2017. URL: http://info.saucelabs.com/rs/468-XBT-687/images/Sauce%20Labs%20-%20State%20of%20Testing%20Survey%20Results%20Jan,%202017.pdf.