2.1.2 Geschäftsmodelle für Apps

In der Regel stehen geschäftliche Ziele hinter der Entwicklung einer mobilen App. Im Zuge dessen wurden im Laufe der letzten Jahre vielfältige Geschäftsmodelle für Apps entwickelt. Wir werden wichtige, weitverbreitete Modelle in diesem Kapitel vorstellen und kurz ihre Vor- und Nachteile diskutieren. Dabei ist zu beachten, dass die Modelle sowohl in Reinform als auch in Mischformen zum Einsatz kommen und daher die Grenzen häufig nicht eindeutig erkennbar sind. Für den Tester ist ein Verständnis der Modelle wichtig, um in der Teststrategie und bei der Testplanung entsprechende Tests berücksichtigen zu können. Solche Tests prüfen, ob das Geschäftsmodell für die App sinnvoll anwendbar ist oder aber negativen Einfluss auf das Nutzererlebnis hat.

Kauf-Apps

Apps können verkauft werden wie jede andere Software auch. In der Regel stellen die Stores der Plattformbetreiber sowie Stores von Drittanbietern Funktionen bereit, über die der Anwender einen festgelegten Betrag bezahlt, bevor er die Applikation herunterladen und installieren kann.

Aufgrund der vielen Apps in den Stores, von denen viele kostenfrei heruntergeladen und genutzt werden können, ist dieses Modell nur für wenige Apps geeignet. Dies ist mit einer der Gründe, warum die meisten Apps in den Stores kostenfrei verfügbar sind. Zu fast jeder Kauf-App gibt es kostenlose Konkurrenz mit mehr oder weniger gleichem Funktionsumfang (vgl. Abb. 2–4).

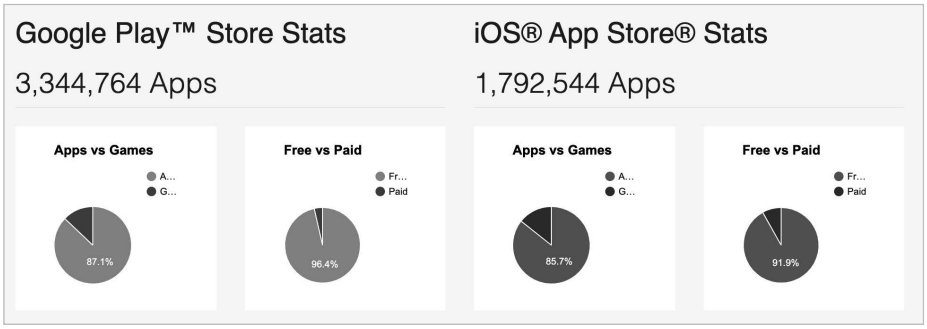

Abb. 2–4 *Anteil Kauf-Apps (Paid) in Google Play Store und iOS App Store [URL: 42matters]*

Außerdem sollten wir in der Wirtschaftlichkeitsrechnung berücksichtigen, dass die Store-Betreiber relative hohe Verkaufsprovisionen einbehalten. Hinzu kommt, dass laut Bitkom in 2018 nur ca. 5 % des gesamten Umsatzes im deutschen App-Markt durch direkten Verkauf von Apps erwirtschaftet wurde [URL: Bitkom].

Apps, für die dieses Modell nutzbar ist, sollten über einzigartige, schwer zu kopierende Funktionen verfügen. Eine andere Möglichkeit, sich vom Wettbewerb

abzusetzen, ist eine bessere Benutzbarkeit. Oft bietet gerade die Benutzbarkeit dem Anwender einen hohen Mehrwert, für den er zu zahlen bereit ist.

In einer Umfrage von Perfecto Mobile haben Nutzer die sie am meisten störenden Auffälligkeiten oder Fehler nach ihrer Art bewertet. Auch für uns war es auf den ersten Blick überraschend, dass Benutzbarkeit und Performanz noch vor Funktionalität genannt wurden. Je mehr wir darüber nachdachten, desto mehr wurde uns klar, dass es uns selbst ebenso geht. Intuitive Benutzung oder keine Wartezeiten sind häufig wichtiger als Funktionalität und anderes (vgl. Abb. 2–5).

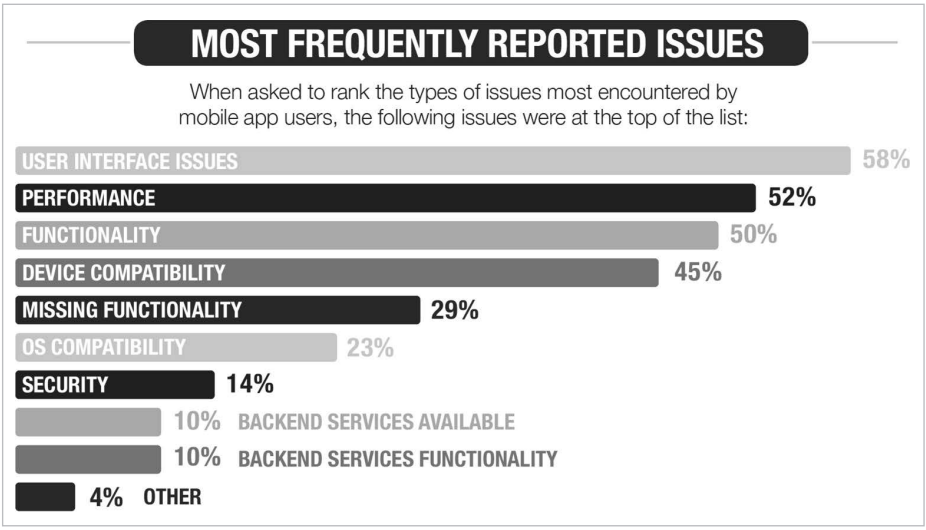

Abb. 2–5 *Häufigkeit von Fehlerarten, die durch Anwender berichtet werden [URL: Perfecto 2014]*

Für die Testplanung ist zudem zu berücksichtigen, dass Budgets geplant und Zahlungswege bereitgestellt werden müssen, die es ermöglichen, die App nach der Veröffentlichung aus dem jeweiligen Store zu kaufen. Durch den Kauf können wir prüfen, ob die veröffentlichte Version der letzten getesteten Version entspricht.

Freemium-Apps

Bei Freemium-Apps stehen den Nutzern Basisfunktionalitäten kostenfrei zur Verfügung. Für zusätzliche Funktionen muss der Nutzer bezahlen. Wie die Bezahlung durch den Nutzer erfolgt, kann auf vielfältige Weise realisiert werden. Dies reicht von In-App-Käufen über Zusatzpakete, die in den Stores für die jeweilige Plattform zum Kauf angeboten werden, bis hin zu Abo-Modellen, bei denen der Nutzer eine monatliche Gebühr bezahlt.

Eine andere Variante der Freemium-Apps erlaubt die kostenfreie Nutzung der App für eine gewisse Zeit. Für die weitere Nutzung muss bezahlt werden.

Das Modell der Freemium-Apps ist sehr weit verbreitet. Es findet in den verschiedensten Bereichen Anwendung. Bekannte Beispiele sind Spiele, die kostenlos gespielt werden können, aber den Nutzern die Möglichkeit bieten, zusätzliche Level, besondere Ausrüstung und Ähnliches käuflich zu erwerben. Ein anderes Beispiel sind Nachrichtenseiten, bei denen gewisse Artikel kostenfrei gelesen werden können, andere Inhalte jedoch zahlenden Kunden vorbehalten sind.

Die Verbreitung und wirtschaftliche Bedeutung dieses Modells zeigt sich auch in dem bereits oben aufgeführten Bitkom-Bericht [URL: Bitcom]. Laut diesem wurden in 2018 77 % des Umsatzes im deutschen App-Markt über kostenpflichtige Angebote innerhalb der Apps erzielt. Allerdings beinhalten diese 77 % auch die später noch aufgeführten transaktionsbasierten Apps.

Im Test darf nicht vergessen werden, die zahlungspflichtigen Erweiterungen zu testen. Diese müssen funktionsfähig verfügbar sein, unabhängig davon, ob sie in die laufende App integriert werden oder ob die App nach dem Kauf erneut installiert wird, z. B. weil der Nutzer sein Smartphone durch ein neues Gerät ersetzt. Dadurch erhöht sich der Testaufwand stark!

Transaktionsbasierte Apps

Diese Apps sind in der Regel kostenfrei nutzbar. Kosten entstehen erst, wenn Transaktionen ausgeführt werden. Dabei muss der Anwender pro Transaktion eine Gebühr bezahlen. Diese Gebühr kann dabei als Pauschalbetrag oder als prozentualer Anteil am Transaktionsvolumen realisiert sein.

Dieses Modell wird eher selten genutzt, da es nur für Apps nutzbar ist, in denen Transaktionen einen wichtigen Teil der Funktionalität darstellen. Ein Schwerpunkt der Nutzung dieses Modell liegt bei Finanz-Apps wie z. B. digitalen Geldbörsen (Wallet), bei denen der Anwender für jede Übertragung von Geld auf ein anderes Konto bezahlt. Weitere Beispiele sind Apps zum Handel von Aktien, bei denen für jeden Kauf oder Verkauf eine Gebühr zu bezahlen ist.

Auch bei Nachrichten-Apps kommt dieses Modell manchmal zum Einsatz, häufig in Kombination mit einem Abo-basierten Modell. Dabei können Nutzer, die kein Abo haben, einzelne Artikel käuflich erwerben, die ansonsten nur Nutzern mit Abo vorbehalten sind.

Werbefinanzierte Apps

Werbung innerhalb von Apps anzuzeigen ist eine sehr weit verbreitete Methode, mit deren Hilfe App-Anbieter Umsatz generieren. Sie ist in fast allen Arten von Apps zu finden.

Die Einbindung von Werbung ist relativ einfach umzusetzen, da sowohl die Plattformanbieter als auch Drittanbieter entsprechende Programmbibliotheken zur Verfügung stellen. Diese müssen durch den App-Anbieter lediglich in die App eingebunden werden. Die Werbeinhalte werden dabei von den Anbietern der Biblio-

theken bereitgestellt, sodass sich der App-Betreiber nicht einmal um Werbepartner kümmern muss. Auch die Ausschüttung des erzielten Umsatzes erfolgt über diese Anbieter. Somit handelt es sich um eine gute Möglichkeit für einen App-Anbieter, ohne großen Aufwand Umsatz zu erzielen (vgl. Abb. 2–6).

Abb. 2–6 *Podcast-App »Overcast« – werbefinanziert*

Die größte Herausforderung besteht darin, die Werbung so in das User Interface zu integrieren, dass die Werbung durch die Nutzer wahrgenommen wird, ohne die Nutzbarkeit stark zu beeinträchtigen. Wird die Nutzbarkeit der App durch die Werbung zu stark eingeschränkt, z. B. indem die Werbung wesentliche Teile der App überlagert, kann dies dazu führen, dass die App nicht genutzt wird. Wird die App nicht genutzt, wird auch keine Werbung angezeigt und somit kein Umsatz über angezeigte Werbung generiert. Dies ist auch der Grund, warum in der Regel der mit dieser Art der Finanzierung erzielbare Umsatz mit der Dauer der Nutzersessions steigt. Denn je länger die App genutzt wird, desto mehr Werbung wird dem Anwender gezeigt.

Beim Test von Apps, die dieses Geschäftsmodell nutzen, sollten wir darauf achten, dass bereits im Test echte Werbung aus den gleichen Quellen wie später in der produktiven Nutzung bezogen wird. Wenn im Test nur simulierte Quellen genutzt werden, besteht das Risiko, dass Probleme mit der Schnittstelle zur produktiven Quelle nicht gefunden werden.

Häufig werden Werbefinanzierung und Freemium kombiniert. Bei dieser Kombination kann eine werbefinanzierte App durch Bezahlung in eine werbefreie Version gewandelt werden. Im Test ist daher darauf zu achten, dass in der werbefreien Version wirklich alle Werbung entfernt wurde.

Gratis- und Unternehmens-Apps

Das Feld der Gratis- und Unternehmens-Apps ist sehr groß. Es lässt sich dabei in drei Hauptgruppen einteilen. Zum einen in die Gruppe der Apps, die für die eigenen Mitarbeiter und Partner entwickelt werden, damit diese geschäftliche Aufgaben direkt von ihrem Smartphone oder Tablet aus erledigen können (vgl. Praxisbeispiel 2–2).

Praxisbeispiel 2–2: Unternehmens-App

Ein Abrechnungsdienstleister im Energiesektor hat eine Android-App entwickeln lassen, mit der die Ableser vor Ort beim Kunden die Verbrauchswerte für Strom, Heizung und Wasser erfassen konnten. Jeder Ableser erhielt ein entsprechendes Android-Gerät. Erfasste Daten wurden bei bestehender Netzwerkverbindung direkt übertragen. Wenn während der Erfassung keine Netzwerkverbindung bestand, wurde die Übertragung verzögert, bis das Gerät das nächste Mal eine Netzwerkverbindung aufbauen konnte.

Durch diese App wurde die Arbeit der Ableser vor Ort stark vereinfacht, da sie das Formular für den jeweiligen Kunden nicht mehr aus einer Vielzahl von Papierformularen heraussuchen mussten. Zudem wurden auch nur die Eingabefelder angezeigt, die abzulesen waren. Außerdem wurde angezeigt, wo die Messinstrumente, die abgelesen werden mussten, installiert waren, was die Zeiten zur Suche der Instrumente reduzierte.

Ein weiterer Vorteil war, dass keine Formulare verloren gingen, was früher bei der Nutzung von Papierformularen gelegentlich vorgekommen war.

Zusätzlich konnten Einsparungen erzielt werden, da der Aufwand zum Drucken und Verteilen der Formulare an die Ableser entfiel. Auch war keine manuelle Übertragung der Daten von den Papierformularen in das Abrechnungssystem mehr notwendig. Damit wurde die Anzahl der Übertragungsfehler reduziert, was zu weniger falschen Abrechnungen und damit verbundenen Kundenbeschwerden führte. Neben den Einsparungen wurde somit zusätzlich die Kundenzufriedenheit gesteigert.

Die zweite Gruppe richtet sich an Endverbraucher. Unternehmen stellen ihren Kunden Apps kostenfrei zur Verfügung, mit denen die Angebote des Unternehmens genutzt werden können. Umsatz und Kosteneinsparungen werden dann durch die genutzten Angebote realisiert (vgl. Praxisbeispiel 2–3).

Praxisbeispiel 2–3: Onlinebanking-App

Im Rahmen des Projektes wurde eine Onlinebanking-App von Grund auf neu entwickelt. Wir waren von der ersten Stunde an dabei und teilweise werden heute noch (zehn Jahre nach der ersten Veröffentlichung im App Store) Regressionstests von uns durchgeführt.

Die App wurde nativ für iOS und Android entwickelt. Die gesamte Funktionalität der App wurde vollständig neu implementiert. Dabei wurde die bestehende Webanwendung als Vorbild für Abläufe, Masken usw. herangezogen.

Die App selbst ist eine typische Onlinebanking-App, wie sie mehr oder weniger jeder heutzutage auf seinem Smartphone hat, wenn vielleicht aus Sicherheitsgründen nicht die seiner Hausbank, dann aber vielleicht Paypal oder Ähnliches. Der Nutzer kann dort seine Kundendaten anpassen, also beispielsweise seine Adresse ändern. Als Kernfunktionalität kann er Überweisungen durchführen und sich seinen Kontostand anzeigen lassen.

Alle Kosten für die Entwicklung und den Test sowie die fortlaufenden Wartungskosten werden vollständig von der Bank getragen. Die App dient der Bank als Instrument zur Kundenbindung und zum Generieren von Neugeschäft. Der Vorteil für den Kunden ist, dass er mit der App ausgewählte Bankgeschäfte immer und überall durchführen kann. Für Kunden steht die App kostenlos zur Verfügung.

Der Umsatz für die Bank entsteht aus den Bankgeschäften, die die Kunden mithilfe der App eigenständig ausführen können, z. B. in Form der üblichen Überweisungsgebühren. Einsparungen kann die Bank erzielen, da kein Mitarbeiter Überweisungsformulare in das System übertragen muss.

Die dritte Gruppe sind Apps, die von Unternehmen, Organisationen oder Privatpersonen, ohne die Absicht Umsatz zu generieren, veröffentlicht werden. Dies ist vergleichbar zu Freeware und Open-Source-Software, wie wir sie auch im PC-Bereich kennen. Als sehr bekanntes Beispiel möchten wir hier den Browser Mozilla Firefox aufführen. Dieser ist nicht nur für PC, sondern auch für Android und iOS verfügbar. Ein weiteres Beispiel ist die von der Bundesregierung bzw. dem Robert-Koch-Institut bereitgestellte Corona-Warn-App.

Bei diesen freien und Unternehmens-Apps ist genau wie bei den zuvor aufgeführten Geschäftsmodellen im Test darauf zu achten, dass nicht nur die App an sich, sondern auch das Zusammenspiel mit dem Backend berücksichtigt wird.

Grundsätzlich darf auch bei diesem Geschäftsmodell der Test nicht vernachlässigt werden. Denn nur weil der Anwender nichts für die App bezahlt, bedeutet dies nicht, dass er Fehler leichter verzeiht. Der Schaden für das Image und den Umsatz des Unternehmens kann bei schlechter Qualität der App höher sein als der über die App generierte Wert.

6 Automatisierung der Testausführung

In diesem abschließenden Kapitel werden Ansätze zur automatisierten Testausführung für mobile Apps vorgestellt. Außerdem diskutieren wir die Stärken und Schwächen der verschiedenen Methoden zur Testautomatisierung von mobilen Apps und geben Hinweise, wie ein geeignetes Werkzeug für die Testautomatisierung ausgewählt werden kann. Zum Schluss weisen wir darauf hin, was bei der Einrichtung eines Testautomatisierungslabors berücksichtigt werden sollte, und bewerten Testautomatisierung im Kontext von mobilen Apps.

Die Automatisierung weiterer Aktivitäten im Testprozess wird in diesem Kapitel nicht behandelt.

Schlüsselbegriffe aus dem englischen Syllabus:
API (Application programming interface), User-Agent-basiertes Testen (user-agent-based testing), gerätebasiertes Testen (device-based testing), Testbericht (test report)

Weitere Schlüsselbegriffe in diesem Kapitel:
OCR

6.1 Automatisierungsansätze

Das Thema Testautomatisierung wird vom ISTQB® ebenfalls als wichtig genug erachtet, um dafür einen eigenen Syllabus, den »Advanced Level Test Automation Engineer, zu erarbeiten [URL: ISTQB Downloads; GTB-Lehrplan 2019b]. Das passende Lehrbuch »Basiswissen Testautomatisierung« von Manfred Baumgartner, Stefan Gwihs, Richard Seidl, Thomas Steirer und Marc-Florian Wendland [Baumgartner et al. 2021] befand sich während des Schreibens dieses Buches in der Drucklegung. Daher werden wir in diesem Kapitel lediglich auf die wichtigsten sowie die mobilspezifischen Punkte eingehen.

Im Bereich der mobilen Apps gibt es verschiedene Ansätze zur Automatisierung der Testausführung. Diese Ansätze unterscheiden sich dabei nicht von den Ansätzen, die auch für nicht mobile Apps zum Einsatz kommen.

Bei mobilen Apps kommt jedoch eine Möglichkeit hinzu, die bei nicht mobilen Apps keine Rolle spielt. Neben der automatisierten Testausführung auf der Zielplattform, auf der die Applikation produktiv genutzt werden soll, können mobile Web-Apps auch in Desktop-Browsern getestet werden. Dabei sprechen wir von einer User-Agent-basierten Testautomatisierung.

6.1.1 User-Agent-basierte Testautomatisierung

Dieser Ansatz nutzt Desktop-Browser für den automatisierten Test von mobilen Web-Apps. Er sollte nicht mit anderen Ansätzen verwechselt werden, bei denen es erforderlich sein kann, einen Softwareagenten auf den Mobilgeräten zu installieren.

Die Bezeichnung »User-Agent-basiert« ist darauf zurückzuführen, dass Browser sich gegenüber Servern ausweisen. Dazu schicken sie in ihren Anfragen an die Server den sogenannten »User Agent Identifier String« mit. In diesem steht, von welchem Browser auf welcher Plattform die Anfrage kommt. Dieser String kann durch die Server ausgewertet werden, um dem Anwender für seine Plattform angepasste Inhalte zu liefern.

Moderne Browser erlauben es, einen gefälschten User Agent Identifier String zu schicken. Somit weist sich der Browser als ein anderer Browser bzw. als auf einer anderen Plattform laufender Browser aus. Als Ergebnis erhält der Browser von den Servern den Inhalt geliefert, wie er für das Gerät bzw. den Browser bestimmt ist, als der er sich identifiziert. Dies kann mit einem gefälschten Ausweis verglichen werden, der Zutritt zu Bereichen ermöglicht, die ohne diesen Ausweis nicht zugänglich sind.

Bei diesem Ansatz ist es möglich, Kombinationen von Plattform und Browser festzulegen, die bei der Ausführung auf Mobilgeräten nicht machbar sind. Beispielsweise ist in Abbildung 6–1 zu sehen, dass sich Apples Safari auf MacOS als Microsoft Edge auf einem iPhone ausweisen kann. Daher muss bei der Auswahl darauf geachtet werden, nur Kombinationen zu wählen, die auch bei echten Geräten vorkommen können. Ansonsten werden automatisierte Tests auf Geräte- und Browserkombinationen ausgeführt, die bei echten Nutzern nicht vorkommen können, und dadurch Ressourcen verschwendet.

Einen gefälschten User Agent Identifier String können wir auch für manuelle Tests nutzen. Alle gängigen Browser erlauben es, innerhalb der enthaltenen Entwicklerwerkzeuge festzulegen, als was sich der Browser ausweisen soll. Insbesondere für den Test von Responsive und Adaptive Web Apps ist diese Möglichkeit sehr gut geeignet, da auf Knopfdruck zwischen den unterschiedlichen Darstellungsvarianten gewechselt werden kann (vgl. Abb. 6–1).

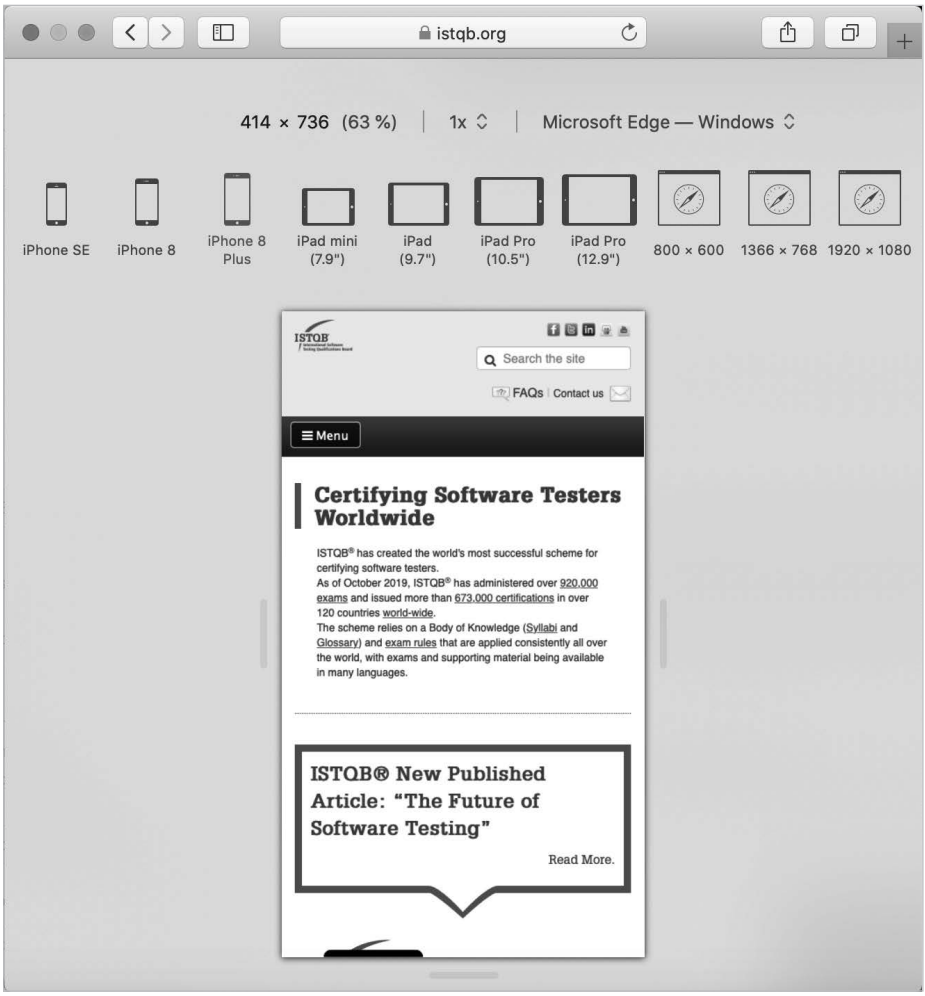

Abb. 6–1 *Apple Safari, der sich als Microsoft Edge auf einem iPhone 8 Plus ausweist.*

Der Vorteil dieses Ansatzes ist, dass vorhandenes Know-how und bekannte Werkzeuge für automatisierte Tests von Webapplikationen genutzt werden können, um mobile Web-Apps automatisiert zu testen. Der Nachteil besteht darin, dass es sich aber immer noch um den tatsächlichen Desktop-Browser handelt und nicht um den mobilen Browser, als den er sich ausgibt. Somit können Fehler, die aufgrund der Rendering-Engine oder JavaScript-Engine des mobilen Browsers auftreten, nicht gefunden werden. Gleichzeitig können Fehler gefunden werden, die auf dem mobilen Browser nicht auftreten. Dieses Risiko sollte dem Team bewusst sein, wenn auf einen User-Agent-basierten Ansatz für die Testautomatisierung gesetzt wird (vgl. Praxisbeispiel 6–1).

Praxisbeispiel 6–1: Auswirkungen des User-Agent-basierten Ansatzes

In einem Projekt, in dem eine mobile Web-App entwickelt wurde, hatten wir den Fall, dass ein Entwickler einen Fix, inkl. Screenshot, wie die App nach dem Fix aussieht, zum Nachtest bereitgestellt hat. Auf dem Screenshot war zu sehen, dass mit Safari auf einem Mac gearbeitet wurde und nur mithilfe der Entwicklertools vorgegeben wurde, dass die Seite mit Chrome auf iPhone 6/7/8 aufgerufen wurde. Wenn wir uns die App aber mit Chrome auf einem echten iPhone 8 angeschaut haben, war der Fehler weiterhin sichtbar.

6.1.2 Gerätebasierte Testautomatisierung

Beim Ansatz der gerätebasierten Testautomatisierung wird die zu testende App auf Mobilgeräten ausgeführt. Als Mobilgeräte können sowohl echte physikalische als auch emulierte bzw. simulierte Mobilgeräte zum Einsatz kommen. Daher können mit diesem Ansatz nicht nur Web-Apps, sondern auch native und hybride Apps getestet werden. Allerdings ist es erforderlich, dass spezielle Werkzeuge und Frameworks zum Einsatz kommen. Diese bieten entsprechende Schnittstellen, mit denen auf die Mobilgeräte zugegriffen werden kann. Manche der Werkzeuge erfordern es, einen speziellen Softwareagenten für die Interaktion mit der App auf den Mobilgeräten zu installieren oder den Anwendungscode zu instrumentieren. Dies ist aber nicht für alle Werkzeuge zum automatisierten Test von mobilen Apps notwendig. Da Querwirkungen zwischen App und Agent oder Auswirkungen der Instrumentierung nicht auszuschließen sind, sollte dies bei der Auswahl eines Werkzeuges und der Testplanung berücksichtigt werden.

Plattformspezifische Werkzeuge werden von den Plattformanbietern angeboten bzw. sind bereits in den jeweiligen SDKs enthalten. Für Android wird durch Google derzeit Espresso [URL: Espresso] und UI Automator [URL: UI Automator] bereitgestellt:

- Espresso ermöglicht lediglich die Interaktion mit einer einzigen App.
- UI Automator hingegen ermöglicht die Interaktion mit mehreren Apps und erlaubt es somit, Anwenderszenarien zu automatisieren, die mehrere Apps beinhalten.

Für iOS wird von Apple XCUITest bereitgestellt. XCUITest ist Bestandteil von XCTest [URL: XCTest]. Dieses Testframework von Apple erlaubt es Entwicklern, Tests für alle Teststufen in einem einzigen Framework zu erstellen. Angefangen bei Unit Tests bis hin zum Test von Benutzerszenarien auf der grafischen Oberfläche der App.

Zusätzlich gibt es Testautomatisierungswerkzeuge von weiteren Anbietern. Häufig lassen sich diese Werkzeuge plattformübergreifend einsetzen. Es gibt immer gute Gründe, warum ein Werkzeug besonders gut für das eine oder das andere Projekt geeignet ist. Wir verzichten daher auf eine Auflistung vieler Hersteller

und ihrer Werkzeuge und beschränken uns in diesem Buch auf ein gängiges und frei erhältliches Werkzeug.

Derzeit ist der Einsatz von Appium [URL: Appium] sehr weit verbreitet. Appium ist ein Open-Source-Projekt, das den automatisierten Test von nativen, hybriden und Web-Apps unter Android, iOS und Windows erlaubt. Es setzt dabei auf das WebDriver-Protokoll auf [URL: WebDriver], wie es z.B. auch von Selenium [URL: Selenium] genutzt wird. Die Verwendung dieses Protokolls macht es sehr einfach, Tests für mobile Apps zu automatisieren, wenn entsprechendes Selenium-Know-how vorhanden ist. Hauptunterschied ist, dass mit den Testskripten eine entsprechende Konfiguration in Form von sogenannten »Desired Capabilities« übergeben werden muss. Diese Konfiguration legt fest, auf welchem Gerät, mit welchem Betriebssystem in welcher Version und welche App getestet werden soll. Die Übergabe von Desired Capabilities ist dabei sehr ähnlich wie beim Einsatz von Selenium Grid [URL: Selenium Grid] (vgl. Praxisbeispiel 6–2).

Praxisbeispiel 6–2: Desired Capabilities für Appium

Beispiel für eine Appium-Konfiguration zum automatisierten Test einer mobilen Web-App auf einem iPhone:

```
public void setUp() throws Exception {
   DesiredCapabilities capabilities = new DesiredCapabilities();
   capabilities.setCapability("automationName", "Appium");
   capabilities.setCapability("platformName", "IOS");
   capabilities.setCapability("platformVersion", "11.3");
   capabilities.setCapability("deviceName", "iPhone 6");
   capabilities.setCapability("browserName", "Safari");
   driver = new RemoteWebDriver(new URL("http://127.0.0.1:4723/wd/hub"),
   capabilities);
}
```

Zur Umsetzung dieser Herangehensweise ist es notwendig, ein entsprechendes Testlabor zu nutzen. Ob dieses selbst lokal aufgebaut wird oder Angebote in der Cloud genutzt werden, muss im Kontext der zu testenden App entschieden werden. Die in Abschnitt 5.4 vorgestellten Vor- und Nachteile für lokale und entfernte Testlabore gelten dabei auch für die Testautomatisierung. Auf automatisierungsspezifische Vor- und Nachteile wird in Abschnitt 6.5 eingegangen.

6.1.3 Herkömmliche Testautomatisierungsansätze

Oben beschriebene Ansätze müssen mit den herkömmlichen Testautomatisierungsansätzen kombiniert werden, die auch bei nicht mobilen Apps zum Einsatz kommen. Verschiedene weitverbreitete Ansätze werden in den folgenden Abschnitten vorgestellt.

Aufnahme und Wiedergabe

Dieser Ansatz ist auch als »Capture & Replay« bzw. »Record & Playback« bekannt. Er basiert darauf, dass ein Testfall Schritt für Schritt durch einen Tester manuell ausgeführt und dabei aufgenommen wird. Durch das Werkzeug werden die einzelnen Schritte protokolliert. Dieses Protokoll kann später genutzt werden, um die Schritte von einer Maschine wiederholen zu lassen. Häufig geschieht diese Protokollierung in der Form, dass aus den manuell ausgeführten Schritten durch das Werkzeug ein Skript generiert wird.

Auch wenn dieser Ansatz sehr verlockend klingt, zeigt die Erfahrung, dass dieser Ansatz meistens nicht sehr nachhaltig ist, da sich eine App in ihrem Lebenszyklus ändert und die Wartung und Pflege der Skripte sehr aufwendig ist. Abhängig vom genutzten Werkzeug kann es sogar notwendig sein, bei jeder Änderung in der App die Testfälle neu aufzunehmen. Das ist ein schier unendliches und teures Unterfangen.

Aktuell ist der Trend zu beobachten, dass Frameworks zur Aufnahme und Wiedergabe mit KI ausgestattet werden, was die Wartung und Pflege vereinfachen oder sogar weitestgehend überflüssig machen soll. Ob dieses Versprechen in der Praxis eingehalten wird, kann derzeit durch uns nicht abschließend beurteilt werden.

Manuelle Skripterstellung

Dieser Ansatz nutzt die Testfälle in der Form von Skripten, die von Hand programmiert werden. Diese Skripte werden ausgeführt, um den jeweiligen Test durchzuführen. Dabei ist zu berücksichtigen, dass die Entwicklung der Testskripte Softwareentwicklung ist. Wie wir Tester wissen, kommt es bei der Entwicklung von Software zu Fehlern. Somit müssen die Testfallskripte selbst getestet werden. Der mit der Entwicklung und dem Test der Skripte verbundene Aufwand sollte nicht unterschätzt werden. Zudem müssen die Skripte gepflegt werden, um Änderungen in der damit getesteten Software zu berücksichtigen. Um die langfristige und nachhaltige Nutzung zu vereinfachen, empfehlen wir, die Regeln und Methoden zur Erstellung von Software auch bei der Erstellung der Testskripte anzuwenden. Testskripte sollten u.a, wie jeder andere Code, einer Versionsverwaltung unterliegen. Dies erleichtert es, festzustellen, welche Version des Testfalls zu welcher Softwareversion passt.

Datengetriebene Testautomatisierung

Dieser Ansatz nutzt Testskripte, die es erlauben, Testein- und -ausgaben aus einer Datenquelle, z. B. einer Excel-Tabelle oder Datenbank, zu übergeben. Der Testablauf innerhalb des Skriptes wird dann mit jedem Datensatz einmal wiederholt. So ist es möglich, mit einem Testskript eine Vielzahl von Testvariationen abzudecken. Damit ist dieser Ansatz insbesondere in den Situationen effizient, in denen Abläufe sich lediglich durch die Ein- und Ausgabewerte unterscheiden.

Schlüsselwort- und verhaltensgetriebene Testautomatisierung

Dieser Ansatz basiert darauf, die geschäftliche von der technischen Logik des Testfalls zu trennen. Die Testfälle werden als Abfolge von Schlüsselworten definiert, die das Nutzer- und/oder Systemverhalten beschreiben. Damit diese Testfälle ausgeführt werden können, ist es notwendig, im Hintergrund festzulegen, wie pro Schlüsselwort technisch mit dem getesteten System interagiert wird.

Durch die Trennung ergibt sich der große Vorteil, dass die technische Umsetzung für ein Schlüsselwort nur einmal realisiert werden muss, aber in einer Vielzahl von Testfällen genutzt werden kann. Wenn die Schlüsselworte bekannt sind, können auch Tester ohne Programmierkenntnisse die fachliche Logik in automatisch ausführbaren Testfällen abbilden (vgl. Praxisbeispiel 6–3).

In den letzten Jahren hat insbesondere im Umfeld von BDD (Behavior Driven Development) und ATDD (Acceptance Test Driven Development) dieses Vorgehen Verbreitung gefunden. Dabei werden die Anforderungen an die zu entwickelnde Software direkt mit den Schlüsselworten definiert. Bei der Entwicklung wird dann die technische Logik parallel zur Anwendung selbst erstellt.

Unserer Erfahrung nach kann eine nachträgliche Definition von Schlüsselworten, abhängig von der zu testenden App, sehr aufwendig sein. Eine solche Architekturentscheidung zur Testautomatisierung sollte demzufolge möglichst früh im Projekt getroffen werden.

Praxisbeispiel 6–3: Vereinfachtes Beispiel für ein Automatisierungsskript (Hinweis: Code hat keine Produktionsqualität.)

Unten stehender Selenium-Beispielcode für den Test einer Webanwendung trennt die geschäftliche Nutzeraktivität, beschrieben als Schlüsselwort, von der technischen Umsetzung der einzelnen Automatisierungsschritte. Der Testfall besteht aus zwei Klassen.

Klasse 1 mit dem Namen »Login« enthält die geschäftliche Logik für die Testfälle. Im Beispiel ist lediglich ein Testfall in der Klasse. In diesem wird geprüft, dass sich der Nutzer mit gültigem Usernamen und Passwort am System anmelden kann. Neben dem Keyword »Login« sind Username und Passwort als Parameter enthalten.

```
class Login extends BaseTest {
   @Test
   void testlogin() throws Exception {
      login("Username", "Passwort")
   }
}
```

→

Klasse 2 hat den Namen »BaseTest«. In dieser Klasse gibt es u.a. die unten stehende Methode »login«. Sie enthält alle technischen Schritte zur Anmeldung des Nutzers und nutzt die übergebenen Parameter:

```
protected void login(String USER, String PW) throws Exception {
   getDriver().get(Config.URL);
   WebElement userTextBox =
   getWaitr().until(ExpectedConditions.visibilityOf(driver.findElement
   (By.id("user_login"))));
   assertTrue(userTextBox.isEnabled(), "Field Username not enabled.");
   userTextBox.sendKeys(USER);
   assertEquals(USER, userTextBox.getAttribute("value"), "Expected text
   not entered");
   WebElement pwdBox = driver.findElement(By.id("user_pass"));
   assertTrue(pwdBox.isEnabled(), "PWDfield is not enabled.");
   pwdBox.sendKeys(PW);
   assertEquals(PW, pwdBox.getAttribute("value"), "Expected text not
   entered");
   WebElement submit = driver.findElement(By.id("submit"));
   assertTrue(submit.isEnabled(),"Submitbutton not enabled");
   submit.click();
}
```

6.2 Testautomatisierungsframeworks

Testautomatisierungswerkzeuge liegen in der Regel als sogenannte Rahmenwerke oder Frameworks vor. Sie bilden den Rahmen, in dem die konkrete Testautomatisierungslösung umgesetzt wird. Sie können dabei sehr rudimentär oder sehr komplex sein, und selbst wiederum andere Frameworks enthalten.

Ein Testautomatisierungsframework sollte folgende wichtige Fähigkeiten aufweisen:

- **Objektidentifizierung**
  Zur Erkennung der Elemente der App, mit denen im Testfall interagiert werden soll. Beispielsweise können so Buttons, Eingabefelder und Ausgabefelder erkannt werden. Da die Objekterkennung einer der kritischsten Punkte bei der Testautomatisierung ist, kann es sinnvoll sein, schon bei der Entwicklung die Objekterkennung durch das Testautomatisierungswerkzeug zu berücksichtigen, z. B. indem eindeutige IDs für Objekte vergeben werden.
- **Objektoperationen**
  Um mit den erkannten Objekten zu interagieren, z. B. Antippen von Buttons, Eingeben von Daten in Eingabefelder oder Lesen von Ausgaben.