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).

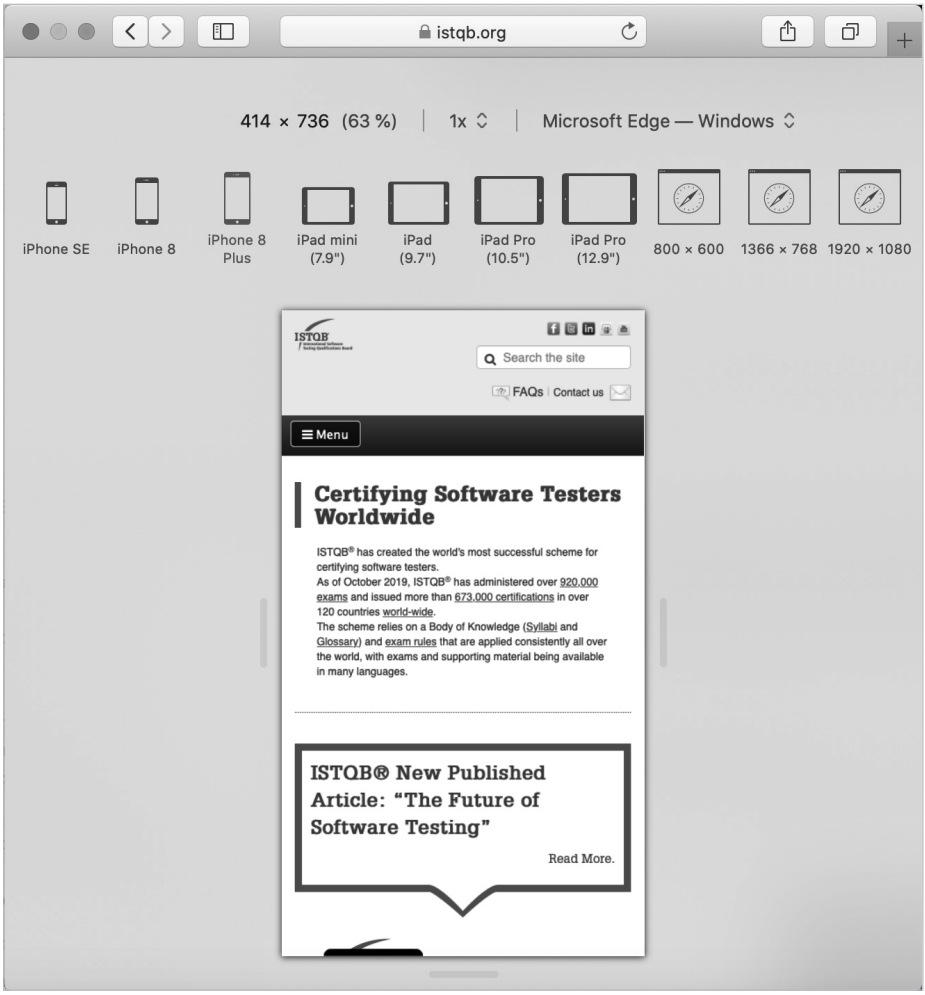

Abb. 6–1 *Apple Safari, der sich als Microsoft Edge auf einem iPhone 8 Plus ausweist.*

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.