A4Q
Selenium Tester Foundation
Lehrplan

Freigegebene
Version 1.3 / 2019

Alliance for Qualification

Version 2018

Änderungshistorie

Version	**Datum**	**Bemerkung**
Version 1.0	Aug 2018	1. freigegebene Version
Version 1.1 / 1.2	Nov 2018	Minimal Korrektion
Version 1.3	Mai 2019	Minimal Korrektion

Inhaltsverzeichnis

0 Einführung

0.1 Zweck dieses Lehrplans

Dieser Lehrplan legt den geschäftlichen Nutzen, Lernziele und Konzepte dar, die dem Training und der Zertifizierung des Selenium Tester Foundation zugrunde liegen.

0.2 Prüfbare Lernziele und kognitive Stufen des Wissens

Lernziele unterstützen den geschäftlichen Nutzen und werden verwendet, um die Selenium Tester Foundation Prüfungen zu erstellen.
Im Allgemeinen sind alle Inhalte dieses Lehrplans auf K1-Stufe prüfbar, außer der Einführung und der Anhänge. Das bedeutet, vom Prüfling kann gefordert werden, einen Schlüsselbegriff oder ein Konzept aus jedem der vier Kapitel erkennen, sich daran erinnern oder wiedergeben zu können. Die Wissensstufen der spezifischen Lernziele werden am Beginn jedes Kapitels genannt und sind wie folgt klassifiziert:

- K1: erinnern
- K2: verstehen
- K3: anwenden
- K4: analysieren

Die Definitionen aller Begriffe, die als Schlüsselbegriffe unterhalb der Kapitelüberschrift aufgelistet sind, sollen erinnert werden (K1), auch wenn sie nicht ausdrücklich in den Lernzielen erwähnt werden.

0.3 Die Selenium Tester Foundation Prüfung

Die Selenium Tester Foundation Prüfung basiert auf diesem Lehrplan und dem akkreditierten A4Q Selenium Tester Foundation Trainingskurs. Die Beantwortung der Prüfungsfragen können die Nutzung von Materialien auf Grundlage von mehr als einem Abschnitt dieses Lehrplans und/oder des Selenium Tester Foundation Trainingskurses erfordern. Alle Abschnitte dieses Lehrplans und des Selenium Tester Foundation Trainingskurses sind prüfungsrelevant, außer der Einführung und der Anhänge.
Standards, Bücher und ISTQB®-Lehrpläne sind als Referenzen genannt, aber deren Inhalt ist nicht über das hinaus prüfungsrelevant, was in diesem Lehrplan selbst aus diesen Standards, Büchern oder ISTQB®-Lehrplänen in zusammengefasster Form enthalten ist.

Die Prüfung besteht aus 40 Multiple-Choice-Fragen. Jede richtige Antwort erhält einen Punkt. Zum Bestehen der Prüfung müssen mindestens 65% der Fragen (also 26 Fragen) korrekt beantwortet werden. Die Prüfungszeit beträgt 60 Minuten. Wenn die Muttersprache des Bewerbers nicht die Prüfungssprache ist, kann dem Bewerber eine zusätzliche Zeit von 25% (15 Minuten) gewährt werden.

Prüfungen dürfen ausschließlich nach Absolvierung eines akkreditierten A4Q Selenium Tester Foundation Trainingskurses abgelegt werden, da die Bewertung der Kompetenz des Bewerbers in den Übungen durch den Kursleiter Teil der Zertifizierung ist.

0.4 Akkreditierung

Der A4Q Selenium Tester Foundation Trainingskurs ist der einzige akkreditierte Trainingskurs.

0.5 Detaillierungsgrad

Der Detaillierungsgrad dieses Lehrplans erlaubt international konsistente Prüfungen. Um dieses Ziel zu erreichen, enthält der Lehrplan Folgendes:

- Allgemeine Lernziele, die die Intention des A4Q Selenium Tester Foundation Level beschreiben
- Eine Liste von Begriffen, die die Studierenden kennen müssen
- Lernziele für jeden Wissensbereich, die das zu erzielende kognitive Lernergebnis beschreiben
- Eine Beschreibung der wichtigen Konzepte sowie der zugehörigen Referenzen auf Quellen wie allgemein anerkannte Fachliteratur oder Standards/Normen

Der Inhalt des Lehrplans ist keine Beschreibung des gesamten Wissensgebiets „Testautomatisierung mit Selenium“. Er spiegelt den Detaillierungsgrad wider, der in Foundation-Level-Trainingskursen abgedeckt wird. Er konzentriert sich auf Testkonzepte und -verfahren, die auf alle Softwareprojekte angewendet werden können, auch auf agile Projekte. Dieser Lehrplan beinhaltet keine spezifischen Lernziele, die sich auf spezielle Softwareentwicklungslebenszyklen oder -methoden beziehen, sondern befasst sich damit, wie diese Konzepte in verschiedenen Softwareentwicklungslebenszyklen angewendet werden.

0.6 Wie dieser Lehrplan aufgebaut ist

Es gibt vier Kapitel mit prüfungsrelevantem Inhalt. Die Hauptüberschrift für jedes Kapitel legt die Zeit fest, die für das Kapitel vorgesehen ist. Für die Unterkapitel ist keine Zeitangabe vorhanden. Für die A4Q Selenium Tester Foundation Trainingskurse erfordert dieser Lehrplan mindestens 16,75 Std. Unterricht, der sich wie folgt auf die vier Kapitel aufteilt:

Kapitel 1: Grundlagen der Testautomatisierung - 105 Minuten
Kapitel 2: Internettechnologien für die Testautomatisierung von Webanwendungen - 195 Minuten
Kapitel 3: Verwendung des Selenium WebDrivers - 495 Minuten
Kapitel 4: Erstellung wartbarer Testskripte - 225 Minuten

0.7 Geschäftlicher Nutzen

SF-BO-1	Korrekte Anwendung von Testautomatisierungsgrundsätzen zur Erstellung einer wartbaren Testautomatisierungslösung
SF-BO-2	In der Lage sein, die richtigen Testautomatisierungswerkzeuge auszuwählen und zu implementieren
SF-BO-3	In der Lage sein, Selenium WebDriver-Skripte zu implementieren, die funktionale Webanwendungstests ausführen
SF-BO-4	In der Lage sein, wartbare Testskripte zu implementieren

0.8 Abkürzungen

AKA:	Also Known As
API:	Application Programming Interface
CERN:	European Council for Nuclear Research (französisch)
CI:	Continuous Integration (deutsch „kontinuierliche Integration)
CSS:	Cascading Style Sheets (deutsch „Kaskadenstilentwurf”)
DOM:	Document Object Model (deutsch „Dokumenten-Objekt-Modell“)
GUI:	Graphical User Interface (deutsch „grafische Benutzerschnittstelle”)
HTTP:	HyperText Transfer Protocol
ISTQB®:	International Software Testing Qualifications Board
KDT:	Keyword Driven Testing (deutsch „schlüsselwortgetriebener Test”)
REST:	Representational State Transfer
ROI:	Return on Investment (deutsch „Kapitalrendite”)
SDLC:	Software Development Life Cycle (deutsch „Softwareentwicklungslebenszyklus”)
SOAP:	Simple Object Access Protocol
SUT:	System Under Test (Testobjekt)
TAA:	Test Automation Architecture (deutsch „Testautomatisierungsarchitektur”)
TAE:	Test Automation Engineer (deutsch „Testautomatisierungsentwickler”)
TAS:	Test Automation Solution (deutsch „Testautomatisierungslösung”)
TCP:	Transmission Control Protocol (deutsch „Übertragungssteuerungsprotokoll”)
UI:	User Interface (deutsch „Benutzerschnittstelle”)
W3C:	World Wide Web Consortium (deutsch „W3-Konsortium”)

Kapitel 1 – Grundlagen der Testautomatisierung

Schlüsselbegriffe

Architektur, Erfassung/Wiedergabe, Komparator, exploratives Testen, Fehler Attacke, Framework, Hook, Pestizid Paradoxon, technische Schuld, Testbarkeit, Test-Harness, Test-Orakel, Software

Lernziele für Grundlagen der Testautomatisierung

STF-1.1	(K2) Die Ziele, Vorteile, Nachteile und Einschränkungen der Testautomatisierung erklären können
STF-1.2	(K2) Die Beziehung zwischen manuellen und automatisierten Tests verstehen
STF-1.3	(K2) Die technischen Erfolgsfaktoren eines Testautomatisierungsprojekts identifizieren können
STF-1.4	(K2) Die Risiken und Nutzen der Verwendung des Selenium WebDrivers verstehen
STF-1.5	(K2) Die Stellung des Selenium WebDrivers in der TAA (Testautomatisierungsarchitektur) erklären können
STF-1.6	(K2) Den Sinn und Zweck des Sammelns von Metriken in der Automatisierung erklären können
STF-1.7	(K2) Die Ziele der Verwendung der Selenium Toolfamilie (WebDriver, Selenium Server, Selenium Grid) verstehen und vergleichen können

1.1 Überblick über die Testautomatisierung

Testautomatisierung ist vielerlei. In diesem Lehrplan wird die Behandlung von Automatisierung auf die Definition von Testautomatisierung als die automatische Ausführung von funktionalen Tests eingeschränkt, die zumindest in gewisser Weise so entworfen sind, dass sie einen Menschen simulieren, der manuelle Tests ausführt. Tatsächlich gibt es viele verschiedene Definitionen (siehe ISTQB® Certified Tester Advanced Level - Test Automation Engineer); dies ist die für diesen Lehrplan am besten geeignete Definition.

Während die Testausführung weitgehend automatisiert ist, werden Testanalyse, Testentwurf und Testrealisierung normalerweise noch immer manuell durchgeführt. Die Erstellung und Bereitstellung der Daten, die während des Tests verwendet werden, kann teilweise automatisiert sein, wird jedoch oft manuell durchgeführt. Die Bewertung, ob ein Test bestanden oder nicht bestanden wurde, kann – muss aber nicht immer - Teil der Automatisierung sein (über einen Komparator, der in die Automatisierung eingebaut ist).

Automatisierung erfordert den Entwurf, die Erstellung und die Wartung von verschiedenen Testmitteln, einschließlich der Umgebung, in der die Tests ausgeführt werden, der verwendeten Werkzeuge, der Code-Bibliotheken, die Funktionen bereitstellen, der Testskripte und Testrahmen,

sowie der Protokollierungs- und Berichtsstrukturen zur Bewertung der Testergebnisse. Abhängig von den verwendeten Werkzeugen kann die Überwachung und Steuerung der Testausführung eine Kombination aus manuellen und automatisierten Prozessen sein.

Es gibt viele mögliche Ziele, die mit der Automatisierung funktionaler Tests erreicht werden sollen. Dazu gehören:

- Verbesserung der Effizienz des Testens durch eine Reduktion der Kosten für die einzelnen Testläufe
- Mehr und andere Gegenstände testen, die möglicherweise nicht manuell getestet werden könnten
- Reduktion des Zeitaufwands für die Ausführung der Tests
- Es können mehr Tests früher im SDLC vorangetrieben werden, um Fehler im Code früher aufzudecken und zu entfernen (englisch „shift left")
- Erhöhung der Häufigkeit, mit der Tests ausgeführt werden können

Wie bei allen Technologien gibt es auch beim Einsatz der Testautomatisierung Vor- und Nachteile.

Nicht alle Vorteile lassen sich in allen Projekten erzielen, und nicht alle Nachteile treten in allen Projekten auf. Durch die sorgfältige Beachtung der Details und die Verwendung guter technischer Prozesse ist es möglich, bessere Ergebnisse zu erzielen bzw. die schlechten Ergebnisse, die sich aus einem Automatisierungsprojekt ergeben können, zu minimieren. Denn eines ist klar: Eine Organisation, die rein zufällig nebenbei ein erfolgreiches Automatisierungsprojekt aufgebaut hat, gibt es nicht.

Die möglichen Vorteile von Automatisierung sind u.a.:

- Das Ausführen automatisierter Tests ist möglicherweise effizienter als die manuelle Testausführung
- Ausführen bestimmter Tests, die nicht oder nur sehr schwer manuell durchgeführt werden können (z. B. Zuverlässigkeits- oder Effizienztests)
- Reduzierung der für die Testausführung benötigten Zeit, sodass mehr Tests pro Build ausgeführt werden können
- Erhöhung der Häufigkeit, mit der einige Tests ausgeführt werden können
- Entlasten von manuellen Testern, um diese für interessantere und komplexere manuelle Tests (z. B. explorative Tests) verfügbar zu machen
- Reduzierung von Fehlhandlungen durch gelangweilte oder abgelenkte manuelle Tester, insbesondere bei der Wiederholung von Regressionstests
- Ausführung von Tests früher im Prozess (z.B. bei der automatischen Ausführung von Unit-, Komponenten- und Integrationstests als Bestandteile des Continuous-Build), um schneller Rückmeldungen zur Systemqualität zu erhalten und Fehler früher aus der Software entfernen zu können
- Ausführen von Tests außerhalb der normalen Geschäftszeiten

- Erhöhung des Vertrauens in den Software-Build

Zu den Nachteilen der Automatisierung können gehören:

- Höhere Kosten (einschließlich hoher Anlaufkosten)
- Verzögerungen, Kosten und Fehlhandlungen in Zusammenhang mit dem Erlernen neuer Technologien durch die Tester
- Im schlimmsten Fall kann die Komplexität überwältigend werden
- Inakzeptable Steigerung des Umfangs der automatisierten Tests, der möglicherweise den Umfang des zu testenden Systems (SUT) überschreitet
- Im Testteam werden Softwareentwicklungsfähigkeiten benötigt, bzw. die Bereitstellung dieser Fähigkeiten für das Testteam
- Erheblicher Wartungsaufwand für die erforderlichen Werkzeuge, Umgebungen und Test-Assets
- Die technische Schuld wird leicht mehr, besonders wenn zusätzliche Programmierung hinzugefügt wird, um den Kontext und die Angemessenheit der automatisierten Tests zu verbessern; diese technische Schuld ist aber nur schwer zu reduzieren (wie bei jeder Software)
- Automatisierung erfordert alle Prozesse und Disziplinen der Softwareentwicklung
- Die Konzentration auf die Automatisierung kann dazu führen, dass Tester das Risikomanagement für das Projekt aus den Augen verlieren
- Das Pestizid-Paradoxon erhöht sich durch die Verwendung von Automatisierung, da jedes Mal genau der gleiche Test durchgeführt wird
- Falsch positive Ergebnisse treten auf, wenn die Ursachen der Fehlerwirkungen nicht Fehler des SUTs sind, sondern Fehler in der Automatisierung selbst
- Anders als Tester sind Werkzeuge ohne geschickte Programmierung der automatisierten Tests starr und nicht intelligent
- Werkzeuge sind eher eindimensional – damit ist gemeint, dass sie nur nach einem Ergebnis für ein Ereignis suchen, während Menschen herausfinden können, was passiert ist, und spontan bestimmen, ob es korrekt war

Für Automatisierungsprojekte gibt es eine Reihe von Einschränkungen; einige davon (aber nicht alle) lassen sich durch geschickte Programmierung und bewährte technische Praktiken beherrschen. Einige dieser Einschränkungen sind technischer Natur, manche sind vom Management verursacht. Derartige Einschränkungen sind u.a.:

- Unrealistische Erwartungen, durch die das Management die Automatisierung oft in die falsche Richtung lenkt
- Eine kurzfristige Denkweise, die ein Automatisierungsprojekt zerstören kann; nur durch eine langfristige Denkweise kann das Automatisierungsprogramm gelingen
- Ein gewisser Reifegrad der Organisation ist erforderlich, um erfolgreich zu sein; eine Automatisierung basierend auf schlechten Testprozessen führt nur dazu, dass schlechtere Test schneller ausgeführt werden (bzw. manchmal sogar langsamer)

- Einige Tester sind mit dem manuellen Testen vollkommen zufrieden und möchten gar nicht automatisieren
- Automatisierte Testorakel können sich von manuellen Testorakeln unterscheiden, daher ist es erforderlich, sie zu identifizieren
- Nicht alle Tests lassen sich automatisieren, und es sollten auch nicht alle automatisiert werden
- Manuelle Tests sind weiterhin erforderlich (exploratives Testen, bestimmte Fehlerangriffe usw.)
- Testanalyse, Testentwurf und Testrealisierung erfolgen wahrscheinlich weiterhin manuell
- Menschen finden die meisten Fehler; Automatisierung kann letztlich nur das finden, was programmiert wurde, und ist durch das Pestizid-Paradoxon begrenzt
- Aufgrund der großen Anzahl automatisierter Tests, die ausgeführt werden, ohne viele Fehler zu finden, wird ein trügerisches Sicherheitsgefühl vermittelt
- Technische Probleme für das Projekt bei der Verwendung modernster Werkzeuge oder Methoden
- Zusammenarbeit mit der Entwicklung ist erforderlich, wodurch organisatorische Probleme entstehen können

Automatisierung kann gelingen, und sie gelingt oft. Aber dieser Erfolg ist nur das Ergebnis einer sorgfältigen Beachtung von Details, guter Engineering-Praktiken und sehr harter Arbeit über einen längeren Zeitraum.

1.2 Manuelle vs. automatisierte Tests

Frühe Versionen von Testautomatisierungswerkzeugen waren ein katastrophaler Misserfolg. Sie verkauften sich wirklich gut, arbeiteten aber selten wie beworben. Ein Grund dafür ist, dass sie das Testen von Software nicht gut modelliert haben und dass sie die Rolle des Testers im Testprozess nicht verstanden haben.

Einfache Mitschnittwerkzeuge (auch Capture/Replay-Werkzeuge genannt) wurden mit folgenden oder ähnlichen Anweisungen verkauft:

> ***Verbinden Sie das Werkzeug mit dem zu testenden System. Schalten Sie den Schalter ein, um mit der Aufzeichnung zu beginnen. Lassen Sie den Tester den Test ausführen. Schalten Sie nach Abschluss das Werkzeug aus. Das Werkzeug erzeugt ein Skript (in einer Programmiersprache), das genau das tut, was der Tester getan hat. Sie können dieses Skript jedes Mal abspielen, wenn Sie den Test ausführen möchten.***

Oft funktionierten diese Skripte nicht einmal dann, wenn sie zum ersten Mal ausgeführt wurden. Jegliche Änderung von Bildschirmkontext, Timing, GUI-Eigenschaften oder unzähligen anderen Ursachen würde das aufgezeichnete Skript zum Scheitern bringen.

Um die Testautomatisierung zu verstehen, muss man verstehen, was ein manuelles Testskript ist und wie es verwendet wird.

Ein auf das Mindestmaß reduziertes manuelles Testskript hat meistens nur Informationen in drei Spalten.

Die erste Spalte enthält eine abstrakte Aufgabe; in abstrakter Form deshalb, damit diese nicht geändert werden muss, wenn sich die Software ändert. Zum Beispiel ist die Aufgabe „Datensatz zur Datenbank hinzufügen“ eine abstrakte Aufgabe, die unabhängig von Version und Art der Datenbank ausgeführt werden kann - vorausgesetzt, ein manueller Tester verfügt über das Domänenwissen, um diese in konkrete Aktionen umzuwandeln.

Die zweite Spalte spezifiziert, welche Daten zur Ausführung der Aufgabe zu verwenden sind.

Die dritte Spalte informiert darüber, welches Verhalten zu erwarten ist.

Tabelle 1: Manueller Test (Auszug)

1	Add a record to the database	First name: Gerry Last name: Franklin SSN: 234-34-5678	Record created, Record # returned
2	Search for the name	Franklin, Gerry	Expect to find it
3	Edit the record	Occupation: Lawyer Income: $125,000	Expect dialog verifying change
4	Check record ordering	Record # from step 1	Expect valid ordering
5	Etc.		

Das manuelle Testen funktioniert seit vielen Jahren sehr gut, da wir in der Lage waren, diese manuellen Testskripte zu erstellen. Allerdings ist es nicht das Skript selbst, das das Testen möglich macht. Nur wenn Skript und ein fachkundiger manueller Tester zusammenkommen, lässt sich daraus Nutzen ziehen.

Was trägt der Tester zum Skript bei, dass der Test korrekt ausgeführt wird? Die Antwort lautet: **Kontext** und **Angemessenheit**. Jede Aufgabe wird durch das Wissen des Testers gefiltert: „Welche Datenbank? Welche Tabelle? Wie führe ich diese Aufgabe mit dieser Version dieser Datenbank durch?“ Der Kontext führt zu einigen Antworten, die Angemessenheit führt zu anderen, damit der Tester die Aufgaben im Testskript erfüllen kann.

Ein automatisierter Testlauf durch ein automatisiertes Werkzeug hat begrenzten Kontext und Angemessenheit. Um eine bestimmte Aufgabe (z. B. „Datensatz hinzufügen“) auszuführen, muss sich das Werkzeug genau am gleichen Ort befinden, um die Aufgabe auszuführen. Die Annahme ist, dass der letzte Schritt des automatisierten Testfalls genau da endet, wo die Aufgabe „Datensatz hinzufügen“ ausgeführt werden kann. Und wenn nicht? Pech gehabt! Dann schlägt nicht nur die Automatisierung fehl; auch die Fehlermeldung ist dann wahrscheinlich nichtssagend, da die tatsächliche Fehlerwirkung sehr wahrscheinlich schon während des vorherigen Schritts aufgetreten ist. Ein Tester, der den Test ausführt, wird dieses Problem nie haben.

Ebenso wird bei der Ausführung eines manuellen Tests das Ergebnis einer Aktion vom manuellen Tester überprüft. Wenn der Test beispielsweise das Öffnen einer Datei durch den Tester erfordert (d.h. Aufgabe = „Datei öffnen“), gibt es eine Vielzahl von Dingen, die passieren können. Vielleicht wird die Datei geöffnet. Oder es wird eine Fehlermeldung, eine Warnmeldung, eine Informationsmeldung oder ein Timing-Fehler angezeigt. In jedem Fall liest der manuelle Tester einfach die Nachricht, behandelt das Problem angemessen und im Kontext und tut das Richtige.

Das ist der Kern des Unterschieds zwischen einem automatisierten Skript und einem manuellen Skript. Das manuelle Testskript wird nur dann wertvoll, wenn es in den Händen eines manuellen Testers ist. Wir könnten Zeilen in das manuelle Skript einfügen und dem Tester Hilfestellung geben, wie mit einem Problem umzugehen ist, aber meistens ist das nicht nötig, weil manuelle Tester denken können und über genügend Intelligenz und Erfahrung verfügen, um das Problem selbst zu lösen. Und im schlimmsten Fall können sie telefonieren und einen Experten fragen.

Hier sind einige der Fragen, die ein manueller Tester beantworten kann:

- Wie lange muss ich auf etwas warten?
- Was passiert, wenn ich ein leicht zu behebendes Ergebnis erhalte? (z.B. beim Versuch, eine Datei zu öffnen, kam die Fehlermeldung „Laufwerk nicht zugeordnet“)
- Was passiert, wenn ich eine Warnmeldung bekomme?
- Was passiert, wenn ich 5 Sekunden warten soll, es aber tatsächlich 5,1 Sekunden gedauert hat?

Keines dieser Ergebnisse kann standardmäßig von einem Automatisierungswerkzeug selbst verarbeitet werden. Ein Automatisierungswerkzeug hat keine Intelligenz; es ist halt nur ein Werkzeug. Ein automatisiertes Skript, das aufgezeichnet ist, hat nur begrenzte Möglichkeiten zu entscheiden, was zu tun ist, wenn etwas Unerwartetes eintritt. Eines ist jedoch immer möglich: eine Fehlermeldung ausgeben.

Es ist jedoch möglich, einem automatisierten Skript Intelligenz hinzuzufügen. Als die Testautomatisierer einmal erkannt hatten, dass Intelligenz in das Automatisierungsskript hineinprogrammiert werden kann, konnte die Automatisierung verbessert werden. Durch Programmierung lassen sich die Denkprozesse von manuellen Testern, insbesondere hinsichtlich des Kontextes und der Angemessenheit, erfassen; und damit kann die Automatisierung zu einem Mehrwert gemacht werden, indem Tests häufiger erfolgreich durchgeführt werden und nicht frühzeitig fehlschlagen. Leider gibt es jedoch nichts umsonst – durch zusätzliche Programmierung kann später auch mehr Wartung erforderlich sein.

Nicht alle manuellen Tests sollten automatisiert werden. Da ein automatisiertes Skript mehr Analyse, mehr Entwurf, mehr Engineering und mehr Wartung als ein manuelles Skript benötigt, müssen die Kosten für die Erstellung berücksichtigt werden. Und wenn das automatisierte Skript erstellt ist,

muss es auf Dauer gewartet werden; auch dies ist zu berücksichtigen. Wenn sich das SUT ändert, müssen die automatisierten Skripte häufig auch geändert werden.

Es wird sich nicht vermeiden lassen, dass ein automatisiertes Skript aus anderen Gründen fehlschlagen könnte: Es reicht schon ein Rückgabewert, der bislang noch niemals vorgekommen ist, und die Automatisierung schlägt fehl. In diesem Fall muss das Skript geändert, erneut getestet und erneut bereitgestellt werden. Beim manuellen Testen ist all das im Allgemeinen kein Problem.

Ein Business Case muss schon vor und auch während der Testautomatisierung erstellt werden. Werden sich Gesamtkosten für die Automatisierung dieses Tests amortisieren, wenn dieser N-mal in den nächsten M Monaten ausgeführt werden kann? Die Antwort auf diese Frage ist häufig ein Nein. Einige Tests werden auch weiterhin manuell bleiben, da ihre Automatisierung nicht rentabel wäre (kein positiver ROI).

Manche manuelle Tests können auch gar nicht automatisiert werden, da der Denkprozess des manuellen Testers für den Erfolg des Tests wesentlich ist. Exploratives Testen, Fehlerangriffe und einige andere Arten von manuellen Tests sind auch weiterhin für erfolgreiches Testen notwendig.

1.3 Erfolgsfaktoren

Automatisierung ist kein Zufallsprodukt. Eine erfolgreiche Automatisierung erfordert normalerweise:

- Einen langfristigen Plan, der auf die Bedürfnisse des Unternehmens abgestimmt ist
- Solides, intelligentes Management
- Viel Aufmerksamkeit fürs Detail
- Prozessreife
- Formale Testautomatisierungs-Architektur und -Framework
- Angemessene Ausbildung
- Ausgereifte Dokumentationsebenen.

Ob Automatisierung möglich ist oder nicht, hängt häufig von der Testbarkeit des SUT ab. In vielen Fällen sind die Schnittstellen eines SUT einfach nicht zum Testen geeignet. Wenn die Entwickler des Systems spezielle oder eigene Schnittstellen (oft als „Hooks“ bezeichnet) zur Verfügung stellen, kann das oft ausschlaggebend für den Erfolg oder Misserfolg der Automatisierung sein.

Es gibt mehrere verschiedene Schnittstellenebenen, auf denen ein SUT automatisiert werden kann:

- Auf GUI-Ebene (häufig die fehleranfälligste Ebene)
- Auf API-Ebene (unter Verwendung von Anwendungsprogrammierschnittstellen, die die Entwickler für die öffentliche oder geschützte Verwendung bereitstellen)
- Eigene Hooks (APIs, die speziell zum Testen dienen)
- Auf Protokollebene (HTTP, TCP usw.)
- Auf Serviceebene (SOAP, REST usw.)

Zu beachten ist, dass Selenium auf UI-Ebene arbeitet. Dieser Lehrplan enthält Tipps und Techniken, um die Anfälligkeit zu reduzieren und die Benutzerfreundlichkeit beim Testen auf dieser Ebene zu verbessern.

Die folgenden Punkte sind allesamt Erfolgsfaktoren für die Automatisierung:

- Management, das gelernt hat zu verstehen, was möglich ist und was nicht (unrealistische Erwartungen des Managements sind einer der Hauptgründe für das Scheitern von Software-Testautomatisierungsprojekten)
- Ein Entwicklungsteam für das SUT (bzw. die SUTs), das die Automatisierung versteht und bereit ist, bei Bedarf mit den Testern zu kooperieren
- Ein SUT (bzw. SUTs), das für gute Testbarkeit ausgelegt ist
- Entwicklung eines kurz-, mittel- und langfristigen Business Case
- Vorhandensein der richtigen Werkzeuge für die Umgebung und für das SUT (bzw. die SUTs)
- Das richtige Training, sowohl in den Test- also auch den Entwicklungsdisziplinen
- Eine gut entworfene und gut dokumentierte Architektur und Framework zur Unterstützung der einzelnen Skripte (der ISTQB® Certified Tester Advanced Level - Test Automation Engineer Lehrplan nennt dies TAS bzw. „Test Automation Solution" (deutsch „Testautomatisierungslösung"))
- Eine gut dokumentierte Testautomatisierungsstrategie, die vom Management gesponsert und unterstützt wird
- Ein formaler und gut dokumentierter Plan für die Wartung der Automatisierung
- Automatisieren auf der für den Kontext der erforderlichen Tests richtigen Schnittstellenebene

1.4 Risiken und Nutzen von Selenium WebDriver

WebDriver ist eine Programmierschnittstelle für die Entwicklung fortgeschrittener Selenium-Skripte unter Verwendung der folgenden Programmiersprachen:

- C#
- Haskell
- Java
- JavaScript
- Objective C
- Perl
- PHP
- Python
- R
- Ruby

Für viele dieser Sprachen gibt es auch Open-Source-Test-Frameworks.

Browser, die von Selenium unterstützt werden (und der Komponente, mit der getestet werden soll), umfassen:

- Chrome (chromedriver.exe)
- Internet Explorer (IEDriverServer.exe)
- Edge (MicrosoftWebDriver.msi)
- Firefox (geckodriver.exe)
- Safari (safaridriver)
- HtmlUnit (HtmlUnit driver)

Selenium WebDriver arbeitet mit seinen APIs (Application Programming Interfaces), um direkte Aufrufe an einen Browser zu tätigen, wobei jeweils die browsereigene Unterstützung für die Automatisierung genutzt wird. Jeder unterstützte Browser arbeitet etwas anders.

Wie bei allen Werkzeugen gibt es auch für die Verwendung von Selenium WebDriver Nutzen und Risiken.
Viele der Vorteile, von denen Unternehmen profitieren können, sind dieselben wie bei anderen Automatisierungswerkzeugen, einschließlich:

- Die Testausführung kann konsistent und wiederholbar sein
- Gut geeignet für Regressionstests
- Da auf UI-Ebene getestet wird, werden Fehler erkannt, die beim Testen auf API-Ebene übersehen wurden
- Geringere Anfangsinvestitionen, da Open-Source
- Funktioniert mit verschiedenen Browsern, Kompatibilitätstests sind somit möglich
- Unterstützt verschiedene Programmiersprachen und kann daher von mehr Personen genutzt werden
- Nützlich in agilen Teams, da ein tieferes Verständnis des Codes erforderlich ist

Es gibt jedoch auch Risiken. Die folgenden Risiken sind mit der Nutzung von Selenium WebDriver verbunden.

- Unternehmen sind oft so sehr mit den GUI-Tests beschäftigt, dass sie vergessen, dass die Testpyramide mehr Unit- bzw. Komponententests fordert
- Beim Testen mit einem kontinuierlichen Integrationsprozess (Continuous Integration bzw. CI) kann durch die Automatisierung der Build viel länger dauern als gewünscht
- Änderungen auf UI-Ebene verursachen mehr Schaden bei Tests auf Browser-Ebene als bei den Tests auf Unit- oder API-Ebene
- Manuelle Tester sind effizienter beim Aufdecken von Fehlern als die Automatisierung
- Tests, die schwer zu automatisieren sind, werden womöglich weggelassen
- Die Automatisierung muss häufig einen positiven ROI (Return on Investment) erzielen. Wenn eine Webanwendung jedoch relativ stabil ist, kann es sein, dass die Automatisierung nicht kostendeckend ist.

1.5 Der Selenium WebDriver in der Testautomatisierungsarchitektur

Die TAA (Test Automation Architecture) ist gemäß der Definition des ISTQB® Certified Tester Advanced Level - Test Automation Engineer Lehrplans eine Menge von Schichten, Diensten und Schnittstellen der TAS (= Test Automation Solution bzw. deutsch Testautomatisierungslösung).

Die TAA besteht aus vier Schichten (siehe nachfolgende Abbildung):

- Testgenerierungsschicht: unterstützt den manuellen oder automatisierten Entwurf von Testfällen
- Testdefinitionsschicht: unterstützt die Testrealisierung und Definition von Testfällen und/oder Testsuiten
- Testausführungsschicht: unterstützt sowohl die Ausführung von automatisierten Tests als auch die Protokollierung und Berichterstattung über die Ergebnisse
- Testadaptierungsschicht: Stellt die erforderlichen Objekte und den Code für die Schnittstelle mit dem SUT (zu testendes System) auf verschiedenen Ebenen bereit

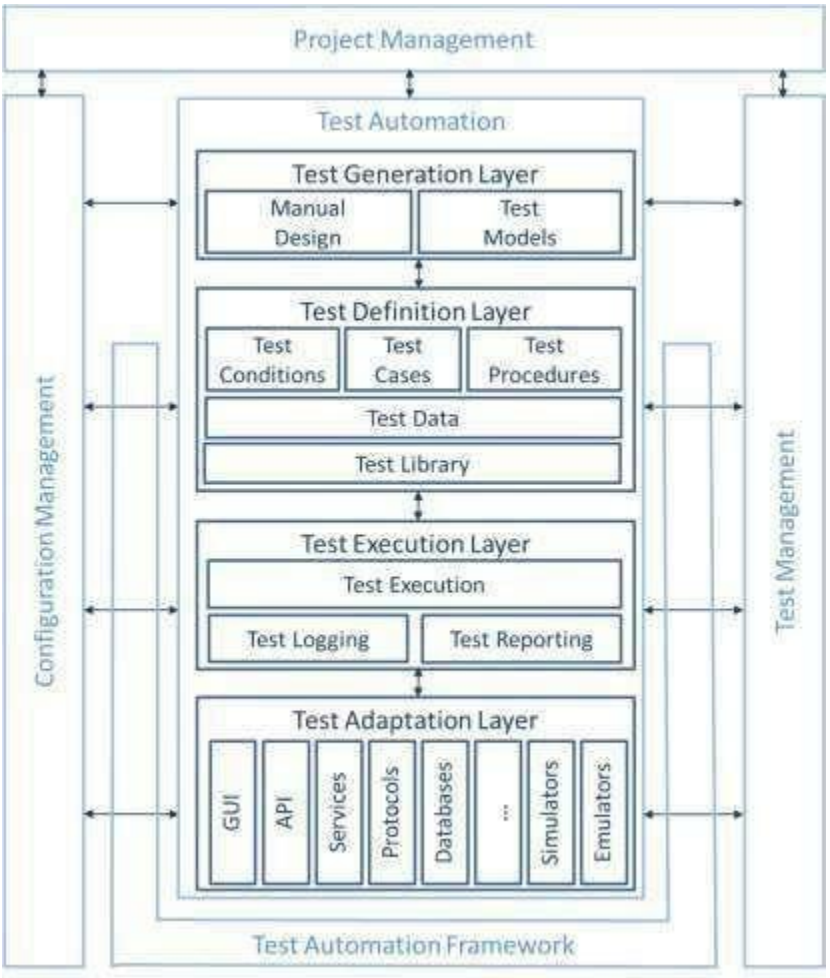

Abb. 1: Generische TAA (aus dem englischsprachigen ISTQB® TAE Lehrplan)

Der Selenium WebDriver gehört in die Testadaptierungsschicht und bietet eine programmatische Möglichkeit, über die Browserschnittstelle auf das SUT zuzugreifen.

Die Testadaptierungsschicht ermöglicht die Trennung des SUT (und seiner Schnittstellen) von den Testfällen und Testsuiten, die das SUT als Testobjekt testen sollen.

Idealerweise können durch diese Trennung die Testfälle abstrakter und vom getesteten System getrennt sein. Wenn sich das zu testende System ändert, müssen die Testfälle selbst möglicherweise nicht geändert werden, vorausgesetzt, dass nach wie vor die gleiche Funktionalität bereitgestellt wird, nur eben auf eine etwas andere Art und Weise.

Wenn sich das SUT ändert, ermöglicht die Testadaptierungsschicht, in diesem Fall der WebDriver, die Modifikation des automatisierten Tests, wodurch der tatsächliche konkrete Testfall mit Bezug auf die modifizierte Schnittstelle des SUT ausgeführt werden kann.

Das automatisierte Skript, das WebDriver verwendet, nutzt eigentlich die API, um zwischen dem Test und dem SUT wie folgt zu kommunizieren:

- Der Testfall ruft eine Aufgabe auf, die ausgeführt werden soll
- Das Skript ruft eine API im WebDriver auf
- Die API stellt eine Verbindung mit dem entsprechenden Objekt im SUT her
- Das SUT antwortet (hoffentlich) wie gewünscht
- Der Erfolg bzw. Misserfolg werden dem Skript gemeldet
- Falls erfolgreich, wird der nächste Schritt im Testfall im Skript aufgerufen

1.6 Zweck der Sammlung von Metriken in der Automatisierung

Es gibt einen alten Spruch in Managementkreisen, der besagt: „Was gemessen wird, wird auch gemacht.“ Messungen sind etwas, das viele Testautomatisierer gerne ignorieren oder verschleiern, weil diese oft schwer zu quantifizieren und nachzuweisen sind.

Das Problem besteht darin, dass die Automatisierung sowohl hinsichtlich des Personalaufwands als auch der benötigten Werkzeuge ziemlich kostspielig ist. Auf kurze Sicht gibt es wenig bis keinen positiven ROI. Wert lässt sich nur auf lange Sicht erzielen; über einen Zeitraum von Jahren und nicht Monaten. Wer das Management um eine große Investition in Zeitaufwand und Ressourcen bittet und keinen stichhaltigen Business Case dafür vorlegt (einschließlich der gesammelten Metriken, um diesen zu untermauern), fordert es auf, an die Tester zu glauben und einfach Vertrauen zu haben. Leider klingen solche Darlegungen in den Ohren der höheren Führungsebene von Unternehmen wie Hohn.

Daher sollten nach Möglichkeit sinnvolle Metriken gesammelt werden, die den Wert der

Automatisierung aufzeigen, wenn das Automatisierungsprojekt nicht Gefahr laufen soll, von Erbsenzählern gestoppt zu werden. Der ISTQB® Certified Tester Advanced Level - TAE Lehrplan nennt einige Bereiche, in denen Metriken nützlich sein können, einschließlich:

- Gebrauchstauglichkeit
- Wartbarkeit
- Performanz
- Zuverlässigkeit

Viele der im TAE-Lehrplan des ISTQB® erwähnten Metriken sind wahrscheinlich eher für Automatisierungsteams in großen Projekten nützlich als für kleine Automatisierungsprojekte, die den Selenium WebDriver verwenden. Dieser Abschnitt konzentriert sich auf kleinere Projekte. Weitere Informationen zu großen, komplexen Automatisierungsprojekten finden Sie im ISTQB® Certified Tester Advanced Level - TAE Lehrplan.

Das Sammeln von unangemessenen Metriken ist für ein kleines Team, das versucht, ein Automatisierungsprojekt erstmals in Gang bekommen, wahrscheinlich eher eine Ablenkung. Ein Problem bei der Identifizierung, welche Metriken Sinn machen, besteht darin, dass die Testausführungszeit bei der Testautomatisierung zwar tendenziell kürzer ist als bei den entsprechenden manuellen Tests, dass aber Analyse, Entwurf, Entwicklung, Fehlerbehebung und Wartung automatisierter Tests tendenziell viel mehr Zeit in Anspruch nehmen als bei den entsprechenden manuellen Tests.

Der Versuch, sinnvolle Metriken zu erfassen, um den Business Case (ROI) zu bestimmen, wird oft zu einem Vergleich von Äpfeln mit Birnen. Die Erstellung eines bestimmten manuellen Testfalls kann beispielsweise 1 Stunde dauern; und 30 Minuten, um diesen auszuführen. Angenommen, wir planen, den Test für die aktuelle Version drei Mal auszuführen, dann brauchen wir für diesen Test insgesamt 2,5 Stunden. Eine einfache Messung beim manuellen Testen.

Wie viel kostet es, diesen Test zu automatisieren? Zunächst muss bestimmt werden, ob die Automatisierung dieses Tests überhaupt einen Mehrwert darstellt. Angenommen, der Test soll automatisiert werden, dann ist zu überlegen, wie automatisiert werden soll, der benötigte Code muss geschrieben und debuggt werden, und es muss sichergestellt werden, dass der Code das tut, was wir erwarten. Das kann mehrere Stunden dauern. Dann unterliegt jeder Test verschiedenen Herausforderungen - wie ist das einzuschätzen?

Der manuelle Test, der abstrakt ist, muss wahrscheinlich nicht geändert werden, wenn sich die GUI des SUT ändert. Das automatisierte Skript müsste wahrscheinlich sogar bei geringfügigen Änderungen an der GUI modifiziert werden. Das ließe sich durch einen guten Entwurf der Architektur und des Frameworks zwar minimieren; dennoch brauchen Änderungen Zeit. Wie kann geschätzt werden, wie viele erwartete Änderungen es geben wird?

Wenn ein manueller Test fehlschlägt, ist die Fehlerbehebung in der Regel unkompliziert. Der Test Analyst kann normalerweise leicht analysieren, was schiefgelaufen ist, und den Fehlerbericht erstellen.

Ein Automatisierungsfehler hingegen kann mehr Zeit zur Fehlerbehebung benötigen (siehe Abschnitt „Protokollierung" weiter unten).

Wie berechnet man, wie oft der Test fehlschlagen könnte - insbesondere, wenn diese Ausfälle häufig nicht vom SUT verursacht sind, sondern von der Automatisierung selbst? Um die Berechnung des ROI noch schwieriger zu machen, erfordert die Automatisierung im Vorfeld hohe Investitionen, die für manuelle Tests nicht erforderlich sind. Diese Investition umfasst den Kauf eines oder mehrerer Werkzeuge, geeignete Umgebungen zum Ausführen der Werkzeuge, die Erstellung der Architektur und des Frameworks für die Automatisierung, die Schulung (oder Einstellung) von Testautomatisierern und weitere Fixkosten.

Diese Investition muss in jede Berechnung des ROI einfließen, die wir vornehmen könnten. Da es sich um fixe Kosten handelt, bedeutet das wahrscheinlich, dass eine große Anzahl von Tests automatisiert und ausgeführt werden müssen, um einen positiven ROI zu erzielen. Dies erfordert ein höheres Maß an Skalierbarkeit, was wiederum die Kosten für die Automatisierung erhöht.

Ein weiteres Problem mit vielen Automatisierungsmetriken besteht darin, dass sie oft eher geschätzt als direkt gemessen werden müssen. Das kann bedeuten, dass sie oft verfälscht werden, um zu beweisen, dass sich die Automatisierung lohnt.

Hier sind einige Metriken, die für ein kleines Start-up-Projekt nützlich sein könnten:

- Fixkosten, um die Automatisierung in Gang zu setzen
- Aufwand für den Regressionstest, der durch die Automatisierung eingespart wurde
- Aufwand des Automatisierungsteams, das die Automatisierung unterstützt
- Überdeckung
    - Auf Unit-Test-Ebene - Anweisungs-/Entscheidungsüberdeckung
    - Auf Integrationsteststufe – Schnittstellen- oder Datenflussüberdeckung
    - Auf Systemtestebene - Überdeckung von Anforderungen, Features oder identifizierten Risiken
    - Getestete Konfigurationen
    - Abgedeckte User-Stories (agiles Projekt)
    - Abgedeckte Anwendungsfälle
- Abgedeckte Konfigurationen, die getestet wurden
- Anzahl erfolgreicher Testläufe zwischen Ausfällen
- Muster von Automatisierungsfehlerwirkungen (Suche nach Gemeinsamkeiten von Problemen durch Verfolgung der Grundursachen der Fehler)
- Anzahl der gefundenen Automatisierungsfehlerwirkungen im Vergleich zu den durch die Automatisierung aufgedeckten SUT-Fehlerwirkungen

Abschließend noch ein letzter Punkt zu Metriken. Versuchen Sie, aussagekräftige Messwerte für Ihr Projekt zu finden, die erklären, warum das Projekt gültig und wichtig ist. Verhandeln Sie mit dem Management und stellen Sie sicher, dass sie verstehen, dass der Nachweis des Werts der Automatisierung am Anfang schwierig ist und dass es oft viel Zeit braucht, bis sinnvolle Metriken erreicht werden können. Es mag verlockend sein, den Wert der Automatisierung mit Hilfe von vielversprechenden Szenarien und etwas „Kreativität" zu beweisen. Wenn das Management jedoch dahinterkommt, könnte es das Ende des Projekts bedeuten.

Wenn es richtig gemacht wird, hat ein Automatisierungsprojekt eine gute Chance, einen Mehrwert für das Unternehmen zu liefern, auch wenn es schwer zu beweisen ist.

1.7 Die Selenium Toolfamilie im Überblick

Der Selenium WebDriver ist nicht das einzige Werkzeug aus dem Hause Selenium. Insgesamt gibt es vier Werkzeuge (allesamt Open-Source), die jeweils unterschiedliche Rollen in der Testautomatisierung spielen:

- Selenium IDE
- Selenium WebDriver
- Selenium Grid
- Selenium Standalone Server (eigenständiger Server)

In der Geschichte der Selenium-Werkzeuge gab es das Werkzeug Selenium RC, das die Selenium-Version 1 implementierte. Dieses Werkzeug wird mittlerweile nicht mehr verwendet.

Selenium IDE ist ein Add-on zu den Chrome- und Firefox-Browsern und funktioniert nicht als eigenständige Anwendung. Seine Hauptfunktion ist die Aufzeichnung und Wiedergabe von Benutzeraktionen auf Webseiten. Selenium IDE ermöglicht es einem Testautomatisierer außerdem, während des Mitschnitts Verifizierungspunkte einzugeben. Die aufgezeichneten Skripte können als HTML-Tabellen gespeichert oder in verschiedene Programmiersprachen exportiert werden.

Die Hauptvorteile von Selenium IDE sind seine Einfachheit und die relativ gute Erkennung von Objekten (durch „Element-Locators"). Der Hauptnachteil ist das Fehlen von Variablen, Verfahren und Kontrollflussanweisungen; daher ist es für die Erstellung robuster automatisierter Tests nicht sehr nützlich. Selenium IDE wird hauptsächlich zum Aufzeichnen provisorischer Skripte (z.B. zu Debugging-Zwecken) oder nur zur Suche nach Element-Locators verwendet.

Der Selenium WebDriver ist hauptsächlich ein Framework, das die Steuerung von Webbrowsern durch Testskripte ermöglicht. Es basiert auf HTTP und wurde vom W3C standardisiert. Dieser Lehrplan zeigt grundlegende Merkmale dieses Protokolls in **Fehler! Verweisquelle konnte nicht gefunden werden.**s. Der Selenium WebDriver unterstützt viele verschiedene Programmiersprachen,

z.B. Java, Python, Rubin, C#. In diesem Lehrplan wird für die Beispiele Python verwendet, um zu zeigen, wie verschiedene WebDriver-Objekte und ihre Methoden verwendet werden können.

Durch die Verwendung von Bibliotheken, die die WebDriver-API für verschiedene Programmiersprachen implementieren, können Testautomatisierer die Fähigkeit des WebDrivers, Webbrowser zu steuern, mit der Leistungsfähigkeit allgemeiner Programmiersprachen kombinieren und nutzen. Dies ermöglicht es den Testautomatisierern, für die Erstellung komplexer Testautomatisierungs-Frameworks, einschließlich Protokollierung, Assertion-Handling, Multi-Threading und vielem mehr, Bibliotheken anderer Sprachen zu verwenden.

Um einen langfristigen Erfolg zu erzielen, muss das Testautomatisierungs-Framework mit Sorgfalt und guten Design-Prinzipien erstellt werden, wie im Abschnitt **Fehler! Verweisquelle konnte nicht gefunden werden...** beschrieben.

Einige Webbrowser benötigen zusätzliche WebDriver-Prozesse, um neue Testinstanzen zu starten und zu steuern. In diesem Fall ruft ein Skript Befehle aus der Bibliothek auf, die die Befehle durch den WebDriver-Prozess an den Webbrowser sendet. Dies wird in Kapitel 3 behandelt.

Ein weiteres Werkzeug, das in einer Testumgebung nützlich sein kann, ist Selenium Grid. Dieses ermöglicht das Ausführen von Testskripten auf mehreren Maschinen mit unterschiedlichen Konfigurationen. Die verteilte und gleichzeitige Ausführung von Testfällen ist möglich. Die Architektur von Selenium Grid ist sehr flexibel. Es kann für die Verwendung vieler physischer oder virtueller Maschinen mit verschiedenen Kombinationen von Betriebssystemen und Webbrowser-Versionen konfiguriert werden.

Das Herzstück von Selenium Grid ist der Selenium Hub, der andere Maschinen (nodes) steuert und als einziger Kontaktpunkt für Testskripte fungiert. Testskripte, die WebDriver-Befehle ausführen, müssen (in den meisten Fällen) nicht geändert werden, um auf verschiedenen Betriebssystemen oder Webbrowsern zu funktionieren.

Das letzte Werkzeug der Selenium Toolfamilie, das in diesem Lehrplan zu erwähnen ist, ist Selenium Standalone Server. Dieses Werkzeug ist in Java geschrieben und wird als JAR-Datei geliefert, die Hubs und Node-Funktionen für Selenium Grid implementiert. Dieses Tool muss separat gestartet werden (außerhalb der Testskripte), und es muss ordnungsgemäß konfiguriert werden, um seine Rolle in der Testumgebung zu spielen.

Eine detailliertere Beschreibung der Werkzeuge Selenium Grid und Selenium Standalone Server ist nicht Gegenstand dieses Lehrplans.

Kapitel 2 – Internettechnologien für die Testautomatisierung von Webanwendungen

Schlüsselbegriffe

CSS-Selektor, HTML, Tag, XML, XPath

Lernziele für Internettechnologien für die Testautomatisierung von Webanwendungen

STF-2.1 (K3) HTML- und XML-Dokumente verstehen und erstellen können
STF-2.2 (K3) XPath zum Durchsuchen von XML-Dokumenten anwenden können
STF-2.3 (K3) CSS-Selektoren anwenden können, um Elemente von HTML-Dokumenten zu finden

2.1 Grundlagen von HTML und XML

2.1.1 Grundlagen von HTML

Es wäre nicht übertrieben zu sagen, dass HTML (Abkürzung für HyperText Markup Language bzw. deutsch „Hypertext-Auszeichnungssprache) das World Wide Web zugänglich gemacht hat. Ein HTML-Dokument ist eine einfache Textdatei, die Elemente enthält, die bestimmte Kontextbedeutungen spezifizieren, wenn das Dokument eingelesen wird und entsprechende Aktionen durchgeführt werden (in der Fachsprache „parsen"). Die Elemente arbeiten zusammen, um zu ermitteln, wie ein Browser diese Teile des Dokuments aufbereiten und darstellen soll (in der Fachsprache „rendern"). Im Wesentlichen beschreibt HTML die Struktur einer Webseite semantisch.

Der vielleicht wichtigste Vorteil bei der Verwendung von HTML zum Spezifizieren von Webseiten ist die universelle Anwendbarkeit der Sprache. Wenn es richtig erstellt ist, kann jeder Browser auf jedem Computersystem die Seite korrekt darstellen.

Wie die Seite dann tatsächlich aussieht, kann sich je nach Ausgabemedium oder Computertyp (z.B. PC gegenüber einem Smartphone), dem Monitor, der Internetverbindungsgeschwindigkeit und dem verwendeten Browser ändern.

Die Erfindung des World Wide Web wird Timothy Berners Lee zugeschrieben. Als er 1980 beim CERN (Europäische Organisation für Kernforschung) arbeitete, schlug er vor, Hypertext zu verwenden, um Informationen unter Forschern auszutauschen und zu aktualisieren. Dann implementierte er 1989 die erste effektive Kommunikation zwischen einem HTTP (HyperText Transfer Protocol)-Client und -Server. Damit legte er die Grundlage für das World Wide Web, das die Welt nachhaltig verändert hat.

HTML-Elemente werden eingeführt und oft von Tags (deutsch: Auszeichner) eingeschlossen, die durch spitze Klammern gekennzeichnet sind:

```
<!DOCTYPE html>
<html>
   <head>
 <title> Page Title</title>
   </head>
   <body>
      <h1>This is a Heading</h1>
      <p>This is a paragraph</p>
   </body>
</html>
```

Abb. 2: HTML Beispiel

Manche Tags führen Inhalte direkt in die gerenderte Seite ein. **(** führt beispielsweise dazu, dass ein Bild auf der Seite platziert wird). Andere Tags umschließen und liefern semantische Informationen darüber, wie das Element gerendert werden soll (siehe Überschrift-Element **<h1> .... </ h1>** im Beispiel oben). Auszeichner bzw. Tags werden weiter unten behandelt.

In der Geschichte des World Wide Web wurde über viele Jahre die HTML-Version 4.01 verwendet. Im Jahr 2014 veröffentlichte Verwaltungsrat, der das HTML steuert, die aktuelle Version HTML 5.

HTML ist relativ flexibel und variabel in der Art und Weise, wie die Tags bzw. Auszeichner verwendet werden (z.B. kann bei einigen Elementen der End-Tag fehlen). Dies hat dazu geführt, dass manche Browser die Seiten falsch rendern. XML, das weiter unten behandelt wird, ist strenger als HTML; hier muss jede Seite „wohlgeformt“ sein, und für jedes öffnende Tag muss es ein schließendes Tag geben. Wenn HTML wohlgeformt ist (d.h. jeder Beginnauszeichner bzw. Start-Tag hat einen Endauszeichner bzw. End-Tag), ist HTML ein Teilsatz von XML.

Automatisierung mit Selenium erfordert ein Verständnis von HTML-Tags. Um eine GUI zu automatisieren, muss der Testautomatisierer jedes eindeutige Steuerelement im Bildschirm, der manipuliert wird, identifizieren können. Der Testautomatisierer kann durch das Durchsuchen der HTML-Seite die Steuerelemente, die auf der vom Browser gerenderten Seite platziert werden, eindeutig erkennen. Um die Steuerelemente zu finden, muss der Testautomatisierer Aufbau und Logik der HTML-Seite verstehen. Dies geschieht durch Verstehen und Parsen der Tags.

HTML-Elemente haben normalerweise ein Start-Tag und ein End-Tag. Das End-Tag ist dasselbe wie das Start-Tag, außer dass dem End-Tag ein Schrägstrich vorangestellt ist, wie unten gezeigt:

** **Textabsatz** **

Manche Elemente können innerhalb des offenen Tags in sich geschlossen werden; siehe folgendes Beispiel eines Elements für einen Zeilenumbruch:

**
**

Eine weniger strenge Implementierung des Zeilenumbruches, die für einige Browser allerdings Probleme verursachen könnte, ist die folgende:

**
**

Die nachfolgende Tabelle enthält Tags, die jeder Selenium-Testautomatisierer kennen sollte.

Tabelle 2: Grundlegende HTML-Tags

Tag (Auszeichner)	Verwendung
<html> … </html>	Bezeichnet den Ursprung des HTML-Dokuments
<!DOCTYPE>	Definiert den Dokumententyp (in HTML 5 nicht erforderlich)
<head> … </head>	Definition und Metadaten für das Dokument
<body> … </body>	Definiert den Hauptinhalt des Dokuments
…	Definiert einen Absatz
 	Fügt einen einzelnen Zeilenumbruch ein
…	Definiert einen Abschnitt im Dokument
<!-- … -->	Definiert einen Kommentar (kann mehrzeilig sein)

Überschrift-Tags definieren unterschiedliche Gliederungsstufen von Überschriften. Das tatsächliche Format des Textes (Größe, Fettdruck, Schriftart) kann in CSS-Formatvorlagen (Stylesheets) festgelegt werden.

Tabelle 3: Header-Tags

Tag (Auszeichner)	Rendering

Heading 1	**Überschrift 1**
Heading 2	Überschrift 2
Heading 3	Überschrift 3
Heading 4	Überschrift 4
Heading 5	Überschrift 5
Heading 6	Überschrift 6

Links und Bilder sind für gut entworfene Browser-Seiten von grundlegender Bedeutung und mit HTML einfach wie folgt zu erstellen:

*****link text*****

Diese Kombination von Symbolen beginnt mit einem **<a ...>** Anker-Tag (bzw. Link-Anker) und endet mit einem ****. Diese definieren einen Hyperlink, der angeklickt werden kann. Die **href =** *"URL"* ist ein Attribut, das das Ziel des Links angibt. Der Text zwischen den Tags (*link text*) stellt den Text dar, der im Link angezeigt wird und auf den der Benutzer klickt, um zum URL-Ziel zu gelangen.

****

Das Basis-Tag in diesem Beispiel (****) definiert ein Bild, das an dieser Stelle im Dokument platziert wird. Das Attribut **src = "**pulpitrock.jpg**"** ist die Linkadresse des tatsächlichen Bildes, das angezeigt wird. Das andere Attribut (**alt = "***Mountain view***"**), enthält den Text, der angezeigt wird, wenn das Bild nicht gefunden oder angezeigt werden kann.

Tabelle 4: Listen und Tabellen-Tags

** … **	**Definiert eine unsortierte Punkte-Liste (Aufzählung)**
** … **	**Definiert eine geordnete (nummerierte) Liste**
** … **	**Definiert ein Listenelement (für oder)**
<table> … </table>	**Definiert eine HTML-Tabelle**

<tr> … </tr>	**Definiert eine Tabellenzeile**
<th> … </th>	**Definiert die Spaltenüberschrift einer Tabelle**
<td> … </td>	**Definiert eine Datenzelle einer Tabelle**
<tbody> … </tbody>	**Gruppiert Body-Inhalt in einer HTML-Tabelle**
<thead> … </thead>	**Definiert den Header (Kopfzeile) einer HTML-Tabelle**
<tfoot> … </tfoot>	**Definiert den Footer (Fußzeile) einer HTML-Tabelle**
<colgroup> … </colgroup>	**Gruppiert Tabellenspalten für die Formatierung**

Listen sind einfach zu definieren und zu rendern. Der folgende HTML-Code rendert die nachfolgende Liste:

```
<!DOCTYPE html>
<html>
<head>

<body>

<h4>An Unordered List containing an Ordered List</h4>
<ul>
      <li>Coffee</li>
    <li>Tea</li>
    <ol>
      <li>Oolong</li>
         <li>Black</li>
         <li>Earl Grey</li>
    </ol>
    <li>Milk</li>
</ul>
</body>
</html>
```

Abb. 3: Beispiel einer HTML-Liste

An Unordered List containing an Ordered List

- Coffee
- Tea
    1. Oolong
    2. Black
    3. Earl Grey
- Milk

Abb. 4: Eine ungeordnete Liste, die eine geordnete Liste enthält

Auch Tabellen sind einfach zu erstellen und zu rendern. Da Testergebnisse oft in Form von Tabellen vorliegen, haben Testautomatisierer häufig mit Tabellen zu tun. Das folgende Codebeispiel wird die Tabelle rendern wie unten dargestellt:

```
<!DOCTYPE html>
<html>
     <head>
          <style>
               table, th, td {
                   border: 1px solid black;
               }
          </style>
     </head>
     <body>

     <table>
          <tr>
               <th>Year</th>
               <th>Car Payment</th>
          </tr>
          <tr>
               <td>2017</td>
               <td>$3,780</td>
          </tr>
          <tr>
               <td>2018</td>
               <td>$2,905</td>
          </tr>
          <tr>
               <td>2019</td>
               <td>$4,812</td>
          </tr>
          <tr>
               <td>2020</td>
               <td>$1,790</td>
          </tr>
     </table>
     </body>
</htm
```

Abb. 5: Code zur Generierung der Tabelle

Tabelle 5: Gerenderte Tabelle

Year	Car Payment
2017	$3,780
2018	$2,905
2019	$4,812
2020	$1,790

HTML-Formulare und die zugehörigen Steuerelemente werden verwendet, um Eingaben von Benutzern zu sammeln. Im Folgenden sind die Tags aufgeführt, die zum Rendern der Formulare und der Steuerelemente erforderlich sind. Benutzer von Selenium WebDriver müssen mit solchen Formularen und Steuerelementen häufig umgehen, um ihre Tests zu automatisieren.

Die folgenden Tags werden zum Erstellen von Steuerelementen auf dem Bildschirm verwendet:

<form> … </form>

Dieser Tag definiert ein HTML-Formular für Benutzereingaben:

<input>

Der nachfolgende Code definiert ein Eingabesteuerelement. Die Art der Steuerung wird durch das Attribut **type=** definiert. Zu den möglichen Typen gehören Text, Radio-Buttons, Kontrollkästchen, Schaltfläche „Submit" usw. Die folgenden Zeilen werden beispielsweise wie folgt gerendert:

```
<form action="/action_page.php">
First name: <input type="text" name="FirstName" value="Mortey"><br>
Last name: <input type="text" name="LastName" value="Moose"><br>
<input type="submit" value="Submit">
</form>
```

Abb. 6: Formularliste

Abb. 7: Eingabefelder aus der Formularliste

<textarea> … </textarea>

Der nachfolgende Code definiert ein mehrzeiliges Eingabesteuerelement. Der Textbereich kann eine unbegrenzte Anzahl von Zeichen enthalten. Die folgenden Zeilen werden beispielsweise wie folgt gerendert:

```
<textarea rows="4" cols="50">
Selenium allows you to automate browsers with
maximum return and minimum effort.
</textarea>
```

Abb. 8: HTML Code für mehrzeiliges Eingabesteuerelement

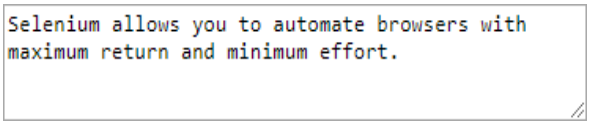

Abb. 9: Mehrzeiliges Eingabesteuerelement

<button>

Dies definiert eine anklickbare Schaltfläche.

<select> … </select>

Dies definiert eine Dropdown-Liste. Wenn der Autor diese zusammen mit den Tags **<option> ... </option>** verwendet, kann er eine Dropdown-Liste definieren und sie wie folgt füllen:

```
<select>
  <option value="volvo">Volvo</option>
  <option value="saab" >Saab</option>
  <option value="mercedes">Mercedes</option>
  <option value="audi">Audi</option>
</select>
```

Abb. 10: Code für das Auswahlsteuerelement

Der obige Code wird das folgende Dropdown-Steuerelement darstellen:

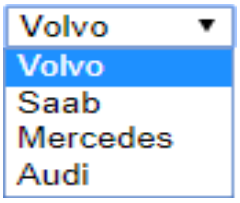

Abb. 11: Dropdown-Liste

<fieldset> … </fieldset>

Mit diesen Tags kann der Autor verwandte Elemente in einem Formular gruppieren. Bei Verwendung mit dem Tag **<legend>** wird eine benannte Box um die Steuerelemente gezeichnet, die als verwandt gelten; siehe Beispiel unten:

```
<fieldset>
   <legend>Child 1:</legend>
     Name: <input type="text"><br>
     Email: <input type="text"><br>
     Date of birth: <input type="text">
 </fieldset>
```

Abb. 12: Fieldset-Code

Der obige Code generiert die folgenden gruppierten Steuerelemente:

Abb. 13: Gruppierte Elemente eines Formulars

2.1.2 Grundlagen von XML

XML (Abkürzung für eXtensible Markup Language bzw. deutsch „erweiterbare Auszeichnungssprache") ist eine Auszeichnungssprache, die zum Definieren von Regeln zum Formatieren von Dokumenten verwendet wird, damit diese maschinenlesbar und auch für Menschen gut lesbar sind. XML wurde entwickelt, um Einfachheit und Benutzerfreundlichkeit im World Wide Web zu befördern. Obwohl in Dokumenten verwendet, erlaubt XML auch die Darstellung von beliebigen Datenstrukturen, die spontan erstellt werden können.

HTML wurde für die Anzeige von Daten entwickelt mit besonderem Fokus darauf, wie die Daten aussehen. XML wurde als Software- und Hardware-unabhängiges Werkzeug entwickelt, mit dem Daten in einem lesbaren Format ausgetauscht und gespeichert werden können.

Anders als HTML-Tags sind XML-Tags nicht vordefiniert. Stattdessen werden Tags vom Ersteller des XML-Dokuments für alle Standards spezifiziert, die sie verwenden möchten. Das Format der Tags entspricht weitgehend dem von HTML. Hier ist ein Beispiel für einen Satz von Feldern in XML:

```
<?xml version="1.0" ?>
<note>
      <date>2018-06-12</date>
      <hour>10:30</hour>
       <to>Francis</to>
       <from>Morrow</from>
       <body>Please pick me up this weekend!</body>
</note>
```

Abb. 14: Beispiel für XML-Code

Es ist zu beachten, dass es für jeden Beginn-Auszeichner, **<from>**, einen passenden End-Auszeichner (-Tag) gibt, **</from>**. Das gesamte Konstrukt wird als Element bezeichnet.

Abschnitte können wie gezeigt in andere Abschnitte eingebettet werden. Ein XML-Dokument bildet immer eine Baumstruktur. Das erste Element in der obigen Abbildung zeigt, dass es sich um ein XML-Dokument handelt.

Zusätzlich zu Tags unterstützt XML Attribute, die zusätzliche Informationen zu dem Element liefern, zu dem sie gehören. Ein Attribut besteht aus einem durch ein Gleichheitszeichen getrennten Begriffspaar. Zum Beispiel:

<person *gender=”female”*>

Das Attribut ist in den Klammern des Elements enthalten. Anstatt ein Attribut zu verwenden, kann dieselbe Information als ein Element verwendet werden. Zum Beispiel enthalten die beiden folgenden Beispiele genau die gleichen Informationen:

```
<person gender="female">
  <firstname>Anna</firstname>
  <lastname>Smith</lastname>
</person>

<person>
  <gender>female</gender>
  <firstname>Anna</firstname>
  <lastname>Smith</lastname>
</person>
```

Abb. 15: Vergleich Attribut und Element

Attribute sind nicht so flexibel wie Elemente. Beispielsweise betont das W3C, das Kontrolle über den XML-Standard hat, die folgenden Punkte:

- Attribute dürfen nicht mehrere Werte enthalten (im Gegensatz zu Elementen)

- Attribute dürfen keine Baumstrukturen enthalten (im Gegensatz zu Elementen)
- Attribute sind nicht einfach erweiterbar (für zukünftige Änderungen)

Verschiedene Computer speichern Daten auf verschiedene Art und Weise; in den meisten Fällen ist diese inkompatibel. XML ermöglicht diesen verschiedenen Computern die gemeinsame Nutzung von Daten, da die XML-Daten in einfachem Textformat (engl. plain text format) gespeichert sind. Es besteht keine Notwendigkeit für komplexe Handshakes zwischen Computern, da sie über Textdateien kommunizieren können.

XML trennt Daten und deren Darstellung. Daher können die gleichen XML-Daten beliebig dargestellt werden.

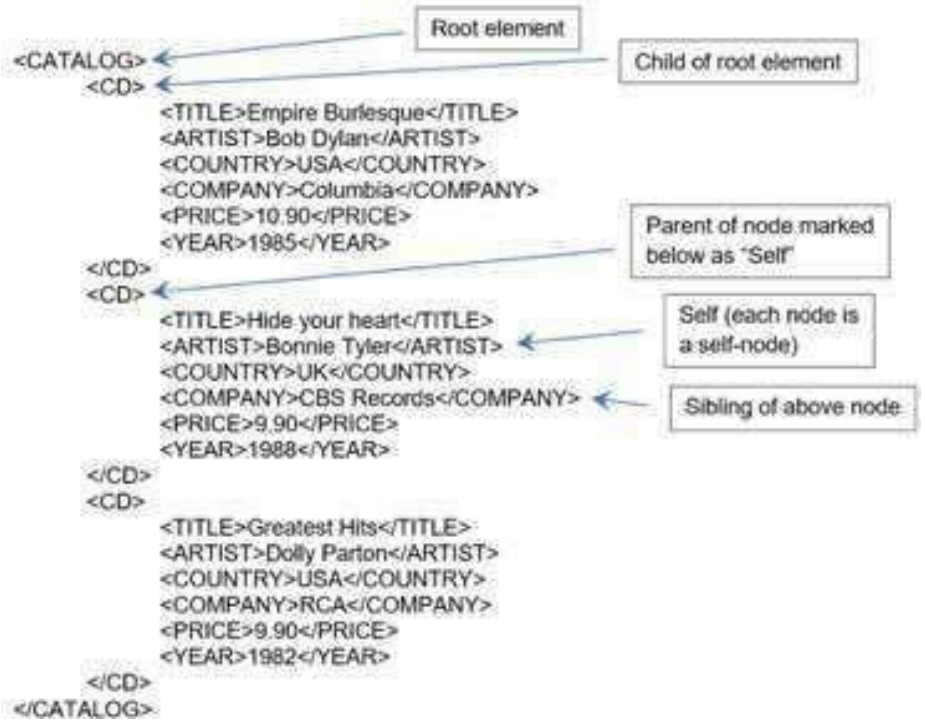

Abb. 16: XML Datenbank (Auszug)

Da XML immer eine Baumstruktur beschreibt, lassen sich bestimmte Beziehungen zwischen Elementen definieren. Diese Beziehungen können wie folgt definiert werden:

- Der **aktuelle** Knoten ist ein beliebiger von uns gewählter Knoten. Alle anderen Beziehungen stammen von diesem gegenwärtigen Knoten ab. In der Abbildung wurde „Bonnie Tyler“ als aktueller Knoten gewählt. Alle Beziehungen beziehen sich dann auf diesen Knoten.
- Jeder Knoten kann sich selbst identifizieren (als **Selbst**).
- Ein übergeordneter Knoten (auch Elter-Knoten bzw. Vorgänger, engl. **parent** node) ist in der Hierarchie immer eine Ebene höher als der aktuell ausgewählte Knoten. Jeder Elementknoten und jedes Attribut hat genau ein übergeordnetes Element (mit Ausnahme des Wurzelelements).
- Ein untergeordneter Knoten (auch Kind-Knoten bzw. Nachfolger, engl. **child** node) befindet sich in der Hierarchie immer eine Ebene unter seinem übergeordneten Element.

- Ein **Geschwister**knoten (engl. **sibling** node) befindet sich auf der gleichen Ebene wie der aktuelle Knoten unter demselben Elter-Knoten.
- **Vorfahren** (engl. **ancestor** node) sind Knoten in einem direkten Pfad vom aktuellen Knoten aufwärts über Eltern-, Großeltern-, Urgroßeltern-Knoten usw.
- **Nachfahren** (engl. **descendent** node) sind Knoten, die sich in einem direkten Pfad in der Hierarchie unterhalb des aktuellen Knoten befinden (d.h. ein Kind, ein Kind eines Kindes, usw.).

2.2 XPath und die Suche in HTML-Dokumenten

Wie bereits erwähnt, muss ein Testautomatisierer, um mit Selenium zu automatisieren, jedes gegebene Objekt oder Element in einem XML-Dokument finden können. Eine der leistungsstärksten Methoden zum Suchen bestimmter Elemente ist die Verwendung von XPath.

XPath verwendet Pfadausdrücke zum Identifizieren und Navigieren von Knoten in einem XML-Dokument. Da HTML ein Teilsatz von XML ist, kann XPath auch zum Durchsuchen von HTML-Dokumenten verwendet werden. Diese Pfadausdrücke beziehen sich auf die Pfadausdrücke, wie sie auch bei herkömmlichen Computerdateisystemen verwendet werden könnten und die es Benutzern ermöglichen, durch eine Ordnerhierarchie zu navigieren.

Die folgenden XPath-Ausdrücke selektieren die folgenden Nodes:

Tabelle 6: XPath-Ausdrücke

Ausdruck	Beschreibung
TheNode	Selektiert alle Elemente mit dem Namen „TheNode“
/	Selektiert aus dem Wurzelelement
//	Gibt untergeordneten (Kind-)Knoten des aktuellen Elements zurück
.	Gibt das aktuelle Element zurück
..	Selektiert den direkt übergeordneten (Eltern-)Knoten
@	Selektiert ein Attribut des aktuellen Elements

Nachfolgend ist ein Beispiel eines XML-Dokuments und einige Beispiele für die XPath-Verwendung.

```
<?xml version="1.0" encoding="UTF-8"?>
<Magazines>
    <Magazine>
        <title lang="en"">Time</title>
        <price>9.99</price>
     </Magazine>
     <Magazine>
        <title lang="en"">Newsweek</title>
        <price>8.95</price>
     </Magazine>
</Magazines>
```

Abb. 17: Beispiel einer XML-Datenbank (Auszug)

Die nachfolgende Tabelle listet einige Pfadausdrücke und deren Ergebnisse.

Tabelle 7: Pfadausdrücke

Pfadausdruck	Ergebnis
Magazines	Selektiert alle Elemente mit dem Namen „Magazines"
/Magazines	Selektiert aus dem Wurzelelement „Magazines"
Magazines/Magazine	Selektiert alle Magazine-Element von Magazines
//Magazines	Selektiert alle Magazine-Elemente im Dokument
Magazines//Magazine	Selektiert alle Magazine-Element, die Magazines nachgeordnet sind
//@lang	Selektiert alle Attribute mit dem Namen „lang"

Prädikate werden verwendet, um ein bestimmtes Element oder ein Element, das einen bestimmten Wert enthält, zu finden. Prädikate sind immer von eckigen Klammern umgeben und erscheinen direkt hinter dem Elementnamen.

Tabelle 8: Pfadausdrücke mit Prädikaten

Pfadausdruck	Ergebnis
/Magazines/Magazine [1]	Selektiert das erste Magazine-Element
/Magazines/Magazine[last()]	Selektiert das letzte untergeordnete Magazin-Element
/Magazines/Magazine[last()-1]	Selektiert das zweitletzte untergeordnete Magazin-Element
//title[@lang]	Selektiert alle Titel-Elemente mit dem Attribut „lang"

//title[@lang='en']	Selektiert alle Titel-Elemente mit dem Attribut lang=en

XPath verfügt über Platzhalter (engl. wildcards), mit denen XML-Knoten anhand bestimmter Kriterien selektiert werden können.

Tabelle 9: Platzhalter bei XPath

Wildcard	Beschreibung
*	Entspricht jedem beliebigen Element-Knoten
@*	Entspricht jedem beliebigen Attribut-Knoten
Node()	Entspricht jedem beliebigen Knoten

Es gibt eine große Vielfalt an Operatoren, die in XPath-Ausdrücken verwendet werden können.

Tabelle 10: Operatoren in XPath

Operator	**Beschreibung**	Beispiel
|	Selektiert mehrere Pfade	//Magazine | //CD
+	Addition	2 + 2
-	Subtraktion	5 – 3
*	Multiplikation	8 * 8
div	Division	14 div 2
mod	Modulus (Rest nach Division)	7 mod 3
=	gleich	price=4.35
!=	nicht gleich	price!=4.35
<	kleiner als	price<4.35
<=	kleiner oder gleich	price<=4.35
>	größer als	price>4.35
>=	größer oder gleich	price>=4.35
or	Verkettung mit oder	price>3.00 or lang="en"

and	Verkettung mit und	price>3.00 and lang="en"
not	Umkehrung	not lang="en"
||	String-Verkettung	"en" || "glish"

Ebenso stehen in XPath viele nützliche Funktionen zur Manipulation von Zeichenketten (Strings) zur Verfügung. Funktionen können mit dem Namespace-Vorzeichen (Präfix) „fn:“ aufgerufen werden. Da das Standardpräfix des Namespace jedoch „fn:“ lautet, benötigen String-Funktionen das Vorzeichen normalerweise nicht.

Tabelle 11: String-Funktionen bei XPath

Name	Beschreibung
string(arg)	Gibt den String-Wert des Arguments zurück
substring(str, start, len)	Gibt einen Teilstring eines Strings der Länge „len“ ab „start“ zurück
string-length(str)	Gibt die Länge des Strings zurück. Wenn kein Argument übergeben wurde, wird die Länge des aktuellen Knotens zurückgegeben
compare(str1, str2)	Gibt -1 zurück, wenn str1 < str2 ist, 0, wenn die Strings gleich sind, +1 wenn str1 > str2 ist
concat(str1, str2, …)	Gibt eine Verkettung aller eingegebenen Strings zurück
upper-case(str)	Konvertiert das String-Argument in Großbuchstaben
lower-case(str)	Konvertiert das String-Argument in Kleinbuchstaben
contains(str1, str2)	Gibt TRUE zurück, wenn str1 str2 enthält
starts-with(str1, str2)	Gibt TRUE zurück, wenn str1 mit str2 beginnt
ends-with(str1, str2)	Gibt TRUE zurück, wenn str1 mit str2 endet

Hier finden Sie einen nützlichen Link zum Testen von XPath-Ausdrücken:

https://www.freeformatter.com/xpath-tester.html

2.3 Cascading Style Sheets (CSS)

Es gibt viele Meinungen darüber, ob CSS eine Programmiersprache ist oder nicht. CSS steht für Cascading Style Sheets (deutsch: gestufte Gestaltungsbögen) und wird hauptsächlich verwendet, um

festzulegen, wie die verschiedenen HTML-Elemente in einem Satz von HTML-Dokumenten auf dem Bildschirm, auf Papier oder in anderen Medien wiedergegeben werden sollen. Externe CSS-Stylesheets werden in CSS-Dateien gespeichert.

HTML ist eine Auszeichnungssprache, CSS eine Stylesheet-Sprache. Auch wenn HTML und CSS sehr leistungsstarke Werkzeuge zum Anzeigen von Materialien sind, betrachten die meisten Experten sie nicht als echte Programmiersprachen.

CSS ist jedoch, wenn es beim Selenium-Testen angewendet wird, sehr nützlich, um HTML-Elemente zu finden, so dass das Testen des Browsers automatisiert werden kann.

In HTML-Dokumenten kann ein CSS auf drei verschiedene Arten verwendet werden:

1. Als externes Stylesheet: Jede HTML-Seite muss einen Verweis auf die externe Stylesheet-Datei innerhalb des Linkelements **<link>** enthalten, das in den **<head>** -Abschnitt gehört
2. Als internes Stylesheet: Wenn eine einzelne HTML-Seite einen einzigartigen Stil haben soll; Die Stile sind im Element **<style>** und innerhalb des **<head>** -Abschnitts des Dokuments definiert
3. Als Inline-Stil: gilt für ein bestimmtes Element und wird diesem Element direkt als Attribut hinzugefügt

Wenn mehrere CSS-Stile für dasselbe Element definiert sind, wird der Wert aus dem zuletzt gelesenen Stylesheet verwendet. Daher ist die Reihenfolge, in der der Stil angewendet wird, wie folgt (von oben beginnend):

1. Inline-Stil (als Attribut in einem HTML-Element)
2. Externe und interne Stylesheets, die im Head-Abschnitt definiert sind
3. Standardwert des Browsers

Ein CSS-Regelsatz besteht aus Selektor(en) und Deklarationsblock bzw. -blöcken wie folgt:

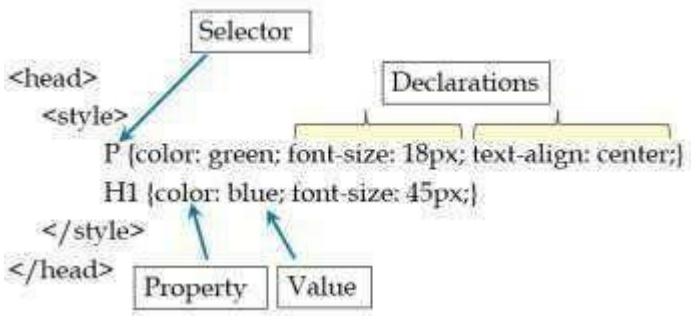

Abb. 18: CSS-Regelsatz

Jeder Selektor zeigt auf ein HTML-Element, das formatiert werden soll. Der Deklarationsblock enthält eine oder mehrere durch Semikolons getrennte Deklarationen. Jede Deklaration enthält

einen CSS-Eigenschaftsnamen und, durch Doppelpunkt getrennt, einen Wert. Deklarationsblöcke sind von geschweiften Klammern umschlossen.

CSS-Selektoren werden verwendet, um Elemente im HTML-Dokument basierend auf Elementnamen, IDs, Klassen, Attributen oder anderen spezifizierten Merkmalen zu finden. In vielen Fällen kann XPath auf die gleiche Weise verwendet werden, um bestimmte Elemente zu finden, wie in der folgenden Tabelle gezeigt:

Tabelle 12: CSS- und XPath-Selektoren

CSS	XPath	Ergebnis
div.even	//div[@class="even"]	*div*-Elemente mit einer Attributklasse *class="even"*
#login	//*[@id="login"]	Ein Element mit *id="login"*
*	//*	Alle Elemente
input	//input	Alle Eingabeelemente (*input*)
p,div		Alle *p*- und alle *div*-Elemente
div input	//div//input	Alle Eingabeelemente (*input*) in allen *div*-Elementen
div > input	//div/input	Alle Eingabeelemente (*input*), die das *div*-Element als übergeordnetes Element haben
br + p		Selektiert alle *p*-Elemente, die unmittelbar nach dem *br*-Element platziert sind
p ~ br		Selektiert alle *p*-Elemente, die unmittelbar vor dem *br*-Element platziert sind

Wie nicht anders zu erwarten, arbeiten CSS-Selektoren auch mit Attributen.

Tabelle 13: CSS-Selektoren für Attribute

CSS	Ergebnis
[lang]	Alle Elemente mit dem Attribut *lang*
[lang=en]	Alle Elemente mit dem Attribut *lang* von genau en
[lang^=en]	Alle Elemente mit dem Attribut *lang*, die mit der Zeichenfolge *en* beginnen
[lang | =en]	Alle Elemente, bei denen mit dem Attribut *lang* gleich *en* ist, oder die mit der Zeichenfolge *en* gefolgt von einem Bindestrich

[lang$=en]	Alle Elemente mit dem Attribut *lang*, die mit der Zeichenfolge *en* enden
[lang~=en]	Alle Elemente mit dem Attribut *lang*, dessen Wert eine durch Leerzeichen getrennte List von Wörtern ist, von denen eines genau Zeichenfolge *en* ist
[lang*=en]	Alle Elemente mit dem Attribut *lang*, die die Zeichenfolge *en* enthalten

Beim Umgang mit Formularelementen der Benutzeroberfläche stehen verschiedene CSS-Selektoren zur Verfügung:

Tabelle 14: CSS-Selektoren für Formularelemente

CSS	Ergebnis
:checked	Selektiert alle Elemente, die angewählt sind (für Kontrollkästchen, Optionsschaltflächen, sowie Optionen, die in den Zustand *Ein* umgeschaltet werden können)
:default	Selektiert jedes Formularelement, das in einer Gruppe verwandter Elemente standardmäßig verwendet wird
:defined	Selektiert alle Elemente, die definiert wurden
:disabled	Selektiert alle Elemente, die gesperrt sind
:enabled	Selektiert alle Elemente, die anwählbar sind
:focus	Selektiert das Element, das den Fokus erhalten hat
:invalid	Selektiert Formularelemente, die nicht validiert werden konnten
:optional	Selektiert Formularelemente, für die das Attribut *required* nicht gesetzt wurde
:out-of-range	Selektiert jedes Eingabeelement (*input*), dessen aktueller Wert außerhalb der Attribute *min* und *max* liegt
:read-only	Selektiert die Elemente, die vom Benutzer nicht bearbeitet werden können
:read-write	Selektiert die Elemente, die vom Benutzer bearbeitet werden können
:required	Selektiert Formularelemente, für die das Attribut *required* gesetzt wurde

:valid	Selektiert Formularelemente, die erfolgreich validiert werden können
:visited	Selektiert die Verweise (Links), die ein Benutzer bereits besucht hat

Schließlich gibt es viele verschiedene CSS-Selektoren, die für einen Testautomatisierer nützlich sein können. Hierzu gehören:

Tabelle 15: CSS-Selektoren, die bei der Automatisierung nützlich sind

CSS	Ergebnis
:not(<selector>)	Selektiert das Element, die nicht mit dem angegebenen Selektor übereinstimmen
:first-child	Selektiert alle Elemente, die das erste Kind ihres Eltern-Elements sind
:last-child	Selektiert alle Elemente, die das letzte Kind ihres Eltern-Elements sind
:nth-child(<n>)	Selektiert alle Elemente, die das *n*-te Kind ihres Eltern-Elements sind
:nth-last-child(<n>)	Selektiert alle Elemente, *n*-te Kind ihres Eltern-Elements sind, gezählt ab dem letzten Kind
div:nth-of-type(<n>)	Selektiert alle Elemente, die *n*-te *div* Kind-Elemente ihrer übergeordneten Eltern-Elemente sind

Das Verständnis von HTML, CSS, XML und XPath ist wesentlich, um mit Selenium erfolgreich zu arbeiten. Was in diesem Kapitel enthalten ist, ist nur die Spitze des Eisbergs.

Das W3C ist ein internationales Konsortium, das mit vielen Organisationen und mit der Öffentlichkeit arbeitet, um Webstandards, einschließlich XML und XPath, zu definieren und zu entwickeln.

Es gibt auf einer Website mit dem Namen W3Schools (die vom W3C unabhängig ist) eine hervorragende Reihe kostenloser Tutorials zu den Themen HTML, XML, XPath, CSS, und außerdem zu vielen anderen Technologien, die beim Testen mit Selenium nützlich sind.

W3Schools und alle Tutorials sind das Copyright von Refsnes Data. A4Q hat keinerlei Verbindung zu W3Schools von Refsnes Data. Hier der Link zu den Tutorials:

https://www.w3schools.com/default.asp

Kapitel 3 – Verwendung des Selenium WebDrivers

Schlüsselbegriffe

Lernziele für Verwendung des Selenium WebDrivers

STF-3.1 (K3) Geeignete Protokollierungs- und Berichterstattungsmechanismen verwenden können

STF-3.2 (K3) Zu verschiedenen URLs mit WebDriver-Befehlen navigieren können

STF-3.3 (K3) Fensterkontext in Webbrowsern mithilfe von WebDriver-Befehlen wechseln können

STF-3.4 (K3) Bildschirmfotos (Screenshots) von Webseiten mit WebDriver-Befehlen erfassen können

STF-3.5 (K4) GUI-Elemente mit verschiedenen Strategien finden können

STF-3.6 (K3) Den Zustand von GUI-Elementen mit WebDriver-Befehlen abrufen können STF-3.7 (K3) Mit GUI-Elementen mithilfe von WebDriver-Befehlen interagieren können STF-3.8 (K3) Mit Benutzeraufforderungen in Webbrowsern mit WebDriver-Befehlen interagieren können

3.1 Protokollierungs- und Berichterstattungsmechanismen

Automatisierte Testskripte sind Softwareprogramme, die Befehle auf dem SUT ausführen. Dabei simulieren sie Menschen, die Tests mit Tastatur und Maus ausführen. Innerhalb der TAS muss ein Mechanismus vorhanden sein, der die Testausführungsschicht implementiert. Eine Möglichkeit, einen solchen Mechanismus zu implementieren, besteht darin, Testautomatisierungsskripte von Grund auf neu zu schreiben. Solche Skripte können wie jedes andere Python-Skript ausgeführt werden.

Anstatt solche Skripte vollständig zu erstellen, können vorhandene Komponententest-Bibliotheken genutzt werden, die die Tests ausführen und deren Ergebnisse berichten können, z. B. *unittest* oder *pytest*. In diesem Lehrplan wird *pytest* als Testausführungsbibliothek verwendet.

Pytest ist ein Test-Framework, das es einfacher macht, Tests für Python zu schreiben und - was noch interessanter ist - auch für Selenium WebDriver, der Python verwendet. Pytest macht es einfach, kleine Tests zu schreiben, ist aber skalierbar, um das Schreiben komplexer Automatisierungssuiten zu unterstützen.

Wenn *pytest* aufgerufen wird, führt *pytest* alle Tests im aktuellen Verzeichnis oder seinen Unterverzeichnissen aus. Es sucht speziell nach allen Dateien mit den Mustern: “test_*.py” oder :*_test.py” und führt sie dann aus.

Wird pytest ohne Flags (**pytest**) durchgeführt, dann führt pytest einfach alle Tests aus, die in den darunter liegenden Verzeichnissen gefunden werden. Alternativ können Flags gesetzt werden, um das Verhalten wie folgt zu ändern:

- (**pytest -v**) Ausführlicher Modus, der vollständige Testnamen anstelle von nur einem Punkt anzeigt
- (**pytest -q**) Stiller Modus, der weniger anzeigt
- (***pytest --html=report.html***) Test(s) mit Bericht an die HTML-Datei ausführen

Tests können markiert werden, um auf besondere Weise behandelt zu werden. Wird beispielsweise (**@pytest.mark.skip**) vor eine Testspezifikation gestellt, führt pytest den Test nicht aus. Wird (**@pytest.mark.xfail**) vor eine Testdefinition gesetzt, informiert es die Laufzeit-Engine, dass der Test voraussichtlich fehlschlägt. Solche Tests werden, wenn sie bei der Ausführung fehlschlagen, nicht auf die gleiche Weise berichtet wie Tests, die fehlschlagen, bei denen aber erwartet wurde, dass sie bestanden werden.

Wenn ein manueller Tester einen Test mit Skript durchführt und dieser fehlschlägt, hat der Tester eine ziemlich gute Vorstellung davon, was passiert ist. Tester wissen dann genau, welche Schritte zur Fehlerwirkung geführt haben, was genau während der Fehlerwirkung passiert ist, welche Daten verwendet wurden und wie die Fehlerwirkung aussah. Beim explorativen Testen, haben Tester zwar kein Skript für die exakten Schritte, sie haben aber eine allgemeine Theorie der Ausführung und die begrenzte Fähigkeit, Fehlerwirkungen zurückzuverfolgen und zu sehen, warum diese aufgetreten sind.

Bei der Testautomatisierung trifft fast nichts davon zu.

In der Automatisierung ist die Fehlermeldung, die vom Werkzeug aufgezeichnet wird, häufig nicht ausreichend für jemanden, der eine fehlgeschlagene Ausführung überprüft, um zu verstehen, was passiert ist. Häufig hat die Fehlermeldung keinen Kontext; der aufgezeichnete Fehler hat möglicherweise nichts mit der tatsächlichen Fehlerwirkung zu tun, die die Ausführung des Tests gestoppt hat.

Beispiel: Angenommen, eine Fehlerwirkung tritt in Schritt ***N*** auf. Je nachdem wie die Fehlerwirkung aufgetreten ist, könnte das Ergebnis bedeuten, dass die Schnittstelle des SUT nicht im korrekten Zustand belassen wird. Wenn das Skript versucht, Schritt ***N+1*** auszuführen, schlägt der Schritt fehl, da sich das SUT nicht im richtigen Zustand befindet, um den Schritt auszuführen. Das Protokoll meldet, dass der Schritt ***N+1*** fehlgeschlagen ist.

Die Fehlersuche setzt dann an der falschen Stelle an. Da der Test Analyst, der den automatisierten Test (normalerweise im Batchmodus) ausgeführt hat, möglicherweise keine direkte Kenntnis über den ausgeführten Test hat, kann es schwierig (und zeitintensiv) sein, zu ermitteln, ob es sich tatsächlich um einen SUT-Fehler oder einen Automatisierungsfehler handelt.

In der Automatisierung muss das aber nicht zwingend so sein. Eine gute Protokollierung kann dem Test Analyst helfen, schnell zu bestimmen, ob der Fehler vom SUT verursacht wurde oder nicht. Es ist wichtig, dass bei der Testautomatisierung auf die Protokollierung des Tests geachtet wird. Gute Protokollierung kann den Unterschied zwischen einem fehlgeschlagenen Automatisierungsprojekt und einem Projekt ausmachen, das der Organisation einen großen Mehrwert liefert.

Protokollierung ist eine Möglichkeit, Ereignisse während der Ausführung zu verfolgen. Testautomatisierer können den Testskripten Protokollaufrufe hinzufügen, um Informationen aufzuzeichnen, um dadurch das Verständnis der Ausführung zu erleichtern. Protokollierung kann vor dem Ausführen eines Schritts und nach dem Ausführen eines Schritts erfolgen, und kann die Daten enthalten, die im Schritt verwendet wurden, und das Verhalten, das als Ergebnis des Schritts aufgetreten ist. Je mehr Protokollierung erfolgt, desto besser verständlich ist das Ergebnis des Tests.

Bei Testen sicherheitskritischer oder einsatzkritischer Software kann eine detaillierte Protokollierung für Auditzwecke ohnehin zwingend erforderlich sein.

Die Protokollierung kann nach Bedarf erfolgen. Dies bedeutet: Daten können an einem Zwischenspeicherort gespeichert werden und nur im formalen Protokoll abgelegt werden, wenn eine Fehlerwirkung auftritt oder wenn ein vollständiger Datensatz der Fehlerbehebung benötigt wird. Die Protokollierung jedes einzelnen Schritts ist möglicherweise nicht wünschenswert, wenn der Test wie erwartet ausgeführt wird. Im Falle einer Fehlerwirkung können diese Protokollierungsdaten jedoch Stunden der Fehlersuche einsparen, wenn Schritt für Schritt aufgezeichnet wurde, was genau während der Ausführung passiert ist und welche Daten verwendet wurden.

Nicht jedes Automatisierungsprojekt benötigt eine so umfassende Protokollierung. Oft beginnt ein Automatisierungsprojekt mit einem oder zwei Testautomatisierern, die eigene Skripte erstellen und ausführen. In einem solchen Fall kann das oben beschriebene Protokollieren als übertrieben angesehen werden. Wenn kleine Automatisierungsprojekte jedoch erfolgreich sind, dann will das Management sicherlich viel mehr Automatisierung haben. Je größer der Erfolg, desto höher die Nachfrage nach mehr.

Das bedeutet, dass das Projekt irgendwann eine Größe erreichen wird, bei der die beschriebene Protokollierung tatsächlich erforderlich ist. Zu diesem Zeitpunkt müssten bestehende Tests mit besserer Protokollierung nachgerüstet werden. Es ist effektiver und effizienter, von Anfang an umfangreich zu protokollieren.

Python verfügt über sehr robuste Protokollfunktionen, die mit WebDriver verwendet werden können. Diese erlauben, dass jederzeit in einem automatisierten Skript Protokollierungsaufrufe hinzugefügt werden können, um die jeweiligen gewünschten Informationen zu melden. Diese können allgemeine Informationen zum späteren Nachverfolgen des Tests umfassen (z. B. „Ich klicke gerade auf die Schaltfläche XYZ“), Warnungen über Vorkommnisse, die noch nicht als Fehlerwirkungen einzustufen sind (z.B. „Datei <ABC> öffnen dauerte länger als erwartet“) oder tatsächliche Fehler, durch die eine Ausnahme ausgelöst wird, die das Testen beendet, wie z. B Umgebung aufräumen und mit dem nächsten Test fortfahren.

Die Python-Protokollierungsbibliothek verfügt über fünf verschiedene Protokollierungsgrade für Protokollmeldungen, die gespeichert werden können. Diese sind wie folgt (vom niedrigsten bis zum höchsten Grad):

- DEBUG: zum Diagnostizieren von Problemen
- INFO: für die Bestätigung, dass es funktioniert hat
- WARNING: etwas Unerwartetes ist aufgetreten, ein potenzielles Problem
- ERROR: ein schwerwiegendes Problem ist aufgetreten
- CRITICAL: ein kritisches/fatales Problem ist aufgetreten

Wenn ein Protokoll auf der Konsole oder in einer Datei ausgedruckt wird, kann mithilfe des Protokollierungsgrads eine dieser fünf Einstellungen (oder benutzerdefinierte Einstellungen) vorgenommen werden, sodass nur die gewünschten Protokollmeldungen angezeigt werden. Wenn für die Konsole beispielsweise der Protokollierungsgrad WARNING gesetzt ist, werden dem Benutzer keine DEBUG- oder INFO-Meldungen angezeigt, sondern nur die Protokollmeldungen der Grade WARNING, ERROR und CRITICAL. Hier ein Codebeispiel für die Protokollierung:

```
import logging
logging.basicConfig(level=logging.WARNING)
# default logging level is WARNING

logging.info("Hello world.")
logging.warning("Title: %d Dalmatians" % 101)
logging.debug("Title: %s" % "101 Dalmatians")
```

Abb. 19: Beispiel für Protokollierung

Da der Standardprotokollierungsgrad WARNING ist, wird Folgendes an die Konsole zum Ausdruck gegeben:

WARNING:root: Title: 101 Dalmatians

Beim Ausführen von Testfällen ist es wichtig, dass tatsächlich etwas getestet wird. Jeder Testfall muss erwartete Ergebnisse und Verhaltensweisen haben, die überprüft werden können. Python bietet hierfür die Zusicherungen (engl. assertions), mithilfe derer geprüft werden kann, ob die richtigen Daten oder das richtige Verhalten aufgetreten sind.

Eine Assertion ist eine Anweisung, von der erwartet wird, dass sie an einem bestimmten Punkt im Skript zutrifft. Wenn Sie beispielsweise einen Taschenrechner testen, können Sie zwei plus zwei addieren und dann bestätigen, dass die Antwort gleich vier sein sollte. Wenn die Berechnung aus irgendeinem Grund nicht korrekt ist, löst das Skript eine Ausnahme aus.

Die Syntax ist wie folgt:

assert sumVariable==4, “sumVariable should equal 4.”

Beim Selenium WebDriver besteht ein Testfallschritt häufig aus folgenden Aktionen:

1. Ein Webelement auf einem Bildschirm identifizieren
2. Das Webelement bearbeiten
3. Sicherstellen, dass das Richtige passiert ist.

Zur Aktion (1): Wenn das Element nicht gefunden oder im falschen Zustand gefunden wird, wird eine Ausnahme ausgelöst.

Zur Aktion (2): Wenn der Versuch, auf das Webelement einzuwirken, fehlschlägt, wird eine Ausnahme ausgelöst.

Zur Aktion (3): Es kann eine Zusicherung bzw. Assertion verwendet werden, um zu prüfen, ob das erwartete Verhalten aufgetreten ist oder ein Wert empfangen wurde.

Dies unterscheidet sich nicht wesentlich von der Art und Weise, wie ein manueller Tester diesen Schritt ausführen würde und ermöglicht es der Person, die einen fehlgeschlagenen Test bewertet, zu verstehen, was aufgetreten ist.

Berichterstellung wird oft mit Protokollierung in Zusammenhang gebracht, ist jedoch etwas Anderes. Die Protokollierung liefert den Test Analysten, die die Testsuite ausgeführt haben, und den Testautomatisierern, die bei Bedarf für die Wartung der Tests verantwortlich sind, Informationen über die Automatisierungsausführung. Die Berichterstattung liefert diese Informationen und wahrscheinlich weitere kontextbezogene Informationen an die verschiedenen (externen und übergeordneten) Stakeholder, die diese haben wollen oder benötigen.

Es ist wichtig, dass die Testautomatisierer feststellen, wer Berichte erwartet oder benötigt, und welche Informationen die einzelnen Stakeholder interessieren. Das Versenden der unbearbeiteten Protokolle an die Stakeholder würde wahrscheinlich viele von ihnen überfordern, da sie mit detaillierten Informationen überschüttet werden, die sie weder brauchen noch haben wollen.

Eine Möglichkeit besteht darin, die Berichte aus den Protokollen und anderen Informationen zu erstellen und den Stakeholdern zum Download zur Verfügung zu stellen, damit diese sie herunterladen können, wann sie wollen. Die andere Möglichkeit besteht darin, die Berichte zu erstellen und diese, sobald sie fertig sind, an die Stakeholder zu senden, die sie haben möchten.

Auf jeden Fall sollten Berichterstellung und -verteilung automatisiert werden, sofern dies möglich ist, um damit eine weitere manuelle Aufgabe einzusparen.

Berichte sollten eine Zusammenfassung mit einem Überblick über das getestete System, die Umgebung(en), in denen die Tests ausgeführt wurden, und die während des Testens erzielten Ergebnisse enthalten. Wie bereits erwähnt, wünscht jeder Stakeholder eventuell eine andere Sicht auf die Ergebnisse, und das Automatisierungsteam sollte die Berichte entsprechend dieser Bedürfnisse bereitstellen. Häufig möchten Stakeholder Trends der Tests sehen, und nicht nur eine Momentaufnahme. Da jedes Projekt wahrscheinlich unterschiedliche Stakeholder mit unterschiedlichen Bedürfnissen hat, ist es umso besser, je mehr die Berichterstattungsaufgaben automatisiert sind.

3.2 Zu verschiedenen URLs navigieren

3.2.1 Eine automatische Testsitzung starten

Es gibt viele verschiedene Browser, die Sie möglicherweise testen möchten. Während die grundlegende Aufgabe eines Browsers darin besteht, dass Sie verschiedene Webseiten anzeigen und mit ihnen interagieren können, verhält sich jeder Browser wahrscheinlich ein wenig anders als andere Browser.

In der Anfangszeit des Internets funktionierten manche Seiten nur in Netscape und andere funktionierten nur korrekt in IE. Zum Glück sind die meisten dieser Probleme schon längst Vergangenheit, obwohl es zwischen Browsern immer noch Inkompatibilitäten geben kann. Wenn eine Organisation eine Website bereitstellen möchte, müssen die Tests dennoch mit verschiedenen Browsern durchgeführt werden, um die Kompatibilität sicherzustellen.

In Kapitel 1.4 wurde bei der Einführung von Selenium WebDriver erwähnt, dass verschiedene Browser unterschiedliche Treiber benötigen, um sicherzustellen, dass die automatisierten Testskripte mit den verschiedenen Browsern funktionieren. Nachfolgend ist die Liste der Browser aus Punkt 1.4:

- Chrome (chromedriver.exe)
- Internet Explorer (IEDriverServer.exe)
- Edge (MicrosoftWebDriver.msi)
- Firefox (geckodriver.exe)
- Safari (safaridriver)
- HtmlUnit (HtmlUnit driver)

Wenn beispielsweise der Chrome-Browser getestet werden soll, muss zunächst der Chrome-Treiber (mit dem Namen ***chromedriver.exe***) heruntergeladen und diese Datei an einem bekannten Ort auf der Test-Workstation installiert werden. Dies geschieht häufig durch Bearbeitung der ***Path***-Umgebungsvariablen, damit das Betriebssystem diese bei Bedarf findet.

Wie bei vielen Technologien ändern sich diese Treiberdateien und die für ihre Verwendung erforderlichen Informationen schnell. Die Informationen in diesem Lehrplan wurden überprüft und sind zum Zeitpunkt der Erstellung des Lehrplans gültig. In der Zukunft ist alles möglich! Testautomatisierer sollten sich daher mit der Hilfedokumentation auf den verschiedenen Support-Websites für die verschiedenen Browser und mit der Selenium WebDriver-Support-Seite vertraut machen.

Um mit Webseiten zu arbeiten, muss zuerst ein Webbrowser geöffnet werden. Dies kann durch Erstellen eines ***WebDriver-Objekts*** erfolgen. Durch das Instanziieren des ***WebDriver-Objekts*** wird die Programmierschnittstelle zwischen einem Skript und dem Webbrowser erstellt. Diese wird auch einen WebDriver-Prozess ausführen, wenn dies für einen bestimmten Webbrowser erforderlich ist. In der Regel wird diese auch den Webbrowser selbst ausführen.

```
from selenium import webdriver
driver = webdriver.Chrome()
```

Abb. 20: Code zum Erstellen eines WebDriver-Objekts

Dieser Code funktioniert erst, nachdem die ***chromedriver.exe*** wie oben beschrieben installiert wurde.

Beim Erstellen eines WebDriver-Objekts wird ein Webbrowser mit einer leeren Seite gestartet. Um zur gewünschten Seite zu navigieren, sollte die Funktion ***get()*** verwendet werden. Es ist zu beachten, dass auf die Funktionalität des Treiberobjekts mithilfe der Punktnotation zugegriffen wird, da Python eine objektorientierte Sprache ist.

```
driver.get('https://www.python.org')
```

Abb. 21: Navigation zur URL

Es ist auch möglich, ein WebDriver-Objekt an einen vorhandenen Webdriver-Prozess anzubinden oder einen WebDriver-Prozess zu erstellen und diesen über Selenium RemoteWebDriver an einen bereits laufenden Browser anzubinden. Die Behandlung dieser Vorgänge ist jedoch nicht Gegenstand dieses Lehrplans.

Nachdem eine Seite geöffnet wurde oder zu einer anderen Seite navigiert wurde, ist es ratsam zu prüfen, ob die richtige Seite geöffnet wurde. Hierfür verfügt das WebDriver-Objekt über zwei nützliche Attribute: ***current_url*** und ***title***. Durch die Überprüfung der Werte dieser Felder kann das Skript die aktuelle Seite verfolgen.

```
assert driver.current_url == 'https://www.python.org/', ErrMsg
assert driver.title == 'Welcome to Python.org', ErrMsg
```

Abb. 22: Korrekte Seite prüfen (assert)

3.2.2 Navigieren und Seiten auffrischen

Zur Simulierung der Vorwärts-/Rückwärts-Navigation im Webbrowser sollten die Methoden ***back()*** und ***forward()*** des WebDriver-Objekts verwendet werden, die entsprechende Befehle an den WebDriver senden. Diese Methoden übergeben keine Argumente.

```
driver.back()
driver.forward()
```

Abb. 23: Browser-Historien navigieren

Es ist auch möglich, durch Aufrufen von ***refresh()*** vom WebDriver aus die aktuelle Seite des Webbrowsers aufzufrischen.

```
driver.refresh()
```

Abb. 24: Auffrischen des Browsers

3.2.3 Browser schließen

Am Ende des Tests müssen der Webbrowser-Prozess und alle weiteren ausgeführten Treiberprozesse geschlossen werden. Wenn der Browser nicht geschlossen wird, bleibt er auch nach

Abschluss des Testens geöffnet. Es empfiehlt sich, die Umgebung auf denselben Status zu setzen, in dem die Tests gestartet wurden. Dadurch kann eine unbegrenzte Anzahl von Tests unbeaufsichtigt ausgeführt werden.

Der von WebDriver gesteuerte Browser wird durch Aufrufen der ***quit()***-Methode des WebDriver-Objekts geschlossen.

```
driver.quit()
```

Abb. 25: Schließen des gesamten Browsers

Die Funktion ***quit()*** muss in dem Teil des Testskripts platziert werden, der unabhängig vom Ergebnis des Tests ausgeführt wird. Ein häufiger Fehler besteht darin, das Schließen des Browsers wie einen der üblichen Testschritte zu behandeln. Wenn der Test dann fehlschlägt, stoppt eine Testlaufbibliothek die Ausführung der Testschritte und der Schritt, der für das Schließen des Browsers verantwortlich ist, wird nicht mehr ausgeführt. Üblicherweise haben Testbibliotheken ihre eigenen Mechanismen zum Definieren und Ausführen von Teardown-Code (z. B. die ***tearDown()***-Methoden des Python-***Unittest***-Moduls). Weitere Informationen zu diesem Thema finden Sie in der Dokumentation der von Ihnen verwendeten Bibliothek.

Wenn im getesteten Browser mehrere Registerkarten geöffnet sind, lässt sich bestimmen, welche der Registerkarten geöffnet ist, indem der Titel des aktuellen Fensters mit ausgewählt wird:

```
cur_win = driver.title
```

Abb. 26: Titel des aktiven Fensters erhalten

Um nur die Registerkarte im Browser (und nicht den gesamten Browser) zu schließen, verwenden Sie die obige Funktion und bewegen Sie sich durch die Registerkarten, bis das Fenster erreicht ist, das geschlossen werden soll. Der Tab wird mit ***close()*** geschlossen. Wenn die letzte geöffnete Registerkarte geschlossen ist, wird der Browserprozess automatisch beendet.

```
driver.close()
```

Abb. 27: Aktive Registerkarte schließen

Diese Methode benötigt keine Parameter und schließt die aktive Registerkarte. Es ist wichtig, den Kontext auf die gewünschte Registerkarte zu setzen, damit diese Registerkarte geschlossen werden kann. Sobald das Schließen erfolgt, löst das Aufrufen aller WebDriver-Befehle außer des Befehls ***driver.switch_to.window()*** die Ausnahme ***NoSuchWindowException*** aus, da die Objektreferenz immer noch auf das Fenster verweist, das nicht mehr existiert. Es ist erforderlich, vor dem Aufrufen von WebDriver-Methoden proaktiv zu einem anderen Fenster zu wechseln. Nach dem Wechseln von Registerkarten lässt sich durch die Überprüfung des Titels feststellen, welcher Tab jetzt aktiv ist. Die Steuerung mehrerer Tabs wird weiter unten behandelt.

3.3 Fensterkontext wechseln

Beim Testen komplexerer Anwendungen oder Szenarien ist es manchmal erforderlich, den aktuellen Kontext der zu testenden GUI zu wechseln. Dies kann notwendig sein, weil das Testergebnis in einem anderen System überprüft oder ein Testschritt in verschiedenen Anwendungen ausgeführt werden soll.

In manchen Fällen ist die GUI der getesteten Anwendung so komplex, dass zwischen Frames oder Fenstern gewechselt werden muss.

Ein Webbrowser muss keinen Fokus haben, um Selenium Webdriver-Befehle ausführen zu können, da das WebDriver-Protokoll auf HTTP-Kommunikation basiert. Dadurch kann WebDriver mehrere Tests gleichzeitig ausführen oder mehrere Browser in einem Skript steuern.

Das Wechseln des Skriptkontexts kann auf drei Arten erfolgen:

- Browser wechseln
- Fenster/Registerkarten innerhalb eines Browsers wechseln
- Frames innerhalb einer Seite wechseln

Um zwei Browser zu öffnen, müssen zwei WebDriver-Objekte erstellt werden. Jedes WebDriver-Objekt steuert einen Browser. Jedes WebDriver-Objekt wird dann im Testskript gemäß den Schritten des automatisierten Testfalls verwendet. WebDriver-Objekte können den gleichen Typ von Webbrowsern (z.B. beide Chrome, beide Firefox) oder verschiedene Typen (z.B. einer Chrome und der andere Firefox) steuern. Bei Bedarf können auch mehr als zwei Browser geöffnet werden. Wichtig: Jeder Browser wird von einem Webdriver-Objekt gesteuert. Und alle Browser müssen am Ende des Tests geschlossen werden.

Das Öffnen mehrerer Registerkarten in einem einzigen Browser kann kompliziert werden, da verschiedene Browser und Betriebssysteme unterschiedliche Methoden nutzen. Bei Windows im Chrome-Browser besteht die Möglichkeit, einen Funktionsaufruf im JavaScript-Code wie folgt auszuführen:

```
driver.execute_script("$(window.open(''))")
```

Abb. 28: Aufruf zum Öffnen einer neuen Registerkarte in JavaScript

Eine weitergehende Behandlung des Öffnens mehrerer Tabs in einem Webbrowser ist nicht Gegenstand dieses Lehrplans.

Um zwischen geöffneten Registerkarten in einem Browser zu wechseln, muss zunächst die Liste aller geöffneten Tabs (Fenster) aufgerufen werden. Diese Liste befindet sich im Attribut ***window_handles*** des WebDriver-Objekts. Dabei ist zu beachten, dass die Reihenfolge im Array ***window_handles*** von der Reihenfolge der Registerkarten im Browser abweichen kann.

Mithilfe des folgenden Codes können alle Fenster-Tabs durchlaufen werden:

```
for handle in driver.window_handles:
    driver.switch_to.window(handle)
```

Abb. 29: Geöffnete Fenster durchlaufen

Der sicherste Methode, um festzustellen, welches Fenster das aktuell geöffnete Fenster ist, ist die Verwendung des oben beschriebenen Attributs ***driver.title***.

Der folgende Python-Code öffnet die Startseite von Python, öffnet einen neue Registerkarte, öffnet dann die Startseite von Perl in der zweiten Registerkarte und wechselt dann den Webbrowser zurück auf die Registerkarte mit der Startseite von Python:

```
From selenium import webdriver
driver = webdriver.Chrome()
driver.get('https://python.org')
driver.execute_script("$(window.open(''))")
driver.switch_to.window(driver.window_handles[1])
driver.get('https://perl.org')
driver.switch_to.window(driver.window_handles[0])
```

Abb. 30: Zwei Registerkarten öffnen und zwischen diesen wechseln

Bitte beachten: Dieser Code ist für einen tatsächlichen Produktionstest nicht sicher, da er davon ausgeht, dass die Handles der Registerkarte in Ordnung sind. Er dient nur zu Referenzzwecken. Um diesen Code tatsächlich bei der Ausführung zu beobachten, sollten einige sleep()-Funktionen hinzugefügt werden, damit er nicht zu schnell wechselt.

Die dritte Situation, in der der Kontext in einem Webbrowser gewechselt wird, ist das Wechseln zwischen Frames. Dies ist häufig erforderlich. Wenn der Kontext nicht auf einen bestimmten Frame gewechselt wird, können auch keine Elemente in diesem Frame gefunden werden. Es wäre demzufolge nicht möglich, Tests für die Elemente in diesem Frame zu automatisieren.

Um den Kontext zu einem bestimmten Frame zu wechseln, muss dieser zuerst gefunden werden. Wenn die ***ID*** des Frames bekannt ist, kann wie folgt direkt zu diesem Frame gewechselt werden:

```
driver.switch_to.frame('foo')
```

Abb. 31: Frame wechseln

In diesem Fall ist ***foo*** die ***ID*** des Frames, zu dem der Kontext gewechselt werden soll.

Wenn der Frame keine ID hat oder wenn eine andere Strategie verwendet werden soll, um den Frame zu finden, kann der Kontext in zwei Schritten gewechselt werden. Wie bereits erwähnt, besteht der erste Schritt darin, den Frame als ein WebElement zu finden, und im zweiten Schritt dann zu dem gefundenen Frame zu wechseln. Das Wechseln erfolgt in diesem Fall auf die gleiche Weise wie das Wechseln über ID, allerdings ist das Argument für eine Funktion das gefundene WebElement.

Das nachfolgende Beispiel zeigt einen Abschnitt des Python-Codes zum Wechseln des Frames, wobei hier das Finden des Frames der erste Schritt ist:

```
frm_message = driver.find_element_by_name('message')
driver.switch_to_frame(frm_message)
```

Abb. 32: Frame finden, dann Frame wechseln

Bitte beachten: Während des Aufrufs von ***switch_to.frame()*** werden keine Apostrophe oder Anführungszeichen verwendet, da die Variable ***frm_message*** als Variablenargument und nicht als Zeichenfolge mit der ID des Frames übergeben wird.

Der Code, um wieder zum übergeordneten Frame zurückzuwechseln, ist wie folgt:

```
driver.switch_to.parent_frame()
```

Abb. 33: Zum übergeordneten Frame zurückwechseln

Mit dem folgenden Code kann auch zur gesamten Seite zurückgewechselt werden:

```
driver.switch_to.default_content()
```

Abb. 34: Zum Hauptfenster wechseln

Neben dem Wechseln eines Fensterkontexts oder eines Frame-Kontexts des Webbrowsers kann auch die Fenstergröße eines Webbrowsers mit dem Selenium WebDriver-Framework manipuliert werden. Der Begriff Fenster wird hier als Betriebssystembegriff des gesamten Fensters verwendet und nicht nur als Registerkarte innerhalb eines Webbrowsers.

Selenium ermöglicht es auch, den Webbrowser zu minimieren und zu maximieren und in den Vollbildmodus zu versetzen. Die Python-Bindungen für diese Funktionalitäten sind wie folgt:

```
maximize: driver.maximize_window()
minimize: driver.minimize_window()
fullscreen: driver.fulllscreen_window()
```

Abb. 35: Fenstergröße des Browsers ändern

Diese Funktionen erfordern keine Argumente, da sie auf dem vom ***driver***-Objekt gesteuerten Webbrowser ausgeführt werden.

3.4 Screenshots von Webseiten erfassen

Wenn Tester Testfälle manuell ausführen, interagieren sie visuell mit den GUI-Objekten auf dem Bildschirm. Wenn ein Testfall fehlschlägt, ist es für sie offensichtlich, weil das, was auf dem Bildschirm angezeigt wird, nicht mehr korrekt ist.

Beim automatisierten Testen ist das nicht ganz so einfach.

Testautomatisierungsskripte können das Layout und das Erscheinungsbild von Webseiten nicht zuverlässig überprüfen. Es ist daher häufig sinnvoll, Screenshots des Bildschirms oder eines bestimmten Bildschirmelements zu erstellen und diese im Protokoll oder an einem bekannten Ort zu speichern, damit sie zu einem späteren Zeitpunkt angezeigt werden können, z.B.

- Wenn das automatisierte Skript feststellt, dass eine Fehlerwirkung aufgetreten ist
- Wenn der Test sehr visuell ist und nur durch Betrachten des Bildschirmbildes entschieden werden kann, ob er bestanden oder nicht bestanden wurde
- Beim Umgang mit sicherheits- oder einsatzkritischer Software, für die evtl. ein Audit des Tests vorgeschrieben ist
- Beim Testen der Konfiguration von verschiedenen Systemen

Um die Informationen, die die Bildschirmfotos liefern, optimal zu nutzen, müssen sie im richtigen Moment aufgenommen werden. Häufig werden die Teile von Testskripten, die Screenshots erstellen, unmittelbar nach Testschritten platziert, die die Benutzeroberfläche steuern, oder in den Teardown-Funktionen. Da sie jedoch ein wertvolles Werkzeug zum Verständnis der automatisierten Tests sein können, können sie an beliebiger Stelle aufgenommen werden.

Ein weiterer wichtiger Aspekt ist, dass die Dateien mit eindeutigen Namen und Speicherorten benannt werden, damit die zu einem früheren Zeitpunkt erfassten und gespeicherten Screenshots nicht beim automatisierten Testlauf überschrieben werden. Hierfür steht eine Vielzahl von verschiedenen Möglichkeiten zur Verfügung; diese sind jedoch nicht Gegenstand dieses Lehrplans.

Screenshots können mit verschiedenem Umfang erstellt werden und entweder die gesamte Browserseite oder ein einzelnes Element auf der Browserseite erfassen[1]. Beide verwenden zwar den gleichen Aufruf für die Methode, werden jedoch aus verschiedenen Kontexten heraus aufgerufen.

[1] Die Dokumentation der WebDriver Python-Bibliothek Version 3.13.0 besagt, dass die Funktionalität zum Erstellen eines Screenshots eines WebElements durch Aufruf der Funktion *WebElement.screenshot('Filename.png')* verfügbar ist. Die Autoren dieses Lehrplans konnten nicht verifizieren, dass diese Funktion im Chrome 67-Browser mit dem Chrometreiber 2.36 funktioniert, obwohl im WebDriver W3C technisch empfohlen. Falls diese Funktionalität benötigt wird, gibt es Workarounds, die auf mehreren Websites dokumentiert sind. Versuchen Sie mit einer Google-Suche

Der Methodenaufruf erfolgt mit ***screenshot()***. Das folgende Python-Beispiel zeigt, wie ein Screenshot einer ganzen Seite erstellt und an einem bestimmten Ort abgelegt wird:

```
driver.get_screenshot_as_file('C:\\temp\\screenshot.png')
```

Abb. 36: Screenshot einer ganzen Seite speichern

Das folgende Beispiel zeigt, wie ein Screenshot eines Elements erfasst und an einem bestimmten Ort abgelegt wird:

```
ele = driver.find_element_by_id('btnLogin')
ele.screenshot('c:\\temp\\element_screenshot.png')
```

Abb. 37: Screenshot eines Elements speichern

Das Aufnehmen eines Screenshots im Browser kann etwas Zeit in Anspruch nehmen, da die Workstation viele Verarbeitungsschritte durchlaufen muss. Wenn sich der Status der GUI schnell ändert (z. B. durch gleichzeitiges Ausführen von AJAX-Bibliotheken), zeigt der Screenshot möglicherweise nicht genau den erwarteten Zustand der Seite oder des Elements. Auch hier gibt es Abhilfe; dies ist jedoch nicht Gegenstand dieses Lehrplans.

Wenn Sie einen Screenshot aufnehmen, diesen aber nicht als eine Datei haben möchten, dann gibt es im WebDriver Alternativen. Angenommen, statt einer HTML-Datei für die Protokollierung wird stattdessen die Protokollierung in einer Datenbank verwendet. Wenn nun ein Screenshot des Bildschirms oder eines Elements erfasst wird, soll damit keine ****.png***-Datei erstellt werden, sondern es soll direkt in einen Satz der Datenbank gestreamt werden. Die folgenden Aufrufe würden ein Image als Base64-codierte String-Version des Snapshots erzeugen. Die base64-codierte Version ist wesentlich sicherer als eine Binärdatei (wie z. B. eine ****.png***-Datei). Der erste Aufruf im Code-Beispiel erfasst das gesamte Fenster, der zweite ein einzelnes Element:

```
img_b64 = driver.get_screenshot_as_base64()
img_b64 = element.screenshot_as_base64
```

Abb. 38: Base64-String aus Image erzeugen

Wenn Sie eine binäre Zeichenfolge abrufen möchten, die ein Image darstellt, ohne diese in einer Datei zu speichern, können Sie mit den folgenden Aufrufen die Binärdarstellung einer ****.png***-Datei zurückerhalten. Der erste Aufruf betrifft das Image eines gesamten Bildschirms, der zweite ein Bild eines einzelnen Elements:

---

nach „Python WebDriver Screenshot von WebElement“. Diese Funktion scheint beim Testen des Firefox-Browsers zu funktionieren.

```
png_str = driver.get_screenshot_as_png()
png_str = element.screenshot_as_png()
```

Abb. 39: Binär-Zeichenfolge aus Image erzeugen

3.5 GUI Elemente finden

3.5.1 Einführung

Um die meisten Vorgänge mit WebDriver auszuführen, müssen zunächst die UI-Elemente (Benutzeroberflächenelemente) gefunden werden, die im derzeit aktiven Bildschirm bearbeitet werden sollen. Dies lässt sich mit den Methoden ***find_element_*** oder ***find_elements_*** innerhalb von WebDriver erzielen. Bei beiden Methoden stehen unterschiedliche Möglichkeiten für die Suche zur Auswahl. Hierzu gehören beispielsweise die folgenden Kategorien:

- ***by_id (id_)***
- ***by_class_name (name)***
- ***by_tag_name (name)***
- ***by_xpath (xpath)***
- ***by_css_selector (css_selector)***

In all diesen Fällen ist das Argument der Funktion eine Zeichenkette (String), die die gesuchten Elemente repräsentiert. Es werden jedoch unterschiedliche Werte zurückgegeben. Zum Beispiel wird bei ***find_element_*** (in Einzahl) ein einzelnes WebElement zurückgegeben (falls eines gefunden wird), während bei ***find_elements_*** (in Mehrzahl) eine Liste von WebElementen zurückgegeben wird, die dem Argument entspricht.

Für das weitere Verständnis muss an dieser Stelle ein Konzept eingeführt werden, das in der Web-Entwicklung und beim -Testen verwendet wird: das DOM (Abkürzung für Document Object Model, deutsch „Dokumenten-Objekt-Modell"). Wenn eine Webseite in den Browser geladen wird, erstellt der Browser ein DOM und modelliert die Webseite als Baum von Objekten. Dieses DOM definiert einen Standard für den Zugriff auf die Webseite. Die Definition des W3C ist wie folgt:

> *Laut W3C ist das Dokumenten-Objekt-Modell (DOM) eine plattform- und programmiersprachenunabhängige Programmierschnittstelle, die es Programmen und Skripten ermöglicht, dynamisch auf den Inhalt, die Struktur und die Darstellung (Style) eines Dokuments zuzugreifen und diese zu aktualisieren.*

Das DOM definiert:

- Alle HTML-Elemente als Objekte

- Die Eigenschaften aller HTML-Elemente
- Die Methoden, mit denen auf alle HTML-Elemente zugegriffen werden kann
- Die Ereignisse, die alle HTML-Elemente betreffen

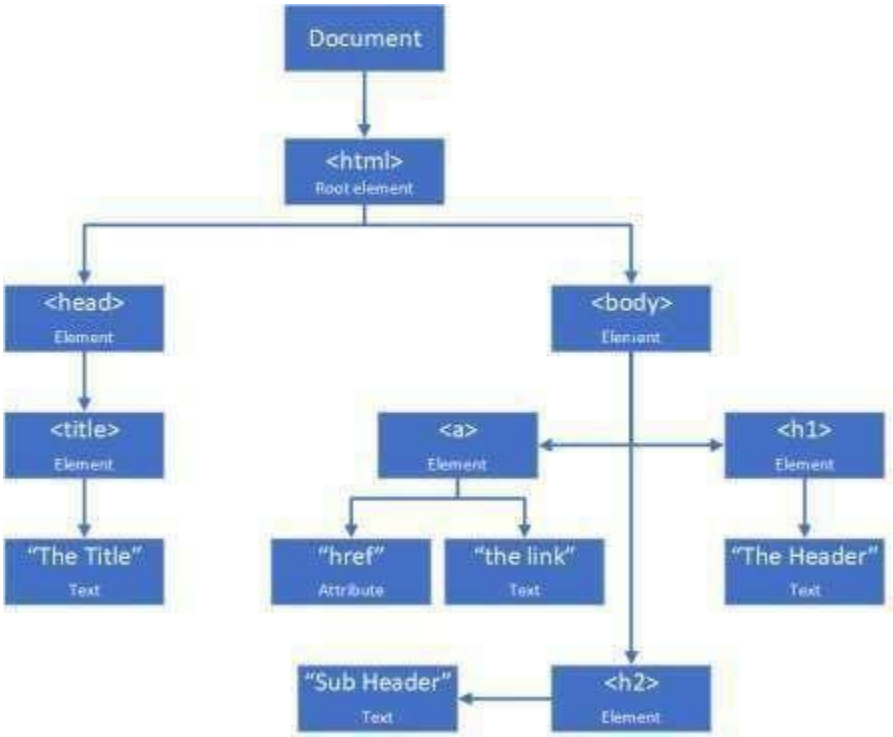

Abb. 40: DOM Baumstruktur

3.5.2 HTML-Methoden

Die nachfolgend beschriebenen Methoden basieren darauf, Bildschirmelemente von Webseiten zu finden, indem das Dokument nach HTML-Artefakten durchsucht wird. Die erste Methode, die hier beschreiben ist, verwendet das HTML-Attribut ***ID***. Für jede der unterschiedlichen Methoden zur Identifizierung von WebElementen, gibt es Vor- und Nachteile.

```
<element id="unique_id">
```

Abb. 41: HTML-Schnipsel

Beispiel für Python WebDriver Identifizierungs-Code:

```
element_found = driver.find_element_by_id('unique_id')
```

Abb. 42: Identifizierung über ID

Vorteile:

- Effizient
- Jede ID im HTML-Dokument sollte definitionsgemäß eindeutig sein
- Tester können dem SUT leicht IDs hinzufügen (Hinweis: Diese Änderungen sollten einem Peer-Review unterzogen werden)

Nachteile:

- IDs können automatisch generiert werden, d.h. sie können sich dynamisch ändern
- IDs sind nicht für Code geeignet, der an mehreren Stellen verwendet wird, z.B. eine Vorlage, mit der Kopf- und Fußzeilen für verschiedene modale Dialoge generiert werden
- Der Tester darf das SUT möglicherweise nicht ändern

Die zweite Möglichkeit, ein Element einer Webseite zu finden, ist die Suche nach seinem Klassennamen. Diese bezieht sich auf das HTML-Attribut ***class*** auf einem DOM-Element; hier als Beispiel das HTML-Schnipsel:

```
<element class="class-name1">
```

Abb. 43: HTML-Schnipsel

Beispiel für Python WebDriver Identifizierungs-Code:

```
element_found = driver.find_element_by_class_name('class-name1')
```

Abb. 44: Identifizierung über Klasse

Vorteile:

- Klassennamen können an mehreren Stellen im DOM verwendet werden; die Suche lässt sich jedoch auf die geladene Seite beschränken (z.B. auf einen Popup-Modaldialog)
- Tester können dem SUT leicht Klassennamen hinzufügen (Hinweis: Diese Änderungen sollten einem Peer-Review unterzogen werden)

Nachteile:

- Da Klassennamen an mehreren Stellen verwendet werden können, muss sorgfältig darauf geachtet werden, dass nicht das falsche Element gefunden wird
- Der Tester darf das SUT möglicherweise nicht ändern

Die dritte Möglichkeit, ein Element einer Webseite zu finden, ist über den HTML-Tag. Diese bezieht sich auf den Tag-Namen eines Elements im DOM; hier als Beispiel das HTML-Schnipsel:

```
<h2>
```

Abb. 45: HTML-Tag

Beispiel für Python WebDriver Identifizierungs-Code:

```
heading2_found = driver.find_element_by_tag_name('h2')
```

Abb. 46: Identifizieren über den Tag

Vorteil: Wenn ein Tag für eine Seite eindeutig ist, lässt sich der Bereich einschränken, in dem gesucht werden soll.

Nachteil: Wenn ein Tag für eine Seite nicht eindeutig ist, könnten das falsche Element identifiziert werden.

Die vierte Möglichkeit zur Identifizierung von Elementen einer Webseite ist der Linktext. Dies bezieht sich nur auf ein Anker-Tag (oder Link-Anker), dessen Text hervorgehoben ist, damit Benutzer diesen anklicken können. Tatsächlich stehen zwei ähnliche Methoden zur Verfügung: es kann die gesamte Textzeichenfolge oder ein Teil der Zeichenfolge spezifiziert werden. Im folgenden HTML-Beispiel wurde der Linktext fett dargestellt.

```
<html>
 <body>
  <p class="paragraph">XyzAbc.</p>
  <a href="next.html">Next Page</a>
 </body>
</html>
```

Abb. 47: HTML-Schnipsel

Beispiele für Python WebDriver Identifizierungs-Code:

```
element = driver.find_element_by_link_text('Next Page')
```

Abb. 48: Identifizieren über Linktext

```
element = driver.find_element_by_partial_link_text('Next Pa')
```

Abb. 49: Identifizieren über partiellen Linktext

Vorteile:

- Wenn der Linktext für eine Seite eindeutig ist, lässt sich das Element identifizieren

- Der Linktext ist für den Benutzer (in den meisten Fällen) sichtbar; es ist also kein Geheimnis, wonach der Testcode sucht
- Es ist etwas weniger wahrscheinlich, dass sich partieller Linktext (Teil-String des Linktextes) ändert als der vollständige Linktext

Nachteile:

- Der Linktext ändert sich wahrscheinlich eher als ID oder Klassenname
- Die Verwendung eines Teil-Strings kann die eindeutige Identifizierung eines einzelnen Links erschweren

3.5.3 Xpath-Methoden

Wie in Abschnitt 2.2 beschrieben, ist XPath (kurz für XML Path Language) eine Sprache, mit der bestimmte Knoten mithilfe verschiedener Kriterien in einem XML-Dokument selektiert werden können. Da HTML ein Teilsatz von XML ist, kann es mit XPath durchsucht werden. Es kann Probleme geben, wenn der HTML-Code nicht wohlgeformt ist (z.B. wenn nicht alle End-Tags enthalten sind). Hier als Beispiel ein HTML-Fragment:

```
<html>
 <body>
  <form id="sample_form1">
   <input name="text1" type="text" />
   <input name="text2" type="text" />
   <input name="submit_button" type="submit" value="Enter" />
  </form>
 </body>
</html>
```

Abb. 50: HTML-Schnipsel

WebDriver kann XPath verwenden, um einen bestimmten Knoten zu finden, und von dort aus kann ein Element identifiziert werden. Es könnte ein absoluter Pfad angegeben werden, aber das ist nicht ratsam, da jede Änderung wahrscheinlich den Code ungültig machen würde. Es ist daher besser, einen relativen Pfad ausgehend von einem gefundenen Knoten zu identifizieren, der einem Kriterium entspricht. Die nachfolgenden Beispiele zeigen die Spezifikation eines absoluten Pfads (Abb. 51) und eine robustere Version, die das gewünschte Element über XPath findet (Abb. 51). Beide geben das zweite Eingabefeld im HTML-Schnipsel (siehe oben) zurück.

```
element = driver.find_element_by_xpath('/html/body/form[2]')
```

Abb. 51: XPath mit absolutem Pfad

```
element = driver.find_element_by_xpath("//form[@id='sample_form1']/input[2]")
```

Abb. 52: XPath mit relativem Pfad

Die Definition eines relativen Pfades erfordert oft ein paar Eingaben mehr, aber die Suche nach dem gewünschten Knoten wird damit viel seltener fehlschlagen. Das ist ein ziemlich guter Kompromiss.

Innerhalb einer XPath-Zeichenfolge kann über ***ID***, ***Name***, ***Klasse***, ***Tag-Name*** usw. gesucht werden. Daher ist es möglich, generische Suchfunktionen mit XPath zu erstellen, indem ein Attributtyp an die Funktion übergeben wird, oder eine Pfadzeichenfolge, in die das Attribut integriert ist. Zum Beispiel:

```
def find_by_xpath(driver, path_string):
    element = driver.find_element_by_xpath('path_string')
    return element
```

Abb. 53: Generische XPath Suchfunktion

Die Variable ***path_string*** kann dann basierend auf dem Attribut zusammengestellt werden, das gesucht wird. Die beiden nachfolgenden Beispiele zeigen zuerst eine Suche über die ID (by ***id***) und dann eine Suche über die Klasse (by ***class***):

```
path_string = "//*[@id = '%s']" % 'id_to_find'
path_string = "//*[@class = '%s']" % 'class_to_find'
```

Abb. 54: Zusammenstellung eines Xpath-Strings

Der Aufruf der generischen Funktion würde dann wie folgt durchgeführt:

```
element_found = find_by_xpath(driver, path_string)
```

Abb. 55: Aufruf einer generischen Xpath-Funktion

Das ist nicht elegant, aber es zeigt die Flexibilität von XPath zum Identifizieren von Elementen.

Vorteile:

- Es können Elemente gefunden werden, die keine eindeutigen Attribute haben (ID, Klassenname usw.)
- XPath kann für die Suche mit generischen Suchfunktionen verwendet werden, indem je nach Bedarf die Suche über ID, über Klasse usw. verwendet wird

Nachteile:

- Absoluter XPath-Code ist nicht robust und kann bei der geringsten Änderung der HTML-Struktur brechen
- Relativer XPath-Code kann möglicherweise den falschen Knoten finden, wenn das gesuchte Attribut oder Element auf der Seite nicht eindeutig ist

- Da XPath möglicherweise zwischen Browsern unterschiedlich implementiert ist, wird möglicherweise zusätzlicher Aufwand erforderlich, um WebDriver-Tests in diesen Umgebungen auszuführen

3.5.4 Methoden mit CSS-Selektoren

Wie in Abschnitt 2.3 beschrieben, können Elemente auch mit CSS-Selektoren identifiziert werden. Zum Beispiel mit diesem HTML-Schnipsel:

```
<html>
 <body>
  <p class="paragraph">Some Latin nonsense.</p>
 </body>
</html>
```

Abb. 56: HTML-Schnipsel

Unter Verwendung der in Abschnitt 2.3 beschriebenen Regeln für CSS-Selektoren sollte der folgende Code den erwarteten Knoten identifizieren:

```
element = driver.find_element_by_css_selector('p.paragraph')
```

Abb. 57: WebElement über CSS-Selektor identifizieren

Vorteil: Wenn ein Element für eine Seite eindeutig ist, kann der Bereich, in dem gesucht wird, eingeschränkt werden.

Nachteil: Wenn ein Element für eine Seite *nicht* eindeutig ist, könnte das falsche Element gefunden werden.

3.5.5 Suche anhand von „erwarteten Bedingungen"

Selenium mit Python-Anbindungen hat ein Modul ***expected_conditions***, das aus ***selenium.webdriver.support*** mit mehreren vordefinierten Bedingungen importiert werden kann. Es ist möglich, benutzerdefinierte ***expected_condition***-Klassen zu erstellen, aber die vordefinierten Klassen sollten die meisten Bedürfnisse abdecken. Diese Klassen sind deutlich spezifischer als die oben erwähnten Suchfunktionen. Das heißt, sie bestimmen nicht einfach, ob ein Element existiert, sie prüfen alle auf einen bestimmten Zustand, in dem sich dieses Element befindet. Zum Beispiel bestimmt ***element_to_be_selected()*** nicht nur, dass das Element existiert, sondern es prüft auch, ob es in einem ausgewählten Zustand ist.

Hier sind einige Beispiele für Zustände (Liste ist nicht vollständig):

- alert_is_present
- element_selection_state_to_be(element, is_selected)

- element_to_be_clickable(locator)
- element_to_be_selected(element)
- frame_to_be_available_and_switch_to_it(locator)
- invisibility_of_element_located(locator)
- presence_of_element_located(locator)
- text_to_be_present_in_element(locator, text_)
- title_is(title)
- visibility_of_element_located(locator)

Dies wird in Abschnitt 4.2 weiter behandelt, da viele davon auch als Wartemechanismen verwendet werden.

3.6 Den Zustand von GUI-Elementen erhalten

Häufig genügt es nicht, den Ort eines bestimmten WebElements zu erhalten. Bei der Testautomatisierung gibt es mehrere verschiedene Gründe, warum auch auf einige Informationen zu einem Element zugegriffen werden muss. Zu derartigen Informationen kann die aktuelle Sichtbarkeit gehören, ob es aktiviert ist oder nicht, oder ob es ausgewählt ist oder nicht. Zu den Gründen können gehören:

- Es soll sichergestellt werden, dass der Zustand zu einem bestimmten Zeitpunkt im Test wie erwartet ist
- Es soll sichergestellt werden, dass sich ein Steuerelement in einem Zustand befindet, in dem es bei Bedarf im Testfall manipuliert (d.h. aktiviert) werden kann
- Es soll sichergestellt werden, dass sich das Steuerelement nach der Manipulation im erwarteten Status befindet
- Es soll sichergestellt werden, dass die erwarteten Ergebnisse nach dem Testlauf korrekt sind

Verschiedene WebElemente haben verschiedene Möglichkeiten, auf ihre Informationen zuzugreifen. Zum Beispiel haben viele WebElemente eine Texteigenschaft, die unter Verwendung des folgenden Codes abgerufen werden kann:

```
element_text = Element.text
```

Abb. 58: Texteigenschaft abrufen

Nicht jedes WebElement hat eine Texteigenschaft. Untersucht man zum Beispiel einen Header-Knoten (evtl. mit der Methode ***find_by_css_selector***('h1')), würden man erwarten, dass dieser eine Texteigenschaft hat. Andere WebElemente verfügen möglicherweise nicht über die gleiche Eigenschaft. Der Kontext von WebElementen und wie diese verwendet werden, kann helfen zu verstehen, welche Eigenschaften sie wahrscheinlich haben.

Auf einige WebElement-Eigenschaften muss mithilfe einer WebElement-Methode zugegriffen werden. Angenommen, Sie möchten ermitteln, ob ein bestimmtes WebElement derzeit aktiviert oder deaktiviert ist. Sie können die folgende Methode aufrufen, um diesen booleschen Wert zu erhalten:

```
cur_state = element.is_enabled()
```

Abb. 59: Prüfen, ob WebElement aktiviert ist

Die folgende Tabelle ist nicht umfassend, listet aber viele der häufigen Eigenschaften und Zugriffsmethoden auf, die Sie möglicherweise für die Automatisierung benötigen.

Tabelle 16: Häufige Eigenschaften und Zugriffsmethoden

Eigenschaft/ Methode	Argumente	Gibt zurück	Beschreibung
get_attribute()	Eigenschaft abrufen	Eigenschaft, Attribut oder *None*	Ruft die Eigenschaft ab. Wenn keine Eigenschaft mit diesem Namen vorhanden ist, wird das Attribut dieses Namens abgerufen. Wenn auch nicht vorhanden, kommt *None* zurück
get_property()	Eigenschaft abrufen	Eigenschaft	Ruft die Eigenschaft ab.
is_displayed()		Boolescher Wert	Gibt *true* zurück, wenn es für den Benutzer sichtbar ist
is_enabled()		Boolescher Wert	Gibt *true* zurück, wenn das Element aktiviert ist
is_selected()		Boolescher Wert	Gibt *true* zurück, wenn Kontrollkästchen oder Radiobutton ausgewählt ist
location		X/Y-Position	Gibt die X/Y-Position auf der darstellbaren Fläche zurück
size		Höhe, Breite	Gibt die Höhe und Breite des Elements zurück
tag_name		Eigenschaft tag_name	Gibt den tag_name des Elements zurück
text		Text für das Element	Gibt den Text zurück, der dem Element zugeordnet ist

3.7 Mit UI-Elementen mithilfe von WebDriver-Befehlen interagieren

3.7.1 Einführung

Angenommen, das WebElement, mit dem der automatisierte Test interagieren soll, wurde gefunden (siehe Abschnitt 3.5), und es wurde sichergestellt, dass das Element im richtigen Zustand ist, so dass es je nach Testfall manipuliert werden kann (siehe Abschnitt 3.6). Jetzt ist es an der Zeit, die gewünschte Manipulation durchzuführen.

Einer der Hauptgründe dafür, dass grafische Benutzeroberflächen populär wurden, besteht darin, dass es eine begrenzte Anzahl von Steuerelementen gibt, die manipuliert werden können. Jedes Steuerelement ist im Grunde ein Objekt auf dem Bildschirm, das leicht mit der Tastatur und/oder der Maus manipuliert werden kann. Jedes Steuerelement ist so konzipiert, dass es auch von unerfahrenen Computernutzern leicht verstanden werden kann. Es kann Text in ein Textfeld getippt, auf ein Optionsfeld (Radiobutton) geklickt, ein Element aus einer Liste ausgewählt werden, usw.

Das Automatisieren der Manipulation dieser Steuerelemente ist jedoch schwieriger zu verstehen. Was Tester manuell machen, geschieht oft unbewusst. Bei der Automatisierung ist jedoch sicherzustellen, dass alle Nuancen verstanden wurden, wie die Veränderungen an den Bildschirmsteuerelementen gemacht werden.

In den folgenden Abschnitten wird das Manipulieren folgender grafischen Objekte behandelt:

- Textfelder
- Anklickbare WebElemente (Felder, auf die man klicken kann)
- Kontrollkästchen
- Dropdown-Listen

Für jedes WebElement, das manipuliert werden soll, ist möglicherweise folgendes relevant:

- dass das WebElement existiert
- dass das WebElement angezeigt wird
- dass das WebElement aktiviert ist

Abhängig von der Website, der Art und Weise wie der HTML-Code geschrieben ist, ob AJAX verwendet wird und einer Reihe anderer Bedingungen, kann es einen Unterschied machen, ob das WebElement, das bearbeitet werden soll, tatsächlich angezeigt werden muss oder nicht, damit es manipuliert werden kann. Beispiel: Die Bearbeitung kann möglicherweise nicht funktionieren, wenn sich das WebElement zwar auf der aktiven Seite befindet und aktiviert ist, aber weggescrollt wurde, so dass es nicht mehr sichtbar ist. In Chrome durchgeführte Tests haben gezeigt, dass auf manchen Seiten die Arbeit mit einem WebElement, das aktuell nicht sichtbar ist, funktioniert, während dies auf anderen Seiten eine Ausnahme auslöst. Es sollte daher sichergestellt werden, dass alle WebElemente, die bearbeitet werden sollen, sichtbar auf dem Bildschirm angezeigt werden.

Da ein Browser-Bildschirm dynamisch modifiziert werden kann (zum Beispiel über AJAX) oder basierend auf früheren Aktionen aktualisiert werden kann, sollten diese Prüfungen wahrscheinlich unter Verwendung der ***expected_condition***-Klassen durchgeführt werden, die eine Synchronisierung des Timings ermöglichen. Dazu mehr in Kapitel 4.

3.7.2 Textfelder manipulieren

Bei der Eingabe von Text in ein editierbares Textfeld, möchte man in der Regel zuerst das Element löschen und dann die gewünschte Zeichenfolge in das Steuerelement eingeben. Angenommen, es wurde bereits überprüft, dass das Element existiert, angezeigt wird und aktiviert ist. Wenn der Zustand des Steuerelements nicht sichergestellt wird und einer oder mehrere davon nicht nicht zutreffen, wird eine versuchte Manipulation des Elements eine Ausnahme verursachen.

Im Beispiel wird angenommen, dass das Eingabesteuerelement bereits gefunden wurde, und zwar in der Variablen mit dem Namen *element*.

```
# First clear the control
element.clear()

# Now type into the control
string_to_type = 'XYZ'
element.send_keys(string_to_type)
```

Abb. 60: Text in ein Eingabefeld eingeben

3.7.3 WebElemente anklicken

Der Klick auf ein Element simuliert einen Mausklick. Es kann auf einen Radiobutton, einen Link oder ein Bild geklickt werden; im Grunde auf alles, das manuell mit der Maus angeklickt werden kann. In diesem Abschnitt werden Kontrollkästchen nicht behandelt (siehe nächster Abschnitt). Auch hier ist es wichtig zu überprüfen, ob das WebElement gerade anklickbar ist. Es kann die Methode ***element_to_be_clickable*** der ***expected_condition***-Klasse verwendet werden, um darauf zu warten, dass das Element anklickbar wird.

Auch hier wird angenommen, dass das WebElement bereits gefunden wurde. Möglicherweise muss gewartet werden, um sicherzustellen, dass es anklickbar ist (***expected_condition***-Methode verwenden):

```
Driver.support.expected_conditions.element_to_be_clickable(locator)
```

Abb. 61: Synchronisierungsmethode

Synchronisation wird in Kapitel 4 beschrieben. Vorausgesetzt, dass das WebElement durch das variable Element referenziert wurde und angeklickt werden kann, dann wird es wie folgt aufgerufen:

```
element.click()
```

Abb. 62: WebElement anklicken

Wenn das WebElement ein Link oder eine Schaltfläche wäre, dann würde man erwarten, dass eine Kontextänderung stattfindet, die auf dem Bildschirm verifiziert werden könnte. Wenn es sich um ein Kontrollkästchen handelt, kann das Ergebnis ausgewählt oder nicht ausgewählt sein. Dies wird im nächsten Abschnitt behandelt. Wenn es sich jedoch um eine Optionsschaltfläche handelt, die angeklickt wird, dann lässt sich mit dem folgenden Aufruf überprüfen, ob die Schaltfläche tatsächlich ausgewählt wurde:

```
element.is_selected()
```

Abb. 63: Zustand des WebElements überprüfen

3.7.4 Kontrollkästchen manipulieren

Wenn Kontrollkästchen (engl. checkbox) angeklickt werden, müssen sie anders als andere anklickbare Steuerelemente behandelt werden. Wenn Sie mehrmals auf einen Radiobutton klicken, dann bleibt dieses Optionsfeld *ausgewählt*. Wenn jedoch mehrmals auf ein Kontrollkästchen geklickt wird, dann wechselt der Zustand mit jedem Klick zwischen *ausgewählt* und *nicht ausgewählt* und wieder *ausgewählt*, usw.

Daher ist es wichtig zu wissen, welcher Zustand erreicht werden soll, wenn das Kontrollkästchen manipuliert wird. Es kann eine Funktion erstellt werden, die das Kontrollkästchen und den gewünschten Zustand übernimmt und die richtige Aktion ausführt, unabhängig vom Zustand, in dem sich das Kontrollkästchen zunächst befindet.

Angenommen, das Kontrollkästchen wurde in die Variable *checkbox* geladen. Angenommen, dass eine boolesche Variable *want_checked* übergeben wurde, um den gewünschten Endzustand zu bestimmen.

```
# if we want it selected and it is not, then click on it
# if we want it deselected and it is selected, click on it
def set_check_box(element, want_checked):
    if want_checked and not element.is_selected():
            element.click()
    elif element.is_selected() and not want_checked:
            element.click()
```

Abb. 64: Funktion zum Anklicken der Checkbox

```
set_checkbox (checkbox, True)
```

Abb. 65: Checkbox-Funktion aufrufen, um Kontrollkästchen anzukreuzen

3.7.5 Dropdown-Steuerelemente manipulieren

Dropdown-Listenfelder (Auswahlsteuerelemente) werden von vielen Websites verwendet, damit Benutzer eine von vielen Optionen auswählen können. Manchmal ist es sogar möglich, mehrere Elemente aus der Liste gleichzeitig auszuwählen.

Es gibt viele Möglichkeiten, mit der Liste eines Auswahlsteuerelements zu arbeiten. Dazu gehören Optionen zum Auswählen einzelner Elemente oder mehrerer Elemente. Es gibt auch verschiedene Möglichkeiten, Elemente abzuwählen.

Mögliche Auswahloptionen sind u.a.:

- Im HTML-Code nach dem gewünschten Element suchen und auf Element klicken
- Nach einem Wert auswählen (***select_by_value***(*value*))
- Alle Elemente auswählen, die einen bestimmten Text anzeigen (***select_by_visible_text***(*text*))
- Ein Element nach Index auswählen (***select_by_index***(*index*))

Mögliche Abwahloptionen sind u.a.:

- Alle Element abwählen (***deselect_all***())
- Nach Index abwählen (***deselect_by_index***(*index*))
- Nach Wert abwählen (***deselect_by_value***(*value*))
- Nach angezeigtem Text abwählen (***deselect_by_visible_text***(*text*))

Sobald Auswahl bzw. Abwahl der Elemente in der Dropdown-Liste abgeschlossen ist, gibt es mehrere Möglichkeiten, zu sehen, was ausgewählt ist:

- Die Option ***all_selected_options*** gibt eine Liste aller ausgewählten Objekte zurück
- Die Option ***first_selected_option*** gibt das erste ausgewählte Element zurück (oder nur ein einzelnes Auswahl-Steuerelement)
- Mit ***options*** wird eine Liste aller Optionen angezeigt

Da das Klicken auf Listenelemente eine Aufgabe ist, die häufig durchgeführt werden muss, kann dafür eine Funktion erstellt werden (wie für Kontrollkästchen). In diesem Fall muss zuerst auf die Dropdown-Schaltfläche geklickt werden, um die vollständige Liste anzuzeigen. Sobald die Liste geöffnet ist, wird die gewünschte Option ausgesucht und angeklickt. Hier eine entsprechende Funktion:

```
def click_dropdown_option_by_id_and_id(driver, dropdown_id, option_id):
    dropdown_element = driver.find_element_by_id('dropdown_id')
    dropdown_element.click()
    option_element = driver.find_element_by_id('option_id')
    option_element.click()
```

Abb. 66: Funktion zum Anklicken von Elementen in Dropdown-Listen

3.7.6 Mit modalen Dialogen arbeiten

Ein modaler Dialog ist ein Fenster, das über einem Browserfenster angezeigt wird, und keinen Zugriff auf das darunter liegende Fenster erlaubt, bis es bearbeitet wurde. Diese ähneln zwar den Dialogfeldern mit Benutzeraufforderungen, unterscheiden sich aber doch soweit, um hier getrennt behandelt zu werden. Im nächsten Punkt werden dann die Benutzeraufforderungen, wie z.B. Warnungen, beschrieben.

Im Allgemeinen werden modale Dialoge aufgerufen, wenn der Autor der Website spezifische Eingaben vom Benutzer erhalten möchte (z.B. ein Popup zur Abfrage von Benutzername/Passwort), oder wenn er möchte, dass bestimmte Aufgaben vom Benutzer ausgeführt werden. Wenn Sie beispielsweise auf einer E-Commerce-Website ein Element zum Einkaufswagen (engl. cart) hinzufügen, wird möglicherweise ein Informationsfenster eingeblendet; siehe folgendes Beispiel:

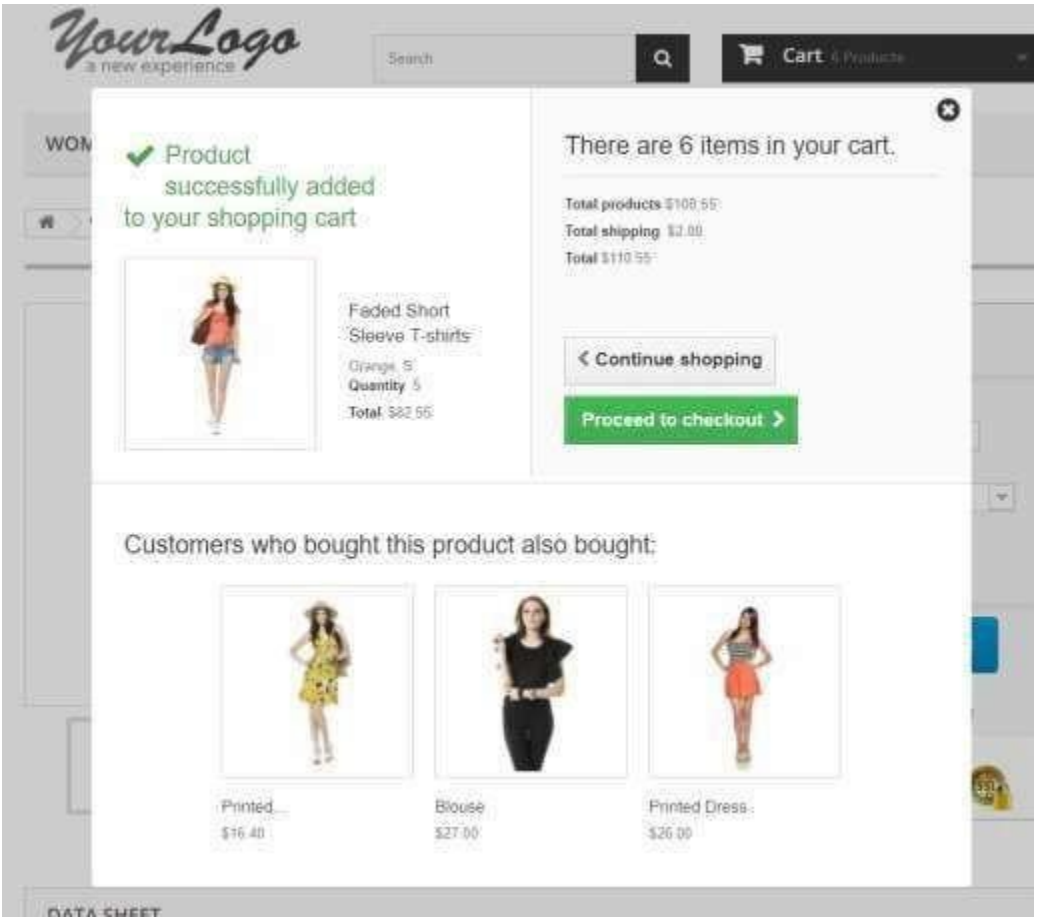

Abb. 67: Modales Fenster für E-Commerce-Einkaufswagen

In diesem Beispiel kann der Benutzer entscheiden, was er tun möchte: *Zur Kasse gehen* oder *Weiter einkaufen*. Darüber hinaus kann jede Menge zusätzlicher Informationen im modalen Dialog angezeigt werden. In diesem Fall versucht die E-Commerce-Website den Benutzer dazu zu verleiten, noch etwas anderes zu kaufen, basierend auf dem, was sich bereits im Einkaufswagen befindet.

Der gesamte Code für den modalen Dialog ist im HTML-Code enthalten, der den modalen Dialog aufgerufen hat. Das Bearbeiten des modalen Dialogfelds ist einfach; es muss nur der Code in der aufrufenden Seite gefunden und manipuliert werden. Hierfür gelten die gleichen Regeln wie für das Auffinden und Bearbeiten von Formularsteuerelementen, wie bereits im Lehrplan beschrieben.

Angenommen, in diesem modalen Dialogfeld soll auf die Schaltfläche *Zur Kasse gehen* geklickt werden. Der erste Schritt wäre, den Ort des Codes für den modalen Dialog zu bestimmen. In diesem Fall ist die ***ID*** des Abschnitts, der das modale Element darstellt, ***layer_cart***. Für dieses Element des Codes würde eine WebDriver Objektreferenz wie folgt erstellt:

```
modal = driver.find_element_by_id('layer_cart')
```

Abb. 68: Modales Element suchen

Im nächsten Schritt muss dann das Element identifiziert werden, das die Schaltfläche darstellt; hierzu wird die *Inspect* -Funktion des Browsers verwendet. Im vorliegenden Fall lautet der Klassenname der Schaltfläche ***button_medium***. Noch einmal: es soll ein Verweis auf dieses Element erstellt werden, damit es manipuliert werden kann. Hierfür kann folgender Code verwendet werden:

```
proceed_button = modal.find_element_by_class_name('button-medium')
```

Abb. 69: Eine Schaltfläche im modalen Element identifizieren

Nachdem die Schaltfläche gefunden wurde, ist das Drücken dieser so einfach wie das Aufrufen der click ()-Methode für diesen Verweis:

```
proceed_button.click()
```

Abb. 70: Schaltfläche drücken

Zu diesem Zeitpunkt wird der modale Dialog geschlossen und man gelangt zu einem neuen Ort im Hauptbrowserfenster, im vorliegenden Fall zur folgenden Einkaufswagen-Übersichtsseite:

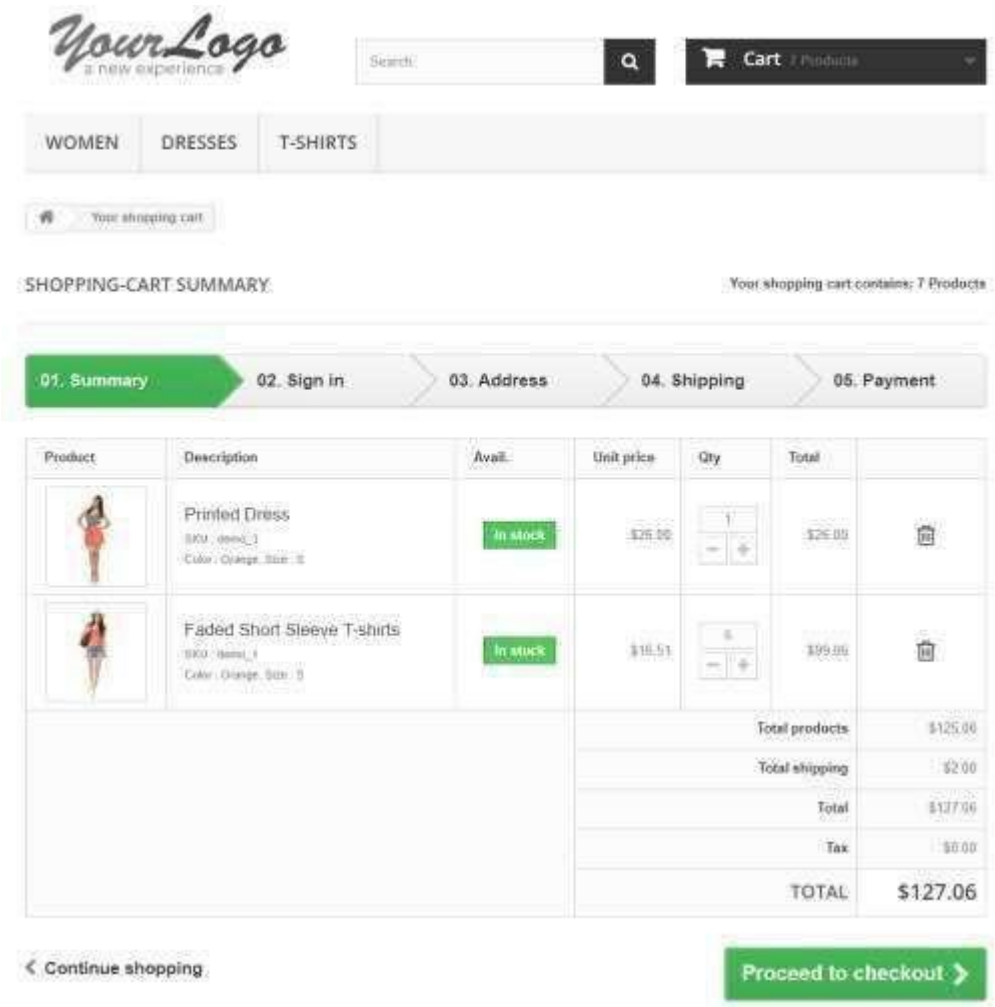

Abb. 71: Einkaufswagen-Übersichtsseite

3.8 Mit Benutzeraufforderungen in Webbrowsern mithilfe von WebDriver-Befehlen interagieren

Benutzeraufforderungen (engl. user prompt) sind modale Fenster, mit denen Benutzer interagieren müssen, bevor sie mit den Steuerelementen im Browserfenster selbst weiter interagieren können.

Für Automatisierungszwecke werden Benutzeraufforderungen normalerweise nicht automatisch verarbeitet. Wenn ein Skript versucht, die Eingabeaufforderung zu ignorieren und weiterhin Befehle an das Browserfenster selbst zu senden, wird von der Aktion ein ***unexpected alert open***-Fehler zurückgegeben.

Jeder Benutzereingabeaufforderung ist eine Meldung zugeordnet (die möglicherweise NULL ist). Dies ist ein Textfeld, auf das mit dem unten gezeigten Code zugegriffen werden kann.

Das W3C definiert drei verschiedene Dialoge des Typs *Alert*:

- *Alert* (Warndialog)
- *Confirm* (Bestätigungsdialog)
- *Prompt* (Aufforderungsdialog)

Da alle drei so definiert sind, dass sie fast gleich funktionieren, wird hier nur die Warnmeldung *Alert* behandelt.

Das Warndialogfeld wird häufig verwendet, um sicherzustellen, dass der Benutzer auf einige bedeutsame Informationen aufmerksam gemacht wird.

Da Warndialogfelder nicht Teil der Webseite sind, erfordern sie eine spezielle Verarbeitung. WebDriver mit Python verfügt über eine Reihe von Methoden, mit denen der Warndialog vom Automatisierungsskript aus gesteuert werden kann. Diese Methoden gelten für alle drei Arten der oben erwähnten Dialoge.

Die erste Methode erstellt eine Referenz und verwendet hierzu eine Methode des WebDriver-Objekts ***switch_to***. Die Syntax zur Verwendung dieser Methode ist wie folgt:

```
alert = driver.switch_to.alert
```

Abb. 72: Ein Alert-Objekt erstellen

Der Text kann abgerufen werden über:

```
msg_text = alert.text
```

Abb. 73: Text aus Warnung abrufen

Um zu bestimmen, ob der erwartete Text in der Warnung enthalten ist, wird zur Warnung gewechselt, der Text abgerufen und dann überprüft, ob der erwartete Text in der Warnung enthalten war. Wenn der Text vorhanden ist, wird die Variable ***passed*** auf ***True*** gesetzt. Wenn nicht, wird die Variable ***passed*** auf ***False*** gesetzt.

```
alert = driver.switch_to.alert
msg_text = alert.text
expected_text = 'XYZ'
assert expected_text in msg_text, "The expected text not found"
```

Abb. 74: Text der Warnung vergleichen

Warndialoge können auf zwei Arten geschlossen werden: mit der ***accept()***-Methode (entspricht dem Drücken der Schaltfläche *OK*) und mit der ***dismiss()***-Methode (entspricht dem Drücken der Schaltfläche *Abbrechen*), oder mit der Schaltfläche zum Schließen des Fensters in der oberen Ecke des Dialogfensters.

```
alert = driver.switch_to.alert
alert.accept()
alert.dismiss()
```

Abb. 75: Möglichkeiten zum Schließen des Warndialogs

Kapitel 4 – Erstellung wartbarer Testskripte

Schlüsselbegriffe

Fixture, Page Object Pattern, Persona

Lernziele für Erstellung wartbarer Testskripte

STF-4.1 (K2) Verstehen, welche Faktoren die Wartbarkeit von Testskripten unterstützen und beeinflussen

STF-4.2 (K3) Geeignete Wartemechanismen verwenden können

STF-4.3 (K4) Die GUI des SUT analysieren und Page Objects verwenden können, um Abstraktionen zu erstellen

STF-4.4 (K4) Testskripte analysieren und Grundsätze des schlüsselwortgetriebenen Tests für die Erstellung von Testskripten anwenden können

4.1 Wartbarkeit von Testskripten

In Kapitel 1.2 haben wir den Unterschied zwischen einem manuellen Test und einem automatisierten Skript behandelt. Dieses Thema muss an dieser Stelle nochmals aufgegriffen werden, da es direkten Einfluss auf die Wartbarkeit der Automatisierungssoftware hat.

Auf einen essentiellen Kern reduziert, ist ein manueller Test ein gezielter Satz abstrakter Anweisungen, die nur wertvoll sind, wenn sie von einem manuellen Tester verwendet werden, um den Test auszuführen. Die Daten und erwarteten Ergebnisse sollten auch vorhanden sein, aber das Herzstück sind die durchzuführenden Schritte. Der manuelle Tester fügt diesen abstrakten Anweisungen Kontext und Angemessenheit hinzu, sodass Tests fast jeder Komplexität erfolgreich ausgeführt werden können.

Wenn ein Schritt im manuellen Skript fordert, auf eine Schaltfläche zu klicken, können Tester dies problemlos tun und brauchen nicht lange darüber nachdenken, ob das Steuerelement sichtbar ist, ob es aktiviert ist oder ob es das richtige Steuerelement für die Aufgabe ist. Tatsache ist jedoch, dass sie darüber unbewusst nachdenken. Wenn das Steuerelement nicht sichtbar ist, können Sie versuchen, es sichtbar zu machen. Wenn es nicht aktiviert ist, werden Tester es nicht einfach trotzdem benutzen, sondern werden versuchen herauszufinden, warum es deaktiviert ist und ob sie das beheben. Sobald sie an das Steuerelement gelangen, können sie das Richtige tun. Und wenn mit

diesem Steuerelement etwas nicht stimmt, können sie identifizieren, was aufgetreten ist und einen Fehlerbericht erstellen und/oder die manuelle Testvorgehensweise ändern.

Automatisierungswerkzeuge sind nicht intelligent wie manuelle Tester. Egal wie teuer, alle Werkzeuge haben einen sehr begrenzten Kontext und fast keine Angemessenheit. Automatisierer sind dagegen menschlich und intelligent. Sie können einem Werkzeug Kontext und Angemessenheit hinzufügen und in die automatisierten Skripte hineinprogrammieren.

Je mehr man jedoch versucht, diese Intelligenz in das automatisierte Skript hineinzuprogrammieren, desto komplexer wird das Skript, und desto höher wird die Wahrscheinlichkeit, dass es aufgrund seiner ureigenen Komplexität zu Ausfällen des Skripts kommt.

Automatisierung für das Testen zu erstellen, wird immer wieder mit dem Versuch verglichen, eine Kugel mit einer anderen Kugel abzuschießen.

Dieser Spruch bedeutete immer, dass Automatisierung sehr komplex ist. Komplexität ist jedoch ein zweischneidiges Schwert. Einerseits muss Intelligenz in die Skripte eingebaut werden, damit sie einen menschlichen Tester, der einen Test ausführt, besser simulieren können. Andererseits nimmt die Komplexität der SUTs, das wir testen, immer mehr zu.

Aber je mehr Komplexität der Automatisierung hinzugefügt wird, desto mehr Fehler sind in der Automatisierung zu erwarten.

Ganz gleich, wie erfahren die Testautomatisierer sind, es gibt Grenzen hinsichtlich der Bedingungen, die durch das automatisierte Skript gesteuert werden können.

Manuelle Tester können versuchen herauszufinden, warum ein Steuerelement nicht sichtbar ist. Testautomatisierer können dies wahrscheinlich nicht. Automatisierung ist darauf beschränkt, Code in das Skript zu schreiben, um das Skript eine begrenzte Zeit warten zu lassen, in der Hoffnung, dass sich das Problem selbst löst. Das Gleiche gilt, wenn das Steuerelement aktiviert ist.

Testautomatisierer können allerdings sicherstellen, dass das Steuerelement im richtigen Zustand ist, bevor es verwendet wird, und wenn dies nicht der Fall ist, eine bestimmte Zeit zu warten; falls es dann immer noch nicht benutzt werden kann, wird der Fehler protokolliert, die Ausführung des Skripts abgeschlossen und mit dem nächsten Test fortgefahren. Wenn Tester auf das Steuerelement zugreifen können, können sie überprüfen, ob sich das Steuerelement nach der Manipulation wie erwartet verhält. Und wenn es ein Problem gab, können sie eine nützliche Protokollnachricht schreiben, damit das Problem effektiver und effizienter behoben werden kann.

Ziel ist es, die Arbeit von Testern erleichtern. Das bedeutet, Intelligenz in abrufbare Funktionen einzubauen, anstatt Intelligenz in jedes einzelne Skript hineinprogrammieren zu müssen. Mit anderen Worten: die Intelligenz außerhalb des Skripts auf der Ebene des TAA und/oder TAF anzusiedeln.

Indem mehr Intelligenz weiter oben angesiedelt und aus den Skripten selbst herausgenommen wird, werden die Skripte wartbarer und skalierbarer. Wenn der Code, der Intelligenz hinzufügt, fehlschlägt, dann werden zwar viele Skripte fehlschlagen, aber die Fehlerbehebung kann an einem einzigen Kontaktpunkt erfolgen.

Der Lehrplan enthält einige Beispiele für die Erstellung dieser aufrufbaren Funktionen. Diese Codebeispiele sind jedoch für eine produktionsreife Automatisierung nicht annähernd ausreichend. Gute Testautomatisierer erstellen eher Aggregatfunktionen, die manche auch als Wrapper oder Wrapper-Funktionen bezeichnen. Das nachfolgende Beispiel zeigt eine Wrapper-Funktion mit eingebauter Intelligenz in abstrakter Form. Angenommen, der Tester möchte auf ein Kontrollkästchen klicken. Dann wird er versuchen, die Funktion so zu bauen, dass sie das imitiert, was ein manueller Tester tatsächlich denkt und tut.

Die Argumente für diese Wrapper-Funktion beinhalten sicherlich: das zu verwendende Kontrollkästchen, den gewünschten Endzustand (aktiviert oder deaktiviert), eventuell eine Wartezeit für den Fall, dass es nicht sofort bereit ist. Die Aufgaben würden der folgenden Logik folgen:

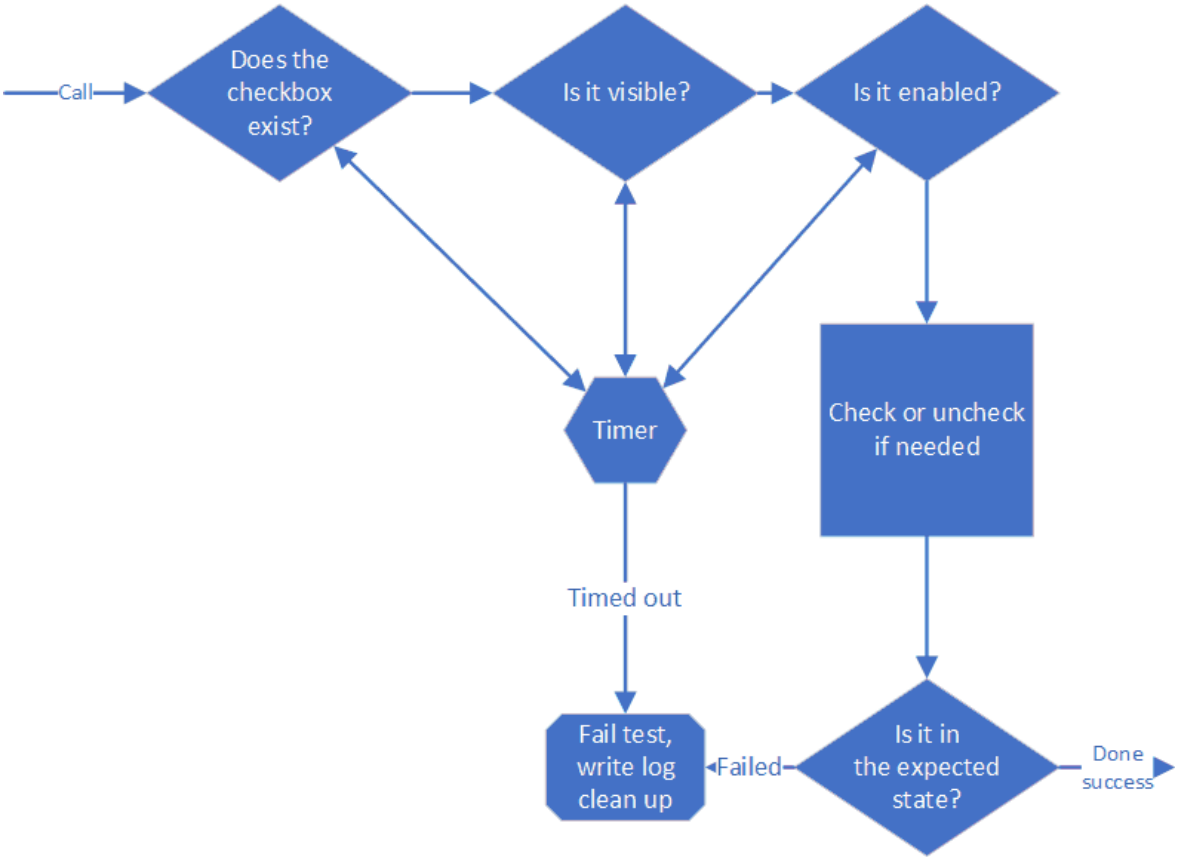

Abb. 76: Logik einer Wrapper-Funktion

Es sollte viel Aufwand in den Block *failed* gesteckt werden. Die Protokollinformationen sollten umfassend sein, um den Aufwand für die Fehlerbehebung nach einem fehlgeschlagenen Test zu reduzieren. Die Aufräumfunktionalität sollte sicherstellen, dass die Testsuite mit dem nächsten und dem übernächsten Test usw. fortfahren kann. Ein erfahrener Testautomatisierer hat einmal gesagt, dass der wichtigste Test, den Testautomatisierer berücksichtigen müssen, immer der nächste ist. Wenn es immer möglich ist, den jeweils nächsten Test auszuführen, dann kann die gesamte Testsuite ausgeführt werden, unabhängig davon, wie viele Tests im Laufe der Ausführung fehlschlagen.

Letztendlich versuchen Tester herauszufinden, wo das SUT nicht richtig funktioniert, um so Vertrauen aufzubauen, wenn es richtig zu funktionieren scheint. Wenn die Automatisierung gut funktioniert, werden erwartungsgemäß auch Fehlerwirkungen auftreten. In diesem Fall müssen die Fehlerinformationen erfasst werden, dann kann mit dem nächsten Test fortgefahren werden.

Tester sollten daran denken, dass jeder Test dazu dient, das SUT, seine Umgebung und seine Verwendung zu untersuchen, damit sie Dinge erfahren, die sie noch nicht wussten. Ein Test, der dem Tester nichts sagt, was er noch nicht wusste, ist kein wertvoller Test, ob er nun automatisiert ist oder nicht. Im ISTQB® Certified Tester Foundation Level Lehrplan werden Testentwurfsverfahren wie die Äquivalenzklassenbildung und die Grenzwertanalyse beschrieben. Diese Vorgehensweisen sollen dazu dienen, die Anzahl der Tests zu reduzieren, um so schnell wie möglich die wesentlichen Informationen über das SUT herauszufinden.

Wrapper sind wichtig, aber es können auch andere Funktionen zur Verbesserung der Automatisierung erstellt werden. Jeder Code, den die Automatisierung wiederholt aufrufen muss, sollte als generische Funktion erstellt werden. Dies folgt den guten Entwicklungspraktiken der funktionellen Zerlegung und des Refactorings. Es sollten Bibliotheken mit Funktionen erstellt werden, die im Code enthalten sein können, damit alle im Team darauf zurückgreifen können. Wo immer möglich, sollte alles aus den eigentlichen Skripten in diese aufrufbaren Bibliotheken verschoben werden.

Es braucht Zeit und Ressourcen, diese Bibliothekfunktionen zu erstellen. Für erfolgreiche Testautomatisierer könnte dies jedoch die beste Investition sein, die Sie jemals getätigt haben.

Beim Testen eines Browsers sollte sichergestellt werden, dass für die Suche von WebElementen mit XPath- und CSS-Selektoren keine absoluten Pfade verwendet werden. Wie im vorherigen Kapitel erläutert, führt jede Änderung des HTML-Codes dazu, dass sich diese Pfade ändern und die Automatisierung dadurch fehlschlägt. Natürlich können auch relative Pfade fehlschlagen; aber da dies seltener vorkommt; benötigen sie weniger Wartungsaufwand.

Es ist unbedingt ratsam, bei der Automatisierung die Entwickler im Unternehmen mit ins Boot zu holen und bei diesen ein Problembewusstsein zu schaffen. Sie können durch die Art, wie sie den Code schreiben, viel bewirken und verhindern, dass die Automatisierung, die direkt mit dem HTML arbeitet, ständig fehlschlägt.

Es ist weiterhin ratsam, übergreifende Namen für die Variablen, Konstanten, Funktionen usw. zu definieren, die aussagekräftig sind. Dies macht den Code nicht nur lesbarer, sondern erleichtert auch die Wartung des Automatisierungscodes. Wenn Code einfacher zu pflegen ist, bedeutet dies in der Regel weniger Regressionsfehler, wenn Änderungen vorgenommen werden. Es erfordert nur ein paar Sekunden mehr, sich gute Namen auszudenken, kann aber am Ende Stunden einsparen.

Außerdem wichtig: Kommentare. Viele Kommentare. Aussagekräftige Kommentare. Viele Automatisierer gehen davon aus, dass es ihr eigener Code ist und sie sich daran erinnern werden, warum sie ihn auf eine bestimmte Art und Weise geschrieben haben. Oder sie denken, dass Automatisierung „anders“ ist, nicht wie echter Code. Sie haben nicht recht. Sie könnten bereits am nächsten Tag vergessen haben, wie sie ein Problem genial gelöst haben. Es ist zu vermeiden, immer wieder ganz von vorne zu beginnen. Wenn sich ein Automatisierungsprojekt erfolgreich entwickelt, dann werden früher oder später auch Kollegen mit dem Code arbeiten. Es muss für andere so einfach wie möglich gemacht werden.

Es sollten Testkonten und -Fixtures (sinngem. *festes Inventar*) erstellt werden, die speziell für die Automatisierung verwendet werden. Auf keinen Fall sollten diese mit manuellen Testern geteilt werden. Die Automatisierung muss Sicherheit in Bezug auf die Daten haben, die in der Automatisierung verwendet werden. Manuelle Tester, die Änderungen vornehmen, könnten die Automatisierung wahrscheinlich leicht beschädigen.

Diese Testkonten und -Fixtures sollten nicht personengebunden sein. Generische Konten, die echte Konten nachahmen, sind viel besser zu isolieren und vor Änderungen zu schützen. Es sollten genügend unterschiedliche Konten zur Verfügung stehen, dass sich die Daten bei mehreren Tests nicht gegenseitig beeinträchtigen.

Für diese Konten müssen genügend Daten erzeugt werden, damit sie echte Konten simulieren. Auch verschiedene Personas für diese Konten dürfen nicht vergessen werden. Wenn das SUT verschiedene Arten von Benutzern hat (z.B. Anfänger, Technik-Freaks, versierte Power-User usw.), dann sollten diese in den Testkonten modelliert werden.

Protokollierung wurde in Kapitel 3.1 dieses Lehrplans behandelt. Protokollierung muss ernst genommen werden. Wenn das Automatisierungsprojekt erfolgreich ist, wird das Management mehr wollen. Viel mehr. Eine der Möglichkeiten, die Automatisierung skalierbarer zu machen, ist eine gute Protokollierung von Anfang an. Schließlich will niemand im Nachhinein hunderte (oder tausende?) Skripte mit guter Protokollierung nachrüsten.

Python bietet robuste Protokollierungsfunktionen, es können aber auch eigene erstellt werden. Bei der Protokollierung muss über den Tellerrand hinausgedacht werden. Solide Protokolle können den Zeitaufwand für die Fehlerbehebung reduzieren, wenn Fehlerwirkungen auftreten. Die Bedürfnisse der Organisation müssen berücksichtigt werden. Wenn das SUT einsatz- oder sicherheitskritisch ist,

müssen die Protokolle möglicherweise so umfassend sein, dass sie einem Audit unterzogen werden können.

Die Denkprozesse des manuellen Testers müssen modelliert werden. Was finden diese über das SUT heraus, wenn sie einen manuellen Test durchführen? Bei der Automatisierung sollte versucht werden, ob dieselben Informationen erfasst und protokolliert werden können. Der Kreativität des Automatisierers sind dabei keine Grenzen gesetzt, wenn dieser festlegt, wie, was, wann und wo protokolliert wird.

Die Kontrolle über die erstellten Dateien darf nicht verloren gehen. Es müssen konsistente Namenskonventionen verwendet und die Dateien in konsistenten Verzeichnissen abgelegt werden. Beispielsweise sind Ordner mit einem Zeitstempel im Namen ratsam, die sie alle Dateien aus einem Testlauf enthalten. Falls die Automatisierung auf mehreren Maschinen läuft oder wenn Tests in verschiedenen Umgebungen ausgeführt werden, macht es Sinn, wenn der Name der Workstation bzw. der Name der Umgebung dem Ordnernamen hinzugefügt wird.

Es ist sinnvoll, Zeitstempel in den Dateinamen einzufügen. Dies garantiert, dass frühere Testlaufergebnisse nicht von der Automatisierung überschrieben werden. Wenn die Dateien in Zukunft nicht mehr benötigt werden, sollten sie in Verzeichnissen abgelegt werden, die gelöscht werden können, ohne dass Schaden angerichtet wird. Wenn die Dateien allerdings gespeichert werden müssen, dann sollte diese Information in der Ordnerstruktur enthalten sein.

Die Verwendung konsistenter Dateinamen ist wichtig. Gemeinsam mit den anderen Testautomatisierern sollte eine Vereinbarung über Stil- und Namenskonventionen getroffen werden. Diese Standards und Richtlinien müssen an neue Testautomatisierer im Team weitergegeben werden. Es dauert nicht lange, bis Chaos herrscht, wenn jeder auf Dauer bei seinen Artefakten nach Lust und Laune handelt.

Traditionell waren die Automatisierer eher die Rebellen der Entwicklung, die immer ihr eigenes Ding gemacht haben. Der Umfang der Investitionen in die Automatisierung ist jedoch sowohl hinsichtlich der benötigten Ressourcen als auch des Zeitaufwands beträchtlich. Es sollten also Mindeststandards und Leitlinien festgelegt und eingehalten werden, um sicherzustellen, dass die Investition zumindest eine Chance hat, sich auszuzahlen.

4.2 Wartemechanismen

Wenn ein manueller Tester einen Test ausführt, ist das Warten kein Thema, das berücksichtigt werden müsste. Beispiel: Angenommen, der Tester öffnet eine Datei. Wenn die Datei rechtzeitig geöffnet wird (d.h. „rechtzeitig" nach Ansicht des Testers), wird nicht weiter darüber nachgedacht. Jedes Mal, wenn ein manueller Tester auf ein Steuerelement klickt oder das SUT anderweitig manipuliert, tickt in seinem Kopf unbewusst eine Uhr. Wenn Tester das Gefühl haben, dass etwas zu

lange dauert, werden sie wahrscheinlich den Test erneut ausführen und dabei dem Timing mehr Aufmerksamkeit schenken.

Was nie passiert, ist, dass ein Tester stundenlang dasitzt und geduldig auf eine bestimmte Aktion wartet.

Im vorliegenden Lehrplan wurde bereits mehrfach der Kontext erwähnt, den ein manueller Tester zu einem Test hinzufügt. Timing ist einer dieser kontextuellen Faktoren, die Testautomatisierer verstehen müssen.

Angenommen, ein Tester öffnet eine sehr kleine Datei mit ein paar hundert Bytes. Wenn diese sich nicht sofort öffnet, ist der Tester wahrscheinlich beunruhigt und kann einen Fehlerbericht erstellen. Angenommen, derselbe Tester versucht, eine Datei mit zwei Gigabyte zu öffnen. Wenn es dreißig Sekunden dauert, sie zu öffnen, ist das wahrscheinlich keine Überraschung. Wenn eine Informationsmeldung angezeigt wird, dass das Öffnen etwas Zeit in Anspruch nimmt, würden sie warten, ohne mit der Wimper zu zucken.

Der Kontext ist also wichtig.

In der Automatisierung ist, egal welches Werkzeug verwendet wird, wenig bis gar kein Kontext implizit vorhanden.

Bei Werkzeugen ist im Allgemeinen eine Zeit hinterlegt, die diese bereit sind, auf eine Aktion zu warten. Wenn die erwartete Aktion nicht innerhalb dieses Zeitrahmens auftritt, zeichnet das Tool einen Fehler auf und fährt mit der Ausführung fort (wo genau es fortfährt, hängt vom Werkzeug, dem Setup und einer Vielzahl anderer Faktoren ab).

Wenn im Werkzeug eingestellt ist, dass es 3 Sekunden wartet, wird eine Aktion, die erst nach 3,0001 Sekunden auftritt, als Fehler betrachtet (auch wenn dies vielleicht innerhalb des kontextuellen Zeitbereichs eines manuellen Testers liegt).

Wenn der Testautomatisierer alle Timing-Überlegungen weglässt, ist es möglich, dass das Werkzeug ewig auf eine Aktion wartet. Es kein gutes Gefühl, an einem Montagmorgen zur Arbeit zu kommen und herauszufinden, dass die Automatisierungssuite, die am Freitagabend kurz vor Feierabend gestartet wurde, beim zweiten Test hängengeblieben ist und seither auf eine Aktion wartet.

Testautomatisierer müssen die Anforderungen der Automatisierung und die Funktionsweise der verwendeten Werkzeuge verstehen.

Selenium WebDriver mit Python verfügt über verschiedene Wartemechanismen, die Testautomatisierer verwenden können, um die Synchronisation für eine Automatisierung einzurichten. Ein expliziter Wartemechanismus sollte eigentlich selten verwendet werden, ist jedoch eine der häufigsten Methoden, die viele Automatisierer verwenden.

Den folgenden Code kennt wahrscheinlich fast jeder, der jemals Tests automatisiert hat:

```
import time
…
time.sleep(5)
```

Abb. 77: Ein expliziter Wartemechanismus

Auch wenn dies mal eine gute Lösung sein kann, ist es das in den meisten Fällen nicht. Es gibt Fälle, in denen Automatisierer so viele explizite Wartemechanismen benutzt haben, dass die automatisierten Tests langsamer liefen und mehr Zeit benötigten als ein manueller Tester, der genau dieselben Tests ausführt.

Diese Art von Wartemechanismus zielt immer auf den schlimmstmöglichen Fall ab. Da der schlimmstmögliche Fall nicht so oft auftritt, ist die meiste Wartezeit einfach verschwendete Zeit.

Für diesen Lehrplan wird davon ausgegangen, dass die ***sleep ()***-Funktion nur für Übungszwecke beim Debuggen benutzt wird; ansonsten sollte sie nicht verwendet werden.

Selenium mit WebDriver hat zwei Hauptarten von Wartemechanismen: implizites Warten (engl. implicit wait) und explizites Warten (explicit wait). Im Lehrplan wird das explizite Warten genauer behandelt (siehe weiter unten), aber zunächst kurz zum impliziten Warten.

Eine implizite Wartezeit in WebDriver wird angewendet, wenn das WebDriver-Objekt zum ersten Mal erstellt wird. Der folgende Code erstellt den Treiber und legt die implizite Wartezeit fest:

```
driver = webdriver.Chrome()
driver.implicitly_wait(10)
```

Abb. 78: Implizites Warten setzen

Die implizite Wartezeit, wie oben definiert, ist wirksam, bis der WebDriver nicht mehr existiert. Diese implizite Wartezeit weist WebDriver an, das DOM für eine bestimmte Zeit abzufragen, während dieses versucht, ein Element zu finden, das nicht sofort gefunden wird. Die Standardeinstellung für die Wartezeit ist 0 Sekunden. Jedes Mal, wenn ein Element nicht sofort gefunden wird, fordert der Code im obigen Beispiel WebDriver auf, zehn Sekunden lang alle paar Millisekunden abzufragen: „Bist du schon da?“ Wenn das Element innerhalb dieses Zeitraums gefunden wird, fährt das Skript den Testlauf fort. Die implizite Wartezeit funktioniert für alle Elemente bzw. für alle Elemente, die im Skript definiert sind.

Explizite Wartezeiten erfordern, dass der Automatisierer genau definiert (normalerweise für ein bestimmtes Element), wie lange WebDriver auf dieses bestimmte Element warten soll, bzw. wie lange WebDriver auf einen bestimmten Zustand dieses Elements warten soll. Wie oben erwähnt, ist

der extreme Fall eines expliziten Wartens die ***sleep ()***-Funktion. Die Verwendung dieser Funktion wird als eine eher grobe Methode angesehen.

Abgesehen von der ***sleep ()***-Funktion sind explizite Wartezeiten einfach zu handhaben, da Python-, Java- und C#-Bindungen bequeme Methoden enthalten, die mit dem Warten auf bestimmte erwartete Bedingungen funktionieren. In Python werden diese Wartemechanismen mithilfe der WebDriver-Methode ***WebDriverWait()*** in Verbindung mit ***ExpectedCondition*** codiert. Die erwarteten Bedingungen sind für viele allgemeine Bedingungen definiert, die auftreten können und auf die zugegriffen wird, indem das Modul *selenium.webdriver.support* hinzugefügt wird, siehe Beispiel:

```
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC
```

Abb. 79: Explizite Wartemethoden importieren

Mithilfe dieses Codes hat der Automatisierer die Möglichkeit, alle vordefinierten erwarteten Wartevorgänge zu verwenden, die verfügbar sind. Der Aufruf des expliziten Wartens kann mit dem folgenden Code erfolgen (vorausgesetzt, die obigen Importe wurden durchgeführt):

```
wait = WebDriverWait(driver, 10)
element = wait.until(EC.element_to_be_clickable((By.ID, 'someID')))
```

Dieser Code wartet bis zu 10 Sekunden und überprüft alle 500 Millisekunden, ob ein Element 'someID' über die ***ID*** identifiziert wurde. Wenn das WebElement nach 10 Sekunden nicht gefunden wird, löst der Code aufgrund der Zeitüberschreitung die Ausnahme *TimeoutException* aus. Falls das WebElement gefunden wird, wird ein Verweis darauf in der Variablen *element* platziert und die Automatisierung wird fortgesetzt.

Viele Testautomatisierer integrieren diesen Code in Wrapper-Funktionen, so dass jedes Element vor langsamen WebElementen geschützt ist.

Es gibt eine Vielzahl dieser erwarteten Bedingungen, die für die Verwendung in Python definiert sind, einschließlich:

- title_is
- title_contains
- presence_of_element_located
- visibility_of_element_located
- visibility_of
- presence_of_all_elements_located
- text_to_be_present_in_element
- text_to_be_present_in_element_value
- frame_to_be_available_and_switch_to_it

- invisibility_of_element_located
- element_to_be_clickable
- staleness_of
- element_to_be_selected
- element_located_to_be_selected
- element_selection_state_to_be
- element_located_selection_state_to_be
- alert_is_present

Es können auch benutzerdefinierte Wartebedingungen erstellt werden, aber das ist nicht Gegenstand dieses Lehrplans.

4.3 Page Objects

Bereits in Abschnitt 4.1 wurde empfohlen, wo immer möglich die Komplexität aus den Skripten zu entfernen und sie in der TAA bzw. dem TAF zu platzieren. Dieses Thema wird in diesem Abschnitt nun in Zusammenhang mit den Page Objects etwas eingehender behandelt, die für ein Entwurfsmuster (*Page Object Pattern*) repräsentativ sind.

Ein Page Object stellt einen Bereich in der Oberfläche einer Webanwendung dar, mit dem der (automatisierte) Test interagieren soll. Es gibt drei Hauptgründe, weshalb Page Objects verwendet werden sollten:

- Es wird wiederverwendbarer Code erstellt, der von mehreren Testskripten gemeinsam genutzt werden kann
- Es reduziert die Menge an dupliziertem Code
- Es reduziert den Kosten- und Wartungsaufwand für Testautomatisierungsskripte
- Es kapselt alle Operationen auf der GUI des SUT in einer Schicht
- Es liefert eine klare Trennung zwischen den geschäftlichen und den technischen Teilen für den Testautomatisierungsentwurf
- Wenn Änderungen auftreten (was unvermeidlich ist), gibt es einen einzigen Punkt, um alle relevanten Skripte zu ändern

Page Objects und Page Object Patterns wurden viele Male in diesem Lehrplan erwähnt. Page Object Pattern bedeutet die Verwendung von Page Objects in der Testautomatisierungsarchitektur, so dass die beiden Begriffe fast austauschbar verwendet werden können.

Eines der Kernprinzipien beim Aufbau einer wartbaren TAA besteht darin, diese in Schichten zu unterteilen. Eine dieser Schichten ist gemäß Definition des ISTQB® TAE-Lehrplans die Test-Abstraktionsschicht in der generischen TAA. Diese Schicht sorgt für die Verbindung zwischen der Logik des Testfalls und den physikalischen Anforderungen der SUT-Ausführung. Page Objects sind Teil dieser Abstraktionsschicht; sie abstrahieren normalerweise die Benutzeroberfläche des SUT.

Abstrahieren bedeutet hier, Details darüber, wie die GUI gesteuert wird, in Funktionen zu verbergen, die in anderen Skripten verwendet werden. Das folgende Beispiel zeigt ein Codestück, das aus einem Skript ohne Verwendung von Page Object Patterns stammen könnte. Dieser Code ist offensichtlich nicht leicht zu lesen. Auch sind die Selektoren für bestimmte Steuerelemente eng mit dem Code verknüpft.

```
first_name = self.browser.find_element_by_css_selector("#id_first_name")
first_name.send_keys("Clem")
last_name = self.browser.find_element_by_css_selector("#id_last_name")
last_name.send_keys("Kaddidlehopper")
password = self.browser.find_element_by_css_selector("#id_password")
password.send_keys("QWERTY")
email = self.browser.find_element_by_css_selector("#id_email")
email.send_keys("test+43@example.com")
product = self.browser.find_element_by_css_selector("#id_the_product")
product.send_keys("test")
self.browser.find_element_by_css_selector('#create_account_form button').click()
self.browser.find_element_by_css_selector('#next-step').click()
self.browser.find_element_by_css_selector('#next-step').click()
self.browser.find_element_by_css_selector('#next-step').click()
```

Abb. 80: Etwas „schwacher" Code ohne Verwendung von Page Objects

Dieser Code würde nach der Erstellung eines Page Objects wie folgt aussehen:

```
signup_form = homepage.getSignupForm()
signup_form.setName("Clem", "Kaddidlehopper")
signup_form.setPassword("QWERTY")
signup_form.setEmail('test+43@example.com')
signup_form.setProductName('test')
onboarding_1 = signup_form.submit()
onboarding_2 = onboarding_1.next()
onboarding_3 = onboarding_2.next()
items_page = onboarding_3.next()
```

Abb. 81: Der gleiche Code mit Verwendung von Page Objects

Sehr viel lesbarer und prägnanter!

Alle Teile, die das erste Codebeispiel so unübersichtlich machen, existieren noch. Sie wurden nur in das Page Object gekapselt und tauchen im zweiten Codebeispiel als Funktionsaufrufe auf. Das Page Object würde ungefähr so aussehen:

```
class SignupPage(BasePage):
    url = "http://localhost:8000/account/create/"

    def setName(self, first, last):
        self.fill_form_by_id("id_first_name", first)
        self.fill_form_by_id("id_last_name", last)

    def setEmail(self, email):
        self.fill_form_by_id("id_email", email)

    def setPassword(self, password):
        self.fill_form_by_id("id_password", password)
        self.fill_form_by_id("id_password_confirmation", password)

    def setProductName(self, name):
        self.fill_form_by_id("id_first_product_name", name)

    def submit(self):
        self.driver.find('#create_account_form button').click()
        return OnboardingInvitePage(self.driver)
```

Abb. 82: Beispiel eines Page Objects

Ein Großteil der Komplexität aus dem Skript wurde abstrahiert und in die TAA verschoben. Die Komplexität ist immer noch da – keine Frage. Die Komplexität ermöglicht es, nützliche Automatisierung zu erzeugen, und Komplexität wird benötigt. Indem diese im TAA verborgen wird, lassen sich korrekte und leistungsstarke Testskripte einfacher und schneller erstellen.

Dieser Umgang mit Komplexität ist nicht neu, sondern ist eine Best Practice beim Programmieren. Zu Beginn vor vielen Jahren erfolgte die Programmierung durch Anschließen von Patchkabeln, die Eingänge direkt mit dem Prozessor verbanden. Später wurde dann alles in Assembler programmiert, ein Code, der im Wesentlichen die gleiche Sprache wie der Computerprozessor sprach, aber keine physischen Verbindungen benötigte. Dann kamen höhere Sprachen, wie Fortran, Cobol, C, die viel einfacher zu programmieren waren. Der Code wurde kompiliert, um den Objektcode zu erstellen, der mit dem Prozessor arbeitete. Die Programmierer mussten den Prozessormikrocode nicht kennen. Dann kamen objektorientierte Sprachen (C++, Delphi, Java), die das Programmieren noch einfacher machten.

In diesem Lehrplan wird Python verwendet, das den größten Teil der Komplexität des Umgangs mit den Browser-Objekten verbirgt. Die Verwendung des Page Object Patterns geht nur einen Schritt weiter.

Ein Page Object ist eine Klasse, ein Modul oder ein Satz von Funktionen, das die Schnittstelle zu einem Formular, einer Seite oder einem Teil einer Seite des SUT enthält, die ein Testautomatisierer steuern möchte.

Ein Page Object exportiert Geschäftsvorgänge, die als Testschritte in der Testausführungsschicht verwendet werden können. Ein Page Object Pattern ist eigentlich eine der Implementierungen des

schlüsselwortgetriebenen Testens, mit dem der Wartungsaufwand eines Projekts reduziert werden kann. Der schlüsselwortgetriebene Test wird im nächsten Abschnitt behandelt.

Page Object Patterns sind insbesondere hinsichtlich der Pflege von Testskripten besonders wichtig, denn sie können den Zeitaufwand für die Aktualisierung der Skripte nach Änderungen an der GUI des SUT erheblich reduzieren. Es ist zu beachten, dass Page Object Patterns kein Allheilmittel für alle Probleme bei der Wartung von Testautomatisierungen sind. Auch wenn sich damit in manchen Situationen tatsächlich Kosten reduzieren lassen, beispielsweise beim Ändern der Logik eines SUT, kann das Aktualisieren von Testskripten immer noch viel Zeit in Anspruch nehmen. Im Übrigen ist dies kein spezifisches Problem der Automatisierung. Wenn sich die Logik des SUT ändert, müssten auch manuelle Testfälle geändert werden. Glücklicherweise kommt es nicht oft vor, dass die SUT-Logik geändert wird, da dies bedeuten würde, dass die Benutzer der Anwendung ihre Verwendung neu erlernen müssten.

Bei der Unterteilung der TAA in Schichten und beim Entwurf von Page Objects sollten die folgenden allgemeinen Regeln beachtet werden:

- Page Objects sollten keine geschäftlichen Zusicherungen (engl. assertions) oder Verifizierungspunkte enthalten
- Alle Assertions und technischen Verifizierungen bezüglich der GUI (z.B. prüfen, ob eine Seite vollständig geladen wurde) sollten nur innerhalb von Page Objects ausgeführt werden
- Alle Wartemechanismen sollten in Page Objects gekapselt sein
- Nur das Page Object sollte Aufrufe an Selenium-Funktionen enthalten
- Ein Page Object muss nicht die gesamte Seite oder das ganze Formular abdecken. Es kann einen Abschnitt oder einen anderen spezifischen Teil davon steuern.

Bei einer gut entworfenen TAA kapselt ein Page Object in einer Testabstraktionsschicht Aufrufe an Selenium WebDriver-Methoden, sodass die Testausführungsschicht nur mit der Geschäftslogik befasst ist. In solchen (gut entworfenen) TAAs gibt es keinen Import oder keine Verwendung der Selenium-Bibliothek in der Testausführungsschicht.

4.4 Schlüsselwortgetriebener Test

In Abschnitt 1.2 dieses Lehrplans wurde die Wirksamkeit von manuellen Testfällen behandelt und folgende Aussage gemacht:

> *Ein auf das Mindestmaß reduziertes manuelles Testskript hat meistens nur Informationen in drei Spalten. Die erste Spalte enthält eine abstrakte Aufgabe; in abstrakter Form deshalb, damit diese nicht geändert werden muss, wenn sich die Software ändert. Zum Beispiel ist die Aufgabe „Datensatz zur Datenbank hinzufügen“ eine abstrakte Aufgabe, die unabhängig von Version und Art der Datenbank ausgeführt werden kann - vorausgesetzt, ein manueller Tester verfügt über das Domänenwissen, um diese in konkrete Aktionen umzuwandeln.*

Die Idee einer *abstrakten Aufgabe* hat viel Potenzial; zum Beispiel die Aufgabe, ein Dokument in einem Textverarbeitungsprogramm zu öffnen. Nennen wir diese Aufgabe *OpenFile*. Dateien mussten von Textverarbeitungsprogrammen in DOS, in Windows 3.1., in Windows 95, in Macintosh, in Unix geöffnet werden können. Die Aufgabe musste in jeder Version jedes Textverarbeitungsprogramms, das je produziert wurde, ausgeführt werden. „Wie“ eine Datei geöffnet wird, musste für jedes jemals erstellte Textverarbeitungsprogramm betrachtet werden; aber das „Was” bleibt immer gleich. *OpenFile* ist ein Prototyp für ein Schlüsselwort.

Das Kernstück von KDT (= Keyword Driven Testing, deutsch *schlüsselwortgetriebener Test*) ist die Idee, dass jede Anwendung eine Reihe verschiedener Aufgaben hat, die zur Verwendung der Anwendung ausgeführt werden müssen. Dateien werden geöffnet und gespeichert. Datenbankeinträge werden erstellt, gelesen, aktualisiert und gelöscht. Diese Aufgaben sind tendenziell eine abstrakte Darstellung dessen, was Benutzer tun müssen, um mit der Software zu interagieren. Da es sich um abstrakte Ideen handelt, ändern sie sich selten, wenn die Versionen der Software aktualisiert werden.

Manuelle Tests nutzen diese Abstraktion aus; die funktionalen Aufgaben für fast jede Anwendung ändern sich selten. Sobald ein Benutzer die Schnittstelle erlernt, bleiben die funktionalen Aufgaben, die dieser mit der Anwendung ausführen muss, fast immer gleich. Was sich aber ändern kann, ist die Art und Weise, wie die Benutzer die einzelnen Aufgaben ausführen; die Konzepte hinter den Aufgaben ändern sich jedoch nicht. Manuelle Tests müssen nicht bei jeder Version geändert werden, da das „Was“, das wir testen, im Vergleich zum „Wie“ sehr stabil ist.

Die erstellten Schlüsselworte sind im Wesentlichen eine Metasprache, die zum Testen verwendet werden kann. Wie manuelle Tester, die manuelle Testskripte schreiben, werden auch hier die Aufgaben (das „Was“) in der Software identifiziert und mit einem Begriff umschrieben, dem sog. Schlüsselwort. Es sind die Test Analysts, die idealerweise die Schlüsselworte definieren, die sie zum Testen einer Anwendung benötigen.

Eine der wichtigsten Voraussetzungen für die Implementierung wartbarer, manueller oder automatisierter Testskripte ist eine gute Struktur, bei der die auszuführenden Aufgaben abstrakt bleiben und sich selten ändern. Auch hier zeigt sich, dass die Trennung der physikalischen Schnittstelle von der abstrakten Aufgabe ein ausgezeichnetes Entwurfsverfahren ist.

In Kapitel 1.5 wurde bei der Beschreibung der Testautomatisierungsarchitektur die klar definierte Einteilung der Testautomatisierung in folgende Schichten erwähnt: der Testfall, der in der Testdefinitionsebene abstrakt beschrieben wird; die Testadaptionsschicht, die die Schnittstelle zwischen dem Testfall und der physischen GUI des SUT bildet; die Testausführungsschicht, die das Werkzeug enthält, das die Tests des SUT ausführt.

Das ISTQB® definiert KDT als ein in skriptbasiertes Verfahren, das nicht nur Testdaten und vorausgesagte Ergebnisse aus Dateien einliest, sondern auch spezielle Schlüsselworte (die

abstrakten Aussagen darüber, „was“ das System tun soll). Schlüsselworte werden oft auch als *Aktionsworte* bezeichnet. Die Definition des ISTQB® besagt weiter, dass die Schlüsselworte von speziellen Skripten interpretiert werden können und den Test während der Laufzeit steuern.

Genauer betrachtet ist KDT ein skriptbasiertes Verfahren, was bedeutet, dass es ermöglicht, Tests zu automatisieren. Die KDT-Dateien enthalten Testdaten, erwartete Ergebnisse und Schlüsselworte.

Schlüsselworte benennen Geschäftsvorgänge oder Schritte eines Testfalls. Diese entsprechen genau der ersten Spalte bei manuellen Testfällen.

Wenn automatisierte Tests nach den Grundsätzen des schlüsselwortgetriebenen Tests implementiert werden, enthalten die Testfälle selbst keine spezifischen Aktionen, die am SUT auszuführen sind (das „Wie“ der Aktion). Die automatisierten Testfälle enthalten Folgen von aufeinanderfolgenden Testschritten, die üblicherweise abstrakte Geschäftsvorgänge auf hoher Ebene sind (z.B. „Einloggen ins System“, „Zahlung ausführen“, „ein Produkt finden“).

Die genauen physischen Aktionen, die auf dem SUT auszuführen sind, bleiben innerhalb der Implementierung der Schlüsselworte verborgen (z.B. in den Page Objects, wie im vorhergehenden Abschnitt beschrieben). Im Grunde bildet dies die Trennung zwischen einem manuellen Testfall und dem menschlichen Tester ab, der dem Test Kontext und Angemessenheit hinzufügt, während er den manuellen Test am GUI ausführt.

Dieser Ansatz hat mehrere Vorteile:

- Der Testfallentwurf ist vom SUT entkoppelt
- Es wird deutlich zwischen den Testausführungs-, Testabstraktions- und Testdefinitions-schichten unterschieden
- Eine fast vollständige Arbeitsteilung:
    - Test Analysts entwerfen Testfälle und schreiben Skripte unter Verwendung von Schlüsselworten, Daten und erwarteten Ergebnissen
    - Technical Test Analysts (Automatisierer) implementieren Schlüsselworte und das Testautomatisierungs-Framework, das zum Ausführen der Tests benötigt wird
- Wiederverwendung von Schlüsselworten in verschiedenen Testfällen
- Verbesserte Lesbarkeit von Testfällen (sie sehen fast aus wie manuelle Testfälle)
- Weniger Redundanz
- Niedrigere Wartungskosten und -aufwand
- Wenn die Automatisierung offline ist, kann ein Tester den Test manuell direkt anhand des automatisierten Skripts durchführen
- Da Tests mit Schlüsselworten abstrakt sind, können die Skripte, die Schlüsselworte verwenden, lange vor dem Testen des SUT geschrieben werden (genau wie bei manuellen Tests)

- Der vorige Punkt bedeutet auch, dass die Automatisierung früher zur Verfügung steht, und nicht nur für Regressionstests, sondern auch schon vorher für Funktionstests eingesetzt werden kann
- Einige wenige technische Testautomatisierer können eine unbegrenzte Anzahl von Test Analysts unterstützen, wodurch die Architektur vollständig skalierbar wird
- Da die Skripte vollständig von der Implementierungsebene getrennt sind, können verschiedene Werkzeuge austauschbar verwendet werden

Manchmal verwenden Schlüsselwort-Bibliotheken andere Schlüsselworte, so dass die gesamte Schlüsselwortstruktur ziemlich komplex werden kann. In einer solchen Struktur haben manche Schlüsselworte ein hohes Abstraktionsniveau, z.B. *„Fügen Sie der Rechnung ein Produkt X mit dem Preis Y und der Menge Z hinzu“*, und manche haben ein ziemlich niedriges Abstraktionsniveau, z.B. *„Click >>Cancel<< button”*.

Daher haben Schlüsselworte in der allgemeinen Testautomatisierungsarchitektur mehrere Verwendungsmöglichkeiten:

- Schlüsselworte mit niedrigem Abstraktionsniveau sind als Page Objects in der Testadaptionsschicht implementiert; sie führen die Aktionen im SUT aus
- Schlüsselworte mit niedrigem Abstraktionsniveau werden als Teil von Schlüsselworten mit hohem Abstraktionsniveau in der Testausführungsschicht verwendet
- Schlüsselworte mit hohem Abstraktionsniveau werden in der Testbibliothek implementiert
- Schlüsselworte mit hohem Abstraktionsniveau werden als Testschritte in den Testprozeduren in der Testausführungsschicht verwendet

Die ISTQB®-Definition des KDT besagt, dass Schlüsselworte von speziellen Skripten interpretiert werden. Das trifft zu, wenn der schlüsselwortgetriebene Test mithilfe von speziellen Tools wie Cucumber oder RobotFramework implementiert wird.

Es ist jedoch auch möglich, die Grundsätze des schlüsselwortgetriebenen Tests zu befolgen, wenn das TAF mit universellen Programmiersprachen wie Python, Java oder C# implementiert wird. In diesem Fall wird jedes Schlüsselwort zum Aufruf einer Objektfunktion oder -methode der Testbibliothek verwendet.

In der Praxis gibt es zwei Möglichkeiten, KDT zu implementieren. Die erste, das klassische KDT, wie von Dorothy Graham in ihrem Buch[2] *Software Test Automation*, beschrieben, ist ein Top-Down-Ansatz. Der erste Schritt besteht darin, Testfälle so zu entwerfen, wie sie für manuelle Tests entwickelt werden. Dann werden die Schritte dieser Testfälle als High-Level-Schlüsselworte abstrahiert, die dann im nächsten Schritt in Low-Level-Schlüsselworte zerlegt oder in einem Werkzeug oder einer Programmiersprache implementiert werden können. Diese Methode ist am wirtschaftlichsten, wenn die Testbibliothek bereits viele Funktionen/Methoden enthält, die

[2] „Software Test Automation”, Mark Fewster and Dorothy Graham, Addison-Wesley Professional, 1999

Aufgaben im SUT ausführen. Es kann auch nützlich sein, wenn automatisierte Skripte von manuellen Testfällen konvertiert werden und nicht von Grund auf neu geschrieben werden.

Die zweite Möglichkeit beim KDT ist die Implementierung von Skripten in einem Bottom-up-Ansatz. Dies bedeutet, dass Skripte von einem Werkzeug (z.B. Selenium IDE) aufgezeichnet und dann zu einer geeigneten KDT-Testautomatisierungsarchitektur umstrukturiert / neu strukturiert werden. Dieser Ansatz ermöglicht es Testteams, schnell mehrere Testautomatisierungsskripte zu schreiben und sie im SUT auszuführen.

Wenn jedoch mit diesem groben Ansatz mehr als zwanzig bis dreißig Skripte erstellt werden, ohne diese in ein gut entworfenes TAF aufzunehmen, wird das Automatisierungsprojekt in der Zukunft dem Risiko hoher Wartungsanstrengungen und -kosten ausgesetzt sein. Darüber hinaus besteht bei Tests, die von Anfang an ohne manuelle Testfälle geschrieben werden, das Risiko, dass sie keine guten Tests werden (d.h. evtl. werden wichtige Risiken, die getestet werden müssen, nicht abgedeckt).

Bei der Implementierung des TAF auf Basis von KDT-Grundsätzen sind auch die folgenden Aspekte zu beachten:

- Fein granulierte Schlüsselworte ermöglichen spezifischere Szenarien, allerdings auf Kosten der Komplexität der Skriptwartung
- Je feiner granuliert ein Schlüsselwort ist, desto wahrscheinlicher ist es eng mit der Schnittstelle des SUT verknüpft und desto weniger abstrakt ist es (z.B. das erwähnte Beispiel *„Click >>Cancel<< button"* für ein Schlüsselwort mit ziemlich niedrigem Abstraktionsniveau)
- Der anfängliche Schlüsselwort-Entwurf ist wichtig, aber letztlich werden neue und andere Schlüsselworte benötigt, die sowohl die Geschäftslogik als auch die Automatisierungsfunktionalität für die Ausführung beinhalten

Es hat sich gezeigt, dass KDT, wenn es richtig gemacht wird, ein guter Automatisierungsansatz ist, der Skripte mit konstant niedrigen Wartungskosten erzeugen kann. Dies ermöglicht den Aufbau gut strukturierter Testautomatisierungs-Frameworks und verwendet dabei die Best Practices des Softwareentwurfs.

Anhang – Glossar der Selenium-Begriffe

Klassenattribut: Ein HTML-Attribut, das auf eine Klasse in einem CSS-Stylesheet verweist. Es kann auch von einem JavaScript verwendet werden, um Änderungen an HTML-Elementen mit einer bestimmten Klasse vorzunehmen

Komparator: Ein Werkzeug, um den Vergleich der erwarteten Ergebnisse mit den tatsächlichen Ergebnissen zu automatisieren

CSS-Selektor: Selektoren sind Muster, die die HTML-Elemente ansprechen, die Sie stylen möchten

Document Object Model (DOM, deutsch „Dokumenten-Objekt-Modell"): Eine Anwendungsprogrammierschnittstelle, die ein HTML- oder XML-Dokument als Baumstruktur behandelt, wobei jeder Knoten ein Objekt ist, das einen Teil des Dokuments darstellt

Fixture (sinngem. *festes Inventar*): Ist ein Mock-Objekt oder eine Umgebung, die verwendet wird, um einen Gegenstand, ein Gerät oder eine Software konsistent zu testen

Framework: Stellt eine Umgebung bereit, in der automatisierte Testskripte ausgeführt werden können, einschließlich der Werkzeuge, Bibliotheken und Test-Fixtures

Funktion: Eine Python-Funktion ist eine Gruppe wiederverwendbarer Anweisungen, die eine bestimmte Aufgabe ausführen

Hook: Eine Schnittstelle, die in ein System eingebaut wird und hauptsächlich dazu dient, eine verbesserte Testbarkeit für dieses System bereitzustellen

HTML (HyperText Markup Language): Die Standardauszeichnungssprache zum Erstellen von Webseiten und Webanwendungen

ID: Ein Attribut, das eine eindeutige Identifikationszeichenkette (-String) für ein HTML-Element angibt. Der Wert muss innerhalb des HTML-Dokuments eindeutig sein

I-frame: Ein HTML-Inlineframe zum Einbetten eines anderen Dokuments in ein HTML-Dokument

Modaler Dialog: Ein Dialogfenster (bzw. eine Dialogbox), die den Benutzer zwingen, mit ihm zu interagieren, bevor auf den Bildschirm darunter zugegriffen werden kann

Page Object Pattern: Ein Entwurfsmuster in der Testautomatisierung, das erfordert, dass technische Logik und Geschäftslogik auf verschiedenen Ebenen behandelt werden

Pestizidparadoxon: Ein Phänomen, bei dem derselbe Test, wenn er mehrfach wiederholt wird, immer weniger Defekte findet

Persona: Ein Benutzerprofil, das erstellt wird, um einen Benutzertyp darzustellen, der auf eine übliche Weise mit einem System interagiert

pytest: Ein Python-Test-Framework

Tag: HTML-Elemente werden durch Tags (deutsch: Auszeichner) gekennzeichnet, die in spitzen Klammern geschrieben sind

Technische Schuld: Impliziert die zusätzlichen Kosten für Nachbesserungen, die dadurch verursacht werden, dass ein schlechter Entwurf oder schlechte Implementierungen auf kurze Sicht lieber ignoriert werden

WebDriver: Die Schnittstelle, mit der Selenium-Tests geschrieben (und ausgeführt) werden. Verschiedene Browser können über verschiedene Java-Klassen (z.B. ChromeDriver, FirefoxDriver, usw.) gesteuert werden

Wrapper: Eine Funktion in einer Softwarebibliothek, deren Hauptzweck darin besteht, eine andere Funktion aufzurufen, häufig um Funktionalität hinzuzufügen oder zu erweitern, dabei aber die Komplexität zu verstecken

XML (eXtensible Markup Language): Eine Auszeichnungssprache, die einen Satz von Regeln für die Kodierung von Dokumenten in einem Format definiert, das maschinenlesbar und auch für Menschen gut lesbar ist

XPath (XML Path Language): Eine Abfragesprache zum Auswählen von Knoten aus einem XML-Dokument