Testautomatisierung mit Selenium

- Teilautomatisierte Generierung von Page Objects -

Fakultät für Informatik und Mathematik
der Hochschule München

Masterarbeit

vorgelegt von

Matthias Karl
Matrikel-Nr: 03280712

am 29.01.2016

Prüfer:	Prof. Dr. Ullrich Hafner
Zweitprüfer:	Prof. Dr. Oliver Braun

Eidesstattliche Erklärung

Hiermit versichere ich, die vorliegende Studienarbeit selbstständig und nur unter Verwendung der von mir angegebenen Quellen und Hilfsmittel verfasst zu haben. Sowohl inhaltlich als auch wörtlich entnommene Inhalte wurden als solche kenntlich gemacht. Die Arbeit hat in dieser oder vergleichbarer Form noch keinem anderem Prüfungsgremium vorgelegen.

Datum: _____ Unterschrift: _____

Zusammenfassung / Abstract

Diese Arbeit befasst sich mit dem Themenschwerpunkt Testautomatisierung. Zu Beginn werden zunächst die Grundlagen im Bereich der Software-Qualität und dem Testen im allgemeinen geschaffen, um dann einen Überblick über die verschiedenen Möglichkeiten der Testautomatisierung vorzustellen. Anhand der geschaffenen Grundlagen wird das Testautomatisierungs-Tool Selenium vorgestellt. Hierbei liegt ein besonderes Augenmerk auf dem Page Object Design Pattern. Bezüglich des Page Object Design Pattern wird eine selbst erstellte Software-Lösung vorgestellt, welche die Verwendung dieses Pattern durch eine teilautomatisierte Generierung von Page Object-Klassen vereinfachen soll.

Inhaltsverzeichnis

Abbildungsverzeichnis

1. Einleitung

Software hat in der heutigen Zeit für Unternehmen und Privatpersonen an großer Bedeutung gewonnen. Die Komplexität der Softwareprodukte steigt stetig an, gleichzeitig auch die Anforderungen an die Qualität. Das hat zur Folge, dass bei der Entwicklung von Software ein immer größeres Augenmerk auf den Bereich des Testens gelegt wird. Vor allem die Testautomatisierung rückt hierbei immer mehr in den Vordergrund. Man verspricht sich dadurch eine stetige Qualitäts- und Effizienzsteigerung.
Eine Vielzahl der gängigen Software wird heutzutage in Form von Webanwendungen genutzt. Um diese automatisiert zu Testen, wird oft auf Testfälle zurückgegriffen, die automatisch Eingaben auf der Oberfläche der Anwendung tätigen, um anschließend das Verhalten der Anwendung zu überprüfen. Ein gängiges Tool, mit welchem diese Form der Testfälle umgesetzt werden kann, ist Selenium [Sel15a]. Selenium ist ein Tool, welches auch bei it@M, dem externen IT-Dienstleister der Landeshauptstadt München, für automatisierte Softwaretests eingesetzt wird. Im Rahmen dieser Arbeit soll eine Möglichkeit aufgezeigt werden, wie die Verwendung dieses Tools bei der Landeshauptstadt München einfacher und effizienter gestaltet werden kann.

1.1. Motivation

Testautomatisierung hat oft das Problem, dass Testfälle zwar wiederholt und einfach ausgeführt werden können, der initiale Mehraufwand für die Erstellung der Testfälle ist aber, verglichen mit manuellen Tests, häufig so hoch, dass die erreichten Vorteile nur gering zum Tragen kommen. Ein möglicher Weg, um Verbesserungen in der Testautomatisierung zu erzielen, ist es daher, diesen Mehraufwand zu minimieren.
Testfälle, die mit Hilfe des Selenium WebDiver entwickelt werden, leiden oftmals unter eben dieser Schwierigkeit. Die initiale Erstellung der Tests ist in der Regel sehr aufwändig. Das liegt unter anderem daran, dass bei der Verwendung des WebDrivers ein bestimmtes Design Pattern, das Page Object Pattern, verwendet wird. Von diesem Design Pattern verspricht

man sich möglichst wartbare und stabile Testfälle. Bestandteil des Pattern ist es, eine Vielzahl von sogenannten Page Objects zu erstellen, welche die einzelnen Seiten einer Webanwendung repräsentieren.
In Hinblick auf den initialen Mehraufwand weist die Erstellung dieser Page Object-Klassen ein großes Einsparungspotential auf. Aufgrund ihrer generischen Struktur bieten die Page Objects nämlich das Potential automatisch erzeugt zu werden. In Zusammenarbeit mit dem IT-Dienstleister der Landeshauptstadt München (it@M) soll daher eine Software entwickelt werden, mit deren Hilfe die Erstellung der Page Object-Klassen vereinfacht werden kann.

1.2. Roadmap

Der Hauptteil dieser Arbeit gliedert sich in vier Kapitel. In Kapitel 2 werden zunächst die Grundlagen in den Bereichen der Software-Qualität, dem Testen im Allgemeinen und der Testautomatisierung im Speziellen gelegt. Kapitel 3 geht näher auf die Testautomatisierung ein und soll einen Überblick über die verschiedenen Bereiche und Möglichkeiten geben, welche die Testautomatisierung bietet. Kapitel 4.1 beschäftigt sich mit dem Testautomatisierungstool Selenium und erläutert in diesem Zusammenhang das Page Object Pattern. In Kapitel 5 wird eine Softwarelösung vorgestellt mit deren Hilfe die Verwendung des Page Object Pattern unterstützt werden kann.

2. Grundlagen

2.1. Software-Qualität

Nahezu jeder Programmierer ist schon einmal mit dem Begriff der Software-Qualität in Berührung gekommen. Diesen Qualitätsbegriff jedoch genau zu erfassen, erweist sich als schwierig. Die DIN-ISO-Norm 9126 definiert Software-Qualität wie folgt:
„Software-Qualität ist die Gesamtheit der Merkmale und Merkmalswerte eines Software-Produkts, die sich auf dessen Eignung beziehen, festgelegte Erfordernisse zu erfüllen."[ISO01]
Im Jahr 2005 ist die DIN-ISO-Norm 9126 von der DIN-ISO-Norm 25010 [ISO11] abgelöst worden. Die Definition der neuen Norm unterscheidet sich von der alten Definition vor allem darin, dass nun die Erfüllung von Benutzerbedürfnissen und nicht mehr die Erfüllung von Erfordernissen im Vordergrund steht.
Folgt man der Analyse von Hoffmann [Hof13, vgl. S.6 ff.] wird deutlich, dass es sich bei dem Begriff der Software-Qualität um eine multikausale Größe handelt. Das bedeutet, dass zur Bestimmung der Qualität einer Software nicht nur ein einzelnes Kriterium existiert. Vielmehr verbergen sich hinter dem Begriff eine ganze Reihe verschiedener Kriterien, die je nach den gestellten Anforderungen in ihrer Relevanz variieren. Sammlungen solcher Kriterien werden in sogenannten Qualitätsmodellen zusammengefasst. Die DIN-ISO-Norm 25010 [ISO11] bietet ein solches Qualitätsmodell und definiert damit eine Reihe von wesentlichen Merkmalen, die für die Beurteilung der Software-Qualität eine Rolle spielen. Die Merkmale der DIN-ISO-Norm 25010 [ISO11] sind in Abbildung 2.1 zusammengefasst. Eine nähere Definition der einzelnen Begriffe des Qualitätsmodells kann beispielsweise dem Buch 'Software-Qualität' von Dirk W. Hoffmann [Hof13, S.7 ff.] entnommen werden.
Um die Qualität einer Software zu steigern, bietet die moderne Software-Qualitätssicherung laut Hofmann [Hof13, vgl. S.19 ff.] eine Vielzahl von Methoden und Techniken: Ein Teil der Methoden versucht durch eine Verbesserung des Prozesses der Produkterstellung die Entstehung von qualitativ hochwertigen Produkten zu begünstigen. Diese Methoden fallen in den Bereich der Prozessqualität. Einen weiteren Bereich bilden die Methoden, die zur Verbesserung der Produktqualität dienen. Bei diesen Methoden wird das Softwareprodukt direkt

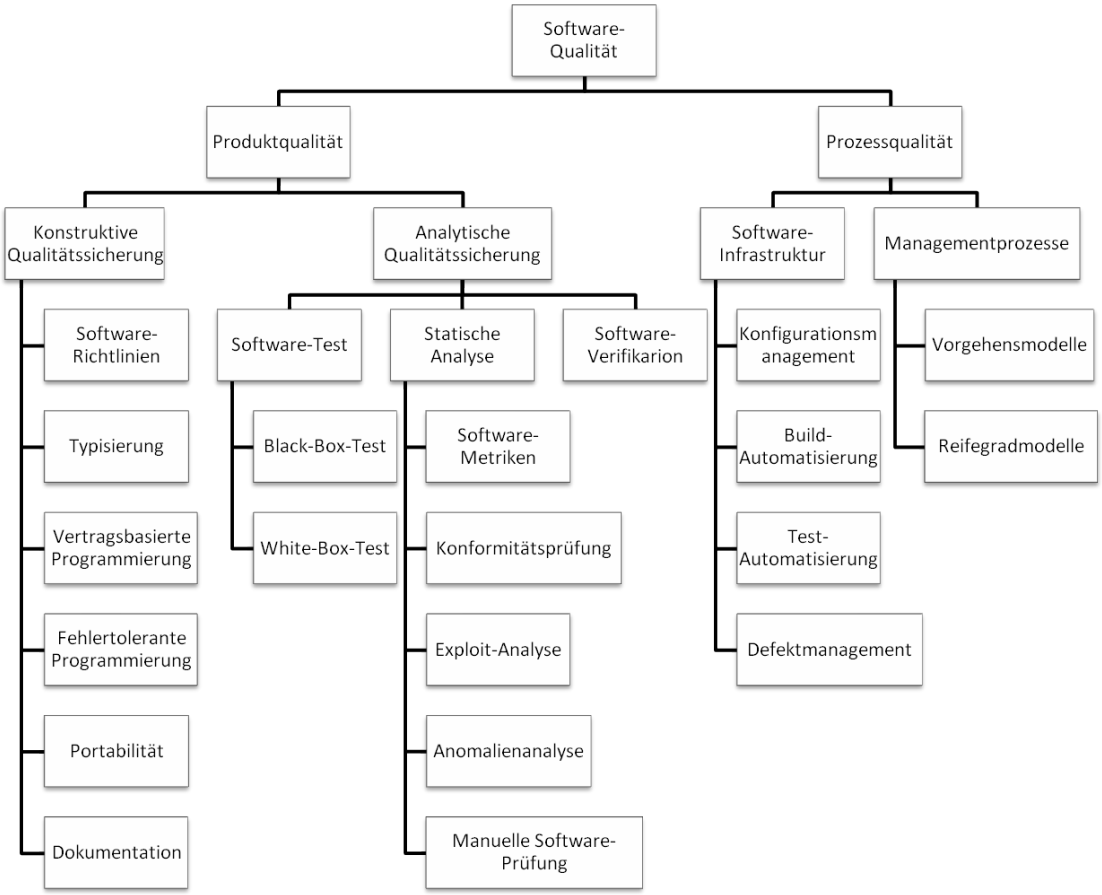
Quelle: [ISO11]

Abbildung 2.1.: Qualitätsmerkmale von Softwaresystemen (ISO 25010)

bezüglich der Qualitätsmerkmale überprüft. Dieser Bereich unterteilt sich in die konstruktive und analytische Qualitätssicherung. Unter konstruktiver Qualitätssicherung versteht man z.B. den Einsatz von Methoden, Werkzeugen oder Standards, die dafür sorgen, dass ein Produkt bestimmte Forderungen erfüllt. Unter analytischer Qualitätssicherung versteht man den Einsatz von analysierenden bzw. prüfenden Verfahren, die Aussagen über die Qualität eines Produkts machen. In diesem Bereich der Qualitätssicherung befindet sich unter anderem der klassische Software-Test. Eine Übersicht über das gesamte Gebiet der Software-Qualitätssicherung, wie es sich uns gegenwärtig darstellt, ist in Abbildung 2.2 dargestellt.

2.2. Softwaretest

Im Laufe der Zeit wurden viele Versuche unternommen, um die Qualität von Software zu steigern. Besondere Bedeutung hat hierbei der Software-Test erlangt. Der IEEE Standard 610.12 definiert den Begriff Test als das Ausführen einer Software unter bestimmten Bedingungen mit dem Ziel, die erhaltenen Ergebnisse auszuwerten, also gegen erwartete Werte zu vergleichen. (Im Original: „An activity in which a system or component is executed under specific conditions, the results are observed or recorded, and an evaluation is made of some aspect of the system or component.“ [IEE91])
Bereits zu Beginn der Softwareentwicklung wurde versucht, Programme vor ihrer Ausliefe-



Quelle: [Hof13, vgl. S.20]

Abbildung 2.2.: Übersicht über das Gebiet der Software-Qualitätssicherung

rung zu testen. Der dabei erzielte Erfolg entsprach nicht immer den Erwartungen. Im Laufe der Jahre wurde das Testen daher auf eine immer breitere Grundlage gestellt. Es entwickelten sich Unterteilungen des Software-Tests, die bis heute Bestand haben. Thaller [Tha02, vgl. S.18] nennt hier beispielsweise:

- White-Box-Test
- Black-Box-Test und externe Testgruppe
- Volume Test, Stress Test und Test auf Systemebene

Jeder dieser Begriffe beschreibt bestimmte Techniken, die bei konsequenter Anwendung dazu führen können, Fehler in Softwareprodukten zu identifizieren.
Nach Hoffmann [Hof13, vgl. S.22] spielt neben der Auswahl der richtigen Techniken für ein bestimmtes Problem in der Praxis die Testkonstruktion eine zentrale Rolle. Bereits für kleine

Programme ist es faktisch nicht mehr möglich das Verhalten der Software für alle möglichen Eingaben zu überprüfen. Es muss sich daher immer auf eine vergleichsweise geringe Auswahl an Testfällen beschränkt werden. Testfälle unterscheiden sich jedoch stark in ihrer Relevanz. Die Auswahl der Testfälle hat daher einen großen Einfluss auf die Anzahl der gefundenen Fehler und damit auch auf die Qualität des Endprodukts.
Laut Hofmann [Hof13, vgl. S.22] ist der Software-Test eine der verbreitetsten Techniken zur Verbesserung der Software-Qualität. Um über lange Sicht gute Software produzieren zu können, reicht es jedoch nicht aus, sich nur auf eine Technik der Software-Qualitätssicherung zu stützen. Ein großer Nachteil des Software-Tests ist laut Thaller [Tha02, vgl. S.18], dass Fehler erst in einer relativ späten Phase der Entwicklung identifiziert werden. Je später ein Fehler jedoch erkannt wird, desto teurer wird auch seine Beseitigung. Abbildung 2.2 zeigt, dass der Software-Test nur eine von vielen Techniken des Qualitätsmanagement darstellt. Um eine möglichst qualitativ hochwertige Software zu erhalten, ist es daher ratsam, sich bei der Qualitätssicherung möglichst breit aufzustellen und sich nicht nur auf die analytische Qualitätssicherung in Form des Software-Tests zu verlassen.

2.3. Testautomatisierung

Das Testen von Software macht in heutigen Projekten einen großen Teil der Projektkosten aus. So sprechen beispielsweise Harrold [Har00] und auch Ramler [RW06] davon, dass das Testen für 50% und mehr der gesamten Projektkosten verantwortlich sein kann. Mit steigender Komplexität der Software müssen, um eine konstante Qualität der Software zu gewährleisten, immer höhere Ausgaben für das Testen getätigt werden. Um diese Kosten zu reduzieren, haben sich im Laufe der Zeit die bestehenden Testmethoden immer weiter entwickelt und auch neue Ansätze herausgebildet. Harrold [Har00] beschreibt als einen Ansatz, Software-Tests möglichst automatisiert durchzuführen. Diesen Ansatz fast man mit dem Begriff Testautomatisierung zusammen. Seidl et al. [SBB12, S.7] definieren Testautomatisierung als „die Durchführung von ansonsten manuellen Testtätigkeiten durch Automaten.“ Diese Definition zeigt, dass das Spektrum der Testautomatisierung breit gefächert ist. Testautomatisierung beschränkt sich nicht nur auf das automatisierte Durchführen von Testfällen, sondern erstreckt sich über alle Bereiche des Software-Tests. Die verschiedenen Möglichkeiten der Testautomatisierung werden in Kapitel 3.2 dargestellt. Aus Sicht des Qualitätsmanagement ist die Testautomatisierung sowohl den Methoden zur Steigerung der Produktqualität als auch der Prozessqualität zugeordnet. Ein automatisierter Software-Test hat immer noch den selben Charakter wie ein manueller Software-Test und ist daher ein Teil der analytischen Quali-

tätssicherung. Allerdings erfordert Testautomatisierung laut Hoffmann [Hof13, vgl. Seite 25] auch immer infrastrukturelle Anpassungen. Automatisierte Testfälle benötigen in der Regel eine besondere Software-Infrastruktur, wie etwa ein Automatisierungsframework. Solche Maßnahmen, die den Programmentwickler aus technischer Sicht in die Lage versetzen, seiner täglichen Arbeit in geregelter und vor allem produktiver Weise nachzugehen, werden den Methoden zur Verbesserung der Prozessqualität zugeordnet (siehe Abbildung 2.2).

2.4. Testprozess

Um Software-Tests effektiv und strukturiert durchzuführen, wird ein feiner Ablaufplan für die einzelnen Testaufgaben benötigt. Diesen Ablaufplan fassen Splinner und Linz [SL07] im fundamentalen Testprozess zusammen. Die einzelnen Arbeitsschritte, die im Lebenszyklus eines Software-Tests anfallen, werden dabei verschiedenen Phasen zugeordnet. Durch den Testprozess wird die Aufgabe des Testens in kleinere Testaufgaben untergliedert.
Testaufgaben, die Splinner und Linz [SL07] dabei unterscheiden sind:

- Testplanung und Steuerung
- Testanalyse und Testdesign
- Testrealisierung und Testdurchführung
- Testauswertung und Bericht
- Abschluss der Testaktivitäten

Laut Seidl et al. [SBB12, S. 9] wird jeder, der strukturiert testet, diese Aktivitäten auf die eine oder andere Weise abbilden.
„Obgleich die Aufgaben in sequenzieller Reihenfolge im Testprozess angegeben sind, können sie sich überschneiden und teilweise auch gleichzeitig durchgeführt werden.“ [SL07, S.19]
Auf Grundlage des fundamentalen Testprozesses nach Splinner und Linz [SL07, S.20ff] werden im Folgenden diese Teilaufgaben näher beschrieben. Diese Beschreibung wird durch Ausführungen von Seidl et al. [SBB12, S. 9 ff.], vor allem bezüglich der Testautomatisierung, erweitert.

2.4.1. Testplanung und Steuerung

Um dem Umfang und der Komplexität heutiger Software-Tests gerecht zu werden, benötigt man zu Beginn des Testprozesses eine genaue Planung. Ziel dieser Planung ist es, den Rahmen für die weiteren Testaktivitäten festzulegen. Die Aufgaben und die Zielsetzungen der

Tests müssen ermittelt wurden. Eine Ressourcenplanung wird benötigt und eine geeignete Teststrategie muss ermittelt werden. In Kapitel 2.2 wurde bereits erwähnt, dass das vollständige Testen einer Anwendung in der Regel nicht möglich ist. Die einzelnen Systemteile müssen daher nach Schwere der zu erwartenden Fehlerwirkung priorisiert werden. Um so schwerwiegender die zu erwartende Fehlerwirkung ist, um so intensiver muss der betrachtete Systemteil auch getestet werden. Ziel der Teststrategie ist also „die optimale Verteilung der Tests auf die »richtigen« Stellen des Softwaresystems." [SL07, S.21]
Steht das Softwareprojekt unter einem hohen Zeitdruck, müssen Testfälle zusätzlich priorisiert werden. Um zu verhindern, dass das Testen zu einem endlosen Prozess wird, werden geeignete Testendekriterien festgelegt. Anhand dieser Kriterien kann später entschieden werden, ob der Testprozess abgeschlossen werden kann.
Bereits zu Beginn des Testprozesses werden auch wichtige Grundsteine für eine spätere Testautomatisierung gelegt. Es muss entschieden werden, in welchen Teststufen und Testbereichen eine Automatisierung eingesetzt werden soll. Vor allem ist die Frage zu klären, ob und in welchem Ausmaß eine Automatisierung überhaupt sinnvoll ist. Es kann durchaus vorkommen, dass eine Analyse ergibt, dass eine Testautomatisierung für ein Projekt unwirtschaftlich ist. Entscheidet man sich für eine Testautomatisierung, hat das in der Regel große Auswirkungen auf die einzusetzenden Ressourcen und die zeitliche Planung und Aufwandsschätzung. Oftmals wird im Rahmen der Tests eine besondere Werkzeugunterstützung oder Infrastruktur benötigt. Derartige Punkte müssen auch bereits in der frühen Planungsphase berücksichtigt werden.
Die gesamten erarbeiteten Rahmenbedingungen werden in einem Testkonzept dokumentiert. Eine mögliche Vorlage für dieses Dokument bietet die internationale Norm IEEE 829-2008 [IEE08]. Neben der frühzeitigen Planung der Tests muss während des gesamten Testprozesses eine Steuerung erfolgen. Hierfür werden die Ergebnisse und Fortschritte der Tests und des Projekts laufend erhoben, geprüft und bewertet. Werden Probleme erkannt, kann so rechtzeitig gegengesteuert werden.

2.4.2. Testanalyse und Testdesign

In dieser Phase wird zunächst die Qualität der Testbasis überprüft. Alle Dokumente, die für die Erstellung der Testfälle benötigt werden, müssen in ausreichendem Detailgrad vorhanden sein. Mit Hilfe der qualitätsgesicherten Dokumente kann die eigentliche Testfallerstellung beginnen. Anhand der Informationen aus dem Testkonzept und den Spezifikationen, werden mittels strukturierter Testfallerstellungsmethoden logische Testfälle erstellt. Diese logischen

Testfälle können dann in einer späteren Phase konkretisiert werden, indem ihnen z.B. tatsächliche Eingabewerte zugeordnet werden. Für jeden Testfall müssen die möglichen Rand- und Vorbedingungen sowie ein erwartetes Ergebnis bestimmt werden.
In dieser Phase beginnen auch erste Aufgaben, die mit der Testautomatisierung in Zusammenhang stehen. Abgestimmt auf die ausgewählten Automatisierungswerkzeuge und die zu testende Software, muss die Umgebung für die Testautomatisierung vorbereitet werden. Anhand der Vorgaben des Testkonzeptes können dann jene Testfälle und Testabläufe ausgewählt werden, die im Zuge der Testautomatisierung implementiert werden sollen. Hierbei wird noch einmal die technische Umsetzbarkeit geprüft. Bei der Auswahl der Testfälle sollte zu Beginn eine möglichst breite Testabdeckung angestrebt werden. Problemfelder können dann später durch weitere Testfälle in der Tiefe getestet werden.

2.4.3. Testrealisierung und Testdurchführung

In diesem Schritt des Testprozesses werden aus den logischen Testfällen der vorangegangenen Phase konkrete Testfälle gebildet. Diese werden anhand ihrer fachlichen und technischen Zusammengehörigkeit zu Testszenarien gruppiert und anhand der Vorgaben aus dem Testkonzept priorisiert. Sobald die zu testende Software zur Verfügung steht, kann mit der Abarbeitung begonnen werden. Die dabei erhaltenen Ergebnisse werden vollständig protokolliert. Werden im Zuge der Durchführung Fehler aufgedeckt, muss darauf in geeigneter Weise reagiert werden. Es kann beispielsweise ein zuvor definierter Fehlerprozess gestartet werden. Korrekturen und nachgehende Veränderungen am Testobjekt werden durch eine Wiederholung der Testläufe abgedeckt.
Aus Sicht der Testautomatisierung beginnt in dieser Phase die technische Umsetzung. In vielen Fällen bedeutet das Programmiertätigkeit. Diese Programmiertätigkeiten sind wiederum anfällig für eigene Fehler und müssen daher in angemessener Weise selbst qualitätsgesichert werden. Auch bei der Testautomatisierung ist eine Zusammenfassung von Testfällen sinnvoll. Auf diese Weise kann man funktionalen und logischen Abhängigkeiten zwischen den Testfällen gerecht werden. Nach der Implementierung können die geplanten Testfälle durchgeführt werden. Gerade bei der Automatisierung ist eine genaue Protokollierung der Ergebnisse besonders wichtig. Nur dadurch ist es später möglich, aufgetretene Fehler überhaupt zu lokalisieren.

2.4.4. Testauswertung und Bericht

In dieser Phase des Prozesses wird geprüft, ob die im Testkonzept definierten Testendekriterien erreicht wurden. Sind alle Forderungen erfüllt, kann es zu einem Abschluss der Testaktivitäten kommen. Kommt es zu Abweichungen im Bezug auf diese Kriterien, muss darauf entsprechend reagiert werden. Es können Fehlerkorrekturen durchgeführt werden oder neue Testfälle erstellt werden. Aber auch der umgekehrte Fall ist möglich. Es kann dazu kommen, dass Endekriterien nur mit unverhältnismäßig hohem Aufwand erreicht werden und daher bestimmte Testfälle entfallen oder Kriterien überdacht werden müssen.
Für die Testautomatisierung ist die wesentliche Aufgabe dieser Phase die Auswertung und Aufarbeitung der erhaltenen Ergebnisse. Automatisierte Tests generieren oftmals eine Fülle an Log-Dateien und Protokollen. Um aus diesen Ergebnissen die richtigen Schlüsse zu ziehen und sie für Dritte zugänglich zu machen, müssen sie in eine lesbare Form gebracht werden.
In jedem Fall muss über die erhaltenen Ergebnisse und das daraus resultierende Vorgehen ein Testbericht erstellt werden. Je nach Umfang und Phase der Tests kann dieser mehr oder weniger formal ausfallen. Für einen Komponententest reicht beispielsweise eine formlose Mitteilung. Höhere Teststufen erfordern einen formaleren Bericht.

2.4.5. Abschluss der Testaktivitäten

Sind die Testaktivitäten beendet, sollten zum Schluss alle im Laufe des Testprozesses gemachten Erfahrungen analysiert werden. So können die gewonnenen Erkenntnisse für spätere Projekte genutzt werden. Dadurch kann eine stetige Verbesserung des Testprozesses erreicht werden. Die während des Prozesses erstellten Testfälle, sowie alle sonstigen Ergebnisse, sollten archiviert werden. Auf diese Weise stehen sie für folgende Regressionstests zur Verfügung. Die Kosten für Wartung und Pflege der Software können damit gesenkt werden. Bei der Testautomatisierung bedeutet das, die Wiederherstellbarkeit der Testumgebung und des Sourcecodes sicherzustellen.

Abschließend ist zu sagen, dass sich die Testautomatisierung in der Regel gut in einen bereits bestehenden Testprozess integrieren lässt. Sie wird allerdings „den Prozess nicht verbessern oder gerade richten, sondern nur unterstützen.“ [SBB12, S.21]
Ist der Testprozess schon vor Einführung einer Automatisierung schlecht organisiert gewesen, wird er sich nach der Einführung nicht verbessern. Die Testautomatisierung ist also nicht als Heilmittel für schlecht laufende Prozesse gedacht, sondern als Möglichkeit einen bereits gut etablierten Prozess effizienter und effektiver zu gestalten.

2.5. Vorgehensmodelle

Der in Kapitel 2.4 beschriebene Testprozess ist nicht als losgelöster, eigenständiger Prozess zu betrachten. Vielmehr ist der Testprozess immer ein Teil eines größeren Entwicklungsablaufes bei der Erstellung eines Softwareproduktes. Einen solchen Entwicklungsablauf versucht man mit Hilfe von sogenannten Softwareentwicklungsmodellen, auch Vorgehensmodelle genannt, abzubilden. Ein Projekt wird dazu in einzelne Phasen untergliedert, an deren Ende ein gewisses Ziel bzw. Ergebnis steht. Auf gröbster Ebene lassen sich die Abläufe auf vier Hauptphasen reduzieren. Diese Phasen finden sich mehr oder weniger ausgeprägt in den meisten der gängigen Vorgehensmodelle wieder und werden auch so von Seidl et al. [SBB12, S.21 ff.] verwendet:

- Spezifikation
- Design
- Entwicklung
- Test

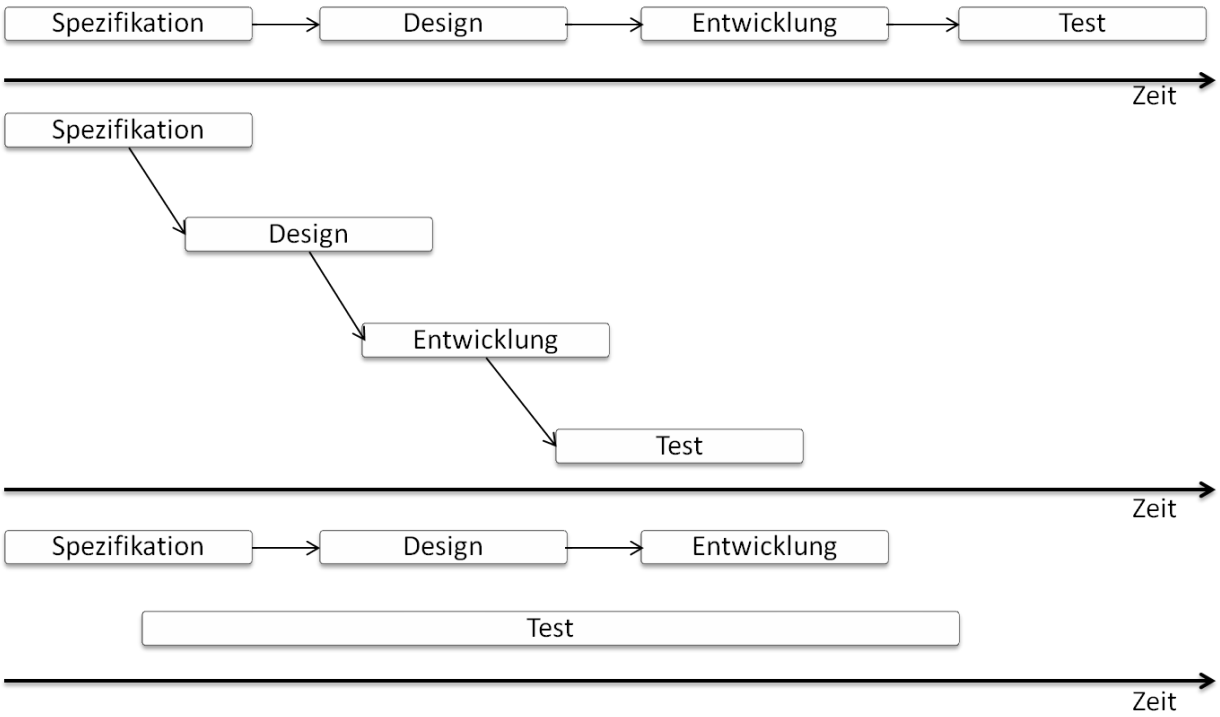
Das Testen, bzw. der Testprozess ist eine von mehreren Phasen in einem solchen Entwicklungsmodell. Es gibt eine Vielzahl von unterschiedlichen Softwareentwicklungsmodellen. Der Hauptunterschied liegt meist in der zeitlichen Koppelung und der inhaltlichen Ausprägung der einzelnen Phasen. Die einzelnen Phasen können sich innerhalb eines Vorgehensmodells überschneiden und wiederholen und müssen auch nicht immer, wie in der Auflistung angegeben, sequenziell abgearbeitet werden. Aus der Sicht der Testautomatisierung ist nach Seidl et al. [SBB12, vgl. S.21 ff.] eine Einteilung der verschiedenen Vorgehensmodelle in zwei Gruppen sinnvoll:

- Klassische Entwicklungsmodelle, die eher sequentiell ausgerichtet sind
- Iterative und agile Entwicklungsmodelle, die sich durch Parallelisierung und kurze Iterationen auszeichnen.

2.5.1. Klassische Entwicklungsmodelle

Die hier als klassische Entwicklungsmodelle betitelten Vorgehensmodelle zeichnen sich vor allem dadurch aus, dass die einzelnen Phasen sequenziell ausgeführt werden. Der bekannteste Vertreter dieser Vorgehensmodelle ist das Wasserfallmodell [Roy87]. In diesem Modell sind

alle Phasen strikt voneinander getrennt. Eine neue Phase kann erst begonnen werden, wenn eine vorangegangene Phase abgeschlossen wurde. Rücksprünge in vorangegangenen Phasen sind unerwünscht. In der Praxis wird dieses Vorgehen, laut Seidl et al. [SBB12, vgl. S.22], jedoch oft nicht ganz so strikt umgesetzt. Es kommt zu Mischformen, bei denen die einzelnen Phasen nicht mehr voll sequenziell abgearbeitet werden, sondern sich teilweise überlagern. Vor allem im Bereich des Testens geht man oft zu einer solchen Mischform über. Das Testen ist meist keine getrennte Phase am Ende des Entwicklungsprozesses, sondern erstreckt sich über den gesamten Prozess ausgehend von der frühen Spezifikationsphase. Mögliche Ausprägungen klassischer Entwicklungsmodelle sind in Abbildung 2.3 dargestellt.



Quelle: [SBB12, vgl. S.22]

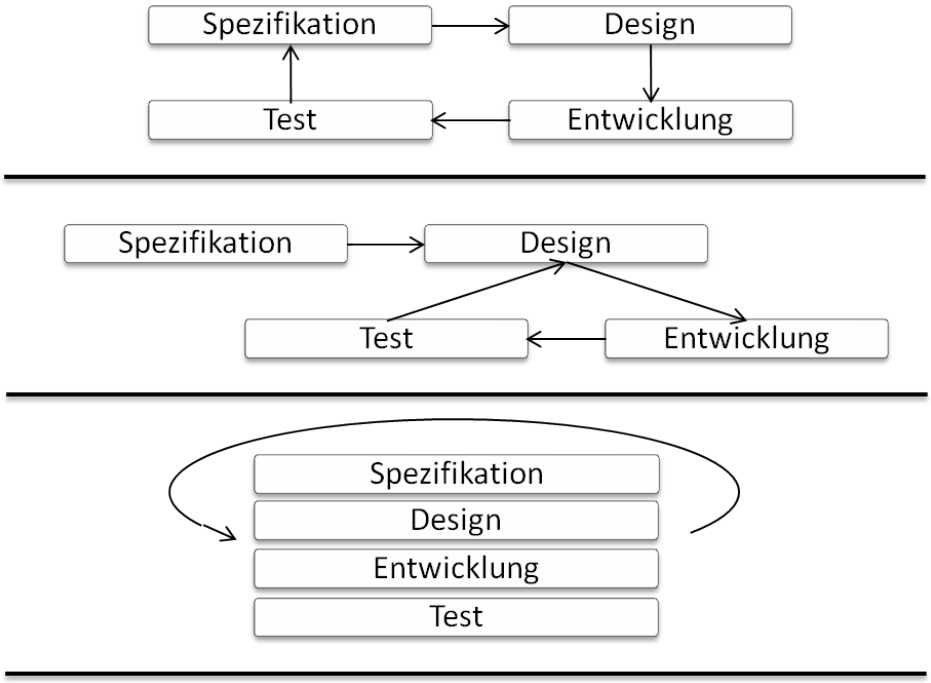
Abbildung 2.3.: Verschiedene Ausprägungen klassischer Entwicklungsmodelle

Seidl et al. [SBB12, vgl. S.22] stellen fest, dass in Projekten, in denen ein sequenzielles Vorgehen gewählt wird, bereits in der frühen Planungsphase des Testprozesses genau abzuwägen ist, ob eine Automatisierung der Testfälle überhaupt sinnvoll ist. Wenn zu Beginn des Projektes schon klar ist, dass die Testfälle nur ein einziges Mal, am Ende des Entwicklungsprozesses, ausgeführt werden, steht eine Automatisierung oft nicht in Relation zu den erhöhten Kosten, die bei der Erstellung der Tests anfallen würden. Es ist allerdings zu beachten, dass Software meist mit dem Ende eines Projektes nicht seinen finalen Stand erreicht hat. Feh-

ler sowie geänderte Anforderungen führen meist dazu, dass sich Softwareprodukte ständig weiterentwickeln. Diese Weiterentwicklung ist zwangsläufig mit Codeänderungen verbunden, die wiederum zu Fehlern in bereits bestehendem Code führen können. Um solche Fehler zu entdecken müssen im Rahmen von Regressionstests auch Testfälle wiederholt werden, die bereits erfolgreich abgeschlossen wurden. Solche Regressionstests lassen sich bei einer vorhandenen Testautomatisierung besonders leicht durchführen. Sind also in der Software nach Projektabschluss größere Änderungen zu erwarten, kann sich eine Automatisierung über längere Sicht durchaus lohnen. Neben Regressionstests kann nach Seidl et al. [SBB12, vgl. S.23] auch die Notwendigkeit einer höheren Testtiefe oder einer breiteren Testabdeckung ein Faktor sein, sich für eine Automatisierung zu entscheiden. In manchen Fällen wie beispielsweise Lasttests, mit mehreren hundert Usern, kann eine Automatisierung auch unabdingbar werden.

2.5.2. Iterative und agile Entwicklungsmodelle

Als weitere Gruppe der Vorgehensmodelle nennen Seidl et al. [SBB12, vgl. S.23 ff.] die iterativen und agilen Entwicklungsmodelle.
Im Gegensatz zu den klassischen Entwicklungsmodellen sind in iterativen Modellen Rücksprünge in vorangegangene Phasen explizit erlaubt. Eine oder alle Phasen werden in diesen Modellen wiederholt durchlaufen. Auf diese Weise kann das Softwareprodukt inkrementell wachsen. Durch ein derartiges Vorgehen ist es einfacher möglich, auf den Umstand zu reagieren, dass sich Anforderungen in Softwareprojekten häufig ändern.
Auch agile Vorgehensmodelle leben von solch einem iterativen Vorgehen. Die einzelnen Phasen werden in kleinen Zyklen viele Male durchlaufen. Ein bekannter Vertreter der agilen Methoden ist Scrum [Sch02]. In Scrum wird ein Softwareprodukt in kurzen, sogenannten Sprints realisiert. Innerhalb eines Sprints wählt das Team selbständig eine Teilaufgabe des Projekts aus. Diese Teilaufgabe wird spezifiziert, entworfen, entwickelt und getestet. Am Ende eines Sprints steht ein Softwareprodukt, welches um einen weiteren Baustein ergänzt wurde. Der Sprint ist das zentrale Element dieses Prozessmodells und kennzeichnet eine Iteration. Abbildung 2.4 zeigt verschiedene Ausprägungen iterativer und agiler Entwicklungsmodelle.
Für das Testen stellen diese kurzen Iterationen laut Seidl et al. [SBB12, vgl. S.24] ein Problem dar. Jeder Entwicklungszyklus bringt neue Features hervor, die mit Testfällen abgedeckt werden müssen. Der agile Charakter in diesen Vorgehensmodellen bedingt, dass sich Anforderungen ständig ändern und somit auch bereits fertiger Code oft angepasst werden muss. Darüber hinaus ist nicht ausgeschlossen, dass neue Features Auswirkungen auf alten



Quelle: [SBB12, vgl. S.24]

Abbildung 2.4.: Verschiedene Ausprägungen iterativer und agiler Entwicklungsmodelle

Code haben können. Neben den neu implementierten Teilen muss daher zum Ende einer jeden Iteration auch sämtlicher alter Code getestet werden. Dies bedingt einen enormen zusätzlichen Testaufwand. In agilen Vorgehensmodellen, wie Scrum, ist der Testaufwand nach wenigen Sprints bereits oft so hoch, dass ein Testdurchlauf zusammen mit allen Regressionstests manuell nicht mehr zu bewältigen ist. Gerade in Projekten, die einem derartigen Vorgehensmodell folgen, ist es daher sinnvoll Testautomatisierung einzusetzen. Einmal implementierte Testfälle können zum Ende einer jeden Iteration erneut ausgeführt werden. Die höheren Kosten, die bei der Automatisierung entstehen, sind so schnell amortisiert.
Das sich ständig ändernde Testobjekt bedingt nicht nur die Notwendigkeit von automatisierten Testfällen, sondern erhöht gleichzeitig auch die Anforderungen an die Qualität. Häufige Änderungen am zu testenden Code lassen einmal implementierte Testfälle schnell veralten. Es muss daher bei der Erstellung der automatisierten Tests besonders auf die Wartbarkeit geachtet werden. Testfälle sollten daher möglichst robust in ihrem Design sein, um nicht schon bei kleinen Änderungen zerstört zu werden. Der Qualitätsstandard sollte den gleichen Anforderungen unterliegen, der bereits für den vorhandenen Code verwendet wurde. Anpassungen werden sonst zu zeitaufwendig. Die Pflege der bereits implementierten Testfälle ist dann nicht mehr wirtschaftlich und die Akzeptanz der Tests im Projekt sinkt.

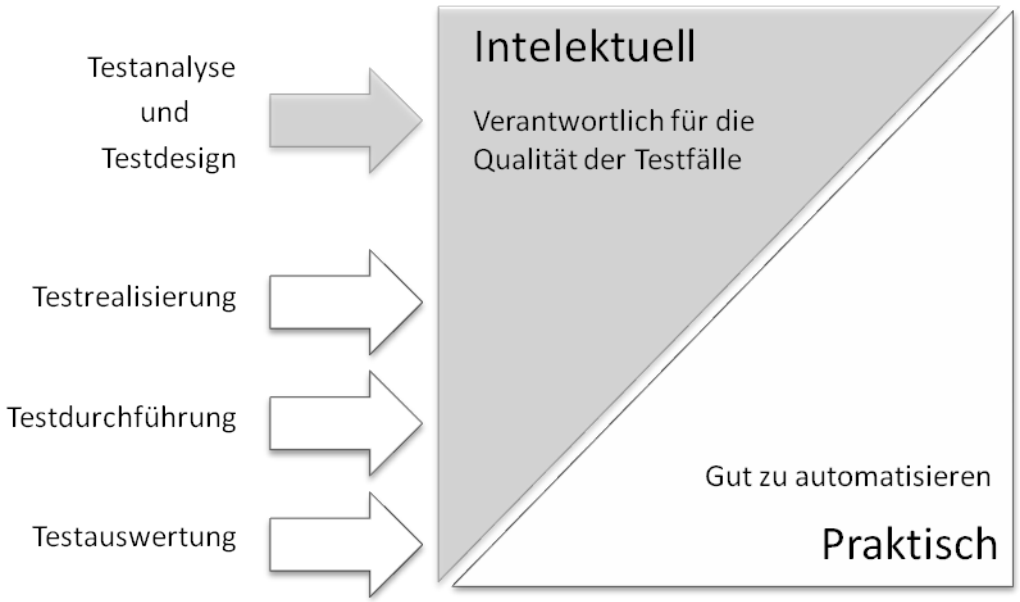
3. Testautomatisierung

In Kapitel 2.3 wurde der Begriff Testautomatisierung bereits eingeführt. Die darin dargelegte Definition hat gezeigt, dass man unter Testautomatisierung nicht nur das automatisierte Ausführen von Testfällen versteht. Testautomatisierung ist in allen Bereichen des Entwicklungs- bzw. Testprozesses möglich. „Das Spektrum umfasst alle Tätigkeiten zur Überprüfung der Softwarequalität im Entwicklungsprozess, in den unterschiedlichen Entwicklungsphasen und Teststufen sowie die entsprechenden Aktivitäten von Entwicklern, Testern, Analytikern oder auch der in die Entwicklung eingebundenen Anwender. Die Grenzen der Automatisierung liegen darin, dass diese nur die manuellen Tätigkeiten eines Testers übernehmen kann, nicht aber die intellektuelle, krative und intuitive Dimension dieser Rolle.“ [SBB12, S.7]
Die intellektuelle Dimension ist vor allem in den frühen Phasen des Testprozesses gefordert. Diese Phasen sind maßgeblich für die spätere Qualität der einzelnen Testfälle. Testautomatisierung wird daher nie die Arbeit eines Testanalysten voll ersetzen können.
Um so weiter der Testprozess voranschreitet, um so praktischer werden auch die zu erledigenden Aufgaben. Das Potential für eine Automatisierung steigt also im Laufe des Testprozesses. Fewster und Graham [FG99, vgl. S.18] stellen diesen Zusammenhang in einer Grafik bildlich dar. Abbildung 3.1 greift diese Darstellung auf und passt sie auf den in Kapitel 2.4 vorgestellten Testprozess an. Die verschiedenen Möglichkeiten der Testautomatisierung werden in Kapitel 3.2 geklärt. Zunächst soll jedoch die Frage beantwortet werden, weshalb eine Automatisierung von Testfällen überhaupt sinnvoll ist.

3.1. Warum Testautomatisierung

Richtig durchgeführt kann Testautomatisierung eine Reihe von Vorteilen bringen. Dustin et al. [Dus01, S.44 ff.] stellen drei Hauptvorteile der Testautomatisierung fest:

1. Erstellung eines zuverlässigen Systems
2. Verbesserung der Testqualität und Testtiefe
3. Verringerung des Testaufwands und Reduzierung des Zeitplans



Quelle: [FG99, vgl. S.18]

Abbildung 3.1.: Grenzen und Möglichkeiten der Testautomatisierung

In der Literatur gibt es zahlreiche Listen von Vorteilen der Testautomatisierung, die sehr viel feiner gegliedert sind, als die von Dustin et al. [Dus01, S.44 ff.] gewählten Oberpunkte. So nennen beispielsweise Fewster und Graham [FG99, vgl. S.9 ff.] oder auch Thaller [Tha02, vgl. S.28 ff.] eine Reihe von positiven Aspekten.
Gleicht man diese Vorteile mit den von Dustin et al. gewählten Oberpunkten ab, zeigt sich, dass vor allem die Verbesserung der Testqualität und Testtiefe (2), sowie die Verringerung des Testaufwands und die Reduzierung des Zeitplans (3), gut durch diese repräsentiert werden. Diese Oberpunkte lassen sich leicht mit den feiner ausformulierten Vorteilen unterfüttern.
Die Erstellung eines zuverlässigen Systems (1) ist hingegen nur schwer direkt durch feiner formulierte Vorteile zu untermauern. In der Regel wird dieser Punkt indirekt durch eine Verbesserung in den letzten beiden Bereichen (2,3) beeinflusst:
Eine Verringerung des Testaufwands für einzelne Tests schafft mehr Kapazität, die in bessere und breiter angelegte Tests investiert werden kann. Zusätzlich wird die Testqualität und Testtiefe direkt verbessert. Dies bedingt wiederum, dass mehr Fehler im System aufgedeckt werden können. Dadurch kann eine höhere Qualität des Endproduktes erreicht werden, die sich in einem zuverlässigeren System zeigt.
Da die Erstellung eines zuverlässigen Systems (1) nur schwer direkt durch feiner gegliederte Vorteile belegt werden kann und eher eine Folge der Verbesserung in den anderen beiden Bereichen (2,3) ist, wird sie im weiteren nicht näher betrachtet.

Fewster und Graham [FG99, vgl. S.10] fassen die Vorteile der Testautomatisierung noch weiter zusammen und reduzieren sich in ihrem Fazit auf die Worte 'Qualitäts- und Effizienzsteigerung'. Diese Begriffe entsprechen weitestgehend den von Dustin et al. gewählten Oberpunkten. Qualitätssteigerung fasst dabei die Verbesserung der Testqualität und Testtiefe zusammen. Die Verringerung des Testaufwands und Reduzierung des Zeitplans entspricht der Effizienzsteigerung.
Um die Vorzüge der Testautomatisierung auf eine feinere und damit greifbarere Ebene zu bringen, werden im Folgenden die Vorteile, wie sie Fewster und Graham beschreiben, verwendet und den von Dustin et al. gewählten Oberpunkten zugeordnet.

3.1.1. Verringerung des Testaufwands und Reduzierung des Zeitplans

Die Vorteile in folgender Aufzählung beschreiben nach Fewster und Graham [FG99, vgl. S. 9 ff.], dass der Aufwand, der für das Testen einer Software betrieben werden muss, mit Hilfe von Automatisierung reduziert werden kann. Reduzierter Aufwand in den Tests, sowie eine schnellere und wiederholbare Abarbeitung der Testfälle, führen dann meist dazu, dass der gesamte Zeitplan des Projekts positiv beeinflusst wird. Sein volles Potential entfaltet Automatisierung immer dann, wenn Testfälle wiederholt ausgeführt werden. Regressionstests, die vor jedem neuen Releasezyklus einer Software durchgeführt werden, sind daher prädestiniert dazu, automatisiert zu werden. Tester können so von sich wiederholenden Testaufgaben entlastet werden. Das reduziert den Testaufwand und stellt Tester für andere Aufgaben frei, was wiederum das gesamte Projekt beschleunigt.

- *Ausführen existierender Regressionstests für eine neue Version der Software*
  Der Aufwand, um Regressionstests manuell durchzuführen, kann schnell sehr groß werden. Sind Testfälle automatisiert, ist es möglich, sie bei Änderungen am System mit wenig Aufwand erneut durchzuführen.

- *Besserer Einsatz von Resourcen*
  Mittels Automatisierung lässt es sich vermeiden, Tester durch generischen Aufgaben, wie beispielsweise das immer gleiche erzeugen von Testeingaben, zu binden. Die frei gewordenen Resourcen können für andere Aufgaben verwendet werden. Der Zeitplan des Projektes kann so verkürzt werden.

- *Wiederverwendbarkeit von Testfällen*
  Neue Projekte können von den Ergebnissen der Testautomatisierung aus vorangegangenen Projekten profitieren. Auch innerhalb eines Projektes können Teile von automa-

tisierten Testfällen oftmals wiederverwendet werden. Eine Reduzierung des Zeitplans ist dadurch möglich.

- *Frühere Markteinführung*
  Richtig eingesetzt, beschleunigt Testautomatisierung den gesamten Testprozess. Das verkürzt letztendlich auch die Zeit bis zur Markteinführung des Softwareprodukts.

3.1.2. Verbesserung der Testqualität und Testtiefe

Auch zeigen die beschriebenen Vorteile nach Fewster und Graham [FG99, vgl. S. 9 ff.], dass sich mit Hilfe der Testautomatisierung Verbesserungen im Bereich der Testqualität und Testtiefe erreichen lassen. Eine bessere Testqualität wird meist dadurch erzielt, dass die Testfälle in ihrer Gesamtheit ein höheres Potential erreichen, Fehler aufzudecken. Vor allem eine höhere Testtiefe und eine breitere Testabdeckung sind hier die treibenden Faktoren. Auch die Qualität einzelner Testfälle kann mittels Testautomatisierung direkt verbessert werden. Eine bessere Wiederholbarkeit ist hier der maßgebende Faktor.

- *Mehr Testfälle öfter ausführen*
  Aus Zeitmangel müssen sich Tester oft auf einen geringeren Testumfang zurückziehen, als eigentlich gewünscht ist. Vor allem bei sehr generischen Testfällen, die sich beispielsweise nur in verschiedenen Maskeneingaben unterscheiden, ist es mit Hilfe von Testautomatisierung möglich, in weniger Zeit ein Vielfaches an Testfällen durchzuführen. Eine tiefere Testabdeckung ist die Folge. Da solche Testfälle in der Regel auf einem zentralen Basistestfall beruhen, ist es hier besonders einfach möglich, eine durchgehend hohe Testqualität zu gewährleisten.

- *Testfälle durchführen, die ohne Automatisierung schwer bis unmöglich wären*
  Einen Lasttest mit z.B. mehr als 200 Benutzern manuell durchzuführen, erweist sich als nahezu unmöglich. Die Eingaben von 200 Benutzern lassen sich mit Hilfe von automatisierten Lasttests jedoch gut simulieren. Über Testfälle, die ohne Automatisierung gar nicht möglich wären, lässt sich die Testbreite erhöhen. Die Qualität der Tests steigt durch das erhöhte Potential, Fehler zu entdecken.

- *Konsistenz und Wiederholbarkeit von Testfällen*
  Testfälle, die automatisch durchgeführt werden, werden immer auf die gleiche Weise ausgeführt. Eine derartige Konsistenz ist auf manuellem Wege kaum zu erreichen. Fehler können somit vermieden und die Qualität gesteigert werden.

- *Erhöhtes Vertrauen in die Testfälle und Software*
  Das Wissen, dass eine Vielzahl an Testfällen erfolgreich vor jedem Release durchgeführt wurden, erhöht das Vertrauen von Entwickler und Nutzern, dass unerwartete Fehler ausbleiben. Werden Testfälle regelmäßig ausgeführt, erhöht das zusätzlich das Vertrauen, dass diese Testfälle stabil sind und keine Falschmeldungen auftreten.

3.1.3. Kosten als Bewertungsgrundlage für die Testautomatisierung

Die unter Kapitel 3.1.2 und Kapitel 3.1.1 aufgezeigten Vorteile der Testautomatisierung haben das Problem, dass sie sich meist nur schwer messen und mit genauen Zahlen belegen lassen. Um das zu erreichen, muss man sich auf die kleinste gemeinsame Größe berufen, auf die all die genannten Vorteile hinarbeiten: Eine Kostenreduzierung beim Testen. Sowohl eine Verbesserung der Testqualität und Testtiefe, als auch eine Verringerung des Testaufwands und Reduzierung des Zeitplans, verfolgen in letzter Instanz immer das Ziel, Kosten einzusparen.
Um den Nutzen der Testautomatisierung für ein Projekt messbar zu machen und nachvollziehbar zu begründen, bieten nach Ramler und Wolfmaier [RW06] daher die direkten Kosten, die durch das Erstellen und Ausführen der Testfälle entstehen, den besten Ansatzpunkt. Ramler und Wolfmaier [RW06] zitieren eine Fallstudie von Linz und Daigl [DRP99], die eine Unterteilung der Kosten in zwei Komponenten vornimmt.

$V :=$ Ausgaben für Testspezifikation und Implementierung

$D :=$ Ausgaben für einen einzelnen Testlauf

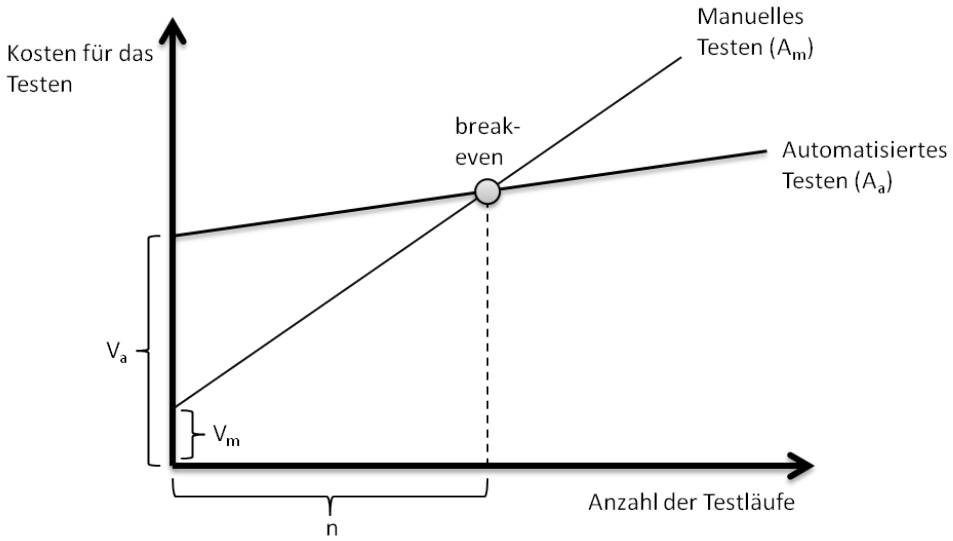
Mit Hilfe dieser beiden Variablen können die Kosten (A_a) für einen einzelnen automatisierten Testfall wie folgt angegeben werden:

$$A_a := V_a + n * D_a \tag{3.1}$$

V_a symbolisiert die Kosten, die für die Spezifikation und Implementierung des automatisierten Testfalls anfallen. D_a die Kosten, die für das einmalige Ausführen des Testfalles entstehen und n steht für die Anzahl der durchgeführten Testläufe. Um zu bestimmen, wie sich das manuelle und automatisierte Testen zueinander verhalten, kann analog die selbe Gleichung für das manuelle Testen aufgestellt werden.

$$A_m := V_m + n * D_m \tag{3.2}$$

Mit Hilfe dieser beiden Gleichungen lässt sich zeigen, dass sich ab einer gewissen Anzahl an Testläufen, die Automatisierung gegenüber der manuellen Ausführung, bezüglich der Kosten lohnt. Es wird dabei davon ausgegangen, dass die initiale Investition V_a für die Automatisierung höher ist, als die initiale Investition für das manuelle Testen V_m. Die Kosten A_a des automatisierten Testfalls steigen mit jeder Testausführung n jedoch langsamer an. Beide Funktionen schneiden sich daher in einem Break-even-Point, ab dem die Automatisierung die günstigere Alternative darstellt. Abbildung 3.2 veranschaulicht diesen Zusammenhang grafisch.



Quelle: [RW06]

Abbildung 3.2.: Break-even-Point für Testautomatisierung

Zusammenfassend lässt sich feststellen, dass vor allem wiederholt ausgeführte Testtätigkeiten hohes Einsparungspotential bei einer Testautomatisierung bieten.

3.1.4. Probleme der Testautomatisierung

Ramler und Wolfmaier [RW06] zitieren ein Erreichen des Break-even-Point, wie er in Kapitel 3.1.3 beschrieben ist, nach einer Ausführung von 2-20 Testläufen. Diese große Spanne macht deutlich, wie wichtig es ist, in der Testplanung und Steuerung genau abzuwägen, ob und wo eine Automatisierung sinnvoll einzusetzen ist. Eine Automatisierung kann oft auch unwirtschaftlich sein, vor allem immer dann, wenn Tests nur ein einziges mal ausgeführt werden. Implementierungs- und Wartungsaufwand von automatisierten Testfällen sind meist sehr viel höher, als die von manuellen Testfällen. Dieser Mehraufwand muss in irgend einer Wei-

se gerechtfertigt sein. Laut Fewster und Graham [FG99, vgl. S. 22 ff.] und auch Thaller [Tha02, vgl. S.230 ff.] wird dieser Punkt oftmals vernachlässigt und die Testautomatisierung als Heilmittel für schlecht laufende Prozesse und zu hohe Kosten gesehen. Dabei ist genau das Gegenteil der Fall. Es ist sinnvoller, zunächst die Qualität der Tests und des eigenen Testprozesses zu optimieren, bevor eine Automatisierung eingeführt wird.
Ein weiterer Schwachpunkt der Automatisierung ist nach Fewster und Graham [FG99, vgl. S. 22 ff.], dass die Automatisierung von Software-Tests zwar einen Mehraufwand bedeutet, jedoch die Anzahl der gefundenen Fehler nur geringfügig erhöht wird. Bevor Testfälle automatisiert werden, müssen sie in der Regel zuvor einmal manuell durchgeführt werden, um sicher zu stellen, dass der angedachte Test auch sinnvoll und realisierbar ist. Meist werden Fehler bereits bei dieser manuellen Überprüfung festgestellt. Wird der Testfall dann automatisiert, deckt er weit weniger wahrscheinlich einen Fehler auf, als bei seiner ersten, manuellen Ausführung. Darüber hinaus können automatisierte Testfälle Fehler nur über die Akzeptanzkriterien aufdecken, die ihnen explizit hinterlegt wurden. Oft reichen die hinterlegten Kriterien aber nicht aus, um alle Fehlerquellen abzudecken. Das kann laut Fewster und Graham [FG99, vgl. S. 23 ff.] dazu führen, dass Testfälle als positiv gekennzeichnet werden, obwohl sie in Wirklichkeit fehlerhaft waren. Bei manuellen Tests fallen vergessene Akzeptanzkriterien eher auf bzw. werden durch den Tester intuitiv ergänzt. Die Anzahl der fälschlicherweise positiv gewerteten Testfälle sinkt also mit der manuellen Durchführung.
Einen weiteren positiven Aspekt, den Fewster und Graham [FG99, vgl. S. 24 ff.] in einem manuellen Tester sehen, ist eine höhere Stabilität in den Testfällen. Ein Tester kann sich auf kleinere Änderungen in der zu testenden Software leicht einstellen. Er kann selbst entscheiden, ob eine Abweichung vom Testfall als Fehler zu werten oder zu vernachlässigen ist. Automatisierte Testfälle bieten diesen Luxus nicht. Sie verfolgen einen festen Ablaufplan und können nur schwer mit Veränderungen in der zu testenden Software umgehen. Das macht sie, im Vergleich zu manuellen Tests, instabil und erfordert einen erhöhten Wartungsaufwand. In manchen Fällen kann diese Abhängigkeit zwischen Software und automatisiertem Test sogar so weit führen, dass sie die Entwicklung der Software behindern. Beispielsweise dann, wenn eine Änderung so große Auswirkungen auf die automatisierten Testfälle hätte, dass es aus wirtschaftlicher Sicht nicht mehr sinnvoll ist sie durchzuführen. Die Kosten für die Anpassungen wären dann so hoch, dass sie den Nutzen der Softwareänderung übersteigt.
Analog zu den in Kapitel 3.1.1 und 3.1.2 genannten Vorteilen haben Fewster und Graham [FG99, vgl. S. 10 ff.] auch eine Liste mit bekannten Nachteilen der Testautomatisierung aufgestellt, welche die gerade angesprochenen Probleme noch einmal aufgreifen und ergänzen:

- *Unrealistische Erwartungen*

Testautomatisierung wird oft als Lösung für alle Testprobleme gesehen. Es ist wichtig, dass die Erwartungen aller Beteiligten realistisch bleiben.

- *Schlechte Testpraxis*
  Wenn das Testen in einem Unternehmen bereits eine Schwachstelle darstellt, ist es nicht sinnvoll eine Testautomatisierung einzuführen. Es ist besser, zunächst die vorherrschenden Prozesse zu optimieren.

- *Die Anzahl der gefundenen Fehler wird sich nicht stark verändern*
  Fehler werden meist bei der ersten Durchführung eines Testfalls aufgedeckt. Wird ein Testfall wiederholt, sinkt auch sein Potential Fehler aufzudecken. In der Regel werden Fehler bereits beim Entwickeln der automatisierten Testfälle entdeckt, nicht erst bei weiteren Testläufen.

- *Trügerische Sicherheit*
  Die Tatsache, dass alle automatisierten Testfälle positiv waren bedeutet nicht, dass die Software auch frei von Fehlern ist. Möglicherweise sind nicht alle Bereiche der Software mit Testfällen abgedeckt oder die Akzeptanzkriterien der Testfälle sind nicht umfassend genug gewählt worden. Auch ist es möglich, dass die Testfälle fehlerhaft sind und falsche Ergebnisse anzeigen.

- *Wartung*
  Automatisierte Testfälle haben einen hohen Wartungsaufwand. Änderungen an der Software bedingen oft auch, dass die Testfälle überarbeitet werden müssen. Wird dieser Wartungsaufwand so hoch, dass es günstiger wäre die Testfälle manuell durchzuführen, werden die automatisierten Tests unwirtschaftlich.

- *Technische Probleme*
  Die Automatisierung von Testfällen stellt eine komplexe Aufgabe dar und ist daher oft mit Problemen verbunden. Die verwendeten Tools bzw. Frameworks sind meist selbst nicht befreit von Fehlern. Oft gibt es auch technische Probleme mit der zu testenden Software selbst.

- *Probleme in der Organisation*
  Eine erfolgreiche Testautomatisierung stellt hohe Anforderungen an die technischen Fähigkeiten der Entwickler und erfordert starken Rückhalt in der Führungsebene. Testautomatisierung läuft nicht immer sofort reibungslos und benötigt oft Anpassungen in vorherrschenden Prozessen.

3.2. Möglichkeiten der Testautomatisierung im Testprozess

Wie bereits zu Beginn dieses Kapitels erwähnt, beschränkt sich die Testautomatisierung nicht nur auf das automatisierte Ausführen von Testfällen, sondern erstreckt sich über den gesamten Testprozess. Innerhalb des Testprozesses haben Amannejad et al. [Ama+14] vier Hauptaufgaben identifiziert, die aus Sicht der Testautomatisierung besonders interessant sind:

- Testdesign: Erstellen einer Liste von Testfällen.
- Testcodeerstellung: Erstellen von automatisiertem Testcode.
- Testdurchführung: Ausführen von Testfällen und Aufzeichnen der Ergebnisse.
- Testauswertung: Auswerten der Testergebnisse.

3.2.1. Testdesign

Beim Testdesign handelt es sich um eine Aufgabe die laut Thaller [Tha02, vgl. S. 231] nicht unbedingt prädestiniert dafür ist, automatisiert zu werden. Abbildung 3.1 hat bereits gezeigt, dass es sich um eine eher intellektuell geprägte Aufgabe handelt. Dennoch gibt es eine Reihe von Möglichkeiten, wie das Entwerfen von Testfällen automatisiert werden kann. Die verschiedenen Ansätze beschränken sich in der Regel darauf, unterschiedliche Testeingaben für die zu testende Software zu finden. Ein großes Problem, welches die Tools zum Entwerfen der Testfälle laut Fewster und Graham [FG99, vgl. S. 19] dabei haben ist, dass sie einem fest vorgegebenem Algorithmus folgen. Dieses Vorgehen wird jedoch der intellektuellen Komponente dieser Aufgabe nicht gerecht. Ein Tester kann ähnlich strukturierte Testfallerstellungsmethoden heranziehen, wie sie beim automatisierten Testdesign verwendet werden. Um eine komplexe Software umfassend zu testen, reicht es jedoch meist nicht aus, einem festen Algorithmus nachzugehen. Eine tiefere Analyse durch einen Tester wird benötigt. Dieser ist in der Lage, außerhalb eines vorgegebenen Algorithmus, Testfälle zu identifizieren, fehlende Anforderungen zu finden oder sogar, aufgrund von persönlicher Erfahrung, Fehler in der Spezifikation aufzudecken.
Ein weiteres Problem das Fewster und Graham [FG99, vgl. S. 19] nennen, ist die Menge an Testfällen, die mit Hilfe von automatisierten Methoden erzeugt werden. Die Anzahl der Testfälle kann schnell so groß werden, dass sie nicht mehr in einem vertretbarem Zeitaufwand durchgeführt werden können. In diesem Fall müssen aus der Menge an Testfällen, die

wichtigsten identifiziert werden. Testfälle nach ihrer Relevanz zu filtern ist wiederum eine intellektuelle Aufgabe, die nur schwer in einem Automatisierungstool abgebildet werden kann. Trotz ihrer Probleme haben Methoden zum automatisierten Testfalldesign durchaus ihre Daseinsberechtigung. Sie können die Arbeit des Testers beschleunigen, indem sie ihm einen Grundstock an Testfällen an die Hand geben, der durch weitere Testfälle erweitert werden kann. Sinnvoll ist es, sich nicht ausschließlich auf die Möglichkeit des automatisierten Testfalldesigns zu stützen, sondern sie mit den Fähigkeiten eines Tester zu kombinieren.
Seidel et al. [SBB12, vgl. S. 27] beschreiben eine Reihe an Testfallentwurfsmethoden, die sie unter dem Oberbegriff der Kombinatorik zusammenfassen. Diese Entwurfsmethoden zielen darauf ab, aus einer Fülle von möglichen Eingaben diejenigen herauszufiltern, die ein hohes Fehlerpotenzial in sich bergen. Darunter fallen beispielsweise:

- Äquivalenzklassenbildung
- Grenzwertanalyse
- Klassifikationsbaummethode

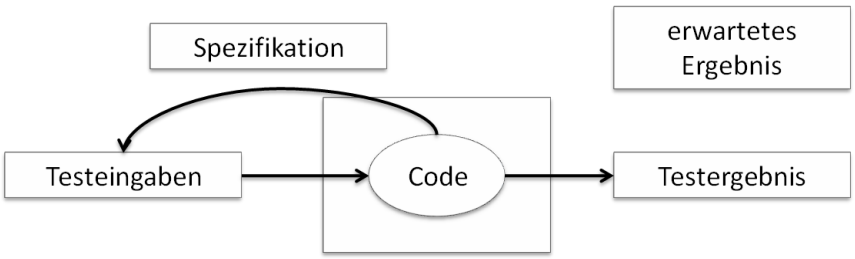
Diese Entwurfsmethoden kommen vor allem beim manuellen Designen von Testfällen zum Einsatz, bieten aber auch die Möglichkeit toolgestützt und damit automatisiert abzulaufen. Fewster und Graham [FG99, vgl. S. 19 ff.] beschreiben eine Reihe von weiteren Methoden zum Erzeugen von Eingabedaten, die vermehrt in automatisierten Tools zum Einsatz kommen:

- Code-basierte Generierung von Eingabedaten
- Interface-basierte Generierung von Eingabedaten
- Spezifikations-basierte Generierung von Eingabedaten

3.2.1.1. Code-basierte Generierung von Eingabedaten

Die Generierung der Eingabedaten erfolgt bei diesem Ansatz anhand der Struktur des Codes (siehe Abbildung 3.3). Jede Eingabe bedingt einen fest vorbestimmten Ablauf durch das Programm. Anhand des Codes können daher die benötigten Eingaben ermittelt werden, die für das durchlaufen von unterschiedlichen Pfaden im Programm benötigt werden. Fewster und Graham [FG99, vgl. S. 19 ff.] sehen in diesem Ansatz jedoch Probleme. Ein Testfall benötigt immer auch ein erwartetes Ergebnis. Über die Code-basierte Generierung ist es nicht möglich diese Ergebnisse zu ermitteln. Die generierten Testfälle sind also unvollständig. Ein weiteres Problem dieses Vorgehens ist, dass ausschließlich der Code getestet wird, der

bereits implementiert wurde. Fehlende Funktionalitäten können so nicht erkannt werden. Es wird getestet, dass der Code das ‘tut, was er tut und nicht das, was er tun soll.‘ [FG99, vgl. S. 20]

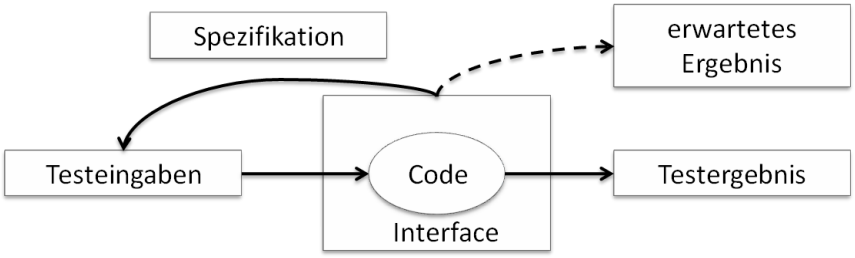


Quelle: [FG99, vgl. S. 19]

Abbildung 3.3.: Code-basierte Generierung von Testfällen

3.2.1.2. Interface-basierte Generierung von Eingabedaten

Bei dieser Methode erfolgt nach Fewster und Graham [FG99, vgl. S. 20] die Generierung anhand von gut definierten Schnittstellen, wie der Benutzeroberfläche einer Desktop- oder Web-Anwendung (siehe Abbildung 3.4). Wird als Schnittstelle die Benutzeroberfläche gewählt, kann beispielsweise getestet werden, ob eine Checkbox nach einer Interaktion aktiviert bzw. deaktiviert wurde.
Eine weitere Möglichkeit wäre es, rekursiv jeden Link in einer Webanwendung zu durchlaufen. Alle defekten Links der Anwendung könnten auf diese Weise identifiziert werden.
Mit Hilfe dieses Ansatzes ist es laut Fewster und Graham [FG99, vgl. S. 21] auch möglich, einfache Akzeptanzkriterien zu generieren. Werden beispielsweise die Links einer Web-Anwendung rekursiv durchlaufen, könnte als Akzeptanzkriterium geprüft werden, ob nach dem ausführen eines Links auch eine neue Webseite geladen wurde.

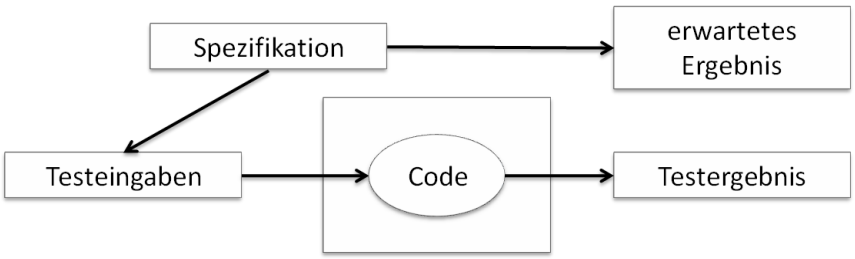


Quelle: [FG99, vgl. S. 20]

Abbildung 3.4.: Interface-basierte Generierung von Testfällen

3.2.1.3. Spezifikations-basierte Generierung von Eingabedaten

Mit Hilfe von Spezifikations-basierter Generierung ist es nach Fewster und Graham [FG99, vgl. S. 21] möglich sowohl Testeingaben, als auch die zugehörigen erwarteten Ergebnisse zu erzeugen (siehe Abbildung 3.5). Als Basis wird dazu eine Spezifikation benötigt, die automatisiert analysiert werden kann. Die Möglichkeiten dafür reichen von natürlicher Sprache, die gewissen Strukturen folgt, bis hin zu technischen Modellen.
Vor allem die Benutzung von Modellen hat in den letzten Jahren immer mehr an Bedeutung gewonnen und ist heute unter dem Namen 'modellbasiertes Testen' bekannt. Als Referenz auf diesem Gebiet kann das Werk von Roßner et al. [Ros10], „Basiswissen Modellbsierter Test", dienen.
Ein Vorteil des Spezifikations-basierten Ansatzes ist nach Fewster und Graham [FG99, vgl. S. 21] auch, dass die Testfälle nicht auf Basis einer Implementierung, sondern auf Basis einer Spezifikation erzeugt werden. Damit wird sichergestellt, dass sie nicht nur, wie bei der Code-basierten Generierung, überprüfen 'was die Software tut', sondern 'was die Software tun soll'.



Quelle: [FG99, vgl. S. 21]

Abbildung 3.5.: Spezifikations-basierte Generierung von Testfällen

3.2.2. Testcodeerstellung

Bei der Testautomatisierung versteht man unter Testcodeerstellung das Erzeugen von Testcode, der später wiederholt gestartet werden kann. Der erzeugte Code setzt die Testfälle um, die zuvor in der Designphase erarbeitet wurden. In vielen Fällen handelt es sich hierbei um einen manuellen Schritt. Die Testcodeerstellung ist oft eine reine Entwicklertätigkeit, bei der ein Tester die angedachten Testfälle als Testcode implementiert. Aber auch in diesem Schritt sind laut Amannejad et al. [Ama+14] Möglichkeiten für eine Automatisierung gegeben.
Es existieren beispielsweise teilautomatisierte Ansätze. Mit Hilfe von sogenannten 'record-and playback'-Tools (R&PB) können die Interaktionen eines Benutzers mit der zu testenden

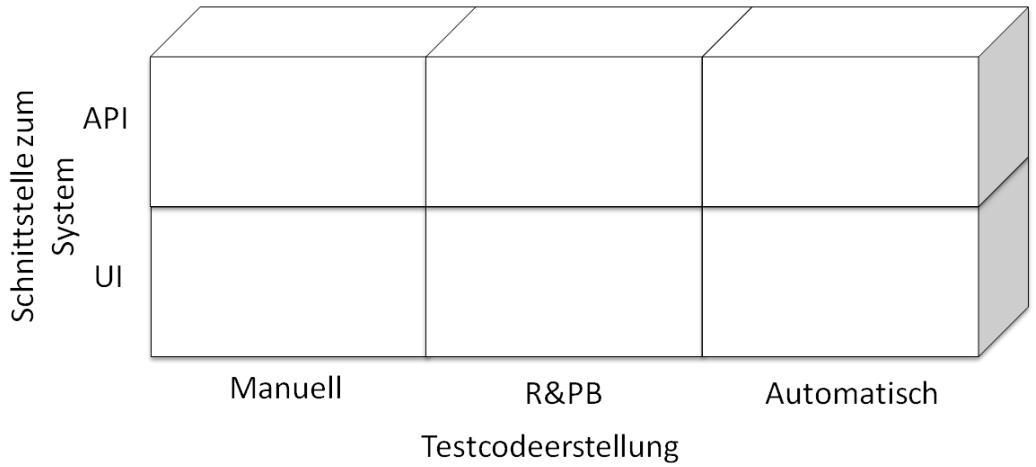
Software aufgezeichnet werden. Die aufgezeichneten Abläufe können dann verwendet werden, um automatisierte Testskripte zu generieren.
Auch ein vollautomatisierter Ansatz ist möglich, wenn auch nicht so weit verbreitet, wie der manuelle bzw. teilautomatisierte Ansatz. Mittels modellbasiertem Testen ist es nicht nur möglich, wie in Kapitel 3.2.1.3 angesprochen, Testfalldesigns abzuleiten. Liegen die Modelle in einem entsprechend hohen Detailgrad vor, kann daraus sogar direkt Testcode erzeugt werden. Bouquet et al. [Bou+08] beschreiben beispielsweise einen modellbasierten Ansatz, mit dessen Hilfe das Designen, Erstellen und Ausführen von Testfällen automatisiert geschehen kann.
Zusammengefasst ist die Testcodeerstellung damit in drei unterschiedlichen Automatisierungsgraden möglich:

- Manuell
- Teilautomatisiert (R&PB)
- Automatisiert

Allen Ansätzen gemeinsam ist, dass sie immer eine Schnittstelle zum System benötigen, über welche der automatisierte Testcode mit der zu testenden Anwendung kommunizieren kann. Meszaros et al. [Mes03] unterscheiden dabei zwei Hauptangriffspunkte:

- API (application programming interface): Als Schnittstelle wird der Code der zu testenden Anwendung direkt benutzt.
- UI (user interface): Als Schnittstelle wird die Benutzeroberfläche der Anwendung verwendet.

Kombiniert man die verschiedenen Ansätze der Testcodeerstellung mit den Schnittstellen, ergibt sich eine Matrix, welche die verschiedenen Möglichkeiten der Testcodeerstellung beim automatisierten Testen abbildet. Eine grafische Darstellung dieser Matrix bietet Abbildung 3.6.
Meszaros et al. [Mes03] haben eine ähnliche Matrix aufgestellt, die noch um eine weitere Dimension der Testgranularität (Unit-, Integrations- und System-Test) erweitert ist. Diese Dimension hat allerdings nur wenig Auswirkung auf die Herangehensweise in der Testautomatisierung und wurde deshalb in Abbildung 3.6 nicht mit aufgeführt. Im Folgenden wird, mit Hilfe von Beispielen, auf die Schnittstellen API und UI in Kombination mit der Manuellen- und R&PB-Testcodeerstellung näher eingegangen. Die vollautomatisierte Testcodeerstellung mittels modellbasierten Testen, dargestellt durch die beiden rechten



Quelle: vgl. [Mes03]

Abbildung 3.6.: Verschiedene Möglichkeiten der Testcodeerstellung

Quader in Abbildung 3.6, befindet sich außerhalb des Rahmens dieser Arbeit und wird daher nicht genauer betrachtet.

3.2.2.1. API

Unter der Abkürzung API sind in diesem Zusammenhang alle Schnittstellen zu verstehen, die intern von der zu testenden Anwendung angeboten werden. Darunter fällt beispielsweise das direkte aufrufen von Servicemethoden, die als Businesslogik innerhalb der Anwendung bereitgestellt werden.
Eine weit verbreitete Gruppe von Frameworks, welche oft diese Art der Schnittstelle verwenden, sind die modernen 'XUnit-Frameworks'. In den meisten Programmiersprachen existiert ein Unit-Framework, mit dem es möglich ist, direkt die im Programm angebotene Logik mit automatisierten Testfällen zu überprüfen. In der Praxis wird diese Aufgabe meist direkt von einem Entwickler, in Form von Unit-Tests (Komponenten-Tests) übernommen. Der Begriff 'XUnit-Frameworks' kann leicht missverstanden werden, da er impliziert, dass es sich bei den mit Hilfe des Frameworks umgesetzten Testfällen immer um Unit-Tests handelt. Es können jedoch Testfälle aller Teststufen, von Unit- bis System-Test, mit Hilfe dieser Frameworks umgesetzt werden. Die Testcodeerstellung erfolgt dabei manuell, womit diese Art der Automatisierung ein Beispiel für den oberen linken Quadranten in Abbildung 3.6 darstellt.
Neben dem manuellen Vorgehen beschreiben Meszaros et al. [Mes03] auch die Möglichkeit 'record-and playback' für Testfälle zu verwenden, die als Schnittstelle die API ansteuern. Hierfür muss eine 'record-and playback'-Mechanismus auf API-Ebene für die zu testende

Software entwickelt werden. Darüber ist es dann bei einem manuellem Testlauf möglich, all das aufzuzeichnen, was den Zustand des Systems beeinflusst hat. Aus diesen Aufzeichnungen kann dann Testcode erzeugt werden, der es ermöglicht die gewünschten Abläufe zu wiederholen. Solch ein Vorgehen wäre ein Beispiel für den oberen mittleren Quadranten in Abbildung 3.6. Diese Art der Testcodeerstellung findet aber keine verbreitete Anwendung.

3.2.2.2. UI

Bei Testfällen, die als Schnittstelle zum System das User-Interface verwenden, ist laut Meszaros et al. [Mes03] vor allem ‘record-and playback‘ ein sehr weit verbreiteter Ansatz. Es existieren zahlreiche kommerzielle Testwerkzeuge, um dieses Vorgehen zu unterstützen, wie beispielsweise HP Unified Functional Testing [HP15], aber auch Open-Source-Lösungen wie Selenium [Sel15a].
Diese Tools ermöglichen es Abläufe aufzuzeichnen, um sie später automatisch zu wiederholen. Die Testfälle werden dafür zunächst manuell auf der Benutzeroberfläche der Anwendung durchgeführt. Während dieser manuellen Testausführung können die vom Tester getätigten Eingaben vom Tool gespeichert werden. Die gespeicherten Interaktionen werden dann verwendet, um daraus Testcode zu generieren, mit dem die Abläufe auf der Benutzeroberfläche beliebig oft wiederholt werden können. Dieses Art der Testcodeerstellung wird in Abbildung 3.6, durch den unteren mittleren Quadranten dargestellt.
Der untere linke Quadrant steht für manuell erstellten Testcode, der als Schnittstelle zum System das User-Interface verwendet. Oft handelt es sich dabei um Testfälle, die analog zu Abschnitt 3.2.2.1 ein ‘XUnit-Frameworks‘ zur Testausführung verwenden. Diese Frameworks können um Aufsätze, wie beispielsweise HttpUnit [Htt15] oder Selenium [Sel15a] erweitert werden. Mit Hilfe dieser Erweiterungen ist es möglich, Testfälle nicht mehr nur gegen die API der zu testenden Software zu entwickeln, sondern zur Steuerung die Benutzeroberfläche zu verwenden. Neben Aufsätzen für vorhandene Frameworks existieren auch zahlreiche eigenständige Tools, wie beispielsweise MicroFocus SilkTest [Sil15], welche neben Webanwendungen auch die Möglichkeit zum Testen von Desktopanwendungen bieten.
Testcode von Hand zu schreiben ist ein weit verbreitetes Vorgehen. Vor allem immer dann, wenn kein vorhandenes Testtool verwendet wird, sondern das Automatisierungsframework für die zu testende Software selbst entwickelt wurde.

3.2.3. Testdurchführung

Unter Testdurchführung versteht man das Ausführen der Testfälle, sowie das Aufzeichnen von Testergebnissen bzw. Testausgaben. Nach den Erfahrungen von Amannejad et al. [Ama+14] ist dieser Bereich der Testautomatisierung derjenige, der von den meisten Testern am engsten mit der Automatisierung verbunden wird.
Die Möglichkeit zur Automatisierung hängt in diesem Schritt stark von den gewählten Methoden in den vorangegangenen Phasen des Testprozesses ab. Oft ist eine automatisierte Testdurchführung bereits ohne zusätzlichen Aufwand möglich. Vor allem dann, wenn der Testcode zuvor Tool bzw. Framework unterstützt erstellt wurde. Die meisten Tools und Frameworks ermöglichen bereits eine automatisierte Testdurchführung.
Im besonderen Maße sind an dieser Stelle, aufgrund ihrer hohen Verbreitung, erneut die 'XUnit-Frameworks' zu nennen. In Abschnitt 3.2.2 wurde aufgezeigt, dass diese Frameworks oft in der manuellen Testcodeerstellung verwendet werden. Das Unit-Framework übernimmt dabei die automatisierte Ausführung des Testcodes, sowie die Präsentation der Testergebnisse und ist damit ein Beispiel für ein Framework, welches die Testdurchführung automatisiert.
Probleme ergeben sich, wenn Testcode losgelöst von bereits vorhandenen Tools bzw. Frameworks entwickelt wurde. Der Testcode könnte beispielsweise gekapselt in Funktionen vorliegen und seine Ergebnisse in die Konsole loggen. Steigt die Anzahl der Testfälle, wird die Ausführung und vor allem die folgende Auswertung zum Problem. Der Aufwand, der später betrieben werden muss, um die Testfälle anhand ihrer Logmeldungen auszuwerten, kann so eine Dimension erreichen, die nicht mehr wirtschaftlich ist. Ein weiteres Problem stellt dar, dass eine gesonderte Ausführung von einzelnen Testfällen bzw. kleineren Testfallgruppen nur mit zusätzlichem Aufwand möglich ist.
Werden Testfälle daher ohne Tool- bzw. Framework-unterstützung erstellt, müssen zeitnah Überlegungen angestellt werden, wie die diese einfach und auswertbar zur Ausführung gebracht werden können. Diese Überlegungen resultieren meist in einem eigen entwickelten Testframework, welches die Ausführung des automatisierten Testcodes und die Aufzeichnung der Ergebnisse übernimmt.

3.2.4. Testauswertung

Nachdem die zu testende Software mittels eines Testfalls ausgeführt wurde, muss bestimmt werden, ob dieser erfolgreich oder fehlerhaft war. Diese Prüfung kann manuell erfolgen. Handelt es sich jedoch um automatisierte Testfälle, ist diese Prüfung meist über feste Akzeptanzkriterien direkt im Testfall hinterlegt. Im Fall von Testfällen, deren Durchführung

über ein ‘XUnit-Frameworks‘ automatisiert wurde, werden Akzeptanzkriterien beispielsweise über sogenannte ‘Assertions‘ festgelegt.
Um die erwarteten Ergebnisse für einen Testfall zu finden, werden sogenannte Testorakel verwendet. Jede Quelle, die Auskunft über ein zu erwartendes Ergebnis gibt, kann als Testorakel dienen. Meist handelt es sich um Spezifikationen, die manuell ausgewertet werden. Darüber hinaus gibt es aber auch zahlreiche Möglichkeiten und Versuche, diese zu automatisieren [MPS00] [RAO92] [SKM09]. Automatisierte Testorakel können dann wiederum für die Auswertung der Testergebnisse dienen. Eine manuelle Auswertung kann so entfallen.
Bereits mit fest hinterlegten Akzeptanzkriterien, die manuell gefunden wurden, gilt laut Amannejad et al. [Ama+14] die Testauswertung als automatisiert. Mittels automatisierter Testorakel kann diese jedoch noch ein höheres Level an Intelligenz erreichen.

4. Testautomatisierung mit Selenium

Laut Seidel et al. [SBB12, vgl. S. 48] ist der am häufigsten genutzte »Angriffspunkt« für Testautomatisierung die grafische Benutzerschnittstelle. Seidel et al. [SBB12, S. 48] nennen dafür folgende Gründe:

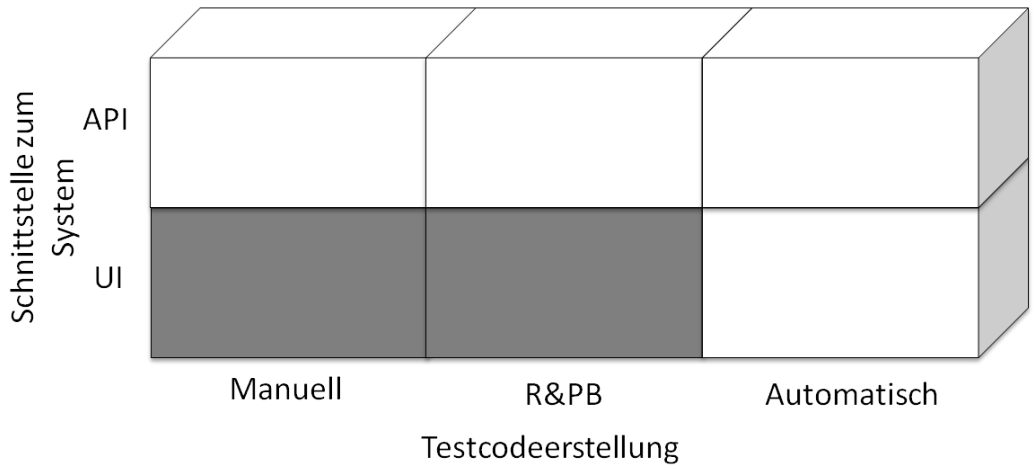
- „Sie ist für Tester und Automatisierer anschaulich und leicht greifbar."
- „Sie stellt zumeist das Verhalten im realen Umfeld am besten nach."
- „Die Dokumentation von Systemen ist auf dieser Ebene meist am vollständigsten."
- „Der klassische Systemtest wird oft über diese Schnittstelle abgewickelt."
- „Hinter der grafischen Benutzerschnittstelle liegende Systeme werden implizit getestet."

Ein großer Teil der heutzutage entwickelten Anwendungen werden in Form einer Webapplikation realisiert. Für automatisierte Testfälle, die als Schnittstelle die grafische Benutzeroberfläche verwenden, stellen diese Webapplikationen einen Sonderfall dar. Elemente auf der Oberfläche können besonders gut angesprochen werden. Im Gegensatz zu gewöhnlichen Desktopanwendungen gibt es bei dieser Art von Anwendung, wie Seidel et al. [SBB12, vgl. S. 88] feststellen, „keinen spezifischen Client für eine Applikation, sondern einen generischen - den Browser." Dieser schafft nach Seidel et al. [SBB12, vgl. S. 59] für Werkzeuge eine sehr gute Basis, um auf Elemente der Seite zuzugreien. Die einzelnen HTML-Elemente und deren Attribute können verwendet werden, um die Bestandteile einer Seite zu adressieren. Ein weit verbreitetes Tool für die Automatisierung, welches diesen Ansatz verfolgt, ist Selenium [Sel15a].

4.1. Selenium

Seidel et al. [SBB12, S. 142] beschreiben Selenium als „eines der gängigsten Open-Source-Automatisierungswerkzeuge für Webapplikationen."
Ordnet man Selenium der in Kapitel 3.2.2 erstellten Unterteilung in der Testcodeerstellung

zu, ist dieses im unteren mittleren und unteren linken Quadranten angesiedelt. Abbildung 4.1 hinterlegt diese Bereiche farblich. Selenium ist also ein Tool, mit welchem Testfälle erstellt werden können, die als Schnittstelle die grafische Benutzeroberfläche eine Webanwendung verwenden. Die Testcodeerstellung erfolgt dabei manuell oder teilautomatisiert mittels 'record-and-playback'.



Quelle: vgl. [Mes03]

Abbildung 4.1.: Einordnung von Selenium in die verschiedene Möglichkeiten der Testcodeerstellung

Selenium besteht nicht nur aus einem einzelnen Tool, sondern aus einer Vielzahl an Tools, die unter dem Namen 'Selenium' zusammengefasst werden. In seiner aktuellen Ausprägung 2.48 lassen sich, abgesehen von Komponenten, die der Abwärtskompatibilität dienen, laut Dokumentation [Sel15b], drei Komponenten unterscheiden:

- Selenium IDE
  Bei der Selenium IDE handelt es sich um ein Firefox Plug-in, das verwendet werden kann, um Selenium-Testskripte zu erstellen. Testskripte können von Hand erstellt, oder mittels eines 'record-and-playback'-Mechanismus direkt im Browser aufgezeichnet werden. Die erstellten Testfälle können mit Akzeptanzkriterien angereichert und innerhalb der IDE wieder abgespielt werden.

- Selenium WebDriver
  Der Selenium WebDriver bietet für verschiedene Programmiersprachen eine Schnittstelle zur Steuerung eines Browsers aus dem Programmcode heraus. Der WebDriver

bildet damit die Kernkomponente für alle Selenium-Testfälle, die außerhalb der Selenium IDE entwickelt werden.

- Selenium Server/Grid
  Mit Hilfe des Selenium Servers ist es möglich, Selenium-Testfälle nicht nur auf dem eigenen Rechner auszuführen, sondern die Ausführung auf einen Server auszulagern. Einen wichtigen Teil des Selenium Server bildet Selenium Grid. Selenium Grid bietet die Möglichkeit, die Ausführung über einen Server hinaus auf eine Vielzahl von Knoten zu verteilen. Der Selenium Server dient dann als Hub, der die Testfallanfragen auf registrierte Knoten zur Ausführung weiterleitet.

4.2. Testdurchführung mit Selenium

Abhängig davon, ob die Testfälle für die Selenium IDE entwickelt wurden, oder sich auf den Selenium WebDriver stützen, unterscheiden sich die Möglichkeiten zur Ausführung der Testskripte.
Testfälle, die mit der Selenium IDE entwickelt wurden, verwenden eine eigene Sprache, mit dem Namen ‘Selenese‘. Diese Testfälle können in späteren Testläufen wieder über die Selenium IDE zur Ausführung gebracht werden.
Dem gegenüber stehen Testfälle, die den Selenium WebDriver verwenden. Testfälle die mittels Selenium WebDriver entwickelt wurden, sind für ihre Ausführung nicht an ein spezielles Tool gebunden. Beim WebDriver handelt es sich um eine API, mit deren Hilfe ein Browser ferngesteuert werden kann. Die Integration der API wird dem Ersteller selbst überlassen. In der Praxis hat sich als Best Practice herausgestellt, den WebDriver in Verbindung mit einem Unit-Framework einzusetzen. Im Java-Umfeld wären hier beispielsweise JUnit oder TestNG zu nennen. Die Testfälle können analog zu den klassischen Unit Tests entwickelt werden, verwenden jedoch als Schnittstelle zum System die Benutzeroberfläche der Webanwendung. Die Ausführung erfolgt identisch zu den klassischen Unit Tests.
Bei Verwendung des WebDrivers ist Selenium durch die eingesetzten Unit-Frameworks in den meisten Programmiersprachen bereits gut in die Infrastruktur integriert. Testfälle in Java, die mittels JUnit ausgeführt werden, sind in allen gängigen IDEs durch Plugins unterstützt. Große Bedeutung hat auch eine gute Integration in den Buildprozess. Werden in Java Standardtools, wie beispielsweise Gradle oder Maven zum Bauen der Projekte verwendet, können die Testfälle ohne Mehraufwand im Rahmen des Buildprozesses ausgeführt werden.

4.3. Testcodeerstellung mit Selenium

Die Testcodeerstellung bei Testfällen, die Selenium verwenden, kann auf zwei Arten erfolgen. Der Testcode kann manuell erstellt oder über einen ‘record-and-playback‘-Mechanismus teilautomatisiert generiert werden.

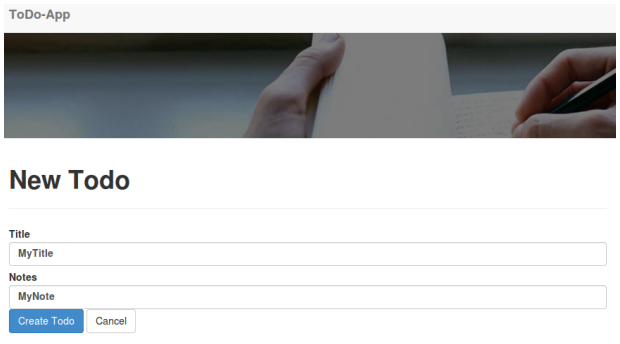
4.3.1. Record-and-playback

Die Selenium IDE bietet die Möglichkeit, Testskripte mittels ‘record-and-playback‘ zu erzeugen. Die im Testfall gewünschten Abläufe werden dazu zunächst im Browser durchgeführt. Die Selenium IDE zeichnet während dessen die durchgeführten Schritte, in der Selenium eigenen Sprache ‘Selense‘, auf. Diese Aufzeichnungen können zu einem späteren Zeitpunkt von der Selenium IDE erneut aufgerufen werden, um die Testabläufe zu wiederholten. Testskripte können nach der Aufzeichnung überarbeitet werden. Auf diese Weise ist es beispielsweise möglich, Akzeptanzkriterien an bestimmten Stellen einzuarbeiten.
Testfälle, die über den ‘record-and-playback‘-Mechanismus erstellt wurden, sind nicht an die Sprache ‘Selenese‘ gebunden. Die Selenium IDE bietet die Möglichkeit, Testskripte in eine Reihe von Programmiersprachen zu exportieren, darunter auch Java. Diese Testfälle benutzen dann, wie in Abschnitt 4.2 beschrieben, den Selenium WebDriver für die Kommunikation mit dem Browser und ein Unit-Framework für die Ausführung.
Die ‘record-and-playback‘-Funktionalität bietet einen besonders einfachen und schnellen Weg, Testfälle zu erstellen. Dennoch wird in der Dokumentation von Selenium [Sel15b] davon abgeraten, sich bei der Testfallerstellung alleine auf dieses Tool zu stützen. Die Selenium IDE wird als Prototyping-Tool verstanden, mit dem kleinere Aufgaben, die nicht für den längerfristigen Einsatz gedacht sind, schnell automatisiert werden können.

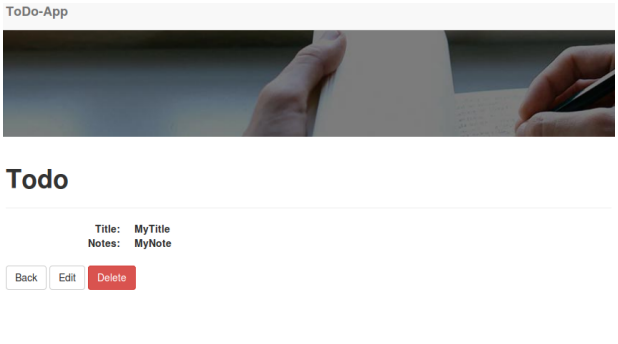
4.3.1.1. Probleme von record-and-playback

Testabläufe die über den ‘record-and-playback‘-Mechanismus der IDE erstellt werden, unterliegen eine Reihe von Einschränkungen. Als Limitierungen werden von Leotta et al. [Leo+13] das Fehlen von bedingten Anweisungen, Schleifen, Logging, Ausnahmebehandlungen sowie parametrisierten (a.k.a. data-driven) Testfällen genannt. Neben diesen Limitierungen besteht zusätzlich das Problem einer schlechten Wartbarkeit. Nach Leotta et al. [Leo+13] liegt das vor allem daran, dass die Testfälle sehr stark mit der Struktur der Webseiten verwoben sind und einen hohen Anteil an dupliziertem Code aufweisen. Die Limitierungen der IDE können zwar durch das Exportieren der Testfälle in eine Programmiersprache überwunden werden,

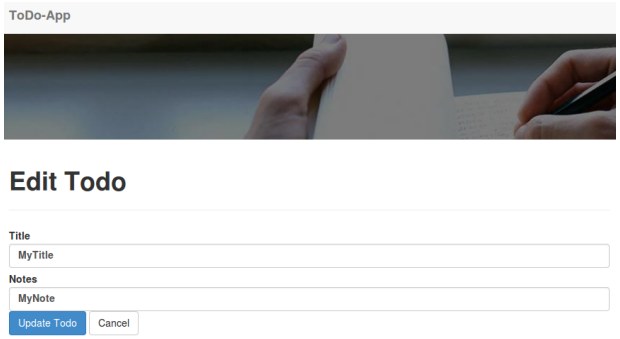
die Qualität im Bezug auf ihre spätere Wartbarkeit kann nach Leotta et al. [Leo+13] jedoch nicht auf diesem Weg verbessert werden.
Um die starke Koppelung mit der Webanwendung und den hohen Anteil an dupliziertem Code zu veranschaulichen, wurden zwei Testfälle über den ‘record-and-playback‘-Mechanismus von Selenium erstellt und in die Programmiersprache Java exportiert. Diese beiden Testfälle sollen das Anlegen und Editieren eines Datensatzes auf einer Webanwendung überprüfen. Drei Seiten der Anwendung werden hierfür verwendet. Die Seiten sind in Abbildung 4.2 dargestellt.



(a) Anlegen eines neuen Datensatzes



(b) Anzeigen eines Datensatzes



(c) Editieren eines Datensatzes

Abbildung 4.2.: Anlegen, Editieren und Anzeigen eines neuen Datensatzes

Zur besseren Lesbarkeit wurden die Testfälle im Listing 4.1 überarbeitet, in ihrem grundlegendem Aufbau jedoch nicht verändert.

Listing 4.1: Exportierte Testfälle

```
1  /**
2  * Testfall legt einen neuen Datensatz an.
```

```
3   */
4   @Test
5   public void testCreateNewRecord() {
6     WebDriver driver = new FirefoxDriver();
7     driver.get("http://localhost:3000/todos/new");
8     driver.findElement(By.id("todo_title")).sendKeys("MyTitle");
9     driver.findElement(By.id("todo_notes")).sendKeys("MyNote");
10    driver.findElement(By.name("commit")).click();
11
12    assertEquals("MyTitle", driver.findElement(By.id("title")).getText());
13    assertEquals("MyNote", driver.findElement(By.id("note")).getText());
14  }
15
16  /**
17  * Testfall legt einen neuen Datensatz an und editiert ihn.
18  */
19  @Test
20  public void testEditRecord() {
21    WebDriver driver = new FirefoxDriver();
22    driver.get("http://localhost:3000/todos/new");
23    driver.findElement(By.id("todo_title")).sendKeys("MyTitle");
24    driver.findElement(By.id("todo_notes")).sendKeys("MyNote");
25    driver.findElement(By.name("commit")).click();
26    driver.findElement(By.linkText("Edit")).click();
27    driver.findElement(By.id("todo_title")).clear();
28    driver.findElement(By.id("todo_title")).sendKeys("MyTitleEdit");
29    driver.findElement(By.id("todo_notes")).clear();
30    driver.findElement(By.id("todo_notes")).sendKeys("MyNoteEdit");
31    driver.findElement(By.name("commit")).click();
32
33    assertEquals("MyTitleEdit", driver.findElement(By.id("title")).getText());
34    assertEquals("MyNoteEdit", driver.findElement(By.id("note")).getText());
35  }
```

Die in im Listing 4.1 gezeigten Testfälle sind in ihren Abläufen recht ähnlich. Beide legen zunächst in der Anwendung einen neuen Datensatz an. Die hierfür verwendete Logik ist dupliziert worden.

Das Duplizieren von Code ist ein Problem, unter dem viele auf diesem Weg erzeugte Testfälle leiden. In den meisten Testabläufen finden sich wiederkehrende Aufgaben, die nicht nur von einem Testfall benötigt werden. Selbst wenn sich die Abläufe zwischen den Testfällen stark unterscheiden, werden Elemente, die von der Anwendung angeboten werden, in der Regel mehrmals benutzt. Das führt dazu, dass der Code zum Adressieren dieser Felder in einer Vielzahl von Testfällen benutzt wird.
Als weiteres Problem haben Leotta et al. [Leo+13] eine starke Koppelung der Testfälle mit der Webanwendung genannt. Die Selektoren, die in den gezeigten Testfällen verwendet werden, um die Elemente der Webanwendung anzusteuern, sind direkt in den Code eingearbeitet. Änderungen an der Seitenstruktur der Anwendung, haben damit direkten Einfluss auf die Testfälle. Obwohl sich die eigentliche Testfallspezifikation durch eine Änderung in der Seitenstruktur nicht ändert, müssen die Testfälle somit trotzdem überarbeitet werden.
Der duplizierte Code und der hohe Grad an Koppelung mit der Anwendung innerhalb der Testfälle bedingt, dass selbst kleine Änderungen in der zu testenden Anwendung Korrekturen an vielen Stellen in den Testfällen notwendig macht.

4.3.2. Manuell

Eine weitere Möglichkeit bildet das manuelle Erstellen der Testskripte. Für die Ausführung und Kommunikation mit der Anwendung, verwenden die manuell erstellten Skripte das selbe Toolset, wie die mittels ‘record-and-playback‘ generierten und exportierten Testfälle. Analog zu den in Listing 4.1 gezeigten Testfällen, wird auch bei diesen Skripten der Selenium WebDriver für die Kommunikation mit der Anwendung verwendet. Die Ausführung geschieht in der Regel ebenfalls über ein Unit-Framework.
Im Vergleich zu den generierten Testfällen, ist das manuelle Entwickeln der Testskripte aufwendiger. Es bietet jedoch die Möglichkeit, den in Abschnitt 4.3.1.1 genannten Problemen der ‘record-and-playback‘-Variante entgegenzuwirken.
Bei einem manuellen Ansatz kann von Anfang an auf eine wartbare und wiederverwertbare Struktur geachtet werden. Als Best Practice hat sich zu diesem Zwecke das Page Object Design Pattern durchgesetzt.

4.3.3. Page Object Pattern

Im Page Object Pattern wird versucht, die Funktionalität, welche die zu testende Anwendung anbietet, in einem objektorientierten Ansatz zu kapseln. Alle Seiten der zu testenden Anwendung werden dazu als Kassen, sogenannte Page Objects, modelliert. Jede dieser Klassen

verwaltet zentral alle Informationen, sowie die Funktionalität, die von der jeweils korrespondierenden Webseite angeboten wird. Die Hauptbestandteile eines Page Objects bilden Selenium-WebElemente. Im Rahmen dieser Arbeit werden diese als ‘Elemente‘ bezeichnet. Elemente werden in den Page Objects meist als globale Variablen angeboten und bilden die Verbindung zu den HTML-Elementen, welche auf der entsprechenden Webseite der Anwendung vorhanden sind. Um die Elemente auf der HTML-Seite zu identifizieren wird meist ein XPath-Ausdruck oder CSS-Selektor verwendet. Diese Ausdrücke bzw. Informationen die zum erzeugen dieser Ausdrücke verwendet werden, nennt man ‘Lokatoren‘.
Manche Elemente, wie beispielsweise Links oder bestimmte Buttons, haben die Besonderheit, dass sie einen Seitenwechsel in der Webanwendung hervorrufen. Diese Elemente werden in der Arbeit als ‘Transitionen‘ bezeichnet. Transitionen werden in den Page Objects meist in Form einer Methode abgebildet, die als Rückgabe das Page Object der entsprechenden Zielseite liefert. Funktionalität, die von einer Seite angeboten wird, wird innerhalb eines Page Objects ebenfalls gekapselt in Methoden abgebildet.
Ein Page Object ist also eine objektorientierte Klasse, die als Interface für eine Seite der zu testenden Anwendung dient. Sämtliche Interaktion mit der zu testenden Anwendung, geschieht über die in den Page Objects angeboten Schnittstellen. Änderungen an der Oberfläche der zu testenden Anwendung haben so keinen direkten Einfluss mehr auf die Testfälle. Bei Änderungen an der Benutzeroberfläche muss nur noch Code an einer Stelle, innerhalb der Page Objects, angepasst werden.
Um das Zusammenspiel zwischen Page Objects und Testfall zu verdeutlichen, wurde der Test zum Anlegen eines neuen Eintrags aus Listing 4.1 mit dem Page Object Pattern nachgebaut. Der Testfall arbeitet mit zwei Seiten der Anwendung. Die Seiten wurden bereits in Abbildung 4.2 dargestellt. Für jede der beiden Seiten wird ein Page Object als Kommunikationsschnittstelle benötigt. Das Page Object CreatePage in Listing 4.2 repräsentiert die Seite zum Anlegen eines neuen Datensatzes (Abbildung 4.2 a). Das Page Object ShowPage in Listing 4.3 repräsentiert die Seite zum Anzeigen eines Datensatzes (Abbildung 4.2 b).

Listing 4.2: Page Object CreatePage

```
1  public class CreatePage extends BasePo {
2      public final Control tfTodotitle = control(by.textField("todo_title"));
3      public final Control tfTodonotes = control(by.textField("todo_notes"));
4      public final Control bCreateTodo = control(by.button("commit"));
5
6      public CreatePag(PageObject po) {
7         super(po);
```

```
8       }
9       public ShowPage createEntry(String title, String note){
10         tfTodotitle.sendKeys(title);
11         tfTodonotes.sendKeys(note);
12         bCreateTodo.click();
13         return new ShowPage(this);
14      }
15  }
```

Listing 4.3: Page Object ShowPage

```
1   public class ShowPage extends BasePo {
2       public final Control idTitle = control(by.id("title"));
3       public final Control idNotes = control(by.id("note"));
4
5       public ShowPage(PageObject po) {
6          super(po);
7       }
8   }
```

Die Funktionalitäten, die von den jeweiligen Seiten angeboten werden, sind innerhalb des Page Objects gekapselt. Das Page Object CreatePage bietet beispielsweise die Eingabefelder für Title und Note, sowie den Button zum Anlegen eines Datensatzes, als globale Objekte vom Typ Control an.
Die Klasse Control dient in den dargestellten Page Objects als Wrapper für Selenium Webelemente. Control-Objekte sind also analog zu Webelementen zu verstehen.
Die Funktionalität zum Anlegen eines neuen Eintrags, die im späteren Testfall benötigt wird, bietet das Page Objekt CreatePage, als Methode ‘createEntry()‘ an.

Der Testfall in Listing 4.4 verwendet beide Page Objects, um einen neuen Eintrag in der Anwendung anzulegen und zu überprüfen.

Listing 4.4: Page Object Testfall

```
1   /**
2   * Testfall legt einen neuen Datensatz an.
3   */
```

```
4    @Test
5    public void testCreateNewRecord(){
6       CreatePage createPage = new CreatePage(po);
7       ShowPage showPage = createPage.createEntry("MyTitle", "MyNote");
8
9       assertEquals("MyTitle", showPage.idTitle.resolve().getText());
10      assertEquals("MyNote", showPage.idNotes.resolve().getText());
11   }
```

Im Gegensatz zum Testfall in Listing 4.1 ist es auf Grund des Page Object Pattern nicht mehr nötig, explizite Referenzen auf die Struktur der Seite in den Testfällen zu machen. Alle Details der Seite sind innerhalb der Page Objects gekapselt. Der Testfall verwendet lediglich die im Page Object angebotene Funktionalität.

4.3.3.1. Vorteile des Page Object Pattern

Folgende Vorteile ergeben sich bei der Verwendung des Page Object Pattern, die auch in der Dokumentation von Selenium [Sel15c] angegeben werden:

1. Es gibt eine klare Trennung zwischen Testcode und seitenspezifischem Code, wie beispielsweise Elementadressierungen und Layout.

   Die Adressierung der Elemente ist nicht mehr über die gesamten Testfälle verteilt, sondern befindet sich an einem zentralen Ort, dem Page Object. Die hohe Koppelung der Testfälle mit den Seiten der Anwendung, die Leotta et al. [Leo+13] als Problem genannt haben, kann somit überwunden werden.

2. Es gibt einen einzigen Ort für Elemente und Operationen, die von einer Seite angeboten werden.

   Alle Informationen, die eine Seite der Anwendung betreffen, sind an einem zentralem Ort, dem Page Object gesammelt. Seitenspezifischer Code muss somit nicht mehr in den einzelnen Testfällen dupliziert werden. Die Funktionalitäten der Seite können über das entsprechende Page Object abgerufen werden.

Die genannten Vorteile zeigen, dass Änderungen, die an einer Seite durchgeführt werden, nur Auswirkungen an einer zentralen Stelle haben. Die Wartbarkeit der gesamten Testfälle wird so erhöht. Leotta et al. unterstützen diese These mit einer Fallstudie [Leo+13], in der sie

eine herkömmliche Testsuite mit einer Testsuite, die das Page Object Pattern implementiert, hinsichtlich ihrer Wartbarkeit verglichen haben. Das Ergebnis dieser Studie hat gezeigt, dass die Zeit für die Wartung der Testfälle um ca. 65% reduziert werden konnte. Die Anzahl der anzupassenden Codezeilen reduzierte sich um ca. 87%.

4.3.3.2. Probleme des Page Object Pattern

Den in Kapitel 4.3.3.1 genannten Vorteile stehen allerdings auch Nachteile gegenüber. Wird das Page Object Pattern verwendet, steigt die Komplexität des gesamten Testprojekts. Testfälle können nicht mehr beliebig programmiert werden, sondern sind in den Kontext von Page Objects zu stellen. Das erfordert tiefgründige Designentscheidungen. So ist beispielsweise zu klären, wie über die Page Objects ein generischer Einstieg in die Anwendung angeboten werden kann, oder wie der WebDriver über mehrere Page Objects und Testfälle hinweg verwaltet wird.
Ein weiterer Nachteil begründet sich darin, dass zunächst die Page Objects entwickelt werden müssen, bevor mit dem Erstellen von Testfällen begonnen werden kann. Das Erstellen der Page Objects benötigt bei komplexen Webanwendungen viel Zeit und behindert damit die eigentliche Testfallerstellung.
Verglichen mit einem herkömmlichen Ansatz steigt daher initial der Aufwand, der zum Erstellen eines Testprojektes betrieben werden muss.
Wie in Kapitel 4.3.3.1 bereits erwähnt, haben Leotta et al. [Leo+13] allerdings auch dargestellt, dass sich diese Investition, über längere Zeit gesehen, durchaus lohnen kann.

5. Teilautomatisierte Generierung von Page Objects

Ein großer Teil des in Kapitel 4.3.3.2 angesprochenen initialen Mehraufwands, bei der Verwendung des Page Object Pattern beläuft sich auf die Erstellung der Page Objects. Wie in Listing 4.2 und 4.3 zu sehen ist, handelt es sich bei Page Objects um weniger komplexe Klassen. In der Praxis zeigt sich, dass ein Großteil der Arbeit darin besteht, die verschiedenen Lokatoren der Elemente aus dem Quelltext der Seite zu extrahieren und in die generische Form eines Page Objects zu überführen. Diese Aufgabe kostet zwar viel Zeit, ist allerdings nicht sehr anspruchsvoll. Möchte man den initialen Mehraufwand bei der Verwendung des Page Object Pattern entgegenwirken, liefern die Page Objects somit einen guten Ansatzpunkt. Aufgrund ihrer generischen Natur bieten sie gute Voraussetzungen, um automatisch generiert zu werden.

5.1. Übersicht über die Idee

Selenium, in Verbindung mit dem Page Object Pattern, ist auch ein Teil des Technologiestacks des IT-Dienstleisters (it@M) der Landeshauptstadt München und wird dort zum Testen komplexer Webanwendungen verwendet. Auch it@M hat in Bezug auf die Erstellung von Page Objects die Erfahrung gemacht, dass es sich um eine generische und zeitaufwendige Arbeit handelt. In Zusammenarbeit wurde daher die Idee entwickelt, das Erstellen von Page Objects mit Hilfe einer Softwarelösung zu unterstützen. Anhand des Seitenquelltextes der zu testenden Webanwendung sollen die verschiedenen Elemente des Page Objects identifiziert und zur Generierung der Klassen verwendet werden. Zwei Ansätze wurden dabei diskutiert. Eine vollautomatisierte und eine teilautomatisierte Generierung von Page Objects.
Ein vollautomatischer Ansatz würde beinhalten, dass ohne weiteres Zutun aus dem Seitenquelltext ein vollständiges Page Object generiert wird. Dieser Ansatz hat jedoch mit zahlreichen Problemen zu kämpfen. Oft wird nur ein Bruchteil der Elemente einer Webseite für die Testfälle benötigt. Selenium kann aber prinzipiell jedes Element, dass im Seitenquell-

text bereitgestellt wird, ansprechen. Bei einer vollautomatischen Generierung müssten daher entweder alle Elemente einer Seite übernommen, oder eine definierte Auswahl getroffen werden. Wird eine Auswahl getroffen, besteht das Risiko, dass Elemente ausgelassen werden, die vom Tester möglicherweise benötigt werden. Werden alle Elemente übernommen, werden die Page Objects schnell überfüllt und unübersichtlich. Das Überfüllen der Page Objects geschieht dann auf Kosten ihrer Robustheit. Strukturelle Änderungen in der Website wirken sich auch auf die Lokatoren der Elemente aus. Um die Page Objects stabil zu halten, müssen diese bei Änderungen in der Seitenstruktur berichtigt werden. Es ist daher nicht sinnvoll, Elemente in den Page Objects zu pflegen, die nicht verwendet werden. Unbenutzte Elemente bedeuten entweder zusätzlichen Wartungsaufwand oder veralten unbemerkt.
Ein weiteres Problem des vollautomatischen Ansatzes stellen die Übergänge zwischen den Seiten einer Webanwendung dar. Interaktionen, wie beispielsweise das Betätigen eines Links oder Button, führen oft zum Aufrufen einer neuen Seite der zu testenden Webanwendung. Im weiteren werden diese Übergänge als Transitionen bezeichnet. Diese Transitionen werden auch in den Page Objects abgebildet. Das Page Object gibt dazu das entsprechende Page Object der Zielseite als Rückgabe eines Methodenaufrufs der Ausgangsseite zurück. In der Methode ‘CreatePage.createEntry()‘ im Listing 4.2 ist diese Vorgehen dargestellt. Allein aus dem Seitenquelltext zu ermitteln, welche Seite das Ziel einer Transition ist, erweist sich oft als sehr schwierig bis unmöglich.
Um die Komplexität des Projektes aufgrund der genannten Probleme nicht zu groß werden zu lassen, entschied man sich für eine teilautomatisierte Lösung. Ziel ist es also nicht, ein vollständiges Page Object vollautomatisiert zu generieren, sondern den Entwickler bei der Generierung der Page Objects zu unterstützen. Anhand des Quelltextes sollen dem Entwickler die möglichen Elemente der Seite in einer Vorauswahl bereitgestellt werden. Aus diesen Elementen können dann diejenigen ausgewählt werden, die im späteren Page Object benötigt werden. Auf diese Weise wird ein Überladen verhindert und gleichzeitig sichergestellt, dass die Elemente, die benötigt werden, vorhanden sind. Ob es sich bei einem Element um eine Transition handelt, also ein Element, welches auf eine neue Seite führt, muss nicht mehr automatisch anhand des Quelltextes ermittelt werden, sondern wird vom Entwickler direkt bei der Auswahl der benötigten Elemente mit angegeben. Die so vom Entwickler ausgewählten Informationen können später verwendet werden, um daraus das fertige Page Object zu generieren. Im Rahmen des Projektes ‘SeleniPo‘ soll dieser Ansatz in Zusammenarbeit mit it@M in Form einer Desktopanwendung umgesetzt werden.

5.2. Abgrenzung zu bestehenden Ansätzen

Sowohl für die vollautomatische Generierung von Page Objects, als auch für eine teilautomatisierte Generierung, gibt es bereits mögliche Lösungsansätze. Stocco et al. [Sto+15] beschreiben in einem Paper das von ihnen entwickelte Framework APOGEN, mit deren Hilfe Page Objects vollautomatisch generiert werden können. Die Generierung geht dabei weit über das Anlegen von Elementen hinaus und schließt auch die Funktionalitäten der einzelnen Webseiten in Form von Methoden mit ein. Das Framework analysiert dazu die Struktur der Webanwendung mittels eines Crawlers. Die Informationen, die über den Crawler zusammengetragen wurden, wie beispielsweise das DOM der einzelnen Webseiten, werden anschließend über eine statische Analyse aufbereitet und für die Generierung der Page Objects verwendet. Nach Angaben der Forschungsgruppe sollen ca. 75% des von APOGEN generierten Codes ohne Anpassungen verwendet werden können. Die restlichen 25% benötigen nur kleine Änderungen.
Bei APOGEN handelt es sich jedoch um ein noch sehr junges Projekt. Das entsprechende Paper wurde im Mai 2015 veröffentlicht. APOGEN ist daher eher ein Prototyp, der zwar die Möglichkeiten aufzeigt, die in diesem Bereich gegeben sind, jedoch noch nicht für den produktiven Einsatz in einem großen Unternehmen geeignet ist. Nach eigenen Angaben leidet das Projekt noch unter einigen Einschränkungen. Eine der genannten Einschränkungen ist die Limitierung durch den Crawler. APOGEN kann nur Webseiten in Page Objects umwandeln, die auch durch den Crawler erreicht werden. Für einfache Webanwendungen stellt das kein großes Problem dar, für sehr komplexe Anwendungen mit einer ausgeprägten logischen Validierung allerdings schon. Viele Seiten, die hinter logisch validierten Eingaben liegen, können vom Crawler nicht erreicht werden und stehen somit für die Generierung nicht zur Verfügung.
Neben der vollautomatischen Generierung existieren noch eine Reihe von Open-Source-Framworks, die einen teilautomatisierten Ansatz verfolgen, ähnlich wie es das Projekt SeleniPo erreichen will. Stocco et al. [Sto+15] nennen in ihrem Paper die drei derzeit wichtigsten Vertreter:

- *OHMAP* [vir15]: Bei OHMAP handelt es sich um eine Webseite, die es dem Benutzer erlaubt, HTML-Code in eine Textarea zu kopieren. Aus dem übergebenen HTML-Code generiert das Tool eine einfache Java-Klasse, die für jedes gefundene Input-Feld ein WebElement enthält. Die Variablennamen werden dabei aus den HTML-Attributen gebildet. Als Lokator wird ein einfacher XPath-Ausdruck verwendet.

- *SWD Page Recorder* [Dmy15]: Der SWD Page Recorder ermöglicht es dem Benutzer eine beliebige Webanwendung zu starten und das GUI der Anwendung mit einem ‘click&record‘-Mechanismus zu inspizieren. Nach jedem Klick auf das Interface der Anwendung wird ein Dropdown-Menü angezeigt, in welches manuell ein Name für das ausgewählte Element eingetragen werden kann. Als Lokator wird ein einfacher XPath-Ausdruck generiert. Das so erstellte Modell der Anwendung kann in verschiedene Sprachen exportiert werden, wie beispielsweise Java, C#, Python, Ruby oder Perl. Beim SWD Page Recorder handelt es sich um eine .NET Anwendung.

- *WTF PageObject Utility Chrome Extension* [DL15]: WTF unterstützt den Entwickler beim Erstellen von Page Objects, indem Lokatoren der Form id, name, CSS oder XPath erstellt werden. Der Code wird in Python generiert.

Der Technologiestack von it@M sieht eine Entwicklung der Selenium-Testfälle in Java vor. Als Betriebssystem kommt darüber hinaus Linux zum Einsatz. Zwei der genannten Lösungsansätze scheiden mit dieser Einschränkung für den produktiven Einsatz beim IT-Dienstleister der Landeshauptstadt München aus. Beim SWD Page Recorder handelt es sich um eine .NET Anwendung, die nur schwer unter Linux betrieben werden kann. Die WTF PageObject Utility Chrome Extension kann nur im Python-Umfeld betrieben werden. OHMAP wäre aus technischer Sicht eine mögliche Lösungsalternative. Allerdings sind Komfort und Umfang der Anwendung aus Sicht von it@M nicht ausreichend. Ohne eigene Konfiguration ist es mit OHMAP nur möglich, input-Felder zu erkennen. Darüber hinaus muss der HTML-Quelltext händisch aus der zu testenden Anwendung extrahiert werden.
Sowohl OHMAP, als auch der SWD Page Recorder haben zusätzlich das Problem, dass die erzeugten XPath Ausdrücke oft sehr einfach gewählt werden und damit sehr stark von der Position der Elemente innerhalb der Seite abhängig sind. Die eigentlichen Charakteristika der Elemente werden oft nicht beachtet. Listing 5.1 zeigt einen solche von OHMAP generierten XPath.

Listing 5.1: Einfacher XPath-Lokator des Projektes OHMAP

```
1  public class YourPageObjectName {
2     //...
3     @FindBy(xpath = "/html/body/div/div[1]/div[1]/h1/a[2]")
4     public WebElement followVirtuetechGmbH;
5     //...
6  }
```

Der zu adressierende Link in Listing 5.1 wird alleine über seine Position innerhalb des DOM der Seite bestimmt. Um den Lokator zu zerstören würde es genügen, ein weiteres div-Tag vor dem Link einzufügen. Bezieht man den XPath auf die eigentlichen Charakteristika, wie beispielsweise ein id-Attribut, können sehr viel stabilere Ausdrücke erzeugt werden.
Negativ ist an allen gezeigten Lösungen, dass sie immer nur ein Page Object auf einmal betrachten. Transitionen, also Übergänge zwischen den einzelnen Webseiten der zu testenden Anwendung, werden nicht beachtet. Die dynamische Komponente der Anwendung wird beim Generieren der Page Objects außer Acht gelassen und muss nachträglich händisch hinzugefügt werden.
Mit SeleniPo soll der Versuch unternommen werden, die Schwachstellen der bereits existierenden Lösungsansätzen zu verbessern und eine plattformunabhängige Lösung zu schaffen, die in der IT-Infrastruktur von it@M ausgeführt werden kann.

5.3. SeleniPo - Page Object Generator

Abbildung 5.1 zeigt die Denktopanwendung (Page Object Generator), die im Rahmen des Projektes SeleniPo entwickelt wurde. Mit Hilfe dieser Anwendung können Page Objects teilautomatisiert generiert werden. Es wird dazu die Möglichkeit geboten, einen Browser zu starten und über vorgefertigte Selektoren die Webanwendung nach benötigten Elementen bzw. Transitionen zu durchsuchen. Auf diese Weise kann ein Modell der Anwendung erstellt werden, dass zur Generierung der Page Objects verwendet wird.

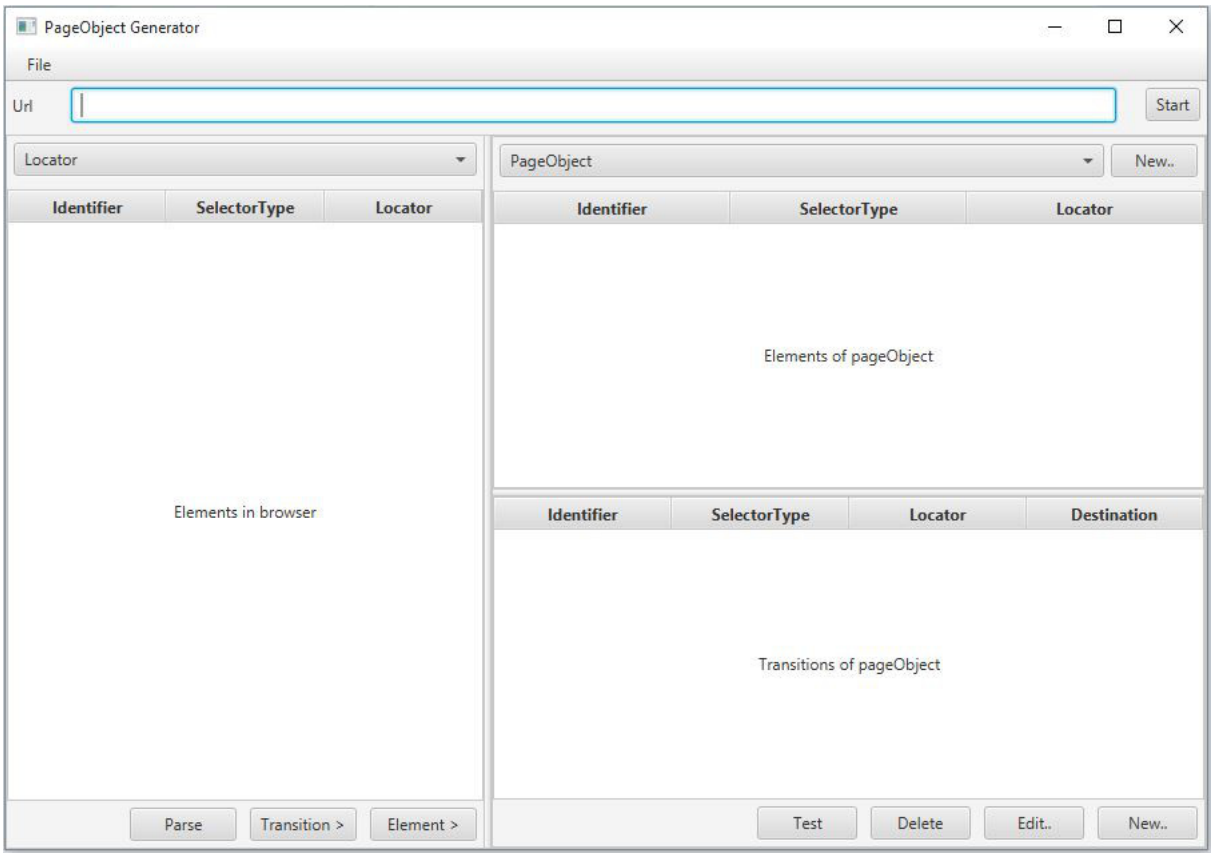


Abbildung 5.1.: SeleniPo - Page Object Generator

Die Benutzeroberfläche des Page Object Generators teilt sich in drei Bereiche:

- Das aktuelle Page Object Modell (Abbildung 5.2)
- Den HTML-Parser (Abbildung 5.3)
- Das Menü (Abbildung 5.4)

Abbildung 5.2 zeigt den Bereich des Generators, mit dem das aktuelle Page Object Modell der zu testenden Anwendung verwaltet werden kann. Mit diesem Teil der Anwendung können neue Page Objects angelegt und bearbeitet werden. Elemente und Transitionen können manuell hinzugefügt, editiert oder gelöscht werden. Darüber hinaus besteht die die Möglichkeit, existierende Elemente und Transitionen auf ihre Richtigkeit zu testen.

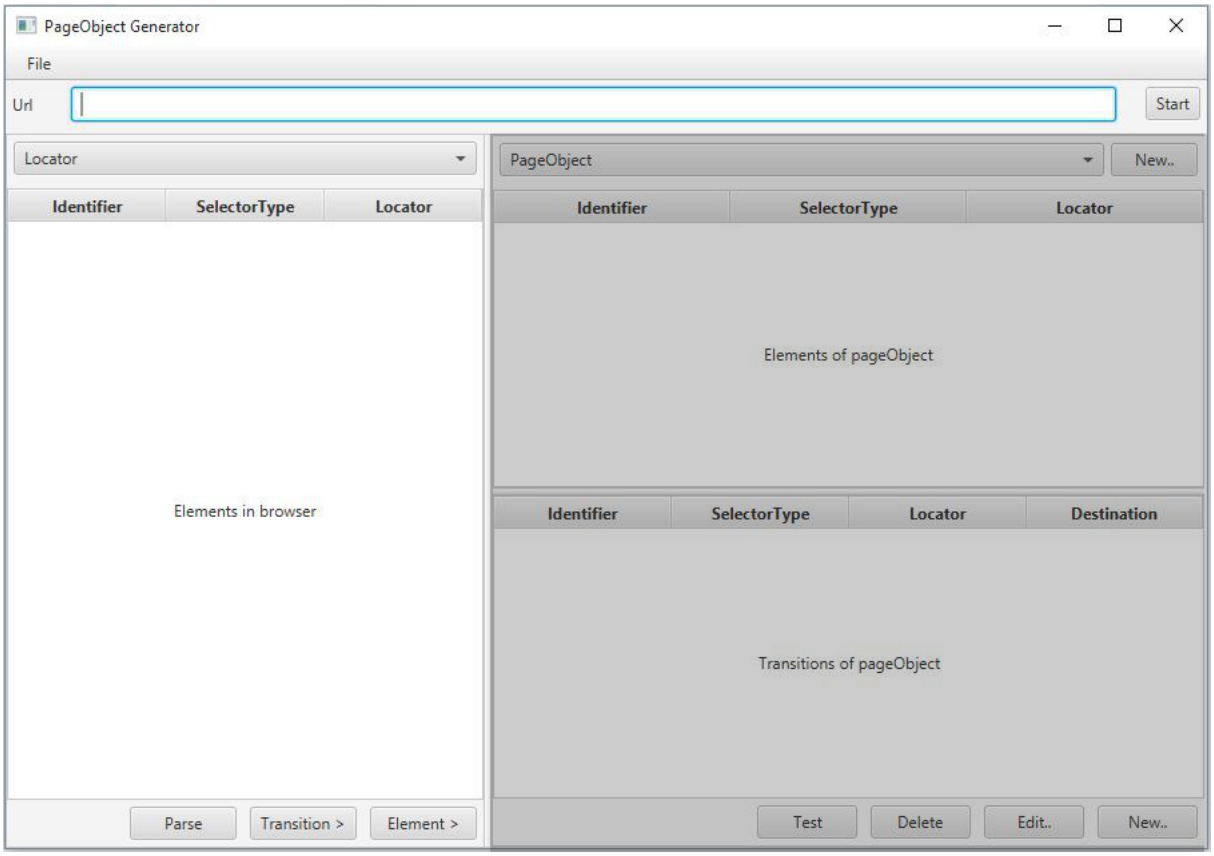


Abbildung 5.2.: SeleniPo - Page Object Generator - Page Object Model

Abbildung 5.3 zeigt den Bereich des Generators, mit dem der Entwickler bei der Erstellung von Elementen und Transitionen im Page Object unterstützt werden kann. Über den Start-Button kann ein Browser gestartet werden. Über das Lokator-Dropdown kann mittels vorgefertigten Selektoren die aktuell im Browser dargestellte Webseite nach Elementen bzw. Transitionen durchsucht werden. Im Page Object benötigte Elemente und Transitionen können dann in das ausgewählte Page Object übernommen werden, um so ein Page Object Modell der zu testenden Webanwendung zu erstellen.

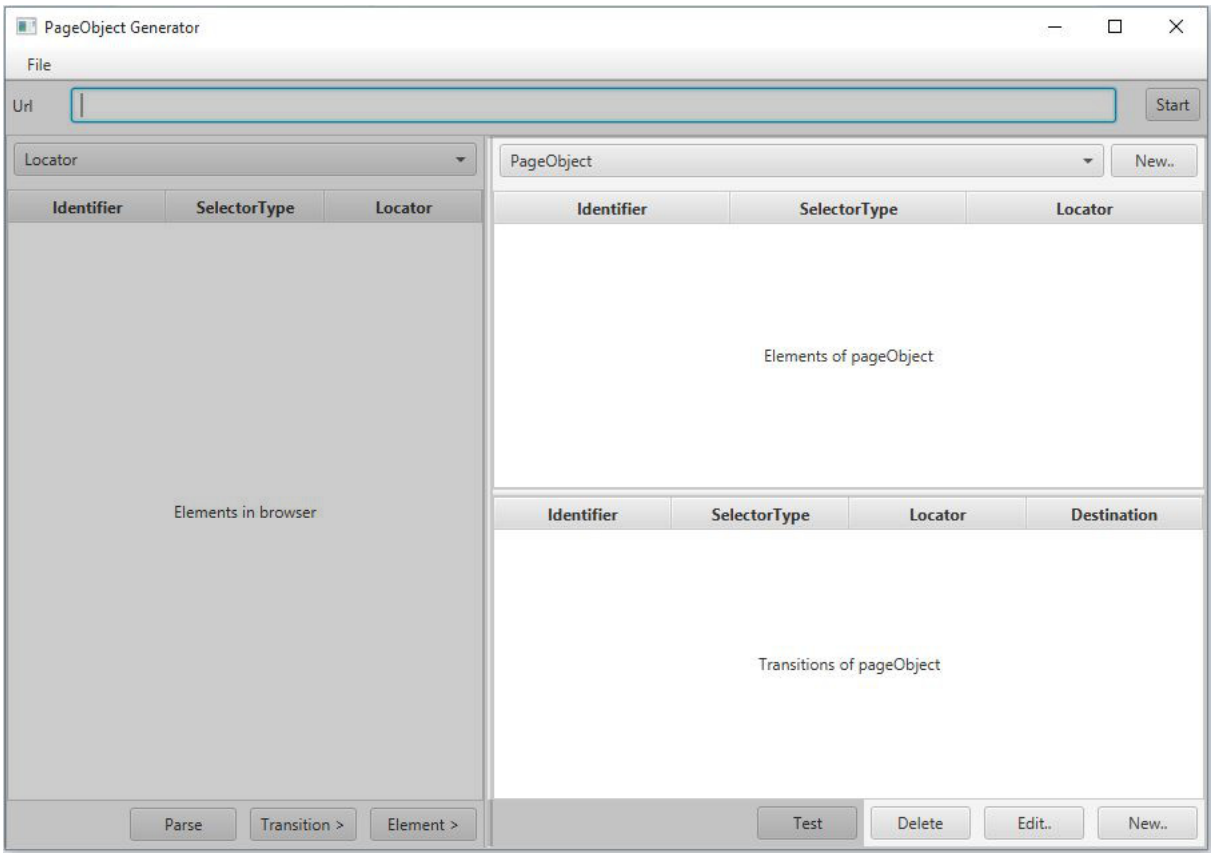


Abbildung 5.3.: SeleniPo - Page Object Generator - HTML Parser

Abbildung 5.4 markiert das Menü des Page Object Generators. Mit Hilfe des Menüs können Zwischenstände des Page Object Modells gespeichert und geladen werden. Über das Menü wird auch die Generierung der Page Objects aus dem aktuell geladenen Modell ausgelöst.

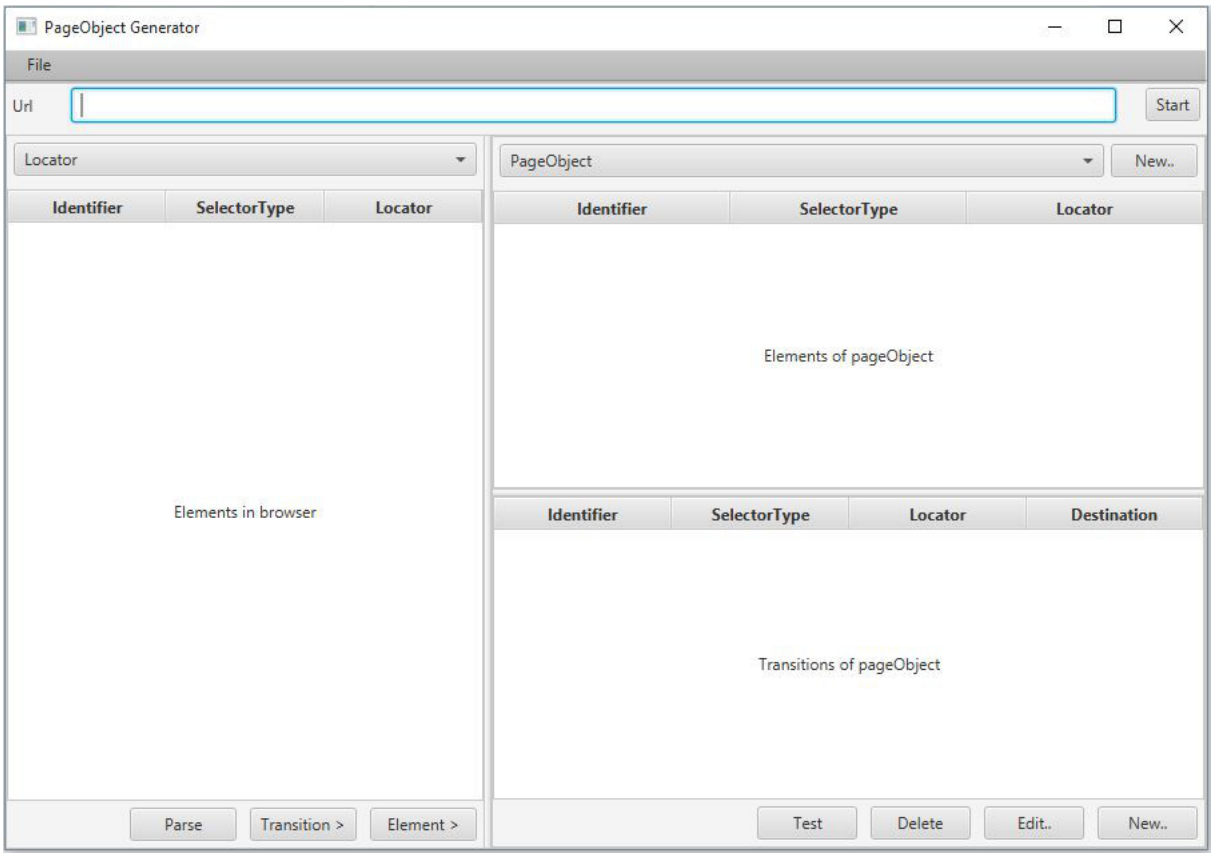


Abbildung 5.4.: SeleniPo - Page Object Generator - Menü

5.3.1. Einordnung des Page Object Generator in den Gesamtkontext

Abbildung 5.5 zeigt die Einordnung des Page Object Generator in die Infrastruktur von it@M. Anhand dieser Abbildung soll gezeigt werden, in welchem Bezug sich der Generator zu einer zu testenden Webanwendung und dem späteren Testprojekt befindet.
Zwei übergeordnete Teilbereiche werden unterschieden. Die virtualisierte Serverumgebung (MIA) und der lokale Rechner eines Entwicklers.
Auf der virtualisierten Serverumgebung werden die entwicklerübergreifenden Infrastrukturkomponenten, wie beispielsweise eine Versionsverwaltung, bereitgestellt. Unter dem Entwicklungsrechner ist der Arbeitsplatz eines einzelnen Projektteilnehmers zu verstehen.
Auf dem Entwicklungsrechner wird die zu testende Webanwendung entwickelt. Zu Test-

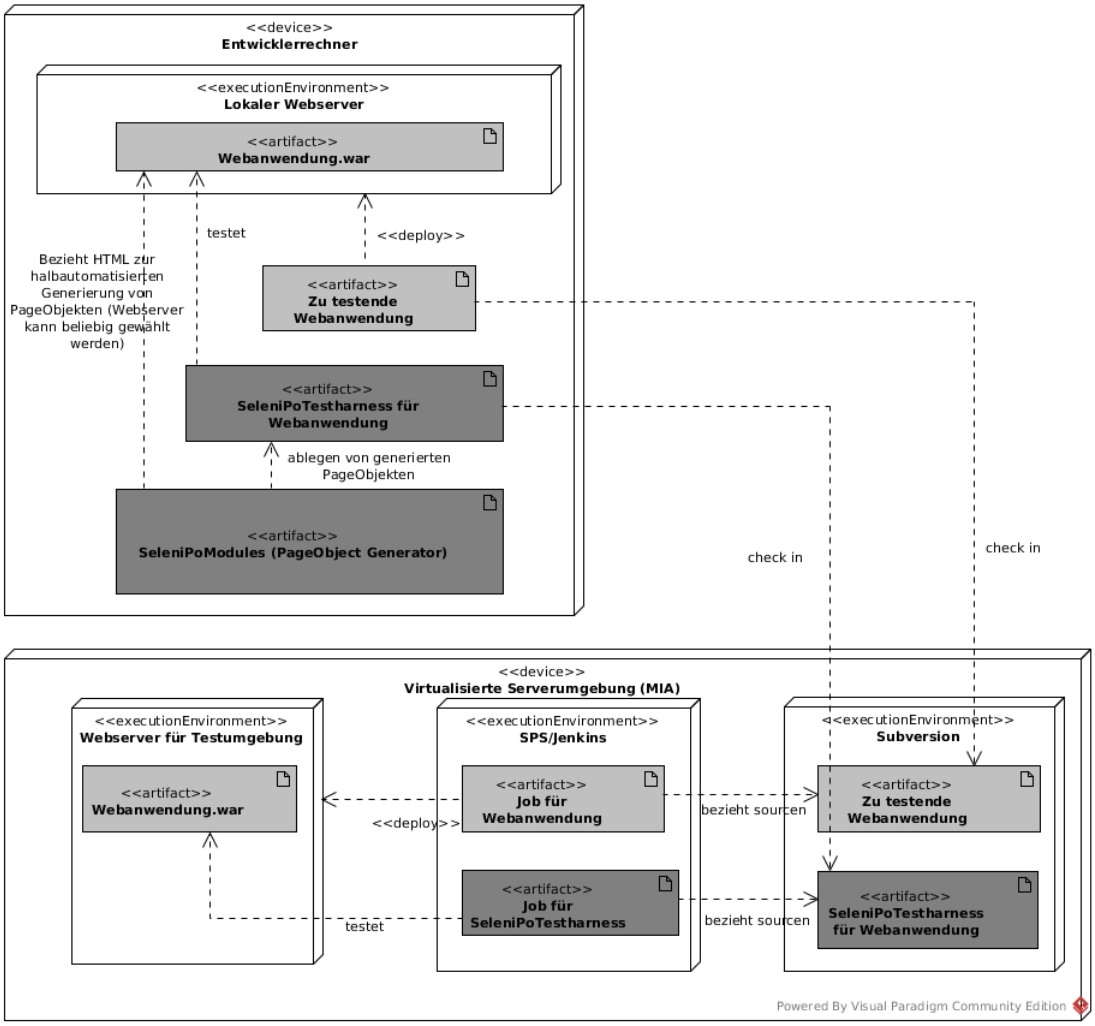


Abbildung 5.5.: Einordnung des Page Object Generator in die Deploymentsicht

zwecken kann diese Anwendung, in ihrem aktuellen Entwicklungsstand, auf einem lokalen Webserver bereitgestellt werden. Der lokale Rechner des Entwicklers ist darüber hinaus auch der Ort, an dem der Page Object Generator eingesetzt wird. Die vom Entwickler lokal bereitgestellte Webanwendung kann verwendet werden, um für die verschiedenen Seiten der zu testenden Anwendung Page Objects mit Hilfe des Generators zu erzeugen. Diese Page Objects werden in ein Test-Projekt abgelegt, in welchem später auch die Selenium-Testfälle entwickelt werden. In Abbildung 5.5 wird dieses Projekt als SeleniPoTestharness bezeichnet und kann vom Entwicklerteam entweder selbst erstellt oder als leeres Quickstart-Projekt vorgefertigt bezogen werden.

Mit Hilfe der Page Objects im Testharness können Testfälle entwickelt werden, die während der Erstellung auf dem lokalen Entwicklungsrechner, gegen die lokal bereitgestellte Weban-

wendung ausgeführt werden.
Über die virtualisierte Serverumgebung werden die lokal erstellten Ergebnisse zusammengeführt. Der Sourcecode der zu testende Webanwendung, sowie des SeleniPoTestharness, wird in einer Versionsverwaltung in der MIA abgelegt. Bei it@M kommt zu diesem Zweck ‘Subversion‘ zum Einsatz.
Über die Versionsverwaltung kann dann ein Integration Server bedient werden, der das Bauen, Bereitstellen und Testen der Webanwendung automatisiert. It@M verwendet hierfür den Continuous Integration Server ‘Jenkins‘ in Verbindung mit Maven und einem Artifactory.
Jenkins bezieht die jeweils aktuellen Sourcen für das Testprojekt und die Webanwendung aus der Versionsverwaltung. So kann regelmäßig eine aktuelle Version der Webanwendung gebaut und auf einem Testsystem in der virtuellen Serverumgebung bereitgestellt werden. Gegen dieses Testsystem können dann mit Hilfe des Jenkins-Servers die im Testharness entwickelten Selenium Testfälle zur Ausführung gebracht werden.

5.3.2. Beispielhafter Ablauf bei der Benutzung des Page Object Generators

Abbildung 5.6 zeigt auf hoher Abstraktionsebene einen Standardablauf bei der Benutzung des Page Object Generators.
Über das Menü (siehe Abbildung 5.4) hat der Benutzer des Page Object Generators die Möglichkeit, einen bereits zuvor angelegten Zwischenstand des Page Object Modells aus einer Save-Datei zu laden. Die bereits angelegten Page Objects werden nach dem Laden im Dropdown-Menü im Bereich des Page Object Modells (siehe Abbildung 5.2) angezeigt. Der Benutzer hat nun die Möglichkeit, mit den bereits vorhandenen Page Objects weiterzuarbeiten oder ein neues Page Object anzulegen. Entscheidet er sich ein neues Page Object anzulegen, wird dies im Dropdown-Menü vorausgewählt angezeigt. Das Page Object kann nun manuell mit Elemente bzw. Transitionen befüllt werden.
Um das Page Object jedoch teilautomatisiert zu befüllen wird ein Webbrowser gestartet. Im Browser muss die Seite der Webanwendung aufgerufen werden, die dem aktuell ausgewählten Page Object entspricht. Mit dem Dropdown des HTML-Parser (siehe Abbildung 5.3) wird die ausgewählte Webseite nach Elementen bzw. Transitionen durchsucht. Passende Ergebnisse können dann in das ausgewählte Page Object übernommen und dort bei Bedarf noch einmal überarbeitet werden. Der neu generierte Zwischenstand wird wiederum über das Menü gespeichert.
Um die Page Objects letztendlich als Code aus dem Modell zu erzeugen, kann über das Menü

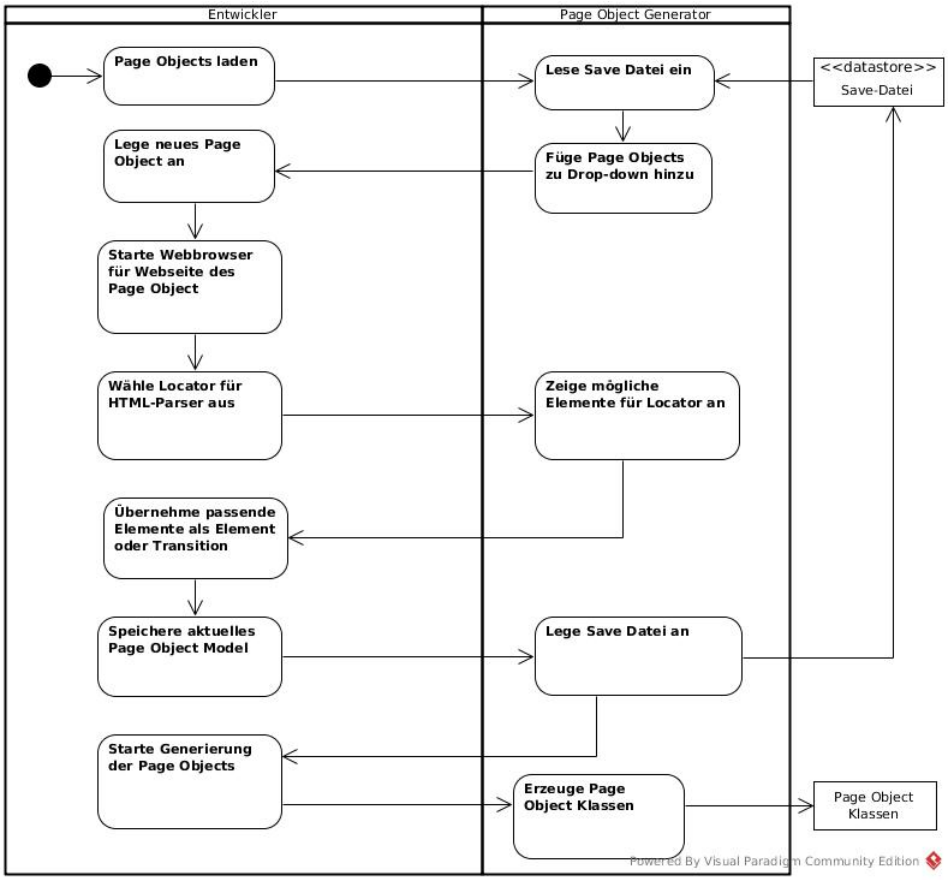


Abbildung 5.6.: Beispielhafter Ablauf bei der Benutzung des Page Object Generator

die Generierung gestartet werden. Bei richtiger Konfiguration des Page Object Generators muss dazu lediglich das Rootverzeichnis des entsprechenden Testprojektes als Zielort der Generierung gewählt werden.

5.3.3. Anwendungsfälle des Page Object Generator

Die konkreten Anwendungsfälle des Page Object Generators sind in Abbildung 5.7 dargestellt.
Eine detailierte Ausformulierung der Anwendungsfälle befindet sich im Anhang A.1

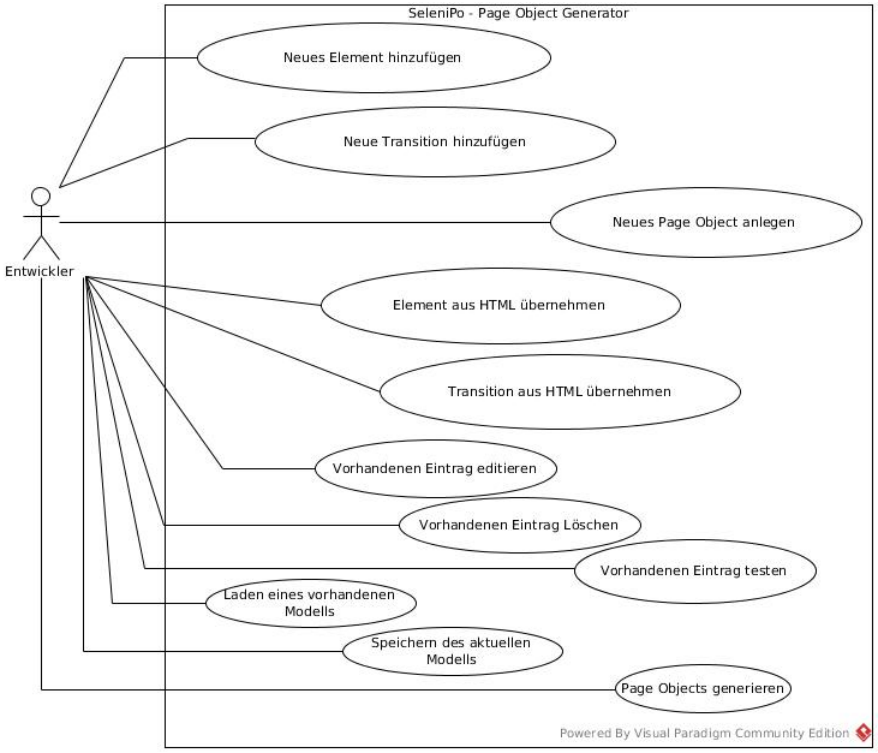


Abbildung 5.7.: Anwendungsfälle des Page Object Generators

5.3.4. Aufbau und technische Aspekte der Anwendung

In seiner internen Struktur ist der Page Object Generator in vier Module aufgeteilt, die unterschiedliche Aufgaben übernehmen. Diese Module sind auf Projektebene in einem übergeordneten Modul mit dem Namen 'SeleniPoModules' zusammengefasst. Abbildung 5.8 zeigt die verschiedenen Module und deren Abhängigkeiten zueinander. Neben den Modulen des Page Object Generators zeigt Abbildung 5.8 zusätzlich den SeleniPoTestharness, der dazu verwendet werden kann, die vom Page Object Generator erzeugten Klassen in einen ausführbaren Kontext zu stellen.
Im Folgenden wird auf die Aufgaben der einzelnen Module kurz näher eingegangen.

5.3.4.1. SeleniPoEditor

Das zentrale Modul des Page Object Generators ist der SeleniPoEditor. Ausgehend von diesem Modul werden die übrigen Module verwendet, um die in Kapitel 5.3.3 vorgestellten Anwendungsfälle zu verwirklichen. Das Modul SeleniPoEditor stellt dazu die grafische Benutzeroberfläche (GUI) der Anwendung bereit. Als Technologie wird dazu JavaFX [Ora15] verwendet, das mit der Java Version 8 Einzug in den Oracle JDK gefunden hat.

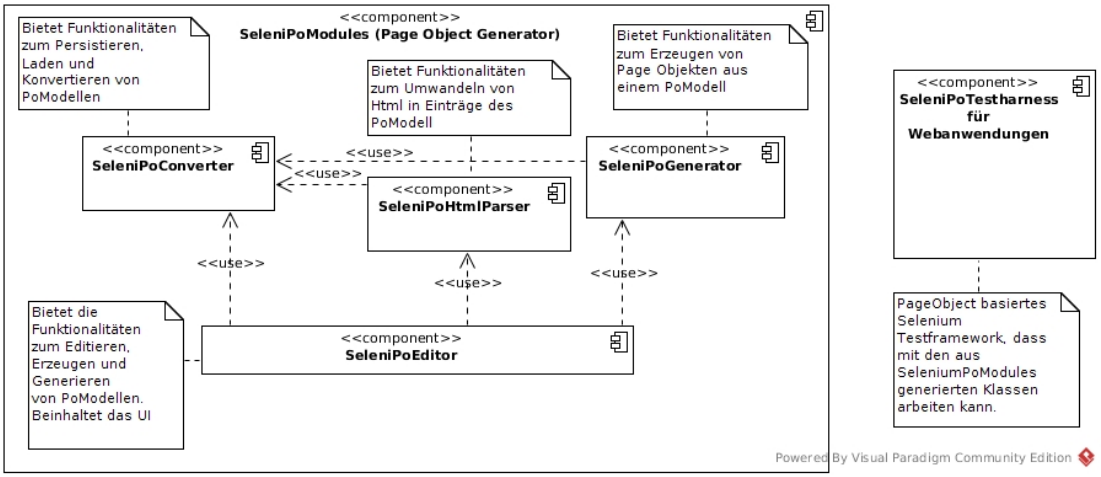


Abbildung 5.8.: Module des Page Object Generators

Um eine hohe Wartbarkeit dieser Komponente zu gewährleisten, wurde die GUI nach dem Model-View-Controller Prinzip verwirklicht. Das Modell besteht aus einem anwendungsspezifischem Java Objekt, welches eine Liste von Page Objekten mit deren zugehörigen Attributen darstellt. Für die View wird die XML-basierte Sprache FXML verwendet, um losgelöst von der Applikationslogik die Struktur der Benutzeroberfläche zu beschreiben. Der Controller besteht aus Java Klassen, mit deren Hilfe das Verhalten der GUI bei Benutzerinteraktion beschrieben wird. Komplexere Logik, wie beispielsweise das Generieren der fertigen Page Object Klassen wird mittels Services durch die übrigen Modulen bereitgestellt.
Das Verhalten der GUI wird über einen Zustandsautomaten gesteuert. Interaktionen mit der Oberfläche beeinflussen den Zustand, in dem sich die Anwendung befindet. Je nach Zustand variieren die Aktionen, die von den verschiedenen Buttons der Anwendung ausgelöst werden. Mit Hilfe dieses Vorgehens kann beispielsweise beim Editieren nach der Auswahl eines Elements, ein anderer Dialog angezeigt werden, als nach der Auswahl einer Transition. Die verschiedenen Zustände der GUI, sowie die Events, die in diesen Zuständen verarbeitet werden, sind im Anhang A.2 über einen erweiterten endlichen Automaten dargestellt.

5.3.4.2. SeleniPoConverter

Das Modul SeleniPoConverter dient der Verwaltung des internen Modells des Page Object Generators. In dieser Komponente wird einerseits das Modell der Anwendung definiert, andererseits werden Services für andere Module bereitgestellt, um mit diesem Modell zu arbeiten. Abbildung 5.9 zeigt die Interface-Stuktur, welche das Modell abbildet.
Kern des Modells ist das Interface PoModel, welches eine Liste von PoGenerics beinhaltet.

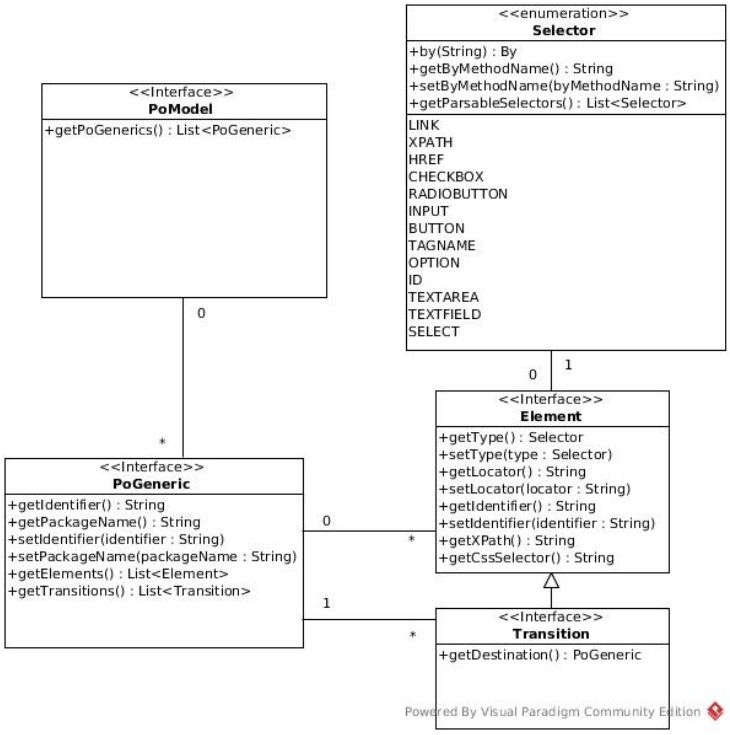


Abbildung 5.9.: Vereinfachte Struktur des internen Modells des Page Object Generators

Ein PoGeneric repräsentiert die Informationen, die benötigt werden, um eine einzelne Page Object Klasse zu erzeugen. Dementsprechend vereint dieses Interface eine Menge an Elementen und Transitionen. Ein Element steht dabei für eine beliebige Komponente einer Webseite, wie beispielsweise ein Eingabefeld. Transitionen sind von Elementen abgeleitet. Es handelt sich also um eine speziellere Form von Elementen. Mit Hilfe von Transitionen ist es möglich, auch die dynamische Komponente einer Webseite in Form von Seitenübergängen abzubilden. Als Transition werden all die Komponenten einer Webseite bezeichnet, die einen Übergang auf ein neues Page Object auslösen. Im Gegensatz zu Elementen bieten Transitionen daher noch ein Page Object Ziel mit an. Über die Methode ‘getDestination()‘ kann dieses Page Object Ziel erfragt wrerden
Sowohl Elemente als auch Transitionen werden über eine Selektor-Enumeration genauer definiert. Der Selektor gibt für ein Element im Modell an, welche Suchstrategie beim Auflösen des Lokators gegen die Webseite verwendet werden soll. Die verfügbaren Strategien werden durch die verschiedenen Aufzählungstypen der Enumeration festgelegt.
Die Kerninformation zum Adressieren eines Elements auf der Webseite bildet der Lokator. Vom Generator wird diese Variable mit einem Wert befüllt, der charakteristisch für das zu adressierende Element ist. Je nach Element handelt es sich hierbei oft um ein HTML-

Attribut, wie beispielsweise id- oder value-Attribute des entsprechenden HTML-Tags.
In Verbindung mit dem ausgewählten Selektor kann über die Klasse ByFactory aus dem Lokator ein repräsentativer XPath-Ausdruck bzw. CSS-Selektor für das Element erzeugt werden. Im Kapitel 5.3.4.5 wird darauf näher eingegangen.
Abbildung 5.9 zeigt lediglich die Interface-Struktur des zugrunde liegenden Modells. Der Page Object Generator kennt zwei konkrete Implementierungen. Das vollständige Modell ist in Anhang A.3 abgebildet.
Zwei verschiedene Implementierungen sind notwendig, da JavaFX und XStream [Joe15], welches für die Persistierung in XML verwendet wird, unterschiedliche Anforderungen an das Modell stellen. JavaFX benötigt spezielle Datentypen. Anstelle einer gewöhnlichen List wird beispielsweise eine ObservableList verwendet, um die View der GUI automatisch mit dem Modell synchron zu halten. Diese Datentypen können von XStream aufgrund eines fehlenden argumentlosen Konstruktors jedoch nicht mehr deserialisiert werden. Dementsprechend werden vom Converter eine Reihe von Services bereitgestellt, die das Wandeln des Modells in zwei Implementierungen ermöglicht. Eine Implementierung kann verwendet werden, um das Modell zu persistieren und wieder zu deserialisieren. Die Zweite Implementierung wird verwendet um im Umfeld von JavaFX zu arbeiten.
Abbildung 5.10 zeigt alle vom Converter bereitgestellten Services.

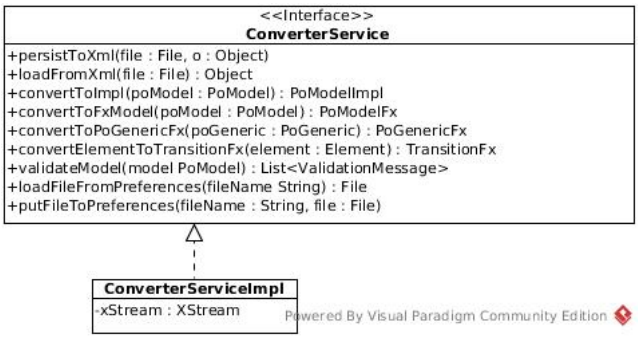


Abbildung 5.10.: Services, die von SeleniPoConverter bereitgestellt werden

Neben den Services zum Wandeln des Modells wird in diesem Modul auch die Funktionalität zum Speichern und Laden, sowie zur fachlichen Validierung bereitgestellt. Zusätzlich wird die Möglichkeit geboten, benutzerspezifische Informationen, wie beispielsweise der Pfad zur zuletzt verwendeten Save-Datei, in den Properties des Anwenders abzulegen.

5.3.4.3. SeleniPoHtmlParser

Das SeleniPoHtmlParser Modul beinhaltet die Funktionalität zum teilautomatisierten befüllen der Page Objects mit Elementen bzw. Transitionen. Das Modul stellt dazu eine Methode bereit, die es ermöglicht HTML-Quelltext anhand von einem übergebenen Selektor auszuwerten. Abbildung 5.11 zeigt das Klassendiagramm für diesen Service.

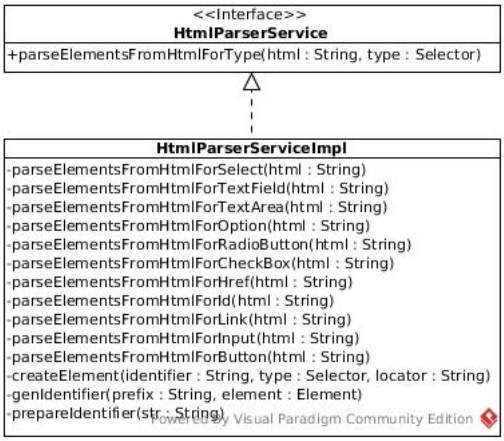


Abbildung 5.11.: Services, die von SeleniPoHtmlParser bereitgestellt werden

Der benötigte HTML-Quelltext wird direkt aus einem Webbrowser bezogen, der über die Benutzeroberfläche der Anwendung gestartet werden kann. Das Parsing des Quelltextes übernimmt die freie Bibliothek jsoup [Hed15]. Für jeden in der Selektor-Enumeration (siehe Kapitel 5.3.4.2) definierten Aufzählungstypen wurde dazu eine eigene Strategie implementiert. Die für die jeweiligen Selektoren implementierte Filterstrategie orientiert sich an der Strategie, die später in den Testfällen verwendet wird, um die Elemente auf der Webseite zu adressieren.
Listing 5.2 zeigt beispielhaft die Implementierung für den Aufzählungstypen LINK, der Links auswählt, welche eines der HTML-Attribute id, text oder title besitzen:

Listing 5.2: Parser für den Aufzählungstypen LINK

```
1   /**
2    * Sucht nach Links fuer die vorhanden ist: id or text or title
3    *
4    * @param html Quelltext der zu untersuchenden Seite
5    * @return PoGeneric - neues Page Object mit den gefundenen Elementen
6    */
7   private PoGeneric parseElementsFromHtmlForLink(String html) {
```

```
8       final String PREFIX = "a";
9       PoGeneric poGeneric = new PoGenericImpl();
10      Document doc = Jsoup.parse(html);
11      Elements elements = doc.select("a");
12      for (Element element : elements) {
13         if (element.hasAttr("id")) {
14            de.muenchen.selenipo.Element createdElement = createElement(
15                  genIdentefier(PREFIX, element), Selector.LINK,
16                  element.attr("id"));
17            poGeneric.getElements().add(createdElement);
18         }
19         else if (element.hasText()) {
20            de.muenchen.selenipo.Element createdElement = createElement(
21                  genIdentefier(PREFIX, element), Selector.LINK,
22                  element.text());
23            poGeneric.getElements().add(createdElement);
24         }
25         else if (element.hasAttr("title")) {
26            de.muenchen.selenipo.Element createdElement = createElement(
27                  genIdentefier(PREFIX, element), Selector.LINK,
28                  element.attr("title"));
29            poGeneric.getElements().add(createdElement);
30         }
31      }
32      return poGeneric;
33   }
34  }
```

Dieses Vorgehen hat den Nachteil, dass die Möglichkeit besteht, Elemente zu verwerfen, die vom Benutzer für den ausgewählten Filter zwar erwartet werden, den implementierten Filterkriterien jedoch nicht entsprechen. Im Beispiel aus Listing 5.2 wären das alle Links, für die weder das Attribut id, text oder title gesetzt ist.
Durch dieses Vorgehen ist allerdings sichergestellt, dass für einen untersuchten Selektor nur Elemente zur Auswahl gestellt werden, die in den späteren Testfällen auch aufgelöst werden können. So wird verhindert, dass über die Page Objects Elemente bereitgestellt werden, die später in den Testfällen zu Fehlern führen.
Die Palette der vorgefertigten Filter deckt einen großen Teil der in HTML vorhandenen

Elemente bereits ab. Sollte es dennoch vorkommen, dass Elemente über die existierenden Selektoren nicht erreicht werden, besteht die Möglichkeit mit Hilfe des Selektors XPATH, Elemente über einen eigenen XPath-Ausdruck anzusprechen.

5.3.4.4. SeleniPoGenerator

Das Modul SeleniPoGenerator ermöglicht es, aus einem Modell des SeleniPoConverters, Page Object Klassen zu generieren. Abbildung 5.12 zeigt die Services, die von diesem Modul angeboten werden.

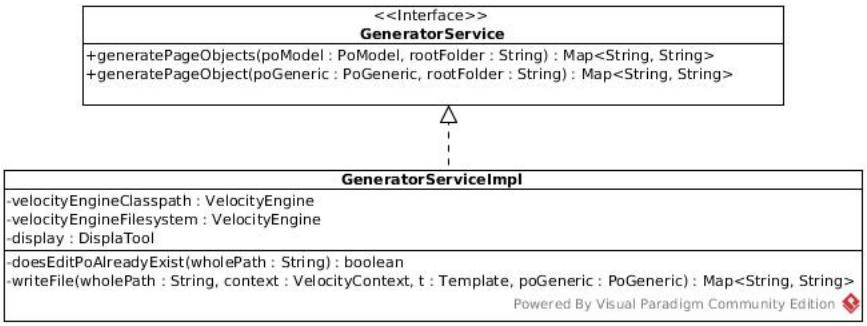


Abbildung 5.12.: Services die von SeleniPoHtmlParser bereitgestellt werden

Für die Generierung des Codes aus dem Modell der Anwendung wird die Template-Engine Velocity [Apa15] verwendet. Mit Hilfe von Velocity kann ein beliebiges Modell der Anwendung über vordefinierte Templates in die gewünschten Page Object Klassen gewandelt werden. Diese Templates können vom Benutzer des Page Object Generators beliebig editiert werden. So ist es möglich, die im Modell der Anwendung hinterlegten Informationen in jede, vom Anwender gewünschte Form aufzubereiten. Anhang A.4 zeigt als Beispiel den Templatevorschlag, der im Rahmen dieser Arbeit im Page Object Generator verwendet wird.
Für jedes Page Object im Modell werden jeweils zwei Klassen erzeugt. Eine Klasse, welche die gesamte generierte Logik enthält, sowie eine weitere Klasse, welche abgesehen von einem Konstruktor leer ist und von dieser Klasse ableitet. Die komplexe Klasse wird im folgenden als ‘PageObject_Generated‘ bezeichnet. Das zugehörige Template befindet sich in Listing A.1. Die bis auf den Konstruktor leere Klasse wird im folgenden als ‘PageObject_Dynamisch‘ bezeichnet und ist im Template, in Listing A.2 dargestellt.
Abbildung 5.13 zeigt die Abhängigkeit zwischen den beiden zu generierenden Klassen.
Mit Hilfe der Aufteilung eines Page Objects in zwei Klassen soll verhindert werden, dass bei mehrmaligem Generieren des gleichen Page Objects Änderungen, die vom Testentwickler im Code vorgenommen wurden, überschrieben werden. Als Konvention gilt daher, dass

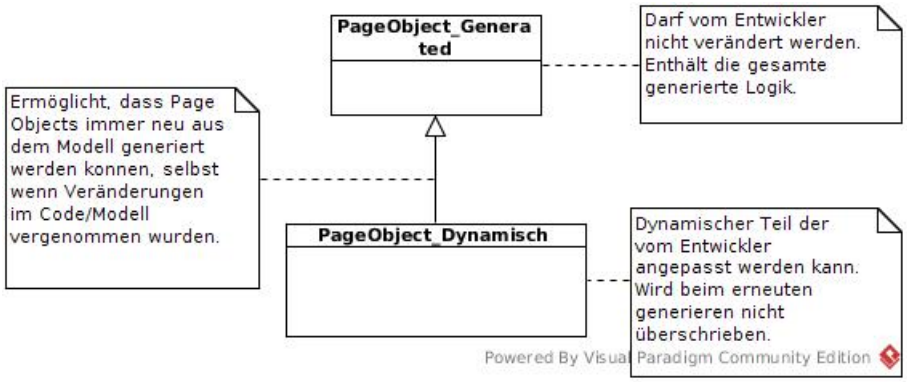


Abbildung 5.13.: Abhängigkeit zwischen dem generierten und dem dynamischen Teil eines Page Objects

Änderungen durch den Entwickler nur im PageObject_Dynamisch vorgenommen werden dürfen. Der generierte Teil des Page Objects darf vom Entwickler nicht verändert werden. Bei einem erneuten generieren der Page Objects werden lediglich die PageObject_Generated überschrieben, der dynamische Teil bleibt unangetastet. Dadurch ist sichergestellt, dass Änderungen, die im dynamischen Teil geschehen nicht überschrieben werden, jedoch Anpassungen die über den Page Object Generator vorgenommen wurden, Einzug in den generierten Teil finden können. Page Objects können so über die gesamte Projektlaufzeit iterativ im Page Object Generator aufgebaut werden und müssen nicht bereits zu Beginn im finalen Zustand modelliert werden.
Der Ort an dem die fertigen Page Objects abgelegt werden, ist über eine Konfigurationsdatei und eine Variable in den Page Objects einstellbar. Abbildung 5.14 stellt grafisch dar, wie die endgültige Paket-Struktur eines Page Objects gebildet wird. In der Grundkonfigurati-

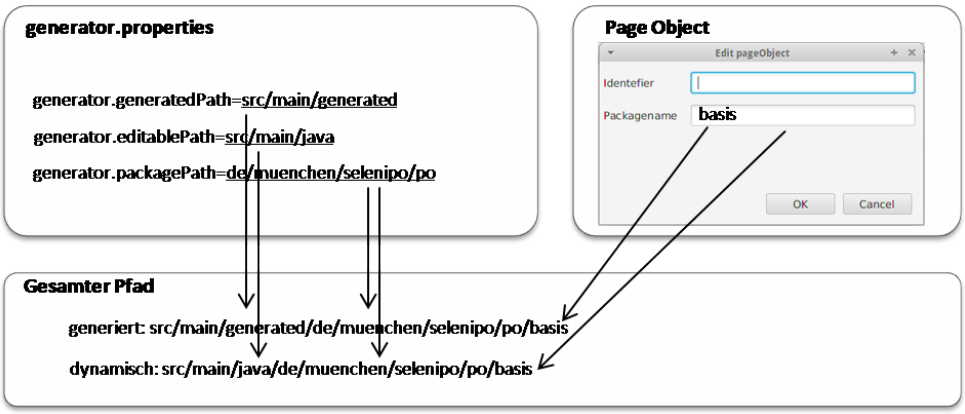


Abbildung 5.14.: Erzeugen der Paket-Struktur eines Page Object

on wird der generierte Teil eines Page Objects unterhalb des Ordners ‘src.main.generated‘ abgelegt. Der dynamische Teil befindet unterhalb von ‘src.main.java‘. Diese Strukturierung bildet die getroffene Konvention ab, dass bei Verwendung der Page Objects der dynamische teil als Instanz in den Testfällen verwendet wird, der generierte Teil aber nicht editiert werden darf. Ausgehend von diesen Verzeichnissen ist es über die Konfiguration möglich, den Pfad weiter zu verfeinern. So kann konfiguriert werden, dass der generierte Teil eines Page Objects immer im Paket ‘de.muenchen.selenipo.po‘ abgelegt wird, der dynamische im Paket ‘de.muenchen.selenipo.po‘.
Während der Erstellung im Page Object Generator kann auf Ebene der Page Objects eine weitere Verfeinerung der Struktur vorgenommen werden. Ausgehend von den global konfigurierten Ziel-Paketen ist es so möglich, jedes Page Object in ein beliebiges Paket abzulegen. Bei richtiger Konfiguration kann dann für die Generierung der Klassen, einfach das Rootverzeichnis des Testprojektes ausgewählt werden.

5.3.4.5. SeleniPoTestharness

Im Rahmen dieser Arbeit wurde auch ein Testprojekt entwickelt, welches mit den Page Objects, die über den Page Object Generator erzeugt wurden, arbeiten kann. Das Testprojekt wird im folgenden als Testharness bezeichnet.
Damit Testharness und Page Object Generator zusammenarbeiten können, muss die Struktur der Templates des Generators mit der Struktur des Testharness zusammenpassen. Die Struktur der Templates kann vom Benutzer des Page Object Generators jederzeit angepasst werden. Prinzipiell kann dadurch jede Struktur eines Testharness unterstützt werden.
Um mit den Templates, die im Rahmen dieser Arbeit entwickelt wurden, zusammenzuarbeiten, muss der Testharness allerdings einige Anforderungen erfüllen. Abbildung 5.15 zeigt die Page Object Struktur, die sowohl in den Templates als auch im Testharness verwendet wird.

Die Wurzel der Vererbungshierarchie eines Page Objects im Testharness bildet die Klasse PageObject. Die Klasse PageObject beinhaltet den Selenium WebDriver. Neben dem WebDriver enthält die Klasse PageObject noch eine zusätzliche Kernkomponente, die Klasse ByFactory. Diese Klasse zählt nicht zum Standard bei der Verwendung des Page Object Pattern sondern ist spezifisch für die in dieser Arbeit verwirklichte Implementierung eines Testharness. Die Helferklasse ByFactory ermöglicht es in einfacher Art, komplexe ‘org.openqa.selenium.By‘-Ausdrücke zu erzeugen. Die erzeugten Ausdrücke werden verwendet, um über den WebDriver Elemente auf der Webseite zu lokalisieren. Als Übergabe erhält die Factory dazu einen einfachen Lokator-String, der beispielsweise dem value- oder

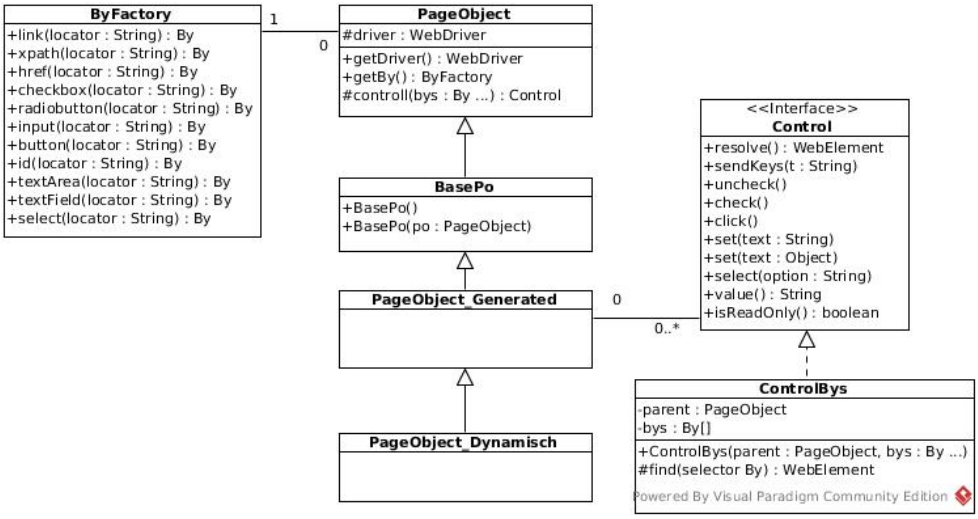


Abbildung 5.15.: Page Object Struktur des Testharness

id-Attribut des gesuchten HTML-Tags entspricht. Die Klasse ByFactory ist mit dem Page Object Generator abgestimmt. Für jeden Selektor des Generators gibt es eine entsprechende Factory-Methode. Die vom Generator erzeugten Lokatoren sind so gewählt, dass sie von der ByFactory interpretiert werden können. Damit ist sichergestellt, dass ein vom Generator erzeugter Lokator von der entsprechenden Factory-Methode aufgelöst werden kann.
Die Verwendung der Factory-Klasse bringt den Vorteil, dass komplexe XPath Ausdrücke einfach erzeugt sowie gekapselt und lesbar in den Page Objects abgelegt werden können. Das Modell des Generators bietet auch die Möglichkeit die bereits aufgelösten Ausdrücke, ohne die zusätzliche Verwendung der ByFactory, zu erhalten. Mit den Methoden ‘Element.getXPath()‘ bzw. ‘Element.getCssSelector()‘ können je nach verwendetem Selektor, der XPath bzw. der CssSelector als String erfragt werden.
Die Klasse BasePo ist das nächste Kind in der Vererbungshierarchie. Als einzige Klasse bietet BasePo einen parameterlosen Konstruktor und bildet somit, bei der Verwendung der Page Objekts, den Einstiegspunkt in einen Testfall. Bei der konstruktorlosen Initialisierung des BasePo wird ein neuer WebDriver erzeugt, der auf einer über die Konfiguration hinterlegbaren Adresse startet. Alle weiteren Page Objects erhalten bei der Instanziierung als Übergabe ein bereits vorhandenes Page Object, aus dem der WebDriver übernommen werden kann. Damit kann der Zustand des WebDrivers innerhalb eines Testfalls von Page Object zu Page Object übergeben werden.
Ausgehend von der Klasse BasePo erben die vom Page Object Generator erzeugten Klassen, PageObject_Generated und PageObject_Dynamisch (siehe Kapitel 5.3.4.4). Die verschiedenen Elemente der Webseite werden im Page Object als Implementierung des Interfaces

Control angeboten. Die Klasse ControlBys implementiert dieses Interface und kapselt ein Selenium WebElement. Als Referenz gegen die Webseite wird bei der Instanziierung ein By-Ausdruck übergeben, der mittels ByFactory erzeugt werden kann. Für die Interaktion mit dem referenzierten Element bietet die Klasse eine Vielzahl von Methoden an. Eine davon ist die Methode ‘Control.resolve()‘, welche das gekapselte WebElement zurückliefert.
Die Kapselung eines WebElements in die Klasse ControlBys hat den Vorteil, dass WebElemente in einem PageObject als globale Variablen geführt werden können. Die Klasse ControlBys verhindert, dass ein WebElement sofort bei der Instanziierung eines PageObject gegen die geladene Webseite im Webdriver aufgelöst wird. Erst beim Aufruf einer Methode wird der übergebene By-Ausdruck verwendet, um das eigentliche WebElement über die WebDriver-Methode ‘findElement()‘ zu erhalten. So wird verhindert, dass ein Page Object bereits bei der Instanziierung versucht, Elemente auf der Webseite abzufragen, die möglicherweise noch gar nicht geladen sind oder erst durch vorangegangene Interaktion erreichbar gemacht werden müssen.
Zum einfachen Erzeugen von Controls bietet die Klasse PageObject die statische Methode ‘control(bys : By ...)‘ an, die auch in den Templates verwendet wird.

5.4. Praxistest

Um im Praxisbetrieb erste Erfahrungen mit dem Page Object Generator zu sammeln, wurde eine frühe Version des Generators dazu verwendet, Page Objects für ein bereits etabliertes Projekt mit geringer Testabdeckung zu erzeugen.
Für das System zur Verwaltung von Schulversäumnissen in Schulen der Stadt München sollten Systemtests mit Hilfe des Selenium WebDrivers erstellt werden. Mit Hilfe des Page Object Generators wurden die benötigten Page Objects erzeugt. Der Fokus im ersten Praxistest wurde darauf gelegt, Erfahrungen zu sammeln, wie intuitiv die Bedienung des Page Object Generators für neue Anwender nach einer kurzen Einweisung ist. Die Erkenntnisse, die dabei gesammelt wurden, lieferten maßgebliche Hinweise, die vor allem zu großen Verbesserungen im Bereich der Benutzerfreundlichkeit führten.
Wichtige Bereiche der Anwendung sind aufgrund des Feedbacks nun über Tastatureingaben zu steuern und nicht mehr nur exklusiv über die in der GUI angebotenen Interaktionskomponenten.
Darüber hinaus hat sich gezeigt, dass in der Regel eine Vielzahl von Elementen und Transitionen für ein Page Object auf einmal übernommen werden können und nicht, wie zuvor angenommen, immer nur einzelne Einträge. Die Übernahme von Elementen und Transitionen

ist daher nicht mehr nur auf einzelne Ergebnisse des Parsers beschränkt. Per Mehrfachauswahl können nun auch eine Vielzahl von Treffern in einem Schritt übernommen werden.
Die Anwendung in der Praxis hat auch gezeigt, dass das Laden, Speichern und Generieren von Page Objects Aktionen sind, die häufiger ausgeführt werden, als zu Beginn erwartet. Aufgrund dieser Erkenntnis wurde die Anwendung dahingehend verbessert, dass sie sich den letzten, vom Benutzer für die jeweilige Aktion ausgewählten Pfad, merkt. Lange Navigationswege bei der Auswahl der Zielordner können so vermieden werden.
Die genannten Veränderungen sind nur einige von einer Vielzahl von Verbesserungen, die durch den ersten Praxistest des Page Object Generators vorgenommen werden konnten.
Mit der überarbeiteten Version wurde ein zweiter Praxistest durchgeführt. In diesem Testbetrieb sollten für die Anwendung zum Erstellen der Jahresstatistiken der Kindertagesstätten der Stadt München Page Objects erzeugt werden. Die Verbesserungen aus dem ersten Praxistest wurden gut angenommen. Die Bereiche die bei der Bedienung zuvor noch Probleme verursachten, zeigten nun intuitive Bedienbarkeit. Der zweite Praxistest zeigte vor allem Bereiche im Page Object Generator auf, die im Hinblick auf eine höhere Flexibilität verbessert werden könnten.
Für die Generierung der eindeutigen Namen von Elementen und Transitionen werden in einem fest vorgegebenem Algorithmus die HTML-Attribute des jeweiligen HTML-Tags ausgewertet. Das gewählte Vorgehen liefert nicht immer sinnvolle Namen. In ungünstigen Konstellationen kann es vorkommen, dass der gleiche Wert mehrmals erzeugt wird, was zu unnötigem Mehraufwand bei der Überarbeitung der Namen führt.
Darüber hinaus hat sich gezeigt, dass die Aufteilung der Page Objects in einen generierten und einen dynamischen Teil, wie in Kapitel 5.3.4.4 beschrieben, nicht von allen Testanwendern intuitiv angenommen wird. Auch wenn diese Aufteilung als sinnvoll erachtet wird, bietet es sich an dem Benutzer die Entscheidungsfreiheit zu überlassen, die Generierung des dynamischen Teils zu verhindern.
Auch bei der Erzeugung der Paket-Struktur, wie in Abbildung 5.14 gezeigt, wurde eine zu eingeschränkte Konfigurierbarkeit bemängelt. Sowohl für den dynamischen als auch für den generierten Teil eines Page Objects wird die selbe Variable zum Erzeugen des Pfades unterhalb von ‘src.main.java‘ bzw. ‘src.main.generated‘ verwendet. Ein Ablegen des dynamischen und des generierten Teil eines Page Objects in gänzlich verschiedenen Paketen ist damit nicht möglich.
Abgesehen von den Verbesserungsvorschlägen des ersten und zweiten Praxistests wurde der Page Object Generator in beiden Prüfungen von den Anwendern durchwegs als positiv bewertet. Die Generierung der Page Objects wurde als Arbeitserleichterung empfunden, welche

eine Zeitersparnis im Vergleich zum manuellen Erzeugen mit sich bringt. Die Zeitdifferenz zum manuellen Erstellen der Page Objects, wurde in den Praxistests allerdings nicht wissenschaftlich gemessen und überprüft.

6. Fazit

Das manuelle Testen von Software ist ein Bereich der Softwareentwicklung, der in der Literatur mittlerweile gut und umfassend erläutert wird. Anders verhält es sich mit der Testautomatisierung. Dieser Bereich des Testens ist zwar schon lange bekannt, wird in der Literatur jedoch immer noch nicht umfassend genug behandelt. Mit dieser Arbeit wurde ein Werk geschaffen, welches sich mit einer Sparte der Softwareentwicklung befasst, die noch Entwicklungspotential in sich birgt. Die Arbeit behandelte zunächst die Grundlagen des Testens im Allgemeinen und hat herausgearbeitet, dass Testen eine sinnvolle Möglichkeit, ist um die Qualität von Softwareprodukten zu verbessern. Im weiteren Verlauf rückte die Automatisierung von Testfällen in den näheren Fokus. Die verschiedenen Bereiche und Möglichkeiten der Testautomatisierung wurden aufgezeigt, sowie die Vor- und Nachteile herausgearbeitet. Die Testautomatisierung zeichnete sich dabei als Möglichkeit ab, die Erfolge, die mit Hilfe von Softwaretests erzielt werden, weiter zu verbessern und gleichzeitig die dabei eingesetzten Mittel zu reduzieren. Als ein Problem der Testautomatisierung zeigte sich jedoch der oft erhöhte initiale Mehraufwand bei der Erstellung. Dieses Problem wurde auch als mögliches Verbesserungspotential in der Testautomatisierung erkannt.
Um die gewonnenen Erkenntnisse anhand eines Beispiels zu konkretisieren, wurde das Tool Selenium vorgestellt. Auch für dieses Tool hat sich gezeigt, dass der initiale Mehraufwand bei der Testfallerstellung, verglichen mit der manuellen Durchführung der Testfälle, erhöht ist. Darüber hinaus ergab sich, dass dieser Aufwand noch weiter ansteigt, wenn Selenium wie vorgeschlagen in Verbindung mit dem Page Object Pattern verwendet wird.
Für den Einsatz von Selenium in Verbindung mit dem Page Object Pattern wurde daher eine Softwarelösung entwickelt und vorgestellt, welche den Aufwand bei der Erstellung von automatischen Tests reduziert. Die Softwarelösung in Form eines Page Object Generators vereinfacht dazu die Erstellung der Page Objects die beim Einsatz des Page Object Patterns benötigt werden. Page Objects müssen nicht mehr händisch programmiert, sondern können teilautomatisiert aus dem Quelltext einer Webseite generiert werden. Damit trägt der Page Object Generator dazu bei, den initialen Mehraufwand bei der Erstellung des Testfälle zu reduzieren.

6.1. Ausblick

Der Page Object Generator hat einen Stand erreicht, der sich für den produktiven Einsatz eignet. Praxistests zeigten jedoch, dass Generator sowie der zugehörige Testharness, durchaus noch Verbesserungspotenzial bieten. Ein Maximum an Zeitersparnis lässt sich mit dem Page Object Generator erreichen, wenn der Konfigurationsaufwand von Testprojekt und Generator möglichst gering gehalten wird. Ein sinnvoller Ansatzpunkt, um das Projekt SeleniPo weiter voranzutreiben, wäre daher, die Entwicklung des als Testharness bezeichneten Testprojekts zu verbessern. Bei der Implementierung des Testharness, wie in Kapitel 5.3.4.5 beschrieben, handelt es sich um einen Prototypen, der zwar als Grundlage für den produktiven Einsatz verwendet werden kann, jedoch noch unausgereift ist. Durch einen perfekt auf den Generator abgestimmten Testharness könnte die Hemmschwelle für die Benutzung des Tools gesenkt werden, was wiederum die Akzeptanz beim Anwender steigern würde.
Darüber hinaus lässt sich mit einem gut vorbereiteten Testharness die Verwendung von Best Practice Ansätzen über das Page Object Pattern hinaus unterstützen.
Ein weitere Möglichkeit, um den der Page Object Generator zu verbessern, wäre, verschiedene Templates für die Generierung der Page Objects anzubieten. Die derzeit im Page Object Generator angebotenen Templates und damit die generierten Page Object Klassen haben sich vom Selenium vorgeschlagenen Standard für Page Objects entfernt. Selenium schlägt die Verwendung von Page Objects mit annotierten WebElements als Variablen vor. Die mit Locatoren annotierten WebElemente werden bei der Instanziierung über eine PageFactory-Klasse aufgelöst und befüllt. Im Gegensatz dazu erzeugen die im Page Object Generator derzeit hinterlegten Templates PageObject-Klassen, die an Stelle von WebElements die Klasse Control verwenden und abhängig von der der Klasse ByFactory sind (siehe Kapitel 5.3.4.5). Um die von Selenium angebotene PageFactory zu unterstützen, müssen neue Templates geschaffen werden, welche die von der PageFactory-Klasse benötigten Konventionen erfüllen. Templates, die den von Selenium vorgeschlagenen Standard erfüllen, hätten den Vorteil, dass sie den Einstieg für Entwickler, die bereits Erfahrungen mit Selenium und dem WebDriver gesammelt haben, erleichtern würden.
Unabhängig davon, für welchen Weg sich bei der Weiterentwicklung des Generators entschieden wird, ist es für die Landeshauptstadt München vor allem wichtig, den Generator nun über den Pilotbetrieb hinaus in den produktiven Einsatz auszurollen.

A. Anhang

A.1. Anwendungsfallbeschreibung

A.1.1. Neues Page Object anlegen

Kurzbeschreibung:	Entwickler legt ein neues Page Object an.
Akteure:	Entwickler
Motivation:	Entwickler benötigt Page Object in den Testfällen.
Vorbedingung:	
Eingehende Daten:	Name und Paket des Page Object.
Ergebnisse:	Page Object ist ausgewählt.
Nachbedingungen:	Page Object wurde im Modell der Anwendung angelegt.

Ablauf

1. Entwickler startet den Vorgang zum Anlegen eines neuen Page Object.
2. System zeigt Dialog an.
3. Entwickler trägt Namen des Page Object im Dialog ein.
4. Entwickler bestätigt den Dialog.
5. System prüft Eingaben.
6. System wählt Page Object aus.

Vorgang abgebrochen Statt Schritt 4-6:

4. Entwickler bricht Vorgang ab.
5. System ändert internen Zustand nicht.

Validierung fehlgeschlagen Statt Schritt 6:

6. System zeigt Fehlermeldung an.
7. Entwickler bestätigt Fehlermeldung.

Weiter mit Punkt 3.

Paketstruktur des Page Object verfeinern Statt Schritt 4:

4. Entwickler trägt Paket des Page Object ein.

Weiter mit Punkt 4.

A.1.2. Neues Element hinzufügen

Kurzbeschreibung:	Entwickler legt ein Element in einem Page Object an.
Akteure:	Entwickler
Motivation:	Entwickler benötigt Element der Seite in den Testfällen.
Vorbedingung:	Page Object bereits angelegt.
Eingehende Daten:	Interner Name des Elements, Art des Selektors, Locator, der das Element in der Seite identifiziert
Ergebnisse:	Element wird in den Elementen des Page Object angezeigt.
Nachbedingungen:	Element wurde im Modell der Anwendung angelegt.

Ablauf

1. Entwickler wählt Page Object aus.
2. Entwickler wählt Bereich für die Elemente des Page Objects aus.
3. System wechselt in den internen Zustand zum Bearbeiten von Elementen.
4. Entwickler startet den Vorgang zum Anlegen eines neuen Eintrags.
5. System zeigt Dialog an.
6. Entwickler befüllt Dialog.
7. Entwickler bestätigt den Dialog.
8. System prüft, ob alle Felder befüllt sind.
9. System zeigt Element in den Elementen des Page Object an.

Vorgang abgebrochen Statt Schritt 7-9:

7. Entwickler bricht Vorgang ab.
8. System ändert internen Zustand nicht.

Validierung fehlgeschlagen Statt Schritt 9:

9. System zeigt Fehlermeldung an.
10. Entwickler bestätigt Fehlermeldung.

Weiter mit Punkt 6.

A.1.3. Neue Transition hinzufügen

Kurzbeschreibung:	Entwickler legt einen Seitenübergang zu einem anderen Page Object an.
Akteure:	Entwickler
Motivation:	Entwickler benötigt einen Übergang zu einer anderen Seite in den Testfällen.
Vorbedingung:	Page Object bereits angelegt. Page Object, das Ziel des Seitenübergangs ist, wurde bereits angelget.
Eingehende Daten:	Interner Name des Elements, Art des Selektors, Locator der das Element in der Seite identifiziert, Ziel Page Object.
Ergebnisse:	Transition wird in den Transitionen des Page Object angezeigt.
Nachbedingungen:	Transition wurde im Modell der Anwendung angelegt.

Ablauf

1. Entwickler wählt Page Object aus.
2. Entwickler wählt Bereich für die Transitionen des Page Objects aus.
3. System wechselt in den internen Zustand zum Bearbeiten von Transitionen.
4. Entwickler startet den Vorgang zum Anlegen eines neuen Eintrags.
5. System zeigt Dialog an.
6. Entwickler befüllt Dialog.
7. Entwickler bestätigt den Dialog.

8. System prüft ob alle Felder befüllt sind.
9. System zeigt Transition in den Transitionen des Page Object an.

Vorgang abgebrochen Statt Schritt 7-9:

7. Entwickler bricht Vorgang ab.
8. System ändert internen Zustand nicht.

Validierung fehlgeschlagen Statt Schritt 9:

9. System zeigt Fehlermeldung an.
10. Entwickler bestätigt Fehlermeldung.

Weiter mit Punkt 6.

A.1.4. Element aus HTML übernehmen

Kurzbeschreibung:	Entwickler legt teilautomatisiert ein neues Element im Page Object an.
Akteure:	Entwickler
Motivation:	Entwickler benötigt Element der Seite in den Testfällen.
Vorbedingung:	Page Object bereits angelegt.
Eingehende Daten:	Art des Locators, nach dem die Seite durchsucht werden soll.
Ergebnisse:	Element wird in den Elementen des Page Object angezeigt.
Nachbedingungen:	Element wurde im Modell der Anwendung angelegt.

Ablauf

1. Entwickler wählt Page Object aus.
2. Entwickler startet aus der Anwendung heraus den Webbrowser mit der zum ausgewählten Page Object korrespondierenden Seite.
3. Entwickler wählt Art des Locators, für den die Seite durchsucht werden soll, aus.
4. Entwickler wählt Aktion zum Analysieren der ausgewählten Webseite aus.
5. System zeigt die für den ausgewählten Locator auf der Seite identifizierten Elemente an.

6. Entwickler wählt gewünschtes Element aus den Treffern aus.
7. Entwickler wählt Aktion zum Übernehmen des Elements in das Page Object aus.
8. System zeigt Element in den Elementen des Page Object an.

Testen des Elements Statt Schritt 7-8:

7. Entwickler wählt Aktion zum Testen des ausgewählten Elements aus.
8. System zeigt an, ob das Element auf der ausgewählten Webseite erfolgreich erreicht werden konnte.

Weiter mit Punkt 7.

A.1.5. Transition aus HTML übernehmen

Kurzbeschreibung:	Entwickler legt teilautomatisiert einen neuen Seitenübergang an.
Akteure:	Entwickler
Motivation:	Entwickler benötigt einen Übergang zu einer anderen Seite in den Testfällen.
Vorbedingung:	Page Object bereits angelegt. Page Object, das Ziel des Seitenübergangs ist, wurde bereits angelget.
Eingehende Daten:	Art des Locators nach dem die Seite durchsucht werden soll, Ziel Page Object.
Ergebnisse:	Transition wird in den Transitionen des Page Object angezeigt.
Nachbedingungen:	Transition wurde im Modell der Anwendung angelegt.

Ablauf

1. Entwickler wählt passendes Page Object aus.
2. Entwickler startet aus der Anwendung heraus den Webbrowser mit der zum ausgewählten Page Object korrespondierenden Seite.
3. Entwickler wählt Art des Locators, für den die Seite durchsucht werden soll, aus.
4. Entwickler wählt Aktion zum Analysieren der ausgewählten Webseite aus.
5. System zeigt die für den ausgewählten Locator auf der Seite identifizierten Elemente an.
6. Entwickler wählt gewünschtes Element aus den Treffern aus.

7. Entwickler wählt Aktion zum Übernehmen des Elements als Transition in das Page Object aus.
8. System zeigt Dialog mit den Informationen des ausgewählten Elements an.
9. Entwickler ergänzt die Informationen um das Ziel Page Object der Transition.
10. Entwickler bestätigt den Dialog.
11. System prüft, ob alle Felder befüllt sind.
12. System zeigt Element in den Transitionen des Page Object an.

Testen der Transition Statt Schritt 7-12:

7. Entwickler wählt Aktion zum Testen des ausgewählten Elements aus.
8. System zeigt an, ob das Element auf der ausgewählten Webseite erfolgreich erreicht werden konnte.

Weiter mit Punkt 7.

Vorgang abgebrochen Statt Schritt 10-12:

10. Entwickler bricht Vorgang ab.
11. System ändert internen Zustand nicht.

Validierung fehlgeschlagen Statt Schritt 12:

12. System zeigt Fehlermeldung an.
13. Entwickler bestätigt Fehlermeldung.

Weiter mit Punkt 6.

A.1.6. Vorhandenen Eintrag editieren

Kurzbeschreibung:	Entwickler editiert bereits vorhandenen Eintrag.
Akteure:	Entwickler
Motivation:	Entwickler möchte einen bereits vorhandenen Eintrag überarbeiten.
Vorbedingung:	Eintrag bereits vorhanden.
Eingehende Daten:	Geänderte Werte.
Ergebnisse:	Eintrag wird in aktualisierter Form angezeigt.
Nachbedingungen:	Eintrag wurde im Modell der Anwendung aktualisiert.

Ablauf

1. Entwickler wählt den zu editierenden Eintrag aus.
2. Entwickler löst die Aktion zum Editieren der aktuellen Auswahl aus.
3. System zeigt Dialog mit den aktuellen Werten des Eintrags an.
4. Entwickler nimmt die gewünschten Änderungen vor.
5. Entwickler bestätigt den Dialog.
6. System prüft die Felder des Dialogs.
7. System zeigt den Eintrag in aktualisierter Form an.

Vorgang abgebrochen Statt Schritt 5-7:

5. Entwickler bricht Vorgang ab.
6. System ändert internen Zustand nicht.

Validierung fehlgeschlagen Statt Schritt 7:

7. System zeigt Fehlermeldung an.
8. Entwickler bestätigt Fehlermeldung.

Weiter mit Punkt 4.

A.1.7. Vorhandenen Eintrag löschen

Kurzbeschreibung:	Entwickler löscht bereits vorhandenen Eintrag.
Akteure:	Entwickler
Motivation:	Entwickler möchte einen bereits vorhandenen Eintrag löschen.
Vorbedingung:	Eintrag bereits vorhanden.
Eingehende Daten:	
Ergebnisse:	Eintrag wird nicht mehr angezeigt.
Nachbedingungen:	Eintrag wurde aus dem Modell der Anwendung entfernt.

Ablauf

1. Entwickler wählt den zu löschenden Eintrag aus.
2. Entwickler löst die Aktion zum Löschen der aktuellen Auswahl aus.
3. System entfernt den ausgewählten Eintrag.

Zu löschender Eintrag ist gesamtes Page Object Statt Schritt 3:

3. System zeigt einen Bestätigungsdialog an.
4. Entwickler bestätigt Dialog.

Weiter mit Punkt 3.

Vorgang abgebrochen Statt Schritt 4 des Sonderfalls ‘Zu löschender Eintrag ist gesamtes Page Object‘:

4. Entwickler bricht Vorgang ab.
5. System ändert internen Zustand nicht.

A.1.8. Vorhandenen Eintrag testen

Kurzbeschreibung:	Entwickler testet einen vorhandenen Eintrag.
Akteure:	Entwickler
Motivation:	Entwickler möchte die Erreichbarkeit eines bereits vorhandenen Eintrags prüfen.
Vorbedingung:	Eintrag bereits vorhanden.
Eingehende Daten:	
Ergebnisse:	Eintrag ist mit dem Testergebnis markiert.
Nachbedingungen:	Modell der Anwendung ist unverändert.

Ablauf

1. Entwickler startet aus der Anwendung heraus den Webbrowser auf der dem Page Object Korrespondierenden Webseite.
2. Entwickler wählt den zu testenden Eintrag aus.
3. Entwickler löst die Aktion zum Testen der aktuellen Auswahl aus.
4. System zeigt Ergebnis des Tests an.

A.1.9. Laden eines vorhandenen Modells

Kurzbeschreibung:	Entwickler lädt ein zuvor angelegtes und gespeichertes Modell.
Akteure:	Entwickler
Motivation:	Entwickler möchte einen alten Speicherstand laden.
Vorbedingung:	Save Datei vorhanden.
Eingehende Daten:	Save-Datei
Ergebnisse:	Zustand der Save-Datei wiederhergestellt.
Nachbedingungen:	Modell der Anwendung wurde mit den Werten aus der Save-Datei befüllt.

Ablauf

1. Entwickler wählt die Aktion zum Laden einer Save-Datei aus.
2. System öffnet Auswahldialog.
3. Entwickler wählt Save-Datei aus.
4. System zeigt die Einträge der Save-Datei an.

Vorgang abgebrochen Statt Schritt 3:

3. Entwickler bricht Vorgang ab.
4. System ändert internen Zustand nicht.

A.1.10. Speichern eines vorhandenen Modells

Kurzbeschreibung:	Entwickler speichert einen Stand der Anwendung.
Akteure:	Entwickler
Motivation:	Entwickler möchte einen Stand der Anwendung zur späteren Wiederherstellbarkeit speichern.
Vorbedingung:	
Eingehende Daten:	
Ergebnisse:	Save-Datei wurde angelegt.
Nachbedingungen:	Save-Datei im Zielpfad vorhanden.

Ablauf

1. Entwickler wählt die Aktion zum Speichern eines Zwischenstands aus.
2. System öffnet Auswahldialog.
3. Entwickler wählt Ort zum Ablegen der Save-Datei aus.
4. Entwickler gibt Namen für die Save-Datei an.
5. Entwickler bestätigt Dialog.
6. System erzeugt Save-Datei.

Vorgang abgebrochen Statt Schritt 5-6:

5. Entwickler bricht Vorgang ab.
6. System ändert internen Zustand nicht.

A.1.11. Page Objects generieren

Kurzbeschreibung:	Entwickler erzeugt Quellcode aus dem in der Anwendung erzeugten Modell.
Akteure:	Entwickler
Motivation:	Entwickler möchte aus einem Stand der Anwendung Page Object Klassen generieren.
Vorbedingung:	Page Objects vorhanden.
Eingehende Daten:	
Ergebnisse:	Page Object Klassen wurden erzeugt.
Nachbedingungen:	Page Objects im Zielpfad vorhanden.

Ablauf

1. Entwickler wählt die Aktion zum Generieren der Page Objects aus.
2. System öffnet Auswahldialog.
3. Entwickler wählt Ort, in dem die Page Objects abgelegt werden sollen.
4. System generiert aus den Informationen des aktuellen Modells der Anwendung die entsprechenden Page Object Klassen.

Vorgang abgebrochen Statt Schritt 3:

3. Entwickler bricht Vorgang ab.
4. System ändert internen Zustand nicht.

A.2. Zustände des Page Object Generator

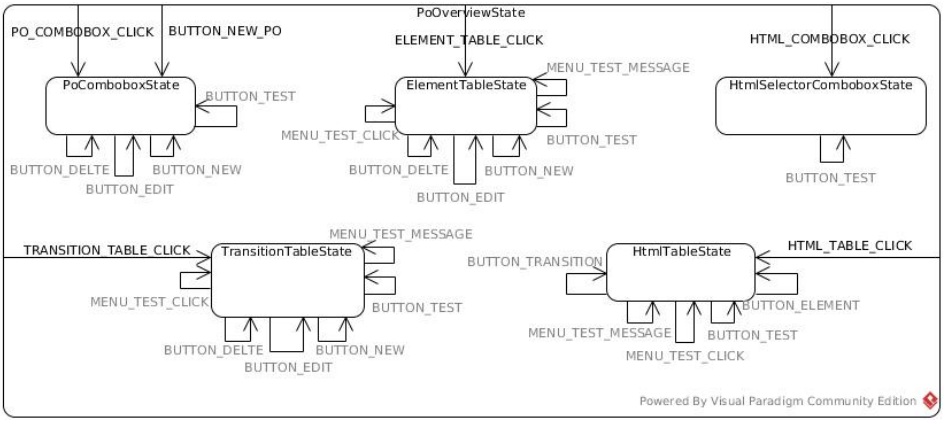


Abbildung A.1.: Zustandsmodell des SeleniPoEditor

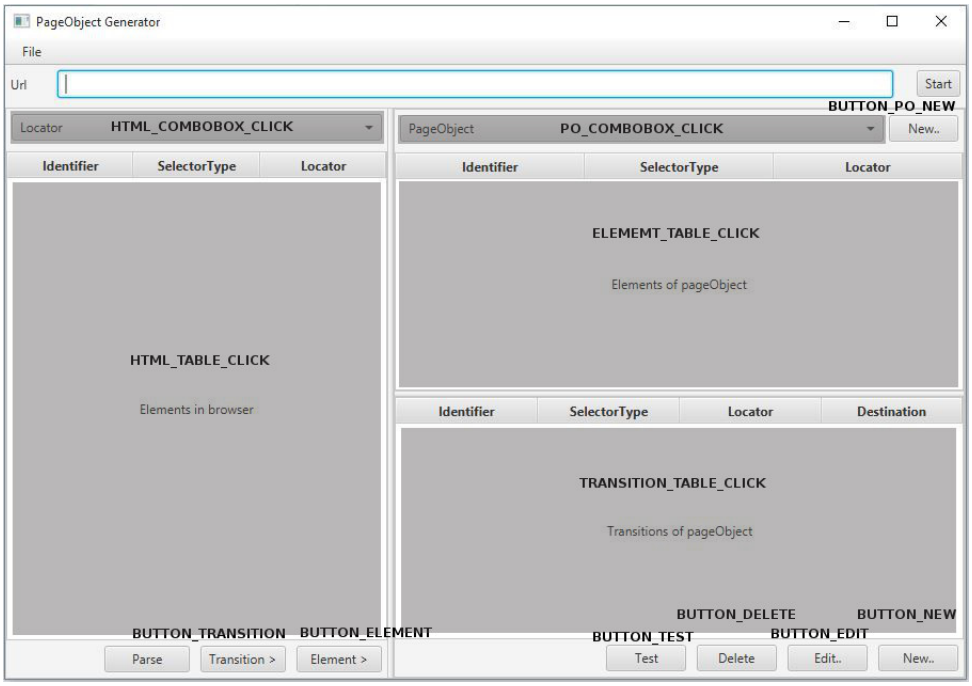
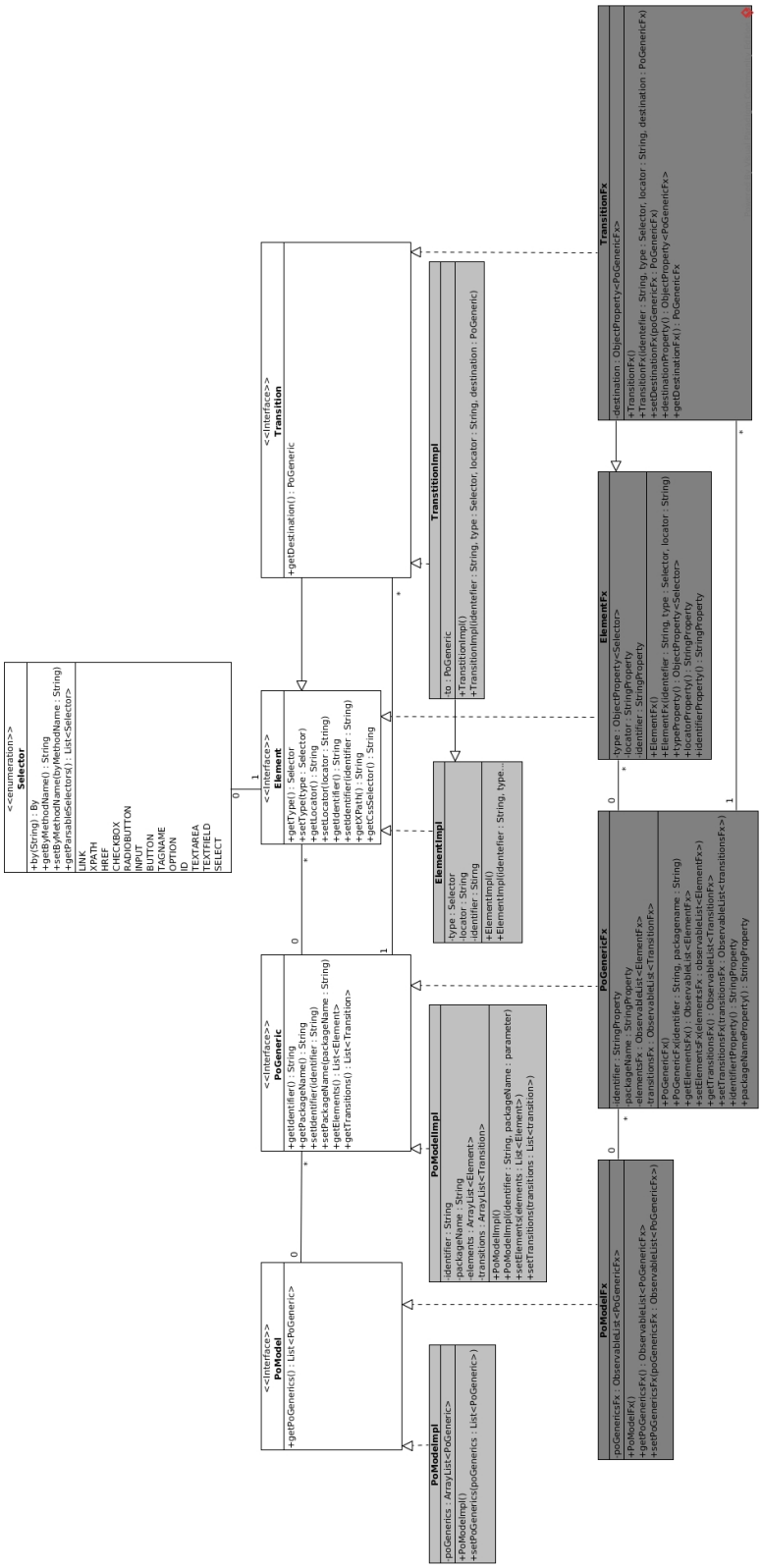


Abbildung A.2.: Zuordnung der in Abbildung A.1 verwendeten Namen

A.3. Vollständiges technisches Modell



A.4. Beispiel für ein Velocity Template

Listing A.1: poGenerated.vm

```
#set( $BASE_PACKAGE = $basePackagePath )
#set( $GENERATED_PACKAGE = "$BASE_PACKAGE#if ( $poGeneric.getPackageName() &&
    $poGeneric.getPackageName().trim().size() != 0
    ).$poGeneric.getPackageName()#end" )
#set( $EDIT_PACKAGE = "$BASE_PACKAGE" )
#set( $CLASS_NAME = "$poGeneric.getIdentifier()Generated" )

package $GENERATED_PACKAGE;

import ${BASE_PACKAGE}.BasePo;
import ${BASE_PACKAGE}.Control;
import ${BASE_PACKAGE}.PageObject;
#foreach( $destPo in $destionationPos )
import $EDIT_PACKAGE#if ( $destPo.getPackageName() &&
    $destPo.getPackageName().trim().size() != 0
    ).$destPo.getPackageName()#end.$destPo.getIdentifier();
#end

public class $CLASS_NAME extends BasePo {

#foreach( $element in $poGeneric.getElements() )
   public final Control $element.getIdentifier() =
       control(by.$element.getType().getByMethodName()("$element.getLocator()"));
#end
#foreach( $transition in $poGeneric.getTransitions() )
   public final Control $transition.getIdentifier() =
       control(by.$transition.getType().getByMethodName()("$transition.getLocator()"));
#end
   public $CLASS_NAME(PageObject po) {
      super(po);
   }
#foreach( $transition in $poGeneric.getTransitions() )

```

```
28    public $transition.getDestination().getIdentifier()
          click${display.capitalize($transition.getIdentifier())}() {
29        ${transition.getIdentifier()}.click();
30        return new $transition.getDestination().getIdentifier()(this);
31    }
32 #end
33 #foreach( $element in $poGeneric.getElements() )
34    /**
35     * Get the Control for the Element $transition.getIdentifier().
36     * @return $transition.getIdentifier() - Element
37     */
38    public Control get${display.capitalize($element.getIdentifier())}() {
39        return $element.getIdentifier();
40    }
41 #end
42 #foreach( $transition in $poGeneric.getTransitions() )
43    /**
44     * Get the Control for the Transition $transition.getIdentifier().
45     * @return $transition.getIdentifier() - Transition
46     */
47    public Control get${display.capitalize($transition.getIdentifier())}() {
48        return $transition.getIdentifier();
49    }
50
51 #end
52 }
```

Listing A.2: poEditable.vm

```
1 #set( $BASE_PACKAGE = $basePackagePath )
2 #set( $EDIT_PACKAGE = "$BASE_PACKAGE#if ( $poGeneric.getPackageName() &&
      $poGeneric.getPackageName().trim().size() != 0
      ).$poGeneric.getPackageName()#end" )
3 #set( $GENERATED_PACKAGE = "$BASE_PACKAGE#if ( $poGeneric.getPackageName() &&
      $poGeneric.getPackageName().trim().size() != 0
      ).$poGeneric.getPackageName()#end" )
4 #set( $CLASS_NAME = "$poGeneric.getIdentifier()" )
```

```
5  #set( $CLASS_NAME_SUPER = "$poGeneric.getIdentifier()Generated" )
6
7  package $EDIT_PACKAGE;
8
9  import ${BASE_PACKAGE}.BasePo;
10 import ${BASE_PACKAGE}.Control;
11 import ${BASE_PACKAGE}.PageObject;
12 import $GENERATED_PACKAGE.$CLASS_NAME_SUPER;
13
14 public class $CLASS_NAME extends $CLASS_NAME_SUPER {
15
16    public $CLASS_NAME(PageObject po) {
17       super(po);
18    }
19 }
```

Literatur

[Ama+14] Y. Amannejad u. a. „A Search-Based Approach for Cost-Effective Software Test Automation Decision Support and an Industrial Case Study“. In: *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. 2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops (ICSTW). März 2014, S. 302–311. DOI: 10.1109/ICSTW.2014.34.

[Apa15] Apache Software Foundation. *Apache Velocity Site - The Apache Velocity Project*. 16. Nov. 2015. URL: http://velocity.apache.org/ (besucht am 16. 11. 2015).

[Bou+08] Fabrice Bouquet u. a. „A Test Generation Solution to Automate Software Testing“. In: *Proceedings of the 3rd International Workshop on Automation of Software Test*. AST '08. New York, NY, USA: ACM, 2008, S. 45–48. ISBN: 9781605580302. DOI: 10.1145/1370042.1370052. URL: http://doi.acm.org/10.1145/1370042.1370052 (besucht am 17. 11. 2014).

[DL15] Bondurant Daniel und Kevin London. *wiredrive/wtframework*. GitHub. 20. Okt. 2015. URL: https://github.com/wiredrive/wtframework (besucht am 20. 10. 2015).

[Dmy15] Dmytro Zharii. *dzharii/swd-recorder*. GitHub. 20. Okt. 2015. URL: https://github.com/dzharii/swd-recorder (besucht am 20. 10. 2015).

[DRP99] Elfriede Dustin, Jeff Rashka und John Paul. *Automated Software Testing: Introduction, Management, and Performance*. Addison-Wesley Professional, 28. Juni 1999. 602 S. ISBN: 9780672333842.

[Dus01] Elfriede Dustin. *Software automatisch testen*. Xpert.press. Berlin [u.a.]: Springer, 2001. XXIII, 649 S. ISBN: 9783540676393.

[FG99] Mark Fewster und Dorothy Graham. *Software Test Automation Effective use of test execution tools*. Addison-Wesley, 1999. ISBN: 0201331403.

[Har00] Mary Jean Harrold. „Testing: A Roadmap“. In: *Proceedings of the Conference on The Future of Software Engineering*. ICSE '00. New York, NY, USA: ACM, 2000, S. 61–72. ISBN: 1581132530. DOI: 10.1145/336512.336532. URL: http://doi.acm.org/10.1145/336512.336532 (besucht am 28. 10. 2014).

[Hed15] Jonathan Hedley. *jsoup Java HTML Parser, with best of DOM, CSS, and jquery*. 24. Nov. 2015. URL: http://jsoup.org/ (besucht am 24. 11. 2015).

[Hof13] Dirk W. Hoffmann. *Software-Qualität*. 2013. Aufl. Berlin: Springer, 2013. 568 S. ISBN: 9783540763222.

[HP15] HP. *Testautomatisierung, Unified Functional Testing, UFT | HP® Deutschland*. 11. Aug. 2015. URL: http://www8.hp.com/de/de/software-solutions/unified-functional-automated-testing/ (besucht am 11. 08. 2015).

[Htt15] HttpUnit. *HttpUnit Home*. 11. Aug. 2015. URL: http://httpunit.sourceforge.net/ (besucht am 11. 08. 2015).

[IEE08] IEEE. *IEEE Std 829-2008 IEEE Standard for Software and System Test Documentation*. Juli 2008.

[IEE91] IEEE. „IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries“. In: *IEEE Std 610* (Jan. 1991), S. 1–217. DOI: 10.1109/IEEESTD.1991.106963.

[ISO01] ISO/IEC. *ISO/IEC 9126. Software engineering – Product quality*. 2001.

[ISO11] ISO/IEC. *ISO/IEC 25010:2011 - Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models*. 2011. URL: http://www.iso.org/iso/catalogue_detail.htm?csnumber=35733 (besucht am 15. 12. 2015).

[Joe15] Joe Walnes. *XStream - About XStream*. 16. Nov. 2015. URL: http://x-stream.github.io/ (besucht am 16. 11. 2015).

[Leo+13] M. Leotta u. a. „Repairing Selenium Test Cases: An Industrial Case Study about Web Page Element Localization“. In: *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation (ICST)*. 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation (ICST). März 2013, S. 487–488. DOI: 10.1109/ICST.2013.73.

[Mes03] Gerard Meszaros. „Agile Regression Testing Using Record & Playback“. In: *Companion of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*. OOPSLA '03. New York, NY, USA: ACM, 2003, S. 353–360. ISBN: 1581137516. DOI: 10.1145/949344.949442. URL: http://doi.acm.org/10.1145/949344.949442 (besucht am 17. 11. 2014).

[MPS00] Atif M. Memon, Martha E. Pollack und Mary Lou Soffa. „Automated Test Oracles for GUIs“. In: *Proceedings of the 8th ACM SIGSOFT International Symposium on Foundations of Software Engineering: Twenty-first Century Applications*. SIGSOFT '00/FSE-8. New York, NY, USA: ACM, 2000, S. 30–39. ISBN: 1581132050. DOI: 10.1145/355045.355050. URL: http://doi.acm.org/10.1145/355045.355050 (besucht am 17. 11. 2014).

[Ora15] Oracle. *Client Technologies: Java Platform, Standard Edition (Java SE) 8 Release 8*. 16. Nov. 2015. URL: http://docs.oracle.com/javafx (besucht am 16. 11. 2015).

[RAO92] D.J. Richardson, S.L. Aha und T.O. O'Malley. „Specification-based test oracles for reactive systems“. In: *International Conference on Software Engineering, 1992*. International Conference on Software Engineering, 1992. 1992, S. 105–118. DOI: 10.1109/ICSE.1992.753494.

[Ros10] Thomas Rossner. *Basiswissen modellbasierter Test*. 1. Aufl. Heidelberg: dpunkt-Verl., 2010. XX, 404 S. ISBN: 9783898645898.

[Roy87] W. W. Royce. „Managing the Development of Large Software Systems: Concepts and Techniques“. In: *Proceedings of the 9th International Conference on Software Engineering*. ICSE '87. Los Alamitos, CA, USA: IEEE Computer Society Press, 1987, S. 328–338. ISBN: 9780897912167. URL: http://dl.acm.org/citation.cfm?id=41765.41801 (besucht am 13. 07. 2015).

[RW06] Rudolf Ramler und Klaus Wolfmaier. „Economic Perspectives in Test Automation: Balancing Automated and Manual Testing with Opportunity Cost“. In: *Proceedings of the 2006 International Workshop on Automation of Software Test*. AST '06. New York, NY, USA: ACM, 2006, S. 85–91. ISBN: 1595934081. DOI: 10.1145/1138929.1138946. URL: http://doi.acm.org/10.1145/1138929.1138946 (besucht am 28. 10. 2014).

[SBB12] Richard Seidl, Manfred Baumgartner und Thomas Bucsics. *Basiswissen Testautomatisierung*. 1. Aufl. Heidelberg: dpunkt-Verl., 2012. ISBN: 9783898647243.

[Sch02] Ken Schwaber. *Agile software development with Scrum*. Pearson Internat. Ed. Series in agile software development. Upper Saddle River, NJ: Pearson Education International, 2002. XVI, 158 S. : Ill., graph. Darst. ISBN: 9780132074896.

[Sel15a] Selenium. *Selenium Documentation*. 31. Aug. 2015. URL: `http://www.seleniumhq.org/docs/` (besucht am 31. 08. 2015).

[Sel15b] Selenium. *Selenium - Test Design Considerations*. 29. Sep. 2015. URL: `http://www.seleniumhq.org/docs/06_test_design_considerations.jsp` (besucht am 29. 09. 2015).

[Sel15c] Selenium. *Selenium - Web Browser Automation*. 11. Aug. 2015. URL: `http://www.seleniumhq.org/` (besucht am 11. 08. 2015).

[Sil15] Silk Test. *Borland | Silk Test: Automation testing tool*. 11. Aug. 2015. URL: `http://www.borland.com/Products/Software-Testing/Automated-Testing/Silk-Test` (besucht am 11. 08. 2015).

[SKM09] S.R. Shahamiri, W.M.N.W. Kadir und S.Z. Mohd-Hashim. „A Comparative Study on Automated Software Test Oracle Methods“. In: *Fourth International Conference on Software Engineering Advances, 2009. ICSEA '09*. Fourth International Conference on Software Engineering Advances, 2009. ICSEA '09. Sep. 2009, S. 140–145. DOI: `10.1109/ICSEA.2009.29`.

[SL07] Andreas Spillner und Tilo Linz. *Basiswissen Softwaretest*. 3. Aufl. Heidelberg: dpunkt-Verl., 2007. XV, 228 S. : Ill., graph. Darst. ISBN: 3898643581.

[Sto+15] A. Stocco u. a. „Why Creating Web Page Objects Manually If It Can Be Done Automatically?“ In: *2015 IEEE/ACM 10th International Workshop on Automation of Software Test (AST)*. 2015 IEEE/ACM 10th International Workshop on Automation of Software Test (AST). Mai 2015, S. 70–74. DOI: `10.1109/AST.2015.26`.

[Tha02] Georg Erwin Thaller. *Software-Test*. 2., aktualisierte und erw. Aufl. Hannover: Heise, 2002. XI, 383 S. ISBN: 3882291982.

[vir15] virtuetech GmbH. *OHMAP - Create page objects faster than ever before*. 20. Okt. 2015. URL: `http://ohmap.virtuetech.de/` (besucht am 20. 10. 2015).