h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES

Hochschule Darmstadt

– Fachbereich Informatik –

Vergleich von Page Object Pattern und Screenplay Pattern am Beispiel der CAT-Anwendung

Abschlussarbeit zur Erlangung des akademischen Grades

Bachelor of Science (B.Sc.)

vorgelegt von

Hongmei Piao

Matrikelnummer: 762495

Referent	:	Prof. Dr. Kai Renz
Korreferent	:	Prof. Dr. Jürgen Kilian

ERKLÄRUNG

Ich versichere hiermit, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die im Literaturverzeichnis angegebenen Quellen benutzt habe.

Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder noch nicht veröffentlichten Quellen entnommen sind, sind als solche kenntlich gemacht.

Die Zeichnungen oder Abbildungen in dieser Arbeit sind von mir selbst erstellt worden oder mit einem entsprechenden Quellennachweis versehen.

Diese Arbeit ist in gleicher oder ähnlicher Form noch bei keiner anderen Prüfungsbehörde eingereicht worden.

Darmstadt, 28. August 2023

Hongmei Piao

ABSTRACT

In recent years, test automation has proliferated and become an indispensable part of agile projects. Automated User Interface (UI) tests can effectively assess the end user's perspective early on to ensure that the application under test has as few bugs and defects as possible. However, implementing well-maintainable UI tests requires considerable time and cost. For this reason, a design pattern such as the *Page Object Model (POM)* is widely used in UI test automation. However, the classic POM is less suitable for complex applications due to code duplication and violations of SOLID principles. To address these challenges, the classic POM can be extended or another design pattern such as the *Screenplay Pattern (Screenplay)* can be used.

The objective of this study is to perform a comparative analysis between the classical POM, the extended POM and the *Screenplay* based on different criteria. The criteria analyzed include the readability of the feature files, the execution of tests with multiple users, Quality Model for Object-Oriented Design (QMOOD), maintainability, and compliance with the principles of software development. For this purpose, UI tests are implemented for a real application using these three approaches. The focus is to identify the differences between these approaches and determine in which specific project environments each approach is better suited.

The results of the comparison illustrate that the *Screenplay* approach offers clear advantages for applications that are used interactively by multiple users. In contrast to the classic POM, the extended POM and the *Screenplay* have a higher quality and are more maintainable. Both approaches can be used for large-scale applications, but with different strengths. The *Screenplay* is particularly suitable when an application has many identical elements. It is also well suited for projects that take a *Behavior Driven Development (BDD)* approach and where the product owner and customers work closely with the feature file. The extended POM is characterized by good maintainability and the step definitions can be reused, which reduces the maintenance effort compared to the *Screenplay*.

Keywords: UI test automation, Design pattern, Page object model, Screenplay, SOLID, DRY, QMOOD

ZUSAMMENFASSUNG

In den vergangenen Jahren hat sich die Testautomatisierung stark verbreitet und ist zu einem unverzichtbaren Bestandteil agiler Projekte geworden. Durch automatisierte Oberflächentests kann frühzeitig die Perspektive des Endnutzers effektiv bewertet werden, um sicherzustellen, dass die zu testende Anwendung möglichst wenige Fehler und Mängel aufweist. Allerdings erfordert die Implementierung gut wartbarer Oberflächentests beträchtlichen Zeit- und Kostenaufwand. Aus diesem Grund wird ein Entwurfsmuster wie das POM verwendet, das in der Testautomatisierung von Oberflächentests weit verbreitet ist. Jedoch ist das klassische POM für komplexe Anwendungen aufgrund von Code-Duplizierung und Verstößen gegen SOLID-Prinzipien weniger geeignet. Um diesen Herausforderungen zu begegnen, kann das klassische POM erweitert oder ein anderes Entwurfsmuster wie das *Screenplay Pattern (Screenplay)* verwendet werden.

Das Ziel dieser Arbeit besteht darin, eine vergleichende Analyse zwischen dem klassischen POM, dem erweiterten POM und dem *Screenplay* anhand unterschiedlicher Kriterien durchzuführen. Zu den analysierten Kriterien zählen die Lesbarkeit der *Feature*-Dateien, die Durchführung von Tests mit mehreren Benutzern, QMOOD, die Wartbarkeit sowie die Einhaltung der Prinzipien der Softwareentwicklung. Hierfür werden Oberflächentests anhand einer realen betrieblichen Anwendung unter Verwendung dieser drei Ansätze durchgeführt. Der Fokus liegt darauf, die Unterschiede zwischen diesen Ansätzen zu identifizieren und zu bestimmen, in welchen spezifischen Projektumgebungen welcher Ansatz besser geeignet ist.

Die Ergebnisse des Vergleichs verdeutlichen, dass das *Screenplay* klare Vorteile für Anwendungen bietet, die von mehreren Benutzern interaktiv genutzt werden. Im Gegensatz zum klassischen POM weisen das erweiterte POM und das *Screenplay* eine höhere Qualität auf und sind besser wartbar. Beide Ansätze können für umfangreiche Anwendungen eingesetzt werden, jedoch mit unterschiedlichen Stärken. Das *Screenplay* ist insbesondere geeignet, wenn eine Anwendung viele identische Elemente aufweist. Zudem eignet es sich gut für Projekte, die einen BDD-Ansatz verfolgen und bei denen der Product Owner und die Kunden eng mit der *Feature* Datei zusammenarbeiten. Das erweiterte POM zeichnet sich durch gute Wartbarkeit aus, da die *Step Definitions* wiederverwendet werden können, wodurch der Wartungsaufwand im Vergleich zum *Screenplay* geringer ausfällt.

Schlagwörter: Automatisierte Oberflächentest, Entwurfsmuster, Page Object Model, Screenplay, SOLID, DRY, QMOOD

DANKSAGUNG

An dieser Stelle möchte ich meinen aufrichtigen Dank an all jene Menschen richten, die mich während meiner Bachelorarbeit begleitet haben.

Zuerst möchte ich mich bei Prof. Dr. Kai Renz bedanken, der mich mit wertvollen und konstruktiven Anregungen sowohl bei der Themenauswahl als auch beim Verfassen meiner Abschlussarbeit unterstützt hat.

Ein herzliches Dankeschön gebührt meinen Kollegen bei der Accelerated Solutions GmbH, insbesondere meinen beiden Betreuern, Dr. Nikolaus Fröhlich und Jenja Schurse. Sie haben sich stets die Zeit genommen, mir zur Seite zu stehen, und mir wertvolle Ratschläge für die Ausarbeitung gegeben. Zusätzlich möchte ich meinem Personalverantwortlichen Dr. Martin Kronenburg danken, der sich die Mühe gemacht hat, meine Bachelorarbeit zu lesen und mir hilfreiches Feedback zukommen zu lassen. Mein Dank gilt auch Manuela Simons, die mir während meines Praktikums umfassendes Wissen vermittelt und mir die Inspiration für diese Bachelorarbeit geschenkt hat.

Des Weiteren möchte ich meinem Ehemann sowie meinen Eltern meinen Dank aussprechen, die mich während meiner gesamten Studienzeit bis hin zur Fertigstellung meiner Bachelorarbeit in vielerlei Hinsicht tatkräftig unterstützt haben.

INHALTSVERZEICHNIS

ABBILDUNGSVERZEICHNIS

TABELLENVERZEICHNIS

ABKÜRZUNGSVERZEICHNIS

Accso	Accelerated Solutions GmbH
POM	Page Object Model
SRP	Single Responsibility Principle
OCP	Open Closed Principle
LSP	Liskov Substitution Principle
ISP	Interface Segregation Pricinciple
DIP	Dependency Inversion Principle
DRY	Don't repeat yourself
QMOOD	Quality Model for Object-Oriented Design
BDD	Behavior Driven Development
TQI	Total Quality Index
CAT	Controlling and Timereporting
UI	User Interface
LoC	Lines of Code
REST-API	Representational State Transfer-Application Programming Interface

1 EINLEITUNG

1.1 MOTIVATION UND PROBLEMSTELLUNG

In den vergangenen Jahren hat sich die Verbreitung der Testautomatisierung signifikant erhöht. Dies lässt sich wesentlich auf den Übergang vom traditionellen Wasserfall-Ansatz hin zum agilen Ansatz zurückführen. [Axe18] Im agilen Umfeld nimmt das Testen eine äußerst bedeutende Rolle ein, da es sicherstellt, dass die Software auch bei kontinuierlichen Veränderungen stets fehlerfrei ausgeführt werden kann. Um die zügige und inkrementelle Bereitstellung der Software fortlaufend zu testen, erweist sich die Testautomatisierung als unverzichtbarer Bestandteil in agilen Projekten.

Darüber hinaus bieten automatisierte Oberflächentests in agilen Projekten die Möglichkeit, frühzeitig die Perspektive des Endanwenders zu evaluieren. Jedoch haben sie die große Herausforderung, dass auch kleine Modifikationen an dem UI viele Tests zum Fehlschlagen bringen können. Zudem gestaltet sich die Implementierung gut wartbarer Oberflächentests als zeitaufwändig und kostenintensiv. [Coh10] Deshalb wird für die automatisierten Oberflächentests häufig ein Entwurfsmuster verwendet, um die Effizienz und Wartbarkeit der Tests zu verbessern. *Page Object Model* (POM), auch *Page Object Pattern* genannt, ist ein weit verbreitetes und beliebtes Entwurfsmuster in der Testautomatisierung von Oberflächentests. Ein Hauptvorteil vom POM besteht darin, dass die Testcodes in *Step Definitions* und die seitenspezifischen Codes voneinander getrennt sind. Dies ermöglicht es, bei Veränderungen an dem UI lediglich die Seitenklassen anzupassen, anstatt an mehreren Stellen Änderungen vornehmen zu müssen.

Wenn allerdings die zu testende Anwendung komplex ist, führt das POM schnell dazu, dass *Page Objects* groß werden und viele Code-Duplizierungen enthalten. Außerdem verstößt das POM gegen die SOLID-Prinzipien, die dazu dienen, ein gut wartbares und leicht verständliches objektorientiertes Design zu fördern. [Mar+16b]

Um dies zu vermeiden, kann das POM so erweitert werden, dass die generischen Webelemente wie Textfeld oder Button in zentrale Klassen ausgelagert und wiederverwendet werden. Die *Accelerated Solutions GmbH (Accso)* hat diese Methodik entwickelt und sie erfolgreich in verschiedenen Kundenprojekten angewendet. In dieser Arbeit wird sie als erweitertes POM bezeichnet, um eine klare Abgrenzung vom klassischen POM zu gewährleisten.

Ein weiteres Entwurfsmuster, das als eine Alternative zum POM bezeichnet wird, ist das sogenannte *Screenplay Pattern* (*Screenplay*). Im Gegensatz zum POM wird beim *Screenplay* der benutzerzentrierte Ansatz verfolgt. Außerdem basiert *Screenplay* auf den SOLID-Prinzipien und ermöglicht es, sauberen, lesbaren, skalierbaren und gut wartbaren Testcode zu erstellen. [Mar+16a]

Die drei Entwurfsmuster haben unterschiedliche Stärken und Schwächen. Deshalb ist es sinnvoll, eine Vergleichsanalyse zwischen dem klassischen POM, dem erweiterten POM und dem *Screenplay* unter Verwendung unterschiedlicher Kriterien durchzuführen, um die Vor- und Nachteile der jeweiligen Entwurfsmuster abzuwägen.

1.2 ZIEL UND VORGEHENSWEISE

Aus der Problemstellung leitet sich folgende Forschungsfrage ab: Für welche Art der Web-Anwendungen kommt das klassische POM, erweiterte POM oder *Screenplay* in Frage? Wie unterscheiden sich das klassische POM, erweiterte POM und *Screenplay* voneinander? In welchen Projektkontexten ist das klassische POM, erweiterte POM oder *Screenplay* sinnvoller oder geeigneter?

Das Ziel dieser Arbeit besteht darin, eine vergleichende Analyse zwischen dem klassischen POM, dem erweiterten POM und dem *Screenplay* anhand unterschiedlicher Kriterien durchzuführen. Hierzu werden die Oberflächentests für eine tatsächliche betriebliche Anwendung unter Verwendung dieser drei Ansätze implementiert. Der Fokus liegt darauf, die Unterschiede zwischen diesen Ansätzen zu identifizieren und zu bestimmen, in welchen spezifischen Projektumgebungen welcher Ansatz besser geeignet ist.

Um einen Vergleich zwischen den Entwurfsmustern zu ermöglichen, bedarf es der Verwendung identischer Tools, Konfigurationen und derselben Programmiersprache. In dieser Arbeit werden *Cucumber*, *Serenity BDD*, *Selenium* und *Java* als Programmiersprache verwendet. Zur Untersuchung der Unterschiede zwischen den Entwurfsmustern steht eine interne Plattform von der Accso zur Verfügung.

1.3 AUFBAU DER ARBEIT

In Kapitel 2 werden zunächst die verwandten Arbeiten und die Grundlagen vorgestellt, die für diese Arbeit relevant sind. Diese beinhalten BDD, SOLID-Prinzipien, Don't repeat yourself (DRY)-Prinzip und *QMOOD*.

Im Anschluss folgt Kapitel 3, in dem das klassische POM, erweiterte POM und *Screenplay* vorgestellt werden. Zuerst wird die Vorgehensweise des klassischen POM erläutert, einschließlich der damit verbundenen Probleme. Anschließend werden zwei alternative Ansätze, das erweiterte POM und *Screenplay*, präsentiert. Zur Veranschaulichung der Unterschiede zwischen den drei Entwurfsmustern wird ein gemeinsamer Testfall herangezogen, um ihre jeweilige Struktur und Funktionsweise detailliert zu erläutern.

Kapitel 4 behandelt die Verwendung der drei Entwurfsmuster auf ein konkretes Beispiel: die *CAT*-Anwendung von der Accso. Dabei werden sie anhand verschiedener Kriterien verglichen, um die Unterschiede zwischen den Ansätzen herauszufinden und festzustellen, welches Entwurfsmuster in welchem Projektkontext geeigneter ist.

Im abschließenden Kapitel werden die Ergebnisse zusammengefasst und es wird ein Ausblick auf zukünftige Arbeiten gegeben.

2 GRUNDLAGEN UND VERWANDTE ARBEITEN

In diesem Abschnitt werden zunächst die Veröffentlichungen vorgestellt, die sich mit dem POM, dem *Screenplay* und der Bewertung der Qualität der Enwurfsmuster, beschäftigt haben. In dem zweiten Abschnitt werden die grundlegende Konzepte behaldelt, die für diese Arbeit relevant sind, darunter BDD, QMOOD sowie Prinzipien der Softwareentwicklung.

2.1 STAND DER FORSCHUNG

Maurizio Leotta, Diego Clerissi, Filippo Ricca und Cristiano Spadaro haben über eine industrielle Fallstudie in einem kleinen italienischen Unternehmen berichtet. Darin wurde untersucht, wie POM zur Verbesserung der Wartbarkeit von Testfällen beitragen kann. Dabei wurden der Zeitaufwand und die Anzahl der geänderten Zeilen des Quellcodes (*LoC*) zur Reparatur der zwei äquivalenten Testsuiten verglichen, eine mit dem POM und eine ohne. Die zentrale Erkenntnis dieser Untersuchung ist, dass die Verwendung vom POM signifikante Vorteile mit sich bringt. Es konnte eine Reduzierung des Zeitaufwands für die Reparatur der Testsuiten um 65,32 % festgestellt werden. Zusätzlich wurde eine erhebliche Verringerung der Anzahl der zu ändernden LoC um 87,7 % erreicht. [Leo+13]

Fadi Wedyan und Somia Abufakher verrichteten eine systematische Durchsicht der Fachliteratur, um die Auswirkungen von Entwurfsmustern auf die Qualität von Software zu untersuchen. Im Rahmen dieser Untersuchung wurden folgende Forschungsfragen beantwortet: Welche spezifischen Qualitätsattribute werden analysiert? Was genau wird gemessen? Welche Metriken kommen zur Anwendung? Dabei kristallisierte sich heraus, dass das am häufigsten untersuchte Qualitätsattribut die Wartbarkeit ist. Als vorherrschende Metriken wurden besonders die Anzahl der Modifikationen an einer Klasse sowie Veränderungen in den LoC verwendet. Darüber hinaus wurde in drei Untersuchungen QMOOD als Metrik verwendet. Die resultierenden Erkenntnisse aus diesen Studien zeigten übereinstimmend auf, dass die Anwendung von Entwurfsmustern im Allgemeinen eine Verbesserung der Wiederverwendbarkeit und Flexibilität bewirkt. [WA20]

Jagdish Bansiya und Carl G. Davis haben ein hierarchisches Modell QMOOD zur Bewertung von Qualitätsattributen in objektorientierten Entwürfen beschrieben. Dieses Modell bewertet strukturelle und verhaltensorientierte Designeigenschaften von Klassen, Objekten und ihren Beziehungen mithilfe einer Reihe von objektorientierten Designmetriken. Außerdem ermittelt es den Wert für Funktionalität, Wiederverwendbarkeit, Flexibilität, Verständlichkeit, Wirksamkeit und Erweiterbarkeit jedes zu vergleichenden Entwurfsmusters. Die gemessenen Werte dieser sechs Qualitätsattribute werden summiert, um

den *TQI* für das Projekt bereitzustellen. Die Fähigkeit des Modells, die Gesamtqualität des Entwurfs anhand von Entwurfsinformationen abzuschätzen, wurde anhand mehrerer funktional gleichwertiger Projekte validiert. [BD02]

Eine vergleichende qualitative Analyse vom POM und dem *Screenplay* unter Verwendung des QMOOD wurde von Dini Yuniasri, Tessy Badriya und Umi Sa'adah durchgeführt. Die Ergebnisse dieser Analyse zeigen, dass das POM in Bezug auf die Bewertung von Funktionalität und Effektivität eine höhere Qualität aufweist als das *Screenplay*. Im Gegensatz dazu zeigt das *Screenplay* eine bessere Wiederverwendbarkeit, Flexibilität und Erweiterbarkeit. [YBS20] Es sei jedoch angemerkt, dass im Gegensatz zur Studie von Jagdish Bansiya und Carl G. Davis die erzielten Zahlen in dieser Analyse nicht normalisiert wurden, obwohl sie sich in unterschiedlichen Wertebereichen bewegen.

2.2 BEHAVIOR DRIVEN DEVELOPMENT (BDD)

BDD ist eine Software-Entwicklungsmethodik, die das Ziel verfolgt, die Lücke zwischen Fachbereichen und der umsetzenden IT-Abteilung hinsichtlich des Verständnisses der zu lösendenden Problemstellung zu minimieren. Die Kernidee ist die Schaffung einer automatisiert validierbaren Systemdokumentation, die auf Grundlage des tatsächlich implementierten Systemverhaltens generiert wird. [Cuca]

Die Umsetzung dieser Ziele erfolgt durch eine Fokussierung der interdisziplinären Zusammenarbeit auf konkrete, realitätsnahe Beispiele. Diese Beispiele fungieren als Leitfaden, um den gesamten Entwicklungsprozess von der anfänglichen Konzeption über die Umsetzung bis hin zum Test und der Systemdokumentation anzutreiben.

Um diese Ziele zu erreichen, sind Werkzeuge erforderlich, um ausführbare Beispiele (Testfälle) in einer für Fachbereiche verständlichen Weise zu formulieren. Die Beschreibungssprache *Gherkin* erfüllt genau diesen Zweck, indem sie die Formulierung von automatisierbaren Tests in natürlicher Sprache ermöglicht. *Cucumber* ist ein Testframework, das *Gherkin* zur Beschreibung der Testfälle verwendet und in dieser Arbeit genutzt wird.

Die zentralen Schlüsselwörter von *Cucumber* sind *Feature*, *Given*, *When*, *Then* und *And*. Die Funktion des Schlüsselworts *Feature* besteht darin, eine übergreifende Beschreibung einer Software-Funktionalität zu liefern und verwandte Szenarien zu gruppieren. Jeder Schritt eines Tests beginnt mit einem der Schlüsselwörter *Given*, *When* oder *Then*. Dabei beschreibt *Given* die notwendigen Voraussetzungen für das Szenario und bereitet das Testumfeld vor. *When* gibt die auszuführende Aktion an, während *Then* die erwarteten Resultate beschreibt. *And* kann eingesetzt werden, wenn mehrere Bedingungen in *Given* oder *Then* verwendet werden sollen, um einen besseren Lesefluss zu erreichen. [Cucb]

2.3 PRINZIPIEN DER SOFTWAREENTWICKLUNG

2.3.1 *SOLID-Prinzipien*

Die Abkürzung SOLID setzt sich aus den Anfangsbuchstaben der Prinzipien *Single Responsibility Principle (SRP)*, *Open Closed Principle (OCP)*, *Liskov Substitution Principle (LSP)*, *Interface Segregation Pricinciple (ISP)* und *Dependency Inversion Principle (DIP)* zusammen. Diese Prinzipien zielen darauf ab, ein gutes objektorientiertes Design zu fördern. Sie wurden von Experten publiziert, darunter Robert C. Martin, Bertrand Meyer und Barbara Liskov. Robert C. Martin prägte das Akronym SOLID, um diese Prinzipien zu vereinen. [Wik]

Das SRP lautet: „Es sollte nie mehr als einen Grund geben, eine Klasse zu ändern." [MMM06] Eine Klasse, die mehrere Gründe für die Änderungen aufweist, verletzt das SRP, da dies darauf hinweist, dass sie mehrere unterschiedliche Aufgaben erfüllt und somit mehrere Verantwortlichkeiten besitzt. Dies führt zu einem erhöhten Risiko von Fehlern, da Änderungen in einer Verantwortlichkeit versehentlich eine andere Verantwortlichkeit beeinträchtigen können. [Gol18]

Das OCP stammt von Bertrand Meyer und lautet: „Module sollten sowohl offen (für Erweiterungen) als auch verschlossen (für Modifikationen) sein". [Mey97] Konkret bedeutet das OCP, dass es möglich sein sollte, Klassen zu erweitern, ohne dabei deren Verhalten zu ändern. Dies kann beispielsweise durch die Vererbung ermöglicht werden, weil die Basisklasse erweitert werden kann, ohne den bereits vorhandenen und lauffähigen Code dieser Klasse zu verändern. [Gol18]

Das LSP von Barbara Liskov fordert, dass eine abgeleitete Klasse nahtlos die Rolle ihrer Basisklasse einnehmen können sollte. Häufig wird das LSP dadurch verletzt, dass eine überschreibende Methode in einer abgeleiteten Klasse entweder keine Funktionalität oder eine eingeschränkte Funktionalität im Vergleich zur entsprechenden Methode der Basisklasse aufweist. [Gol18]

Beim ISP von Robert C. Martin wird gefordert, dass Schnittstellen in kleinere und spezifischere Schnittstellen aufgespaltet werden sollten, anstatt sie unnötig groß zu gestalten. Auf diese Weise erhalten die Clients nur die Schnittstelle, die sie tatsächlich benötigen und verwenden. [Gol18]

Robert C. Martin hat das DIP wie folgt formuliert: „A. Module hoher Ebenen sollten nicht von Modulen niedriger Ebenen abhängen. Beide sollten von Abstraktionen abhängen. B. Abstraktionen sollten nicht von Details abhängen. Details sollten von Abstraktionen abhängen". [MMM06] Verständlicher bedeutet dies, dass eine Klasse auf einer höheren Ebene nicht direkt von einer Klasse auf einer tieferen Ebene abhängig sein sollte und die höherliegende Klasse soll eine Abstraktion ihrer benötigten Services vorgeben.

2.3.2 *DRY-Prinzip*

Das *DRY*-Prinzip wurde von Dave Thomas und Andy Hunt formuliert. [HT03] Dieses Prinzip legt nahe, dass jegliche Form der Redundanz konsequent vermieden werden sollte. Das Ziel besteht darin, das Risiko zu minimieren, dass bei späteren Änderungen eine oder mehrere Kopien übersehen werden. Durch die Einhaltung des DRY-Prinzips wird das Potenzial für Fehler aufgrund von Inkonsistenzen verringert und es ist nicht erforderlich, dieselbe Logik mehrfach zu testen. Dies wiederum führt zu einer Reduzierung des Codeumfangs, verbessert die Lesbarkeit und erleichtert die Wartbarkeit. [Gol18]

2.4 QMOOD

Jagdish Bansiya und Carl G. Davis haben ein hierarchisches Modell namens QMOOD zur Bewertung von Qualitätsattributen in objektorientierten Entwürfen entwickelt und beschrieben. Dieses Modell besteht aus vier Ebenen (Abbildung 2.1):

- Qualitätsattributen (quality attributes)
- Entwurfseigenschaften (design properties)
- Metriken (metrics) und
- objektorientierten Designkomponenten.

Das Modell schafft Verknüpfungen zwischen Entwurfseigenschaften wie Kapselung, Modularität, Kopplung und Kohäsion sowie hochwertigen Qualitätsattributen wie Wiederverwendbarkeit, Flexibilität und Komplexität. Die Messung der Entwurfseigenschaften erfolgt mithilfe der Metriken.

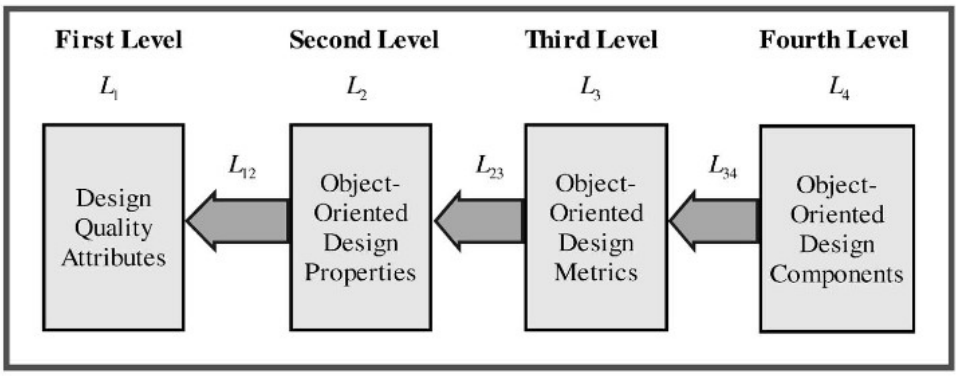

Abbildung 2.1: Ebenen und Verknüpfungen in QMOOD [BD02]

Durch die dreifache Verbindung zwischen den vier Ebenen entstehen Beziehungen und Verbindungen zwischen Entwurfseigenschaften und Qualitätsattributen. Diese Verbindungen werden entsprechend ihrer Auswirkung und Bedeutung gewichtet, wobei die Berechnungsformel in Abbildung 2.2

verwendet wird. Auf diese Weise werden Werte für die Qualitätsattribute Wiederverwendbarkeit, Flexibilität, Verständlichkeit, Funktionalität, Erweiterbarkeit und Effektivität ermittelt. Die jeweilige Bedeutung dieser Attribute wurde in Abbildung 2.3 zusammengefasst. [BD02]

Quality Attribute	Index Computation Equation
Reusability	-0.25 * Coupling + 0.25 * Cohesion + 0.5 * Messaging + 0.5 * Design Size
Flexibility	0.25 * Encapsulation - 0.25 * Coupling + 0.5 * Composition + 0.5 * Polymorphism
Understandability	-0.33 * Abstraction + 0.33 * Encapsulation - 0.33 * Coupling + 0.33 * Cohesion - 0.33 * Polymorphism - 0.33 * Complexity - 0.33 * Design Size
Functionality	0.12 * Cohesion + 0.22 * Polymorphism + 0.22 Messaging + 0.22 * Design Size + 0.22 * Hierarchies
Extendibility	0.5 * Abstraction - 0.5 * Coupling + 0.5 * Inheritance + 0.5 * Polymorphism
Effectiveness	0.2 * Abstraction + 0.2 * Encapsulation + 0.2 * Composition + 0.2 * Inheritance + 0.2 * Polymorphism

Abbildung 2.2: Berechnungsformeln für Qualitätsattribute [BD02]

Quality Attribute	Definition
Reusability	Reflects the presence of object-oriented design characteristics that allow a design to be reapplied to a new problem without significant effort.
Flexibility	Characteristics that allow the incorporation of changes in a design. The ability of a design to be adapted to provide functionally related capabilities.
Understandability	The properties of the design that enable it to be easily learned and comprehended. This directly relates to the complexity of the design structure.
Functionality	The responsibilities assigned to the classes of a design, which are made available by the classes through their public interfaces.
Extendibility	Refers to the presence and usage of properties in an existing design that allow for the incorporation of new requirements in the design.
Effectiveness	This refers to a design's ability to achieve the desired functionality and behavior using object-oriented design concepts and techniques.

Abbildung 2.3: Definition von Qualitätsattributen [BD02]

Die Werte der sechs Qualitätsattributmaße werden für jedes Projekt summiert, um den *Total Quality Index* (TQI) für das Projekt zu berechnen. Ein höherer TQI-Wert weist auf eine bessere Qualität des Projekts hin.

3

PAGE OBJECT MODEL UND SCREENPLAY PATTERN

In diesem Kapitel werden die Funktionsweisen der drei zu vergleichenden Entwurfsmuster im Detail präsentiert. Um die Unterschiede zu veranschaulichen, wird ein einfacher *Login*-Test als Beispiel implementiert.

3.1 VERWENDETE TOOLS FÜR DIE IMPLEMENTIERUNG

In dieser Arbeit werden drei verschiedene Entwurfsmuster hinsichtlich ihrer Gestaltung verglichen. Für die Oberflächentests kommen einheitliche Tools zum Einsatz: *Serenity BDD*, *Selenium*, *Cucumber*, *REST Assured*, *Maven* und *Java*.

Serenity BDD ist eine Open-Source-Bibliothek, um automatisierte Tests schneller und wartbarer zu gestalten. Ein großer Vorteil der Verwendung von *Serenity BDD* liegt darin, dass es nicht erforderlich ist, Zeit in den Aufbau und die Wartung eines eigenen Automatisierungs-Frameworks zu investieren. *Serenity BDD* bietet außerdem eine enge Integration mit *Selenium*, *REST Assured* und *Cucumber*. [BDDa]

Zusätzlich werden aussagekräftige Berichte von *Serenity BDD* erstellt, die nicht nur darüber informieren, welche Anforderungen getestet wurden, sondern auch wie sie getestet wurden. Diese Berichte enthalten detaillierte Informationen zum Ablauf eines Tests, inklusive optionaler Screenshots, um den Testvorgang anschaulich zu dokumentieren und darzulegen, was genau in der Anwendung passiert. [BDDa]

Für alle Entwurfsmuster wurde eine *User*-Klasse (Abbildung 3.1) für den Benutzer und die *UserPersonas*-Klasse (Abbildung 3.3) implementiert. In dem Ordner ‚testdata' sind die Zugangsdaten in der Datei ‚user.conf' gespeichert (Abbildung 3.2). So kann durch den Namen des Benutzers dessen E-Mail, Passwort und ID gefunden werden.

```
@Data
@RequiredArgsConstructor
public class User {
    private final String email;
    private final String password;
    private final String userId;
}
```

Abbildung 3.1: *User*-Klasse

```
Bob: {
  email: "bob.beyer@accso.de"
  password: "bobspassword"
  userId: "459"
}

Yagmur: {
  email: "yagmur.celik@accso.de"
  password: "yagmurspassword"
  userId: "102"
}
```

Abbildung 3.2: *‚user.conf'*-Datei

```
public class UserPersonas {
    1 usage
    private static Config users
            = ConfigFactory.load( resourceBasename: "testdata/users");

    4 usages  Piao
    public static User findByName(String name) {
        Config userDetails = users.getConfig(name);
        return new User(
                userDetails.getString( path: "email"),
                userDetails.getString( path: "password"),
                userDetails.getString( path: "userId")
        );
    }
}
```

Abbildung 3.3: *UserPersonas*-Klasse

3.2 DAS KLASSISCHE POM UND DESSEN PROBLEME

Michel Nass, Emil Alégroth und Robert Feldt haben eine systematische Literaturrecherche durchgeführt und untersucht, warum die Herausforderungen bei den automatisierten Oberflächentests immer noch vorhanden sind und warum einige dieser Herausforderungen wahrscheinlich auch in Zukunft bestehen bleiben. Unter den 17 Veröffentlichungen zwischen 2001 und 2018 wurde am häufigsten berichtet, dass Änderungen an dem UI in der Anwendung zum Scheitern von Tests führen. [NAF21]

Wenn die Tests ohne die Anwendung von einem Entwurfsmuster implementiert wurden, ergeben sich häufig zahlreiche Anpassungen an den unterschiedlichen Teststellen, wenn auch nur geringfügige Änderungen an dem UI durchgeführt werden. Mit dem klassischen POM wird versucht, diese Herausforderungen abzumildern. Ein *Page Object* enthält sowohl *Locators* von Webelementen als auch die dazugehörigen Methoden. Diese Elemente und Methoden werden in den fachlichen Testfällen genutzt, um eine Interaktion

mit dem UI zu ermöglichen. Im Falle einer Veränderung vom UI erfordert es lediglich eine Anpassung des Codes im jeweiligen *Page Object*.

Abbildung 3.4 zeigt die Implementierung des Testszenario für den *Login*-Test, Abbildung 3.5 die Implementierung vom *Page Object* für die *Login*-Seite. Jede spezifische Seitenklasse (z.B. *LoginPage*) erbt von der *PageObject*-Klasse, die *Serenity BDD* anbietet. Die *PageObject*-Klasse fügt automatisch eine Instanz des *Webdrivers* in die Seitenklasse ein und mittels der Annotation *@FindBy* kann das jeweilige Webelement anhand seiner ID oder XPath identifiziert werden.

```
Scenario: Login successfully
  When User logs in with a valid username and password
  Then User should be given access to his account
```

Abbildung 3.4: ‚login.feature' im klassischen POM

```
public class LoginPage extends PageObject {
    1 usage
    @FindBy(id="email-input")
    public WebElement email;

    1 usage
    @FindBy(id="password-input")
    public WebElement password;

    1 usage
    @FindBy(xpath="//*[contains(text(), 'Anmelden')]")
    public WebElement loginButton;
    1 usage
    Navigate navigate;

    3 usages  Piao
    public void userLogsIn (String emailString, String passwordString) {
        navigate.toTheLoginPage();
        email.sendKeys(emailString);
        password.sendKeys(passwordString);
        loginButton.click();
    }
```

Abbildung 3.5: *LoginPage*-Klasse

Während der Implementierung der einzelnen *Step Definitions* in den Tests können die Methoden der entsprechenden Seitenklasse verwendet werden, ähnlich wie in Abbildung 3.6.

```
@When("User logs in with a valid username and password")
public void userLogsInWithEmailAndPassword () {
    User user = UserPersonas.findByName("Bob");
    loginPage.userLogsIn(user.getEmail(), user.getPassword());
}
```

Abbildung 3.6: *Step Definition* für *Login*

Ein Hauptvorteil vom klassischen POM besteht darin, dass die Testcodes in *Step Definitions* und der seitenspezifische Code voneinander getrennt sind. Dadurch wird es einfacher, Anpassungen an der Anwendung vorzunehmen, da bei Anwendungsänderungen lediglich die Seitenklasse angepasst werden muss, anstatt die Tests selbst zu modifizieren. [Sel23] Zum Beispiel muss nur die *LoginPage*-Klasse angepasst werden, wenn die ID eines Webelements geändert wurde. Die *Step Definitions* selbst bleiben unverändert.

Wenn allerdings eine Seite umfangreich und komplex ist, wird die Seitenklasse groß, weil sie sowohl die *Locators* als auch die entsprechenden Methoden enthält. Entsprechend hat diese Klasse auch zwei Verantwortungen, die Webelemente zu identifizieren und die entsprechende Methode für die Interaktion zu beschreiben. Deshalb verstößt die Klasse gegen das SRP von den SOLID-Prinzipien, denn die Klasse hat zwei Verantwortungen und deshalb auch zwei Gründe für die Veränderung. [Mar+16c]

Ein weiteres Problem besteht in der Code-Duplizierung. Angenommen, auf einer Seite befinden sich zehn Buttons, die getestet werden sollen. So benötigt diese für jeden Button eine Methode, um ihn anzuklicken, obwohl der Inhalt der Methoden bis auf das spezifische Webelement exakt identisch ist.

3.3 DAS ERWEITERTE POM

Das erweiterte POM zielt darauf ab, das Problem der Code-Duplizierung, das im klassischen POM vorhanden ist, abzuschwächen. Innerhalb dieses Entwurfsmusters werden die generischen Webelemente zusammen mit den erforderlichen Methoden ausgelagert, damit sie erneut genutzt werden können.

3.3.1 *Struktur und Funktionsweise*

Die abstrakten Klassen *AbstractWebelement* und *AbstractWebelementList* fungieren als fundamentale Grundlage für sämtliche generische Webelemente, die diese erweitern. *AbstractWebelement* repräsentiert das einzelne Webelement und *AbstractWebelementList* repräsentiert die Listen von Webelementen. Die vereinfachte Struktur von *AbstractWebelement* nimmt die nachfolgende Form an (Abbildung 3.7).

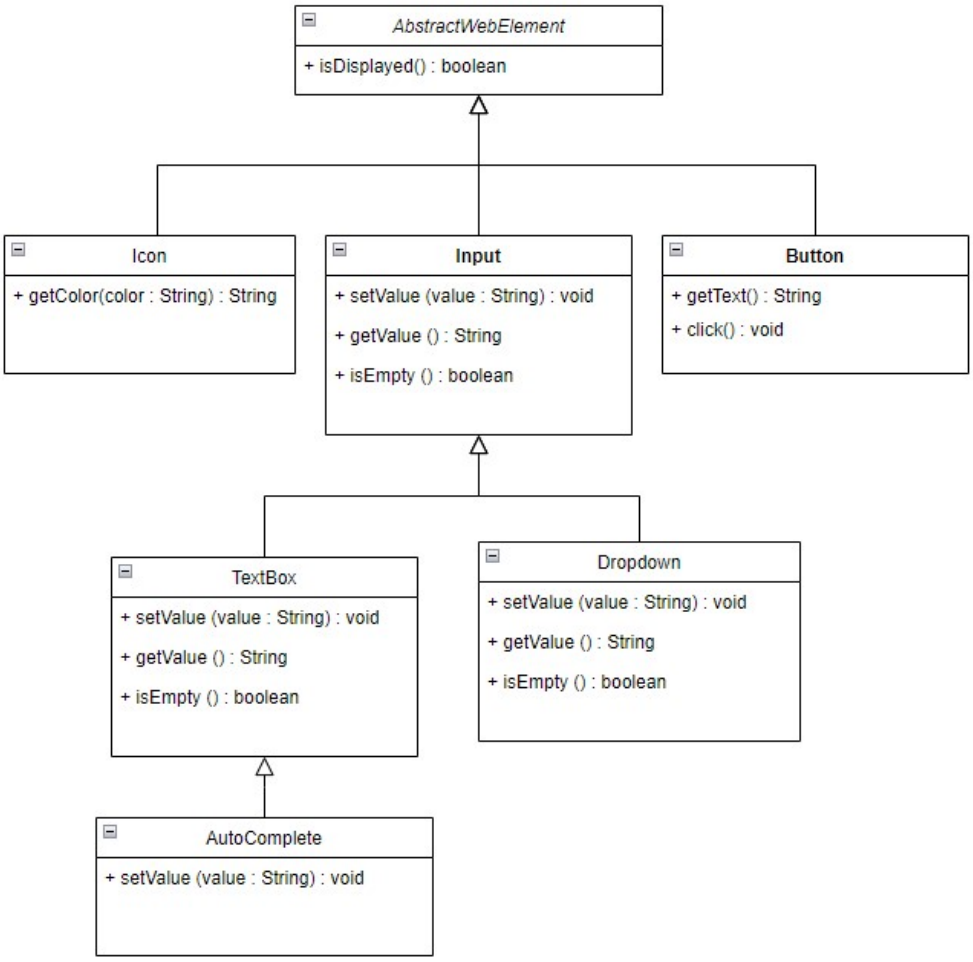

Abbildung 3.7: Struktur von generischen Webelementen

Die Speicherung der *Locators* von Webelementen und der Methoden für deren Interaktionen erfolgt getrennt voneinander. Zu diesem Zweck sind zwei zusätzliche abstrakte Klassen vorhanden, die *AbstractPage*-Klasse und die *AbstractStep*-Klasse.

Die *AbstractPage*-Klasse, enthält zwei *Hash Maps* und eine Funktion *saveWebelementsInHashMap()* (Abbildung 3.8). In diesen *Hash Maps* werden entweder einzelne Webelemente oder Listen der Webelemente gespeichert. Die Funktion dient dazu, dass die Webelemente von einem *Page Object* durch *Java Reflection* in einer *Hash Map* gespeichert werden. Der Schlüssel der *Hash Map* ist der Name des Webelements.

```
public abstract class AbstractPage {
    5 usages
    public Map<String, AbstractWebElement> webElementMap = new LinkedHashMap<>();
    3 usages
    public Map<String, AbstractWebElementList> webElementListMap = new LinkedHashMap<>();

    5 usages  Piao *
    public void saveWebelementsInHashMap(PageObject classname) {
        Arrays.stream(classname.getClass().getDeclaredFields())
                .filter((field) -> field.getType().getName()
                .equals(WebElement.class.getName())
            && field.getAnnotation(RepresentedBy.class) != null)
                .forEach((field) -> {
                    try {
                        webElementMap.put(field.getName(),
                                (AbstractWebElement) field.
                                  getAnnotation(RepresentedBy.class).classname()
                                  .getConstructor(WebElement.class).
                                  newInstance(field.get(classname)));
                    } catch (InstantiationException | IllegalAccessException
                             | InvocationTargetException
                             | NoSuchMethodException e) {
                        throw new RuntimeException(e);
                    }
                });
```

Abbildung 3.8: *AbstractPage*-Klasse

Eine *Pages*-Klasse ist von der *AbstractPage*-Klasse abgeleitet. Innerhalb dieser Klasse werden sämtliche Elemente von verschiedenen *Page Objects* in einer *Hash Map* gespeichert. Es sei angemerkt, dass *Dropdown*-Menüs und Textfelder mit *Autocomplete*-Funktion zwei Parameter im Konstruktor erfordern, wodurch das Webelement manuell in der *Hash Map* abgelegt werden muss (Abbildung 3.9).

```
public class Pages extends AbstractPage {
    1 usage  Piao *
    public Pages(LoginForm loginForm, ExpenseAccountForm expenseAccountForm,
                 HolidayForm holidayForm, WorkingHoursForm workingHoursForm,
                 BookedHoursForm bookedHoursForm) {
        saveWebelementsInHashMap(loginForm);
        saveWebelementsInHashMap(expenseAccountForm);
        saveWebelementsInHashMap(holidayForm);
        saveWebelementsInHashMap(workingHoursForm);
        saveWebelementsInHashMap(bookedHoursForm);
        webElementMap.put("autoCompleteSearchField",
                new AutoComplete(expenseAccountForm.dropDownSearchField,
                        expenseAccountForm.dropDownOptions));
        webElementMap.put("dropDownSearchField",
                new DropDown(expenseAccountForm.dropDownSearchField,
                        expenseAccountForm.dropDownOptions));
    }
```

Abbildung 3.9: *Pages*-Klasse

In der *TestBase*-Klasse werden zuerst alle *Page Objects* durch *Serenity BDD* initialisiert. Die Übergabe der *Page Objects* erfolgt im Konstruktor der *Pages*-

Klasse, wodurch die Speicherung aller Webelemente in *Hash Maps* ermöglicht wird. (Abbildung 3.10)

```
public abstract class TestBase {
    3 usages
    protected Navigate navigate;
    1 usage
    protected LoginForm loginForm;
    1 usage
    protected ExpenseAccountForm expenseAccountForm;
    1 usage
    protected HolidayForm holidayForm;
    1 usage
    protected WorkingHoursForm workingHoursForm;
    1 usage
    protected BookedHoursForm bookedHoursForm;
    3 usages
    protected GetDateAPI getDateAPI;
    4 usages  Piao *
    public Pages getPages() {
        return new Pages(loginForm, expenseAccountForm,
                holidayForm, workingHoursForm,
                bookedHoursForm);
    }
```

Abbildung 3.10: *TestBase*-Klasse

Die abstrakte Klasse *AbstractStep* erbt von der *TestBase*-Klasse und beinhaltet zwei Funktionen, *getWebElement()* und *getWebElements()*. Mit deren Hilfe ist es möglich, anhand des Namens ein bestimmtes Webelement oder eine Liste von Webelementen in der *Hash Map* zu identifizieren (Abbildung 3.11). Deshalb erweitert jede *Steps*-Klasse die *AbstractStep*-Klasse.

```
public abstract class AbstractSteps extends TestBase {
    17 usages  Piao
    protected AbstractWebElement getWebElement(String WebElementName) {
        getPages().webElementMap.get(WebElementName).waitUntilElementAppears();
        return getPages().webElementMap.get(WebElementName);
    }
    13 usages  Piao
    protected AbstractWebElementList getWebElements(String WebElementName) {
        getPages().webElementListMap.get(WebElementName).waitUntilElementsAppear();
        return getPages().webElementListMap.get(WebElementName);
    }
```

Abbildung 3.11: *AbstractSteps*-Klasse

3.3.2 *Ein Beispiel*

Das Testszenario für den *Login*-Test in der *Feature*-Datei könnte wie in Abbildung 3.12 gestaltet sein. Im Gegensatz zum klassischen POM wird der Name des Webelements in der *Feature*-Datei als Parameter übergeben, da dieser als Schlüssel in der *Hash Map* fungiert.

```
Scenario: Login successfully
  Given User visits login page in CAT
  And User enters the value bob.beyer@accso.de in the email field
  And User enters the value bobspassword in the password field
  When User clicks on loginButton
  Then zeiterfassungTab should be displayed
```

Abbildung 3.12: ‚login.feature' im erweiterten POM

Zur Erfassung der E-Mail-Adresse und des Passworts wird die Funktion *setValue()* der *TextBox*-Klasse aufgerufen. (Abbildung 3.13) Innerhalb dieser Funktion wird die *sendKeys()*-Methode von *Selenium* genutzt, um einen Text in ein Eingabefeld einzufügen. (Abbildung 3.14)

```
public class TextBoxSteps extends AbstractSteps {
    5 usages  Piao
    @And("User enters the value {word} in the {word} field")
    public void EnterTheValueInTheField(String value, String textBoxWebelementName) {
        ((TextBox) getWebElement(textBoxWebelementName)).setValue(value);
    }
}
```

Abbildung 3.13: *Step Definition* für die Eingabe in das Textfeld

```
@Override
public void setValue(String text) {
    self.sendKeys(text);
    self.sendKeys(Keys.TAB);
}
```

Abbildung 3.14: *setValue()*-Funktion von *TextBox*-Klasse

Um den *Login*-Button zu klicken, wird die Methode *click()* der *Button*-Klasse aufgerufen. (Abbildung 3.15)

```
public class ButtonSteps extends AbstractSteps {
    1 usage  Piao
    @And("User clicks on {word}")
    public void ClickOn(String buttonWebElementName) {
        ((Button) getWebElement(buttonWebElementName)).click();
    }
```

Abbildung 3.15: *Step Definition*, um die Buttons zu klicken

Im Gegensatz zum klassischen POM können diese beiden Methoden wiederverwendet werden. Das bedeutet, um beispielsweise zehn Buttons auf einer Seite zu testen, müssen nicht zehn neue Methoden geschrieben werden. Es genügt, den entsprechenden Namen des Webelements in der *Feature*-Datei als Parameter zu übergeben, um den jeweiligen Button anzuklicken. Dadurch wird die Code-Duplizierung erheblich reduziert.

Durch die Aufteilung der Interaktionsklassen entsprechend der Webelement-Typen ergibt sich ein weiterer Vorzug: eine erhöhte Klarheit in der Struktur. Jede Klasse beinhaltet lediglich diejenigen Methoden, die für das jeweilige Webelement relevant sind. Ein konkretes Beispiel hierfür ist die Implementierung der Funktion *ClickOn()*, die in der Klasse *ButtonSteps* umgesetzt wird. Aufgrund der Möglichkeit zur Wiederverwendung dieser Methode bleibt die Klasse dadurch frei von überflüssiger Komplexität und Unübersichtlichkeit.

3.4 SCREENPLAY PATTERN

Das *Screenplay* hat in den letzten Jahren zunehmend an Beliebtheit gewonnen, obwohl seine grundlegenden Konzepte schon seit einiger Zeit existieren. Der Ursprung des Entwurfsmusters geht zurück auf das Jahr 2007 beim *Agile Alliance Functional Testing Tools workshop (AAFTT)*, in dem Antony Marcano das Modell *Rollen, Ziele, Aufgaben, Aktionen* vorstellte. Ende 2012 entwickelten Antony Marcano, Andy Palmer und Jan Molak zusammen das sogenannte *screenplay-jvm*, um Herausforderungen und Einschränkungen vom POM anzugehen und die SOLID-Prinzipien auf Akzeptanztests anzuwenden. Im Jahr 2015 arbeiteten John Ferguson Smart und Jan Molak eng mit Andy Palmer und Antony Marcano zusammen, um das *Screenplay* in *Serenity BDD* zu integrieren. Dieser Schritt trug dazu bei, das *Screenplay* in der Java-Test-Community bekannt zu machen und seine Verbreitung zu fördern. [SM23]

Das *Screenplay* ist ein Entwurfsmuster und keine Bibliothek, deshalb gibt es unterschiedliche Implementierungsvarianten, z.B. *Serenity/JS (JavaScript/-TypeScript), ScreenPy (Python), Boa Constrictor (.NET)* und *Cucumber Screenplay (JavaScript/TypeScript)*. [SM23] In dieser Arbeit wird die *Serenity BDD*-Implementierung von *Screenplay (Java)* verwendet.

3.4.1 *Struktur und Funktionsweise*

Das *Screenplay* ist ein benutzerzentrierter Ansatz, der zu Tests führt, die sauber, leichter verständlich und wartbar sind. Wie der Name sagt: Jedes Testszenario ist wie eine Art Drehbuch strukturiert. Ein Drehbuch beschreibt, wie die *Actors* ihre Aktionen ausführen sollen, während sie mit dem zu testenden System interagieren. [Ser] Das Entwurfsmuster besteht aus fünf Elementen: *Actor*, *Interaction*, *Ability*, *Question* und *Task*.

Der *Actor* repräsentiert die Benutzer. Die Benutzer interagieren auf unterschiedliche Weise mit einer Anwendung, um die spezifischen *Tasks* zu erledigen. Mehrere *Tasks* bilden eine *Interaction* mit dem zu testenden System. Darüber hinaus besitzen *Actors* die *Abilities*, die bei der Ausführung dieser *Tasks* behilflich sind. Die am häufigsten benötigte *Ability* ist es, mit einem Webbrowser zu interagieren. *Actors* können auch *Questions* zum aktuellen Zustand des Systems beantworten, um das zu erwartende Verhalten einer bestimmten Funktion zu überprüfen (Abbildung 3.16). [BDDb]

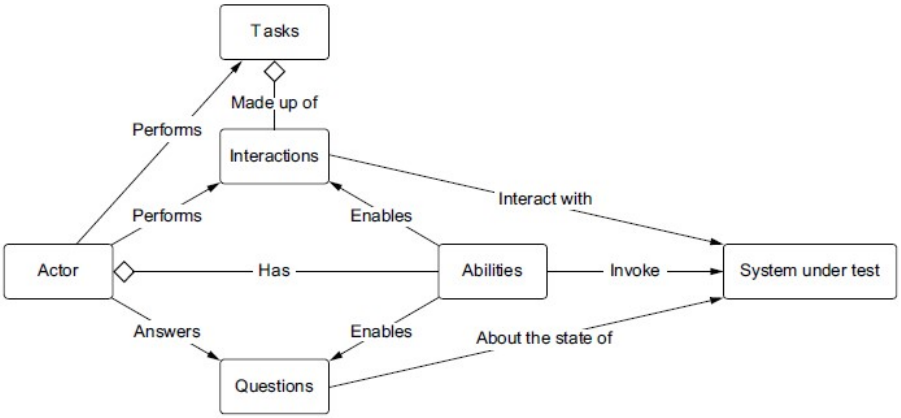

Abbildung 3.16: Struktur vom *Screenplay* [SM23]

3.4.2 *Ein Beispiel*

Bei der Implementierung des *Login*-Tests mithilfe vom *Screenplay* könnte das Testszenario wie in Abbildung 3.17 dargestellt aussehen.

```
Scenario: Login successfully
  When Bob logs in with a valid username and password
  Then He should be given access to his account
```

Abbildung 3.17: ‚login.feature' im *Screenplay*

Aufgrund der benutzerzentrierten Herangehensweise vom *Screenplay* beginnt der Test stets mit einem *Actor*. Um den *Login*-Test zu verfassen, ist der erste Schritt das Öffnen eines Webbrowsers. Dies stellt sicher, dass der *Actor*

mit der zu testenden Anwendung interagieren kann. In dem Kontext vom *Screenplay* wird dieser Schritt als *Ability* bezeichnet. Es ist notwendig, dem *Actor* zuerst die entsprechende *Ability* zuzuweisen, um eine *Interaction* mit der Anwendung zu ermöglichen (Abbildung 3.18).

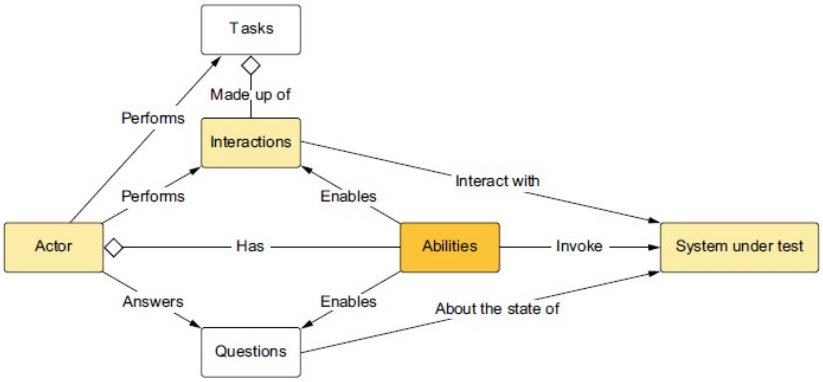

Abbildung 3.18: Struktur vom *Screenplay* (*Abilities*) [SM23]

Bei der Implementierung eines Tests für einen einzelnen Benutzer kann die Zuweisung von *Abilities* an einen *Actor* so wie in der Abbildung 3.19 am unkompliziertesten umgesetzt werden. Durch die Verwendung von *Actor.named()* kann ein neuer *Actor* erstellt werden. Hierbei ist im Rahmen vom *Screenplay* die *Ability*-Klasse *BrowseTheWeb* vorhanden, um mit einem Browser zu interagieren. Allerdings wurde hier der Benutzername ‚*Toby*' im Code festgelegt. Sollte mehr als ein *Actor* getestet werden oder *Interactions* zwischen mehreren *Actors* in einem Testszenario erforderlich sein, sollte die Möglichkeit bestehen, den Namen des *Actors* direkt in der *Feature*-Datei zu übergeben.

```
Actor toby = Actor.named("Toby");
new *
@Before
public void setUpActor() {
    toby.can(BrowseTheWeb.with(Serenity.getDriver()));
}
new *
@Given("Toby visits login page")
public void visitLoginPage() {
    toby.attemptsTo(
            Open.url( targetUrl: "https://example.com")
    );
}
```

Abbildung 3.19: Zuweisung von *Abilities* an einen *Actor*

Das *Screenplay* bietet sich als eine einfache Möglichkeit dafür an. Ein *Cast* ist eine Klasse, die die Aufgabe hat, *Actors* bestimmte *Abilities* zur Verfügung zu stellen. Als Beispiel kann die *OnlineCast*-Klasse genutzt werden, um *Ac-*

tors bereitzustellen, die mit ihren eigenen *Webdriver*-Instanzen ausgestattet sind (Abbildung 3.20).

```
public class ParameterTypes {
    Piao
    @Before
    public void setTheStage() {
        OnStage.setTheStage(new OnlineCast());
    }

    Piao
    @ParameterType(".*")
    public Actor actor(String actorName) {
        return OnStage.theActorCalled(actorName);
    }

    Piao
    @ParameterType(".*")
    public User user(String userName) {
        return UserPersonas.findByName(userName);
    }
```

Abbildung 3.20: Zuweisung von *Abilities* an mehrere *Actors*

Auf einer *Stage* nehmen die *Actors* ihre Rollen ein. Ein Testszenario wird als eine *Stage* betrachtet, in der ein *Actor* anhand seines Namens identifiziert wird. Bei Beginn jedes Szenarios muss der *Stage* ein *Cast* zugeordnet werden. Hierfür kann die Methode *OnStage.setTheStage()* aufgerufen werden, wobei ein bestimmter *Cast* übergeben wird. Nachdem die *Stage* vorbereitet ist, kann ein Actor mithilfe der Methode *OnStage.theActorCalled()* anhand seines Namens abgerufen werden. [BDDb]

Zusätzlich dazu besteht die Möglichkeit, den Benutzer mittels eines benutzerdefinierten *Parametertyps* automatisch in einen *Actor* umzuwandeln. Hierfür kann die Annotation *@ParameterType* von *Cucumber* verwendet werden. Durch die Verwendung von ‚.* ' wird jede beliebige Zeichenfolge akzeptiert. Mithilfe dieser Annotation kann ein eigens definierter *ParameterType* (in diesem Fall *actor*) an die *Step Definition* übergeben und automatisch in seinen entsprechenden Typ (*Actor*) umgewandelt werden (Abbildung 3.21).

```
@When("{actor} logs in with a valid username and password")
public void logsInWithAValidUsernameAndPassword(Actor actor) {
    User user = UserPersonas.findByName(actor.getName());
    actor.attemptsTo(Login.as(user));
}
```

Abbildung 3.21: *Step Definition* für *Login*

Wie bereits zu Beginn dieses Kapitels verdeutlicht wurde, sind die E-Mail-Adresse und das Passwort eines Benutzers in der *User*-Klasse hinterlegt. Daher sollte der *Actor* zunächst in einen Benutzer umgewandelt werden.

Die *OnStage*-Klasse erkennt ebenfalls Pronomen, darunter *he, she, they, it, his, her, their, its*. (Abbildung 3.17) Die Methode *theActorCalled()* identifiziert das Pronomen und ruft den zuletzt aktiven *Actor* mithilfe der Methode *theActorInTheSpotlight()* ab. [BDDb]

Im *Screenplay* werden die *Interactions* als Objekte anstelle von Methoden repräsentiert und jede *Interaction* implementiert eine spezielle *Screenplay*-Schnittstelle *Performable* (Abbildung 3.22). Um eine *Interaction* mit der Anwendung zu ermöglichen, wird die Methode *actor.attemptsTo()* aufgerufen. Diese Methode akzeptiert entweder ein einzelnes Objekt oder eine Liste von Objekten, die den Rückgabetyp *Performable* aufweisen.

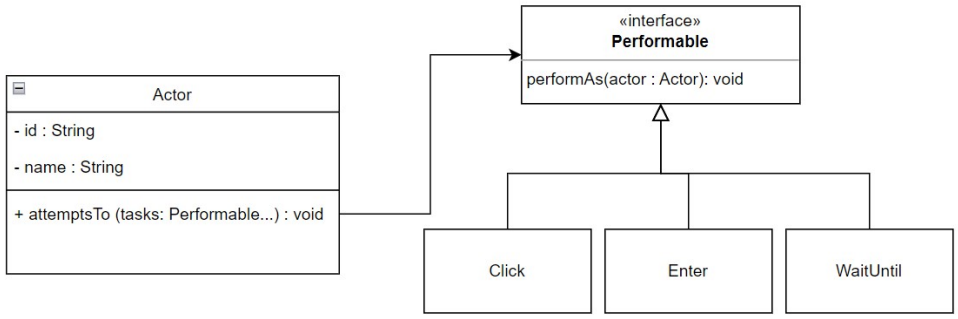

Abbildung 3.22: Struktur von *Interaction*-Klassen

Dieser Ansatz hat den Vorteil, dass mehrere *Tasks* flexibel als eine *Interaction* zusammengeführt werden können, wie die Struktur vom *Screenplay* in Abbildung 3.16 zeigt. Ein Beispiel hierfür ist in Abbildung 3.23 illustriert, in dem verschiedene *Tasks* in der Funktion *as()* innerhalb der *Login*-Klasse ausgelagert und aggregiert werden. Das *Screenplay* bietet die Methode *Task.where()*, die die Modellierung einer Liste von *Tasks* als eine einzige *Interaction* erlaubt. Da diese Funktion den Rückgabetyp *Performable* hat, kann sie mithilfe von *actor.attemptsTo()* aufgerufen werden.

```
public class Login {
    Piao *
    public static Performable as(User user) {
        return Task.where( title: "{0} logs in as " + user.getEmail(),
                Navigate.toTheLoginPage(),
                Enter.keyValues(user.getEmail()).into(LoginForm.EMAIL),
                Enter.keyValues(user.getPassword()).into(LoginForm.PASSWORD),
                Click.on(LoginForm.LOGIN_BUTTON).afterWaitingUntilEnabled()
        );
    }
}
```

Abbildung 3.23: *as()*-Funktion in *Login*-Klasse

Die Zusammenführung dieser *Tasks* in einer einzigen Funktion trägt zur Verbesserung der Lesbarkeit des Testcodes bei. In den *Step Definitions* ist le-

diglich der Ausdruck *actor.attemptsTo(Login.as(user))* zu sehen, was anschaulich und leicht verständlich ist.

Im Gegensatz zum klassischen und erweiterten POM, das sich in der Lösungssprache ausdrückt (z.B. Seiten, UI-Elemente), ermöglicht das *Screenplay* die Beschreibung von Vorgängen in der Sprache des Unternehmens. Jede *Interaction* beschreibt, was der *Actor* in geschäftlicher Hinsicht auszuführen hat. Aus diesem Grund harmoniert das *Screenplay* besonders gut mit einem BDD-Ansatz.

Darüber hinaus erlaubt es die Funktion *as()*, unterschiedliche Benutzer leicht wiederzuverwenden, da die *Login*-Logik unverändert bleibt. Diese Eigenschaft erweist sich als nützlich, insbesondere wenn eine Webanwendung für verschiedene Benutzer unterschiedliche Benutzeroberflächen aufweist. Damit können die verschiedenen Rollen mit unterschiedlichen Berechtigungen abgebildet werden. Im *Screenplay* ist es unkompliziert, diverse Benutzerparameter in der *Feature*-Datei zu verwenden, während dies im klassischen oder erweiterten POM aufwendiger wäre.

Die *Locators* von Webelementen werden in einer eigenständigen Klasse ausgelagert und mithilfe von *PageElement* kann ein Element auf einer Webseite identifiziert werden. Dies trägt zur Einhaltung des SRP von den SOLID-Prinzipien bei, da jede Klasse lediglich eine einzige Verantwortung besitzt (Abbildung 3.24).

```
public class LoginForm {
    1 usage
    public static final Target EMAIL = InputField.withNameOrId("email-input");
    1 usage
    public static final Target PASSWORD = InputField.withNameOrId("password-input");
    1 usage
    public static final Target LOGIN_BUTTON = Button.containingText("Anmelden");
}
```

Abbildung 3.24: *LoginForm*-Klasse

Außerdem ermöglicht die Verwendung von *PageElement* eine intuitivere Identifizierung der Webelemente, da *Serenity BDD* verschiedene *PageElement*-Typen mit den jeweils erforderlichen Methoden bereitstellt. Dadurch kann die Verwendung von XPath- oder CSS-Ausdrücken vermieden werden. [BDDb] Im gezeigten Beispiel in Abbildung 3.24 sind *PageElement*-Typen wie *InputField* und *Button* verfügbar, von denen jeder Typ über unterschiedliche Methoden wie *containingText()* verfügt. Im Gegensatz dazu wird in *Selenium* der Button mithilfe von XPath-Ausdrücken (Abbildung 3.25) identifiziert. Es ist deutlich erkennbar, dass die Verwendung von *PageElement* eine verbesserte Lesbarkeit im Vergleich zu reinem *Selenium* bietet.

```
@FindBy(xpath="//*[contains(text(), 'Anmelden')]")
public WebElement loginButton;
```

Abbildung 3.25: Ein Beispiel von XPath-Ausdrücken in *Selenium*

Der abschließende Aspekt der *Screenplay*-Struktur sind die *Questions*. Im *Screenplay* werden diese für die *Assertion* verwendet, was einen bedeutenden Teil des Testens darstellt. Der *Actor* kann mithilfe seiner *Abilities* auf die *Questions* bezüglich des Systemzustands antworten. Dies ermöglicht, den tatsächlichen Zustand mit den erwarteten Ergebnissen abzugleichen (Abbildung 3.26).

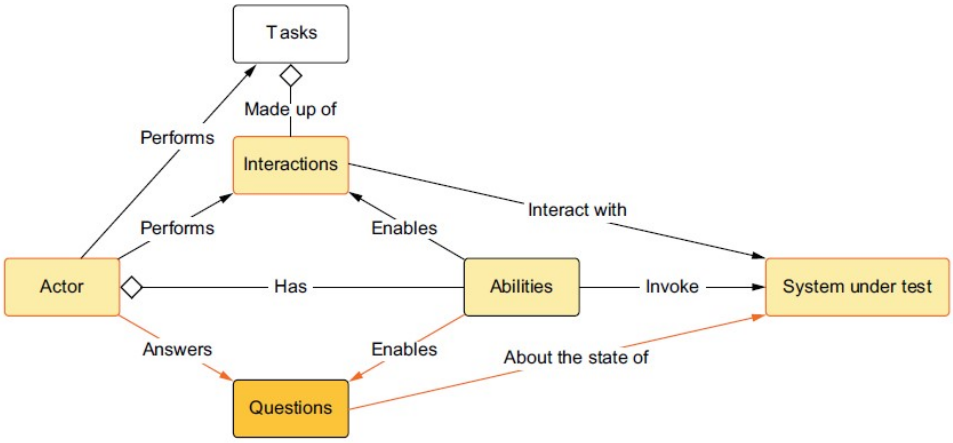

Abbildung 3.26: Struktur vom *Screenplay* (*Questions*) [SM23]

Ein Beispiel hierfür (Abbildung 3.27) zeigt sich in einem der beiden Schritte, bei dem die aktuellen HTML-Attribute des entsprechenden Webelements abgefragt werden können.

```
actor.asksFor(Attribute.of
        (BookedHoursForm.WORKING_HOURS_ICON, attributeName: "class")
);
Attribute.of(iconLocator, attributeName: "class").answeredBy(actor);
```

Abbildung 3.27: Beispiel für *Questions*

Zusätzlich bietet das *Screenplay* eine *Ensure*-Klasse, um die *Assertions* einfacher und lesbarer zu gestalten. Diese Klasse besitzt den Rückgabetyp *Performable* und kann daher ebenfalls in *attemptsTo()* aufgerufen werden. Mit der *Ensure*-Klasse können *Assertions* in Form von *Questions* oder anderen Typen (z. B. *Target* oder *String*) erleichtert werden.

In Abbildung 3.28 wird die *Assertion* mithilfe einer *Question* ausgelöst, während in Abbildung 3.29 die *Assertion* über ein *Target* erfolgt. Für jeden dieser Typen stehen unterschiedliche *Assertion*-Methoden zur Verfügung. Diese Flexibilität macht die Durchführung von *Assertion* im *Screenplay* anpassbar, einfach und gut lesbar. [SM23]

```
@And("{actor} should see that the color of worked hours icon is {word}")
public void colorOfWorkedHoursIconShouldBe (Actor actor, String color) {
    actor.attemptsTo(
            Ensure.that(BookedHoursForm.WORKING_HOURS_ICON)
                    .attribute( attributeName: "class").contains(color)
    );
}
```

Abbildung 3.28: *Assertion* über eine *Question*

```
@Then("{actor} should be given access to his account")
public void heShouldBeGivenAccessToHisAccount(Actor actor) {
    actor.attemptsTo(
            WaitUntil.the(ExpenseAccountForm.ZEITERFASSUNG_TAB, isVisible())
                    .forNoMoreThan( amount: 10).seconds(),
            Ensure.that(ExpenseAccountForm.ZEITERFASSUNG_TAB).isDisplayed()
    );
}
```

Abbildung 3.29: *Assertion* über ein *Target*

4

VERGLEICH DER ENTWURFSMUSTER AM BEISPIEL DER CAT-ANWENDUNG

In diesem Abschnitt erfolgt ein umfassender Vergleich der drei Entwurfsmuster anhand verschiedener Kriterien. Zunächst wird die interne Plattform der Accso CAT vorgestellt, die für die Untersuchung verwendet wurde, sowie die für diese Plattform implementierten Testszenarien. Anschließend werden die drei Entwurfsmuster anhand der fünf Vergleichskriterien im Detail gegenübergestellt. Zum Schluss werden die Ergebnisse des Vergleichs präsentiert und die Auswahl des geeigneten Entwurfsmusters anhand des Projektkontexts zusammengefasst.

4.1 CAT-ANWENDUNG UND DIE TESTSZENARIEN

CAT unterstützt verschiedene unternehmensrelevante Prozesse, wie das Arbeitszeitmanagement, das Urlaubsmanagement für Mitarbeiter, die Teamsteuerung, Projektsteuerung und allgemeine Unternehmensprozesse. Die Plattform ist in zwei Hauptteile unterteilt: Zeiterfassung und Controlling.

Da die Zeiterfassung täglich von jedem Mitarbeiter verwendet wird, ist es von entscheidender Bedeutung sicherzustellen, dass zumindest die wichtigsten Funktionen stets fehlerfrei funktionieren. Aus diesem Grund ist die Implementierung von Oberflächentests für die Zeiterfassung sinnvoll. Zudem beinhaltet die Zeiterfassung verschiedene Webkomponenten, die in anderen Anwendungen ebenfalls häufig vorkommen – wie Textfelder, Buttons und *Dropdown*-Menüs. Daher konzentriert sich diese Arbeit ausschließlich auf die Durchführung von Oberflächentests für die Zeiterfassung (Abbildung 4.1).

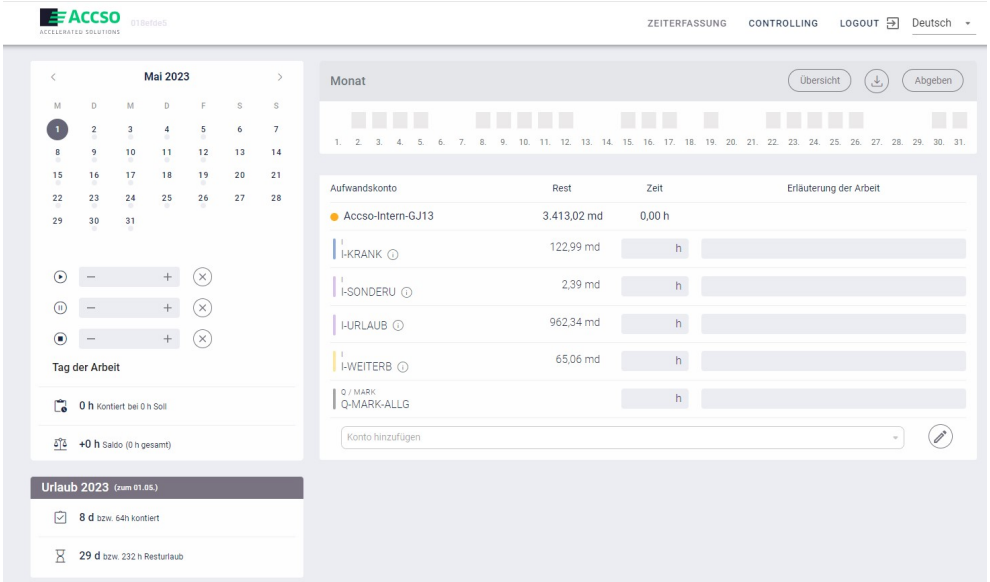

Abbildung 4.1: Zeiterfassung von der CAT-Anwendung

Insgesamt wurden acht Testszenarien entwickelt, die die Basisfunktionalität der Zeiterfassung abdecken und von jedem Mitarbeiter genutzt werden. Alle Testszenarien simulieren einen sogenannten ‚*Happy Path*', bei dem ein Codeabschnitt ohne Ausnahmen fehlerfrei ausgeführt werden soll. Diese acht Testszenarien für die Zeiterfassung sehen wie folgt aus:

1. Ein Benutzer kann sich mit gültiger E-Mail-Adresse und Passwort erfolgreich in CAT einloggen.

2. Wenn ein Benutzer krank ist, kann er Stunden auf das Konto ‚I-KRANK' buchen. Dabei sollten das versteckte Krankheits-Icon und die erfasste Zeit angezeigt werden (Abbildung 4.2).

3. Wenn ein Benutzer Sonderurlaub hat, kann er Stunden auf das Konto ‚I-SONDERU' buchen. Dabei sollten das versteckte Sonderurlaubs-Icon und die erfasste Zeit angezeigt werden (Abbildung 4.2).

4. Ein Benutzer kann seine Arbeitszeiten und Pausenzeiten eingeben und auf ein entsprechendes Konto, z. B. ‚I-WEITERB', buchen, um seine Arbeitszeit zu erfassen (Abbildung 4.2).

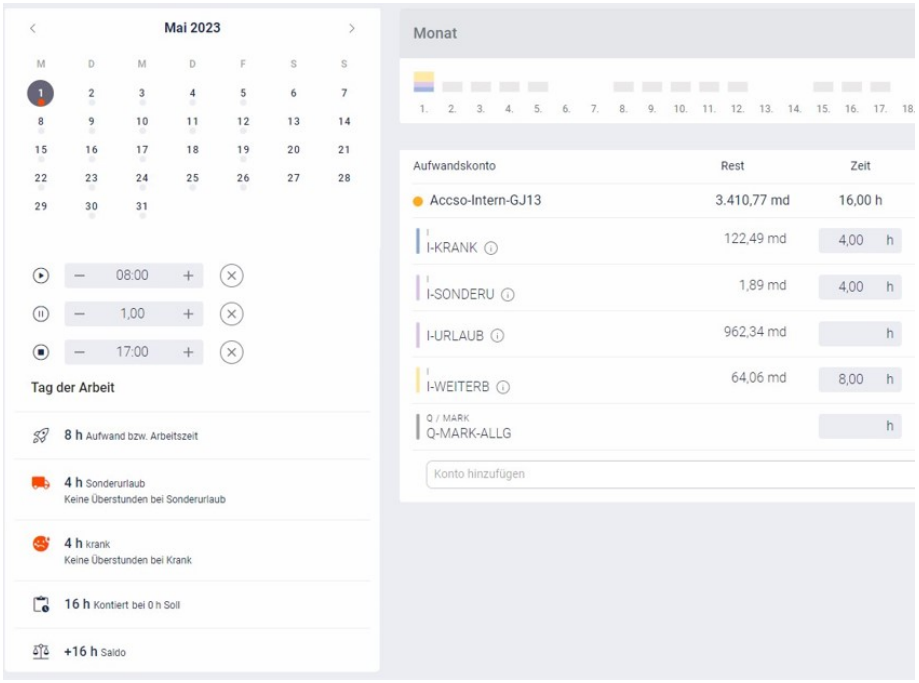

Abbildung 4.2: Krankheits-Icon, Sonderurlaubs-Icon und Icon der kontierten Arbeitszeit in der CAT-Anwendung

5. Wenn ein Benutzer im Urlaub ist, kann er Stunden auf das Konto ‚I-URLAUB' buchen. Dabei sollten das versteckte Urlaubs-Icon, die erfasste Zeit und die aktualisierten Zahlen im Urlaubsfeld angezeigt werden (Abbildung 4.3).

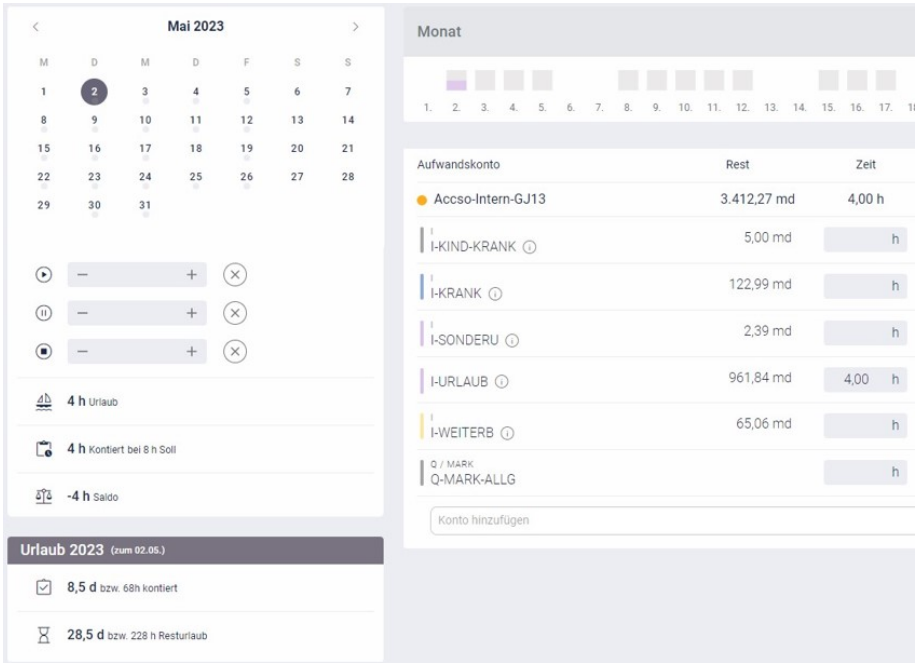

Abbildung 4.3: Urlaubs-Icon und Urlaubsfeld in der CAT-Anwendung

6. Ein Benutzer kann ein *Dropdown*-Menü öffnen und auf ein Aufwandskonto klicken, um es in seiner Favoritenliste zu speichern.

7. Ein Benutzer kann den Namen eines Aufwandskontos in das *Dropdown*-Menü-Feld eingeben und die erste gefundene Option auswählen, um sie in seine Favoritenliste zu speichern.

8. Wenn ein Benutzer beispielsweise Stunden auf das Konto ‚I-URLAUB' bucht, sollte ein zweiter Benutzer sehen können, dass der Restaufwand für dieses Konto entsprechend reduziert wurde.

Im anschließenden Abschnitt werden diese acht Testszenarien anhand verschiedener Kriterien verglichen. Für einige der Testszenarien ist es erforderlich, bestimmte Daten vorab zu löschen oder hinzuzufügen. Ein Beispiel hierfür ist die Überprüfung der Arbeitszeiterfassung. In diesem Fall müssen alle Textfelder für Arbeitszeit und Aufwandskonto leer sein, da andernfalls der Saldo fehlerhaft angezeigt wird und die Tests fehlschlagen. Ebenso: Um zu testen, ob ein Konto zur Favoritenliste hinzugefügt werden kann, sollte das betreffende Konto vor dem Test nicht auf der Liste stehen. Falls es sich bereits auf der Liste befindet, muss es gelöscht werden.

In dieser Arbeit werden derartige Schritte mittels *REST-API*-Anfragen implementiert und dienen als Vorbedingungen für die Tests. Um die Vergleichbarkeit nicht zu beeinflussen, werden sämtliche REST-API-Anfragen mit *RestAssured* exakt gleich umgesetzt. Im *Screenplay* wird der Begriff *User* durch den Namen des *Actors* ersetzt (Abbildung 4.4), da jeder Schritt mit einem *Actor* beginnen soll. Hierdurch wird ermöglicht, dass verschiedene und beliebig viele Benutzer die REST-API-Anfrage aufrufen können. Dies ist im Vergleich zum klassischen POM und erweiterten POM komplexer.

```
Given Bob clears all input field values of first day in zeiterfassung
```

Abbildung 4.4: Eine *REST-API*-Anfrage im *Screenplay*

```
And User deletes a test account in zeiterfassung from his expense account list
```

Abbildung 4.5: Eine *REST-API*-Anfrage im klassischen und erweiterten POM

4.2 VERGLEICHSKRITERIEN

4.2.1 *Lesbarkeit der Feature-Datei*

In der Dokumentation von *Cucumber* werden Ansätze zur Verbesserung der Schreibweise mit *Gherkin* vorgestellt. Es wird empfohlen, die Testszenarien zur Beschreibung des Systemverhaltens (Was) zu nutzen, anstatt auf die Implementierungsdetails (Wie) einzugehen. *Gherkin* legt nahe, dass die *Feature*-Datei unverändert bleibt, selbst wenn sich die Implementierung ändert. Änderungen an der Implementierung sollten lediglich im Hintergrund erfolgen. [Cucc]

Aus diesem Grund empfiehlt es sich, die Testszenarien im deklarativen Stil zu verfassen. Dieser fokussiert sich auf das Verhalten der Anwendung.

Die deklarativen Testszenarien sind leicht lesbar und können als eine Art Testdokumentation angesehen werden. Im Gegensatz dazu sind imperative Testszenarien stark an spezifische Implmentierungsdetails gebunden. Bei Änderungen an der Implementierung müssen auch die Tests entsprechend angepasst werden. [Cucc]

Wenn das Beispiel aus Kapitel 3 erneut betrachtet wird, wird der *Login*-Test sowohl im klassischen POM als auch im *Screenplay* (Abbildung 3.4 und Abbildung 3.17) im deklarativen Stil umgesetzt. Die Details darüber, wie der Benutzer mit dem System interagiert, bleiben verborgen und werden in den *Step Definitions* angegeben. Hingegen verwendet das erweiterte POM den imperativen Stil, bei dem jeder Schritt als genaue Anleitung dargestellt wird (Abbildung 3.12).

Sowohl beim klassischen POM als auch beim *Screenplay* erfolgen Bemühungen, Änderungen an der *Feature*-Datei zu vermeiden, wenn die Implementierung angepasst wird. Das erweiterte POM hingegen erzielt genau das Gegenteil. Hierbei ist es erforderlich, den Schlüssel der *Hash Map* als Parameter an die *Step Definition* zu übergeben, um die entsprechende Funktionalität aufzurufen. Bei Änderungen an der Implementierung sollte lediglich die *Feature*-Datei angepasst werden, da die *Step Definitions* wiederverwendet werden können.

Da die *Step Definitions* für unterschiedliche Webelemente implementiert wurden, müssen die Schritte in der *Feature*-Datei im imperativen Stil verfasst und in kleine Schritte aufgeteilt werden, um die Wiederverwendung der *Step Definitions* zu ermöglichen.

4.2.2 *Test mit mehreren Benutzern*

Das *Screenplay* zeichnet sich durch seinen benutzerzentrierten Ansatz aus, wodurch eine bemerkenswerte Flexibilität und Einfachheit bei der Interaktion unterschiedlicher Benutzer in einem Testszenario erreicht wird. Um die Interaktionen mehrerer Benutzer zu simulieren, wurde ein entsprechendes Testszenario (Testszenario 8) entwickelt. In diesem kontiert ein Benutzer namens *Bob* acht Stunden auf dem Konto ‚I-WEITERB'. Der zweite Benutzer namens *Yagmur* soll daraufhin sehen können, dass der Restaufwand des Kontos ‚I-WEITERB' um eins reduziert wurde. (Abbildung 4.6) Dies resultiert daraus, dass der Restaufwands in Personen-Tagen angezeigt wird und acht Stunden einem Personen-Tag entsprechen.

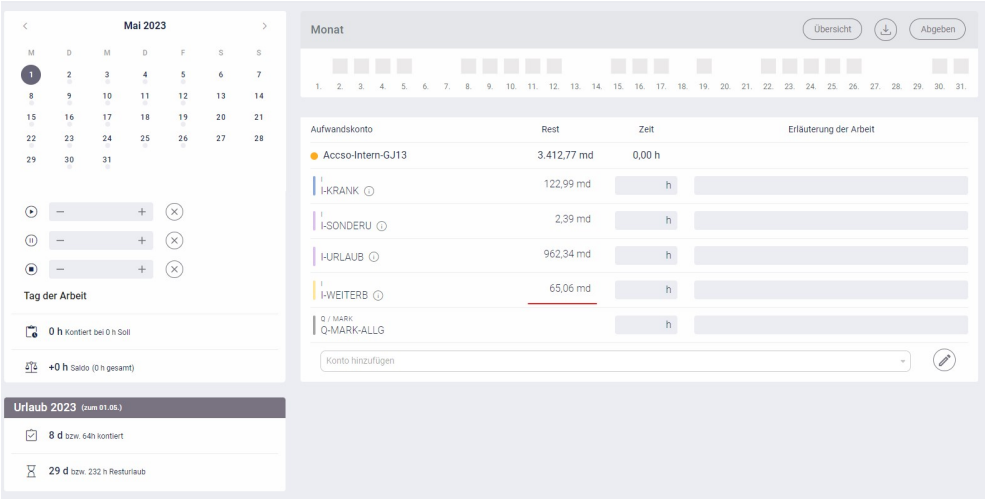

Abbildung 4.6: Restaufwand in der CAT-Anwendung

Sowohl das klassische POM als auch das erweiterte POM implementieren das gleiche Testszenario mit denselben Schritten. Der einzige Unterschied liegt darin, dass das klassiche POM einen deklarativen Stil verwendet, während das erweiterte POM einen imperativen Ansatz nutzt, wie bereits in Abschnitt 4.2.1 erläutert wurde (Abbildung 4.7 und Abbildung 4.8).

```
Scenario: The value of remaining effort changes simultaneously for two users
  And The first window should be saved
  And The old value of remaining effort of the account i-weiterb should be saved
  And User2 clears all input field values of first day in zeiterfassung
  And User2 adds an expense account for working hours in zeiterfassung
  And User2 logs in and gets access to his account
  And User switches to first window
  When User books the working hours 8 under account i-weiterb
  Then User should see that remaining effort of the account i-weiterb has been subtracted by 1
  And User switches to second window
  And User refreshes the page
  And User should see that remaining effort of the account i-weiterb has been subtracted by 1
  And User clears all input field values of first day in zeiterfassung
  And User2 clears all input field values of first day in zeiterfassung
```

Abbildung 4.7: Testszenario mit mehreren Benutzern im klassichen POM

```
Scenario: The value of remaining effort changes simultaneously for two users
  And The first window should be saved
  And User finds the i-weiterb field from the list labelListAufwandkonto and saves its remaining effort
  And User2 clears all input field values of first day in zeiterfassung
  And User2 adds an expense account for working hours in zeiterfassung
  And User2 logs in and gets access to his account
  And User switches to first window
  When User finds the i-weiterb field from the list labelListAufwandkonto and enters the value 8 into t
  Then User finds the i-weiterb field from the list labelListAufwandkonto and remaining effort from the
  And User switches to second window
  And User refreshes the page
  And User finds the i-weiterb field from the list labelListAufwandkonto and remaining effort from the
  And User clears all input field values of first day in zeiterfassung
  And User2 clears all input field values of first day in zeiterfassung
```

Abbildung 4.8: Ausschnitt des Testszenarios mit mehreren Benutzern im erweiterten POM

Mit dem *Screenplay* kann das Testszenario wie in Abbildung 4.9 gestaltet werden.

```
Scenario: The value of remaining effort changes simultaneously for two users
  And The old value of remaining effort of the account i-weiterb should be saved for Bob
  And Yagmur clears all input field values of first day in zeiterfassung
  And Yagmur adds an expense account for working hours in zeiterfassung
  And Yagmur logs in and gets access to his account
  And The old value of remaining effort of the account i-weiterb should be saved for Yagmur
  When Bob books the working hours 8 under account i-weiterb
  Then Bob should see that remaining effort of the account i-weiterb has been subtracted by 1
  And Yagmur refresh the page
  And Yagmur should see that remaining effort of the account i-weiterb has been subtracted by 1
  And Bob clears all input field values of first day in zeiterfassung
  And Yagmur clears all input field values of first day in zeiterfassung
```

Abbildung 4.9: Testszenario mit mehreren Benutzern im *Screenplay*

4.2.2.1 *Lesbarkeit des Testszenarios*

Die Testszenarien mit dem klassischen POM und erweiterten POM gehen implizit davon aus, dass nur ein *Actor* mit dem System interagiert. Wenn jedoch zwei Benutzer in einem Testszenario interagieren, führt dies zu Duplikationen im Code.

Beispielweise müssten für den zweiten Benutzer auch dieselben Methoden für REST-API-Abfragen zur Datenlöschung implementiert werden, obwohl die Implementierung bis auf den Benutzernamen exakt gleich ist (Abbildung 4.10). Um die Wiederverwendung der REST-API-Abfragen zu ermöglichen, könnte die Methode ähnlich dem Vorgehen im *Screenplay* umgestaltet werden. Hierbei würde der Benutzername als Parameter an die Methode übergeben werden. In dieser Arbeit wurden eigenständige REST-API-Abfragen für zwei Benutzer im klassischen POM und im erweiterten POM implementiert. Dies verdeutlicht die Unterschiede zwischen dem *Screenplay* und den beiden anderen Entwurfsmustern.

```
@Given("User adds an expense account for working hours in zeiterfassung")
public void AddOneExpenseAccountForWorkingHoursInZeiterfassung() {
    LocalDate date = getDateAPI.getFirstDayFromTheZeiterfassung( userName: "Bob");
    addOrDeleteAccountApi.AddOneAccountForWorkingHoursInZeiterfassung( userName: "Bob",
            date, accountIdWeiterBildung);
}
1 usage   Piao *
@Given("User2 adds an expense account for working hours in zeiterfassung")
public void AddOneExpenseAccountForWorkingHoursInZeiterfassungForUser2() {
    LocalDate date = getDateAPI.getFirstDayFromTheZeiterfassung( userName: "Yagmur");
    addOrDeleteAccountApi.AddOneAccountForWorkingHoursInZeiterfassung( userName: "Yagmur",
            date, accountIdWeiterBildung);
}
```

Abbildung 4.10: REST-API Anfragen im klassischen und erweiterten *POM*

Außerdem ermöglicht das *Screenplay* eine klare Identifizierung des aktuellen Benutzers, der momentan mit dem Browser interagiert (Abbildung 4.11). Im Gegensatz dazu beginnt jede *Step Definition* im klassischen POM und im erweiterten POM mit dem generischen Begriff *User*, anstatt den Namen des entsprechenden *Actor*s zu verwenden. Aus diesem Grund ist das Testszenario im klassichen POM und im erweiterten POM verwirrend, da es schwierig ist, eindeutig zu erkennen, welcher Benutzer aktuell mit dem Browser interagiert (Abbildung 4.12).

```
When Bob books the working hours 8 under account i-weiterb
Then Bob should see that remaining effort of the account i-weiterb has been subtracted by 1
And Yagmur refresh the page
And Yagmur should see that remaining effort of the account i-weiterb has been subtracted by 1
```

Abbildung 4.11: Ausschnitt von Abbildung 4.9

```
When User books the working hours 8 under account i-weiterb
Then User should see that remaining effort of the account i-weiterb has been subtracted by 1
And User switches to second window
And User refreshes the page
And User should see that remaining effort of the account i-weiterb has been subtracted by 1
```

Abbildung 4.12: Ausschnitt von Abbildung 4.7

4.2.2.2 *Wechseln zwischen mehreren Browsern*

Da das Testszenario die Interaktion von zwei Benutzern umfasst, sollte es möglich sein, zwischen den beiden Browsern zu wechseln.

Um ein neues Fenster zu öffnen, kann die Methode *getWindowHandle()* von *Selenium* verwendet werden. Im klassischen POM und im erweiterten POM müssen alle geöffneten Fenster gespeichert werden. Zudem ist es erforderlich, in einer *for*-Schleife nach dem Identifikator des entsprechenden Fensters zu suchen, um zum gesuchten Fenster zu wechseln. (Abbildung 4.13 und Abbildung 4.14). [Sel]

```
@And("The first window should be saved")
public void firstWindowShouldBeSaved() {
    firstWindow = Serenity.getDriver().getWindowHandle();
}
```

Abbildung 4.13: *Step Definition*, um das erste Fenster zu speichern

```
@And("User switches to first window")
public void switchToFirstWindow() {
    for (String windowHandle : Serenity.getDriver().getWindowHandles()) {
        if(!firstWindow.contentEquals(windowHandle)) {
            secondWindow = windowHandle;
            Serenity.getDriver().switchTo().window(firstWindow);
            break;
        }
    }
    wait.until(ExpectedConditions.visibilityOfAllElements(
            expenseAccountPage.remainingEffortList)
    );
}
```

Abbildung 4.14: *Step Definition*, um zum ersten Fenster zu wechseln (*das klassische und erweiterte POM*)

Im *Screenplay* sind solche Schritte nicht notwendig, da *BrowseTheWeb* von der *Ability*-Klasse verwendet werden kann, um das Surfen im Internet zu ermöglichen.

Mit dem Schritt in Abbildung 4.15 ist es möglich, das Fenster des aktuellen Benutzers zu aktualisieren, ohne das erste oder zweite Fenster zu speichern oder zwischen ihnen wechseln zu müssen.

```
BrowseTheWeb.as(actor).getDriver().navigate().refresh();
```

Abbildung 4.15: *Step Definition*, um zum ersten Fenster zu wechseln (*Screenplay*)

4.2.2.3 *Zwischenspeicherung von Daten*

In diesem Testszenario sollte der alte Restaufwand zuerst gespeichert werden, um später überprüfen zu können, ob der Wert reduziert wurde. Im klassischen POM und im erweiterten POM wurde dieser Schritt so implementiert, dass der alte Wert in einer Variablen gespeichert und später mit dem aktuellen Wert verglichen wird. Wenn jedoch mehrere Benutzer interagieren müssen und der Wert je nach Benutzer unterschiedlich verändert wird, kann dies dazu führen, dass die entsprechende *Step Definition*-Klasse schnell unübersichtlich und umfangreich wird. Es würden viele Variablen benötigt, um die alten Werte für verschiedene Benutzer zu speichern.

Hier kommt die Funktion *remember()* im *Screenplay* ins Spiel, um Informationen in einem Schritt für den aktuellen *Actor* zu speichern. Durch die

Funktion *recall()* können diese Informationen in einem nachfolgenden Schritt wieder abgerufen werden. Diese Herangehensweise ermöglicht eine elegante und übersichtliche Verwaltung von Daten, insbesondere wenn mehrere Benutzer involviert sind und ihre jeweiligen Werte gespeichert und in einem nachfolgenden Schritt wieder abgerufen werden müssen. [BDDb]

4.2.3 *Wartbarkeit*

Um die Wartbarkeit der drei Entwurfsmuster zu vergleichen, wird eine Modifikation an dem UI von CAT vorgenommen. Im Anschluss wird untersucht, wie viele LoC für diese Änderung in den jeweiligen Entwurfsmustern angepasst werden müssen. Dieser Vergleich ermöglicht eine Einschätzung darüber, wie gut die verschiedenen Entwurfsmuster auf Änderungen in dem UI reagieren und wie leicht sie anpassbar und wartbar sind.

Hierfür wird angenommen, dass die drei Textfelder zur Eingabe der Arbeitszeit in zwei separate Felder aufgeteilt werden. (Abbildung 4.16 und 4.17) Diese Änderung hätte zur Folge, dass die bisherigen Testfälle (insbesondere das Testszenario 4) für die Arbeitszeitsbuchung nicht mehr ordnungsgemäß funktionieren würden. Die Trennung der Felder könnte zu Inkonsistenzen und Fehlern führen, da die bisherigen Tests auf die Struktur der drei Felder abgestimmt waren und die neuen Gegebenheiten nicht berücksichtigen. Dies verdeutlicht die Notwendigkeit, die Anpassungen an den Testfällen vorzunehmen, um sie auf das veränderte UI abzustimmen.

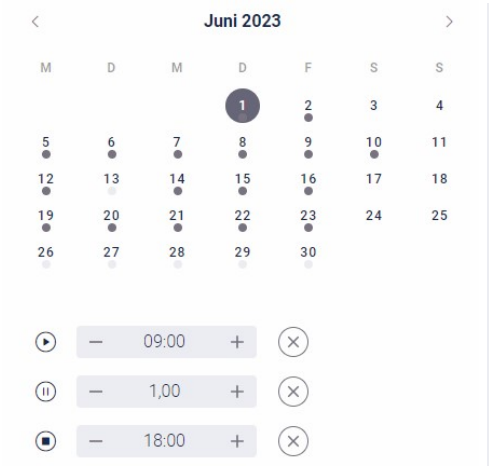

Abbildung 4.16: UI vor der Änderung

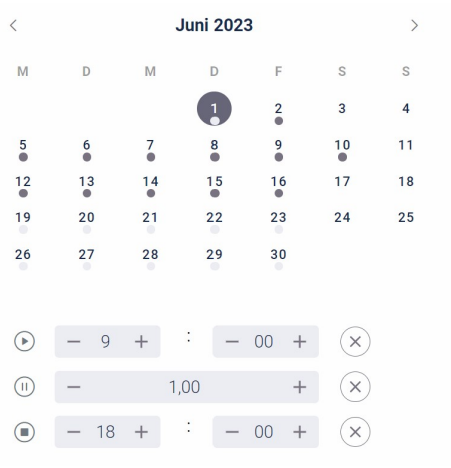

Abbildung 4.17: UI nach der Änderung

Die LoC, die in den jeweiligen Entwurfsmustern für diese Änderung angepasst werden müssen, sind in der Tabelle 4.1 zusammengefasst. Es ist deutlich erkennbar, dass das klassische POM die umfangreichsten Anpassungen erfordert. Der signifikante Unterschied besteht darin, dass das *Screenplay* und erweiterte POM deutlich weniger Änderungen in der Seitenklasse verlangen, da diese Klasse lediglich die Locators der Webelemente enthält. Zusammenfassend benötigt das erweiterte POM die geringsten Anpassungen.

Dies ist darauf zurückzuführen, dass bei diesem die *Step definitions* nicht wie im *Screenplay* angepasst werden müssen.

Klassenname	Working Hours Page	Feature Datei	Working Hours Steps	Summe
Klassisches POM	10	2	6	18
Screenplay	4	2	8	14
Erweiterets POM	4	4	0	8

Tabelle 4.1: Geänderte LoC im Vergleich

4.2.4 *Prinzipien der Softwareentwicklung*

4.2.4.1 *SOLID*

- Das klassische POM

  Im klassischen POM werden *Page Objects* oder Seitenklassen verwendet, die sowohl die Lokalisierung der Webelemente als auch die Interaktionsmethoden für diese Webelemente enthalten. Diese Vorgehensweise verstößt gegen das SRP und das OCP der SOLID-Prinzipien.

  Erstens übernimmt die Seitenklasse zwei Hauptverantwortlichkeiten: die Lokalisierung von Webelementen und die Interaktion mit ihnen. Dadurch ergeben sich zwei separate Gründe, die möglicherweise eine Änderung in dieser Klasse erfordern. Dies verletzt das SRP.

  Zweitens widerspricht das klassische POM dem OCP. Nehmen wir an, es soll ein *Dropdown*-Menü zu einer Seitenklasse hinzugefügt werden. Die konventionelle Vorgehensweise besteht darin, die Seitenklasse zu ‚öffnen' und eine neue Methode für das *Dropdown*-Menü zu implementieren. Um das OCP einzuhalten, sollte die Funktionalität des *Dropdown*-Menüs als eine neue Klasse hinzugefügt werden, ohne die bestehende Seitenklasse zu verändern. Auf diese Weise kann die Erweiterbarkeit des Codes gewährleistet werden, da neue Funktionen durch das Hinzufügen neuer Klassen erreicht werden können, ohne dabei die bereits vorhandenen Klassen zu ändern.

  Die anderen drei Prinzipien – ISP, LSP und DIP – werden in dieser Diskussion nicht behandelt, da das klassische POM eine eher unkomplizierte Struktur aufweist und keine Vererbung oder Schnittstellen verwendet werden.

- Erweitertes POM

  1. SRP

Im erweiterten POM werden die *Locators* der Webelemente in separaten Seitenklassen ausgelagert, wie bereits im Beispiel aus Kapitel 3 illustriert. Ebenso werden die Interaktionen für das jeweilige Webelement in eigenständigen Klassen ausgelagert und implementiert, wie in der Abbildung 4.18 dargestellt. Jede Seitenklasse ist somit für die Identifizierung der Webelemente verantwortlich, während die zugehörige Interaktionsklasse ausschließlich für die Interaktion mit einem spezifischen Element zuständig ist. Hierdurch wird das SRP erfüllt.

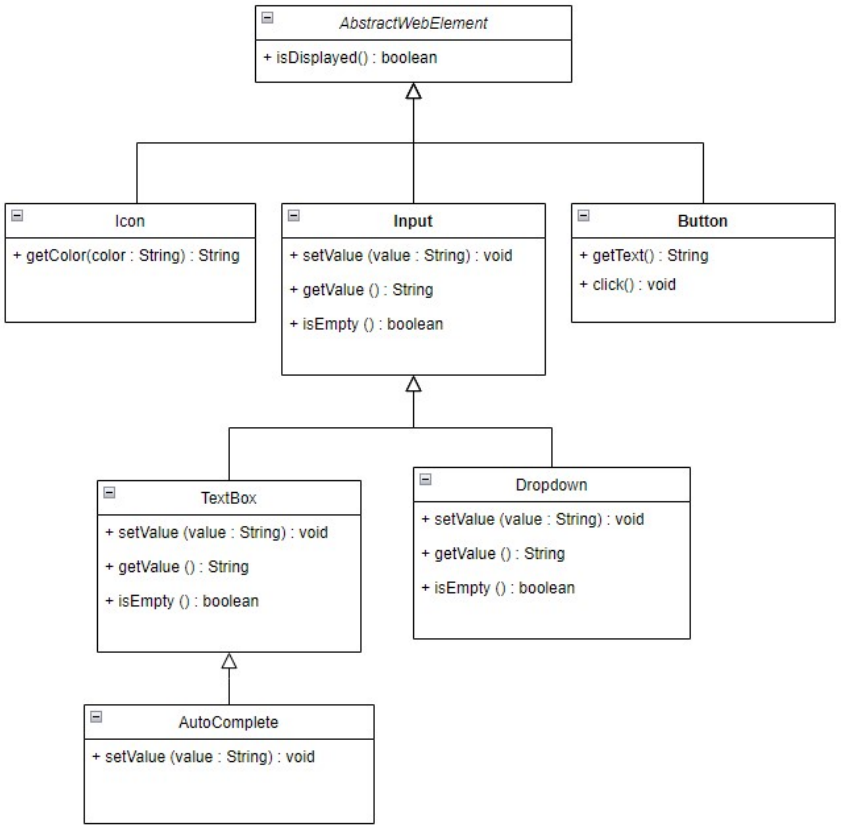

Abbildung 4.18: Struktur von generischen Webelementen

2. OCP

Abbildung 4.18 veranschaulicht, dass das erweiterte POM dem OCP entspricht. Die Interaktionen sind durch Vererbung implementiert, wodurch die *AbstractWebElement*-Klasse flexibel erweitert werden kann, ohne dass eine Änderung in dieser Klasse notwendig ist. Ein Beispiel hierfür wäre die Auslagerung der Interaktionen für ein *Dropdown*-Menü in eine separate Klasse.

Allerdings sollen im erweiterten POM die Webelemente zuerst in der *Hash Map* gespeichert werden. Dieser Prozess wird durch die *Pages*-Klasse (Abbildung 3.9) und die *TestBase*-Klasse (Abbildung 3.10) durchgeführt. Für jede neu hinzugefügte Seite sind Anpassungen an diesen beiden Klassen notwendig, um die Webelemen-

te der neuen Seite in der *Hash Map* abzulegen. Deshalb steht diese Vorgehensweise im Widerspruch zum OCP.

3. LSP

   Zur Einhaltung des LSP sollte es möglich sein, dass ein Typ problemlos durch seine Subtypen ersetzt werden kann. Die *Dropdown*-Klasse ist ein Subtyp der *Input*-Klasse und überschreibt alle geerbten Methoden der *Input*-Klasse. Demnach können beide Klassen problemlos gegeneinander ausgetauscht werden. Zusätzlich ist die *Input*-Klasse ein Subtyp der *AbstractWebElement*-Klasse und erbt die Methode *isDisplayed()*, was bedeutet, dass die *AbstractWebElement*-Klasse ohne Probleme durch die Input-Klasse ersetzt werden kann (Abbildung 4.18).

4. ISP

   Abbildung 4.18 zeigt, dass jede Unterklasse der *AbstractWebElement*-Klasse nur die Methoden enthält, die für das jeweilige Webseiten-Element tatsächlich benötigt werden. Zum Beispiel enthält die *Button*-Klasse die Methode *click()*, während die Input-Klasse über *setValue()* verfügt. Hierdurch wird das ISP gewahrt.

   Allerdings zeigt sich ähnlich wie beim OCP eine Verletzung vom ISP durch die *TestBase*-Klasse und die *Pages*-Klasse. Diese beiden Klassen speichern die Webelemente aller Seiten in den *Hash Maps*. Um das ISP einzuhalten, sollte es dem Client möglich sein, nur auf die Webelemente der jeweiligen Seitenklasse zuzugreifen, anstatt auf die Webelemente aller Seiten.

5. DIP

   Im erweiterten POM wird die Vererbung zur Umsetzung der Interaktion verwendet, wobei die Interaktionen als Methoden und nicht als Klassen wie im *Screenplay* vorliegen. Daher ergeben sich keine Abhängigkeiten von verschiedenen Klassen, was das DIP in diesem Kontext nicht zur Diskussion stellt.

- Screenplay Pattern

  Das *Screenplay* zeichnet sich durch eine klare Abgrenzung von *Interactions*, *Actors* und *Abilities* aus. Zusätzlich werden die *Locators* von Webelementen in separaten Klassen ausgelagert, wie im Beispiel in Kapitel 3 verdeutlicht wird.

  1. SRP

     Anstelle von den Methoden werden im *Screenplay* die Interaktionen als eigenständige Objekte behandelt. Wenn beispielsweise auf einen Button geklickt werden soll, kann die Klasse *Click* verwendet werden, oder die Klasse *Wait*, wenn auf das Erscheinen eines Webelements gewartet werden soll. Im klassischen und erweiterten POM hingegen werden in solchen Fällen typischerweise direkte Methoden wie *click()* oder *wait()* verwendet.

Für komplexere Interaktionen können mehrere einzelne *Tasks* in einer separaten Klasse kombiniert werden, um einen wiederverwendbaren *Interaction* zu erstellen. Dadurch wird sichergestellt, dass jede Klasse nur eine einzige Verantwortung trägt, was dem SRP der SOLID-Prinzipien entspricht.

2. OCP

   Das Beispiel aus dem klassischen POM kann erneut betrachtet werden, um das OCP zu veranschaulichen. Wenn das *Dropdown*-Menü getestet werden soll, kann im *Screenplay* eine neue Klasse namens *SelectDropDown* implementiert werden, wie bereits im vorherigen SRP-Abschnitt erwähnt. Diese Klasse kann dann in verschiedenen Seitenklassen wiederverwendet werden, die dasselbe *Dropdown*-Menü verwenden. Diese Vorgehensweise wahrt das OCP und fördert die Wiederverwendbarkeit von Code (Abbildung 4.19 und Abbildung 4.20).

```
@When("{actor} searches for account {word} in search field with a dropdown and clicks on it")
public void performSearchFieldWithDropdown(Actor actor, String accountName) {
    actor.attemptsTo(
            SelectDropdown.option(accountName).
                    byClicking(ExpenseAccountForm.SEARCH_FIELD,
                            ExpenseAccountForm.SEARCH_FIELD_OPTIONS)
    );
}
```

Abbildung 4.19: *Step Definition* für *Dropdown*-Menü

```
public class SelectDropdown {
    5 usages
    private String option;
    1 usage  Piao
    public SelectDropdown(String option) { this.option = option; }
    2 usages  Piao
    public static SelectDropdown option(String option) { return new SelectDropdown(option); }
    1 usage  Piao *
    public Performable byClicking(Target dropdownLocator, Target dropdownOptions) {
        return Task.where( title: "{0} selects  '" + option,
                actor -> {
                    actor.attemptsTo(
                            Click.on(dropdownLocator).afterWaitingUntilEnabled(),
                            WaitUntil.the(dropdownOptions, isVisible())
                                    .forNoMoreThan( amount: 10).seconds()
                    );
                    actor.attemptsTo(
                            Click.on(dropdownOptions.resolveAllFor(actor).stream()
                                    .filter((element) -> element.getElement().getText()
                                            .trim().equalsIgnoreCase(option))
                                    .findFirst().orElseThrow())
                                    .afterWaitingUntilEnabled()
                    );
                }
        );
    }
```

Abbildung 4.20: *SelectDropdown*-Klasse

3. LSP

   In Abbildung 4.21 wird deutlich, dass *ClickOnTarget*, *ClickOnElement* und *WaitUntilTargetIsReady* die Untertypen von *Performable* sind. Die Methode *performAs()* wird jeweils mit den gleichen Ein-

gabeparametern überschrieben. Daher ist es ohne Probleme möglich, *Performable* durch diese drei Untertypen zu ersetzen und somit das LSP zu wahren.

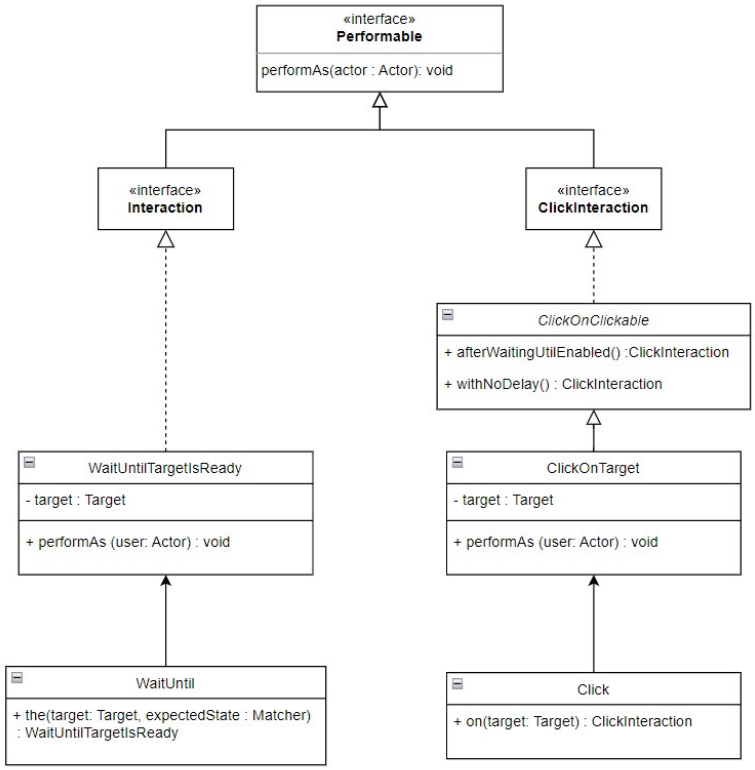

Abbildung 4.21: Struktur von *Click*-Klasse und *WaitUntil*-Klasse

4. ISP

   Hinsichtlich des ISP wird auch aus Abbildung 4.21 ersichtlich, dass sowohl die *Interaction* als auch die *ClickInteraktion* jeweils eigene eigenständige Schnittstellen besitzen. Zusätzlich verfügt die *Ability* auch über eine eigenständige separate Schnittstelle. Diese Unterteilung der Schnittstellen in kleinere Einheiten gewährleistet, dass jede Schnittstelle lediglich diejenigen Methoden enthält, die vom Client tatsächlich benötigt und aufgerufen werden. Hierdurch wird die strikte Einhaltung des ISP der Schnittstellensegregation sichergestellt.

5. DIP

   Im *Screenplay* werden *Interactions* als Klassen anstelle von Methoden verwendet. Ein *Actor* kann in der Methode *attemptsTo()* eine oder mehrere *Interactions* aufrufen. In Abbildung 4.22 wurde beispielsweise die Klasse *Enter* aufgerufen, um Text in ein Textfeld einzugeben.

```java
@Given("{actor} enters {word} as start time")
public void enterStartTime (Actor actor, String time) {
    actor.attemptsTo(
            Enter.keyValues(time).into(WorkingHoursForm.START_TIME_TEXTBOX)
    );
}
```

Abbildung 4.22: *Step Definition* für die Eingabe der Arbeitszeit

In diesem Kontext ist zu erkennen, dass die *Actor*-Klasse auf einer übergeordneten Abstraktionsebene liegt, während die *Interaction*-Klassen und die *Enter*-Klasse auf einer untergeordneten Abstraktionsebene positioniert sind und zwischen ihnen eine Abhängigkeitsbeziehung besteht. Das DIP fordert jedoch, dass Module auf höheren Abstraktionsebenen nicht unmittelbar von Modulen auf niedrigeren Ebenen abhängen sollen. Stattdessen sollten beide von Abstraktionen abhängig sein, um eine flexiblere Struktur zu ermöglichen.

Um das DIP zu erfüllen, wird im *Screenplay* die Abstraktion *Performable* eingeführt. Dadurch wird gewährleistet, dass die Module auf höheren Ebenen (wie die *Actor*-Klasse) von dieser Abstraktion abhängig sind, anstatt eine direkte Abhängigkeit von den konkreten Interaktionsklassen auf der niedrigeren Ebene zu haben.

Abbildung 4.23 verdeutlicht, dass sowohl die *Interaction*-Klassen wie *Click* und *Enter* als auch die *Actor*-Klasse von der abstrakten Schnittstelle *Performable* abhängig sind. Um die *Actor*-Klasse unabhängig von den konkreten Interaktionsklassen zu machen, kommt im *Screenplay* das Konzept der *Dependency Injection* zum Einsatz. Hierbei wird die Instanziierung des Interaktionsobjekts nicht von der *Actor*-Klasse selbst vorgenommen. Stattdessen erfolgt die Bereitstellung des Objekts über eine separate *Injector*-Klasse, die das entsprechende Objekt in die *Actor*-Klasse einspeist. Dadurch wird die Abhängigkeit zwischen der *Actor*-Klasse und den konkreten *Interaction*-Klassen gelockert und eine bessere Entkopplung erreicht.

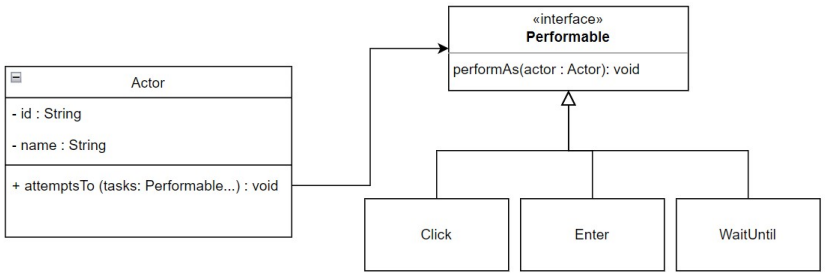

Abbildung 4.23: Struktur von *Interaction*-Klassen

Innerhalb der Methode *attemptsTo()* wird die Schnittstelle *Performable* an die Klasse *ActorPerforms* übergeben. Letztere stellt einen

Konstruktor für einen *Injector* bereit, der die Abhängigkeiten zur Laufzeit reguliert (Abbildung 4.24 und Abbildung 4.25).

```
public class ActorPerforms extends ActorPerformanceEvent {
    2 usages
    private final Performable performable;

    1 usage
    public ActorPerforms(Performable performable, String actor) {
        super(actor);
        this.performable = performable;
    }
```

Abbildung 4.24: *ActorPerforms*-Kalsse

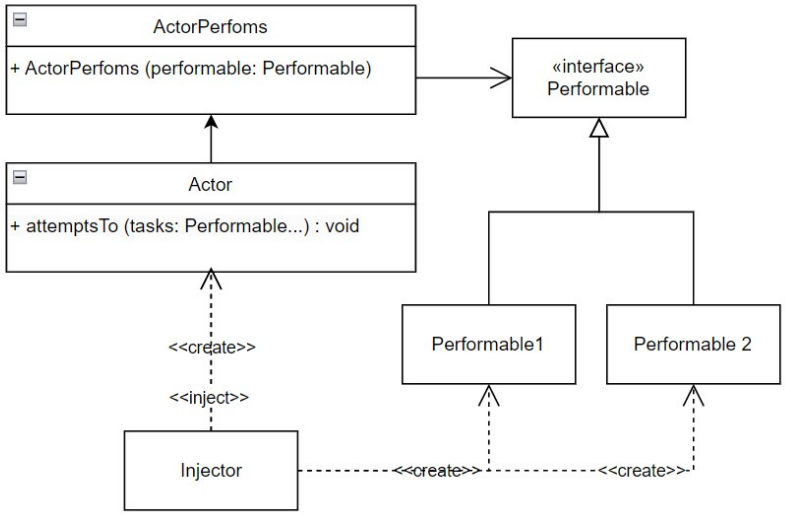

Abbildung 4.25: Die Abhängigkeiten für einen *Dependency-Injector*

Dieser Ansatz führt zu einer gesteigerten Flexibilität, da die enge Kopplung zwischen Klassen und ihren Abhängigkeiten reduziert wird. Die Methode *attemptsTo()* kann wiederverwendet werden, da sie nicht direkt von den *Interaction*-Klassen abhängt. Darüber hinaus wird ermöglicht, den Parameter der Methode *attemptsTo()* flexibel auszutauschen und anzupassen.

Der zweite Aspekt des DIP besagt, dass Abstraktionen nicht von Details abhängen sollten, sondern Details von Abstraktionen. Konkret bedeutet dies, dass die Schnittstelle *Performable* (Abstraktion) so gestaltet sein sollte, dass sie die Anforderungen der *Actor*-Klasse berücksichtigt, anstatt spezifische Anforderungen der *Click*- oder *Enter*-Klasse (Details) zu beachten. Dies wird in Abbildung 4.23 deutlich, da die Details von den Abstraktionen abhängig sind. Zusammenfassend kann festgestellt werden, dass das DIP in diesem Zusammenhang eingehalten wurde.

4.2.4.2 *DRY-Prinzip*

Im Rahmen des DRY-Prinzips wird empfohlen, jegliche Form von Redundanz konsequent zu vermeiden. Durch die Beachtung dieses Prinzips wird das Risiko von Fehlern aufgrund von Inkonsistenzen reduziert, da dieselbe Logik nicht mehrfach implementiert werden muss.

Das klassische POM verstößt gegen das DRY-Prinzip, da es häufig zu erheblichen Code-Duplikaten führt. Ein Beispiel hierfür wäre wieder ein *Dropdown*-Menü auf einer Webseite, für das die Interaktionsmethoden neben den Webelementen in der entsprechenden Seitenklasse implementiert sind. Angenommen, ein ähnliches *Dropdown*-Menü auf einer anderen Seite soll ebenfalls getestet werden und die Funktionalität ist identisch mit dem ersten *Dropdown*-Menü. Daraus folgt, dass im klassischen POM eine neue Methode für das zweite *Dropdown*-Menü in einer anderen Seitenklasse erstellt werden müsste, selbst wenn die Logik der Methode exakt dieselbe ist.

Dies führt zu unnötiger Redundanz und erschwert die Wartung, da Änderungen an der *Dropdown*-Logik in jeder betroffenen Seitenklasse durchgeführt werden müssen. Dies steht im Widerspruch zum DRY-Prinzip.

Eine effektivere Lösung wäre, die Logik des *Dropdown*-Menüs in einer separaten Klasse auszulagern und diese Logik wiederzuverwenden. Dieses Konzept wird im *Screenplay* umgesetzt. Mit dem *Screenplay* muss lediglich die entsprechende *Dropdown*-Klassenmethode aufgerufen werden, unabhängig davon, wie viele *Dropdown*-Menüs auf verschiedenen Seiten getestet werden sollen. Dadurch wird die Menge an redundantem Code erheblich reduziert. Es sei jedoch angemerkt, dass das *Screenplay* besonders geeignet ist, um komplexere Interaktionen zu kombinieren und wiederzuverwenden. Einfache Einzelschritte müssen möglicherweise dennoch wiederholt implementiert werden, zum Beispiel das Eingeben von Text in ein Textfeld mittels *Enter.keyValues()*.

Das erweiterte POM zielt darauf ab, den Code in den *Step Definitions* so gering wie möglich zu halten. Zum Beispiel wird für das Eingeben von Text in ein Textfeld nur eine einzige Methode aufgerufen. Hierbei wird der Name des Webelements als Parameter in der *Feature*-Datei übergeben, um das entsprechende Webelement aus einer *Hash Map* abzurufen. Dadurch wird die Duplikation weiter minimiert. Jedoch ist die *Feature*-Datei im Vergleich zu den anderen beiden Entwurfsmustern technischer, weniger übersichtlich und länger. Dies hat zur Folge, dass die *Feature*-Datei im erweiterten POM dem DRY-Prinzip nicht entspricht, da für jedes Textfeld ein eigener Schritt in der *Feature*-Datei implementiert werden muss.

Abbildung 4.26 verdeutlicht, dass im erweiterten POM zwei Schritte erforderlich sind, um Text in ein Textfeld einzugeben, nämlich für die E-Mail und das Passwort. Dies steht im Gegensatz zum klassischen POM und *Screenplay*, bei denen diese beiden Aktionen in einem Schritt zusammengefasst werden können (Abbildung 4.27).

```
Scenario: Login successfully
  Given User visits login page in CAT
  And User enters the value bob.beyer@accso.de in the email field
  And User enters the value bobspassword in the password field
  When User clicks on loginButton
  Then zeiterfassungTab should be displayed
```

Abbildung 4.26: *‚login.feature‘* im erweiterten POM

```
Scenario: Login successfully
  When User logs in with a valid username and password
  Then User should be given access to his account
```

Abbildung 4.27: *‚login.feature‘* im klassischen POM

4.2.4.3 *Zusammenfassung*

Das Ergebnis des Vergleichs im Hinblick auf die Prinzipien der Softwareentwicklung lassen sich folgendermaßen zusammenfassen, wie in Tabelle 4.2 dargestellt. Für jedes Kriterium wurde das jeweilige Entwurfsmuster bewertet. Eine Bewertung von drei Punkten zeigt die vollständige Einhaltung des Prinzips an, während zwei Punkte auf eine teilweise Erfüllung hinweisen. Ein Punkt deutet darauf hin, dass das Prinzip nicht eingehalten wurde. Nicht behandelte Kriterien wurden mit einem ‚-‘ gekennzeichnet.

Prinzipien	Klassisches POM	Erweitertes POM	*Screenplay*
SRP	1	3	3
OCP	1	2	3
LSP	-	3	3
ISP	-	2	3
DIP	-	-	3
DRY-Prinzip	1	2	2
Summe	3	12	17

Tabelle 4.2: Vergleich von drei Entwurfsmustern durch die Prinzipien der Softwareentwicklung

Die Analyse verdeutlicht, dass das *Screenplay* die SOLID-Prinzipien vollständig erfüllt. Im Gegensatz dazu entspricht das erweiterte POM diesen Prinzipien nur teilweise. Hierbei verstößt lediglich die Speicherung von Webelementen in einer *Hash Map* gegen das OCP und ISP. Diese Verstöße spiegeln sich in der Tabelle durch die Verleihung von zwei Punkten wider.

Hinsichtlich des DRY-Prinzips erhielten sowohl das *Screenplay* als auch das erweiterte POM jeweils zwei Punkte. Jedoch bietet das *Screenplay* in den *Feature*-Dateien mehr Vorteile im Vergleich zum erweiterten POM, welches seine Stärken in den *Step Definitions* aufweist.

Die Gesamtpunktzahl zeigt deutlich, dass sowohl das erweiterte POM als auch *Screenplay* wesentlich mehr Punkte erzielen als das klassische POM. Besonders das *Screenplay* weist die höchste Punktzahl auf. Diese Ergebnisse deuten darauf hin, dass das erweiterte POM und *Screenplay* im Vergleich zum klassischen POM eine verbesserte Wartbarkeit, Lesbarkeit und Erweiterbarkeit aufweisen.

4.2.5 *QMOOD*

Um den Quellcode zu bewerten, wurde das Plug-In ‚*MetricsTree*' für die *IntelliJ-IDE* verwendet. ‚*MetricsTree*' wurde von V.V. Burakov und A.I. Borovkov entwickelt und basiert auf den Forschungsarbeiten von Bansiya und C.G. Davis. Dieses Tool verwendet die Ersatzmetriken, die in der Arbeit von Mandeep K. Chawla und Dr. Indu Chhabra vorgestellt wurden.

Einige Metrikwerte beziehen sich auf Klassen, während andere auf Pakete abzielen. Aufgrund der unterschiedlichen Wertebereiche wurden die Werte in diesem Plug-In normalisiert, wobei z-Werte für die Berechnungen verwendet wurden. ‚*MetricsTree*' präsentiert ausschließlich die berechneten Werte für die jeweiligen Qualitätsattribute, anstatt die spezifischen Werte der individuellen Metriken selbst. Deshalb fokussiert sich diese Arbeit auf die Diskussion der Unterschiede im TQI der drei Entwurfsmuster. Eine ausführliche Behandlung der Unterschiede zwischen den einzelnen Qualitätsattributen wie Effektivität oder Erweiterbarkeit wird nicht im Detail durchgeführt.

Wie in Abbildung 4.28 veranschaulicht wird, zeigen sowohl das *Screenplay* als auch erweiterte POM eine höhere Effektivität, Erweiterbarkeit, Flexibilität, Funktionalität und Wiederverwendbarkeit. Es ist anzumerken, dass die Verständlichkeit das einzige Qualitätsattribut ist, das negative Werte aufweist. Dies deutet darauf hin, dass das erweiterte POM und *Screenplay* anspruchsvoller in Bezug auf Verständnis und Erlernbarkeit sind im Vergleich zum klassischen POM. Diese Beobachtung entspricht den Erwartungen, da sowohl das erweiterte POM als auch *Screenplay* eine komplexere Struktur aufweisen.

Abbildung 4.28: Berechnete Qualitätsattribute von QMOOD

Die gleiche Tendenz zeigt sich ebenfalls im Zusammenhang mit dem TQI, wobei der TQI-Wert für das klassische POM den niedrigsten Wert ergab. Angesichts der ähnlichen TQI-Werte für das erweiterte POM und *Screenplay* wurden zusätzlich 15 weitere Seiten mit *Dropdown*-Menüs getestet, um die TQI-Tendenz genauer zu erfassen. Wie aus Tabelle 4.3 hervorgeht, bleibt die TQI-Tendenz für alle Entwurfsmuster konstant, wenn die Anzahl der Seiten zunimmt. Allerdings hat sich der Unterschied zwischen dem erweiterten POM und *Screenplay* deutlich vergrößert. Dies lässt darauf schließen, dass das *Screenplay* eine höhere Qualität aufweist als die beiden anderen Entwurfsmuster, was die Ergebnisse aus dem vorherigen Abschnitt zu den SOLID-Prinzipien bestätigt.

Entwurfsmuster	TQI (Mit 2 *Dropdown*-Menüs)	TQI (Mit 17 *Dropdown*-Menüs)
Klassisches POM	7,9	9,62
Erweitertes POM	9,6	10,25
Screenplay	9,94	12,87

Tabelle 4.3: TQIs von drei Entwurfsmustern

4.3 DER VERGLEICH UND DAS ERGEBNIS

4.3.1 *Ergebnis des Vergleichs*

Im folgenden Abschnitt wird anhand definierter Vergleichskriterien eine Zusammenfassung durchgeführt, um festzustellen, welches Entwurfsmuster für welches Projekt besser geeignet ist.

1. Lesbarkeit der Feature-Datei

   Der BDD-Ansatz erfordert eine enge Zusammenarbeit zwischen technischen und nicht-technischen Teams. Ein beliebter Ansatz dafür ist ‚*Three Amigos*', der von vielen Teams angewendet wird. In diesem Ansatz kommen Teammitglieder mit unterschiedlichen Rollen - oft ein Entwickler, ein Tester und ein Product Owner - zusammen, um *Features* zu besprechen, bevor die tatsächliche Implementierung beginnt. Gelegentlich werden auch erste Entwürfe für automatisierte Testszenarien mit *Gherkin* zusammen verfasst. Dies trägt dazu bei, ein gemeinsames Verständnis im Team zu fördern. [SM23]

   Im erweiterten POM wird ein imperativer Ansatz verwendet und es werden technische Aspekte in den *Feature*-Dateien integriert. Dies kann dazu führen, dass die Lesbarkeit und die Verständlichkeit der *Feature*-Dateien im Vergleich zu den anderen Entwurfsmustern verringert wird. Darüber hinaus kann die vorherige Erstellung der *Feature*-Dateien mit dem erweiterten POM problematisch sein, da der Name des Webelements als Parameter benötigt wird. Allerdings kann dieser Parameter erst während der Implementierungsphase definiert werden.

Daher eignet sich das erweiterte POM weniger für Projekte, in denen die Zusammenarbeit mit dem Product Owner oder den Kunden eine frühzeitige Einbindung in die Erstellung der *Feature*-Dateien erfordert.

2. Test mit mehreren Benutzern

   Das *Screenplay* weist einen deutlichen Vorteil auf, da es einen benutzerzentrierten Ansatz verfolgt. Jede *Step Definition* beginnt mit einem *Actor*, was zu einer erhöhten Übersichtlichkeit bei Testszenarien mit mehreren Benutzern führt. Darüber hinaus bietet das *Screenplay* äußerst hilfreiche Funktionen wie *remember()* und *recall()*, um Daten für jeden spezifischen *Actor* zu speichern und später abzurufen.

   Im Gegensatz dazu wird in den beiden anderen Entwurfsmustern davon ausgegangen, dass nur ein einzelner Benutzer mit der zu testenden Anwendung interagiert. Bei Verwendung des klassischen oder erweiterten POM gestaltet sich die Umsetzung von Testszenarien mit mehreren Benutzern aufwendiger, da mehrere *Step Definitions* angepasst werden müssen, um die *Feature*-Datei übersichtlich zu gestalten.

3. TQI und Prinzipien der Softwareentwicklung

   Durch den Vergleich mit den Prinzipien der Softwareentwicklung lässt sich feststellen, dass das *Screenplay* auf den SOLID-Prinzipien basiert, während das erweiterte POM diese Prinzipien teilweise erfüllt. Im Gegensatz dazu verletzt das klassische POM das SRP sowie das OCP von den SOLID-Prinzipien.

   Zusätztlich dazu verstößt das klassische POM gegen das DRY-Prinzip, während die beiden anderen Entwurfsmuster dieses teilweise erfüllen. Dies stimmt mit dem TQI des QMOOD überein, der ebenfalls darauf hinweist, dass sowohl das *Screenplay* als auch das erweiterte POM im Vergleich zum klassischen POM eine höhere Qualität aufweisen. Daraus lässt sich schlussfolgern, dass das erweiterte POM und *Screenplay* besonders für umfangreiche Projekte geeignet sind, da sie eine gute Wartbarkeit und Erweiterbarkeit aufweisen.

4. Wartbarkeit

   In Bezug auf die Wartbarkeit der Testfälle weist das erweiterte POM einen deutlichen Vorteil gegenüber den beiden anderen Entwurfsmustern auf. Bei Veränderungen in dem UI muss die implementierte *Step Definition* nicht angepasst werden, was im Gegensatz zu den beiden anderen Mustern der Fall ist. Dies impliziert auch eine Reduktion des zu schreibenden Codes für neue Tests, da die bereits implementierten *Step Definitions* wiederverwendet werden können.

4.3.2 *Auswahl der geeigneten Entwurfsmuster anhand des Projektkontexts*

Basierend auf den Ergebnissen des Vergleichs lässt sich die Auswahl eines Entwurfsmusters für ein Projekt wie folgt zusammenfassen:

1. Test mit mehreren Benutzern

   Es ist zunächst von Bedeutung, zu analysieren, ob die Anwendung eine Interaktion mit mehreren Benutzern erfordert. In dieser Betrachtung erweist sich das *Screenplay* als vorteilhaft und präsentiert einen ansprechenden sowie unkomplizierten Ansatz.

2. Testumfang

   Die Berücksichtigung des Testumfangs stellt auch einen bedeutsamen Faktor dar. Anhand von Kriterien wie QMOOD, der Wartbarkeit und den Prinzipien der Softwareentwicklung wurde festgestellt, dass sowohl das *Screenplay* als auch das erweiterte POM eine höhere Qualität aufweisen und besser wartbar sind im Vergleich zum klassischen POM. Daher eignet sich das klassische POM weniger für komplexe und umfangreiche Anwendungen, da es gegen die SOLID-Prinzipien und das DRY-Prinzip verstößt und somit schwer wartbar ist. Trotzdem kann das klassische POM verwendet werden, wenn die Zeit knapp und der Testumfang begrenzt ist, da seine Struktur einfacher ist als die der anderen beiden Entwurfsmuster.

3. Kenntnisse der Teammitglieder

   Im klassischen POM und im erweiterten POM wurde das weit verbreitete und beliebte Testautomatisierungstool *Selenium* eingesetzt. Wenn Teammitglieder bereits Erfahrung mit *Selenium* haben, kann das erweiterte POM eine geeignete Wahl sein, um gut wartbare Tests zu implementieren. Obwohl die Struktur des erweiterten POM etwas komplexer ist, können Teammitglieder mit weniger Erfahrung dennoch die Oberflächentests effektiv warten, da die implementierte *Step Definition* wiederverwendet werden kann und bei Änderungen nur wenig Code angepasst werden muss. Dies trägt zur Vereinfachung der Wartungsarbeiten und zur Reduzierung potenzieller Fehlerquellen bei.

   Im Gegensatz dazu erfordert die Einführung vom *Screenplay* anfangs eine hohe Einarbeitungszeit, da es eine abweichende Herangehensweise im Vergleich zu *Selenium* bietet. *Serenity BDD* bietet eine ausführliche Dokumentation für das *Screenplay* und die Syntax vom *Screenplay* ist grundsätzlich intuitiv und verständlich. Dennoch erfordert der Übergang eine gewisse Lerninvestition und setzt solide Programmierkenntnisse bei den Teammitgliedern voraus.

4. Zusammenarbeit mit Product Owner und UI der Anwendung

   Wie im vierten Kapitel erörtert wurde, ist das Testszenario im erweiterten POM technisch anspruchsvoller im Vergleich zu den beiden anderen Entwurfsmustern. Zum Beispiel kann ein Testszenario, das ein Formular mit vielen Textfeldern überprüft, schnell umfangreich werden, da für jedes Textfeld ein eigener Schritt im erweiterten POM geschrieben werden muss. Dies kann dazu führen, dass der Fokus auf den Gesamtablauf und die Geschäftslogik verloren geht. Es wird schwierig

zu erkennen, welche spezifischen Funktionalitäten getestet werden. Da der Zweck von *Feature*-Dateien darin besteht, die Zusammenarbeit zu fördern, indem sie für jedes Teammitglied leicht lesbar und schreibbar sind, erweist sich das erweiterte POM in solchen Fällen als weniger geeignet. Dies kann die Zusammenarbeit mit dem Product Owner und anderen Stakeholdern erschweren.

Das erweiterte POM kann jedoch besser für Anwendungen geeignet sein, die eine Vielfalt an verschiedenen Webelementen aufweisen, anstatt eine große Anzahl identischer Elemente, wie sie beispielsweise in Formularen oder Dialogfeldern mit zahlreichen Eingabefeldern auftreten können.

Hingegen ist das *Screenplay* besonders geeignet für Projekte, die einen BDD-Ansatz verfolgen. Die *Feature*-Datei im *Screenplay* ist deklarativer und weniger technisch als im erweiterten POM. Die Syntax vom *Screenplay* harmoniert gut mit dem BDD-Ansatz, da jede Interaktion beschreibt, welche geschäftlichen Aktionen der *Actor* ausführt. Diese Herangehensweise erleichtert die Kommunikation und Zusammenarbeit mit allen Teammitgliedern, einschließlich des Product Owners.

5 ZUSAMMENFASSUNG UND AUSBLICK

5.1 ZUSAMMENFASSUNG

Diese Bachelorarbeit befasst sich mit der Implementierung von Oberflächentests für eine betriebliche Anwendung mithilfe von drei unterschiedlichen Entwurfsmustern: dem klassischen POM, dem erweiterten POM und dem *Screenplay*. Das Hauptziel dieser Arbeit besteht darin, die Unterschiede zwischen diesen drei Ansätzen zu analysieren und zu ermitteln, welcher Ansatz für welche Art von Projekten besser geeignet ist.

Die Ergebnisse des Vergleichs verdeutlichen, dass der *Screenplay*-Ansatz einen klaren Vorteil für Anwendungen bietet, die von mehreren Benutzern interaktiv genutzt werden. Aufgrund seiner nutzerzentrierten Herangehensweise bietet das *Screenplay* im Vergleich zu den anderen beiden Entwurfsmustern eine ansprechendere Lösung.

Außerdem ist es wichtig, den Umfang der Tests zu berücksichtigen. Durch die Bewertung von Kriterien wie dem QMOOD, der Wartbarkeit und den Prinzipien der Softwareentwicklung wurde festgestellt, dass sowohl das *Screenplay* als auch das erweiterte POM eine höhere Qualität aufweisen und besser wartbar sind im Vergleich zum klassischen POM. Daher eignet sich das klassische POM lediglich für die Projekte, bei denen in kurzer Zeit eine begrenzte Anzahl von Tests implementiert werden muss, da es für umfangreiche Anwendungen schwer wartbar ist.

Im Gegensatz dazu eignen sich das erweiterte POM und *Screenplay* für komplexe Anwendungen. Das erweiterte POM zeichnet sich durch gute Wartbarkeit aus und erfordert nur minimale Änderungen in der *Step Definition*, da diese wiederverwendet werden kann. Obwohl die *Feature*-Datei des erweiterten POMs technischer ist als die vom *Screenplay*, ist sie dennoch verständlicher und einfacher als die Implementierung neuer *Step Definitions*. Daher kann das erweiterte POM von Teammitgliedern mit weniger Erfahrung gewartet werden.

Allerdings kann es auch als Nachteil betrachtet werden, dass die *Feature*-Datei vom erweiterten POM technisch ausgerichtet ist. Wenn eine Anwendung eine große Anzahl identischer Elemente enthält, wie z. B. Tabellen oder Dialoge, kann die *Feature*-Datei des erweiterten POM schnell umfangreich werden. Dies kann dazu führen, dass es schwierig wird zu erkennen, welche spezifischen Funktionen getestet werden sollen, was die Zusammenarbeit mit dem Product Owner und anderen Stakeholdern erschweren kann. Darüber hinaus ist das erweiterte POM auch ungeeignet, wenn der Product Owner oder die Kunden eng zusammenarbeiten und eine frühzeitige Einbindung in die Erstellung der *Feature*-Datei erfordern. Denn die *Feature*-Datei benötigt die Parameter, die erst während der Implementierung entstehen.

Im Vergleich dazu ist das *Screenplay* besonders geeignet für Projekte, die den BDD-Ansatz verfolgen. Die *Feature*-Datei vom *Screenplay* zeichnet sich durch ihre deklarative und nicht-technische Ausrichtung aus. Außerdem ist die Syntax vom *Screenplay* intuitiv und verständlich, was gut mit dem BDD-Ansatz harmoniert. Diese Ausrichtung erleichtert die Kommunikation und Zusammenarbeit mit allen Teammitgliedern, einschließlich des Product Owners.

5.2 AUSBLICK

Eine Limitation dieser Arbeit könnte in der begrenzten Testumgebung liegen, da lediglich acht Testszenarien implementiert wurden. Es wäre daher ratsam, zukünftige Forschungen mit einer größeren Anzahl von Testfällen durchzuführen, um umfassendere Erkenntnisse zu gewinnen.

Des Weiteren könnte eine potenzielle Verzerrung durch die Wahl des Plugins *MetricsTree* auftreten. Die durch *MetricsTree* ermittelten Werte könnten verzerrt sein, da Ersatzmetriken anstelle der Originalmetriken von QMOOD verwendet wurden. Zur Validierung der Ergebnisse wäre der Einsatz weiterer Werkzeuge wünschenswert.

Hinzukommend stellt das erweiterte POM in dieser Arbeit lediglich eine von mehreren möglichen Varianten dar, um das klassische POM zu erweitern. Daher bleibt offen, ob andere Erweiterungsvarianten des klassischen POM im Vergleich zum *Screenplay* zu ähnlichen Ergebnissen führen würden.

LITERATUR

[Axe18] Arnon Axelrod. "Complete Guide to Test Automation: Techniques". In: *Practices, and Patterns for Building and Maintaining Effective Software Projects* (2018).

[BDDa] Serenity BDD. *Introduction*. https://serenity-bdd.github.io/docs/guide/user_guide_intro/. [Online; accessed 1-August-2023].

[BDDb] Serenity BDD. *Screenplay Fundamentals*. https://serenity-bdd.github.io/docs/screenplay/screenplay_fundamentals/. [Online; accessed 10-Juli-2023].

[BDo2] Jagdish Bansiya und Carl G. Davis. "A hierarchical model for object-oriented design quality assessment". In: *IEEE Transactions on software engineering* 28.1 (2002), S. 4–17.

[Coh10] Mike Cohn. *Succeeding with agile: software development using Scrum*. Pearson Education, 2010.

[Cuca] Cucumber. *Behaviour-Driven Development*. https://cucumber.io/docs/bdd/, note = [Online; accessed 1-August-2023].

[Cucb] Cucumber. *Gherkin Reference*. https://cucumber.io/docs/gherkin/reference/. [Online; accessed 1-August-2023].

[Cucc] Cucumber. *Writing better Gherkin*. https://cucumber.io/docs/bdd/better-gherkin/. [Online; accessed 1-August-2023].

[Gol18] Joachim Goll. *Entwurfsprinzipien und Konstruktionskonzepte der Softwaretechnik*. Springer, 2018.

[HT03] Andrew Hunt und David Thomas. *Der Pragmatische Programmierer*. Hanser, 2003.

[Leo+13] Maurizio Leotta, Diego Clerissi, Filippo Ricca und Cristiano Spadaro. "Improving test suites maintainability with the page object pattern: An industrial case study". In: *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*. IEEE. 2013, S. 108–113.

[Mar+16a] Antony Marcano, Andy Palmer, Jan Molak und John Ferguson Smart. *Beyond Page Objects: Next Generation Test Automation with Serenity and the Screenplay Pattern*. https://www.infoq.com/articles/Beyond-Page-Objects-Test-Automation-Serenity-Screenplay/. [Online; accessed 10-Juli-2023]. 2016.

[Mar+16b] Antony Marcano, Andy Palmer, Jan Molak und John Ferguson Smart. *Page Objects Refactored: SOLID Steps to the Screenplay/Journey Pattern*. https://dzone.com/articles/page-objects-refactored-solid-steps-to-the-screenp. [Online; accessed 10-Juli-2023]. 2016.

[Mar+16c] Antony Marcano, Andy Palmer, Jan Molak und John Ferguson Smart. *Page Objects Refactored: SOLID steps to the Screenplay/Journey Pattern*. https://ideas.riverglide.com/page-objects-refactored-12ec3541990/. 2016.

[MMM06] Robert C Martin, Micah Martin und Micah Martin. *Agile principles, patterns, and practices in C#*. Pearson Education, Inc, 2006.

[Mey97] Bertrand Meyer. *Object-oriented software construction*. Bd. 2. Prentice hall Englewood Cliffs, 1997.

[NAF21] Michel Nass, Emil Alégroth und Robert Feldt. "Why many challenges with GUI test automation (will) remain". In: *Information and Software Technology* (2021).

[Sel] Selenium. *Working with windows and tabs*. https://www.selenium.dev/documentation/webdriver/interactions/windows/.

[Sel23] Selenium. *Page object models*. https://www.selenium.dev/documentation/test_practices/encouraged/page_object_models/. [Online; accessed 10-Juli-2023]. 2023.

[Ser] Serenity/JS. *Screenplay Pattern*. https://serenity-js.org/handbook/design/screenplay-pattern/. [Online; accessed 10-Juli-2023].

[SM23] John Ferguson Smart und Jan Molak. *BDD in Action: Behavior-driven development for the whole software lifecycle*. Simon und Schuster, 2023.

[WA20] Fadi Wedyan und Somia Abufakher. "Impact of design patterns on software quality: a systematic literature review". In: *The Institution of Engineering and Technology* 14 (2020), S. 1–17.

[Wik] Wikipedia. *Prinzipien objektorientierten Designs*. https://de.wikipedia.org/w/index.php?title=Prinzipien_objektorientierten_Designs. [Online; accessed 1-August-2023].

[YBS20] Dini Yuniasri, Tessy Badriyah und Umi Sa'adah. "A Comparative Analysis of Quality Page Object and Screenplay Design Pattern on Web-based Automation Testing". In: *Proc. of the 2nd International Conference on Electrical, Communication and Computer Engineering (ICECCE)* (2020).