BACHELORARBEIT

# Automatisierte End-to-End-Tests mit Cypress: Anwendung in einem Praxisprojekt

vorgelegt am 26. September 2025

Mohammad Haj Hassan
Matrikel-Nr: [redacted]

Erstprüferin: Prof. Dr. Larissa Putzar
Zweitprüfer: Herr Sune Timon, Maute

**HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN HAMBURG**

Fakultät Design, Medien und Information
Department Medientechnik

**Mohammad Haj Hassan**

**Thema der Bachelorthesis**

Automatisierte End-to-End-Tests mit Cypress: Anwendung in einem Praxisprojekt

**Stichworte**

Agile, Software-Testautomatisierung, UI-Test, QA, Cypress, Selenium, DevOps, JavaScript, ISTQB

**Zusammenfassung**

Automatisiertes Software-Testing stellt einen wesentlichen Bestandteil moderner Qualitätssicherung dar. In dieser Arbeit wurde untersucht, wie sich End-to-End-Tests mit dem Cypress- Framework zur Absicherung einer webbasierten Anwendung einsetzen lassen. Am Beispiel der Anwendung IntelMod konnten die zentralen Geschäftsprozesse für Vermieter:innen und Mieter:innen erfolgreich automatisiert getestet werden.

**Mohammad Haj Hassan**

**Title of the paper**

Automated End-to-End Testing with Cypress: A Practical Application

**Keywords**

Agile, Software test automation, UI-Test, QA, Cypress, Selenium, DevOps, JavaScript, ISTQB

**Abstract**

Automated software testing is an essential part of modern quality assurance. This thesis examined how end-to-end testing can be used with the Cypress framework to validate a web based application. Using the IntelMod application as an example, the central business processes for landlords and tenants were successfully tested automatically.

# Inhaltsverzeichnis

# Abkürzungsverzeichnis

| | |
|---|---|
| API | Application Program Interface |
| CI | Continuous Integration |
| CD | Continuous Delivery |
| DOM | Document Object Model |
| E2E | End-to-End |
| UI | User Interface |
| IEEE | Institute of Electrical and Electronics Engineers |
| IDE | Integrated Development Environment |
| ISTQB | International Software Testing Qualifications Board |
| UAT | User Acceptance Testing |
| VS Code | Visual Studio Code |
| npm | Node Package Manager |
| JS | JavaScript |
| ISO | International Organization for Standardization |

# Glossar

| | |
|---|---|
| User Interface | Benutzer Schnittstelle (z. B.: Webseite) |
| Cypress | Ein Testframework zur Automatisierung von End-to-End-Tests im Browser. |
| Selenium | Framework-Suite für browserübergreifende UI-Tests auf Basis des W3C-WebDriver-Standards. |
| Document Object Model (DOM) | Hierarchische Baumdarstellung einer Webseite, über die Testwerkzeuge wie Cypress Elemente finden, Inhalte prüfen und Interaktionen simulieren. |
| IntelMod | In der Arbeit getestete Webanwendung mit separaten Sichten für Vermieter:innen und Mieter:innen. |
| ISTQB | Internationales Gremium für standardisierte Lehrpläne und Begriffe im Softwaretest. |
| Visual Studio Code (VS Code) | Genutzte Entwicklungsumgebung für das Starten der lokalen Anwendung und die Implementierung/Ausführung der Cypress-Tests. |
| DevOps | Sammlung von Prozessen für die Entwicklung und operationalen IT-Teams |
| Webanwendung | ein Anwendungsprogramm nach dem Client-Server-Modell, z. B. eine Website |
| XPath | Abfragesprache zur Adressierung von Elementen in XML/HTML-Dokumenten; in der Arbeit für das präzise Finden von UI-Elementen eingesetzt. |
| Continuous Delivery (CD) | Methode zur abschließenden Bereitstellungsphase einer automatisierten Software-Release-Pipeline |
| Continuous Integration (CI) | Methode zur automatisierten Integration von Codeänderungen |
| Framework | Vorprogrammierter Code zur einfachen Nutzung, um bestimmte Bereiche oder Probleme zu lösen. |
| Web-Element-Locators | Dies sind Objekte, die Web-Elemente, wie z. B. das Anklicken eines Links auf einer Webseite, anhand einer bestimmten Anfrage finden und zurückgeben. |

# Abbildungsverzeichnis

# Tabellenverzeichnis

# 1 Einleitung

Im folgenden Kapitel werden die Hintergründe, das Ziel der Arbeit und die Forschungsfrage erläutert, was einen ersten Überblick über die Aufgabenstellung und den inhaltlichen Rahmen der Arbeit bietet.

## 1.1 Motivation

In der digitalen Welt von heute ist die Qualität von Softwareprodukten von zentraler Bedeutung. Der Druck auf Unternehmen wächst, Anwendungen in immer kürzeren Zyklen zu entwickeln, während gleichzeitig hohe Standards in Bezug auf Stabilität, Sicherheit und Benutzerfreundlichkeit erfüllt werden müssen. Softwaretests stellen dabei ein zentrales Werkzeug zur Qualitätssicherung dar. Internationale Normen wie die ISO/IEC 25010:2023 legen Qualitätsmerkmale fest, die zur Bewertung von Softwareprodukten dienen (ISO/IEC, 2023). [22]

Mit zunehmender Komplexität sind manuelle Tests häufig nicht ausreichend; die Testautomatisierung hingegen ermöglicht eine effiziente und wiederholbare Überprüfung. Sie verkürzt Entwicklungszyklen und minimiert die Fehleranfälligkeit. [10] [15]
Aktuelle Untersuchungen wie der World Quality Report 2024/25 zeigen, dass Unternehmen verstärkt auf automatisierte Testverfahren setzen, um den Herausforderungen der modernen Softwareentwicklung zu begegnen (Capgemini, 2024). [2]

## 1.2 Forschungsfrage

Die Arbeit untersucht die zentrale Fragestellung:

„Welche Relevanz hat automatisiertes Software-Testing in der heutigen Softwareentwicklung?“

## 1.3 Forschungsziel

Ziel der Untersuchung ist es, den aktuellen Stellenwert automatisierter Testverfahren in der Softwareentwicklung zu erfassen und ihren Beitrag zur Gewährleistung der Softwarequalität zu analysieren. Zuerst wird ein theoretischer Überblick über die Grundlagen der Testautomatisierung präsentiert, um deren Vorteile, Einschränkungen und Anwendungsgebiete systematisch zu erfassen.

Ein weiterer Fokus liegt auf dem Vergleich von Cypress als modernem Testframework mit Selenium. Mit diesem Vergleich werden die Unterschiede zwischen verschiedenen Testwerkzeugen in Bezug auf Architektur, Performance, Wartbarkeit und Praxistauglichkeit aufgezeigt, sowie die Auswirkungen, die diese Unterschiede auf die Auswahl geeigneter Werkzeuge in Projekten haben können.

Zusätzlich hat die Arbeit das Ziel, durch ein praktisches Projekt zu zeigen, wie Cypress in einem realen Anwendungskontext genutzt wird. Die Untersuchung der Auswirkungen von automatisierten

Testverfahren auf die Effizienz, die Testabdeckung und die Fehlererkennung in der Softwareentwicklung erfolgt durch das Implementieren und Bewerten von End-to-End-Tests.
So hilft die Arbeit dabei, die theoretische Bedeutung der Testautomatisierung zu zeigen und gleichzeitig praxisnahe Erkenntnisse zu liefern, die zukünftigen Softwareprojekten als Orientierung dienen können.

# 2 Grundlagen der Testautomatisierung

Die theoretischen Grundlagen der Testautomatisierung werden in diesem Kapitel erläutert. Dies umfasst die allgemeine Wichtigkeit von Softwaretests, die unterschiedlichen Testarten und -stufen, die Funktion agiler Methoden sowie die speziellen Schwierigkeiten beim End-to-End-Testing und die etablierten Teststrategien.

## 2.1 Bedeutung von Softwaretests in der modernen Entwicklung

Im Kontext der modernen Softwareentwicklung kommt Softwaretests eine zunehmend zentrale Bedeutung zu. Es wird deutlich, dass die Qualitätssicherung mittlerweile ein entscheidender Erfolgsfaktor für Softwareprojekte ist. Die frühzeitige Identifikation von Fehlern ist aus ökonomischer Sicht von Relevanz, und die Testautomatisierung spielt in den aktuellen Entwicklungsmodellen eine entscheidende Rolle.

### 2.1.1 Qualität als Wettbewerbsfaktor

Gegenwärtig ist die Qualität von Software ein entscheidender Faktor für den wirtschaftlichen Erfolg digitaler Produkte. Um am Markt erfolgreich zu sein, müssen Anwendungen zuverlässig, sicher und leicht zu bedienen sein. [1]
Systeme, die fehlerhaft oder instabil sind, verursachen nicht nur einen höheren Wartungsaufwand; sie können auch das Vertrauen der Kunden nachhaltig schädigen. Internationale Normen wie das Qualitätsmodell ISO/IEC 25010 gliedern die wesentlichen Qualitätsmerkmale und zeigen auf, dass Softwarequalität weit über die reine Funktionalität hinausgeht. [22] [3]


Abbildung 1: Qualitätsmerkmale nach ISO/IEC 25010 [29]

### 2.1.2 Fehlerkosten im Entwicklungsprozess

Ein wichtiger Punkt in der Testpraxis ist die Verteilung der Kosten, wenn Fehler behoben werden. Je später ein Fehler im Entwicklungsprozess erkannt wird, desto höherer Aufwand ist nötig, um ihn zu korrigieren. Fehler, die in der Anforderungs- oder Designphase passieren, kosten noch wenig, wenn sie dort behebt werden; in der Betriebsphase verursachen sie hingegen ein Vielfaches an Aufwand.
Durch die frühzeitige Erkennung und Beseitigung von Defekten tragen Softwaretests erheblich dazu bei, die Gesamtkosten eines Projekts zu senken. [3] [4] [5]

### 2.1.3 Rolle der Testautomatisierung

Die steigende Komplexität von Softwaresystemen macht es notwendig, dass Teststrategien nicht nur manuelle Prüfungen umfassen. Testautomatisierung hat hier entscheidende Vorteile: Tests können beliebig oft wiederholt, schnell ausgeführt und in bestehende Entwicklungsprozesse eingebaut werden.
Die Automatisierung ist besonders im Kontext von agilen Methoden und DevOps-Praktiken unerlässlich. Sie erlaubt es, Softwareänderungen fortlaufend zu überprüfen, und ist somit die Basis für kurze Release-Zyklen bei gleichbleibend hoher Qualität. Die konkrete Verteilung von automatisierten Tests über die verschiedenen Ebenen wird in anerkannten Teststrategien, wie der Testpyramide, erläutert (vgl. 2.5). [2] [10] [14] [15]

## 2.2 Testarten und -stufen

Ein wesentlicher Bestandteil der Softwarequalitätssicherung ist die systematische Klassifikation von Testverfahren. Diese lassen sich unter anderem danach differenzieren, welchen Aspekt der Software sie überprüfen, beispielsweise die korrekte Funktionsweise, die Leistungsfähigkeit oder die Sicherheit gegenüber Angriffen. Zum anderen lassen sich Tests danach einordnen, wann sie im Entwicklungsprozess stattfinden. [6]
So wird deutlich, zu welcher Phase im Software-Lebenszyklus ein Test gehört. Diese zweifache Sichtweise hilft dabei, eine durchdachte Teststrategie zu entwickeln, die sowohl technische Fragen als auch organisatorische Abläufe berücksichtigt. Bekannte Modelle wie die Testpyramide, der Testdiamant oder andere Test-Shapes greifen dieses Prinzip auf und machen es anschaulich. [3] [5] [14]

### 2.2.1 Testarten

Softwaretests werden nach ihrer Art kategorisiert, je nachdem, welche Merkmale oder Funktionen der Software sie überprüfen sollen. Durch diese Differenzierung wird eine gezielte Teststrategie möglich, die alle Dimensionen der Softwarequalität berücksichtigt. Es wird grundsätzlich zwischen funktionalen, nicht-funktionalen und strukturorientierten Tests unterschieden.

Funktionale Tests kontrollieren, ob die Software die spezifizierten Anforderungen erfüllt und die vorgesehenen Funktionen korrekt bereitstellt. Das System wird aus der Perspektive des Anwenders betrachtet, ohne Rücksicht auf die interne Implementierung. Eingabetests zur Validierung, Überprüfungen der Geschäftslogik oder Schnittstellentests sind typische Beispiele. In der Praxis sind funktionale Tests das Herzstück der Qualitätssicherung, weil sie direkt den Nutzen der Software für den Endanwender absichern. [3] [5]

Nicht-funktionale Tests prüfen die Merkmale der Software, die über das bloße Erfüllen der Funktionen hinausgehen. Aspekte wie Leistung, Zuverlässigkeit, Sicherheit, Benutzbarkeit oder Portabilität gehören dazu. Sie garantieren, dass das System unter Stress, in unterschiedlichen Umgebungen oder bezüglich Datenschutz und Sicherheit den Erwartungen entspricht. In modernen Anwendungen, die hohe Anforderungen an Skalierbarkeit und Verfügbarkeit stellen, sind nicht-funktionale Tests ein absolutes Muss. [5] [22]

Strukturtests, die auch als White-Box-Tests beschrieben werden, kontrollieren den internen Aufbau des Softwarecodes. Ihr Fokus liegt auf dem Wissen über die Implementierung und sie haben die Absicht, den Programmfluss, die logischen Verzweigungen und die Abdeckung der Quellcode-Pfade zu prüfen. Gewöhnliche Verfahren sind Anweisungs-, Zweig- oder Pfadüberdeckungen. Strukturtests sind eine wertvolle Ergänzung zu funktionalen und nicht-funktionalen Tests, weil sie dazu dienen, versteckte Fehler in der Implementierung aufzudecken und die Testabdeckung objektiv zu messen. Automatisierte Unit- und Component-Tests, die häufig die breite Basis einer ausgewogenen Teststrategie in Modellen wie der Testpyramide bilden, werden in der Praxis oft durch Strukturtests vorausgesetzt. [3] [5]

### 2.2.2 Teststufen

Softwaretests können auch nach ihrer Phase im Entwicklungsprozess kategorisiert werden, zusätzlich zur Einteilung nach Testarten. Teststufen sind Gruppen von Testaktivitäten, die sich auf unterschiedliche Abstraktionsebenen des Softwareprodukts beziehen, angefangen bei einzelnen Modulen bis hin zum Gesamtsystem. Eine strukturierte Herangehensweise im Testprozess, die das frühzeitige Erkennen von Fehlern ermöglicht und somit zu einer hohen Produktqualität beiträgt, wird durch die konsequente Anwendung dieser Stufen gewährleistet.

Teststrategien von höherer Ordnung, wie der Testpyramide oder anderen Modellen, legen fest, wie Umfang und Gewichtung dieser Teststufen verteilt sind. [3] [5] [14]

#### 2.2.2.1 Komponententests

Als Komponententests, Modultests oder Unit-Tests bezeichnet, überprüfen sie die kleinste testbare Einheit einer Software. Das umfasst alles, von Methoden und Klassen bis hin zu einzelnen Programmfunktionen. In dieser Phase soll erreicht werden, dass jede Komponente für sich allein korrekt funktioniert, unabhängig von den anderen Modulen.

Typische Programmierfehler wie falsche Berechnungen, ungültige Parameter oder nicht behandelte Ausnahmen lassen sich durch Komponententests frühzeitig erkennen. Die Testpyramide hat sie als Grundlage, weil sie in umfangreicher Anzahl, mit geringem Aufwand und schnell ausgeführt werden können. Frameworks für Unit-Tests, wie JUnit (Java), NUnit (C#) oder PyTest (Python), machen es möglich, dass diese Tests größtenteils automatisiert und kontinuierlich ausgeführt werden können. [3] [5]

#### 2.2.2.2 Integrationstests

Integrationstests prüfen das Zusammenspiel multipler Softwarekomponenten oder -module, nachdem sie erfolgreich auf Einzelebene getestet wurden. Der Fokus liegt auf den Schnittstellen der Module. Es wird beispielsweise überprüft, ob Daten richtig übergeben, verarbeitet und zurückgegeben werden.
Es existieren verschiedene Vorgehensweisen:

- Top-Down-Integration: Beginnend mit den obersten Modulen werden nach und nach die tieferliegenden integriert.
- Bottom-Up-Integration: Zunächst werden die untersten Module integriert und getestet, bevor die höheren Ebenen eingebunden werden.
- Big-Bang-Ansatz: Alle Komponenten werden gleichzeitig integriert, was zwar einfach erscheint, aber ein hohes Fehlerrisiko birgt.

Mit Integrationstests wird überprüft, ob Module zusammen stabil laufen, nicht nur einzeln.
Damit nehmen Integrationstests in gängigen Testmodellen eine zentrale Rolle in der Mitte zwischen Unit- und End-to-End-Tests ein. [5] [14]

#### 2.2.2.3 Systemtests

Systemtests bilden die umfassendste Teststufe innerhalb des Entwicklungsprozesses. Hierbei wird das gesamte System in einer möglichst realistischen Umgebung getestet. Die Überprüfung erfolgt sowohl hinsichtlich der funktionalen Anforderungen als auch der nicht-funktionalen Eigenschaften wie Leistung, Sicherheit, Benutzerfreundlichkeit oder Portabilität.

Systemtests könnten auf verschiedenen Testumgebungen durchgeführt werden, die der späteren Produktivumgebung so nah wie möglich kommen. Typische Beispiele umfassen Lasttests zur Erfassung von Antwortzeiten, Penetrationstests zur Untersuchung sicherheitsrelevanter Schwachstellen und Usability-Tests zur Beurteilung der Benutzerfreundlichkeit.

In dieser Stufe soll eine objektive Aussage über die Einsatzfähigkeit des Gesamtsystems getroffen werden. [5] [14] [22]

#### 2.2.2.4 Abnahmetests

Die Abnahmetests, auch bekannt als User Acceptance Tests (UAT), bilden die abschließende Stufe im Testprozess.
In der Regel werden sie von den Auftraggebern, Kundinnen und Kunden oder Endnutzerinnen und Endnutzern durchgeführt, um zu überprüfen, ob die entwickelte Software den vertraglich festgelegten Anforderungen entspricht und praktisch eingesetzt werden kann.

In dieser Phase stehen nicht allein die technische Funktionsfähigkeit im Vordergrund, sondern insbesondere die geschäftlichen Anforderungen sowie die Nutzererfahrung. Um das Produkt endgültig freizugeben und in den produktiven Betrieb zu überführen, sind erfolgreiche Abnahmetests erforderlich.

Abnahmetests, die in einer realitätsnahen Umgebung durchgeführt werden, bieten dem Kunden ein hohes Maß an Sicherheit. Sie erfordern jedoch eine sorgfältige Planung sowie Abnahmekriterien, die bereits in der Anforderungsphase festgelegt wurden. [5] [14]

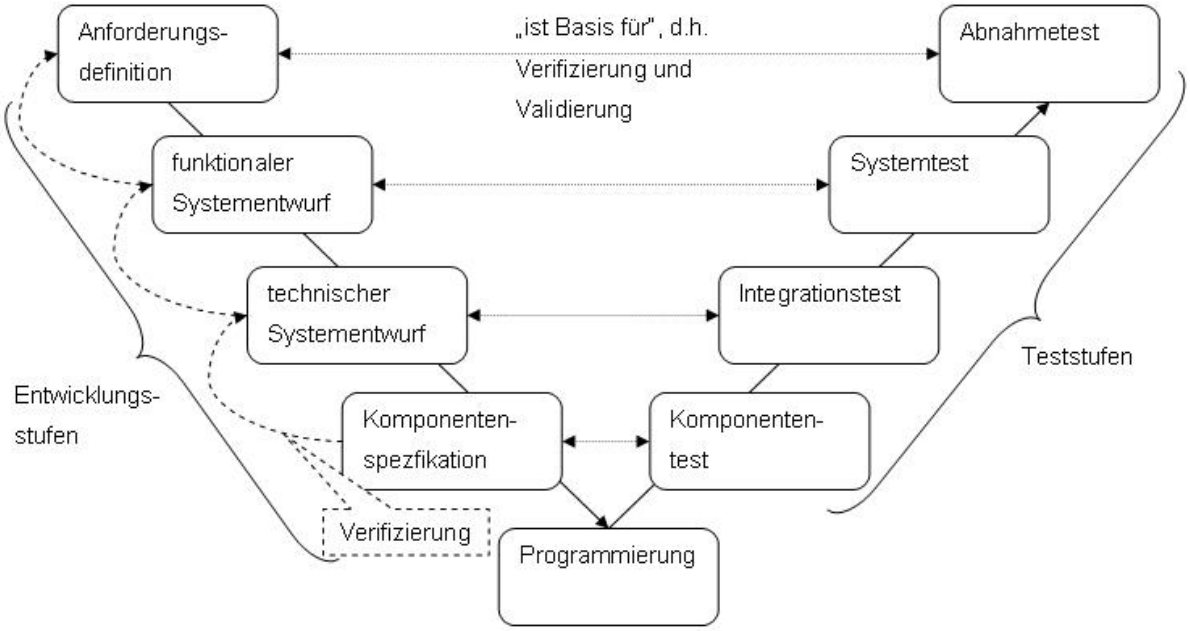


Abbildung 2: Das V-Modell nach ISTQB [5]

## 2.3 Agile Methoden und Testautomatisierung

Die Einführung agiler Vorgehensmodelle wie Scrum oder Kanban hat die Softwareentwicklung nachhaltig geprägt. An die Stelle langwieriger Entwicklungszyklen treten kurze Iterationen, die eine zeitnahe Bereitstellung funktionsfähiger Software ermöglichen. Dies erfordert eine frühzeitige und kontinuierliche Integration der Qualitätssicherung in den Entwicklungsprozess. Eine zentrale Rolle

nimmt dabei die Testautomatisierung ein, da sie eine kontinuierliche Überprüfung der Software erlaubt und somit den Anforderungen moderner Entwicklungsparadigmen entspricht. [9] [10] [15]

### 2.3.1 Testautomatisierung im agilen Umfeld

Die schnelle Lieferung, mit der funktionierende Software bereitgestellt wird, ist das Hauptaugenmerk der agilen Methoden. Weil in jedem Sprint neue Funktionen hinzugefügt und bestehende angepasst werden, ist es essenziell, kontinuierlich zu überprüfen, dass keine Regressionen entstehen. [9] [15]

- Die Bedeutung der Automatisierung: Durch automatisierte Tests wird diese schnelle Taktung abgesichert, indem sie nach jeder Änderung umgehend Feedback zur Stabilität der Anwendung geben.

- Definition of Done (DoD): In agilen Entwicklungsprojekten umfasst die „Definition of Done“ häufig die erfolgreiche Durchführung automatisierter Tests. Erst wenn die automatisierten Tests erfolgreich durchlaufen sind, gilt eine User Story als abgeschlossen.

- Regressionstests: Durch die Automatisierung können Regressionstests in jedem Sprint wiederholt und der manuelle Testaufwand erheblich reduziert werden.

### 2.3.2 Continuous Integration und Delivery

Ein zentrales Element der agilen Softwareentwicklung ist, dass sie in einen automatisierten Build- und Deploymentprozess eingebettet wird.

- Continuous Integration (CI): Automatisierte Build-Prozesse und Tests werden bei jeder Codeänderung ausgelöst, um sicherzustellen, dass neue Funktionen ohne Fehler integriert werden können.

- Continuous Delivery (CD): Die Software wird, gestützt durch automatisierte Tests, fortlaufend in Test- und Produktionsumgebungen bereitgestellt. So wird die Zeitspanne von der Entwicklung bis zur Auslieferung erheblich verkürzt.

- Werkzeuge: In diesem Zusammenhang sind Systeme wie Jenkins, GitLab CI/CD oder GitHub Actions üblich.

Die Verteilung der Tests in agilen Pipelines erfolgt durch Modelle wie die Testpyramide, welche in Abschnitt 2.5 ausführlich erläutert wird. [10] [15]

## 2.4 End-to-End Testing: Konzepte und Herausforderungen

In modernen Softwaresystemen ist es nicht ausreichend, nur einzelne Komponenten oder Schnittstellen isoliert zu prüfen. In der Praxis durchlaufen Nutzerinteraktionen oft viele Systemschichten, die alle nahtlos zusammenarbeiten müssen, von der Benutzeroberfläche über Backend-Services bis zu Datenbanken und externen Schnittstellen. End-to-End-Tests (E2E-Tests) werden genutzt, um diese kompletten Abläufe realistisch zu simulieren. Sie prüfen ein System aus der Perspektive des Anwenders

und gewährleisten, dass die gesamte Anwendung wie beabsichtigt funktioniert. Sie haben eine hohe Aussagekraft über die Produktqualität, stellen aber auch spezifische Herausforderungen dar. [5] [14] [25]

### 2.4.1 Definition

End-to-End-Tests kontrollieren komplette Geschäftsprozesse oder Benutzerflüsse vom Anfang bis zum Ende, also von der Eingabe bis zur Ausgabe. Mit „End-to-End“ ist gemeint, dass der Test an einem festgelegten Startpunkt beginnt (z. B. „Login-Seite der Anwendung“) und erst dann erfolgreich ist, wenn das erwartete Ergebnis am Endpunkt erreicht ist (z. B. „Bestätigung einer Bestellung per E-Mail“).
Beispiel aus der Praxis:

Ein End-to-End-Test in einem Online-Shop umfasst:

- Benutzer meldet sich an.
- Artikel wird in den Warenkorb gelegt.
- Bestellung wird ausgelöst.
- Zahlungsprozess wird abgeschlossen.
- Bestätigungs-E-Mail wird empfangen.

Nur wenn jeder einzelne Schritt erfolgreich verläuft, gilt der Test als bestanden. [5] [14]

### 2.4.2 Vorteile von End-to-End Tests

End-to-End-Tests sind besonders wertvoll, weil sie aus der Sicht der Anwender die tatsächliche Nutzung simulieren. [5] [14] [25]
Die wichtigsten Vorteile sind:

1. Realitätsnähe

- Nutzerverhalten wird exakt simuliert.
- Geschäftsprozesse werden in der Testphase so geprüft, wie sie im echten Betrieb ablaufen.

2. Ganzheitliche Qualitätssicherung

- Es ist möglich, Fehler zu finden, die nur durch das Zusammenspiel mehrerer Komponenten entstehen.
- Auch Interaktionen mit externen Systemen (z. B. Zahlungsanbieter, E-Mail-Dienste, Schnittstellen) werden überprüft.

3. Erhöhung der Kundenzufriedenheit

- Da die Tests aus Sicht des Endanwenders ablaufen, ist ihre Aussagekraft für die spätere Benutzererfahrung besonders hoch.

- Mit größerer Sicherheit können Unternehmen die Auslieferung vornehmen, was die Kundenakzeptanz erhöht.

4. Reduzierung von Risiken

- Besonders bei Prozessen, die sicherheitskritisch oder geschäftsrelevant sind, können End-to-End-Tests dafür sorgen, dass keine fehlerhaften Systeme in die Produktion gelangen.

### 2.4.3 Herausforderungen und Grenzen

End-to-End-Tests sind wertvoll, bringen aber auch erhebliche Nachteile und Herausforderungen mit sich.

1. Hoher Erstellungsaufwand

- In der Regel erfordern E2E-Tests eine detaillierte Programmierung oder Konfiguration.
- Benutzeroberflächen ändern sich häufig, was Anpassungen der Tests erfordert.

2. Wartungsintensität

- Selbst kleine Anpassungen im Frontend (wie neue CSS-IDs oder geänderte Buttons) können dazu führen, dass Tests fehlschlagen, obwohl die Funktionalität weiterhin gegeben ist.
- Test-Skripte erfordern eine kontinuierliche Pflege.

3. Testdatenmanagement

- Um realistische Tests durchzuführen, sind konsistente Testdaten erforderlich (wie z. B. Kundenprofile, Produktkataloge oder Bestellhistorien).
- Um sicherzustellen, dass Tests reproduzierbar sind, müssen Testdaten regelmäßig aktualisiert und synchronisiert werden.

4. Flakiness (Instabilität)

- End-to-End-Tests sind anfällig für sogenannte „flaky tests“, also Tests, die mal bestehen und mal fehlschlagen, obwohl sich am Code nichts geändert hat.
- Ursachen: Netzwerkverzögerungen, langsame Datenbanken oder zeitabhängige Prozesse.

5. Lange Ausführungszeiten

- End-to-End-Tests sind im Vergleich zu Unit- oder Integrationstests deutlich langsamer.

Wenn zu viele E2E-Tests eingeplant werden, erschwert das die Integration in Continuous Integration Pipelines. [5] [14] [25]

### 2.4.4 End-to-End Tests im Kontext von Cypress

Das moderne Testframework Cypress ist speziell für End-to-End-Tests von Webanwendungen konzipiert und bringt einige Vorteile mit sich, die typische Herausforderungen lösen. [11] [19]

1. Unmittelbare Ausführung im Browser

- Cypress läuft im selben Ausführungskontext wie die Anwendung. Auf diese Weise sind Tests in der Lage, Benutzerinteraktionen wie Klicks, Eingaben oder Navigationen realistisch zu simulieren.

2. Automatisches Warten (Automatic Waits)

- Ein häufiges Problem bei Selenium-Tests ist das manuelle Hinzufügen von Wartezeiten. Cypress minimiert die Flakiness, indem es automatisch auf das Laden von Elementen wartet.

3. Integriertes Debugging

- Jeden einzelnen Testschritt visualisiert Cypress in einer grafischen Oberfläche. Tests können von Entwicklern angehalten, inspiziert und nachvollzogen werden, wo genau ein Fehler auftritt.

4. Snapshots und Videos

- Cypress bietet die Möglichkeit, während der Testausführung Screenshots und Videos aufzuzeichnen. Das erleichtert die Fehleranalyse erheblich.

5. Grenzen von Cypress

- Der Schwerpunkt liegt eindeutig auf Web-Frontends, aber komplexe Multi-Tab- oder Multi-Browser-Szenarien sind limitiert.
- Für Mobile-Apps oder heterogene Testumgebungen ist Selenium/ Appium flexibler.

## 2.5 Teststrategien und Test-Shapes

Teststrategien bestimmen, wie verschiedene Testarten und -stufen in einem Projekt abgestimmt und gewichtet werden, um eine effektive Prüfung zu gewährleisten. Ihr Zweck ist es, die Testabdeckung effizient zu gestalten, Risiken gezielt zu adressieren und ein ausgewogenes Verhältnis zwischen Kosten, Wartbarkeit und Aussagekraft sicherzustellen. [21] [23] [24]

### 2.5.1 Ziel und Bedeutung von Teststrategien

Die übergeordnete Ausrichtung und Priorisierung der verschiedenen Testarten und -stufen im Verlauf eines Projekts wird durch eine Teststrategie definiert. Sie beantwortet die Frage, wie die Gesamtheit der Tests effizient geplant werden, um sowohl funktionale als auch nicht-funktionale Anforderungen zu erfüllen, während Kosten, Wartbarkeit und Aussagekraft berücksichtigt werden. Ein wichtiges Ziel ist

es, ein Gleichgewicht zu finden zwischen schnellen, kleinteiligen Tests und umfassenden, aber ressourcenintensiven End-to-End-Tests.

### 2.5.2 Die Testpyramide als Referenzmodell

Das Testpyramiden-Modell, welches von Mike Cohn eingeführt wurde, ist ein wesentlicher Bestandteil der Testautomatisierung. Es stellt eine breite Grundlage aus Unit- und Komponententests dar, die durch eine mittlere Schicht aus Service- bzw. Integrationstests ergänzt wird. End-to-End-Tests sind auf der obersten Ebene der Testpyramide einzuordnen: Sie bieten zwar eine hohe Aussagekraft, sind aber auch mit hohen Kosten und einer hohen Anfälligkeit für Wartung verbunden.
Die Testpyramide zeigt also, dass eine gute Teststrategie sich auf viele stabile und schnelle Tests der unteren Ebenen stützt, während die oberen Ebenen nur kritische Szenarien abdecken sollten. [13] [21] [23]

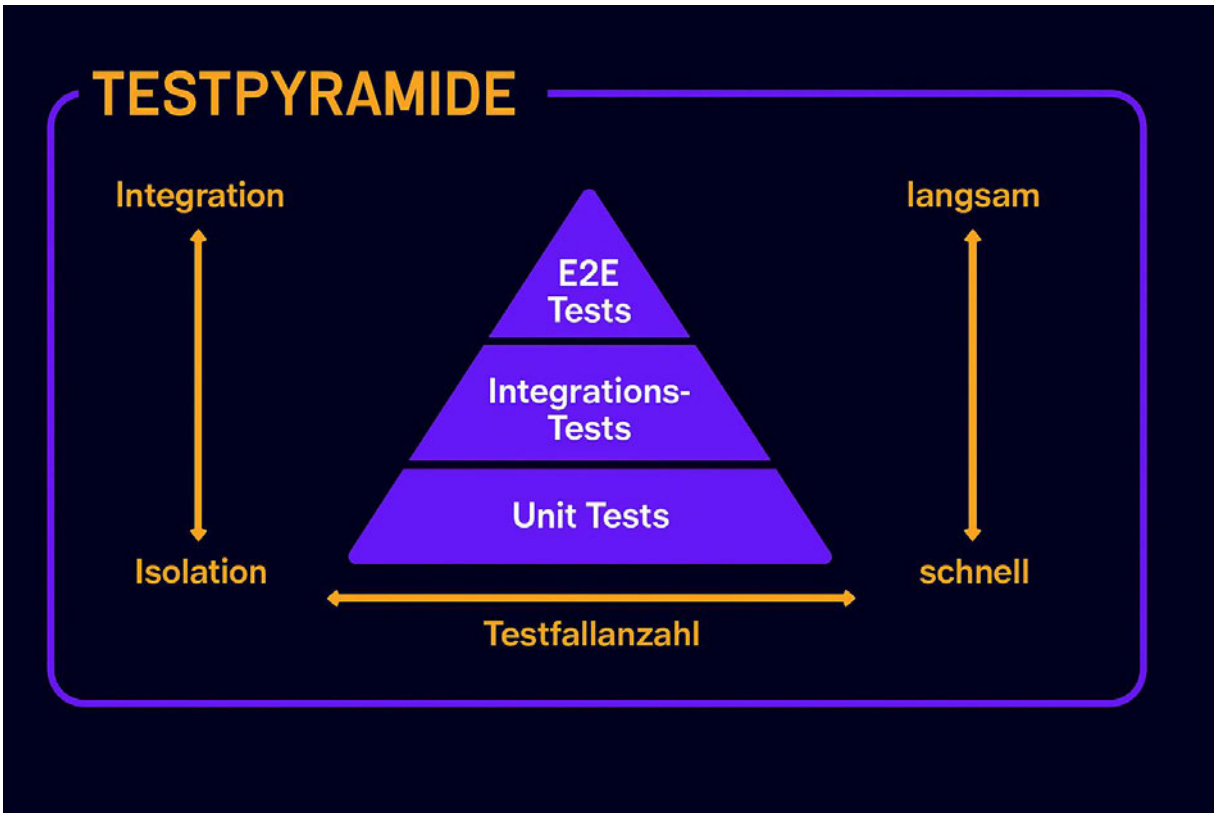


Abbildung 3: Testpyramide [23]

### 2.5.3 Alternative Modelle

Es gibt neben der klassischen Testpyramide auch andere Modelle, die die Testebenen anders gewichten. [24]

- Testdiamant: hebt die Service- und Integrationstests hervor, während die Basis aus Unit-Tests etwas schmaler ist. Dieses Modell eignet sich besonders für Systeme, die eine stark serviceorientierte Architektur nutzen.

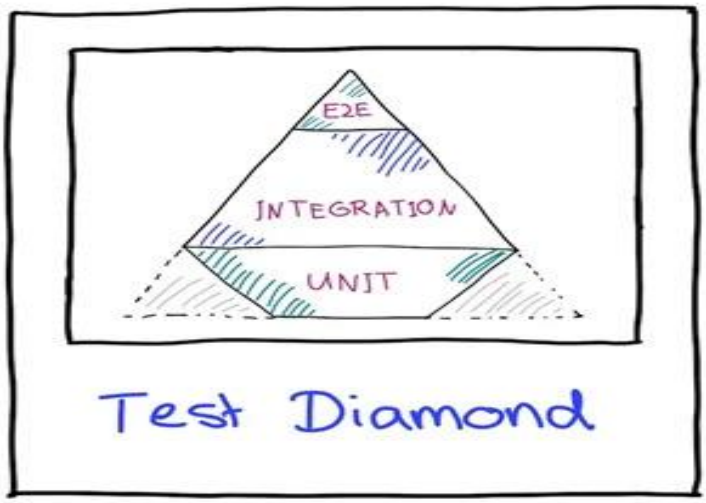


Abbildung 4: Test-Diamant mit Schwerpunkt auf Service- und Integrationstests [30]

- Test-Trophäe: Die mittlere Ebene der Integrationstests wird durch die solide Grundlage der Unit-Tests und die kleine Spitze der End-to-End-Tests noch stärker betont. Dieses Modell stellt eine realitätsnahe Abbildung dar, insbesondere in komponentenbasierten Frontend Architekturen.

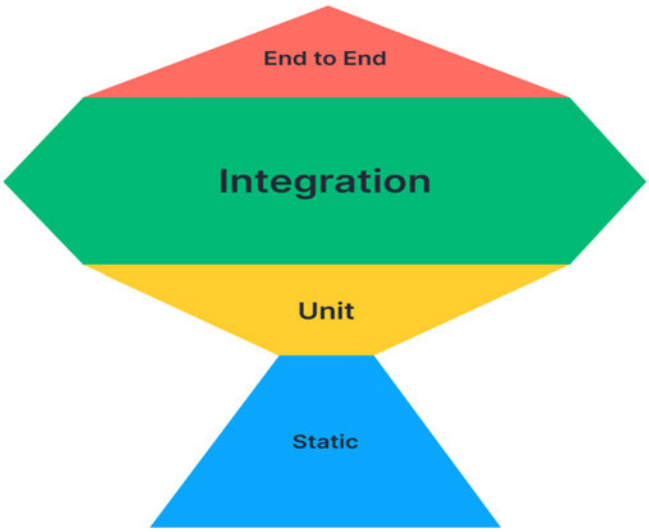


Abbildung 5: Test-Trophäe mit Schwerpunkt auf Integrationstests [31]

- Honeycomb-Modell: bietet eine Erweiterung um weitere Qualitätsdimensionen wie Performance, Sicherheit oder Usability. Sie sind in „Waben“ untergebracht, die jeweils für sich stehen, und zeigen, dass Teststrategien über funktionale Aspekte hinausgehen sollten.


Abbildung 6: Das Honeycomb [32]

- Testeiswaffel bzw. Test-Pizza: Dieses Modell stellt ein Antimuster dar, bei dem der überwiegende Anteil der Tests auf der Benutzeroberfläche (User Interface, UI) angesiedelt ist. Werden nahezu ausschließlich UI-Tests eingesetzt und die unteren Testebenen weitgehend vernachlässigt, führt dies zu langen Ausführungszeiten, erhöhter Fragilität der Tests sowie hohem Wartungsaufwand. Aus diesen Gründen gilt die Testeiswaffel in der Literatur als ineffiziente Teststrategie.

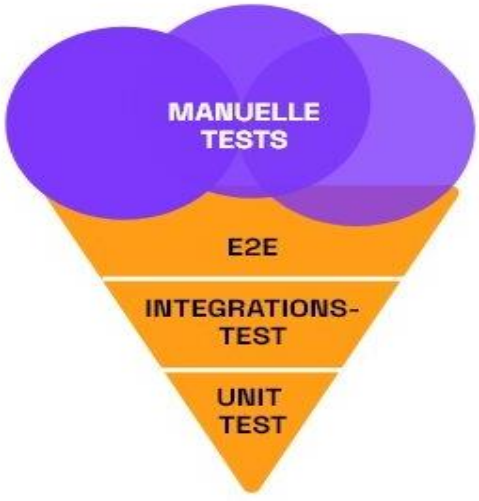


Abbildung 7: Test-Eiswaffel - Anti-Pattern [33]

### 2.5.4 Auswahlkriterien für eine geeignete Strategie

Die Auswahl einer geeigneten Teststrategie in einem Projekt ist kein zufälliger Prozess, sondern wird durch unterschiedliche Rahmenbedingungen und projektspezifische Faktoren bestimmt, wie im Folgenden erläutert wird:
[24] [27] [26]

- Systemarchitektur: Klassische Pyramidenmodelle kommen monolithischen Systemen zugute, während serviceorientierte Architekturen eher dem Testdiamanten folgen.

- Risiko- und Geschäftsrelevanz: Kritische Geschäftsprozesse erfordern eine stärkere Absicherung durch End-to-End-Tests.

- Teamkompetenzen und Toollandschaft: Die vorhandenen Programmiersprachen, Frameworks und CI/CD-Umgebungen legen fest, welche Testebenen effizient automatisiert werden können.

- Nicht-funktionale Anforderungen: Projekte mit hohen Anforderungen an Performance, Sicherheit oder Usability sollten diese Aspekte explizit in die Teststrategie integrieren.

### 2.5.5 Einordnung im Kontext dieser Arbeit

Die Testpyramide ist das Basismodell für die Webanwendung, die in dieser Arbeit betrachtet wird. Sie gibt eine Orientierung zur Verteilung der Tests und fördert agile Vorgehensweisen durch schnelles sowie kontinuierliches Feedback. Die in dieser Arbeit implementierten End-to-End-Tests mit Cypress repräsentieren die oberste Ebene der Testpyramide, die auf die Validierung vollständiger Nutzerflüsse ausgerichtet ist. Eine bessere Abdeckung durch Unit- und Integrationstests wäre zusätzlich sinnvoll, um die Stabilität und Wartbarkeit langfristig zu verbessern.
Es wird evident, dass die gewählte Strategie nicht nur den aktuellen Projektkontext abbildet, sondern auch als Leitfaden für zukünftige Erweiterungen dient.

Tabelle 1: Ziele, Beispieltests und typisches Tooling (eigene Darstellung nach ISTQB CTFL 2018, Fowler 2018/2021)

| Ebene | Ziel | Beispieltests | Typisches Tooling |
|---|---|---|---|
| **Unit /Komponententests** | Korrektheit einzelner Einheiten, schnelle, feingranulare Rückmeldung, günstige Regression | Berechnungslogik, Eingabevalidierung, Fehlerbehandlung, UI Komponenten in Isolation | JUnit / NUnit / PyTest, Jest / Vitest, Jasmine / Karma |

| | | | |
|---|---|---|---|
| **Service /Integrationstests** | Korrektes Zusammenspiel von Komponenten/Services, stabile Schnittstellen/Verträge, Entkopplung vom UI | REST-API Status und Schema, Fehlerpfade, DB-Integration, Messaging, Mocks/Stubs | REST Assured, SuperTest, Postman/Newman,Pact (CDC),WireMock, Testcontainers |
| **UI- /End-to-End Tests** | Realitätsnahe Validierung kompletter Nutzerflüsse, Absicherung von Kernprozessen | Geschäftsfallanlegen/ändern, Formular-Flows, Ende-zu Ende-Prozesse inkl. Navigation | Cypress, Playwright, Selenium WebDriver |
| **Statische Analyse (ergänzend)** | Frühe Erkennung von Qualitätsproblemen ohne Programmausführung, Wartbarkeit erhöhen | Linting, Typprüfungen; Code-Smells, Sicherheits Checks (SAST) | ESLint, TypeScript, SonarQube, Semgrep |
| **Übergreifend (alle Ebenen)** | Nicht-funktionale Qualität: Performance, Sicherheit, Barrierefreiheit,Testdaten/Umgebungen | Page-Load/Latency, Last /Stresstests, OWASP Checks, Accessibility Checks | k6, JMeter, Lighthouse, OWASP ZAP, axe-core, Faker/Testdaten Generatoren |

# 3 Cypress im Vergleich zu Selenium

Die Auswahl geeigneter Testwerkzeuge ist ein entscheidender Faktor, der darüber entscheidet, ob die Testautomatisierung in Softwareprojekten erfolgreich ist. Es gibt eine Vielzahl von Frameworks und Tools auf dem Markt, die verschiedene Ansätze und Stärken haben. Besonders hervorstechen dabei Selenium, das sich über die Jahre als der Standard etabliert hat, und Cypress, ein modernes Testframework, das speziell für Webanwendungen entwickelt wurde.

Seit vielen Jahren ist Selenium das Maß der Dinge für die Automatisierung von webbasierten Tests. Es wird durch die breite Unterstützung für verschiedene Programmiersprachen und Browser ausgezeichnet und ist dank des W3C-WebDriver-Standards in vielen Projekten weltweit im Einsatz. Cypress verfolgt hingegen einen innovativen Ansatz: Es ist stark mit der Browserumgebung verbunden und bietet Entwicklern eine moderne und benutzerfreundliche Umgebung für End-to-End-Tests von Webanwendungen.

Zuerst werden in diesem Kapitel beide Frameworks eingeführt, um ihre Funktionsweise, Anwendungsgebiete und Besonderheiten zu erläutern. Daraufhin wird ein systematischer Vergleich in Bezug auf Architektur, unterstützte Programmiersprachen, Performance, Wartbarkeit und

die Integration in CI/CD-Umgebungen durchgeführt. Die Analyse der jeweiligen Vor- und Nachteile sowie die Diskussion über die Bedeutung für den praktischen Einsatz stehen im Fokus. Zum Schluss wird ein Fazit gezogen, das erklärt, warum Cypress für diese Arbeit ausgewählt wurde.

## 3.1 Einführung in Cypress

Cypress ist ein modernes Framework, das Webanwendungen automatisiert testet und eine Alternative zu traditionellen Testwerkzeugen wie Selenium ist. Es wurde konzipiert, um typische Herausforderungen der Testautomatisierung wie instabile Tests, komplizierte Konfigurationen und geringe Transparenz bei Fehleranalysen direkt zu beheben. Cypress zielt mit einem cleveren Architekturansatz und der engen Verzahnung mit der Browserumgebung hauptsächlich auf Entwickler ab, die im agilen Umfeld schnell und zuverlässig End-to-End-Tests erstellen wollen.

### 3.1.1 Entstehung und Zielsetzung

Cypress wurde 2014 entwickelt, und die erste stabile Version kam 2017 auf den Markt. Cypress wurde im Gegensatz zu vielen anderen Testwerkzeugen, die über die Jahre schrittweise Verbesserungen erfahren haben (wie Selenium von RC über WebDriver zum W3C-Standard), von Anfang an mit dem Ziel entwickelt, die Testautomatisierung für moderne Web-Frontends radikal zu vereinfachen.
Das Hauptziel ist es, eine Plattform zu schaffen, die Tests in der gleichen Sprache wie die Anwendung (JavaScript/TypeScript) ermöglicht, um eine entwicklerfreundliche Umgebung zu schaffen und die Einstiegshürden zu senken. Cypress stellt sich somit gezielt als die Antwort für agile Teams dar, die auf kurze Feedbackzyklen, Continuous Integration und Continuous Delivery setzen. [11] [19]

### 3.1.2 Architektur und Funktionsweise

Die Architektur von Cypress unterscheidet sich wesentlich von klassischen WebDriver-basierten Ansätzen:

- Direkte Ausführung im Browser-Kontext: Cypress arbeitet direkt in der gleichen Laufzeitumgebung wie die Webanwendung, die getestet wird. Es hat direkten Zugriff auf das Document Object Model (DOM), Netzwerk-Requests und Browser-APIs.

- Automatisches Warten: Anders als bei Selenium, wo häufig explizite Wartezeiten eingebaut werden müssen, wartet Cypress automatisch, bis Elemente verfügbar sind oder Seiten geladen sind. So werden „flaky tests“ minimiert.

- Asynchrones Command-Queue-Modell: Alle Testbefehle werden in einer Befehlswarteschlange gesammelt und sequentiell ausgeführt. Das sorgt für deterministische Ergebnisse und eine Struktur.

- Transparente Testausführung: Cypress präsentiert während der Testausführung in einem interaktiven Dashboard jeden Schritt und ermöglicht so Debugging. Testläufe können von

Entwicklern angehalten, untersucht und nachvollzogen werden.

- Integration in CI/CD: Durch Plugins kann Cypress reibungslos in CI/CD-Systeme wie Jenkins, GitLab oder GitHub Actions integriert werden, was eine automatisierte Qualitätssicherung während des Build-Prozesses ermöglicht.

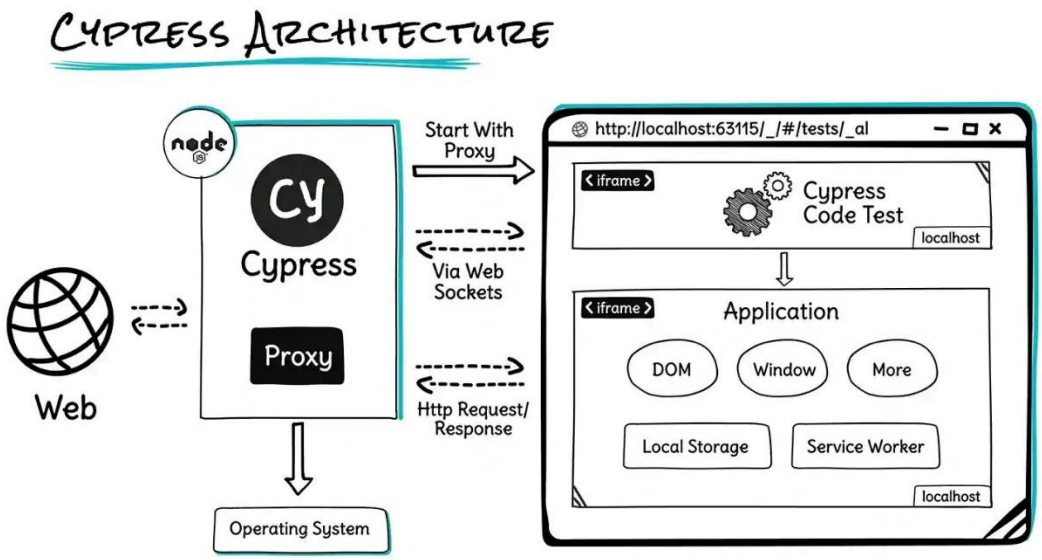


Abbildung 8: Cypress-Architektur [34]

### 3.1.3 Typische Einsatzszenarien

Cypress ist primär für Webanwendungen gedacht und bietet hier vielfältige Einsatzmöglichkeiten:

1. End-to-End-Tests (E2E)
    - Darstellung der gesamten User Journeys, wie zum Beispiel „Login → Warenkorb → Bestellung → Bestätigung“.
    - Validierung von Geschäftsprozessen aus Sicht des Endanwenders.
2. Integrationstests
    - Testen von Schnittstellen zwischen Frontend-Komponenten.
    - Simulation von API-Aufrufen (REST, GraphQL) mit integrierten Stubs und Mocks.
3. Regressionstests in agilen Projekten
    - Automatisierte Wiederholung von Kernfunktionen nach jedem Sprint.
    - Sofortiges Feedback über Auswirkungen von Codeänderungen.
4. CI/CD-Pipelines
    - Einbettung in DevOps-Prozesse für kontinuierliche Tests.

- Generierung von Reports, Screenshots und Videos zur besseren Nachvollziehbarkeit von Fehlern.

### 3.1.4 Stärken und Schwächen

Stärken:

- Einfache Einrichtung: Cypress benötigt zur Installation ein npm-Paket.
- Moderne Entwicklererfahrung: Tests werden in JavaScript/TypeScript geschrieben, was die Einstiegshürde für Webentwickler reduziert.
- Stabile Testausführung: Durch automatisches Warten und direkte Browserintegration treten weniger instabile Tests auf.
- Hohe Transparenz: Interaktives Dashboard, Screenshots und Videoaufzeichnung unterstützen die Fehleranalyse.
- Nahtlose Integration in CI/CD-Workflows.
- JavaScript/TypeScript-Bindung: Cypress ist nicht sprachunabhängig wie Selenium, sondern auf das Web-Ökosystem beschränkt

Grenzen:

- Eingeschränkte Browserunterstützung: Cypress funktioniert am besten in Chromium-basierten Browsern und Firefox, bietet aber weniger Vielfalt als Selenium.
- Kein Multi-Tab-/Multi-Window-Support: Szenarien mit verschiedenen parallelen Browserfenstern sind nur eingeschränkt abbildbar.
- Nur Webfokus: Desktop- oder mobile Apps können nicht direkt getestet werden.
- JavaScript/TypeScript-Bindung: Cypress ist nicht sprachunabhängig wie Selenium, sondern auf das Web-Ökosystem beschränkt.

## 3.2 Einführung in Selenium

Eines der ältesten und zugleich bekanntesten Frameworks zur Automatisierung von Webanwendungstests ist Selenium. Seit seinem Launch im Jahr 2004 ist es zu einem De-facto-Standard geworden und ist die Basis für zahlreiche Teststrategien in Unternehmen und Organisationen rund um den Globus. Selenium ist bis heute ein wichtiges Werkzeug in der Softwarequalitätssicherung, weil es offen ist, keine spezifische Sprache oder Plattform benötigt und in den W3C-WebDriver-Standard integriert ist. [17] [18]

### 3.2.1 Entstehung und Zielsetzung

Ursprünglich wurde Selenium im Jahr 2004 von Jason Huggins bei ThoughtWorks entwickelt, um das Automatisieren von Webanwendungen für wiederkehrende Tests zu ermöglichen. [18]

- Selenium Core (2004): Erste Version, die JavaScript im Browser nutzte, um DOM-Interaktionen zu simulieren.

- Selenium RC (Remote Control, 2006): Einführung eines Server-Proxy-Ansatzes, um Browsersteuerung außerhalb der Sandbox zu ermöglichen.
- Selenium WebDriver (2009): Ablösung von RC durch eine direkte Browsersteuerung über native APIs.
- Selenium 3/4 (2016-2021): Integration des W3C-WebDriver-Standards, verbesserte Browserkompatibilität, vereinheitlichte API.

  Heute ist Selenium 4 das aktuelle Release und vollständig W3C-konform.

### 3.2.2 Architektur und Funktionsweise

Die Architektur von Selenium basiert auf einer Client-Server-Architektur. [17] [18]

- Client Libraries: Entwickler schreiben Tests in einer unterstützten Sprache (z. B. Java, Python, C#, JavaScript, Ruby).
- JSON Wire Protocol / W3C WebDriver: Die Befehle werden an den Browser weitergeleitet, entweder über das alte JSON Wire Protokoll (Selenium 3) oder den standardisierten W3C WebDriver Standard (Selenium 4).
- Browser Driver: Browser-spezifische Treiber wie ChromeDriver, GeckoDriver oder EdgeDriver implementieren das WebDriver-Protokoll und ermöglichen dadurch die programmgesteuerte Interaktion und Steuerung des jeweiligen Browsers.
- Browser Engine: Führt schließlich die Anweisungen aus (Klicks, Eingaben, Navigation).

Diese Architektur macht Selenium flexibel, allerdings auch komplexer in der Einrichtung im Vergleich zu Cypress.

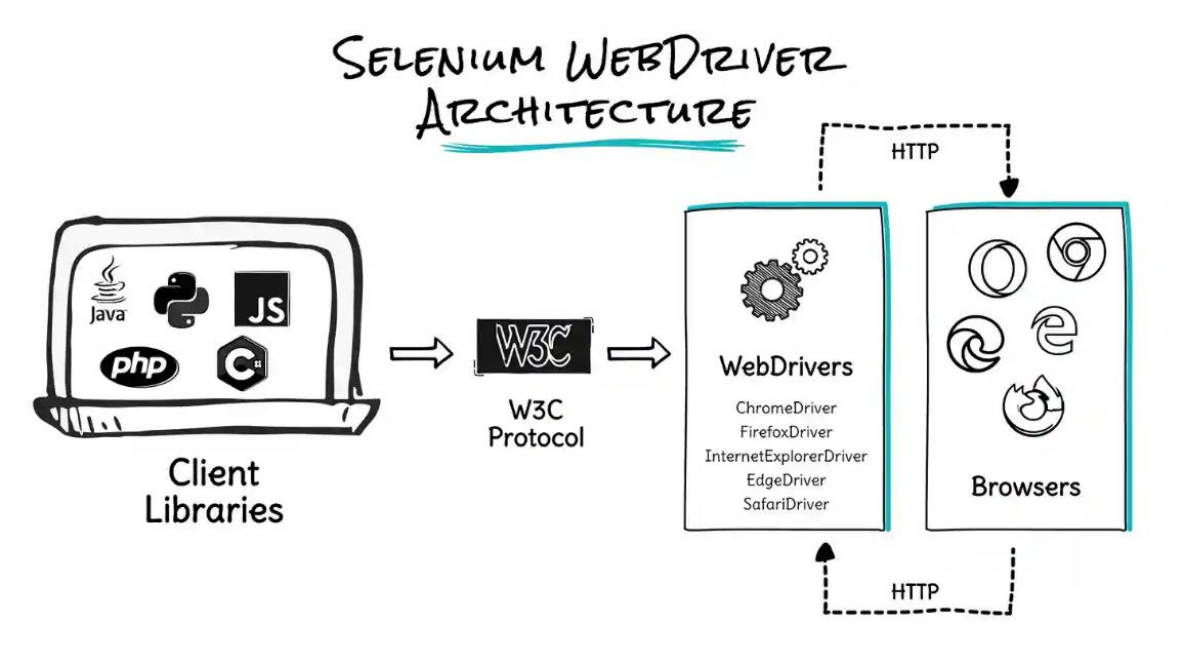


Abbildung 9: Selenium/WebDriver-Architektur [35]

### 3.2.3 Typische Einsatzszenarien

Selenium ist nicht nur ein Framework, sondern Teil eines größeren Ökosystems. [12]

- Selenium WebDriver: Zentrales Modul für die Browsersteuerung.
- Selenium IDE: Eine Recorder- und Playback-Erweiterung für schnelle Tests.
- Selenium Grid: Ermöglicht die parallele Ausführung von Tests auf verteilten Systemen und in unterschiedlichen Browsern.
- Integration in CI/CD: Selenium wird weltweit in Verbindung mit Jenkins, GitLab CI, Azure DevOps und anderen Plattformen eingesetzt.

Typische Einsatzbereiche:

- Cross-Browser-Tests (Chrome, Firefox, Safari, Edge, Opera).
- Regressionstests bei komplexen Websystemen.
- Lastverteilung von Tests über Selenium Grid in Cloud-Umgebungen.
- Verwendung in Kombination mit Testframeworks wie TestNG, JUnit, PyTest oder NUnit.

### 3.2.4 Stärken und Schwächen

Stärken:

- Sprachvielfalt: Unterstützung für Java, C#, Python, Ruby, JavaScript u. a.
- Breite Browserunterstützung durch W3C-WebDriver (Chrome, Firefox, Edge, Safari).
- Hohe Flexibilität: Möglichkeit zur Integration in nahezu jedes Framework oder Tool.
- Skalierbarkeit: Verteilte Testausführung über Selenium Grid oder Cloud-Anbieter (z. B. BrowserStack, Sauce Labs).
- Offener Standard: Durch W3C-Standardisierung ist Selenium zukunftssicher und plattformübergreifend.

Schwächen:

- Komplexere Einrichtung im Vergleich zu Cypress (z. B. Treiberverwaltung, Konfiguration von Grid).
- Wartungsaufwand: Tests sind anfällig für Änderungen im Frontend (z. B. XPath-Anpassungen).
- Fehlende integrierte Features: Debugging, Screenshot- und Videoaufzeichnungen müssen über Dritttools ergänzt werden.
- Flakiness: Instabilität bei zeitabhängigen Tests (z. B. Wartezeiten, dynamische Inhalte), die manuell behandelt werden müssen.

## 3.3 Vergleich: Architektur, Sprache, Performance, Wartung

Die Testautomatisierung von Webanwendungen erfolgt häufig mit den weit verbreiteten Frameworks Selenium und Cypress. Selenium wird als langjährig etablierter Standard angesehen und hat durch die W3C-Standardisierung eine hohe Akzeptanz gefunden, während Cypress ein modernes Framework ist, das durch einen innovativen Architekturansatz besonders für die Bedürfnisse der agilen Webentwicklung geeignet ist. Eine systematische Gegenüberstellung beider Werkzeuge erfolgt in Bezug auf Architektur, unterstützte Programmiersprachen, Performance, Wartbarkeit und Integration in aktuelle Entwicklungsprozesse.

### 3.3.1 Architekturunterschiede

- Selenium basiert auf einer Client-Server-Architektur. Der W3C WebDriver Standard regelt, dass Befehle vom Testskript über einen Browser-Treiber an die Browser-Engine vermittelt werden. Das schafft Flexibilität, erhöht aber auch die Komplexität. [17]

- Im Gegensatz dazu arbeitet Cypress direkt im Browserkontext. Tests werden in der gleichen Laufzeitumgebung wie die Anwendung ausgeführt. Eine engere Integration und stabilere Tests sind möglich, weil die zusätzliche Kommunikationsschicht entfällt. [11]

### 3.3.2 Unterstützte Programmiersprachen

- Selenium ist mit mehreren Programmiersprachen kompatibel, darunter Java, C#, Python, Ruby, JavaScript und Kotlin. Das macht es besonders interessant für heterogene Entwicklungsteams. [18]

- Cypress fokussiert hauptsächlich auf das Ökosystem von JavaScript und TypeScript. Obwohl diese Spezialisierung die Integration in moderne Webprojekte erleichtert, mindert sie die Flexibilität für Teams, die andere Programmiersprachen nutzen. [11]

### 3.3.3 Performance, Stabilität und Flakiness

- Selenium gilt als vielseitig, leidet aber häufiger unter Schwierigkeiten wie instabilen Tests („flaky tests“), da Synchronisationsprobleme und Wartezeiten oft manuell beseitigt werden müssen. [19] [16] [20]

- Aufgrund der automatischen Wartefunktion auf Elemente und der engen Integration mit Browsern bietet Cypress eine bessere Stabilität. Forschungen wie die von Bonigarcia (QUATIC 2024) belegen, dass Cypress in End-to-End-Szenarien die Fehlerraten signifikant senkt und die Ausführungszeiten verkürzt. [19] [16] [20]

### 3.3.4 Wartungsaufwand und Testbarkeit

- Selenium-Tests sind oft wartungsintensiv, da Änderungen an der DOM-Struktur (z. B. neue IDs oder XPath-Anpassungen) Tests leicht brechen können. Frameworks wie Page Object Models können den Wartungsaufwand reduzieren, aber er bleibt dennoch ein kritischer Faktor. [18]

- Cypress verringert den Wartungsaufwand durch automatische Synchronisation und eingebaute Funktionen wie Snapshot-Vergleiche, Debugging-Oberflächen und integriertes Error-Handling. [11]

### 3.3.5 Integration in CI/CD-Pipelines und DevOps

- Selenium lässt sich flexibel in nahezu jedes Build- oder Deployment-System integrieren (z. B. Jenkins, GitLab, Azure DevOps). Mit Selenium Grid oder Cloud-Diensten ist die parallele Ausführung in verschiedenen Umgebungen möglich. [18]
- Cypress bietet eine nahtlose Integration in CI/CD-Systeme, ist aber stärker auf die Webentwicklung zugeschnitten. Das kommerzielle „Cypress Dashboard“ ermöglicht zusätzlich Reporting, Parallelisierung und Analysen. [11]

Tabelle 2: Vergleichstabelle Cypress vs. Selenium

| Kriterium | Selenium | Cypress |
|---|---|---|
| **Architektur** | Client-Server über W3C WebDriver, zusätzliche Treiberschicht notwendig | Läuft direkt im Browser, eng integriert |
| **Sprachen** | Java, C#, Python, Ruby, JavaScript, Kotlin u. a. | Nur JavaScript/TypeScript |
| **Performance** | Stabil, aber flakiness durch manuelle Synchronisation möglich | Schneller und stabiler durch automatisches Warten |
| **Wartung** | Höherer Aufwand, XPath/Selektoren brechen bei UI-Änderungen | Geringerer Aufwand, integriertes Debugging & Snapshot-Mechanismen |
| **CI/CD** | flexibel, Grid & Cloud-Scaling möglich | Einfach integriert, kommerzielles Dashboard für erweitertes Reporting |
| **Community** | Umfangreich, etabliert, breites Ökosystem | Wachsende Community, moderner Fokus auf Frontend-Teams |

## 3.4 Fazit zum Vergleich

Im Anschluss an die Vorstellung der beiden Frameworks und den Vergleich ihrer Eigenschaften werden die wichtigsten Unterschiede zusammengefasst, es wird eine Empfehlung für den Einsatz je nach

Projektszenario diskutiert, und die Bedeutung der Entscheidung für Cypress im Rahmen dieser Arbeit wird erläutert. [16]

### 3.4.1 Zusammenfassung der Unterschiede

Die Unterschiede zwischen Selenium und Cypress liegen grundlegend in der Architektur, der Zielgruppe und dem Einsatzgebiet. Selenium verwendet den W3C WebDriver Standard und basiert auf einer Client Server-Architektur, was es ermöglicht, viele verschiedene Browser und Programmiersprachen zu unterstützen. Im Gegensatz dazu geht Cypress eine direkte Integration in den Browserkontext ein und ist speziell auf JavaScript/TypeScript ausgerichtet. Selenium punktet mit seiner Flexibilität und Skalierbarkeit, während Cypress eine moderne Entwicklererfahrung mit Funktionen wie automatischem Warten, Debugging und Reporting bietet.
Ein weiterer Unterschied betrifft den Wartungsaufwand: Selenium-Tests müssen oft angepasst werden, weil sie empfindlich auf Änderungen in der DOM-Struktur reagieren, während Cypress diesen Aufwand durch integrierte Synchronisationsmechanismen verringert.

### 3.4.2 Empfehlung nach Projektszenario

- Selenium ist besonders geeignet für großangelegte, heterogene Projekte, die mehrere Sprachen und Browserumgebungen unterstützen müssen. Selenium ist nach wie vor die erste Wahl, besonders wenn es um Cross-Browser-Testing oder das parallele Ausführen von Tests über Selenium Grid oder Cloud-Dienste geht.

- In agilen Projekten, die auf schnelles Feedback angewiesen sind und wo die Anwendungen hauptsächlich im Web-Frontend entwickelt werden, ist Cypress die bevorzugte Lösung. Bei solchen Projekten sind Stabilität, Wartungsfreundlichkeit und eine moderne Entwicklererfahrung wichtiger als Sprach- oder Browservielfalt.

### 3.4.3 Relevanz für diese Arbeit

Für diese Arbeit fiel die Wahl bewusst auf Cypress, da es sich bei dem zu testenden System um eine moderne Webanwendung handelt und der Schwerpunkt auf der Durchführung von End-to-End-Tests liegt.
Mit Cypress ist es möglich, realistische Abbildungen kompletter Nutzerflüsse zu erstellen und dabei von Vorteilen wie automatischer Synchronisation, hoher Stabilität und einfacher Integration in CI/CD-Prozesse zu profitieren. Ein weiterer Grund für die Entscheidung war, dass Cypress im Vergleich zu Selenium eine deutlich bessere Handhabung bietet, was für die Arbeit im Rahmen einer Bachelorarbeit mit begrenztem Zeitrahmen entscheidend ist. Obwohl Selenium das allgemeinere Tool ist, überwiegen im spezifischen Projektszenario dieser Arbeit die Vorteile von Cypress.

# 4 Projektumfeld

Das Kapitel beschreibt das Projektumfeld der Testautomatisierung, die durchgeführt wurde. Ziel ist es, die Rahmenbedingungen und die zu testende Webanwendung darzustellen, die Anforderungen an die Testautomatisierung zu erläutern, die Auswahl der Testfälle zu begründen und die angewandte Teststrategie darzulegen. So wird der theoretische Abschnitt der Arbeit in einen praktischen Rahmen gesetzt und bildet die Basis für die folgende Umsetzung.

## 4.1 Beschreibung der zu testenden Webanwendung

Die Webanwendung IntelMod wurde im Rahmen dieser Arbeit getestet. Die Anwendung ist für Vermieter:innen und Mieter:innen gleichermaßen gedacht und hat für beide Gruppen eigene Funktionsbereiche.
Die Vermietersicht unterstützt bei der Planung und Umsetzung von Modernisierungsmaßnahmen an Wohngebäuden. Zentrales Element ist das funktionale Kostensplitting, ein Verfahren, das Modernisierungskosten nach Funktionsverbesserung und Funktionserhaltung aufschlüsselt. Dadurch soll eine faire und transparente Kostenverteilung erreicht werden, die sowohl den gesetzlichen Vorgaben entspricht als auch den Interessen beider Parteien gerecht wird. Die Plattform ermöglicht es Vermieter:innen, die erforderlichen Daten effizient zu erfassen und auf dieser Basis präzise Berechnungen durchzuführen. Transparenz, Nachvollziehbarkeit und Fairness stehen dabei im Vordergrund.
Die Mietersicht verfolgt das Ziel, Mieter:innen aktiv in den Prozess einzubeziehen. Über eine benutzerfreundliche Oberfläche können die geplanten Maßnahmen nachvollzogen, interaktiv in einem 3D-Modell betrachtet und bei Bedarf angepasst oder durch eigene Vorschläge ergänzt werden. Darüber hinaus stellt die Anwendung Energieeinsparungen, Kosten und potenzielle Mieterhöhungen übersichtlich dar. Die Kommunikation mit dem Vermieter erfolgt direkt über die Plattform, sodass ein kontinuierlicher Austausch und ein gemeinsames Verständnis gefördert werden.
IntelMod hilft, die digitale Unterstützung für energetische Sanierungsprozesse zu verbessern, indem es die Sichtweisen von Vermieter:innen und Mieter:innen auf einer gemeinsamen Plattform zusammenbringt und für Transparenz durch nachvollziehbare Berechnungslogiken und visuelle Darstellungen sorgt.

## 4.2 Anforderung an die Testautomatisierung

Im Projekt IntelMod hat die Testautomatisierung das Ziel, die zentrale Funktionalität der Webanwendung aus Sicht der Nutzer abzusichern. Wegen der dualen Struktur der Plattform (Vermieter- und Mietersicht) liegt der Fokus auf der Prüfung von durchgängigen End-to-End-Szenarien, die den typischen Arbeitsablauf der beiden Zielgruppen abbilden.

Die grundlegenden Anforderungen sind:

- Abbildung geschäftskritischer Flows: Absicherung der Prozesse, die für die Transparenz und Nachvollziehbarkeit der Modernisierung besonders relevant sind.
- Realitätsnahe Tests: Nutzung der Oberfläche, um die Anwendung so zu prüfen, wie sie später von Vermieter:innen und Mieter:innen bedient wird.
- Deterministische Ergebnisse: Jeder Test muss ein nachvollziehbares definiertes Soll-Ergebnis haben und reproduzierbar sein.
- Stabilität: Vermeidung von Abhängigkeiten zwischen den Testfällen, um isolierte Ausführungen zu ermöglichen.
- Nachvollziehbarkeit: Verständliche Struktur der Testfälle mit Bezug auf die vorliegenden Excel-Testpläne, die Soll- und Ist-Ergebnisse dokumentieren.

Unit- und Integrationstests wurden im Rahmen dieser Arbeit nicht umgesetzt, da der Fokus auf der Erprobung von Cypress für E2E-Tests liegt.

## 4.3 Auswahl der Testfälle

Die Testfallauswahl erfolgte anhand der zwei bestehenden Excel-Testpläne für die Vermieter- und Mietersicht. Sie umfassen eine strukturierte Auflistung von Testfallnummer, Testschritten, erwarteten Ergebnissen, tatsächlichen Ergebnissen und Status. Die geschäftskritischsten Abläufe, die den Kern des Funktionalen Kostensplittings und der Interaktion zwischen Vermieter:innen und Mieter:innen bilden, wurden zur Automatisierung ausgewählt.

Vermietersicht, zentrale Testfälle:

- Anlegen eines Falls und Auswahl der Rolle (Vermieter): stellt die Grundlage für alle Folgeschritte dar.
- Erfassung von Gebäudedaten und Wohnungen: Basiseingaben für alle weiteren Berechnungen.
- Anlage und Auswahl von Modernisierungsmaßnahmen: zentrales Element für das Funktionale Kostensplitting.
- Darstellung von Kostenübersichten: Transparenz und Nachvollziehbarkeit für Mieterhöhungen.
- Generierung und Versand einer Modernisierungsankündigung: rechtlich relevanter Endpunkt des Vermieterprozesses.

Mietersicht, zentrale Testfälle:

- Start und Einführung: Sichtbarkeit zentraler Hinweise und Hilfefunktionen.
- Prüfung und Korrektur von Gebäude- und Wohnungsdaten: aktive Einbindung des Mieters in die Validierung.
- Auswahl oder Anpassung von Modernisierungsmaßnahmen: direkte Mitwirkung am Vorhaben.

- Einsicht in Übersichten zu Energieeinsparungen, Kosten und Nachhaltigkeit: Transparenz für die Mieterseite.
- Abschluss und Übermittlung an den Vermieter: vollständiger Ende-zu-Ende-Fluss.

Die Testfälle, die gewählt wurden, sind die kritischen Hauptpfade („Happy Paths“) der Anwendung, welche für die Funktionsfähigkeit der Plattform und das Vertrauen der Nutzer:innen entscheidend sind. Da das Hauptaugenmerk dieser Arbeit auf der praktischen Demonstration der E2E-Testautomatisierung mit Cypress liegt, wurden weniger zentrale oder rein technische Szenarien (z. B. Edge Cases) nicht berücksichtigt.

## 4.4 Teststrategie und Planung

Die Testpyramide, die in Abschnitt 2.5 erläutert wurde, stellt die Grundlage für das Konzept. In der praktischen Umsetzung dieser Arbeit wurde jedoch bewusst der Fokus auf End-to-End-Tests (E2E) gelegt, weil sie für die zu testende Webanwendung die geschäftskritischen Kernprozesse aus der Perspektive von Vermieter:innen und Mieter:innen am besten abbilden.
Die Strategie dieser Arbeit folgt somit dem Prinzip, die Spitze der Pyramide gezielt einzusetzen, um die wichtigsten Nutzerflüsse abzusichern. Unit- und Integrationstests, die in einer vollständigen Testpyramide üblicherweise die Basis und Mitte bilden, wurden im Rahmen dieses Projekts nicht implementiert, da der Fokus auf der praktischen Erprobung von Cypress als End-to-End-Framework lag.

Konkret bedeutet das:

- E2E/UI-Tests (umgesetzt): Absicherung durchgehender Nutzerflüsse (z. B. Fall anlegen, Daten erfassen, Maßnahmen auswählen, Kostenübersicht prüfen, Ankündigung versenden auf Vermieterseite sowie Datenprüfung, Korrektur, Maßnahmenauswahl und Abschluss auf Mieterseite).
- Service-/Integrationstests (nicht umgesetzt): In strategischen Modellen vorgesehen, aber nicht Teil dieser Arbeit.
- Unit-/Component-Tests (nicht umgesetzt): In der Praxis relevant, aber im Projektumfang nicht berücksichtigt.

Die Planung der Tests orientierte sich an den zuvor definierten Excel-Testplänen für Vermieter- und Mietersicht. Diese dienten als Referenz für die zu automatisierenden Abläufe. Um eine stabile und nachvollziehbare Testausführung sicherzustellen, wurde darauf geachtet, dass die E2E-Tests deterministisch sind, voneinander getrennte Szenarien abbilden und eindeutige Soll-Ergebnisse validieren. [7] [8]
In diesem Sinne verfolgt die Teststrategie dieser Arbeit eine fokussierte E2E-Strategie, die zeigt, wie Cypress seine Stärken zur Abbildung realer Geschäftsprozesse ausspielen kann. Die theoretische Einordnung in die Testpyramide macht deutlich, dass eine umfassendere Teststrategie weitere Ebenen berücksichtigen würde, die jedoch nicht Teil der praktischen Umsetzung sind.

# 5 Implementierung der Tests mit Cypress

Nachdem das Projektumfeld, die Anforderungen und die gewählte Teststrategie im vorherigen Kapitel behandelt wurden, folgt jetzt die Darstellung der praktischen Umsetzung der Testautomatisierung. Dieses Kapitel behandelt die technische Durchführung der Tests mit Cypress. Das umfasst das Einrichten der Testumgebung, die Projektstrukturierung, das Entwickeln von repräsentativen Testfällen sowie den Umgang mit dynamischen Inhalten und asynchronen Prozessen. Abschließend wird auf die Protokollierung und Analyse der Testergebnisse eingegangen.

## 5.1 Einrichtung der Testumgebung

Für die Implementierung der Testautomatisierung wurde eine lokale Entwicklungs- und Testumgebung eingerichtet. Die zu testende Webanwendung IntelMod ist nicht als öffentlich zugänglicher Dienst verfügbar, sondern wird lokal gestartet. Erst nach erfolgreichem Hochfahren der Anwendung kann über den Browser auf die verschiedenen Funktionalitäten für Vermieter- und Mietersicht zugegriffen werden. Die gesamte Umgebung wurde mit Visual Studio Code (VS Code) als zentraler Entwicklungsumgebung realisiert. VS Code diente dabei sowohl zum Starten der Webanwendung als auch zur Erstellung und Ausführung der Cypress-Tests.
Für die Testautomatisierung wurde Cypress über das Node.js-Paketmanagementsystem installiert. Auf diese Weise ließ sich das Framework direkt in die bestehende JavaScript-Umgebung integrieren. Die Konfiguration ermöglichte es, die Tests entweder im interaktiven Modus mit grafischer Oberfläche oder im Headless-Modus für automatisierte Ausführungen auszuführen.

Die wichtigsten Schritte zur Einrichtung sind:

- Installation von Node.js und npm als technische Grundlage,
- Einrichtung des Cypress-Testframeworks im Projekt,
- Verwendung von Visual Studio Code als zentrales Werkzeug für Entwicklung, Testausführung und Verwaltung der Projektdateien,
- Lokales Starten der Anwendung vor Testbeginn, um die Erreichbarkeit der einzelnen Ansichten sicherzustellen.

So entstand eine Umgebung, in der die Webanwendung und die automatisierten Tests eng zusammen entwickelt, gestartet und ausgeführt werden konnten. [11]

## 5.2 Struktur des Testprojekts

Nach der Einrichtung von Cypress wurde die Projektstruktur so gestaltet, dass eine nachvollziehbare Trennung der Testfälle und eine einfache Wartung gewährleistet sind. Cypress erzeugt beim ersten Start automatisch eine Standardstruktur mit den zentralen Verzeichnissen /cypress/e2e (End-to-End-Tests), /cypress/fixtures (Testdaten), /cypress/support (Hilfsfunktionen und Hooks) sowie einer globalen

Konfigurationsdatei. Diese Basis wurde für das vorliegende Projekt angepasst und erweitert. Die Testfälle wurden in zwei Hauptbereiche gegliedert:

- Vermietersicht: umfasst die Abläufe, die von Vermieter:innen durchgeführt werden können (z. B. Fall anlegen, Gebäude- und Wohnungsdaten erfassen, Maßnahmen anlegen, Kostenübersichten anzeigen, Ankündigung generieren).
- Mietersicht: beinhaltet die Szenarien, die den Mieter:innen zur Verfügung stehen (z. B. Einführung, Validierung und Korrektur von Angaben, Auswahl und Annahme von Maßnahmen, Übersicht von Kosten und Energieeinsparungen, Abschluss und Übermittlung).

Für beide Bereiche wurden eigenständige Spec-Dateien angelegt (E2E_INTELMOD-Landlord.cy.js, E2E_INTELMOD-Renter.cy.js).
Dies ermöglicht eine gezielte Ausführung der Tests je nach Perspektive sowie eine bessere Übersichtlichkeit.
Die Cypress-Standardstruktur umfasst unter anderem Verzeichnisse für End-to-End-Tests, Testdaten und Hilfsfunktionen. Für das Projekt relevant waren insbesondere die E2E-Spezifikationen, die getrennt nach Vermieter- und Mietersicht implementiert wurden.
Die resultierende Struktur ist damit nicht nur übersichtlich, sondern auch erweiterbar. Neue Testszenarien können ohne größeren Anpassungsaufwand hinzugefügt werden, da die Trennung nach Vermieter- und Mietersicht von Beginn an systematisch berücksichtigt wurde.

## 5.3 Entwicklung der Testfälle

Die Entwicklung der Testfälle erfolgte auf Grundlage der in Kapitel 4 beschriebenen Excel-Testpläne für Vermieter- und Mietersicht. Aus den dort dokumentierten Testschritten wurden automatisierte End to-End-Tests in Cypress abgeleitet. Ziel war es, die wesentlichen Nutzerflüsse der Anwendung IntelMod realistisch und reproduzierbar abzubilden.

1. Vorgehen bei der Testentwicklung
   Die Testfälle wurden schrittweise in Cypress-Spezifikationsdateien implementiert. Dabei orientierte sich die Struktur eng an den fachlichen Abläufen:
   - Vermietersicht: Erstellung eines neuen Falls, Eingabe von Gebäude- und Wohnungsdaten, Auswahl und Anlage von Maßnahmen, Prüfung von Kostenübersichten sowie Generierung und Versand einer Modernisierungsankündigung.
   - Mietersicht: Einstieg und Einführung, Prüfung und Korrektur von Daten, Auswahl oder Anpassung von Maßnahmen, Einsicht in Kosten- und Nachhaltigkeitsübersichten sowie Übermittlung an den Vermieter.
2. Umsetzung in Cypress
   Die Tests wurden mit Hilfe von XPath-Selektoren und CSS-Attributen entwickelt, um gezielt auf UI-Elemente zuzugreifen. Jeder Testfall besteht aus einer Abfolge von Aktionen (z. B.

Klicks, Eingaben, Navigation) und anschließenden Assertions, die den erwarteten Zustand prüfen. Beispiele hierfür sind:

- Validierung von Überschriften und Labels zur Sicherstellung der korrekten Seitennavigation,
- Eingabe von Daten in Formularfelder und Kontrolle der Übernahme,
- Auswahl von Optionen in Dropdowns oder Buttons,
- Prüfung von generierten Übersichten (Kosten, Maßnahmen, Energieeinsparungen).

3. Kriterien für die Umsetzung

   Bei der Umsetzung wurden folgende Prinzipien beachtet:

   - Isolierung der Testfälle: Jeder Test ist unabhängig von den anderen lauffähig.
   - Deterministische Ergebnisse: Erwartete Werte und UI-Zustände sind definiert und überprüfbar.
   - Lesbarkeit und Nachvollziehbarkeit: Die Testschritte sind so dokumentiert, dass sie den Excel-Testfällen eindeutig zugeordnet werden können.
   - Robustheit: Verwendung stabiler Selektoren, wo möglich, um den Wartungsaufwand bei Änderungen am Frontend zu minimieren.

Damit konnten die aus den Excel-Plänen abgeleiteten Tests in umsetzbare Cypress-Skripte überführt werden, die die kritischen End-to-End-Flows für beide Zielgruppen abdecken.

## 5.4 Umgang mit dynamischen Inhalten und Asynchronität

Da die getestete Anwendung teilweise auf dynamischen Inhalten basiert, beispielsweise durch Ladezeiten von Komponenten, Interaktionen mit Dropdown Menüs oder nachgeladene Formularfelder, mussten die Tests entsprechend sorgfältig gestaltet werden. Cypress hat dafür integrierte Funktionen, wie das automatische Warten auf Elemente, die Retry-Logik bei Assertions und die Möglichkeit, gezielte Wartezeiten (cy.wait()) einzufügen, wenn es durch asynchrone Prozesse notwendig ist. Eindeutige Selektoren kamen außerdem zum Einsatz, um flüchtige Änderungen im DOM zuverlässig zu erfassen. So blieb die Stabilität der Testausführung auch bei unterschiedlichen Ladezeiten gewahrt. [11]

## 5.5 Reporting und Analyse der Testergebnisse

Ein wesentlicher Bestandteil der Testautomatisierung ist die Nachvollziehbarkeit der Ergebnisse. Cypress stellt hierfür standardmäßig umfangreiche Möglichkeiten bereit:

- Test-Reports: Jeder Testdurchlauf liefert eine strukturierte Ausgabe mit Statusinformationen (bestanden/fehlgeschlagen) und einer Übersicht über die einzelnen Testschritte.
- Screenshots: Bei fehlschlagenden Tests werden automatisch Screenshots erzeugt, die den Zustand der Benutzeroberfläche zum Zeitpunkt des Fehlers dokumentieren.

- Videos: Zusätzlich können komplette Testläufe als Video aufgezeichnet werden, wodurch Abläufe und Fehlerquellen im Nachhinein detailliert analysiert werden können.

Diese Möglichkeiten sind eine Hilfe für die Fehlerdiagnose und erlauben es, die Testergebnisse transparent zu kommunizieren. Sie sind besonders wichtig, wenn die Tests in CI/CD-Pipelines eingebunden werden, weil sie es ermöglichen, dass Testergebnisse reproduzierbar und überprüfbar sind, selbst wenn keine manuelle Ausführung stattgefunden hat. [11]

# 6 Evaluation und Diskussion

Im Anschluss an die Implementierung der Tests mit Cypress folgt nun die Auswertung und kritische Betrachtung der Ergebnisse. Dieses Kapitel beleuchtet, in welchem Umfang die definierten Testfälle erfolgreich umgesetzt werden konnten, welche Herausforderungen im Projektverlauf aufgetreten sind und welche Lösungsansätze angewendet wurden. Darüber hinaus wird reflektiert, welchen Beitrag der Einsatz von Cypress im spezifischen Projektkontext geleistet hat und welche Stärken und Schwächen sich im praktischen Einsatz gezeigt haben.

## 6.1 Bewertung der Testergebnisse

Die in Kapitel 4 festgelegten Kernprozesse der Webanwendung IntelMod wurden durch die automatisierten End-to-End-Tests vollständig erfasst. Die vorgesehenen Abläufe von der Fallanlage und Datenerfassung über die Auswahl von Modernisierungsmaßnahmen bis hin zur Kostenübersicht und zum Abschluss konnten aus der Perspektive sowohl der Vermieter als auch der Mieter erfolgreich getestet werden.

Alle die implementierten Testfälle liefen wie erwartet, was es ermöglichte, die wesentlichen Geschäftsprozesse der Anwendung zuverlässig zu bestätigen. Die Ergebnisse zeigen, dass die Anwendung in den getesteten Bereichen funktionsfähig und stabil ist. Cypress hat sich als ein geeignetes Werkzeug präsentiert, um diese Abläufe reproduzierbar und transparent zu validieren.

Ein besonderes Merkmal ist die Nachvollziehbarkeit der Testdurchläufe: Dank der Kombination aus einer konsistenten Teststruktur, automatisierter Dokumentation durch Reports sowie den generierten Screenshots und Videos ist eine überprüfbare Ergebnissicherung möglich. So wurde die Qualität der getesteten Szenarien ohne Abweichungen vom erwarteten Verhalten bestätigt.

Insgesamt bestätigen die Ergebnisse, dass die durchgeführten Cypress-Tests die definierten Ziele der Arbeit erreicht haben: die Abbildung und Absicherung der geschäftskritischen End-to-End-Flows im Projektkontext.

## 6.2 Herausforderung und Lösungsansätze

Während der Entwicklung der Tests wurden keine fachlichen Fehler in der Anwendung, die getestet werden sollte, bemerkt. Trotzdem traten einige technische Schwierigkeiten auf, die sich aus der praktischen Umsetzung der End-to-End-Tests mit Cypress ergeben haben und die durch gezielte Lösungsansätze erfolgreich gemeistert werden konnten:

- Umgang mit dynamischen Inhalten: Da bestimmte Seiteninhalte erst nach Interaktionen oder Ladeprozessen verfügbar wurden, war es notwendig, die integrierten Warte- und Retry-Mechanismen von Cypress zu nutzen. Durch diese Funktionen konnten die Tests stabil ausgeführt werden, ohne dass unnötige manuelle Wartezeiten eingebaut werden mussten.
- XPath-Selektoren für UI-Elemente: Viele UI-Elemente der Anwendung ließen sich nur über komplexe XPath-Ausdrücke adressieren. Dies erhöhte zunächst den Wartungsaufwand, da Änderungen an der Oberfläche leicht zu fehlschlagenden Tests führen können. Durch die konsequente Verwendung eindeutiger Selektoren und eine nachvollziehbare Strukturierung der Testschritte konnte die Stabilität jedoch gewährleistet werden.
- Lokale Ausführung der Anwendung: Da die Webanwendung ausschließlich lokal verfügbar war, musste vor jeder Testausführung sichergestellt werden, dass die Anwendung in einer stabilen Umgebung gestartet ist. Dies erforderte eine enge Verzahnung von Entwicklungsumgebung (Visual Studio Code) und Testframework.
- Testdaten und Eingaben: Viele Tests erforderten die Eingabe von strukturierten Daten (z. B. Adressen, Gebäudedaten, Kosten). Um die Wiederholbarkeit zu gewährleisten, wurden diese Eingaben definiert und standardisiert, sodass konsistente Ergebnisse erzielt werden konnten.

Die Erfahrungen zeigen, dass Cypress trotz seiner hohen Benutzerfreundlichkeit und der integrierten Mechanismen eine sorgfältige Testgestaltung braucht, um langfristig Stabilität und Wartbarkeit zu gewährleisten. Mit den erwähnten Lösungsansätzen wurden alle Herausforderungen erfolgreich bewältigt und stabile Testergebnisse erzielt.

## 6.3 Reflexion über den Einsatz von Cypress im Projektkontext

Im Projekt erwies sich der Einsatz von Cypress als geeignete Lösung zur Automatisierung von End-to End-Tests. Besonders die übersichtliche Syntax, die enge Integration mit JavaScript und die benutzerfreundliche Oberfläche erleichterten die Erstellung und Ausführung der Tests. Dadurch konnte die Testentwicklung ohne größere Einstiegshürden umgesetzt und gezielt an die spezifischen Abläufe der Anwendung IntelMod angepasst werden.

Die Möglichkeit, Tests interaktiv im Cypress Test Runner auszuführen, war besonders wertvoll. Sie erlaubte es, während der Entwicklung sofort Feedback zu geben und Fehlerquellen schnell zu erkennen. Außerdem war die Option, die Tests headless auszuführen, eine nützliche Funktion, um sie ohne manuelle Interaktion und dennoch reproduzierbar zu gestalten.

Außerdem haben sich die automatisch erstellten Screenshots als hilfreiches Werkzeug erwiesen, um Testergebnisse transparent zu dokumentieren und die Nachvollziehbarkeit der Abläufe zu

gewährleisten. Die hohe Qualität und Verlässlichkeit der Testergebnisse ist durch diese Funktionalität gewährleistet worden.

Es wurde auch deutlich, dass die Stabilität der UI-Selektoren einen relevanten Einfluss auf die Tests hat. Um die Wartungskosten nicht unnötig zu erhöhen, ist es wichtig, dass die Selektoren sorgfältig gestaltet werden, wenn Änderungen an der Benutzeroberfläche vorgenommen werden. Obwohl Cypress mit seinen eingebauten Warte- und Retry-Mechanismen viele Probleme bei dynamischen Inhalten löst, ist dies ein Punkt, der in zukünftigen Projekten mehr berücksichtigt werden sollte.

Insgesamt lässt sich festhalten, dass Cypress die definierten Projektziele „die Abbildung und Automatisierung der geschäftskritischen End-to-End-Prozesse“ zuverlässig unterstützt hat.
Es war in diesem Kontext die richtige Entscheidung und hat seine Stärken als praxisnahes Framework für Webanwendungen überzeugend demonstriert.

# 7 Fazit und Ausblick

In diesem abschließenden Kapitel werden die zentralen Ergebnisse der Arbeit zusammengefasst, die Forschungsfrage beantwortet und Empfehlungen für zukünftige Projekte abgeleitet. Ziel ist es, die gewonnenen Erkenntnisse einzuordnen und einen Ausblick auf mögliche Weiterentwicklungen der Testautomatisierung im Projektkontext zu geben.

## 7.1 Zusammenfassung der Ergebnisse

Die Webanwendung IntelMod wurde im Rahmen dieser Arbeit durch automatisierte End-to-End-Tests mit Cypress geprüft. Es wurde bewiesen, dass die zentralen Geschäftsprozesse aus der Sicht der Vermieter und der Mieter zuverlässig und reproduzierbar getestet werden konnten. Die Anwendung konnte in allen abgedeckten Bereichen durch die erfolgreich umgesetzten Testfälle als funktionsfähig bestätigt werden. Cypress hat sich als ein Werkzeug bewährt, das Tests schnell umsetzt und die Nachvollziehbarkeit der Ergebnisse durch Funktionen wie interaktive Ausführung, automatisches Warten auf Elemente und Screenshots unterstützt.

Zudem wurde ersichtlich, dass die Fokussierung auf End-to-End-Tests zwar wertvolle Erkenntnisse über tatsächliche Nutzerflüsse liefert, jedoch nicht den Anspruch einer umfassenden Teststrategie erfüllt. Eine Erweiterung um zusätzliche Ebenen der Testpyramide, insbesondere Unit- und Integrationstests, wäre notwendig, um eine noch umfassendere und robustere Absicherung zu gewährleisten.

## 7.2 Beantwortung der Forschungsfrage

Die zentrale Forschungsfrage lautete:

„Welche Relevanz hat automatisiertes Software-Testing in der heutigen Softwareentwicklung?“

In der Gesamtschau erweist sich das automatisierte Testen von Software als ein zentrales Werkzeug in der modernen Qualitätssicherung. Es minimiert manuelle Aufwände, beschleunigt Entwicklungszyklen und ermöglicht eine reproduzierbare Überprüfung komplexer Systeme. Die Automatisierung ermöglicht die frühzeitige Identifizierung von Fehlern, die zuverlässige Verhinderung von Regressionen und die langfristige Sicherstellung der Stabilität von Anwendungen. Automatisiertes Testen leistet einen wesentlichen Beitrag zur Steigerung von Effizienz, Transparenz und Zuverlässigkeit in Softwareprojekten.

Im Rahmen des Projekts IntelMod konnte am Beispiel von Cypress konkret demonstriert werden, wie diese allgemeinen Vorteile in der Praxis realisiert werden können. Die konzipierten End-to-End-Tests haben die zentralen Geschäftsprozesse sowohl aus der Sichtweise des Vermieters als auch des Mieters erfolgreich validiert. In diesem Kontext zeigte sich Cypress als ein praxisorientiertes und effizientes Framework, das die Durchführung von Tests beschleunigt, die Ergebnisse transparent dokumentiert und die Abdeckung geschäftskritischer Prozesse zuverlässig unterstützt.

Damit lässt sich die Forschungsfrage beantworten: Automatisiertes Software-Testing ist von hoher Relevanz für die heutige Softwareentwicklung. In Forschung, Standards und etablierten Qualitätsmodellen wird es als Best Practice anerkannt, da es Entwicklungszeit verkürzt, Kosten reduziert und manuellen Testaufwand erheblich verringert. Im konkreten Projektkontext konnte zudem gezeigt werden, dass Cypress diese Vorteile durch die zuverlässige Umsetzung von End-to-End-Tests praxisnah bestätigt.

## 7.3 Empfehlungen für zukünftige Projekte

Aus den Erkenntnissen dieser Arbeit können mehrere Empfehlungen abgeleitet werden:

- Erweiterung der Testebenen: Zusätzlich zu End-to-End-Tests sollten auch Unit- und Integrationstests implementiert werden, um die Basis und Mitte der Testpyramide abzudecken und die Stabilität der Testsuite langfristig zu erhöhen.

- Berücksichtigung nicht-funktionaler Anforderungen: Aspekte wie Performance, Sicherheit oder Benutzerfreundlichkeit sollten zukünftig verstärkt in Betracht gezogen werden, da sie in realen Anwendungskontexten eine bedeutende Rolle einnehmen.

- Systematische Verwaltung von Testdaten: Die Verwendung standardisierter Testdaten und Fixtures kann die Reproduzierbarkeit verbessern und den Wartungsaufwand für die Tests reduzieren.

- Optionale Integration in CI/CD-Umgebungen: Obwohl dies im Rahmen des Projekts nicht implementiert wurde, kann die Integration in eine Pipeline den Grad der Automatisierung und die Effizienz in umfangreicheren Projekten erheblich erhöhen.

Diese Empfehlungen verdeutlichen, dass die in dieser Arbeit implementierten End-to-End-Tests mit Cypress eine robuste Basis gelegt haben und auf die zukünftigen Projekte aufbauen können.

# 8 Literaturverzeichnis


[1] International Organization for Standardization (ISO), ISO/IEC 25010:2023 Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - Product quality model, 2. Edition, 2023. [Online]. Available: https://www.iso.org/standard/78176.html

[2] OpenText, Capgemini & Sogeti, World Quality Report 2024-25, 2024. [Online]. Available: https://www.opentext.com/resources/world-quality-report-2024-25

[3] A. Spillner und T. Linz, Basiswissen Softwaretest, 6. Auflage, dpunkt.verlag, 2021, ISBN 978-3-86490-583-4.

[4] A. Basu, Software Quality Assurance, Testing and Metrics, ISBN 978-81-203-5068-7, 2015.

[5] ISTQB, Certified Tester Foundation Level - Syllabus, Version 2018 v3.1, 2018.

[6] IEEE Std 610.12-1990, IEEE Standard Glossary of Software Engineering Terminology, IEEE, 1990.

[7] IEEE Std 829-2008, IEEE Standard for Software and System Test Documentation.

[8] IEEE 29119-3:2013, Software and systems engineering - Software testing - Part 3: Test documentation.

[9] M. Klonk, R. Seidl, H. Pichler, M. Baumgartner und S. Tanczos, Agile Testing - Der agile Weg zur Qualität, ISBN 978-3-446-43264-2, 2013.

[10] Fowler, M., Continuous Integration, IEEE Software, Vol. 30, No. 1, 2013.

[11] Cypress.io, Cypress Documentation - Introduction to Cypress, https://docs.cypress.io/guides/overview/why-cypress

[12] M. Sharma und R. Angmo, Web based Automation Testing and Tools, International Journal of Computer Science and Information Technologies, 2014.

[13] M. Fowler, TestPyramid, [Online]. Available: https://martinfowler.com/bliki/TestPyramid.html

[14] IEEE 29119-1:2013, Software and systems engineering - Software testing - Part 1: Concepts and definitions.

[15] IEEE Software, Continuous Testing in Agile and DevOps, IEEE Computer Society, 2018.

[16] ACM Digital Library, Z. Wang et al., „Test Automation in Web Applications: A Comparative Study of Cypress and Selenium“Proceedings of the 2021 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), 2021.

[17] W3C, WebDriver - W3C Recommendation, 2018. [Online]. Available: https://www.w3.org/TR/webdriver/

[18] Selenium Project, Selenium Documentation - Getting Started, 2022. [Online]. Available: https://www.selenium.dev/documentation/webdriver/getting_started

[19] Boni García, „Exploring Browser Automation: A Comparative Study of Selenium, Cypress, Puppeteer, and Playwright,“ QUATIC 2024 - International Conference on Quality of Information and Communications Technology, 2024. https://bonigarcia.dev/slides/2024_QUATIC_Exploring_Browser_Automation_A_Comparative_Study_of_Selenium_Cypress_Puppeteer_and_Playwright.pdf

[20] Pelivani, M. et al., „A Comparative Study of UI Testing Frameworks,“ MECO 2022 - 11th Mediterranean Conference on Embedded Computing, IEEE, 2022.

[21] M. Cohn, Succeeding with Agile: Software Development Using Scrum, Addison-Wesley, 2009.

[22] ISO/IEC 25010:2023, Systems and software engineering - System and software quality models, International Organization for Standardization, 2023.

[23] Martin Fowler, The Practical Test Pyramid → https://martinfowler.com/articles/practical-test-pyramid.html

[24] Martin Fowler, On the Diverse and Fantastical Shapes of Testing (Test-Shapes 2021) → https://martinfowler.com/articles/2021-test-shapes.html

[25] Google web.dev, Types of Automated Testing → https://web.dev/articles/ta-types

[26] Google web.dev, What to test → https://web.dev/articles/ta-what-to-test

[27] Google web.dev, Testing strategies (get started) → https://web.dev/learn/testing/get-started/test-types

[28] Google web.dev, Testing - Learn / Welcome → https://web.dev/learn/testing/get-started

[29] usecure, „ISO 27001 Security Awareness Training,“ usecure Blog, 2023. [Online]. Available: https://blog.usecure.io/de/iso-27001-security-awareness-training.

[30] „Teststrategien,“ web.dev, 2023. [Online]. Available: https://web.dev/articles/ta-strategies?hl=de.

[31] Bird Eats Bug, „Glossary,“ Bird Eats Bug, 2025. [Online]. Available: https://birdeatsbug.com/glossary

[32] testRigor, „What is the Honeycomb Testing Model?,“ testRigor Blog, 2023. [Online]. Available: https://testrigor.com/blog/what-is-the-honeycomb-testing-model/.

[33] PROCON IT, „Modernes Software Testing,“ PROCON IT Blog, 2022. [Online]. Available: https://www.procon-it.de/blog/modernes-software-testing/.

[34] LambdaTest, „Cypress Tutorial,“ LambdaTest Learning Hub, 2023. [Online]. Available: https://www.lambdatest.com/learning-hub/cypress-tutorial.

[35] LambdaTest, „Selenium WebDriver Architecture,“ LambdaTest Blog, 2023. [Online]. Available: https://www.lambdatest.com/blog/selenium-webdriver-architecture/.

# 9 Anhang

### A.1 Cypress-Projekt IntelMod_Test

Das vollständige Cypress-Projekt (IntelMod_Test) ist im Anhang enthalten. Es umfasst die entwickelten End-to-End-Tests für die Vermieter- und Mietersicht der Anwendung IntelMod sowie die Projektstruktur und die dazugehörigen Konfigurationsdateien.

### A.2 Testplan (Excel-Dateien)

Zur Dokumentation der Testfälle wurden zwei Excel-Dateien erstellt, die die Testpläne für die beiden Anwendungssichten enthalten:

- INTELMOD-M.xlsx: Testplan für die Mietersicht
- INTELMOD-VM.xlsx: Testplan für die Vermietersicht

### A.3 Anleitung zur Projektausführung

Die Datei Anleitung_zur_Projektausführung.pdf enthält eine kurze technische Anleitung zur Ausführung des Projekts und der Testfälle. Sie beschreibt die notwendigen Schritte, um die Webanwendung lokal zu starten und die Cypress-Tests erfolgreich auszuführen.

Die Anhänge sind in elektronische Form auf dem beigefügten Datenträger (USB-Stick) zu finden.

# 10 Eigenständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit mit dem Titel:

Automatisierte End-to-End-Tests mit Cypress: Anwendung in einem Praxisprojekt

selbständig und nur mit den angegebenen Hilfsmitteln verfasst habe. Alle Passagen, die ich wörtlich aus der Literatur oder aus anderen Quellen wie z. B. Internetseiten übernommen habe, habe ich deutlich als Zitat mit Angabe der Quelle kenntlich gemacht.

26.09.2025

Datum

Unterschrift