Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Mohammad Saber Solimany

Dokumentation und Optimierung der Softwarearchitektur des Testautomatisierungstools CuTE

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Mohammad Saber Solimany

Dokumentation und Optimierung der Softwarearchitektur des Testautomatisierungstools CuTE

Bachelorarbeit eingereicht im Rahmen der BachelorprÃ¼fung

im Studiengang Bachelor of Science Wirtschaftsinformatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Ulrike Steffens
Zweitgutachter: Slobodanka Sersik

Eingereicht am: 16. Juli 2018

Mohammad Saber Solimany

Thema der Arbeit
Dokumentation und Optimierung der Softwarearchitektur des Testautomatisierungstools CuTE

Stichworte
Softwarearchitektur, Architekturdokumentation, Architekturanalyse, Quellcodeanalyse, Architektursichten, Architekturerweiterung,

Kurzzusammenfassung
Die vorliegende Bachelorarbeit dokumentiert die Systemarchitektur einer Software für automatisiertes Testen von Webanwendungen. Dafür werden alle relevanten Informationen einer Softwarearchitektur erfasst und gemäß eines praxisorientierten Dokumentation Templates dokumentiert. Anschließend folgt eine Analyse der tatsächlich implementierten Architektur des Prototyps der Software, welche die architektonische Erosion des Prototyps aufzeigt und darüber Aufschluss gibt, ob der Prototyp aufgearbeitet oder komplett neu implementiert wird. Abschließend wird die dokumentierte Architektur im Sinne von Erweiterung um weitere Systembausteine optimiert.

Mohammad Saber Solimany

Title of the paper
Documentation and optimization of the softwarearchitecture of a test automatisation tool named CuTE

Keywords
Software architecure,architecture documentation,architecture analysis, sourcecode analysis, architectural views, achitectural optimization

Abstract
This bachelor thesis documents the system architecture of software for automated testing of web applications. For this, all relevant information of a software architecture is recorded and documented according to a practice-oriented documentation template. This is followed by a sourcecode analysis of the prototyp and finally an optimization of the architecture of it.

Inhaltsverzeichnis

Listings

1 Einleitung

„Software architecture is the set of design decisions which, if made incorrectly, may cause your project to be cancelled.“
- Eoin Woods

Das Zitat von Eoin Woods beschreibt kurz und prägnant die Bedeutsamkeit von Softwarearchitektur für Software und das Scheitern von Projekten aufgrund schlechter oder gar nicht vorhandener Architektur.
Jedes Softwaresystem baut auf einer Architektur auf, welche in den meisten Fällen dokumentiert ist, in wenigen Fällen jedoch nicht, aber sie ist fester Bestandteil einer Software [Mit15]. Ein Softwaresystem um jeden Preis zum Laufen zu bekommen, ist keine Kunst und sollte auch nicht das Ziel bei der Entwicklung von Systemen sein. Vielmehr liegt die Kunst darin, ein Softwaresystem korrekt auf Basis eines dafür vorgesehenen Architekturplans zu implementieren. Wenn das passiert, müssen keine unzähligen Stunden Programmierarbeit in die Lebenserhaltung des Systems gesteckt werden. Das System wird überschaubarer und Änderungen oder Erweiterungen können schneller und effizienter umgesetzt werden [Mar17].
Die Architektur bietet das Fundament eines Softwaresystems, kann jedoch bei schlechter Dokumentation kontraproduktiv wirken. In vielen Entwicklungsteams wird das Dokumentieren von Softwarearchitekturen sehr halbherzig behandelt und dementsprechend ist das Resultat. Aus diesem Grund ist eine lesbare und verständliche Dokumentation ebenfalls Teil einer guten Architektur [Sta15]. Früher besaßen System nicht die Komplexität und den Umpfang wie heute und so konnten einzelne Entwickler gesamte Systeme überblicken. Systeme der heutigen Zeit sind umfangreicher und komplexer. Um die Struktur eines Systems zu beherrschen müssen Auswirkungen von Technologieentscheidungen vorausgesehen werden und diese mit einer Vielzahl von in Frage kommende Hilfsmittel, wie Frameworks, Libraries oder Entwicklungswerkzeuge, zielführend umzusetzen [Sta15].

1.1 Problemstellung und Relevanz des Themas

Wie bereits aus dem Titel dieser wissenschaftlichen Arbeit hervorgeht, wird ein architektonisches Modell für eine Software dokumentiert, analysiert und schließlich optimiert. Die Software, auf die das Architekturmodell abzielt, ist ein im Unternehmen intern entwickeltes Testtool, welches das automatisierte Testen von Webanwendungen unterstützt. Bei dem intern entwickelten Testtool handelt es sich jedoch noch um einen Prototypen, für den aktuell kein Architekturplan besteht. Das Fehlen eines Architekturplans ist damit begründet, dass dieses Tool als Nebenprodukt eines großen Software-Entwicklungsprojekts zustande gekommen ist. Das System wurde von einem einzigen Entwickler geschrieben und hatte den Zweck, den manuellen Testaufwand des damaligen Softwareprojektes zu verringern. Da die Anforderungen an das Programm anfangs sehr gering waren, wurde es ohne architektonischer Modellierung geschrieben. Mit Fortschritt des Projektentwicklungsprozesses wurde auch das Testtool um weitere Funktionalitäten erweitert und ausgebaut und festigte seine Stellung als taugliches und leicht bedienbares Testprogramm. Das Erweitern des Tools fand während einer kurzen Zeit statt und gemeinsam mit dem Faktor des fehlenden Architekturplans führte das zu einem unstrukturierten und nicht nachhaltigen Aufbau der Anwendung. Das hatte zur Folge, dass beim Erweitern oder bei Fehlerbehebungen der Anwendung, oft Folgefehler auftraten oder sogar ganze Grundfunktionalitäten ausfielen. Um diesen negativen Faktoren entgegenzuwirken, soll das Tool anhand eines Architekturplans neu dokumentiert werden. Dadurch soll die einfache Erweiterbarkeit und Flexibilität des Programms angestrebt werden, zumal sich die Nutzung des Tools nicht nur auf ein bestimmtes IT Projekt beschränkt, sondern es sehr universell bei Webanwendungen einsetzbar ist.

1.2 Zielsetzung der Arbeit

Ziel dieser Arbeit ist es, einen reinen und lückenlosen Architekturplan für das beschriebene Tool zu dokumentieren. Dieser Plan soll eine abstrakte Abbildung des Systems darstellen und als Grundlage für dessen Implementierung dienen. Hierfür werden alle für eine Softwarearchitektur relevanten Informationen ermittelt und in einer Dokumentation festgehalten. Diese Dokumentation richtet sich nach einem praktisch gegliederten Template und soll den Lesern eine strukturierte und gut verständliche Erläuterung der Systemarchitektur des Programms bieten. Des Weiteren bietet diese Ausarbeitung Aufschluss darüber, ob ein Refactoring des aktuell vorhandenen Prototyps basierend auf der dokumentierten Architektur lohnend ist oder ob das System neu implementiert werden soll. Bereits aus dem Titel der Bachelorarbeit geht hervor, dass eine Optimierung der Systemarchitektur stattfinden soll. Bei der Optimierung

handelt es sich um eine Erweiterung einer bestimmten Systemkomponente. Diese muss um eine im Prototyp bestehende aber unsauber implementierte Funktionalität erweitert werden. Diese gilt es mit den passenden Bausteinen zu integrieren und in den richtigen Komponenten des Systems unterzubringen.

1.3 Aufbau der Thesis

Die Bachelorarbeit orientiert sich an dem zu untersuchenden Thema und ist in vier Kapitel unterteilt. Nachdem im Kapitel 1 die Problemstellung und die Zielsetzung der vorliegenden Arbeit erläutert wurden, dient Kapitel 2 zum Verständnis der theoretischen Grundlagen der Softwarearchitektur. In diesem Kapitel folgt eine detaillierte Erläuterung der Grundlagen sowie eine Eingrenzung des Themengebiets, da Softwarearchitektur tatsächlich ein weitgreifender Begriff ist. Es werden verschiedene Architekturstile vorgestellt, welche für den praktischen Teil relevant sind. Es folgt ein Abschnitt zur Dokumentation von Systemarchitekturen, sowie empirische Methoden zur Architekturanalyse. Im Kapitel 3 folgt der Hauptteil mit der Dokumentation der Systemarchitektur des Testtools CuTE. Diese richtet sich nach den im Abschnitt 2.3 erläuterten Aktivitäten, um alle für die Architektur relevanten Informationen zu erfassen und festzuhalten. Im Abschnitt 3.2 folgt eine Sourcecode Analyse des Prototyps, in der der Prototyp auf ausgewählte architektonische Merkmale analysiert wird. Schließlich folgt im Abschnitt 3.3 eine Erweiterung der Systemarchitektur um bestimmte Funktionalitäten. Diese soll zur Optimierung der Systemarchitektur beitragen. Im letzten Kapitel 4 folgt eine Zusammenfassung der wesentlichen Punkte und der gewonnenen Erkenntnisse. Neben der Zusammenfassung erfolgt ein Ausblick für zukünftige Fragestellungen und potentielle Folgeprojekte.

2 Theoretische Grundlagen zur Softwarearchitektur und Softwarearchitekturdokumentation

In diesem Kapitel werden theoretische sowie fachliche Grundlagen, welche für das Verständnis des Themas und für die praktische Ausarbeitung relevant sind, behandelt. Zunächst wird erläutert, was Softwarearchitektur ist und aus welchen Bestandteilen sie sich zusammensetzt. Hierfür werden ebenfalls zwei in der praktischen Ausarbeitung verwendete Architekturstile vorgestellt. Darauf aufbauend folgt eine Erläuterung der Bedeutung von Softwarearchitektur auf Softwaresystemen. Anschließend werden die Folgen behandelt, welche bei der Umsetzung von Systemen ohne Architekturvorgaben entstehen, sowie Maßnahmen um so ein Unterfangen zu unterbinden. Abschließend folgt ein Abschnitt über das korrekte Dokumentieren von Systemarchitekturen.

2.1 Architekturbegriff für Softwaresysteme

Der Begriff „Softwarearchitektur“ lässt sich heutzutage nicht mehr durch eine allgemeingültige Definition beschreiben. Stattdessen existieren viele unterschiedliche Definitionen, welche vom Software Engineering Institute der Carnegie Mellon University zusammengetragen und Online veröffentlicht wurden [Zö16]. Folgende Definition ist besonders zutreffend:

> *„Die grundsätzliche Organisation eines Systems, verkörpert durch dessen Komponenten, deren Beziehungen zueinander und zur Umgebung sowie die Prinzipien, die für seinen Entwurf und seine Evolution gelten.“*
> *- Gernot Starke*

Aus dieser Definition geht hervor, dass Softwarearchitektur die abstrakte und allumfassende Abbildungen eines Softwaresystems beschreibt, welche aus mehreren Systembausteinen besteht, die Beziehungen zueinander haben [Sta15]. Die Beschreibung von Strukturen und das Zusammenwirken dieser Bausteine im Sinne von Bauanleitungen, gewährleisten bei korrekter

Implementierung ein reales und funktionsfähiges System[Sta15]. Dazu gehören ebenfalls das Dokumentieren von Entwurfsentscheidungen, die als Grundlage für den Entwurf der Systemkomponenten dienen[Sta15]. Die Softwarearchitektur zeigt verschiedene Sichten auf das System. Jede Sicht beinhaltet eigenspezifische Informationen über das System, die für die jeweiligen Stakeholder relevant sind und ermöglicht es ihnen, so, die Lösungen zu verstehen [Sta15]. Heutzutage ist Flexibilität bei Softwarearchitektur großgeschrieben, da sich im Laufe der Systemkonstruktion Anforderungen ändern und oft neue hinzukommen. Die Architektur muss somit interne und externe Einflussfaktoren, technischer und organisatorischer Sicht berücksichtigen, da diese Auslöser für neue Anforderungen sind [RK12]. Flexibilität zählt wie Verständlichkeit, Performance, Robustheit oder Sicherheit zu den Qualitätseigenschaften eines Systems. „Einen bestimmten Algorithmus ohne jegliche Qualitätseigenschaften zu implementieren, ist erheblich leichter, als dieses Programm gleichzeitig auch noch verständlich, erweiterbar und performant für viele parallele Benutzer zu entwickeln"[Sta15]. So bietet die Architektur eines Systems die Basis für dessen Qualität. Nutzen und Ziele einer Softwarearchitektur werden von dem Autor Starke [Sta15] folgendermaßen zusammengefasst: „Die Architektur stellt die Realisierung von Anforderungen und Zielen bezüglich der Qualität sicher. Vergleichbare Systemprozesse werden durch Integrität (Konsistenz) ähnlich gelöst, Entwurfsentscheidungen, Strukturen und Konzepte werden durch verschiedene Modelle und Dokumentationen klar dargestellt. Dementsprechend wird auch eine erleichterte Wiederverwendung von Systemkomponenten sichergestellt. Die Softwarearchitektur konzentriert sich auf Langfristigkeit und wirkt auf den gesamten Lebenszyklus mit“.

2.2 Grundlegende Aktivitäten beim Systementwurf

Bei der Architekturentwicklung eines Systems gibt es kein deterministisches Verfahren, das angewendet werden kann und eine gute Softwarearchitektur garantiert. Vielmehr unterteilt sich die Architekturentwicklung in grundlegende Aktivitäten, welche bei der Erstellung durchlaufen werden [Mit15]. Bereits vor der Entwurfsentwicklung der Systembausteine erweist es sich als vorteilhaft, Informationen zu Lösungskonzepten ähnlicher Problemstellungen zu sammeln. Diese Lösungskonzepte könnten als Entwurfsvorlage verwendet werden oder als Systemkomponente gänzlich erworben werden [Mit15]. Um die Architekturentwicklung beginnen zu können, sollten die Kernaufgaben des Systems bekannt sein, welche sich aus den funktionalen Anforderungen entnehmen lassen. Es ist ebenfalls von enormer Wichtigkeit, die geforderten Qualitätsanforderungen zu ermitteln [Mit15]. Qualitätsmerkmale lassen sich wie folgt definieren: „Die Qualität von Softwaresystemen wird immer bezogen auf einzelne

Eigenschaften oder Merkmale. Beispiel für solche Merkmale sind Effizienz, Verfügbarkeit, Änderbarkeit und Verständlichkeit“ [Sta15]. Einzelne Qualitätsanforderungen stehen meist im Wettbewerb miteinander und Zeit- sowie Kostenfaktoren und es gilt eine Lösung in Abhängigkeit der Stärken der jeweiligen Qualitätsanforderungen zu finden. Das wird auf Abbildung 2.1 deutlich:

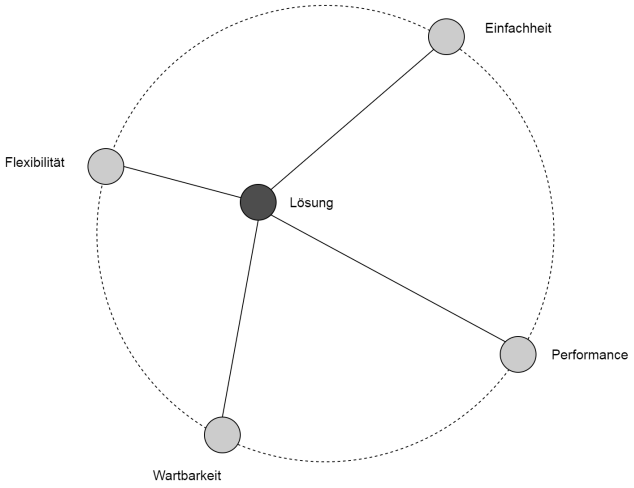

Abbildung 2.1: Gummiband-Diagramm: Lösung als Kompromiss ([Sta15])

Diese erweisen sich ebenfalls als Akzeptanzkriterien der jeweiligen Qualitätsanforderungen und sollten als Bestandteil der Softwarearchitektur dokumentiert werden [Sta15]. Das Ermitteln der Einflussfaktoren und Randbedingungen zählt ebenfalls als Aktivität, die beim Erstellen von Softwarearchitektur durchgeführt werden sollte. Einflussfaktoren beeinträchtigen Entwurfsentscheidungen der Architektur und lassen sich in Faktoren organisatorischer und technischer Art einteilen [Sta15]. Organisatorische Faktoren beeinflussen das System indirekt und beziehen sich im weitesten Sinne auf die Umgebung, in der das System erstellt wird. Hierzu zählen „Aspekte wie Termin- und Budgetplanung, vorhandene Mitarbeiter, technische Ressourcen, Entwicklungsprozesse, Vorgaben bezüglich Werkzeuge und ähnliches“ [Sta15]. Technische Einflussfaktoren hingegen betreffen das System direkt und legen Vorgaben für die Systementwicklung fest. Dazu gehört zum Einen die Ablaufumgebung des Systems und zum Anderen einzusetzende Fremdsysteme, sowie bereits vorhandene Systembauteile [Bal11]. Schließlich muss eine Kontextabgrenzung des Systems erfolgen, um die Tätigkeiten des Systems herauszuarbeiten. Im Systemkontext werden Anwendergruppen, die das System benutzen und Fremdsysteme die zur Ausführung des Systems nötig sind, skizziert. Der Systemkontext klärt

die Verantwortlichkeiten und Grenzen des Systems [Bal11]. Die Systembausteine und deren Beziehungen, die Verteilung der Bausteine auf physikalische Rechner, sowie das Verhalten der Bausteine zur Laufzeit des Systems, stellen die verschiedenen Sichten auf die Architektur dar und beleuchten das System. Diese werden anhand grafischer Dokumentationsmittel dargestellt [Zö16]. Es folgt eine Beschreibung der technischen Konzepte, der querschnittlichen Themen zur Gestaltung der Systembausteine erläutert werden. Zu diesen Themen zählen die Wahl des verwendeten Architekturstils, Maßnahmen zur Umsetzung der Qualitätsmerkmale oder die Auswahl der Methoden, um mit Benutzern oder Fremdsystemen zu interagieren [GS09]. Das Dokumentieren der Risiken welche aus den Entwurfsentscheidungen und Konzepten hervorgehen, ist ebenfalls Teil der Softwarearchitektur. Zum Abschluss dieses Unterkapitels sollte festgehalten werden, dass die vier Sichten auf die Architektur eines Systems, den zentralen Kern einer Softwarearchitektur darstellen. Nachdem alle in diesem Abschnitt erläuterten Aktivitäten durchgeführt und dokumentiert wurden, kann der Entwurf der Systembausteine beginnen. Voraussetzung hierfür ist ein solides Verständnis von Architekturstilen und Entwurfsmustern, welche im 2.5 Abschnitt erläutert werden.

2.3 Dokumentation von Architekturen

„In der Praxis scheitern Projekte immer wieder, weil Softwarearchitekten ihrer Kommunikationsaufgabe nicht angemessen nachkommen.“
- Gernot Starke

Das Dokumentieren einer Softwarearchitektur zählt zu den Kommunikationsaufgaben von Architekten. Anfangs findet die Kommunikation der Architekten und Analysten mit den Stakeholdern mündlich durch gemeinsames Erarbeiten, Erklären, Vorstellen und Überzeugen von Konzepten und Entwürfen. Diese erarbeiteten Ergebnisse gilt es schriftlich zu dokumentieren, damit die Langlebigkeit des Systems gewährleistet werden kann. Der Quellcode eines Systems kann nicht allein als Dokumentationsmittel dienen, da dieser sehr umfangreich sein kann und ein niedriges Abstraktionsniveau enthält. Architekturdokumentationen sollen den Lesern und den Stakeholdern einen Überblick über das System gewähren. So können auftretende Fehler effektiver beseitigt werden, neue oder geänderte Anforderungen mit angemessenem Aufwand erfüllt werden oder auf Änderungen im gesamten technischen Umfeld reagiert werden [Zö16]. Aus diesem Grund muss eine Architekturdokumentation gut lesbar und strukturiert sein, um den verschiedenen Zielgruppen das Verständnis der Dokumentation zu erleichtern. Zu der Struktur einer Architekturdokumentation schreibt der Autor Zöner [Zö16]: „Ein Leser findet sich in einer Architekturbeschreibung schneller zurecht, wenn ihm die Strukturierung

vertraut ist. Er kann beispielsweise gezielt Dinge nachschauen. Dieser Vorteil ist insbesondere in Unternehmen und Organisationen relevant, wo viele Softwareprojekte durchgeführt werden und Teammitglieder häufig hinzustoßen oder wechseln. Da sie im Team schneller arbeitsfähig werden, spart man Zeit und Geld“. Die Verwendung einer einheitlichen Dokumentationsstruktur erspart dem Autor die Zeit zum Gestalten einer individuellen Struktur und erleichtert unerfahrenen Architekten den Einstieg in die Dokumentation [Zö16]. Eine beliebte und verbreitete Strukturvorlage für Architekturdokumentation ist arc42. Arc42 ist eine freie und praxisorientierte Strukturvorlage, zur Beschreibung und Dokumentation von Softwarearchitektur. „Motivation für die Entwicklung von arc42 war die aus Sicht der Autoren mangelnde Praxistauglichkeit und Schwergewichtigkeit der verfügbaren Lösungen. arc42 sollte helfen, dass nicht jedes Projekt bei null starten muss oder bei null Architekturdokumentation bleibt, weil der Aufwand etablierter Ansätze angeblich so hoch ist“ [Zö16]. Eine alternative zu der arc42 Strukturierung ist IEEE 1417. Bei IEEE 1417 handelt es sich um ein Standard zur Dokumentation von Softwarearchitekturen, welche sich aber in der Praxis als unstrukturiert erwiesen hat [Zö16]. Das arc42 Template unterteilt sich in 12 Kapitel. Der erste Teil umfasst drei Kapitel erläutert die Aufgabenstellung. Dazu werden Qualitätsziele, Sinn und Zweck des Systems, Randbedingungen und der Systemkontext beschrieben und dokumentiert. Das vierte Kapitel beschreibt kurz die Zusammenfassung der Lösungsstrategien die beim Entwurf des Systems verwendet werden. Das löst den Lesefluss der Dokumentation und bietet dem Leser einen fließenden Übergang in den Sichten der Architektur [Zö16]. Der zweite Teil von arc42 stellt den Hauptteil der Dokumentation dar und widmet sich den Lösungen. Dazu gehören Architekturentscheidungen, die Zerlegung des Systems in Bausteine, das Verhalten bestimmter Systembausteine zur Laufzeit, die für den Betrieb des Systems benötigte technische Infrastruktur, sowie technische Konzepte und Entwurfsentscheidungen. Es folgt der dritte Teil, in dem die Lösungen bewertet werden, durch Qualitätsszenarien und die Beschreibung der von den Lösungskonzepten hervorgerufenen Risiken. Zum Schluss folgt ein Glossar. Die Reihenfolge der Gliederung schreibt kein deterministisches Vorgehen bei der Erstellung der Architektur vor, sondern dient der besseren Verständlichkeit. Abbildung 2.2 zeigt die Struktur von arc42 mit den einzelnen Kapiteln.

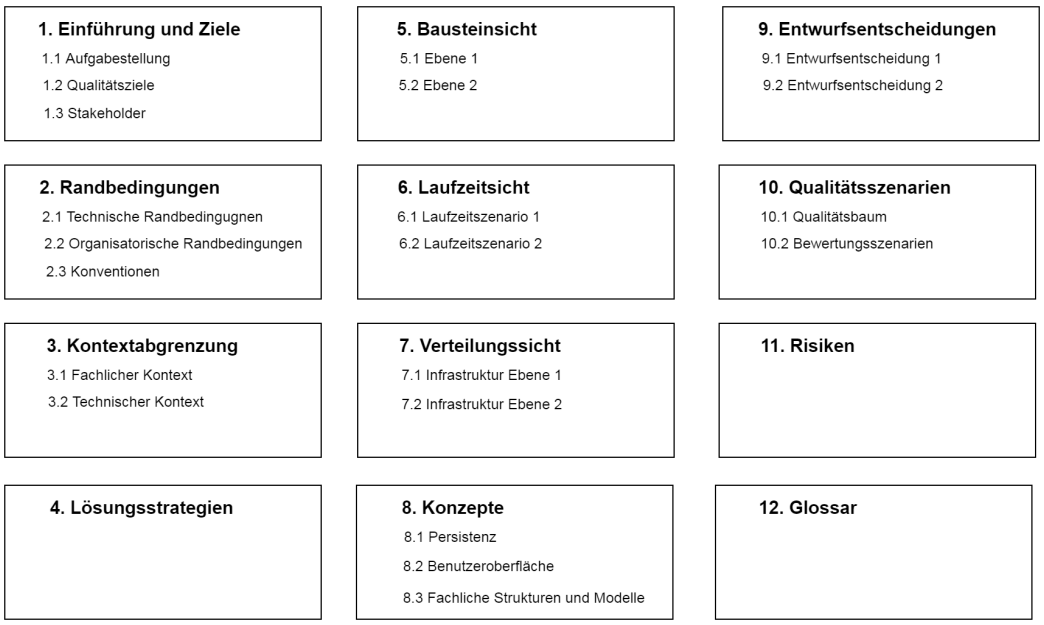

Abbildung 2.2: Übersicht der arc42 Kapitel ([Zö16])

Als nächstes folgt eine kurze Beschreibung der Inhalte und Aktivitäten der in Abbildung 2.2 aufgelisteten Kapitel. Kapitel drei, fünf, sechs und sieben werden ausführlicher behandelt, da das die vier Sichten auf die Systemarchitektur sind und das Herz der Dokumentation darstellen [Zö16].

2.4 Sichten auf die Softwarearchitektur

Softwaresysteme können sehr komplexe Gebilde darstellen und dementsprechend sind die Architekturen. Diese Architektur über eine einzige Darstellung abzubilden, vermag die Komplexität und Vielseitigkeit davon nicht ausdrücken. Aus diesem Grund sollten Architekturen aus verschiedenen Perspektiven beleuchtet werden, um so die Konzentration auf das System, zu ermöglichen. Sichten zeigen einzelne Stakeholder verschiedene Perspektiven der Architektur und ermöglichen ein leichtes Verständnis, zumal in den verschiedenen Sichten gezielt auf Darstellungskomplexität verzichtet wird. So werden Details abstrahiert, die für die jeweilige Sicht nicht von Bedeutung sind. „Die Diagramme oder textlichen Beschreibungen einer Sicht können auch unterschiedliche Abstraktionsebenen oder Darstellungsstufen beschreiben“[Sta15]. Abbildung 2.3 zeigt die vier wichtigsten Arten der Sichten. Die Pfeile symbolisieren mögliche

Abhängigkeiten oder Wechselwirkungen. Die einzelnen Sichten werden in diesem Abschnitt präziser beschrieben.

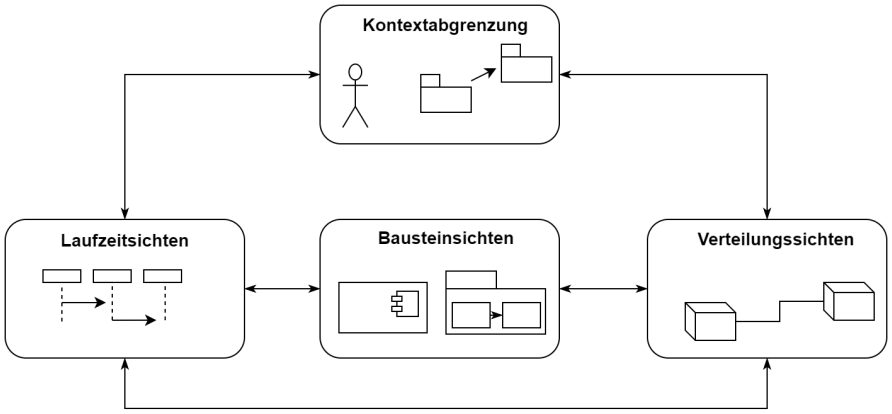

Abbildung 2.3: Vier Sichten auf die Softwarearchitektur([Sta15])

2.4.1 Kontextabgrenzung

Der Systemkontext veranschaulicht anhand grafischer Mittel das Umfeld des Systems, sowie dessen Zusammenhang mit seiner Umwelt [GS09]. So wird das System als Blackbox in den Mittelpunkt gestellt und drum herum werden alle Akteure und Fremdsysteme eingezeichnet mit denen das System interagiert. Daraus gehen besonders die Schnittstellen zur Außenwelt, zu Anwendern, zu Betreibern und zu Fremdsystemen hervor. So stellt der Systemkontext eine Abstraktion der anderen Sichten dar, damit der Fokus auf die Aktuere und das Umfeld des Systems gelegt wird [GS09]. Der Systemkontext ermöglicht bereits am Anfang der Architekturerstellung die Grenzen des Systems zu definieren, und dient als Grundlage für weiterführende Entwurfsentscheidungen. Der Systemkontext unterteilt sich in einem fachlichem und einen technischen Kontext. Während sich der fachliche Kontext auf die Akteure und Fremdsysteme konzentriert, werden im technischen Kontext die Systemschnittstellen sowie der Datenaustausch auf Basis von ausgewählten Protokollen zwischen dem System und seiner Umwelt dargestellt [GS09]. Abbildung 2.4 zeigt ein Beispiel eines Systemkontextdiagramms.

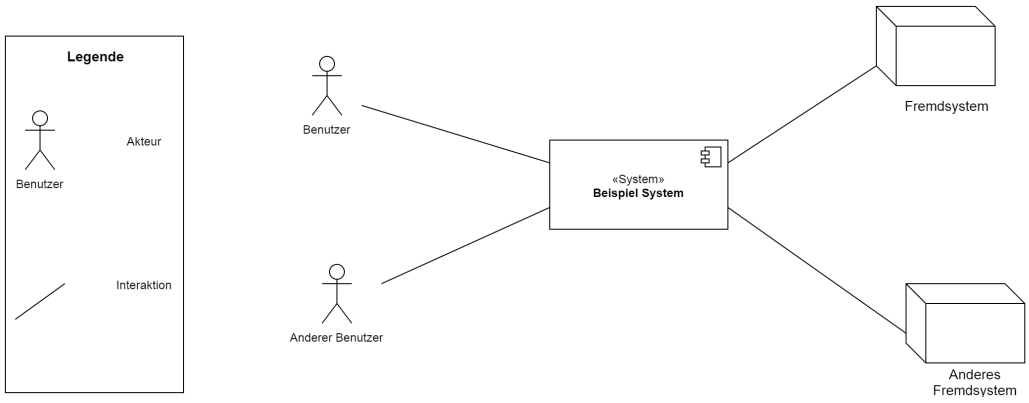

Abbildung 2.4: Beispiel eines Kontextdiagramms ([GS09])

2.4.2 Bausteinsicht

In der Bausteinsicht werden die Aufgaben des Systems auf den jeweiligen Systembausteinen abgebildet. Dabei wird besonders die Struktur der Bausteine sowie deren Beziehungen graphisch dargestellt. Die Bausteinsicht dient als Struktur und abstrakte Abbildung des Quellcodes. Sämtliche architektonische Stile und Entwurfsmuster zur Realisierung der gewünschten Funktionalitäten, sowie Qualitätsmerkmale werden in der Bausteinsicht entworfen. So beinhaltet die Bausteinsicht alle definierten Bausteine, deren Abhängigkeiten, gegenseitige Schnittstellen als auch Schnittstellen zu Fremdsystemen [Zö16]. Der Entwurf der Bausteine beginnt bei der Kontextabgrenzung. Es gilt das System in Architekturelemente wie Subsysteme, Komponenten oder atomare Bestandteile aufzuteilen. Das muss unter Berücksichtigung der Randbedingungen, Einflussfaktoren und Qualitätsanforderungen geschehen. Das System kann anhand verschiedener Strategien zerlegt werden. Das geschieht entweder auf Basis der fachlichen und funktionalen Anforderungen oder auf Basis der qualitativen Anforderungen an das System. Eine Zerlegung gemäß der technischen Infrastruktur, wie zum Beispiel Browser Clients und Server ist ebenfalls eine bewährte Herangehensweise [GS09]. Ein Baustein definiert sich als existierender oder geplanter Quellcode, auf den verschiedenen Abstraktionsebenen. Dazu zählen Klassen, Prozeduren, Programme, Pakete, Komponenten, oder Subsysteme wie Abbildung 2.5 visualisiert.

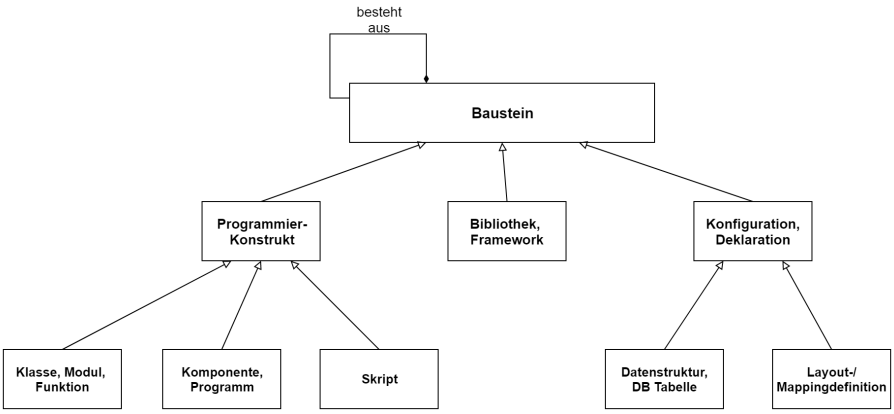

Abbildung 2.5: Metamodell von Bausteinen([Sta15])

Diagrammelemente der Bausteinsicht unterteilen sich in zwei Abstraktionsarten, nämlich Black- und Whiteboxes. Blackboxes sind Bausteine, die ihre innere Struktur und Komplexität verbergen. Sie beinhalten lediglich eine Beschreibung ihrer Funktionalität sowie ihre externen Schnittstellen. Whiteboxes hingegen sind offene Blackboxes, welche die Bausteine und die innere Struktur davon grafisch darstellen [Sta15]. Mit dem Top-Down Ansatz wird das System anfangs als Blackbox dargestellt und Schritt für Schritt als Whitebox verfeinert. So werden abwechseln Black- und Whitebox Darstellungen notiert, welche die Hierarchien und Architekturebenen abbilden. Abbildung 2.6 zeigt den Prozess der Verfeinerung der Systembausteine [GS09].

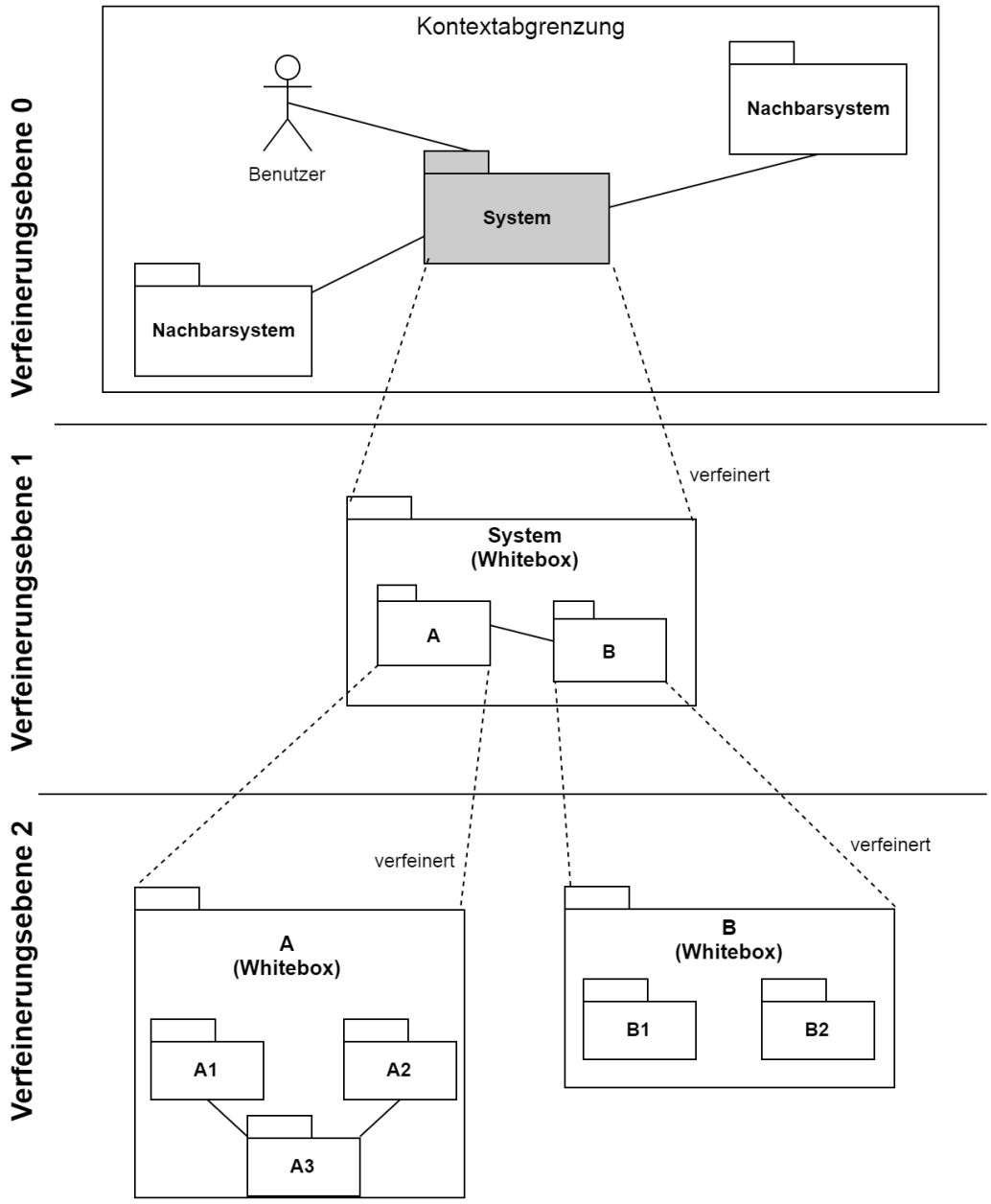

Abbildung 2.6: Verfeinerung von Bausteinen ([GS09])

Jede Whiteboxsicht auf jeder Ebene enthält Beschreibungen zu den eigenen Bausteinen, welche Aufgaben, Verantwortlichkeiten sowie Schnittstellen dokumentieren. Die Darstellung

der Diagramme erfolgt je nach Bausteinebene entweder als UML-Klassen- oder Komponentendiagramme.

2.4.3 Laufzeitsicht

Die Laufzeitsicht zeigt das Verhalten und Interagieren der Systembauseine zur Laufzeit des Systems. Aus der Laufzeitsicht geht hervor, wie und welche Systembausteine zur Laufzeit miteinander arbeiten, wie die wichtigsten Use-Cases durch die Architekturbausteinen abgewickelt werden oder welche Instanzen der Bausteine hierfür gestartet, überwacht und beendet werden [Sta15]. Laufzeitszenarien werden anhand von UML-Sequenz-, Aktivitäts- oder Kommunikationsdiagrammen dargestellt. Im Laufzeitszenario werden Instanzen von Implementierungsbausteinen und deren Beziehungen abgebildet. Beziehungen können Daten- und Kontrollflüsse sein [Zö16].

2.4.4 Verteilungssicht

Die Verteilungssicht beschäftigt sich zum einen mit der Verteilung der Systembausteine auf Hardwarekomponenten wie Prozessoren, Speicher, Netzwerke, Router und Firewalls. Zum anderen bildet sie das Deployment der Systembausteine auf die Hardware ab. Herr Starke [Sta15] formuliert dazu folgendes: „Sie können in der Infrastruktursicht die Leistungsdaten und Parameter der beteiligten Elemente darstellen, wie etwa Speichergröße, Kapazitäten oder Mengengerüste. Außerdem können Sie zusätzlich die Betriebssysteme oder andere externe Systeme aufnehmen.“ Zu den Elementen der Verteilungssicht gehören Knoten, welche echte oder virtuelle Rechner oder Prozessoren darstellen. Außerdem gehören Laufzeitartefakte und Kanäle zwischen den Knoten ebenfalls zur Verteilungssicht. Die Knoten repräsentieren die technische Infrastruktur, auf jeweilige Instanzen der Bausteine ausgeführt werden. Bei Systemen, die nicht auf mehrere Knoten verteilt sind, reicht meistens eine textuelle Beschreibung aus. Bei Systemen die auf mehreren Knoten verteilt sind und für die eine textuelle Beschreibung nicht ausreichen sollte, werden UML-Einsatzdiagramme verwendet [GS09].

2.5 Architekturstile

Ein Architekturstil zeichnet sich dadurch aus, dass er Regeln und Normen zur Strukturierung von Systembausteinen und deren Verbindungen enthält. IT Systeme eines bestimmten Architekturstils können dementsprechend kategorisiert werden, zumal sie die selben Systemstrukturen aufweisen [Bal11]. In dieser Ausarbeitung werden nicht alle Architekturstile behandelt, lediglich werden die beschrieben, die in der praktischen Ausarbeitung verwendet werden oder in

Betracht gezogen werden, für bestimmte Bausteine verwendet zu werden. Heterogene Systeme ist die Bezeichnung für Systeme, die mehrere Architekturstile gleichzeitig aufweisen, was oft bei IT Systemen zutrifft [TB03]. Jeder Architekturstil hat seine Vor- und Nachteile, wobei jeder Stil eine individuelle Lösung auf eine bestimmte Art von Problemstellungen bietet. Architekturstile beeinflussen Systeme auf grobgranularer Ebene [TB03]. Nachfolgend werden drei beliebte und für die praktische Ausarbeitung in Frage kommenden Stile beschrieben.

2.5.1 Schichtenarchitektur

Softwaresysteme mit einer Schichtenarchitektur bestehen aus hierarchischen Schichten, in die Systembausteine aufgeteilt sind. Die oberen Schichten der Hierarchie haben Zugriff auf die untergeordneten Schichten, jedoch nicht umgekehrt . So schreibt [Sta15]: „Eine Schicht bietet den darüberliegenden Schichten bestimmte Dienste an. Eine Schicht kapselt die Details ihrer Implementierung gegenüber ihrer Außenwelt. Sie kann dabei ein beliebig komplexes System darstellen“. Zu den Vorteilen einer Schichtenarchitektur zählen zum einen, dass die Schichten voneinander unabhängig sind, sowohl bei der Erstellung als auch beim Betrieb von Systemen. Außerdem ist eine Schichtenarchitektur ein leicht verständliches Strukturkonzept, zumal die Schichten nur bedingt von anderen Schichten abhängig sind [Vog09]. Ein bedeutender Nachteil einer Schichtenarchitektur ist, dass die Performance des Systems darunter leiden kann, da Anfragen unter Umständen durch mehrere Schichten weitergereicht werden müssen, bis diese schließlich bearbeitet werden können. Das kann mit „Layer-Brindging“ vermieden werden, indem Schichten übersprungen werden, wie Abbildung 2.7 zeigt. Damit entstehen jedoch weitere Abhängigkeiten zwischen den Schichten [Sta15].

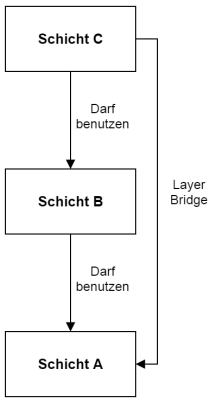

Abbildung 2.7: Layer Bridge ([Sta15])

Bei der Gestaltung einer Schichtenarchitektur können Standardschichten bei der Einteilung der Schichten behilflich sein. Zu den Standardschichten gehören die Präsentations-, die Fachdomänen- und die Infrastrukturschicht. Die Präsentationsschicht ist normalerweise die oberste Schicht und stellt die Benutzeroberfläche des Systems zur Verfügung. In der Fachdomänenschicht werden fachliche Aspekte des Systems implementiert und die Infrastrukturschicht kapselt die Komplexität der technischen Infrastruktur gegenüber den anderen Schichten [Vog09].

2.5.2 Client-Server Architektur

Eine Client-Server Architektur zeichnet sich dadurch aus, dass Verarbeitungs- und Speicherbausteine innerhalb eines Netzwerks verteilt sind. Diese bestehen aus mindestens einem Client und einem Server. Clients nehmen Dienste des ihnen bekannten Servers in Anspruch. Ein Server, der sich entweder auf dem selben physikalischen Rechner oder einem anderen Rechner befindet, stellt den Clients seine Dienste zur Verfügung. Voraussetzung für die Kommunikation ist, dass beide sich im selben Netzwerk befinden [Wikete]. Somit lässt sich die Client-Server Architektur in den Architekturstil verteilter Systeme einordnen, setzt aber auch das Prinzip der Schichtenarchitektur um. So befindet sich der Client in der übergeordneten Schicht und nutzt die Dienste des Servers, welcher sich in der niederen Schicht befindet [Hau10]. Die Kommunikation und der Datenaustausch zwischen den Clients und dem Server hängt von den jeweiligen Diensten ab. Dabei muss der Server jederzeit für die Clients erreichbar sein, da sonst keine Kommunikation stattfinden kann. Die Kommunikation zwischen beiden wird durch Protokolle realisiert und die Art des Protokolls wird auf Basis der gewählten Kommunikationsmethode festgelegt [Wikete]. Als Kommunikationsmethoden bieten sich Remote Procedure Calls, HTTP oder TCP/UDP Protokolle an. Bei Remote Procedure Calls werden Methodenaufrufe gegen entfernte Systemkomponenten ausgeführt. Dafür stellt der Server dem Aufrufer eine Schnittstelle zur Verfügung und beide müssen über Festlegung im Quellcode eng gekoppelt sein. Starke fasst Vor- und Nachteile einer Client-Server Architektur so zusammen: „Den Vorteilen der hohen Flexibilität und klaren Trennung der Verantwortlichkeiten dieses Architekturmodells stehen Nachteile hinsichtlich Sicherheit, höherem Entwicklungsaufwand und vielfältigen Fehlerquellen gegenüber"[Sta15].

2.5.3 Model-View-Controller Muster

Das Model-View-Controller Muster teilt ein System in drei Bestandteile auf, das Model, View und Controller, die über einen Notifizierungsmechanismus ihre gegenseitige Konsistenz sichern.

Das System, bei dem ein MVC (Model-View-Controller) Muster umgesetzt wird, muss ein interaktions-orientiertes System sein, basierend auf der Interaktion mit Benutzern [GS09]. „Das Model enthält die (fachlichen) Daten und Funktionen des Systems- und ist völlig unabhängig von der konkreten UI-Technologie. Benutzer interagieren über den Controller mit dem System. Controller geben Benutzereingaben zur weiteren Bearbeitung an Model oder View weiter. Die View sorgt für eine Anzeige der Model-(Daten) auf einer grafischen Oberfläche“ [Sta15]. Abbildung 2.8 veranschaulicht die Struktur einer MVC-Muster.

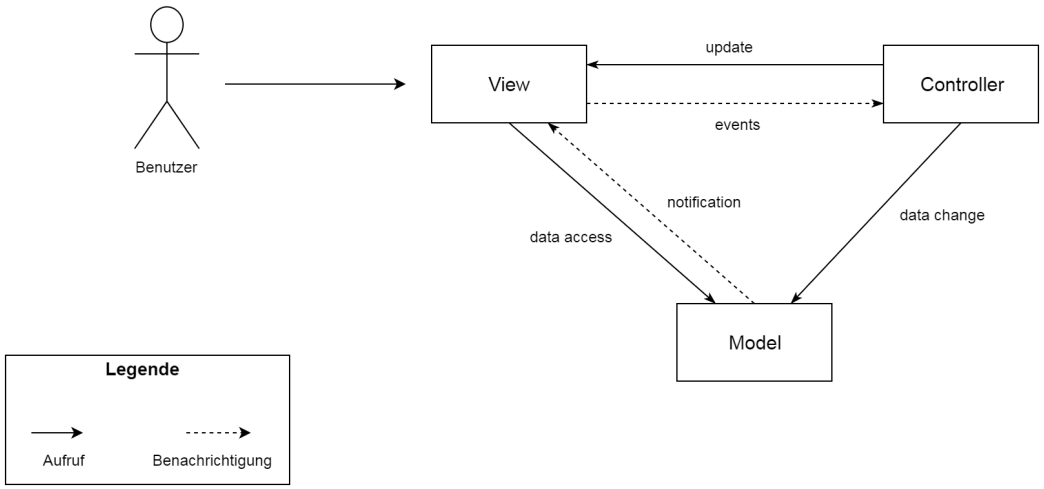

Abbildung 2.8: Layer Bridge ([Sta15])

Eine Ausnahme gilt jedoch bei Webanwendungen, bei denen der Benutzer über den Webbrowser mit der Applikation interagiert. Hier können die klassischen Notifizierungsmechanismen nicht eingesetzt werden, da die Interaktion des Clients mit dem Server über das HTTP-Protokoll abläuft und jede Antwort an den Browser eine HTTP-Anfrage voraussetzt. In so einem Fall werden Client und Server in eigenständige MVC-Triaden zerlegt oder Notifikationen werden über Websockets gesendet [Wiketd]. Ein großer Vorteil des MVC Musters ist eine klare Trennung der Verantwortlichkeiten. So wird bei korrekter Umsetzung des MVC-Musters das Zielsystem flexibel und leicht zu erweitern sein. Andererseits ist nicht definiert, in welcher der MVC Komponenten die Geschäftslogik eines Systems untergebracht wird. Traditionell wird die Geschäftslogik in der Controller Komponente implementiert, wobei einige MVC-Frameworks strikt vorschreiben, die Geschäftslogik in der Model Komponente zu implementieren [Wiketd].

2.6 Folgen von nicht konsequent umgesetzter Architektur

Dieser Abschnitt befasst sich mit den Folgen, die auftreten, wenn Softwaresysteme ohne architektonische Planungsgrundlage implementiert wird. Im Anschluss wird erklärt, wie Softwaresysteme auf bestimmte architektonische Eigenschaften untersucht werden können. Softwaresysteme sind oft länger in Gebrauch als die ursprünglich von den Betreibern geplant. Hinzu kommt, dass diese oft angepasst und erweitert werden. Lediglich Systeme die zur Migration von Daten aus einem Altsystem in die Strukturen der neuen Anwendung ausgerichtet sind, finden nur eine einmalige Verwendung [Lil12]. Viele Unternehmen scheuen eine Neuentwicklung ihrer Softwaresysteme, da eine Neuentwicklung teuer ist und die Einführung neuer Softwaresysteme mit Risiken verbunden ist. Finanzielle Einbußen kommen während der Zeit der Systementwicklung hinzu, da die Organisation durch die Neuentwicklung ausgebremst wäre. Aus diesen Gründen werden bestehende Systeme erweitert und angepasst. Eine korrekt umgesetzte und implementierte Architektur gewährleistet die Erweiterbarkeit von Softwaresystemen. Andernfalls entstehen technische Schulden und die Erweiterbarkeit des Systems erschwert sich [Lil12]. Frau Lilienthal definiert technische Schulden wie folgt: „Technische Schulden entstehen, wenn bewusst oder unbewusst falsche oder suboptimale technische Entscheidungen getroffen werden. Diese falschen oder suboptimalen Entscheidungen führen zu einem späteren Zeitpunkt zu Mehraufwand, der Wartung und Erweiterung verzögert“[Lil12]. Dabei reicht das alleinige Entwerfen einer guten Architektur nicht aus, diese muss auch bei der Entwicklung des Systems umgesetzt werden. Dafür sollten während der Systementwicklung und Erweiterung regelmäßige Refactoring Arbeiten am System vorgenommen werden, damit technische Schulden beseitigt werden und die geplante Architektur ohne Abweichungen implementiert wird [Lil12]. Abbildung 2.9 zeigt ein Diagramm, welches diesen Aufarbeitungsprozess des Systemcodes veranschaulicht.

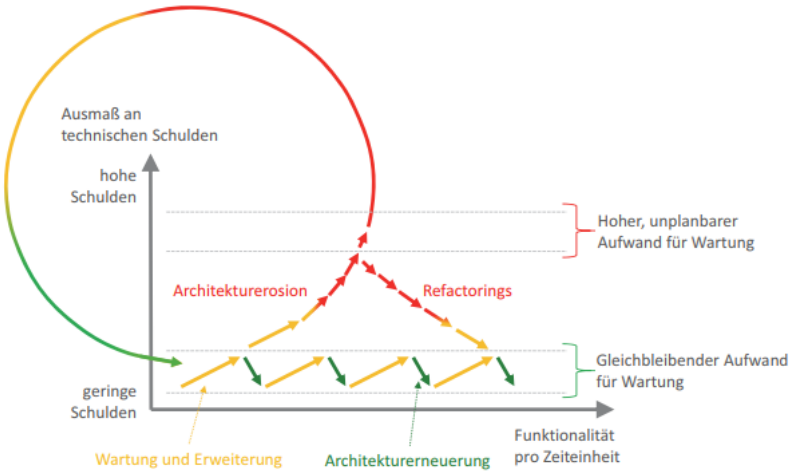

Abbildung 2.9: Technische Schulden und Architekturerosion ([Lil12])

Die gelben Pfeile stellen technische Schulden dar die entstehen, die bei der Wartung und Erweiterung von Softwaresystemen entstehen. Die Enden der Pfeile stellen das Ende der Erweiterungsphase dar. Darauf folgt eine Aufbesserung der technischen Schulden, welche als grüne Pfeile im Diagramm gekennzeichnet ist. So werden die technischen Schulden geringgehalten, was die Software wartbar und skalierbar macht. Im zweiten Szenario finden keine Aufarbeitungsphasen des Quellcodes statt, so steigen die technischen Schulden mit der Erweiterung des Systems immer weiter an und die implementierte Architektur erodiert. „Die Wartung und Erweiterung der Software werden immer teurer bis zu dem Punkt, an dem jede Änderung eine schmerzhafte Anstrengung wird. In Abbildung 2.9 wird dieser Umstand dadurch deutlich gemacht, dass die roten Pfeile immer kürzer werden“ [Lil12]. Die Folgen davon sind, dass Änderungen an dem Quellcode fehleranfällig wird und Änderungen am System sehr schwerfällig werden. Im schlimmsten Fall ist eine Weiterentwicklung des Systems zu aufwendig und somit zu teuer. Das hat zu Folge, dass das System neuentwickelt werden muss [Lil12]. Herr Martin beschreibt Symptome angehäufter technischer Schulden folgendermaßen: "Das System ist unflexibel gegenüber Änderungen. Jede Änderung hat weitere abhängige Änderungen zu Folge, sodass Entwickler mühselig viel Zeit mit der Reparatur davon verbringen. Des Weiteren verhält sich das System sehr fragil, sodass bei Änderungen Folgefehler an unerwarteten Codestellen auftauchen. Weitere Symptome für technische Schulden sind redundante Systemeinheiten. So

werden Konstruktionseinheiten die eine ähnliche Aufgaben lösen, neu implementiert, da die Lösungen nicht wiederverwendet werden können"[Mar06].

2.7 Analyse von implementierten Systemarchitekturen

Das Ziel bei der Analyse von Systemarchitekturen ist das Aufspüren von technischen Schulden. Technische Schulden können in der dokumentierten Architektur eines Systems vorkommen, noch bevor diese implementiert ist. Ein Vorgehen für Architekturentscheidungen ist die ATAM Methode, in der Qualitätsziele herausgearbeitet werden und durch Szenarien konkretisiert werden[Lil12]. Auf diesen Szenarien aufbauend werden die passenden architektonischen Entscheidungen getroffen, damit Abhängigkeiten, empfindliche Punkte und Risiken der Herangehensweise aufgedeckt werden [Wiketa]. Eine andere Methode ist es, den Sourcecode eines Systems auf technische Schulden zu analysieren. Dafür wird die im Sourcecode implementierte Architektur mit der dokumentierten Architektur abgeglichen. Die im Sourcecode implementierte Architektur wird als Ist-Architektur und die dokumentierte Architektur wird als Soll-Architektur bezeichnet. Die Soll-Architektur ist eine Abstraktion der Ist-Architektur und zeigt Bausteine des Systems, welche nicht im Sourcecode zu finden sind wie Komponenten, Subsysteme, Module und Schichten [Lil12]. Damit eine Analyse der implementierten Architektur erfolgen kann, wird auf Analysewerkzeuge zurückgegriffen. Diese werden zum einen mit dem Sourcecode des zu analysierenden Systems und zum anderen mit der dokumentierten Architektur gefüttert. Das Tool erfasst alle Objekte, Methoden, Variablen und Packages und zeichnet alle benutzt- und vererbt Beziehungen davon ein. Schließlich wird die Soll-Architektur auf den Sourcecode gemappt und die eigentliche Analyse kann durchgeführt werden. Der Sourcecode wird daraufhin auf Schichtenverletzungen und zyklische Abhängigkeiten untersucht [Lil12]. Eine Schichtenverletzung liegt vor, wenn Klassen oder Objekte einer niederen Schicht auf die Dienste einer höheren Schichte zugreifen. Voraussetzung hierfür ist, dass bei der Systemarchitektur eine Schichtenarchitektur vorliegen muss. Schichtenverletzungen verringern die Modularität von Systembausteinen, anders ist es bei zyklischen Abhängigkeiten. Eine zyklische Abhängigkeit von Klassen oder Objekten liegt vor, wenn innerhalb einer Menge von Klassen oder Objekten wechselseitige Nutz-Beziehungen existieren. Das wird auf Abbildung 2.10 deutlich, wobei die Knoten Objekte repräsentieren und die Pfeile ihre Beziehungen [Kle14].

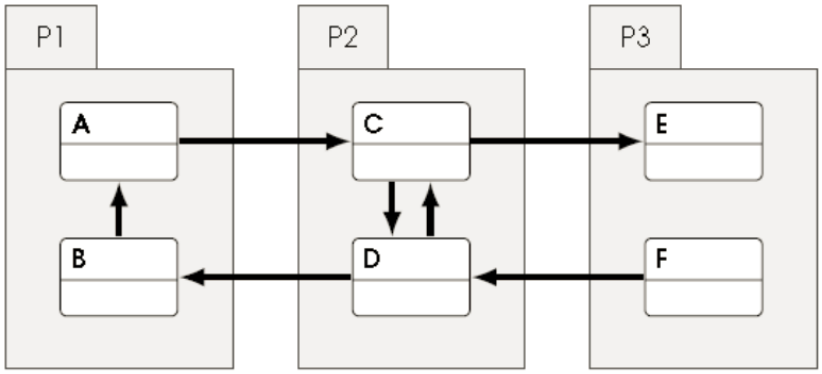

Abbildung 2.10: Zyklische Abhängigkeiten ([Kle14])

Zyklische Abhängigkeiten erschweren die Erweiterbarkeit eines Systems, da Änderungen an einem Baustein direkt zu Änderungen an verknüpften Bausteinen führt. Außerdem breiten sich zyklische Abhängigkeiten aus, da bei Erweiterung des Systems neue Bausteine zum Zyklus hinzukommen. Werden die beschriebenen Merkmale aufgespürt und eliminiert, verringern sich technische Schulden und Qualitätsmerkmale wie Verständlichkeit, Modularität oder Skalierbarkeit können leichter erreicht werden [Lil12].

3 Dokumentation, Analyse und Optimierung der Systemarchitektur von CuTE

Nachdem die theoretischen Grundlagen zur Softwarearchitektur erläutert wurden, folgt in diesem Teil die praktische Ausarbeitung. Hierfür wird in diesem Teil die Softwarearchitektur des Testtools CuTE dokumentiert. Diese Dokumentation richtet sich nach dem arc42 Template. Dabei wird in der Dokumentation, die ursprünglich angestrebte und geplante Architektur, also die Soll-Architektur des Systems dokumentiert. Danach folgt eine Architekturanalyse der im Prototyp tatsächlich implementierten Architektur, welche mit der dokumentierten Architektur abgeglichen wird, um zu ermitteln, wie stark die Architekturerosion des Prototyps ist. Anschließend folgt die Optimierung der dokumentierten Architektur, indem diese um Systemfunktionalitäten erweitert wird, welche in der dokumentierten Architektur nicht untergebracht sind.

3.1 Architekturdokumentation des Systems CuTE

Es folgt nun die Dokumentation der Systemarchitektur von CuTE. 2.3 schildert den Inhalt und den Aufbau einer solchen Architekturdokumentaion. Diese beginnt mit einer Einführung in die Aufgaben und den Zweck des Systems, gefolgt von den Einflussfaktoren und den Randbedingungen. Danach folgen die vier Sichten auf die Architektur, welche den Hauptteil der Dokumentation darstellen. Anschließend werden verwendete Konzepte und Entwurfsentscheidungen begründet, gefolgt von Qualitätsszenarien, Risiken und einem Glossar.

3.1.1 Einführung und Ziele

Zweck des Systems

Dieses Dokument beschreibt die Softwarearchitektur des CuTE Systems zum automatisierten Testen von Webanwendungen. CuTE soll Testteams dabei unterstützen, Testszenarien automatisch gegen eine beliebige Webapplikation auszuführen. Dabei ist das System bis zur Ausführung der Skripte, weitgehend von der Testumgebung gesondert. Testszenarien werden als eine Folge von Befehlen in einem Skript verfasst, welches dann von CuTE gegen eine definierte Testumgebung ausgeführt wird. Dabei ist besonders wichtig, dass diese Skripte in einer systemeigenen Sprache geschrieben werden, wofür keine Programmierkenntnisse notwendig sind, und auf diese Weise Kosten für IT-Experten sparen. Die Runs werden dann schließlich in Reports dokumentiert und verschaffen den Testern einen Überblick über erfolgreich und nicht erfolgreich ausgeführte Skripte. Betreiber und gleichzeitig Entwickler des Systems ist die Firma IBM. Ein Prototyp von CuTE ist bereits als Nebenerzeugnis im Rahmen eines IT-Großprojektes in der Firma entstanden und hatte die Aufgabe, Regression Tests an das unter Konstruktion befindende System auszuführen. Die Einsatzmöglichkeiten beschränken sich nicht auf ein einzelnes System, sondern weiten sich über alle möglichen Webapplikationen, die vom Webbrowser aufgerufen werden, aus.

Leserkreis

1. Alle im Abschnitt 3.1.4 genannten Stakeholder
2. Begutachter dieser Dokumentation

3.1.2 Fachliche Aufgabenstellung

Die wichtigsten funktionalen Anforderungen sind:

1. CuTE ist eine Software, die sich automatisch durch Webapplikationen navigieren kann. Dabei startet es einen Browser, öffnet ein Browserfenster und ruft die URL einer vom Benutzer definierten Webapplikation/Internetseite auf. Es braucht lediglich vom Benutzer erstellte Skripte, in denen die auszuführenden Szenarios definiert sind

2. Ein Skript kann ohne Programmierkenntnisse verfasst werden. Es wird in einer systemeigenen Sprache geschrieben und besteht aus einer Folge von vordefinierten Befehlen, die zum Navigieren durch Webseiten nötig sind. HTML-Seitenelemente, auf denen ein Befehl ausgeführt wird, werden lediglich durch ein Label (Beispiel: Textfeld/Button usw.) und einen Bezeichner für dieses Label identifiziert.

3. Skripte können einzeln oder mehrere parallel ausgeführt werden. Skriptgruppen können ebenfalls gebildet und sequenziell ausgeführt werden

4. Zu jeder Skriptausführung wird ein Report erstellt, der alle ausgeführten Befehle, sowie den Erfolg des jeweiligen Skriptes dokumentiert.

5. CuTE ist weitgehend von der Testumgebung abgekoppelt. Die Testumgebung beeinflusst nicht die Skripterstellung. Erst zur Laufzeit der Skripte besteht eine aktive Verbindung zu der Testumgebung

6. Der Debugg Modus bietet dem Benutzer an, Skripte Schritt für Schritt gegen eine Testumgebung auszuführen, um so vorgekommene Fehler zu reproduzieren

7. Eine benutzerfreundliche Oberfläche erleichtert dem Benutzer das Bedienen der Software

3.1.3 Qualitätsziele

Die primären Qualitätsziele von CuTE werden hier aufgelistet. Die Reihenfolge der Auflistung gibt Aufschluss über die Wichtigkeit der Ziele:

1. **Modifizierbarkeit:** Wie bereits im Abschnitt 3.1.1 erwähnt, führt CuTE eine Folge von Befehlen auf den Elementen einer HTML Seite aus. Diese Elemente werden vom System durch xPath lokalisiert. Damit das Programm ein Skript gegen eine Umgebung/Webapplikation ausführen kann, müssen die xPaths der jeweiligen Umgebung im Programm deklariert werden. Neu hinzugefügte xPaths dürfen nicht mit den bestehenden xPaths im Konflikt stehen oder sonstige Funktionalitäten der Software negativ beeinflussen. Das System muss eine weitere Schnittstelle anbieten, in der es um Skriptfunktionen erweitert werden kann. Skriptfunktionen sind Funktionen, die bei der Skripterstellung in Kombination mit Befehlen eingefügt werden.

2. **Richtigkeit:** Alle im Skript definierten Befehle müssen korrekt ausgeführt werden. Seitenelemente müssen korrekt erfasst werden und dürfen nicht vertauscht oder gar ignoriert werden. Elemente die auf einer Seite nicht vorhanden sind, dürfen nicht als vorhanden markiert werden. Die Reports dienen den Testern als Grundlage, um Defects zu erfassen und aus diesem Grund müssen Reports alle ausgeführten Befehle wahrheitsgemäß dokumentieren. So darf das Reporting keine zweifelhaften oder widersprüchlichen Angaben enthalten und muss beim Fehlschlagen eines Skriptes eine korrekte Aussage über den Grund des Fehlschlagens liefern.

3. **Flexibilität:** Skripte sollen ohne Programmierkenntnisse und ausschließlich durch eine im System definierte Syntax verfasst werden. Zum Zeitpunkt der Skripterstellung besteht keine Verbindung zur Testumgebung. Die einzige Beziehung der Skripte zur Testumgebung sind die im Skript definierten Seitenelemente und die dazugehörigen Bezeichner. Erst zur Laufzeit des Skriptes wird eine Verbindung vom Webbrowser zur Testumgebung aufgebaut. Das System kann Skripte gegen jede Webapplikation ausführen, sofern die nötigen xPaths dafür definiert sind.

4. **Performance:** Das Programm sollte zum Ausführen eines Befehls im Durchschnitt zwei Sekunden benötigen. Voraussetzung hierfür ist, dass der Browser-Cache, auf dem die Umgebung getestet wird, die Seitenelemente beinhaltet. Das System soll in der Lage sein, geringfügige Lasttests gegen ein System zu simulieren. Es muss mindestens 100 Skripte gleichzeitig gegen eine Testumgebung ausführen können.

Nicht-Ziele:

1. CuTE wird ausschließlich unternehemensintern genutzt, es ist keine Standard Software oder Produkt
2. CuTE ist kein wie bisher bekanntes Testtool, sondern ist von der Testumgebung weitgehend abgekoppelt

3.1.4 Stakeholder

Rolle	Beschreibung	Ziel
Management	Management der Watson Health Abteilung in der der Prototyp entwickelt wurde	Interesse am System, um es bei zukünftigen Projekten einzusetzen
Softwarearchitekt	Muss die Softwarearchitektur des Systems entwerfen	Aktive Beteiligung an Entwurfsentscheidungen
Softwareentwickler	Müssen den inneren Aufbau des Systems gestalten und implementieren	Aktive Beteiligung an technischen Entscheidungen
Softwaretester	Aktive Beteiligung an technischen Entscheidungen	Fehlerfreie und reibungslose Nutzung des Tools

Tabelle 3.1: Übersicht der Stakeholder

3.1.5 Einflussfaktoren und Randbedingungen

Technische Randbedingungen

Randbedingung	Erläuterung
Betriebssystem	Das System soll mit Windows und mit Unix kompatibel sein
Bedienung des Systems	Bedienung des Systems erfolgt entweder durch eine Client Shell oder durch eine Client HTML Seite mit Benutzeroberfläche
Kompatible Browser	Das System kann Skripte gegen Chrome, Firefox, Safari und Opera Browser ausführen
Technische Kommunikation	Kommunikation zwischen dem Browser/Shell und dem Server erfolgt durch HTTP, Sockets. Ausführung der Skripte auf den Browsern erfolgt mit RMI (Remote Method Invocation)
Modellierungswerkzeug	UML 2 kompatibles Modellierungswerkzeug zur Modellierung von technischen Diagrammen
Mengengerüst	Das System sollte in der Lage sein, bis zu 100 Skripte parallel gegen eine oder verschiedene Testumgebungen auszuführen. Dementsprechend müsste es bis zu 100 Clients parallel bedienen können
Programmiersprache	Das System wird in Java geschrieben

Tabelle 3.2: Technische Einflussfaktoren

Organisatorische Randbedingungen

Randbedingung	Erläuterung
Zeitplan	In 6 Monaten soll die Beta Version des Systems angefertigt sein.
Team	Das Entwicklerteam sollte aus nicht mehr als fünf Entwicklern bestehen, weil das Systems mit ca. 42.000 Lines of Code ein überschaubares Projekt ist. Bei weiteren Entwicklern wäre das Risiko von architektonischen Erosionen durch mangelnder Kommunikation höher.
Vorgehensmodell	Das System wird in Iterationen entwickelt
Technische Kommunikation	Kommunikation zwischen dem Browser/Shell und dem Server erfolgt durch HTTP, Sockets. Ausführung der Skripte auf den Browsern erfolgt mit RMI (Remote Method Invocation)
Testwerkzeuge	JUnit 4 im Annotationsstil für die Gewährleistung der inhaltlichen Korrektheit
Einsatz von Open Source	Innerhalb des Projektes werden ausschließlich Open Source Bibliotheken und Frameworks verwendet.

Tabelle 3.3: Organisatorische Einflussfaktoren

3.1.6 Kontextabgrenzung

Fachlicher Kontext

Abbildung 3.1 zeigt das Systemkontextdiagram für das System CuTE. Hier sind die interagierenden Akteure, sowie Fremdsysteme eingezeichnet und im Anschluss beschrieben.

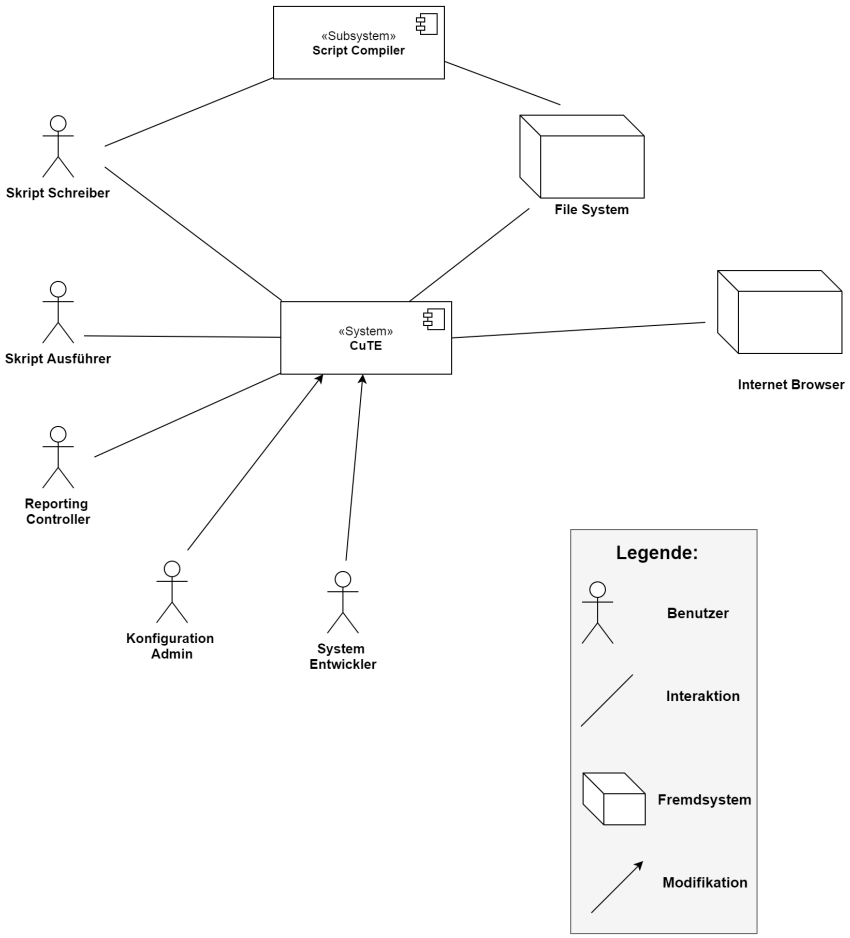

Abbildung 3.1: Fachlicher Systemkontext

Skript Schreiber (Benutzer) Der Skriptschreiber verfasst Testszenarien in für CuTE vorgesehene Testskripte. Ein Skriptschreiber hat keine Programmierkenntnisse und verfasst die auszuführenden Skriptschritte in einer von CuTE definierten Syntax. Die Skripte werden in einer Entwicklungsumgebung freier Wahl geschrieben und von einem unabhängigen Subsystem

in XML Format geparst.

Skript Ausführer (Benutzer) Der Skriptausführer ist für das Starten der Testskripte zuständig. Bevor die Skripte ausgeführt werden, müssen diese in das Skriptverzeichnis des Systems hereingeladen werden. Skripte können nach Zugehörigkeit geordnet und sequenziell oder parallel ausgeführt werden. Skripte müssen von einem Client ausgeführt werden. Dieser ist entweder als Shell oder als HTML Seite mit Benutzeroberfläche verfügbar.

Reporting Controller (Benutzer) Der Reporting Controller überwacht die Ergebnisse der ausgeführten Skripte. Diese Reports kann er sich entweder über den Webclient auf der HTML Seite anschauen oder er lädt sich sie herunter und begutachtet sie auf seinem lokalen Rechner. Fehlgeschlagene Skripte deuten auf Defects in der Testumgebung hin und diese Szenarien sollte der Reporting Controller manuell nachspielen und entscheiden, ob es sich tatsächlich um ein Defect handelt.

Konfiguration Admin (System Admin) Der Konfiguration Admin stellt sicher, dass die xPaths einer Testumgebung, die zum Identifizieren von Seitenelemente notwendig sind, auf dem System vorhanden sind. Er braucht Fachwissen über xPath und über die Baumstruktur der HTML Seiten der jeweiligen Testumgebungen. Das System bietet eine Schnittstelle zur Erweiterung und Kürzung von xPath Ausdrücken. Er hat ebenfalls die Aufgabe, die das System über eine Schnittstelle um Skriptfunktionen zu erweitern. Außerdem stellt er ein, ob die Ausführung der Skripte mit dem Browser auf Server oder Clientseite ausgeführt wird.

Wartungsmanager (Entwickler) Der Wartungsmanager kümmert sich über die Instandhaltung des Systems. So müssen potentielle Bugs mit geringerem Aufwand ausgebessert werden. Des Weiteren muss er das System um neue projektabhängige Funktionalitäten erweitern mit geringem Aufwand erweitern.

Internet Browser (Fremdsystem) Skripte werden mit Hilfe eines Webbrowsers gegen eine definierte Testumgebung ausgeführt. Dabei kann der Browser, auf dem die Testumgebung aufgerufen wird, auf der Maschine gestartet und bedient werden, auf der das System läuft. Die Skripte können auch per Remote auf dem Browser eines anderen Rechners gestartet werden. Voraussetzung dafür ist, dass das CuTE System auch auf der Zielumgebung läuft.

Script Compiler (Subsystem) Der Script Compiler ist ein unabhängiges Subsystem wel-

ches bei dem Skriptschreiber lokal eingerichtet wird. Er unterstüzt eine „Domain Specific Language (DSL)“, die mit einer beliebigen Entwicklungsumgebung verknüpft werden kann. Diese DSL stellt die Syntax, die fachliche Logik, sowie einen Parser der Skriptsprache zur Verfügung. So können, sich wiederholende Befehle als „Templates“ gruppiert werden. Bezeichner für Befehle lassen sich ebenfalls gruppieren und mit verschiedenen Eingabesets kombinieren. Diese werden dann zu einem XML Skript geparset, welche dann vom System gegen eine Testumgebung ausgeführt werden.

File System (Fremdsystem) Das File System dient als Ablageort für Testskripte und Reports. Das System und die Benutzer haben Zugriff drauf.

Technischer Kontext

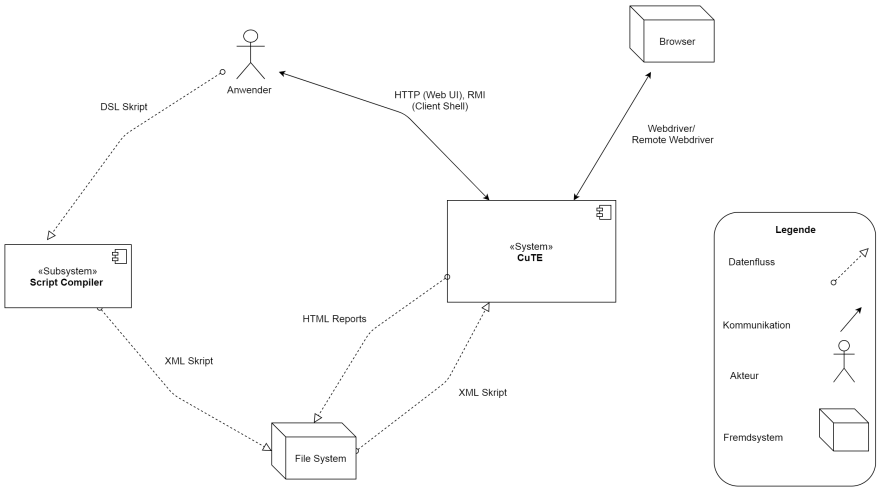

Abbildung 3.2: Technischer Systemkontext

Auf Abbildung 3.2 wird der Datenfluss und die Kommunikation zwischen dem System und dem Anwender, sowie den Fremdsystemen dargestellt. Wichtig hierbei ist, dass CuTE nur XML Skripte verwerten und ausführen kann. Dafür muss der Tester Skripte in der Domänenspezifischen Sprache verfassen, woraus dann der Script Compiler ein XML Skript generiert. Diese gelangen dann in ein File System, worauf CuTE dann zugreift. Von CuTE Seite gelangen die

Reports zu den Test Runs im HTML Format in das File System. Die Kommunikation zwischen dem Anwender und CuTE geschieht entweder über http, wenn dieser die Web UI benutzt oder über RMI, wenn dieser den Shell Client benutzt. Die Interaktion zwischen CuTE und Webbrowsern wird durch Webdriver realisiert. Der Browser kann sich auf andere Rechner als die des Systems befinden, dann findet die Kommunikation per Remote Webdriver statt. Voraussetzung hierfür ist, dass CuTE auf dem Rechner läuft.

System/ Nachbarsystem	**Kommunikation/ Protokoll**
Anwender zu Script Compiler (Subsystem)	Bekommt als Input ein DSL Skript und Generiert daraus ein XML Skript
Tester zu CuTE (System)	XML Skripte werden mit File System ins Skript Verzeichnis des Servers geladen. Kommunikation zwischen Tester und System erfolgt über RMI und ggf. HTTP beim Webclient
Admin zu CuTE (System)	Kommunikation erfolgt über RMI und ggf. http beim Webclient
Script Compiler (Subsystem) zum File System (Fremdsystem)	XML Skripte
CuTE (System) zum File System (Fremdsystem)	Reports in HTML Format
CuTE (System) zum Browser (Fremdsystem)	Kommunikation erfolgt über Webdriver oder Remote Webdriver wenn Skript nicht auf lokalem Browser ausgeführt werden soll

Tabelle 3.4: Kommunikation der Systeme

3.1.7 Lösungsstrategie

Nachfolgend werden die Lösungsstrategien und die wesentlichen Architekturaspekte des Systems dargestellt und veranschaulicht. Eine Ausführliche Beschreibung der Lösungskonzepte sind im Unterkapitel 3.1.13 vorzufinden. Die Struktur von CuTE lässt sich grob in folgende Teile zerlegen:

1. Ein Skript Parser als unabhängiges Subsystem zum Erstellen von Skripten
2. Ein Hauptsystem zum Ausführen von Skripten

Das Hauptsystem basiert im hohen Maße auf eine Client Server Architektur, welche wiederrum in Schichten gegliedert ist. Die Interaktion zwischen Client und Server basiert auf Remote Procedure Calls (RPC). Die automatisierte Steuerung des Browsers und die Ausführung der Skripte erfolgt durch Webdriver, was die Lose Kopplung von der Testumgebung sicherstellt. Im Subsystem wird eine Domänenspezifische Sprache (DSL) definiert, die das Schreiben von Skripten ohne Programmierkenntnisse ermöglicht. Dieses Subsystem erstellt aus dem DSL Skript ein XML Skript welches von Hauptsystem ausgeführt werden kann.

Lösungsansätze für die Qualitätsziele

In Tabelle 3.5 werden zu den Qualitätszielen des Systems passende Architekturansätze zugeordnet, um einen leichten Einstieg in die Lösung zu gewähren.

Qualitätsziel	**Maßnahmen zur Umsetzung**
Erweiterbarkeit für alle Webapplikationen (Modifizierbarkeit)	xPath soll als Identifikationswerkzeug von HTML Seitenelemente dienen.Schnittstelle für Erweiterung von xPath und Skriptfunktionen bereitstellen
Korrektes Ausführen von Befehlen (Richtigkeit)	Hohe Testabdeckung als Sicherheitsnetz der Richtigkeit
Lose Kopplung von Testumgebung (Flexibilität)	Die Skripterstellung geschieht unabhängig von der Testumgebung. Verwendung von Webdriver zum Ausführen von Skripten
Lastest Simulation (Performance)	Asynchrone Ausführung von Skripten. Ausführen der Skripte per Remote Webdriver auf verschiedene Rechner

Tabelle 3.5: Lösungsstrategien zur Umsetzung der Qualitätsziele

3.1.8 Bausteinsicht Level 1

Level 1 des CuTE Systems

CuTE lässt sich in vier Module und ein separates Subsystem aufteilen. Die Module werden im Diagramm als Subsystem bezeichnet. Es handeln sich dabei um zwei Client Module, ein Web Server Modul und ein CuTE Server Modul. Jeder der Level 1 Blackboxen verantwortet unmittelbar einen Teil der CuTE Gesamtfunktionalität. Die gestrichelten Pfeile in Abbildung 3.3 repräsentieren fachliche Abhängigkeiten der Subsysteme untereinander. Die kleinen Quadrate stehen für Interaktionspunkte zu Außenstehenden. Nachfolgend werden alle Bausteine einzeln beschrieben.

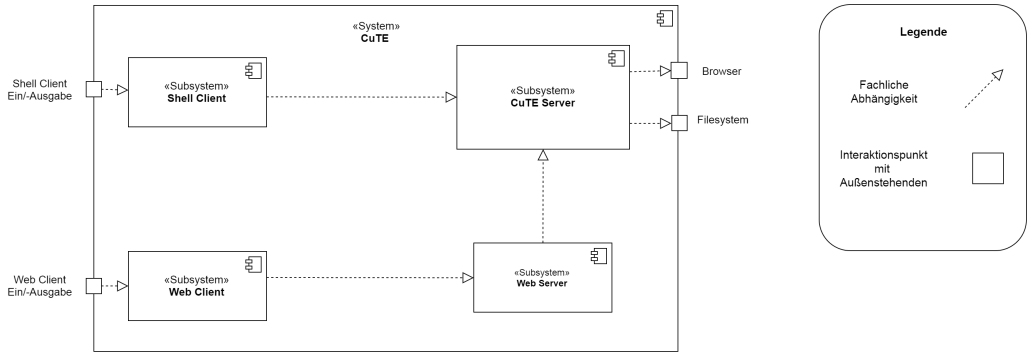

Abbildung 3.3: Bausteinsicht der ersten Ebene von CuTE

CuTE Server (Blackbox)

Zweck/Verantwortlichkeit

Der Shell Client realisiert die Kommunikation mit den Anwendergruppen im Abschnitt 3.1.6 beschriebenen Benutzern. Bedienung erfolgt über eine Shell. Dieses Subsystem stellt das Herz des Gesamtsystems dar. Es bedient als Server die Anfragen der Clients. Sämtliche Programmierlogik des Systems wird in diesem Subsystem implementiert. Diese lässt sich ausschließlich über den Client steuern. Sie ist hauptsächlich zuständig für die Ausführung von XML Testskripten in Browsern und das Erstellen der dazugehörigen Reports. Jegliche Art von Systemkonfigurationen geschieht über den Client, sogar das Erweitern der Testumgebungsspezifischen xPath Ausdrücke.

Schnittstellen

RMI: Die Client RMI Schnittstelle ermöglicht dem Shell Client und dem Web Server die Interaktion mit dem CuTE Server. Diese beinhaltet alle Befehle die das System zur Steuerung anbietet.

Webdriver: Das System implementiert die Webdriver Schnittstelle, um die Browser zu steuern. Diese Schnittstelle wird bei der Ausführung der Skripte im Browser verwendet.

Remote Webdriver: Diese Schnittstelle wird ebenfalls von dem CuTE Server implementiert, um Skripte auf entfernten Maschinen auszuführen.

File System: Diese Schnittstelle wird implementiert, damit eine Interaktion mit dem File System stattfinden kann.

Offene Punkte

Zurzeit wird ein einziges Interface mit allen Funktionen die zur Steuerung des CuTE Servers vorgesehen sind, über RMI den Clients zur Verfügung gestellt. Eine Aufteilung dieses Interfaces nach Funktionalität der Funktionen ist sinnvoller.

Web Client (Blackbox)

Zweck/Verantwortlichkeit

Ähnlich wie der Shell Client realisiert dieses Subsystem die Kommunikation des Benutzers mit dem System über eine HTML UI. Die Funktionalitäten des Web Client sind nicht so breit gefächert wie die des Shell Clients, sondern enthalten nur ein Subset der von der Shell Client angebotenen Funktionalitäten.

Schnittstellen

Web Server http : Diese Schnittstelle ermöglicht dem Web Client über http mit dem Web Server zu kommunizieren, da keine direkte Kommunikation mit dem CuTE Server stattfindet, sondern über den Web Server.

Offene Punkte

Beim Prototyp wurde die UI mit dem Javascript Framework Bootstrap gebaut. Neuere Frameworks wie zum Beispiel Foundation bieten benutzerfreundliche Elemente zum Realisieren der UI.

Web Server (Blackbox)

Zweck/Verantwortlichkeit

Der Web Server regelt die Kommunikation zwischen dem Web Client und dem CuTE Server. Dieser hat die Aufgabe, die Anfragen des Web Clients entgegen zu nehmen und sie gegen den CuTE Server auszuführen und die jeweiligen Antworten des CuTE Servers dem Web Client zur Verfügung zu stellen. Der Web Server kommuniziert mit dem CuTE Server über die selbe RMI Schnittstelle wie der Shell Client auch.

Schnittstellen

http : Diese Schnittstelle kann vom Web Client über http angesprochen werden. Sie dient der Kommunikation zwischen Web Server und Web Client.

CuTE Server RMI: Der Client benutzt die CuTE Server RMI Schnittstelle um die Funktionalitäten des CuTE Servers zu benutzen. Dabei handelt es sich um dieselbe Schnittstelle, welche auch von dem Shell Client genutzt wird.

Offene Punkte

Der Web Server beim Prototyp ist ein Jetty Server. Neuere Serversprachen wie zum Beispiel Node.js lassen sich einfacher implementieren und lassen sich einfacher implementieren.

Shell Client (Blackbox)

Zweck/Verantwortlichkeit

Dieses Subsystem realisiert die Kommunikation des Benutzers mit dem System über eine Shell. Der Client liest Befehle über die Shell ein und überprüft diese auf Korrektheit und führt sie via RMI gegen den Server aus. Die Befehle lassen sich grob als Lesebefehle, Konfigurationsbefehle und Ausführungsbefehle kategorisieren.

Schnittstellen

CuTE Server RMI : Der Client benutzt die CuTE Server RMI Schnittstelle um die Funktionalitäten des CuTE Servers zu benutzen.

Level 1 des Script Compilers

Der Script Compiler basiert auf dem xText Framework, welches aus der Sprachdefinition der domänenspezifischen Sprache (DSL) einen Parser generiert. Eine Komponente Namens „Generator“ benutzt den Parser um aus den DSL Skripten XML Skripte zu generieren, welche mit dem CuTE System kompatibel sind. Die gestrichelten Pfeile in Abbildung 3.4 repräsentieren fachliche Abhängigkeiten und die kleinen Quadrate stehen für Interaktionspunkte zu Außenstehenden. Die durchgehende Linie stellt eine „benutzt“ Beziehung dar.

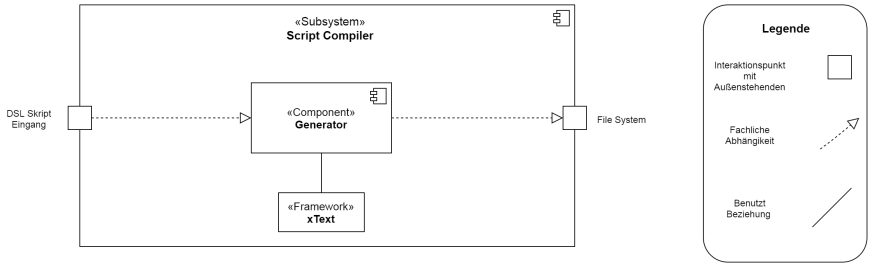

Abbildung 3.4: Bausteinsicht der ersten Ebene von CuTE

Generator (Blackbox)

Zweck/Verantwortlichkeit

Diese Komponente benutzt das xText Frame, um aus den Testskripten welche das Format der CuTE Skriptsprache haben, XML Skripte zu generieren. Das xText Framework stellt ein Syntax Highlighting und einen Parser zur Verfügung. Diese XML Skripte werden dann in das File System abgelegt.

Schnittstellen

Skript Eingang: Diese Schnittstelle realisiert das Einlesen von Skripten in einer definierten Sprache.

File System: Legt die generierten XML Skripte in das File System ab.

3.1.9 Bausteinsicht Level 2

Auf der zweiten Ebene der Bausteinsicht werden die Blackbox Bausteine Web Client, Shell Client, Web Server und CuTE Server aus Abschnitt 3.1.8 beleuchtet. Der Script Compiler wird hier nicht weiter verfeinert, da das weitere Zerlegen der Komponenten bereits auf Implementierungsebene wäre.

Web Client (Whitebox)
Der Web Client realisiert die Bedienung des Systems über eine Benutzeroberfläche. Sie besteht aus einer HTML UI, einer Applikation Komponente und Applets. Die JSApplikation Komponente ist für die Eingabeverarbeitung, Aktualisierung der UI zuständig und die Kommunikation mit dem Web Server zuständig.

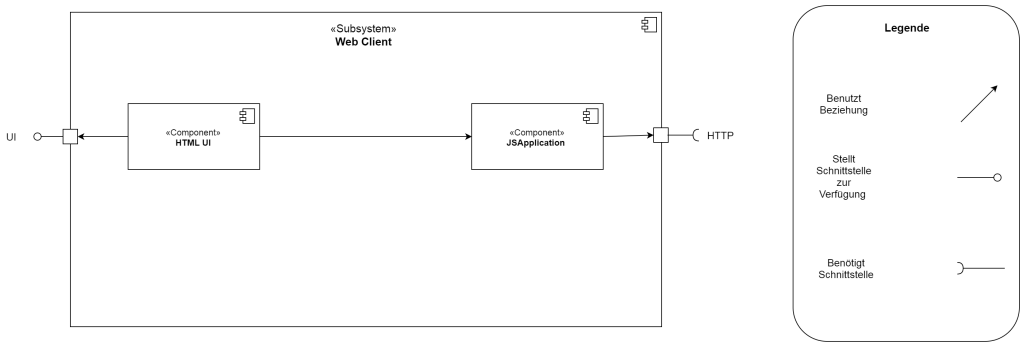

Abbildung 3.5: Level 2 Bausteinsicht des Web Clients

HTML UI (Blackbox)

Zweck/Verantwortlichkeit
Stellt dem Benutzer eine UI zur Verfügung welche über den Browser aufgerufen werden kann, womit dieser dann das CuTE System benutzen kann.

Schnittstellen
UI: Die UI stellt dem Benutzer eine Benutzeroberfläche zu Verfügung
JSApplication : Benutzt JSApplication um Oberflächenelemente mit einer Funktionalität zu verknüpfen.

JSApplication (Blackbox)

Zweck/Verantwortlichkeit

JSApplication ist eine Komponente welche auf Ereignisse des Benutzers und des Servers reagiert und mit diesem kommuniziert. So werden vom Benutzer ausgeführte Befehle an den Webserver weitergeleitet oder bei Ereignissen des Webservers die UI aktualisiert.

Schnittstellen

UI: Die UI stellt dem Benutzer eine Benutzeroberfläche zu Verfügung

http : Kommuniziert über http mit dem Webserver

Web Server (Whitebox)

Des Web Server Subsystem besteht aus einem Jetty Server, welcher über http anzusprechen ist und Servlets. In den Servlets sind die vom CuTE Server bereitgestellten RMI Methoden aufgerufen, mit denen die Anfragen des Clients gegen den CuTE Server ausgeführt werden.

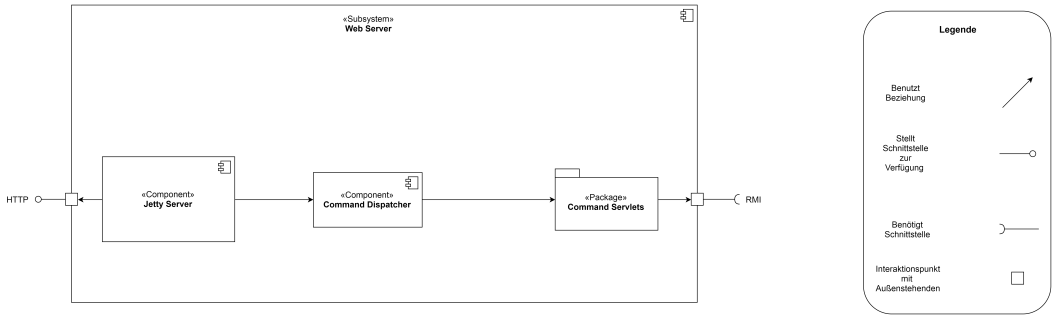

Abbildung 3.6: Level 2 Bausteinsicht des Web Servers

Jetty Server (Blackbox)

Zweck/Verantwortlichkeit

Wie aus dem Namen dieser Komponente hervorgeht, handelt es sich dabei um einen Jetty Webserver, der die http Anfragen der Webclients bedient. Dabei wird bei den Anfragen zwischen normalen Anfragen und Servlet Anfragen unterschieden. Die Servlet Anfragen werden an den Servlet Container weitergeleitet.

Schnittstellen

http : Diese Schnittstelle stellt der Jetty Server für die Clients nach außen zur Verfügung

Servlet Container : Die Servlet Anfragen werden über diese Schnittstelle an den Servlet Container weiter geleitet

Command Dispatcher (Blackbox)

Zweck/Verantwortlichkeit

Der Command Dispatcher bekommt von Jetty Server Servlet spezifische Anfragen und ermittelt das für die Anfrage benötigte Servlet. Außerdem leitet dieser die Antworten der jeweiligen Servlets an den Webserver weiter, worauf sie dann zurück zum Client gesendet werden. So hat der Command Dispatcher eine Brückenfunktion zwischen dem Server und den Servlets.

Schnittstellen

Jetty Server : Bekommt über diese Schnittstelle die Servlet Anfragen des Servers
Servlets : Ruft über diese Schnittstelle die benötigten Servlets auf

Command Servlets (Blackbox)

Zweck/Verantwortlichkeit

Das Command Servlets Package beinhaltet alle Servlets, welche zum Realisieren der Funktionalitäten des Web Clients benötigt werden. Die Servlets sind aufgrund funktionaler Aspekte strukturiert. Diese benutzen das vom CuTE Server bereitgestellt RMI um dessen Funktionen zu nutzen und damit zu kommunizieren. Tabelle 5 im Anhang bietet eine Übersicht über alle im Web Server verfügbaren Servlets.

Schnittstellen

RMI: Dient zur Benutzung der CuTE Server Funktionen, sowie zur Kommunikation mit dem CuTE Server.

CuTE Server (Whitebox)

Der CuTE Server ist fachlich strukturiert und unterteilt sich in einer Execution Engine, einer Reporting Komponente und einer Server Config Komponente. Die Execution Engine stellt das Herzstück des gesamten Systems dar und beinhaltet sämtliche Logik, welche zur Ausführung von Skripten notwendig ist. Die Reporting Komponente ist für die Erstellung von Reports notwendig und in der Server Config Komponente werden Servereinstellungen verwaltet.

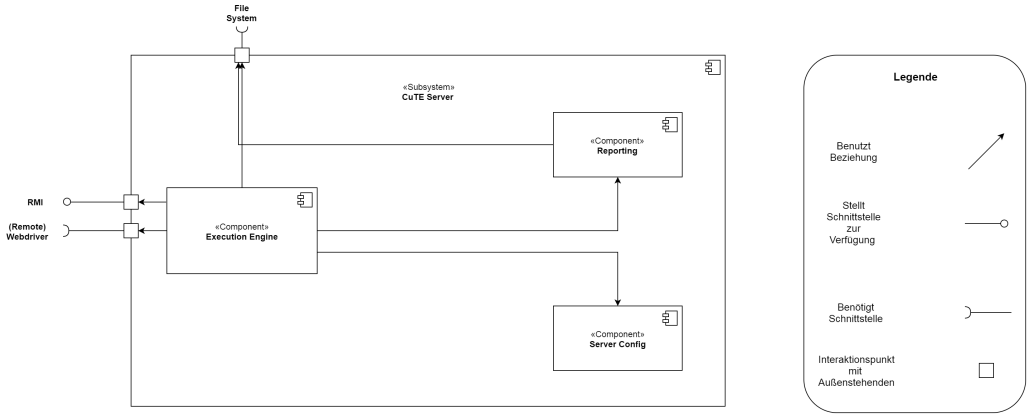

Abbildung 3.7: Level 2 Bausteinsicht des CuTE Servers

Execution Engine (Blackbox)

Zweck/Verantwortlichkeit

Die Execution Engine ist für die Ausführung von Testskripten gegen eine definierte Testumgebung auf dem Browser zuständig. Diese holt sich die Skripte aus dem Filesystem und führt die Befehle davon über den Webdriver im Browser aus.

Schnittstellen

RMI: Bietet diese RMI Schnittstelle den Clients an. Kommuniziert darüber mit dem Client und wird darüber auch vom Client gesteuert.

Webdriver: Bedient über diese Schnittstelle den lokalen Browser.

(Remote) Webdriver: Bedient über diese Schnittstelle Webbrowser die sich nicht auf der lokalen Umgebung befinden.

File System: Zugriff auf Testskripte und Reports.

Reporting (Blackbox))

Zweck/Verantwortlichkeit

Wie aus dem Namen dieser Komponente hervorgeht, ist diese für das Erstellen der Reports der jeweiligen Testruns zuständig.

Schnittstellen

File System: Fertig erstellte Reports werden auf das File System abgelegt .

Execution Engine: Kommunikation mit der Execution Engine zur Erstellung der Reports

Server Config (Blackbox)

Zweck/Verantwortlichkeit

In dieser Komponente werden Applikationsspezifische Einstellungen wie die zu testenden Umgebungen, der Browser auf dem die Skripte ausgeführt werden oder auch auf der Execution Server gespeichert.

Schnittstellen

Execution Engine: Übergibt Ausführungsrelevante Parameter an die Execution Engine

Client Shell (Whitebox)

Die Client Shell wird vom Benutzer in der Kommandozeile ausgeführt. Diese besteht aus vier Kommando Einlese Komponenten, welche die einzelnen Menüs des Shell Clients wiederspiegeln. Jedes Menü hat eigenspezifische Befehle, die nur in dem jeweiligen Menü ausgeführt werden können. Die Applikation startet im ersten Menü und stellt Benutzer Befehle zur Verfügung. Wenn ein bestimmter Befehl ausgeführt wird, wird das zweite Menü gestartet, wo der Benutzer wiederum neue Befehle ausführen kann. So kann der Benutzer sich bis in das vierte Menü durchnavigieren. Alle Befehle werden per RMI gegen den Server ausgeführt. Die Befehle sind in den jeweiligen Packages abgelegt und jede Shell hat Zugriff auf eines dieser Packages. Jeder Befehl wird in mindestens einer separaten Klasse implementiert. Eine Übersicht mit Beschreibung der jeweiligen Shell Befehle wird im folgenden Abschnitt behandelt.

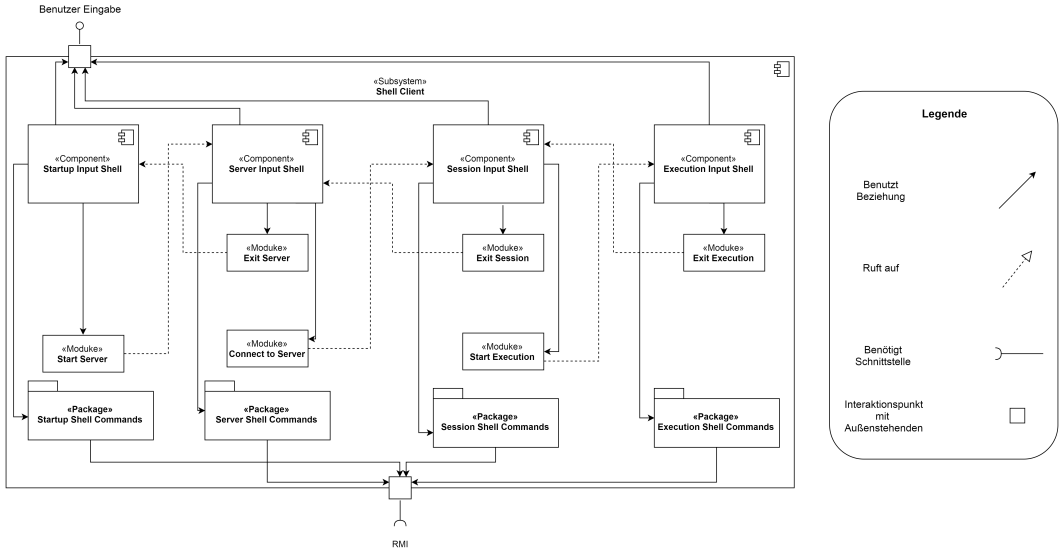

Abbildung 3.8: Level 2 Bausteinsicht des Web Servers

Startup Input Shell (Blackbox)

Zweck/Verantwortlichkeit

Diese Komponente ist die erste Kommando Shell und erscheint, wenn das System gestartet wird. Diese stellt dem Benutzer eine Reihe von Befehlen zur Verfügung, welche im Package „Startup Shell Commands“ vorhanden sind. Einzig das Modul „Start Server“ beinhaltet die Befehle, die einen CuTE Server starten oder sich mit einem bereits laufenden CuTE Server verbinden. Diese ruft dann die neue Command Shell auf, welche in diesem Fall die Server Input Shell ist. Tabelle 5.2 im Anhang zeigt alle verfügbaren Befehle, welche dem Benutzer in der Startup Shell zur Verfügung.

Schnittstellen

Startup Commands: Über diese Schnittstelle werden die vom Benutzer eingelesenen Befehle ausgeführt.

Benutzer Eingabe : Liest Befehle des Benutzers ein.

Server Input Shell (Blackbox)

Zweck/Verantwortlichkeit

Die Server Input Shell wird aufgerufen, wenn der CuTE Server gestartet wird. Genau wie die Startup Input Shell stellt diese Kommandozeile dem Benutzer serverspezifische Befehle zur Verfügung. Diese sind im Package Server Shell Commands abgelegt. Der Befehl „Connect to Server“ ruft die Session Input Shell auf und startet eine Client Session mit dem CuTE Server. Die wichtigsten Befehle werden in Tabelle 5.3 im Anhang aufgelistet und beschrieben.

Schnittstellen

File System: Fertig erstellte Reports werden auf das File System abgelegt .

Execution Engine: Kommunikation mit der Execution Engine zur Erstellung der Reports

Session Input Shell (Blackbox)

Zweck/Verantwortlichkeit

Die Session Input Shell wird aufgerufen, wenn der Benutzer sich über die Server Input Shell mit dem Server verbindet. Genau wie die anderen Shells des Systems stellt auch diese Kommandozeile dem Benutzer Befehle zur Steuerung des Servers zur Verfügung. Die Befehle sind im Package Session Input Commands abgelegt. Die wichtigsten Befehle werden in Tabelle 5.4 aufgelistet und beschrieben.

Schnittstellen

Session Shell Commands : Über diese Schnittstelle werden die vom Benutzer eingelesenen Befehle ausgeführt

Benutzer Eingabe : Liest die Eingaben des Benutzers ein

Execution Input Shell (Blackbox)

Zweck/Verantwortlichkeit

Der Benutzer kommt in die Execution Input Shell, wenn er diese Aufruft oder ein Testrun startet. Hier können hauptsächlich Testskripte ausgeführt werden. Die Befehle die diese Shell dem Benutzer zur Verfügung stellt sind im Package Execution Shell Commands abgelegt. Die wichtigsten Befehle werden im Anhang in Tabelle CommandExecutionShell hier aufgelistet.

Schnittstellen

Benutzer Eingabe: Liest die Eingaben des Benutzers ein

Execution Shell Commands : Führt die eingelesenen Befehle im Execution Shell Commands Package aus

3.1.10 Bausteinsicht Level 3

Dieser Abschnitt detailliert lediglich den Blackbox Baustein die Execution Engine, welche das zentrale Herzstück des Systems darstellt.

Execution Engine (Whitebox)

Die Komponente stellt das Herzstück des Systems dar und ist für die Ausführung der Skripte gegen eine definierte Testumgebung verantwortlich. Das CuTE Session Impl Modul übernimmt die Kommunikation mit den Clients. Für jedes Skript Ausführung wird eine Instanz des CuTE Controllers erstellt, welche die Skriptausführung tätigt und überwacht. In Tabelle 3.1.10 werden alle Bausteine aufgelistet und beschrieben.

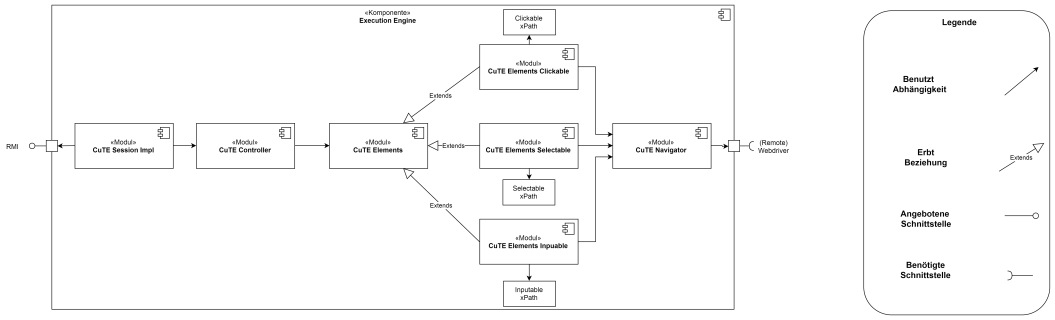

Abbildung 3.9: Whitebox Sicht der Execution Engine

Modul	Beschreibung
CuTE Session Impl	Diese Komponente stellt zum einen die Methoden für das RMI in das dafür vorgesehen Registry zur Verfügung und zum anderen für die Implementierung dieser Methoden zuständig. Wie aus dem Namen des Moduls hervorgeht regelt diese auch die Sessions für die Clients.
CuTE Elements	CuTE Elements ist das Mutterobjekt von dem die drei verschiedenen Elementtypen erben. Diese haben die Aufgabe die Ergebnisse des CuTE Navigators zu verarbeiten und mit damit zu kommunizieren.
CuTE Elements Clickable	Erbt von CuTE Elements und beinhaltet für klickbare Elemente spezifische Such- und Ausführungsoperationen. Liest alle spezifischem xPath über die Properties Datei ein.
CuTE Elements Inputable	Erbt von CuTE Elements und beinhaltet für ausfüllbare Elemente spezifische Such- und Ausführungsoperationen. Liest alle spezifischem xPath über die Properties Datei ein.
CuTE Elements Selectable	Erbt von CuTE Elements und beinhaltet für auswählbare Elemente spezifische Such- und Ausführungsoperationen. Liest alle spezifischem xPath über die Properties Datei ein.
CuTE Navigator	Der CuTE Navigator stellt einen Bruchteil der Funktionen des Webdrivers zur Verfügung. Diese nimmt für die jeweiligen Webdriver Befehle von den verschiedenen CuTE Elements Klassen die benötigten Argumente und führt damit die Funktionen des Web Drivers aus und leitet das Ergebnis wiederum weiter an die CuTE Elements.
Web Driver	Der Web Driver führt alle Operationen gegen den Browser aus. Zu den wichtigsten Operationen die für CuTE ausgeführt werden, zählt zum einen das lokalisieren von Seitenelement anhand von xPath, sowie das Ausführen dieser Elemente.

Tabelle 3.6: Übersicht der Module der CuTE Execution Engine

3.1.11 Laufzeitsicht

Diese Sicht visualisiert das Verhalten der Bausteine zur Laufzeit des Systems. Es werden vier grundlegende Anwendungsszenarien in der Laufzeitsicht dargestellt. Diese werden anhand von Sequenzdiagrammen auf Subsystemebene dargestellt.

Szenario: Shell Client starten und mit dem Server verbinden

Dieses Szenario beschreibt ein Prozedere, welches durchlaufen werden muss, damit der Benutzer das System benutzen kann. Dafür wird der Shell Client gestartet und startet zuallererst den CuTE Server. Nachdem der CuTE Server erfolgreich lokal gestartet wurde, verbindet sich der Shell Client mit diesem Server und kann dann Operationen ausführen. Operationen sind in diesem Fall das Auflisten aller auf dem Server verfügbaren Skripte oder das Ausführen von Skripten.

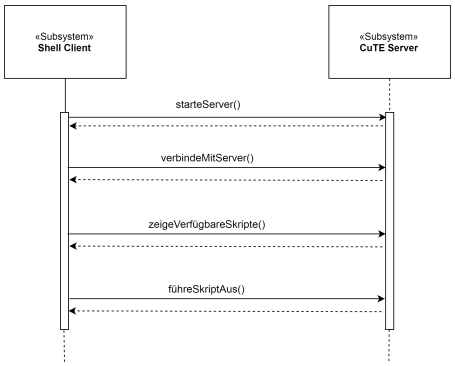

Abbildung 3.10: Laufzeitszenario: Starten und Verbinden des Shell Clients mit dem CuTE Server

Ablauf

1. Der Benutzer ruft den Shell Client auf
2. Anhand eines von Benutzer eingegebenen Befehls wird der Server gestartet
3. Als nächstes verbindet sich der Benutzer dem Server
4. Lässt sich als Beispieloperation alle auf dem Server verfügbaren Skripte liefern
5. Führt als nächste Beispieloperation ein Skript aus

Szenario: Starten des Web Clients

Dieses Laufzeitszenario beschreibt wie der Benutzer den Web Client starten kann. Wie auch der CuTE Server muss der Web Server zuerst gestartet werden. Nur dann kann der Benutzer den Web Client über den Browser aufrufen.

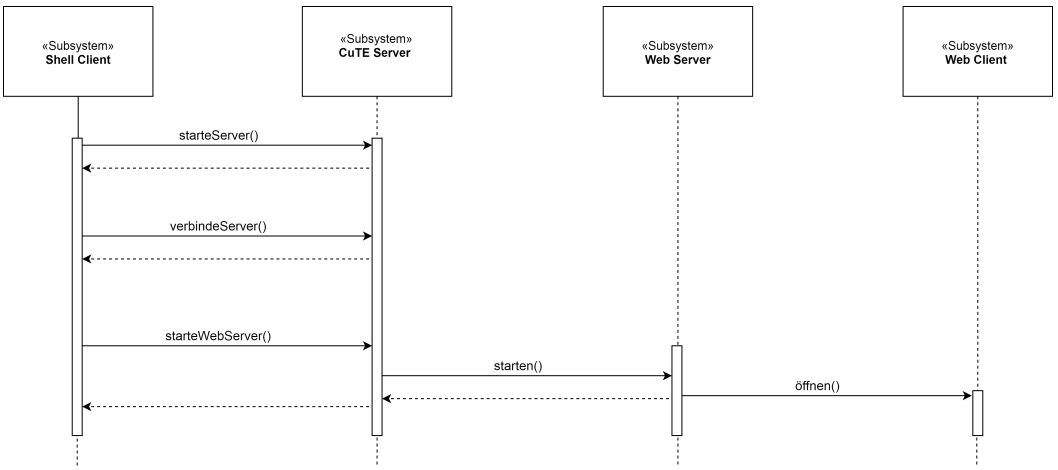

Abbildung 3.11: Laufzeitszenario: Starten des Web Clients

1. Der Benutzer ruft den Shell Client auf
2. Anhand eines von Benutzer eingegebenen Befehls wird der Server gestartet
3. Als nächstes verbindet sich der Benutzer dem CuTE Server
4. Jetzt erfolgt der Befehl zum starten des Web Server an den CuTE Server
5. Der CuTE Server startet den Web Server
6. Der Webserver öffnet den Web Client im Browser

Szenario: Bedienung des Web Clients

In diesem Szenario wird die Interaktion der Subsysteme dargestellt, wenn der Benutzer das System über den Web Client bedient. Die Voraussetzung für dieses Szenario ist, das der Web Client bereits gestartet wurde, wie es im vorherigem Szenario verdeutlicht wurde.

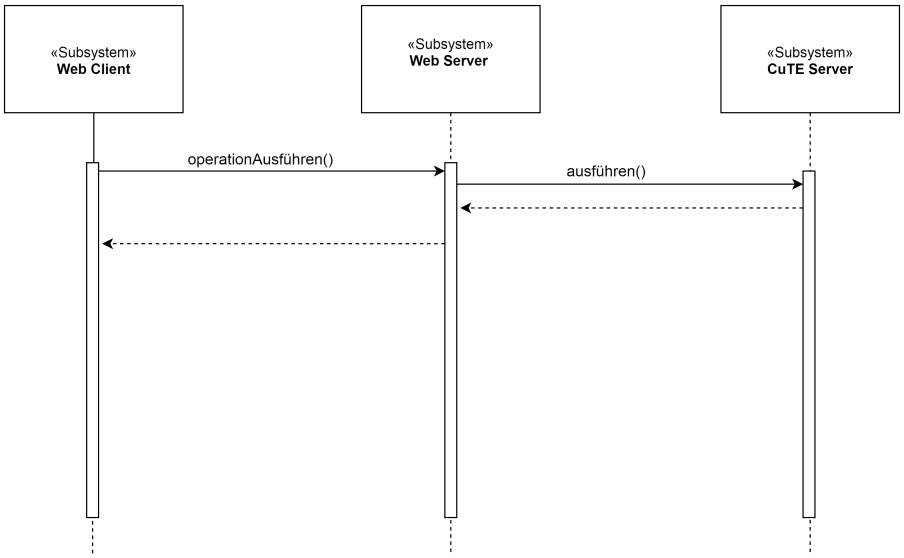

Abbildung 3.12: Laufzeitszenario: Starten des Web Clients

1. Der Web Client führt eine Operation aus
2. Der Web Server empfängt den Befehl des Web Clients und führt ihn gegen den CuTE Server aus
3. Der CuTE Server führt die Operation aus und gibt das Ergebnis an den Web Server
4. Der Web Server leitet das Ergebnis der Operation an den Web Client

Szenario: Ausführen von Skriptbefehlen

Dieses Szenario beschreibt wie die Module der dritten Ebene der Bausteinsicht miteinander interagieren, wenn ein Skript Befehl ausgeführt wird. Dabei ist wichtig zu beachten, dass es sich nur um die Ausführung eines Skriptschrittes handelt. Je nachdem wie viele Schritte ein Skript hat, wird dieser Prozess genauso oft durchgeführt.

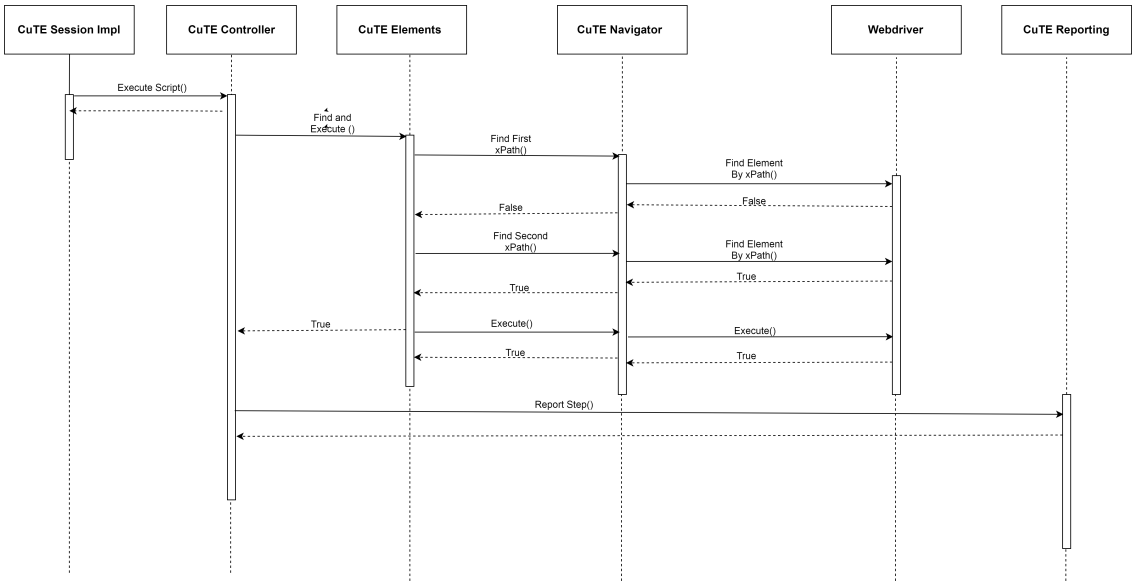

Abbildung 3.13: Laufzeitszenario: Ausführen von Skriptbefehlen

1. Die CuTE Session Impl bekommt eine Anfrage zum Ausführen von einem Testskript vom Client und übergibt es an den CuTE Controller.
2. Der CuTE Controller informiert die CuTE Session Impl, dass die Ausführung gestartet ist.
3. Der CuTE Controller holt sich den ersten Schritt und ruft das CuTE Elements Objekt auf um das gesuchte Element auf der Seite suchen und ausführen zu lassen.
4. CuTE Elements greift auf den CuTE Navigator zurück, welcher die Funktionen des Web Drivers zur Verfügung stellt. Hierfür wird für jeden einzelnen xPath des jeweiligen Elementtyps die „Find()“ Funktion aufgerufen. Hierfür wird der xPath Ausdruck und der Bezeichner des Elements an den CuTE Navigator weitergegeben

5. Der erste xPath wird vom CuTE Navigator erhalten und anhand des Webdrivers ausgeführt. Es wird für den ersten xPath Ausdruck kein Element auf der Seite gefunden also liefert der Webdriver false an den CuTE Navigator zurück. Der gibt das weiter an CuTE Elements.

6. Als nächstes wird der nächste xPath den CuTE Navigator übergeben, welcher den dann anhand des Webdrivers ausführt. Dieser Aufruf liefert ein True zurück. Das Element ist also auf der Seite vorhanden.

7. Jetzt wird das Element mit dem passenden xPath ausgeführt. Das erfolgt wieder von CuTE Elements auf den CuTE Navigator, welcher es anhand des Webdrivers ausführt.

8. Die erfolgreiche Ausführung des Skriptschrittes wird an den CuTE Controller gemeldet, welcher diesen Schritt der Reporting Komponente zum dokumentieren übergibt.

3.1.12 Verteilungssicht

Die Verteilungssicht von CuTE ist nahezu trivial, da sich alle Implementierungsbausteine der Bausteinsicht in ein einziges Java Klassenarchiv zusammengefasst werden. Dieses Java Klassenarchiv wird mit allen benötigten Libraries in eine Zip Datei komprimiert und steht in dieser Form dem Benutzer zur Verfügung. Wenn es sich um Unternehmensexterne Benutzer handelt, dann müssen diese aus rechtlichen Gründen die Libraries selber herunterladen. Maven bietet sich dafür bestens an. Dabei kann das komplette System auf einem Rechner gestartet und betrieben werden. Es ist von Betriebssystem weitgehend unabhängig und die Bedienung kann durch eine Kommandozeile erfolgen und ist somit anspruchslos. Da es sich bei dem System um eine Client Server Architektur handelt, müssen sich der Server und der Client nicht auf der selben Umgebung befinden. Die Clients können sich mit bereits laufenden CuTE Servern verbinden und die Dienste eines entfernten CuTE Systems in Anspruch nehmen. Voraussetzung hierfür ist, dass auf der entfernten Umgebung das CuTE System läuft. Um das Programm zu starten wird lediglich die JAR Datei in der Kommandozeile ausgeführt.

Infrastruktur

Der Rechner auf dem das System laufen soll muss folgende Voraussetzungen erfüllen:

- Es muss entweder ein macOS oder Windows Betriebssystem vorhanden sein
- Es muss eine Java JDK 1.8 (oder höher) vorhanden sein
- Es muss entweder ein Chrome, Firefox oder Opera Internet Browser auf der Umgebung installiert sein

3.1.13 Technische Konzepte

Schichten

Das CuTE System basiert auf einer zwei Schichten Architektur, wobei die CuTE Clients die obere Schicht darstellen und der CuTE Server die niedere Schicht. Somit ist ein Zugriff der oberen Schicht, also der Clients auf die untere Schicht, die des Servers erlaubt, umgekehrt jedoch nicht. Wie aus der Bausteinsicht hervorgeht, sind in diesem Fall der Shell Client und der Web Server Clients des CuTE Servers. Und der Web Client ist der Client des Web Servers. Der Zugriff auf die Dienste des CuTE Servers erfolgen durch entfernte Methodenaufrufe, welche durch das Remote Interface des CuTE Servers ausgeführt werden. Dieses Remote Interface wird für die Bedürfnisse der jeweiligen Clients aufgeteilt und den Clients zur Verfügung gestellt.

Kommunikation

Größtenteils basiert die Kommunikation auf entfernte Methodenaufrufe. Diese erfolgt ohne Middleware oder Gateway, sondern direkt. Die Kommunikation zwischen den Clients und dem Server findet asynchron statt. So wird gewährleistet, dass die Clients nach der Ausführung vom Befehlen für den Benutzer weiter ansprechbar bleiben. Die Komponenten der Executionengine kommunizieren synchron, da der Prozess der Skriptausführung von den Ergebnissen der einzeln ausgeführten Schritte abhängt. Bei der Aktualisierung der Weboberfläche über den Status von laufenden Skripten wird auf Polling, sowie auf den Publish-Subscribe Stil zurückgegriffen.

Automatisierte Skript Ausführung

Die automatisierte Testskriptausführung stellt die zentrale Aufgabe des Systems dar. Die Befehle eines Skriptes werden Schritt für Schritt ausgeführt. Nach jedem Schritt wird die Bedingung geprüft, ob der folgende Schritt erfolgreich ausgeführt wurde oder ob dieser fehlgeschlagen ist. Es wird beim Start einer Skriptausführung ein Report erzeugt, an dem für jeden ausgeführten Schritt ein Reportschritt mit den zur Laufzeit ermittelten Informationen angefügt. Beim Beendigung des Skriptes wird dieses Objekt XML geschrieben und mittels XSL in ein HTML Report umgewandelt.
Bei der Ausführung eines Schrittes wird

1. im Browsercache überprüft, ob zu dem aktuellen Schritt, welcher aus einem Label und einem Bezeichner besteht, bereits ein xPath existiert, welcher im aktuellen Schritt ausgeführt werden soll. Wenn dieser tatsächlich gefunden wird, dann wird dieser ausgeführt

und der Schritt wird als erfolgreich im Report dokumentiert. Wenn der xPath nicht im Cache gefunden wird, dann geht es weiter mit Punkt 2

2. Es wird über alle im System eingepflegten xPath des jeweiligen Elementtyps iteriert und mit den xPath der aufgerufenen HTML Seite verglichen. Elementtypen können in diesem Fall zum Beispiel ein Button oder ein Link sein. Wenn es eine Übereinstimmung des gesuchten xPath mit dem xPath auf der HTML Seite gibt, dann wird dieser Befehl ausgeführt und der Schritt wird im Report dokumentiert. Die Suche nach dem passenden xPath erfolgt über alle Frames und iFrames welche auf der HTML Seite vorhanden sind und beschränkt sich bei modalen Dialogen nur auf diese, da die restlichen Elemente auf der Seite nicht aktiv sind. Ist der Schritt nicht erfolgreich dann geht es weiter mit Punkt 3

3. Wenn das gesuchte xPath nicht auf der HTML Seite vorhanden ist, dann wird die Ausführung gestoppt. Der Schritt an dem es scheitert wird im Report mit einem Screenshot dokumentiert.

Benutzeroberfläche

Die Benutzeroberfläche soll dem Benutzer im Gegensatz zu der Kommandozeile die Bedienung des Systems erleichtern. Die Benutzergruppe ist ziemlich eingeschränkt, da es nur für die unternehmensinterne Nutzung bestimmt ist. Die Benutzergruppen lassen sich aus dem Systemkontext herleiten. Die Benutzeroberfläche des Systems bietet dem Benutzer die Darstellung von Informationen, sowie die Steuerung des Systems an. Zur Gestaltung der HTML Oberfläche wird auf ein Frontend CSS Framework und zur Aktualisierung der Oberfläche bei Events oder anderen Informationen, sowie zur Kommunikation mit dem Webserver wird auf Clientseitige Webframeworks zurückgegriffen. Aktuell werden hierfür Bootstrap und AngularJS benutzt. Um die Benutzeroberfläche nutzen zu können, muss der Benutzer den Web Server starten, welcher die HTML Benutzeroberfläche im lokalem Netz zur Verfügung stellt. Die Benutzer müssen lediglich die Benutzeroberfläche anhand der URL des Servers im Browser aufrufen.

Persistenz

Das Speichermodel des Systems ist ausschließlich dateibasiert und erfolgt auf einem Filesystem. So werden Konfigurationen, Testskripte und Reports in das System Verzeichnis abgelegt. Der Benutzer ist selbst für das Ablegen und Aufteilen der Testskripte verantwortlich. Anders ist es bei den Reports, da diese vom System nach jeder Skriptausführung generiert und im Filesystem abgelegt werden. Dafür erstellt das System einen Verzeichnisbaum, um

dem Benutzer die Suche nach den Reports zu erleichtern. Dieser Verzeichnisbaum hat die folgende Struktur: reportVerzeichnis/NameDerTestumgebung/NameDesCuTEServers/DatumundUhrzeitDerAusführung/report.html Da aber nicht zwei Skripte zur selben Uhrzeit auf die Sekunde genau ausgeführt werden, wird der letzte Ordner im Pfad immer angelegt. Dabei werden die Ordner welche sich im Verzeichnisbaum höher befinden nur dann angelegt, wenn diese nicht bereits existieren.

Ausnahme- und Fehlerbehandlung

Für CuTE relevante Fehler und Ausnahmen betreffen hauptsächlich folgende Kategorien von Fehlern:

1. Syntaktisch fehlerhaftes XML Testskript
2. Nicht Lokalisieren von Seitenelementen, wenn diese jedoch tatsächlich auf der Seite vorahnden sind

Der erste Fall wird vollständig durch den Script Compiler abgedeckt, da dieser dem Skriptschreiber über die DSL nur eine beschränkte Anzahl an Befehlen zur Verfügung stellt und die XML Skripte syntaktisch korrekt generiert. Solange alle Änderungen am Skript über den Script Compiler erfolgen, werden die XML Skripte syntaktische keine Fehler vorweisen. Bei einer manuellen Bearbeitung der XML Skripte mit herkömmlichen Editoren, wo der Script Compiler nicht eingebettet ist, könnten Fehler entstehen. Sobald ein XML Skript syntaktische Fehler aufweist, wird es vom System nicht ausgeführt. Beim dritten Fall muss der Systemadministrator sicherstellen, dass die xPath der Seitenelemente der jeweiligen Testumgebungen korrekt eingepflegt werden. Bei nichtvorhandensein eines bestimmten xPath eines Elements wird die Ausführung fehlschlagen.

3.1.14 Qualitätsszenarien

Die wichtigsten Qualitätsszenarien sind in Tabelle 5.6 im Anhang aufgelistet.

3.1.15 Risiken

Die Risiken für die Umsetzung dieses Projektes sind bekannt, da bei der Umsetzung des Prototyps bereits alle Risiken aufgedeckt wurden. Das einzige noch bestehende Risiko betrifft die dynamische Datenerstellung für die Skripte. Da diese Funktionalität nur teilweise und über alle Komponenten verteilt implementiert wurde, ist der Umfang nicht bekannt.

3.1.16 Glossar

Das Glossar bietet eine Erläuterung von den Fachbegriffen, welche in dieser Dokumentation verwendet wurden. Das Glossar befindet sich im Abschnitt 5.7 des Anhangs vorzufinden.

3.2 Architekturanalyse des aktuell vorhandenen Prototyps

Es wird eine Sourcecode Analyse des aktuell vorhandenen Prototyps CuTE betrieben, damit diese anhand von ausgewählten architektonischen Merkmalen bewertet werden kann. Das Ergebnis dieser Analyse wird festlegen, ob der Sourcecode des Prototyps aufgearbeitet werden kann oder ob dieser neu geschrieben werden sollte. Da für den Prototyp im vorherigem Abschnitt die Softwarearchitektur dokumentiert wurde, kann diese mit der tatsächlich implementierten „Ist-Architektur“ verglichen werden. Des Weiteren wird der Sourcecode des Prototyps auf ausgewählte Architekturmerkmale untersucht. Diese Merkmale sind zum einen die Wartbarkeit und zum anderen die fachliche Flexibilität der Architektur. Grundlage für die Wartbarkeit des Systems ist unter anderem die einfache Verständlichkeit des Systems für die Entwickler, die Fähigkeit schnelle Änderungen am System vorzunehmen, sowie eine schnelle Fehleranalyse. Unter der fachlichen Flexibilität versteht man die Unterstützung der Geschäftsprozesse von verschiedenen Anwendergruppen, sowie die Anpassbarkeit des Systems an geänderten Anforderungen. Ein Indiz für eine gute Wartbarkeit des Systems ist, wenn die Systemkomponenten hierarchische angeordnet sind. Diese sollten in hierarchischen Schichten eingeteilt sein, wobei Komponenten der oberen Schichten auf Komponenten der unteren Schicht zugreifen dürfen jedoch nicht umgekehrt. Ein Merkmal um die fachliche Komplexität zu untersuchen besteht darin, die Softwarekomponenten auf bidirektionale Abhängigkeiten sowie Abhängigkeitszyklen zu untersuchen [Lil12].

3.2.1 Grobe Anfangsanalyse

Die Analyse des Prototyps erfolgt durch das Analysetool Sonargraph von hello2morrow. Der erste Schritt der Analyse besteht darin, den Sourcecode in das Tool einzuspielen. Nachdem das Parsen des Codes beendet ist, zeigt der Sonargraph technische Eckdaten des Systems, wie Abbildung 3.14 darstellt. Wie zu erkennen ist, hat das System ein Umfang von ca. 43.000 Lines of Code (LOC), wobei die Kommentarzeilen nicht miteingerechnet werden. Außerdem sind die Anzahl der Packages sowie der Komponenten aus dieser Abbildung zu entnehmen. Der Eintrag „Component“ steht hier für die Anzahl der Single Source Files (SSF).

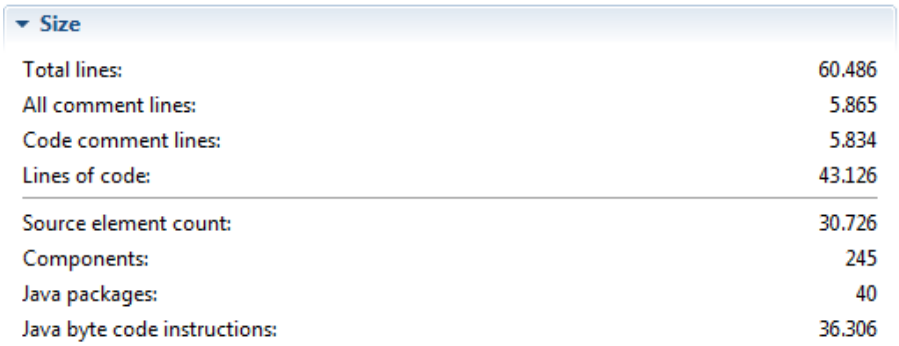

Abbildung 3.14: Eckdaten des Prototyps

Ein weiterer Screenshot aus dem Tool,welches eine signifikante Voraussage zu der Struktur des Systems bietet, ist auf Abbildung 3.15 zu sehen. Der Kürzel „ACD“ steht für „Average Component Dependency“ und gibt an, von wie vielen Single Source Files (SSF) ein einziges Single Source File im Durchschnitt abhängig ist. Das bedeutet, wenn man eine Änderung an einem Single Source File vornehmen würde, würde sich diese Änderung im Durchschnitt auf 83 andere Files auswirken, weil diese voneinander abhängig sind. Der Wert „Cyclic components“ steht für Files des Systems, die zyklisch voneinander abhängen und beträgt 86. Wenn man diesen Wert mit dem Wert der gesamten im System vorhanden Files vergleicht, welcher 245 beträgt, dann wird deutlich, dass ca. ein Drittel der Files zyklisch voneinander abhängen. Eine ähnliche Aussage kann über die im Projekt verwendeten Packages getroffen werden, dessen Gesamtzahl sich auf 41 beläuft, wovon 28 im zyklischen Zusammenhang stehen. Der Eintrag „Structural debt index“ ist eine Schätzung des Arbeitsaufwandes, um die vorhandenen Zyklen aufzulösen und die Architekturerosion zu beseitigen.

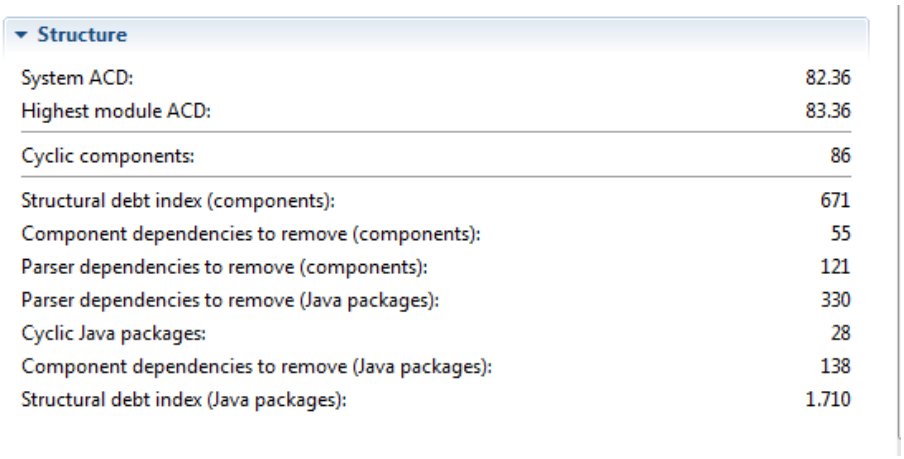

Abbildung 3.15: Eckdaten des Prototyps

3.2.2 Überprüfung auf Schichten und zyklische Abhängigkeiten

Als nächstes wird untersucht, ob die Komponenten des Systems hierarchisch angeordnet sind. Der Sonargraph stellt eine Funktion zur Verfügung mit der man die Beziehungen der einzelnen Softwarekomponenten zueinander grafisch darstellen kann. Abbildung 3.16 zeigt das System in seine Grundbestandteile zerlegt, nämlich einem Server- und einem Clientteil. Die Packages welche diese Systembausteine repräsentieren, sind als braune Rechtecke mit ihrem jeweiligen Namen symbolisiert und die grünen Bogen repräsentieren die Abhängigkeiten zwischen den Packages. Die Bögen auf der linken Seite stellen die Abhängigkeiten dar, die ein Package zu den untergeordneten Packages hat und die Bögen auf der rechten Seite stellen die Abhängigkeiten für die jeweiligen Packages zu den übergeordneten Packages dar. Die Stärke der Bögen repräsentiert die Anzahl der Verbindungen zwischen den Komponenten. Je dicker ein Bogen ist, umso mehr sind die Komponenten mit einander gekoppelt.

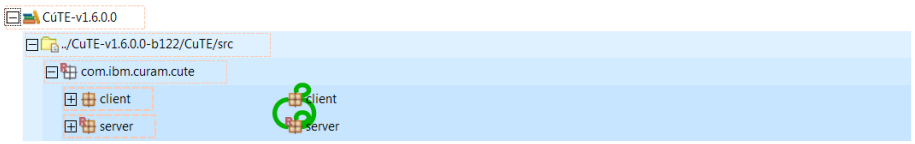

Abbildung 3.16: Aufteilung des Systems in Server und Client

Das Tool bietet an dieser Stelle die Möglichkeit, Systembausteine in Schichten einzuteilen. In diesem Fall bietet es sich an, den Server in eine niedere Schicht und die Clients in eine höhere Schicht einzuteilen, da eine Server-Client Architektur ebenfalls auf eine Schichtenarchitektur basiert. Abbildung 3.17 zeigt die Clients und den Server auf zwei verschiedenen Schichten eingeteilt. Dabei wird besonders ein roter Bogen deutlich, der vom Server zum Client führt. Dieser rote Bogen repräsentiert Abhängigkeiten der Serverkomponenten zu den Clients. Das ist eine eindeutige Verletzung, da ein Server nicht vom Client abhängig sein darf.

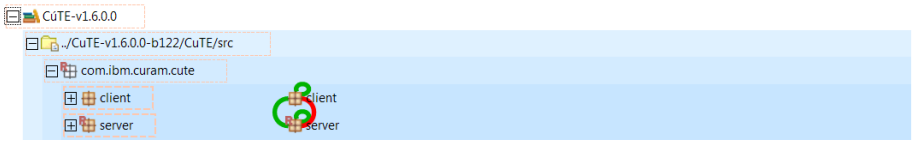

Abbildung 3.17: Definieren von erlaubten und unerlaubten Beziehungen

Wenn die Komponenten nun im Sonargraph um eine Ebene weiter aufgeklappt werden, wird das Ausmaß der Schichtenverletzungen deutlicher. Abbildung 3.18) zeigt die Bestandteile der Server und der Client Komponente mit ihren gegenseitigen Beziehungen. Es fallen besonderes viele rote Bögen auf, welche Schichtenverletzung repräsentieren.

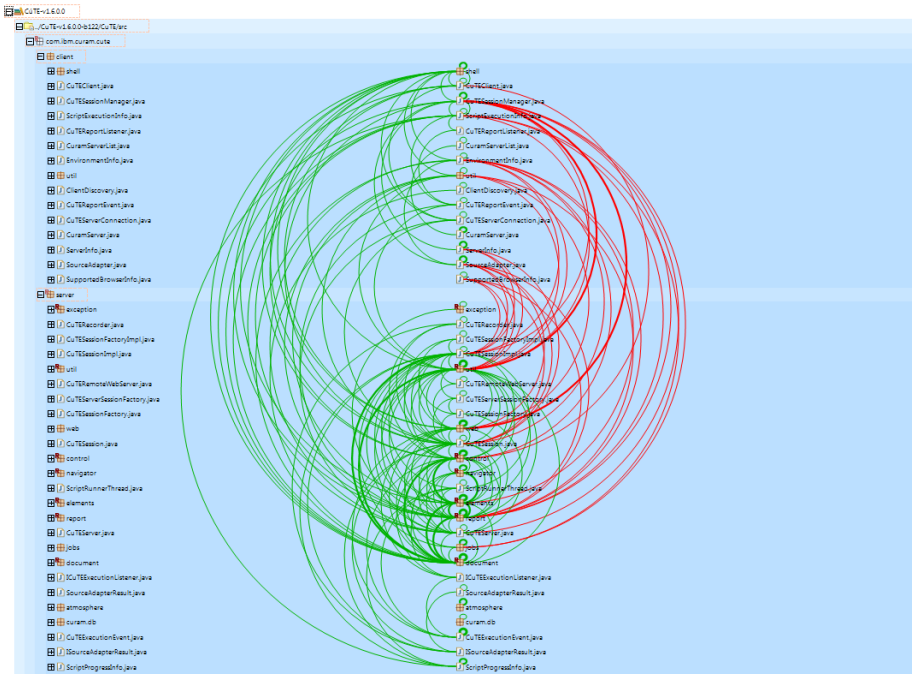

Abbildung 3.18: Weiteres Aufklappen der Komponenten

Der Sonargraph bietet eine weitere Funktion um diese Abhängigkeiten zwischen den Komponenten zu visualisieren. Damit werden die zyklischen Abhängigkeiten zwischen den Komponenten besonders deutlich, wie auf Abbildung 3.19 zu erkennen ist. Die blauen Boxen repräsentieren hier die einzelnen Komponenten und die grünen Pfeile ihre Abhängigkeiten. Ein ausgehender Pfeil bedeutet, dass die Komponente von der Zielkomponente abhängt und ein eingehender Pfeil bedeutet die Umkehrung davon. Besonders gut geht hervor, dass jede Komponente von fast jeder anderen Komponente des Systems abhängig ist. Besonders auffällig sind hier die Abhängigkeitszyklen der Komponenten, welche in einer Vielzahl auftreten, wie die Dicke der Pfeile zeigen. Hieraus geht hervor, dass die Architektur deutliche Defizite aufweist, weil sie bereits aus dieser grobgranularen Sicht sehr viele zyklische Abhängigkeiten vorweist.

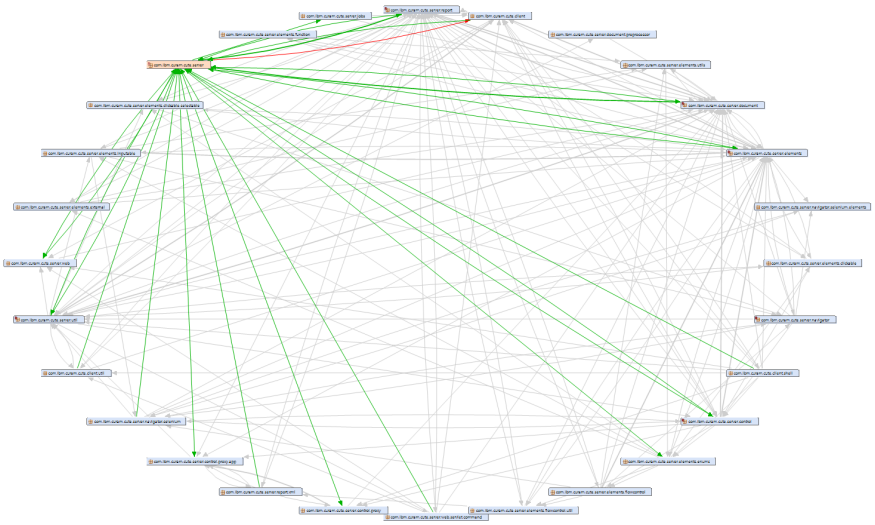

Abbildung 3.19: Zyklische Abhängigkeiten der Serverkomponenten

3.2.3 Fazit der Analyse

Bereits anhand dieser grobgranularen Untersuchung ist zu erkennen, dass das Refactoring des Prototyps sehr aufwendig sein wird. Schwache hierarchische Strukturen, sowie die starke Verbundenheit auf hoher Ebene der Systemkomponenten führen zu der Annahme, dass der Prototyp eine monolithische Struktur aufweist. Im Falle eines Refactoring wäre eine Einarbeitung in die Systemstruktur auf feingranularer Ebene notwendig, um die zyklischen Abhängigkeiten und Schichtenverletzungen aufzulösen und den Code auf Basis des im Abschnitt 3.1 dokumentierten Architekturplans, neu zu strukturieren. Das ist ein zeitaufwendiger Prozess und die Tatsache, dass sogar der Hauptentwickler des Systems keinen absoluten Durchblick des Sourcecodes mehr hat, würde diesen Prozess verlangsamen. Aus diesen Gründen ist es vorteilhafter das System auf Basis der dokumentierten Architektur neu zu implementieren.

3.3 Optimierung der vorhandenen Architektur

Dieser Abschnitt befasst sich mit der Teiloptimierung der im Abschnitt 3.1 dokumentierten Systemarchitektur und klärt über ein grundlegendes Defizit davon auf. Dabei ist es wichtig, zwischen der dokumentierten Systemarchitektur und der tatsächlich umgesetzten Architektur zu differenzieren. Die Analyse des Prototyps hat verdeutlicht, dass die implementierte Architektur von der dokumentierten Architektur stark abweicht. Das bedeutet jedoch nicht, dass die dokumentierte Architektur verworfen werden müsste, sondern dass das System auf Basis dieser Architektur implementiert werden muss. Nichts desto trotz bedarf die dokumentierte Architektur einer Erweiterung. Im Prototyp sind zwei schwerwiegende Funktionalitäten implementiert, welche in der Dokumentation nicht aufgeführt wurden. Der Grund für das nicht Dokumentieren dieser Funktionalitäten liegt darin, dass diese erst bei der Verwendung des Prototyps mit der Zeit stückweise implementiert wurden. Da der zeitliche Rahmen dafür oft knapp war, wurden diese Funktionalitäten unsauber und zum Teil über verschiedene Komponenten verteilt implementiert. Das war mit einer der Faktoren, die zur Anhäufung der zyklischen und bidirektionalen Abhängigkeiten zwischen den Modulen geführt hat. Bei den Funktionalitäten handelt es sich zum einen um zur Laufzeit dynamische Datenerzeugung und zum anderen um Skriptfunktionen. Diese zwei Funktionalitäten werden in diesem Abschnitt durchleuchtet. Außerdem erfolgt eine graphische Erfassung der jeweiligen Module anhand von Komponentendiagramme, welche schließlich mit den bestehenden Systemkomponenten aus Abschnitt 3.1.10 miteingebunden werden.

3.3.1 CuTE Skriptfunktionen

Vereinfacht betrachtet ist CuTE ein System, was einzelne Befehle eines Testskriptes gegen eine definierte Testumgebung ausführt. Anfangs definierten sich Befehle lediglich durch Operationen, welche ein Benutzer im Browser auf einer HTML Seite ausführen kann. Durch immer komplexer werdende Testszenarien stiegen die Anforderungen an das System und die Basisbefehle reichten zum Verwirklichen dieser Testszenarien nicht mehr aus. Das System wurde um neue Funktionen erweitert, welche als Befehlszusätze zu den Basisbefehlen fungierten. Diese Befehlszusätze unterteilen sich in Befehlen welche mit dem Webdriver ausgeführt werden und welche die für die Ausführungssteuerung relevant sind. Tabelle 3.7 listet alle Zusatzbefehle auf welche nicht mit dem Webdriver ausgeführt werden und erläutert die jeweiligen Funktionalitäten davon. Diese werden nicht in Kombination mit den Basisbefehlen ausgeführt.

Befehl	**Beschreibung**
Import	Dieser Befehl importiert vom Benutzer Referenzierte Dateien, die für die Skriptausführung relevant sind. Diese Dateien sind ausschließlich Datensätze, welche in den Testszenarien verwendet werden. Die Datensätze müssen einer bestimmten Syntax entsprechen, sonst können diese nicht verwertet werden. Dateneingabemasken zur Erfassung von Personendaten ist ein Beispiel für dir Verwendung importierter Datensätze.
Assign	Mit Assign kann der Benutzer eine Variable erstellen und ihr ein Wert zuweisen. Diese Variable dient als Bezeichner und kann entweder als Elementtitel oder als Eingabewert genutzt werden.
If else	Wie aus dem Namen bereits hervorgeht, prüft dieser Befehl ob eine vom Benutzer definierte Bedingung zutrifft. Sollte diese Bedingung zutreffen, dann werden die entsprechenden Befehle dazu ausgeführt. Andernfalls werden diese Befehle übersprungen. Wenn die Bedingung nicht zutreffen sollte, dann wird die Ausführung nicht abgebrochen.
Repeat until	Es wird eine Folge von Skriptbefehlen so oft durchgeführt bis die Abbruchbedingung dafür erfüllt wird. Repeat Until wird meistens für massenhafte Datenerstellung genutzt.

Tabelle 3.7: Übersicht der eigenständigen Skriptfunktionen

Die Skriptfunktionen welche als Zusatzfunktionen zu den Basisbefehlen genutzt werden und den Webdriver nutzen sind in Tabelle 3.8 aufgelistet.

Befehl	**Beschreibung**
Wait	Der Wait Befehl aktiviert eine vom Skriptschreiber definierte Wartezeit. Die Wartezeit wird vom Skriptschreiber je nach Befehl festgelegt. Der Wait Befehl wird ausgeführt, nachdem der dazugehörige Basisbefehl ausgeführt wurde. Sinn dieses Befehls ist, auf Ladezeiten der Seiten zu reagieren. Oft wird dieser Befehl dann genutzt, wenn im Basisbefehl auf ein Link oder ein Button geklickt wird und das Laden der neuen Seite ein paar Sekunden in Anspruch nimmt.
Refresh Page	Wie aus dem Namen dieses Befehles hervorgeht, aktualisiert dieser Befehl die HTML Seite indem es sie neu aufruft.
Double Click	Es wird ein Doppelklick auf das im Basisbefehl referenziert Element ausgeführt.
Right Click	Es wird ein Rechtsklick auf das im Basisbefehl referenzierte Element ausgeführt.
Refno	Bei diesem Befehl handelt es sich um eine Projektspezifische Anforderung, bei der die Fallnummer eines Datensatzes erfasst wird, welcher in vorherigen Schritten (vor dem Befehl) erstellt wurde.

Tabelle 3.8: Übersicht der Zusatzbefehle

3.3.2 Einordnung der Module in die Bausteinsicht

Nachdem die Skriptfunktionen nach Funktionalität kategorisiert wurden, gilt es diese als Module in die passende Systemkomponente einzuordnen. Die dafür passende Komponente in der diese Module hineingehören, ist die CuTE Execution Engine, da die CuTE Execution Engine für die Ausführung der Skripte verantwortlich ist und diese Module Erweiterungsfunktionalitäten der Skriptausführung darstellen. Ein Blick auf das Komponentendiagram aus der Level drei 3.1.10 Bausteinsicht zeigt alle Module der CuTE Execution Engine. Aus der Beschreibung der Module geht hervor, dass der CuTE Controller für die Steuerung der auszuführenden Skriptbefehle verantwortlich ist. Da alle Befehle aus Tabelle 3.7 gewissermaßen Funktionalitäten darstellen, welche die Ausführung beeinträchtigen, muss entweder der CuTE Controller um diese Module erweitert werden, da diese ausschließlich in den CuTE Controller implementiert werden müssen. Für die Implementierung dieser Module eignen sich innere Klassen gut, da

diese keinen großen Umfang haben und ausschließlich vom CuTE Controller genutzt werden. Die zweite Gruppe von Skriptfunktionen welche als Zusatzbefehle agieren, werden ebenfalls der CuTE Execution Engine Komponente zugeordnet. Diese werden, anders als der erste Gruppe der Skriptfunktionen nicht im CuTE Controller implementiert, sondern im CuTE Elements Modul. Das CuTE Elements Modul ist das Mutterobjekt, von dem alle anderen Elementtypobjekte erben. Jeder Elementtyp repräsentiert ein Skriptbefehl Typ und hat eigenspezifische xPath. Außerdem muss bei jedem Elementtyp die Funktion „FindAndexecute“, welche von der Mutterklasse geerbt wird, eigenspezifisch implementiert werden. Der CuTE Controller bekommt eine Liste von Elementobjekten und ruft lediglich immer dieselbe Funktion „FindAndExecute“ auf. Am optimalsten erweist sich, wenn die Erbeigenschaft genutzt wird um die zweite Gruppe der Skriptfunktionen im CuTE Element Modul zu implementieren. So wird sichergestellt, dass alle Elementtypen die Skriptfunktionen erben. Diese werden schließlich mit den jeweiligen Elementtyp xPath aufgerufen.

3.3.3 Dynamische Datenerstellung zur Ausführungszeit

Dynamische Daten, welche vom System erst zur Laufzeit eines Skriptes generiert werden, haben sich bei der Benutzung des Systems, bewährt. Diese haben sich ähnlich wie die Skriptfunktionen mit fortschreitender Zeit und Benutzung des Systems über verschiedene Module angesammelt, da diese stückchenweise und ohne feste Struktur implementiert wurden. Mit dynamischen Daten sind Ausdrücke gemeint, welche vom Skriptschreiber als Platzhalter für die Bezeichner der jeweiligen Befehle genutzt werden. Diese sind vor Laufzeit eines Testskriptes noch unbekannt und können jeden möglichen Wert annehmen. Erst bei der Ausführung des Testskriptes werden die Ausdrücke vom System evaluiert und durch generierte Werte oder von der Testumgebung abgefragte Werte ersetzt. Diese dynamischen Daten werden vor allem zum Befüllen von Textfeldern und Eingabemasken genutzt. Als Beispiel nehme man eine Testumgebung, in der Benutzer anhand von Personen Daten registriert und in der Datenbank abgespeichert werden. Nachdem das dafür vorgesehene Testskript ohne die Verwendung von dynamischen Daten verfasst wurde, wird dieses mehrfach ausgeführt. Das Resultat dieser Ausführung wären identische Benutzer Datensätze in der Umgebung. Das würde zu Fehlern auf Seiten der Testumgebung führen, da Benutzer mit denselben Personendaten registriert werden würden.
Dieses Problem wird mit der Nutzung von dynamischen Daten unterbunden, wenn statt festen Werten vom System bereitgestellte Ausdrücke verwendet und zur Laufzeit ersetzt werden. Diese Ausdrücke lassen sich in zwei Kategorien aufteilen. Die erste Kategorie der Ausdrücke haben die Aufgabe, auf deklarierte Skriptvariablen zu referenzieren. Wie bereits erläutert,

kann der Benutzer Skriptvariablen erstellen und diesen Werte zuweisen oder eine Datei mit bereits deklarierten Variablen importieren. Im Skript können diese Variablen für Eingabewerte anhand einer definierten Syntax referenziert werden. Die referenzierten Variablen müssen jedoch vorhanden sein, da sonst der Befehl nicht ausgeführt werden kann. Eine deklarierte Variable kann beliebig oft im Skript verwendet werden. Die Variablen existieren nur während der Skriptlaufzeit in die sie erstellt oder importiert werden und sind nur für die Instanzen der jeweiligen Ausführungssession sichtbar. So kommt es zwischen den Skript Runs nicht zu Komplikationen. Die zweite Kategorie der dynamischen Daten stellen dem Skriptschreiber Ausdrücke zur Verfügung, welche im Skript als Platzhalter genutzt werden können, um zur Laufzeit verschiedene Arten von dynamischen Daten zu generieren. Eine Übersicht über alle diese Ausdrücke sind in Tabelle 3.9 vorzufinden. Diese Ausdrücke enthalten eine definierte Syntax und können immer dann aufgerufen werden, einem Label oder einer Variable ein Eingabewert zugewiesen wird.

Befehl	**Beschreibung**
Concat	Nimmt eine beliebige Anzahl an Zeichenketten und fügt diese zusammen. Das Ergebnis ist einer Zeichenkette die außer Buchstaben auch Zahlen und Leerzeichen enthalten kann.
Random	Nimmt als Parameter eine Obergrenze und eine Untergrenze als Zahl und generiert eine Zahl zwischen den Grenzen.
Today	Liefert das aktuelle Datum in Tag/Monat/Jahr Format zurück
Time	Liefert die aktuelle Zeit zurück.
SQL	Führt SQL Befehle gegen die Testumgebung aus, gegen die das Skript ausgeführt wird. Bedingung dafür ist, dass die Datenbankverbindung im System eingerichtet wird. Außerdem muss Verständnis über die Datenbanktabellen vorhanden sein um die passenden SQL Befehle zu verfassen. Dieser Befehl nimmt die SQL Query als Argument entgegen.

Tabelle 3.9: Übersicht über alle Befehle der dynamischen Datengenerierung

Um die Verwendung von Variablen und Ausdrücken zur dynamischen Datenerstellung besser zu veranschaulichen, wird ein kurzes Beispielskript mit der Benutzung von dynamischen Daten verfasst. Für dieses Skript werden drei Variablen mit unterschiedlichen Werten verfasst, welche dann in den Befehlen als Eingabewerte eingefügt werden. Abbildung 3.20 zeigt dieses Beispielskript. In den ersten drei Befehlen werden die Variablen deklariert. Der ersten Variable wird ein fester Wert und den restlichen dynamische Werte zugewiesen. Um die Befehle aus

Tabelle 3.9 zu verwenden, werden die Ausdrücke mit einer „@“ Notation versehen, so werden diese dementsprechend vom System ausgewertet. In den darauffolgenden Befehlen werden Textfelder mit den Variablen befüllt. Um auf Skriptvariablen zu referenzieren, wird eine $ Notation verwendet. Erst beim Ausführen der jeweiligen Befehle, werden die Werte dafür evaluiert und der Benutzer kann sehen, welchen Wert sie enthalten. Lediglich der ersten Variable wurde ein fester Wert zugewiesen, der schon vorher bekannt ist.

```
assign "Variable1" set "Test Kommentar"
assign "Variable2" set "@today()"
assign "Variable3" set "@sql(select name from Benutzer where Nachname = Tom)"

textfield "Kommentar" set "$Variable1"
textfield "Datum" set "$Variable2"
textfield "Nachname" set "$Variable3"
```

Abbildung 3.20: Beispielskript zur Verwendung von Skriptfunktionen

3.3.4 Einordnung in die Systemkomponente

Eine Einordnung nach Funktionalität der dynamischen Daten erfolgte bereits Abschnitt davor. Beide Module gehören in die CuTE Execution Engine eingeordnet, da diese Funktionalitäten der Testskriptausführung sind. Um das Benutzen und Einsetzen der Skriptvariablen zu realisieren, muss der CuTE Controller um eine Funktionalität erweitert werden, die jeden Befehl auf Variablen oder Ausdrücke prüft. Das Implementieren beider Funktionalitäten in separaten Modulen würde dazu führen, dass zyklische Abhängigkeiten zwischen diesen Modulen entsteht. Das liegt daran, dass Variablen mit Ausdrücken und umgekehrt kombiniert werden können. So ist es möglich, beim Verwenden der Ausdrücke, Variablen als Parameter zu referenzieren. Aus diesem Grund sollten beide Funktionalitäten in einem Modul implementiert werden. Das Modul muss zum einen alle importierten und deklarierten Skriptvariablen dem CuTE Controller auf Abruf bereitstellen. Zum anderen muss es die verschiedenen Befehle aus Tabelle 3.9 verarbeiten und ein korrektes Ergebnis zurückliefern können. Zum Auswerten der Ausdrücke bietet sich eine formale Sprache an, da die Ausdrücke gegenseitig und mit variablen kombiniert werden können. Abbildung 3.21 zeigt das erweiterte Komponentendiagram der CuTE Execution Engine aus der Level drei Bausteinsicht. Dieses wurde um das Modul „Dynamic Data Controller“ erweitert, in der die beschriebenen Funktionalitäten umgesetzt werden.

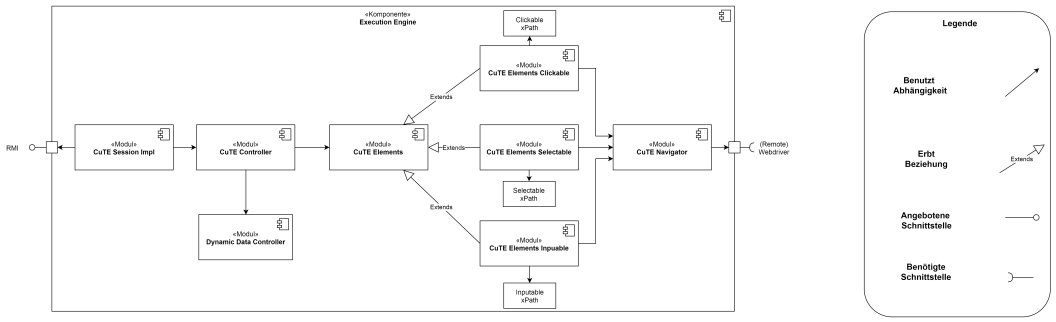

Abbildung 3.21: Erweitertes Level drei Komponentendiagram

Ein Aufbau des Moduls ist auf Abbildung 3.21 zu sehen. Das Modul setzt sich aus mehreren Komponenten zusammen. Der Input Manger wird von CuTE Controller benutzt und bekommt lediglich einen Ausdruck den dieser auswerten muss. Dafür prüft es zuerst den Ausdruck auf Variable und lässt sich diese vom Script Variable Manager liefern. Der Script Variable Manager prüft ob es die Variable schon in der Liste vorhanden ist, also ob diese schon erstellt wurde oder ob diese noch erstellt werden muss. Wenn die zu erstellen Variable eine dynamische Funktion enthalten sollte, dann greift der Script Variable Manager dafür auf den das Dynamic Data Language Objekt zurück. In diesem Objekt wird eine formale Sprache definiert, welche zu evaluieren der Ausdrücke verantwortlich ist. Dafür ruft es die jeweiligen Funktionen im Dynamic Data Functions Objekt auf, welche das Generieren der jeweiligen Datentypen übernehmen. Diese implementiert das Interface Dynamic Functions und implementiert dessen Methoden, welche den Befehlen aus Tabelle 3.9 entsprechen.
Sobald die ermittelten Variablen an den Input Manger zurückgeliefert wurden, prüft dieser ob der Ausdruck dynamische Variablenfunktionen enthält. Wenn das der fall sein sollte, dann wird der Ausdruck mit den ersetzten Variablen an das Dynamic Data Language Objekt zum evaluieren weiter gegeben.

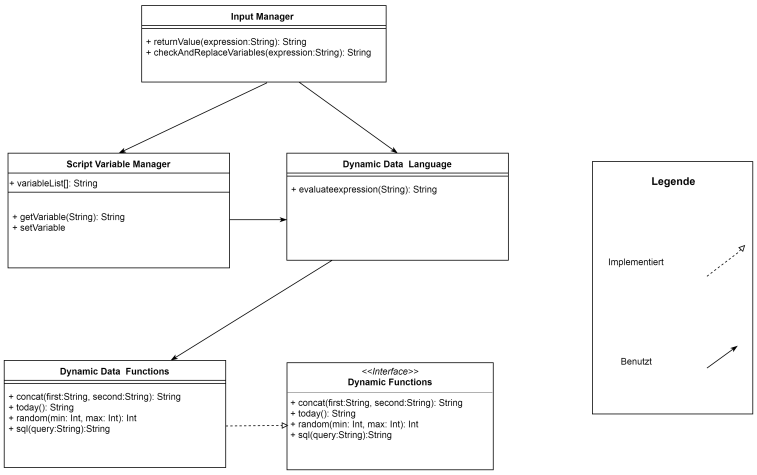

Abbildung 3.22: Aufbau des Dynamic Data Conroller Moduls

4 Zusammenfassung und Ausblick

In diesem Kapitel werden zum Abschluss der vorliegenden Bachelorarbeit die gewonnen Erkenntnisse zusammengefasst. Diese Erkenntnisse sollen ein Ausblick auf das weitere Vorgehen im Bezug auf das untersuchte System geben, vor Allem was die Neuimplementierung davon angeht.

4.1 Zusammenfassung und Erkenntnisse

Im Zuge dieser Bachelorthesis wurden verschiedene Bereiche der Softwarearchitektur erläutert, um darauf basierend die Architektur des Testautomatisierungstools CuTE abzubilden. Im Anschlusss wurde ein Analyse der Architektur durchgeführt und schließlich wurde diese optimiert. Für die Dokumentation wurde auf das arc42 Template zurückgegriffen, welche als Grundlage dafür diente. Das Dokumentieren der Architektur nach arc42, empfand ich als geeignet. Eine vorgegebene Struktur aller Bereiche einer Architketur Dokumentation, wie sie arc42 vorgibt, bietet dem Autor der Dokumentation eine feste Verfahrensweise im Bezug auf die zu dokumentierenden Themen. So fühlte ich mich nie in einer Situation, in der mir bestimmte Sachverhalte und Inhalte der Dokumentation unentschlossen vorkamen. Der Fahrplan, nach dem ich mich zu richten hatte, war zur jeder Zeit gewiss. Als alternative zu arc42 bietet sich IEEE1471 an. Bei dieser Dokumentationsvorlage fehlt die Nähe zur technischen Anwendung. So zielt arc42 primär darauf, dem Leser die technischen Konzepte, zusammen mit der Fachlichkeit zu übermitteln, wobei IEEE1471 vermehrt auf die Fachlichkeit abzielt. Des Weiteren, ist die Literatur für arc42 im deutschsprachigen Raum ausgeprägter als für IEEE1471. Beim zweiten Teil der praktischen Ausarbeitung wurde eine Anlyse und ein Abgleich der implementierten, und der dokumentierten Architektur durchgeführt. Die Sourcecode Analyse wurde mit dem kostenpflichtigen Tool Sonargraph durchgeführt. Das Tool zeichnet sich durch eine hohe Benutzerfreundlichkeit, sowie eine einfache Bedienbarkeit aus. Es erfordert keine lange Einarbeitung und präsentiert aussagekräftige Ergebnisse im Bezug auf die Analyse. Die Analyse wurde zuvor mit einem Open Source Tool durchgeführt, wobei einem das Arbeiten damit schwer fiel und das Resultat unzureichend war. So würde meine Wahl beim nächsten mal direkt auf den Sonargraph fallen. Das Ergebnis der Analyse hat nochmal vor Augen geführt,

wie verheerend die Auswirkungen architektonischer Erosion sein können. Die Erkenntnisse, welche bei dieser Ausarbeitung gewonnen wurden, erstrecken sich auf mehrere Bereiche der Softwareentwicklung, vorallem jedoch, heben Sie die Stellung der Architektur im Softwareentwicklungsprozess hervor. Diese Erkenntnis wird mich in meinen zukünftigen Tätigkeiten in der IT-Branche begleiten und prägen.

4.2 Ausblick

Im Rahmen der Bachelorarbeit ist deutlich geworden, dass das Thema Softwarearchitektur ein sehr breites und fachliches Themengebiet ist. In Softwarearchitektur steckt enormes Potential, welches oft unterschätzt und nicht maximal ausgeschöpft wird. Das war bei der untersuchten Software ebenfalls der Fall. Mit dieser Bachelorarbeit wurde der erste Schritt in die Wege geleitet, um das System auf Basis einer lückenlos dokumentierten Architektur zu implementieren. Außerdem wurden die Folgen veranschaulicht, welche bei planloser und unstrukturierter Implementierung von Softwaresystemen vorkommen. Ziele der nächsten Untersuchungen könnten sein, Schwachstellen der dokumentierten Architektur im Bezug auf Qualitätsziele und fachliche Funktionalitäten zu ermitteln. Somit könnte eine Optimierung der Architektur ähnlich wie sie in Kapitel drei Abschnitt 3.3 stattgefunden hat, angestrebt werden. Außerdem könnte der Einsatz von passenden Entwurfsmustern in Erwägung gezogen werden. Weiterhin kann untersucht werden, ob alternative Architekturstile sich für die Umsetzung eines solchen Systems besser eignen. Die Bachelorarbeit bietet bereits eine Übersicht über alternative Architekturstile, welche für die Architektur in Frage kommen. Es können auch andere Architekturstile in Betracht gezogen werden. So kann aber auch der Einsatz von Services und Microservices bei einer servicebasierten Architektur erwogen werden. Diese basieren meistens auf REST Schnittstellen, worüber die Kommunikation laufen wird. Es gilt, hierfür die Vor- und Nachteile abzuwägen. Außerdem müssten die Risiken, welche von den jeweiligen Architekturstilen ausgehen ermittelt werden, zumal keine Erfahrungswerte davon im Bezug auf das System vorliegen.

5 Anhang

Befehle der Command Servlet Komponente aus Kapitel drei Abschnitt 3.1.9

Befehl	Beschreibung
AlterEnvironmentCmd	Nimmt Änderungen an der Zielumgebung vor
AlterExecutionServerCmd	Konfiguriert den CuTE Server
BringScriptToFron t	Wählt ein von Benutzer ausgewähltes Skript aus und ladet es in die UI
ClearXpathCache	Löscht den xPath Cache des Browsers
DeleteAllReports	Lösche alle Reports welche sich im Reportverzeichnis befinden
DownloadReport	Ladet den Report herunter
ExecuteAllScripts	Führt alle Skripte die in einem Verzeichnis sind aus
ExecuteScript	Führt ein bestimmtes Skript aus
ExecuteStep	Führt einen Skriptschritt im Debug Modus aus
GetTreeView	Liefert den Verzeichnisbaum der Skriptverzeichnisse
ListReportsCmd	Listet alle verfügbaren Reports in einem Verzeichnis auf
ListScriptExecutions	Listet alle ausgeführten und noch aktuell laufende Skripte auf
ListScripts	Listet alle verfügbaren Skripte auf
ReloadPage	Ladet die Umgebungsseite neu
RemoveScriptCmd	Löscht ein Skript aus dem Verzeichnis
ClearScriptExecutions	Löscht alle auf der UI aufgelisteten Skript Ausführungen
SelectExecutionEngine	Wählt einen CuTE Server
SelectSupportedBrowser	Wählt einen Standard Browser aus auf dem die Skripte ausgeführt werden
ShowHtmlReport	Öffnet den HTML Report
StopScript	Unterbricht die Ausführung eines laufenden Skriptes

Tabelle 5.1: Übersicht der wichtigsten Befehle im Command Servlet Package

Befehle der Startup Shell aus Kapitel drei Abschnitt 3.1.9

Befehl	Beschreibung
QS	Beendet das Programm
ISF	Kopiert Testskripte von einem lokalen Verzeichnis in das Skript Verzeichnis des Systems
LX	Listet alle im lokalen Netz verfügbaren CuTE Server auf
CS	Verbindet sich mit einem der aufgelisteten Server, falls welche vorhanden
S	Auf Abbildung **??** repräsentiert das „Start Server“ Modul diesen Befehl. Es startet einen lokalen CuTE Server und ruft die Server

Tabelle 5.2: Übersicht der wichtigsten Befehle in der Startup Shell

Befehle der Input Shell aus Kapitel drei Abschnitt 3.1.9

Befehl	Beschreibung
QS	Beendet das Programm
C	Auf Abbildung ?? repräsentiert das „Connect to Server“ Modul diesen Befehl. Verbindet sich mit dem CuTE Server, startet eine Client Session und ruft die Session Input Shell auf.
ISF	Kopiert Testskripte von einem lokalen Verzeichnis in das Skript Verzeichnis des Systems
LX	Listet alle im lokalen Netz verfügbaren CuTE Server auf
SCX	Setzt einen Ordner fest, in dem die für das System relevanten xPath definiert sind
SCSI	Zeigt aktuelle Informationen über vorhandene Client Sessions
LAS	Zeigt vorhandene Client Sessions an
EXIT	Auf Abbildung 3.1.9 repräsentiert das „Exit Server“ Modul diesen Befehl. Diese beendet den CuTE Server und die Server Input Shell. So wird erneut die Startup Input Shell aufgerufen

Tabelle 5.3: Übersicht der wichtigsten Befehle in der Server Input Shell

Befehle der Session Shell aus Kapitel drei Abschnitt 3.1.9

Befehl	Beschreibung
QS	Beendet das Programm
RS	Listet alle im Skriptverzeichnis des Servers vorhandenen Skripte auf.
GASI	Listet alle Client Session Informationen
LX	Listet alle im lokalen Netz verfügbaren CuTE Server auf
LR	Listet alle im Report Verzeichnis des Systems vorhandenen Reports auf
EX	Auf Abbildung 3.1.9 repräsentiert das „Start Execution“ Modul diesen Befehl. Bei Ausführung des Befehls wird die Execution Input Shell aufgerufen.
QUIT	Beendet alle laufenden Testruns und schließt alle geöffneten Browser
ENV	Listet alle vorhandenen Testumgebungen auf
BX	Öffnet eine Testumgebung im Webbrowser und führt ein Skript auf. Nimmt als Argument die Nummer der aufgelisteten Testumgebungen und den Namen des auszuführenden Skriptes.
WS	Startet den Web Server und ermöglicht es dem Benutzer die Web Server UI im Browser aufzurufen
LSD	Listet alle vorhandenen Skripte Verzeichnisse auf
EXIT	Auf Abbildung 3.1.9 repräsentiert das „Exit Session“ Modul diesen Befehl. Diese beendet die Server Verbindung und die Session Input Shell. So wird erneut die Server Input Shell aufgerufen
UE	Ermöglicht dem Benutzer neue Testumgebungen per URL hinzuzufügen

Tabelle 5.4: Übersicht der wichtigsten Befehle in der Session Shell

Befehle der Execution Shell aus Kapitel drei Abschnitt 3.1.9

Befehl	Beschreibung
QS	Beendet das Programm
BX	Öffnet ein Browser Fenster. Nimmt als Argument den Browsertyp und die URL
M	Befüllt ein Multiples Textfeld. Nimmt als Argument das Label des Textfeldes und den zu befüllenden Wert. Kann nur ausgeführt werden, wenn vorher ein Browser geöffnet wurde.
S	Wählt einen Wert für eine Selectbox aus. Nimmt als Argument das Label der Selectbox und den zu befüllenden Wert.
B	Clickt einen Button. Nimmt als Argument das Label des Buttons
T	Befüllt ein Textfeld. Nimmt als Argument das Label des Textfeldes und den zu befüllenden Wert.
QUIT	Beendet alle laufenden Testruns und schließt alle geöffneten Browser
C	Aktiviert eine Checkbox. Nimmt als Argument das Label der Checkbox
I	Klick auf ein Icon. Nimmt als Argument das Label des Icons
L	Klickt auf einen Link. Nimmt als Argument das Label des Links
XML	Exportiert alle manuell ausgeführten Schritte in ein XML Skript
EXIT	Auf Abbildung 3.1.9 repräsentiert das „Exit Session“ Modul diesen Befehl. Diese beendet die Server Verbindung und die Session Input Shell. So wird erneut die Server Input Shell aufgerufen
R	Führt ein Testskript der Wahl aus
EX	Führt N Schritte eines ausgewählten Skriptes aus
GRS	Holt sich die Reports der ausgeführten Skripte
ENV	Zeigt alle verfügbaren Testumgebungen an

Tabelle 5.5: Übersicht der wichtigsten Befehle in der Execution Shell

Qualitätsszenarien

Nr	Szenario
1	Ein Tester lässt ein Testskript gegen eine definierte Umgebung laufen. Beim erfolgreichen Durchlaufen des Tests, kann sich der Tester sicher sein, dass es für das im Testskript ausgeführte Szenario keine Fehler in der Testumgebung auftauchen
2	Ein Tester lässt ein Testskript gegen eine definierte Testumgebung laufen welcher fehlschlägt. Der Tester weiß, dass ein Fehler in der Testumgebung vorliegen muss.
3	Der Tester kann das System anhand der Weboberfläche bedienen und erlernt die Bedienung innerhalt von wenigen Stunden
4	Der Skript Schreiber kann Skripte in der DSL des Systems schreiben und benötigt dafür keine Programmierkenntnisse
5	Testskripte können in Templates aufgeteilt werden. Templates beinhalten eine Reihe von Befehlen welche ein elementares Testszenario der zu testenden Umgebung, wie z.B. das Registrieren einer Person. Diese Templates können beliebig oft in verschiedenen Skripten wiederverwendet werden.
6	Die Skriptsprache bietet an Templates mit verschiedenen Datenobjekten zu befüllen. Hierfür werden verschiedene Datenobjekte erstellt
7	Der Tester führt mehrere Skripte gleichzeitig aus. Die Skripte können unabhängig voneinander ausgeführt werden und beeinflussen nicht gegenseitig die Testergebnisse.
8	CuTE soll bei einer neuen Testumgebung eingesetzt werden. Diese kann einzig über die URL im System eingespeichert werden.
9	CuTE soll bei einer neuen Testumgebung eingesetzt werden. Der Systemadministrator kann über eine Systemschnittstelle die umgebungsspezifischen xPath einfügen. Diese werden dann bei der Skriptausführung von der Execution Engine verwendet.
10	Testskripte können unabhängig von der Testumgebung ausgeführt werden. Ein Skript kann gegen jede Umgebung ausgeführt werden. Wenn es die falsche Umgebung sein sollte, dann wird das Skript fehlschlagen, da die auszuführenden Elemente auf der Seite nicht vorhanden sein werden.

Tabelle 5.6: Übersicht der wichtigsten Qualitätsszenarien

Glossar

Nr	Szenario
xPath	Ein Tester lässt ein Testskript gegen eine definierte Umgebung laufen. Beim erfolgreichen Durchlaufen des Tests, kann sich der Tester sicher sein, dass es für das im Testskript ausgeführte Szenario keine Fehler in der Testumgebung auftauchen [Wiketf]
Webdriver	Webdriver ist eine Sammlung von Open Source APIs, mit der ein Browser automatisiert bedient werden kann [SS17].
RMI	Remote Method Invocation stellt das Prinzip der entfernten Methodenaufrufe (Remote Procedure Call) in der Programmiersprache Java dar [Wiketb].
Servlet	Servlets sind Java Klassen, deren Instanzen innerhalb von Webservern Anfragen von Clients entgegennehmen und beantworten [Wiketg]
Shell	Eine Shell ist ein anderer Begriff für die Kommandozeile. Diese liest Befehle vom Benutzer ein [Kle]
File System	Ein File System ist eine Ablageorganisation auf einem Datenträger eines Computers [Wiketc].

Tabelle 5.7: Übersicht der wichtigsten Qualitätsszenarien

Literaturverzeichnis

[Bal11] H. Balzert. *Lehrbuch der Softwaretechnik: Entwurf, Implementierung, Installation und Betrieb.* Spektrum Akademischer Verlag, 2011.

[GS09] Peter Hruschka Gernot Starke. *Software-Architektur kompakt: - angemessen und zielorientiert.* Spektrum Akademischer Verlag, 2009.

[Hau10] Kalani Kirk Hausman. *IT Architecture For Dummies.* For Dummies, 2010.

[Kle] Matthias Kleine. Was ist eine shell. *SelfLinux*.

[Kle14] Matthias Kleine. Zyklische abhängigkeiten - terminologie. 2014.

[Lil12] Carola Lilienthat. *Langlebige Softwarearchitekturen.* DPunkt Verlag, 2012.

[Mar06] Robert C. Martin. *Agile Principles, Patterns.* Prentice Hall, 2006.

[Mar17] Robert C. Martin. *Clean Architecture: A craftman guide to software structure and design.* Prentice Hall, 2017.

[Mit15] Tilak Mitra. *Practical Software Architecture- Moving from System Context to deployment.* IBM Press, 2015.

[RK12] Paul Clements Rick Kazman, Len Bass. *Software Architecture in Practice.* Addison Weseley, 2012.

[SS17] David Burns Simon Stewart. Webdriver. *w3*, 30 March 2017.

[Sta15] Gernot Starke. *Effektive Softwarearchitekturen.* Carl Hanser Verlag, 2015.

[TB03] Jürgen Vaupel Thomas Birkhölzer. *IT-Architekturen.* VDE-Verlag, 2003.

[Vog09] Oliver Vogel. *Software-Architektur: Grundlagen - Konzepte - Praxis.* Spektrum Akademischer Verlag, 2009.

[Wiketa] Wikipedi. Atam. *Wikipedia*, zuletzt am 27. Februar 2018 um 12:13 Uhr bearbeitet.

[Wiketb] Wikipedia. Remote method invocation. *Wikipedia*, zuletzt am 1. September 2017 um 19:17 Uhr bearbeitet.

[Wiketc] Wikipedia. Dateisysteme. *Wikipedia*, zuletzt am 24. Januar 2018 um 12:37 Uhr bearbeitet.

[Wiketd] Wikipedia. Model view controller. zuletzt am 25. Februar 2018 um 17:46 Uhr bearbeitet.

[Wikete] Wikipedia. Client-server-modell. zuletzt am 27. Februar 2018 um 12:13 Uhr bearbeitet.

[Wiketf] Wikipedia. Xpath. *Wikipedia*, zuletzt am 31. August 2017 um 09:34 Uhr bearbeitet.

[Wiketg] Wikipedia. Servlet. *Wikipedia*, zuletzt am 5. März 2018 um 09:15 Uhr bearbeitet.

[Zö16] Stefan Zöner. *Softwarearchitekturen Dokumentieren und Kommunizieren.* Carl Hanser Verlag, 2016.

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 16. Juli 2018 Mohammad Saber Solimany