TECHNISCHE HOCHSCHULE LÜBECK

Bachelor-Thesis

Entwurf und Implementierung eines Frameworks zur Automatisierung von Tests webbasierter Anwendungen

Vorgelegt von: Jan-Ole Hübner

Fachbereich: Elektrotechnik und Informatik

Studiengang: Informatik/Softwaretechnik

Erstprüfer/in: Professor Dr. Krause

Ausgabedatum: 04. März2019
Abgabedatum: 04. Juni 2019

A. Hanemann

(Professor Dr. Andreas Hanemann)
Vorsitzender des Prüfungsausschusses

Aufgabenstellung:

Händische Tests können gerade bei Webanwendungen schnell sehr umfangreich und aufwendig werden. Es empfiehlt sich eine Automatisierung dieser Vorgänge. Hierzu soll ein Framework entworfen werden, welches die Erstellung automatisierter Testabläufe für webbasierte Anwendungen stark vereinfacht. Das Framework muss sich dabei möglichst problemlos in die vorhandene Umgebung einfügen lassen und sollte daher in Python 2.7 geschrieben sein.

Wichtig ist ebenfalls eine starke Modularität des Frameworks um Aufgabenteilung sowie Wartung durch verschiedene Abstraktionsebenen und Bereiche zu erleichtern. Benutzern des Frameworks soll es mit möglichst wenig Programmierkenntnissen möglich sein, automatische Abläufe zu erstellen.

Prof. Dr. Krause

TECHNISCHE
HOCHSCHULE
LÜBECK

Fachbereich
Elektrotechnik und Informatik
Dekanat - Prüfungsausschuss

Erklärung zur Bachelorarbeit

Ich versichere, dass ich die Arbeit selbständig, ohne fremde Hilfe verfasst habe.

Bei der Abfassung der Arbeit sind nur die angegebenen Quellen benutzt worden. Wörtlich oder dem Sinne nach entnommene Stellen sind als solche gekennzeichnet.

Ich bin damit einverstanden, dass meine Arbeit veröffentlicht wird, insbesondere dass die Arbeit Dritten zur Einsichtnahme vorgelegt oder Kopien der Arbeit zur Weitergabe an Dritte angefertigt werden.

Lübeck, den 04. Juni 2019

.....

(Unterschrift)

Zusammenfassung der Arbeit / Abstract of Thesis

Fachbereich: **Department:**	Elektrotechnik und Informatik
Studiengang: **University course:**	Informatik / Softwaretechnik
Thema: **Subject:**	Entwurf und Implementierung eines Frameworks zur Automatisierung von Tests webbasierter Anwendungen
Zusammenfassung: **Abstract:**	Händische Tests können gerade bei Webanwendungen schnell sehr umfangreich und aufwendig werden. Diese Arbeit beschäftigt sich mit dem Entwurf und der Erstellung einer skriptbasierten Lösung, welche die Automatisierung von Testabläufen für webbasierte Anwendungen stark vereinfacht. Es wird dafür das vorhandene Framework *Selenium* verwendet.
Verfasser: **Author:**	Jan-Ole Hübner
Betreuender ProfessorIn: **Attending professor:**	Prof. Dr.-Ing. Stefan Krause
WS / SS:	SS 2019

Danksagungen

An dieser Stelle möchte ich allen danken, die diese Arbeit ermöglicht und unterstützt haben. Natürlich gilt das für meinen Betreuer Prof. Dr.-Ing. Stefan Krause, der bei jeder noch so kleinen Frage immer zur Verfügung stand.
Auch Prof. Dr. rer. nat. Andreas Schäfer, meinem Zweitprüfer, sei an dieser Stelle gedankt.

Hauptsächlich gilt mein Dank aber der Plato AG, die mir diese Arbeit ermöglicht und mir bei der Entwicklung viele Freiheiten gelassen hat. Das hat mir nicht nur einen angenehmen Ablauf bei der Bachelorarbeit ermöglicht, sondern auch wertvolle Erfahrungen gegeben.
Im speziellen gilt mein Dank hier Uwe König für die Unterstützung des Projekts und die Ratschläge zum Aufbau der Arbeit und Dennis Schmitz für den regen Austausch während der Entwicklung und die Unterstützung bei der Fehlersuche.
Für das Korrektorat dieser Arbeit danke ich Iris Scheer.

Nicht zuletzt gebührt meinen Eltern und meiner Schwester Jennifer Dank, die mir das Studium erst ermöglichten.

Lübeck, 23.05.2019

Inhaltsverzeichnis

1 Einleitung **1**

1.1 Motivation . . . 1
1.2 Ziele der Arbeit . . . 1
1.3 Gliederung des weiteren Dokumentes . . . 1

2 Anforderungsanalyse **2**

2.1 Problem . . . 2
2.2 Grundlagen und Begriffe . . . 3
2.3 Anforderungen an das Framework . . . 4
2.3.1 Funktionale Anforderungen . . . 4
2.3.2 Nicht funktionale Anforderungen . . . 7
2.4 Anforderungen an die Tester . . . 8
2.4.1 Kern-Entwickler . . . 8
2.4.2 Plugin-Entwickler . . . 9
2.4.3 Tester . . . 9

3 Analyse vorhandener Ansätze und Technologien **10**

3.1 PyAutoGUI . . . 10
3.2 Jubula . . . 10
3.3 Selenium . . . 11

4 Ansatz und Entwurf **12**

4.1 Core . . . 13
4.2 Plugins . . . 14
4.3 Eventuelle Probleme durch die Verwendung von Python 2.7 . . . 14
4.4 Auslieferung des Frameworks . . . 15
4.5 Namensfindung . . . 15

5 Implementierung **16**

5.1 Allgemeiner Ablauf . . . 16
5.2 Betrachtung der einzelnen Schichten . . . 16
5.2.1 Core . . . 16
5.2.2 Plugins . . . 24

5.2.3 Tests 28

6 Test **30**
6.1 Core 30
6.1.1 Fälschlicherweise bestandene Tests 31
6.2 Plugins und Tests 31
6.3 Beurteilung des Aufwandes 32

7 Schlussbetrachtung **33**
7.1 Nutzerfeedback 33
7.2 Zusammenfassung 34
7.2.1 Zeitersparnis 34
7.3 Ausblick 34

Abbildungsverzeichnis **36**

Tabellenverzeichnis **37**

Literatur **38**

Anhang **39**
1 Voraussetzungen für die Nutzung 39
2 SD-Karte 39
2.1 Verwendung der SD-Karte 40
2.2 Die Ordner im Einzelnen 40
3 Die Demo-Applikation 41

1 Einleitung

1.1 Motivation

Ein wichtiger Teil bei der Entwicklung einer Software ist das Testen. Gerade bei Anwendungen, die über den Browser verwendet werden, gewinnen Tests häufig sehr schnell an Komplexität. Tests müssen für jeden zu unterstützenden Browser durchgeführt werden. Da sich diese Tests aber häufig in der Durchführung gar nicht oder nur minimal unterscheiden, findet sich gerade in diesem Punkt viel Potential zur Automatisierung. Dieser Vorgang ist jedoch häufig selbst komplex und erfordert entsprechende Qualifikationen im Bereich der Softwareentwicklung. Viele Unternehmen und Entwicklerteams scheuen deshalb den initialen Arbeitsaufwand, der für einen automatisierten Ablauf der Tests nötig ist, da dieser meist auch schwer einschätzbar ist.

1.2 Ziele der Arbeit

Diese Arbeit soll der Plato AG eine Lösung bieten, die es ermöglicht, den Arbeitsaufwand zum Automatisieren von Testabläufen zu minimieren. Dabei soll die gebotene Lösung so gestaltet sein, dass möglichst wenig Vorkenntnisse des Benutzers vorausgesetzt werden. Dies soll sicherstellen, dass in Entwicklerteams nur eine minimale Einarbeitung notwendig ist, um einfache Abläufe automatisiert umsetzen zu können und so eine effiziente Nutzung vorhandener personeller Ressourcen zu erreichen. Die gebotene Lösung soll dabei zwar auf die Software „Plato e1ns“ [AG19] abgestimmt sein, sollte aber auch auf andere Anwendungen übertragbar sein.

1.3 Gliederung des weiteren Dokumentes

Das Dokument gliedert sich in verschiedene Kapitel, um den Weg zur fertigen Umsetzung oben beschriebener Ziele strukturiert und nachvollziehbar zu erläutern. Die Kapitel folgen grob dem Prozess der Entwicklung einer Software. Behandelt und betrachtet werden neben einer Anforderungsanalyse in Kapitel 2 auch vorhandene Ansätze. Anschließend folgen die wichtigsten Teile dieser Arbeit: Der Entwurf (4) und die Umsetzung der geforderten Lösung (5). Im Anschluss folgen die Tests sowie eine abschließende Betrachtung der Arbeit.

2 Anforderungsanalyse

In diesem Kapitel werden die Anforderungen an die zu entwickelnde Lösung, aber auch an deren Nutzer betrachtet. Außerdem werden grundlegende Begriffe definiert, damit bei deren Verwendung Unklarheiten vermieden werden.

2.1 Problem

Bevor die Entwicklung einer adäquaten Lösung überhaupt möglich ist, bedarf es einer Betrachtung des eigentlichen Problems.
Webanwendungen bieten einem Anwender heute die Möglichkeit mittels eines Browsers Dinge zu erledigen für die es bis vor kurzem nötig war, eigenständige Programme zu installieren. Dazu zählt beispielsweise neben dem Erledigen von Bankgeschäften oder dem Einkaufen auch Tabellenkalkulation. Die Vorteile liegen auf der Hand: Code der für bestimmte Hardware geschrieben werden musste, wird heute durch HTML und Skripte ersetzt, die z.B. in Javascript geschrieben werden. Diese Kombination aus Markup und Skripten wird dann von einem Browser interpretiert und dargestellt. Dieser Vorgang wurde durch die Weiterentwicklung des Internets immer komplexer geworden und es kamen nicht nur neue Technologien wie z.B Cookies und Sessions hinzu, sondern es entwickelten sich mit der Zeit auch neue Browser und Geräteklassen. Während man beim Entwickeln einer Webanwendung zum Unterstützen der verschiedenen Geräte auf Dinge wie Auflösung und Seitenverhältnis achten muss, ist auch eine Optimierung auf verschiedene Browser notwendig. Jeder Browser hat einen leicht unterschiedlichen Funktionsumfang und verhält sich beim Interpretieren der Darstellungsanweisungen unterschiedlich.

Zusammen ergeben allein diese Kombinationen eine Vielzahl an Ausgangssituationen, die bei einem Benutzer gegeben sein können. Will man nun sicherstellen, dass sich die eigene Applikation bei einer bestimmten Untermenge der Kombinationen gleich verhält, muss dies meist aufwändig händisch getestet werden, da die Automatisierung dieser Vorgänge selbst recht zeitaufwendig ist und sich aus der Sicht von Unternehmen und Privatanwendern meistens nicht lohnt.

Die konkrete Vorgehensweise ist häufig das Abarbeiten von Testplänen, die einem menschlichen Tester in natürlicher Sprache erklären, welche Schritte auszuführen sind. Diese werden daraufhin vom jeweiligen Tester ausgeführt und dokumentiert. Dies erfordert jedoch viele personelle Ressourcen und bietet ebenfalls Raum für Fehler, da durch die Verwendung

natürlicher Sprache oft Unklarheiten entstehen.

Um den Aufwand der Testautomatisierung so gering wie möglich zu halten und damit Voraussetzungen zum Automatisieren von Abläufen zu schaffen, muss eine Lösung geschaffen werden, die es ermöglicht Abläufe in verschiedenen Webanwendungen einfach zu automatisieren. Es bietet sich an ein Framework zu entwickeln, dass Funktionalitäten zur Verfügung stellt, die für die Automatisierung verwendet werden können.

2.2 Grundlagen und Begriffe

Bei der Entwicklung eines Frameworks sollte zuerst betrachtet werden, was ein Framework denn überhaupt ausmacht. Die gängigste Definition eines Frameworks lautet wie folgt:

„Eine Klassenbibliothek oder Sammlung von Klassenbibliotheken zur Programmierung bestimmter Anwendungen.“ [Ker15, Seite 1225]

Diese Definition erweist sich jedoch als zu starr, denn Frameworks werden in der Praxis nicht nur zur Programmierung von Anwendungen benutzt. Ein Beispiel hierfür ist ein Testframework wie JUnit. [JUn19] Das Ziel von Testframeworks ist nicht die Erzeugung einer Anwendung, sondern die Verifikation der Funktionalität einer vorhandenen Applikation. Daher liegt dieser Arbeit folgende Definition zu Grunde:

„Als Framework bezeichnen wir hier alles, was mittels systematischer Erweiterung oder Konfiguration angepasst oder erweitert werden kann.“[Sta+07, Seite 127]

Um oben erwähnte Erweiterbarkeit zu erreichen, ist es ebenfalls sinnvoll den Begriff des „Plugin“ zu betrachten.

„Ein Plug-In ist eine Software, die eigenständig eine Teil-Funktionalität abdeckt und damit die Funktionalität eines Gesamtsystems erweitert. Für die Ausführung eines Plug-Ins wird dabei eine übergeordnete Plattform benötigt, die wiederum die benötigten Schnittstellen zur Integration des Plug-Ins zur Verfügung stellt.“ [ITW19]

Des weiteren gilt es zu klären, welcher Teil einer Webanwendung mit der zu entwickelnden Lösung getestet werden soll. Webanwendungen bestehen stark vereinfacht gesagt aus einem oder mehreren Clients und einem oder mehreren Servern. Beide Bestandteile müssen demnach natürlich auch getestet werden. Diese Arbeit beschäftigt sich aber nur mit Tests auf der Seite des Clients. Die Tests auf Seiten des Servers finden separat statt und sollen nicht Gegenstand dieser Arbeit sein. Dies beinhaltet Tests bezüglich Datenbankensystemen oder Sicherheitslücken auf Seiten des Servers. Erkannt werden sollen jedoch beispielsweise logische Fehler, die durch die Auslieferung fehlerhafter Daten entstehen.

2.3 Anforderungen an das Framework

Durch die Erstellung eines Frameworks wird ein Umfeld geschaffen, dass nicht von einer bestimmten Anwendung abhängig ist. Gleichzeitig kann es bei Bedarf für stark spezialisierte Abläufe verwendet werden. „Abläufe“ meint hier die in Abschnitt 2.1 beschriebenen Vorgehensweisen in den Testplänen.
Die konkreten Anforderungen ergeben sich im Falle dieser Arbeit fast ausschließlich aus Gesprächen mit potenziellen zukünftigen Nutzern. Gesammelte Anforderungen werden in 2.2, 2.3 und 2.4 tabellarisch dargestellt und anschließend erläuternd analysiert. Aussagen, die zu diesen Anforderungen getroffen werden, um diese genauer zu spezifizieren, basieren auf einer zweiten Befragung der zukünftigen Nutzer.

		Dringlichkeit	
		dringend	nicht dringend
Wichtigkeit	wichtig	A	B
	nicht wichtig	C	D

Tabelle 2.1: Priorisierungsklassen nach Eisenhower-Prinzip

Es findet außerdem eine Priorisierung der Gesprächsergebnisse mittels der Zwei-Kriterien-Methode wie in [KB18, Seite 216] erläutert statt. Dabei werden die Anforderungen nach dem sogenannten *Eisenhower-Prinzip* in 4 Klassen unterteilt (Siehe Tabelle 2.1). Die verwendeten Kriterien sind in diesem Fall „Dringlichkeit“ und „Wichtigkeit“, wobei „Dringlichkeit“ hier auf den zeitlichen Dringlichkeitsaspekt abzielt und „Wichtigkeit“ die Unverzichtbarkeit für die Funktionsfähigkeit betrachtet.

2.3.1 Funktionale Anforderungen

Skriptbasierte Tests / GUI

Die wohl wichtigste Anforderung an die geforderte Lösung ist, dass Tests in Form von Python-Skripten entstehen sollen. Das geforderte Framework muss also in Form eines importierbaren Python-Moduls vorliegen. Eine grafische Benutzeroberfläche zur Erstellung von Abläufen ist nicht gewünscht.

Browser

Aus der in Tabelle 2.2 erwähnten, geforderten Unterstützung für mehrere Browser folgt eine in Kapitel 4 genauer betrachtete Schicht zur Abstraktion der Kommunikation mit einem Browser. Die Aufgabe des Browsers soll es sein, Bedienungsanweisungen entgegenzunehmen

Titel	Priorisierung	Beschreibung
skriptbasierte Tests	A	Tests werden in Form von Python-Skripten erstellt.
Keine GUI	A	Es wird keine grafische Benutzeroberfläche erstellt.
Browser	B	Google Chrome, Mozilla Firefox, Internet Explorer und Microsoft Edge sollen unterstützt werden.
Schnittstelle	A	Es soll eine Schnittstelle zur Verfügung gestellt und dokumentiert werden, die eine Erweiterung der Funktionalität ermöglicht.
Logging	C	Es erfolgen detaillierte Ausgaben, die nach den Stufen INFO, DEBUG, WARNING und ERROR filterbar sind.
Mehrsprachigkeit	B	Es wird die Möglichkeit geboten, Tests in verschiedenen Sprachen auszuführen

Tabelle 2.2: Beschreibung Funktionaler Anforderungen

und diese auszuführen. Die Überprüfung der korrekten Ausführung liegt nicht in der Verantwortung des Browsers.
„Unterstützung“ meint hier die Ausführbarkeit der geschriebenen Tests, unabhängig vom gewählten Browser.
Um die Ausführbarkeit der Tests sicherstellen zu können, ist es an dieser Stelle notwendig den Begriff des Tests zu definieren. Die Befragung von Mitarbeitern der Plato AG lieferte hier folgende Erwartung: Ein Test ist ein vom Nutzer des Frameworks verfasstes Skript, welches automatisiert Aktionen im Browser ausführt und als Ergebnis „bestanden“ oder „nicht bestanden“ liefert. Die Entscheidung über das Ergebnis wird anhand definierbarer Bedingungen getroffen. Diese Bedingungen sind mit Hilfe von booleschen Ausdrücken und Rückgabewerten des Frameworks vom Autor des Tests innerhalb des Selbigen festlegbar. Eine daraus folgende Funktionalität zur Überprüfung von festlegbaren Bedingungen wird in den Kapiteln 4 und 5 berücksichtigt.

Schnittstelle

Die Forderung nach einer „Schnittstelle zur funktionalen Erweiterung“ zieht bei genauerer Betrachtung einige Komplexität nach sich, da hier die Nachladbarkeit von Code zur Laufzeit gemeint ist. Mit diesem Code soll ein Entwickler den Testern neue Funktionalitäten zur Verfügung stellen können. Diese Codesegmente werden im Folgenden *Plugins* genannt.
Plugins erweitern das Framework in diesem dahingehend, dass durch ein Plugin komnplexe Abläufe, welche durch einen Nutzer der zu testenden Software im Browser ausgeführt werden können, als einfach abrufbare Funktionen zur Verfügung stellt.

So soll vom Tester Beispielsweise die Benutzung einer Navigation nicht jedes mal durch die Aneinanderreihung der Aufrufe für die einzelnen Klicks simuliert werden, sondern es soll ein Plugin eine einzelne, aufrufbare Funktion zur Navigation zur Verfügung stellen.
Weitere Rücksprachen ergaben genauer den Wunsch nach einer Möglichkeit die nachladbaren Codesegmente zu kategorisieren. Es ist also eine Unterteilung in verschiedene Typen gewünscht. Initial sind die Typen „Navigation“ und „Login“ gefordert. Von diesen Typen, wird erwartet, dass sie in jeweiligen Plugins Funktionen zum Starten eines Ablaufs zum Anmelden eines Nutzers bzw. zum Nutzen einer Navigation bereitstellen. Der Ablauf kann so zu einem späteren Zeitpunkt genauer definiert werden.
Plugins sollen hierfür sowohl auf sämtliche Funktionen des Frameworks, als auch auf die Funktionen anderer Plugins zugreifen können. Es soll auch möglich sein, bei Bedarf neue Typen zu definieren. Eine weitere Folge dieser Anforderung ist eine Abhängigkeitenverwaltung um sicherzustellen, dass Plugins zum einen in der richtigen Version zur Verfügung stehen und zum anderen alle Voraussetzungen zur korrekten Ausführung des Plugins gegeben sind, da es mit der Einführung von Plugins möglich wird, neue Plugins auf der Basis von bereits vorhandenen Plugins aufzubauen. Eine Implementierung einer solchen Abhängigkeitsverwaltung ist ebenfalls den in Tabelle 2.3 gelisteten Anforderungen bezüglich der Wartbarkeit zuträglich. Der Sicherheitsaspekt von zur Laufzeit nachgeladenem Code wird in Kapitel 4 genauer betrachtet, ist in diesem Fall aber vermutlich größtenteils zu vernachlässigen, da Tests ohnehin aus Sicht eines Nutzers stattfinden.

Logging

Auch bei der scheinbar simplen Forderung nach konfigurierbaren Fehlerausgaben bedarf es einiger Erläuterung, denn eine Ausgabe kann auf verschiedene Arten erfolgen. Mit Blick auf die technische Anforderung des gewünschten Startens durch *Jenkins* [Kaw11] aus Tabelle 2.4 wird davon ausgegangen, dass die Verwendung der Standardausgabe genügt und keine spezielle Speicherung durch das Framework stattfinden muss.

Mehrsprachigkeit

Wichtig ist zuletzt auch die Unterstützung mehrerer Sprachen. Gemeint ist jedoch keine mehrsprachige Benutzeroberfläche, sondern eine Möglichkeit mit verschiedenen Sprachen in der Benutzeroberfläche der zu testenden Software umzugehen. So soll es möglich sein, einen Test, der beispielsweise die Navigation durch verschiedene Punkte eines Menüs testet, ohne Änderungen am Test mit verschiedenen Spracheinstellungen durchzuführen. Übersetzungen werden, falls benötigt, bereitgestellt. Eine eigenständige Übersetzung von Zeichenketten ist ausdrücklich nicht notwendig.

2.3.2 Nicht funktionale Anforderungen

Aus den geführten Gesprächen ergaben sich auch folgende nicht funktionale Anforderungen.

Titel	**Priorisierung**	**Beschreibung**
Benutzersicht	A	Tests finden, wann immer möglich, aus der Sicht eines Nutzers statt.
Wartung 1	B	Der Wartungsaufwand soll möglichst gering sein.
Wartung 2	B	Die Wartung und Erweiterung soll von mehreren Personen und nach Möglichkeit gleichzeitig geschehen können.
Fehlerquote	B	In produktiv eingesetzten Tests sollte die Fehlerquote maximal 20% betragen

Tabelle 2.3: Beschreibung nicht funktionaler Anforderungen

Die Konsequenzen, die sich aus den Anforderungen in Tabelle 2.3 ergeben, sind vor der eigentlichen Planung des Systems schwer abzuschätzen, da sie nur sehr vage formuliert sind. Die Benutzersicht kann gerade im Kontext von Webanwendungen verschiedene Bedeutungen haben. So kann damit von der direkten Interpretation der Antworten des Servers, oder der Formulierung eigener Anfragen die Rede sein. Auch die Simulation der Steuerung von Programmen, durch einen Benutzer wäre eine mögliche Definition. Durch die funktionale Anforderung der Unterstützung verschiedener Browser und der in Kapitel 2.2 festgelegten Einschränkung auf Tests der Client-Seite, ist in diesem Fall klar, welche der Möglichkeiten gewünscht ist. Trotzdem sollte an dieser Stelle definiert werden, wann diese Einschränkung überschritten werden darf oder sogar sollte.
Muss ein *Plugin* beispielsweise sicherstellen, dass eingegebene Daten korrekt verarbeitet wurden, kann dies meist nicht aus Sicht eines Nutzers geschehen. Das Framework muss sich also Daten, die zur Validierung von Funktionen oder zum Sicherstellen einer korrekten Ausführbarkeit des Test benötigt werden, aus anderen Quellen holen, die einem Benutzer normalerweise nicht zur Verfügung stehen. Dies soll auch dann gelten, wenn die benötigten Informationen einem Benutzer zur Verfügung stehen, aber die Beschaffung die Fehleranfälligkeit oder Geschwindigkeit beeinträchtigt.

Unter „Wartbarkeit“ wird im Kontext dieser Arbeit die Anpassbarkeit des Frameworks und den damit produzierten Testabläufen an neue Versionen der zu testenden Software verstanden. Die daraus resultierende, gewünschte Modularität wird in Kapitel 4 genauer betrachtet.

Zum Erfassen einer Fehlerquote müssen zunächst Fehler definiert werden. Dieser definiert sich in diesem Fall wie folgt: Ein Fehler ist das endgültige, irrtümliche Nicht-Bestehen eines durch einen Nutzer des Frameworks formulierten Testfalls, das durch fehlerhafte Abläufe oder Interpretationen zustande kommt, die nicht im Verantwortungsbereich des Nutzers liegen.

Titel	**Priorisierung**	**Beschreibung**
Python 2.7	A	Das Framework soll mit Python 2.7 geschrieben sein
Modul	A	Installierbar in einer neuen Pythonumgebung als Modul
Jenkins	D	Tests sollen bei Bedarf automatisiert durch *Jenkins* [Kaw11] gestartet werden können
Windows	D	Lauffähigkeit unter Windows

Tabelle 2.4: Beschreibung technischer Anforderungen

Die Konsequenzen des geforderten Python 2.7 werden in Kapitel 4.3 genauer betrachtet. Die restlichen technischen Anforderungen haben keine absehbaren Konsequenzen. *Jenkins* ermöglicht das automatische Starten von Prozessen vor oder nach einem Build. Hier bedarf es keiner speziellen Anpassung des Frameworks. Die Lauffähigkeit unter Windows sollte durch die Verwendung von Python automatisch gegeben sein. Hier ist lediglich darauf zu achten, keine Module zu verwenden, die Windows explizit nicht unterstützen.
Die Auslieferung als eigenständiges Modul erfordert eine starke Kapselung der zu entwickelnden Lösung

2.4 Anforderungen an die Tester

Bei der Umsetzung dieses Projekts sollten nicht nur Anforderungen an die Software definiert werden. Da es sich hierbei nicht um eine gewöhnliche Software mit grafischer Benutzeroberfläche handelt, sondern um ein Framework zur Entwicklung eigener Testabläufe, setzt dies bestimmte Kenntnisse des jeweiligen Benutzers voraus. Hierbei werden drei unterschiedliche Arten von Benutzern unterschieden, an die unterschiedliche Anforderungen gestellt werden.

2.4.1 Kern-Entwickler

Bei diesem Benutzer handelt es sich um einen Entwickler der den Kern des Frameworks wartet und erweitert. Er definiert bei Bedarf neue Schnittstellen und kann die vorhandenen verändern. Hierfür sind fundierte Kenntnisse der Programmiersprache Python notwendig. Auch andere Frameworks oder Sprachen, die im weiteren Verlauf der Entwicklung eventuell Verwendung finden, sollten beherrscht oder selbstständig erlernt werden können.

2.4.2 Plugin-Entwickler

Dieser Benutzer erweitert die Funktionen des Frameworks mit Hilfe der Definition neuer Plugins. Er verwendet die Schnittstellen des Frameworks, erweitert oder verändert diese jedoch nicht. Auch dieser Benutzer sollte über fortgeschrittene Kenntnisse in der Programmiersprache Python verfügen. Kenntnisse in den Bereichen, um die das Framework erweitert werden soll, müssen natürlich ebenso vorhanden sein.

2.4.3 Tester

Dieser Benutzer verwendet das Framework ausschließlich, um Testabläufe in einer gegebenen Anwendung zu automatisieren. Er erweitert die Funktion in keiner Weise und beschäftigt sich ebenfalls nicht mit der Wartung. Von diesem Benutzer werden daher die geringsten Vorkenntnisse erwartet.
Dennoch sollten auch Tester in der Lage sein, einfache Python-Skripts zu schreiben. Kenntnisse der Grundbegriffe objektorientierter Programmierung sind hilfreich, um Fehlermeldungen und die Benutzung der zur Verfügung gestellten Schnittstellen nachvollziehen zu können. Ebenfalls sollten die Grundbegriffe einer Webanwendung bekannt sein. Es kann z.B. nötig sein, Elemente der Webanwendung mittels eines selbst formulierten XPaths zu lokalisieren. Grundsätzlich gilt an dieser Stelle: Je komplexer der zu automatisierende Ablauf, desto fundierter sollten die Kenntnisse sein.

3 Analyse vorhandener Ansätze und Technologien

Dieses Kapitel betrachtet bereits existierende Lösungsansätze, um herauszufinden wo deren Stärken und Schwächen liegen und diese in die Planung mit einzubeziehen. Softwarelösungen die rein kommerzieller Natur sind, ebenso wie Lösungen in Form von Dienstleistungen werden hier nicht betrachtet.

3.1 PyAutoGUI

PyAutoGui [Swe18] ist ein Python-Modul, dass plattformunabhängig einfache Funktionen zum Steuern der Maus und zum Tätigen von Eingaben auf der Tastatur ermöglicht. Ein Vorteil dieser Lösung ist, dass sie neben bereits erwähnten Funktionen auch eine Bilderkennung bietet. So können z.B. mittels eines einzigen Aufrufes die Koordinaten eines gegebenen Bildes auf dem Bildschirm gefunden werden.

Da das Modul außerdem nicht direkt mit einem Browser interagiert, können auch Anwendungen außerhalb des Webkontextes getestet werden. Jedoch beginnen an diesem Punkt auch die Schwächen des Moduls. Es fehlen Möglichkeiten zur Fehlererkennung und spezielle Funktionen für den Webkontext. Das Modul ist zwar bewusst minimal gehalten, damit keine große Komplexität entsteht, aber das zieht bei der Verwendung für webbasierte Anwendungen viel Arbeitsaufwand nach sich. Daher scheint diese Lösung ungeeignet.

3.2 Jubula

Jubula [Fou13] ist weniger ein Framework als vielmehr ein eigenständiges Werkzeug zum Erstellen automatisierter Abläufe. Es ist ebenfalls als Erweiterung für die Entwicklungsumgebung Eclipse verfügbar. Jubula enthält eine Testumgebung, die es ermöglicht, Tests zunächst weitestgehend unabhängig vom eigentlichen Code zu erstellen. Dies passiert per Drag&Drop anhand einer Anwendungsvorlage. So wird erreicht, dass nur wenig Programmierkenntnisse nötig sind.

3.3 Selenium

Selenium [Pro18] ist die Lösung, die der in dieser Arbeit geforderten Lösung am nächsten kommt. Es ist ein Framework zum automatisierten Testen von Webanwendungen. Diese Tests werden in Form eines Skriptes z.B. als Unittest oder Nosetest [Pel17] erstellt. Selenium kümmert sich hierbei hauptsächlich um die Kommunikation mit den verschiedenen Browsern. Es stellt ein sogenanntes Webdriver-Objekt zur Verfügung, das zur Ausführung von grundlegenden Aktion wie dem Selektieren von Elementen und dem Ausführen von Klicks genutzt werden kann. Hier liegt allerdings auch die größte Schwäche von Selenium: Tests, die mit Selenium erstellt werden, sind sehr schnell unflexibel. Die Tests folgen einem streng vorgegebenen Ablauf. Auf unvorhergesehene Änderungen im Ablauf oder der Anwendung, wie z.B. längere Ladezeiten aufgrund von hoher Serverlast, kann selbst mit Fehlerbehandlungen nur schwer reagiert werden. Sind solche Reaktionen gewünscht, steigt die Komplexität des Tests sehr schnell stark an. Dennoch bietet Selenium eine gute Basis für automatisierte Tests, wenn man bereit ist, Komplexität in Kauf zu nehmen. Des weiteren sei erwähnt, dass Selenium die einzige der betrachteten Lösungen ist, deren Entwicklung und Wartung andauert, ohne dass eine Bezahlung oder Registrierung gefordert wird.

4 Ansatz und Entwurf

Unter Berücksichtigung der Betrachtungen aus Abschnitt 3.3 und der für diese Arbeit gegebenen Frist, bietet es sich an, auf Selenium aufzubauen und die komplexe Aufgabe der Kommunikation mit den verschiedenen Browsern Selenium zu überlassen. Die so gesparte Zeit kann verwendet werden, um sich bei der Entwicklung des Frameworks auf die Modularität und das Verbergen der Komplexität von Selenium zu konzentrieren. Gleichzeitig ist so sichergestellt, dass der Funktionsumfang nicht eingeschränkt wird.

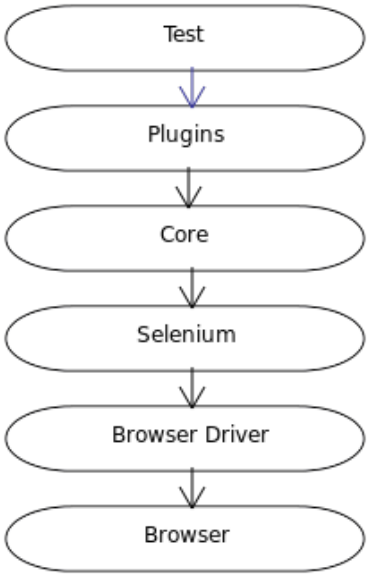

Abbildung 4.1: Schichtung des Frameworks

Aus diesen Überlegungen ergibt sich als erster Entwurf ein Modell mit mehreren Schichten wie in Abbildung 4.1 zu sehen ist. Diese erste Unterteilung in Schichten stellt sicher, dass die Bestandteile des Frameworks innerhalb ihrer Schichten flexibel bleiben und erleichtert die Trennung der in Kapitel 2.4 erwähnten Aufgabenbereiche. Ein weiterer Vorteil der Unterteilung in Schichten ist, dass innerhalb der Schichten unterschiedliche Entwurfsmuster verwendet werden können.

Im Folgenden werden nun die Schichten *Core* sowie *Plugins* kurz einzeln betrachtet. Die Schicht 'Test' hat in der Planung keine Priorität, auch wenn sie entscheidend für den Erfolg

des Frameworks ist. Der Grund dafür ist, dass sie von den darunter liegenden Schichten geformt wird und sich die dort getroffenen Entscheidungen direkt in ihr widerspiegeln.

4.1 Core

Diese Schicht ist das Herzstück des Frameworks. Sie hält Informationen wie die Version der zu testenden Software, die geladenen Plugins sowie die gewählte Konfiguration des Testers und kümmert sich um die Fehlerbehandlungen. Hier treffen sämtliche Schnittstellen aufeinander und müssen miteinander koordiniert werden.

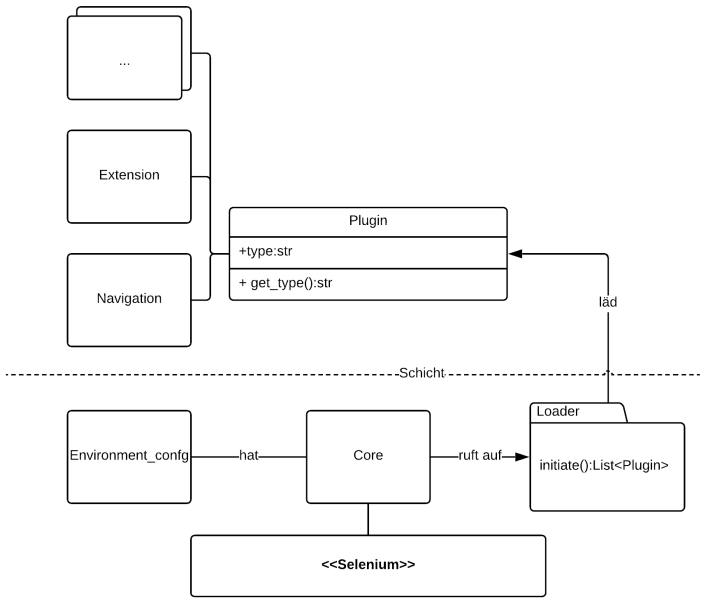

Abbildung 4.2: Pseudo-Klassendiagramm der untersten Schicht

Ein erster, grober Entwurf ist in Abbildung 4.2 zu sehen. Trotz der noch nicht vorhandenen Details wird bereits die zentrale Rolle der Klasse *Core* sichtbar. Sie übernimmt die Rolle eines Vermittlers, wodurch sie sich in diesem Entwurf monolithisch gestaltet. Monolithische Klassen werden in der Softwareentwicklung heutzutage kritisch betrachtet.

> Es besteht die Gefahr zu 'überzentralisieren', also statt 'wendigen', dezentralen Systemen monolithische, zentral organisierte und gesteuerte Systeme zu bauen.
> Der Vermittler kann ein weiterer Single Point of Failure sein. Das ist ein wenig widersprüchlich. Denn einerseits können zentral vermittelte Systeme robuster, weil einfacher und deterministischer sein; andererseits läuft gar nichts mehr, wenn das Vermittlerobjekt eine unbehandelte Exception auslöst.[Gei15, Seite 290]

Diese Nachteile sind in diesem Fall jedoch eher als Vorteile zu betrachten. Zum einen wird dem Benutzer des Frameworks ein zentraler Punkt für dessen Verwendung geboten, was

die Benutzung erheblich vereinfacht. Zum anderen begrenzt sich der Single Point of Failure lediglich auf eine Schicht, was einen entstehenden Fehler leicht auffindbar macht.

4.2 Plugins

Die Plugins dienen der Erweiterung der Grundfunktionalität. Sie werden bei der Initialisierung jedes Test geladen und enthalten Funktionen für komplexere Abläufe, spezifisch für die jeweils zu testende Software. Sie haben sowohl Zugriff auf die im *Core* gespeicherten Information, als auch auf die vom *Core* bereitgestellte Schnittstelle zu Selenium. Dadurch können sie sämtliche Funktionalitäten verwenden. Der *Core* stellt vordefinierte Typen zur Erstellung von Plugins zur Verfügung. Jeder dieser Typen kann dabei eigene Anforderungen an den Entwickler eines Plugins in Hinsicht auf zu implementierende Methoden stellen. Die Typen Login und Navigation ergeben sich hierbei aus den direkten Anforderungen. Der Typ Verification kommt zustande, da sich aus der Analyse der Anforderungen die Notwendigkeit ergibt, Bedingungen zu definieren und diese zu verifizieren. Für diesen Typ bietet es sich an, einen booleschen Wert als Rückgabe festzulegen, da es bei der Verifizierung von Bedingungen lediglich zu den Ergebnissen „trifft zu“ oder „trifft nicht zu“ kommen kann. Ein Plugin vom Typ „Navigation“ könnte z.B. festlegen, dass eine „navigate“ Methode innerhalb des Plugins erwartet wird, ähnlich eines Interfaces in Java. In puncto Architektur stellen die Plugins jedoch keine große Herausforderung dar. Sie benötigen lediglich einen Satz Metadaten, die beim Laden durch den *Core* interpretiert werden können. Diese Metadaten schließen zum Beispiel eine Versionsnummer sowie Abhängigkeiten mit ein.
Das Definieren unterschiedlicher Typen von Plugins mit unterschiedlichen Vorgaben erleichtert es dem Plugin-Entwickler, neue Plugins fehlerfrei zu erstellen, da ihm mit Hilfe vorgegebener Schnittstellen u.a. die Aufgabe der korrekten Initialisierung abgenommen werden kann. Plugins werden dadurch deutlich überschaubarer, da eine einfache Vererbung angewandt werden kann.

4.3 Eventuelle Probleme durch die Verwendung von Python 2.7

Die Aufgabenstellung schreibt die Verwendung von Python 2.7 vor. Python 2.7 erreicht Ende des Jahres 2019 den Status „End of Life“ und wird ab dann nicht mehr weiter entwickelt. Abgesehen von dem daraus resultierenden Sicherheitsrisiko aufgrund fehlender Updates, sind nur wenige, kleinere Probleme zu erwarten. Die meisten bekannten Module unterstützen Python 2.7 noch und es gibt in Python 3 nur wenige Neuerungen, die bei der Umsetzung dieses Projekts hilfreich sein könnten. Auch Performanceverbessurgen sind vermutlich hinfällig, da beim Testen einer Webanwendung viel Zeit damit verbracht wird, auf den Aufbau der Seiten zu warten. Vermutlich wird es jedoch erforderlich sein, Klassen zur Laufzeit zu laden. Hier sind Probleme bei der Kompatibilität zu Python 3 zu erwarten.

4.4 Auslieferung des Frameworks

Nach der groben Betrachtung beider Schichten stellt sich die Frage, wie das Framework möglichst einfach ausgeliefert werden kann. Möglich wäre hier zwar die Verwendung einer Versionsverwaltung, da aber nicht alle Nutzer an dem Framework entwickeln, wäre dies nicht optimal.
Tabelle 2.4 fordert die Auslieferung als Modul. Python bietet hier eine Lösung, die perfekt zu der geschichteten Struktur passt. Es handelt sich dabei um ein sogenanntes „*Wheel*“. Diese *Wheels* sind gepackte Versionen eines Python Moduls, die von dem Paketmanager „*Pip*“ installiert und verwaltet werden können.

Tatsächlich sind *Wheels* lediglich zip-komprimierte Dateien mit der Endung WHL, die ein Python-Modul enthalten, dass der vorhandenen Python Installation hinzugefügt werden soll. Dies hat den Vorteil, dass mehrere Versionen des Frameworks durch die Erstellung einer weiteren Python Umgebung parallel betrieben werden können. Dieser Umstand könnte sich im professionellen Umfeld als nützlich erweisen, wenn es z.B. um die fortlaufende Unterstützung älterer Versionen einer Software geht.
Das Framework kann sich außerdem schneller weiterentwickeln, da durch die Möglichkeit eines parallelen Betriebs mehrerer Versionen der Faktor der Abwärtskompatibilität an Wichtigkeit verliert. Alte Schnittstellen können bei Bedarf so schneller entfernt werden, was die Übersichtlichkeit erhöht. Zudem kann beim Auftreten eines Fehlers kann außerdem schnell auf eine andere Version gewechselt werden.

Ebenfalls wird durch die Verwendung von „*Pip*“ die Installation aller Abhängigkeiten sichergestellt. Um ein Modul als *Wheel* packen zu können, muss eine Datei mit Metadaten vorhanden sein. Diese Metadaten enthalten neben dem Namen des Moduls und des Autors auch eine Liste der Abhängigkeiten. Im weiteren Verlauf der Entwicklung wird weiter darauf eingegangen. Um der oben erwähnten Anforderung gerecht zu werden, wird der *Core* als *Wheel* gepackt und ausgeliefert. Die Plugins sind im *Wheel* nicht enthalten. So können die Plugins gesondert Updates erhalten, oder in speziellen Konfigurationen vorgehalten werden.

4.5 Namensfindung

Die Auslieferung als *Wheel* macht es erforderlich, das Framework zu benennen. Da das Framework im Rahmen dieser Arbeit bei der Plato AG entsteht, wird der Name nach Rücksprache mit der Firma gewählt. Es wird einer Konvention gefolgt, nach der Produkte, oder auch Räume nach griechischen Philosophen benannt werden.
Das Framework wird somit nach dem Philosophen *Pyrrhon von Elis [Wik19]* benannt. Der offizielle Name lautet „**Plato Pyrrhon**“ oder kurz „**Pyrrhon**“.

5 Implementierung

5.1 Allgemeiner Ablauf

Der allgemeine Ablauf der Implementierung gestaltet sich recht einfach. Da eine Schichtung wie in Abbildung 4.1 vorgesehen ist, liegt es nahe, diese Schichten von unten nach oben aufzubauen. Innerhalb dieser Schichten findet dann der Aufbau anhand vorhandener Testpläne der Plato AG für deren Software „Plato e1ns“ statt [AG19], um e1ns spezifische Abläufe schneller erkennen zu können und so möglichst schnell einen Nutzen aus dem Framework ziehen zu können. Der in dieser Arbeit umgesetzte Teil beschäftigt sich dabei aus zeitlichen Gründen hauptsächlich mit Grundfunktionalitäten und dem Modul „e1ns.actions“ der Software.

5.2 Betrachtung der einzelnen Schichten

5.2.1 Core

Der *Core* übernimmt in der Realität deutlich mehr Aufgaben als in Abbildung 4.2 vereinfacht dargestellt ist. Daher werden die Teilaufgaben, die durch die Analyse der Anforderungen offensichtlich geworden sind, nacheinander erläutert.

Abstraktion von Selenium und Konfiguration

Die Hauptaufgabe des Kerns ist das Abstrahieren von Selenium, so dass Kenntnisse von Selenium zum Automatisieren von Abläufen nicht unbedingt erforderlich sind. Selenium arbeitet nach einem ähnlichen Prinzip wie *Pyrrhon*. Um einen Test durchzuführen erzeugt man ein *Webdriver-Objekt*, dass sich um die Kommunikation mit den Schnittstellen der Browser kümmert. Dies sind kleine Programme, die sich um die Ausführung der Aktionen kümmern. Sie werden auch „Browser-Driver“ genannt. Die Browser-Driver öffnen einen Server, der auf lokale Befehle hört und letztendlich den jeweiligen Browser steuert. Die Browser-Driver stehen im Internet frei zur Verfügung.
Durch das Webdriver-Objekt werden also innerhalb des Tests Aktionen ausgeführt.

```
from selenium import webdriver
driver = webdriver.Firefox()
driver.get("https://jan-ole.de")
assert "Jan" in driver.title
```

Abbildung 5.1: Ein einfacher Test in Selenium

In Abbildung 5.1 wird das erste Problem deutlich: Für jeden Browser gibt es einen eigenen Aufruf. Dies erklärt sich dadurch, dass jeder Browser unterschiedliche Einstellungsmöglichkeiten bietet und das Framework zuerst die Konfiguration der Browser festlegen muss.

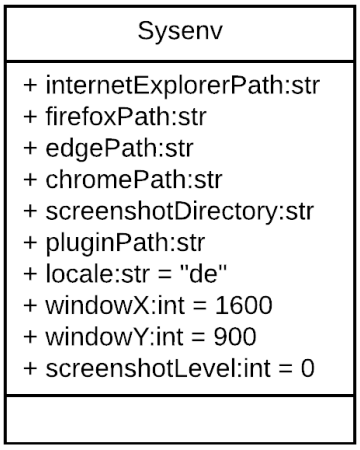

Abbildung 5.2: UML-Darstellung der Klasse Sysenv

Um die gewünschte Konfiguration der Umgebung einfach übergeben zu können, verwendet *Pyrrhon* Die in Abbildung 5.2 dargestellte Klasse. Diese hält alle durch den Benutzer festlegbaren Werte. Die Werte enthalten nicht nur die Pfade der jeweiligen Browser, sondern auch die gewünschte Fenstergröße, den Ort der zu verwendeten Plugins, sowie einige Werte zum Anfertigen von Screenshots. Screenshots sind, obwohl nicht gefordert, eine Erweiterung des Loggings, die keinen großen Entwicklungsaufwand nach sich zieht, da Selenium diese Funktionalität zur Verfügung stellt. Die Screenshots bieten darüber hinaus eine Möglichkeit zur Diagnose von Fehlern, unabhängig davon, ob dieser Fehler seinen Ursprung im Framework oder der zu testenden Software hat. Mit dem Wert **screenshotLevel** bietet *Pyrrhon* hierbei eine Möglichkeit zur Einstufung der Wichtigkeit. Mit diesem Wert kann festgelegt werden, dass Screenshots z.B. nur angefertigt werden, wenn sie in einem bestimmten Plugin erzeugt werden. Es gilt: Screenshots werden angefertigt, wenn der übergebene Wichtigkeitswert des Aufrufs größer oder gleich des konfigurierten Wertes ist.

Ein Vorteil dieser Konfigurationsklasse ist, dass die Quelle der Konfiguration flexibel ge-

halten wird. Die Standardlösung ist eine YAML Datei, die sich im selben Verzeichnis wie die auszuführenden Tests befindet. YAML erlaubt es die benötigte Konfiguration zu deserialisieren. Im Folgenden ist eine beispielhafte Konfiguration dargestellt. (Abbildung)5.3)

```
Sys:
   Driver:
      IE: 'C:\development\Browser\IE\IEDriverServer.exe'
      FF: 'C:\development\Browser\FF\geckodriver.exe'
      Chrome: 'C:\development\Browser\Chrome\chromedriver.exe'
      Edge: 'C:\development\Browser\Edge\MicrosoftWebDriver.exe'
   Plugins: 'C:\development\svn\201904_ss_actions_and_core\pyrrhon_plugins'
   Screenshotdir: 'C:\development\Screenshots'
   Screenshots: 1
   windowX: 1600
   windowY: 900
E1ns:
   Url: 'https://vm-qs-t3.intern.plato-ag.com/e1ns'
   Browser: 'Chrome'
   Locale: 'de'
```

Abbildung 5.3: Bespielkonfiguration

Das Beispiel verdeutlicht, dass es eine weitere Kategorie existiert. Sie enthält Werte, die nichts mit der lokalen Umgebung zu tun haben, für die Ausführung des Tests aber dennoch essentiell sind. Hierzu zählt zum Beispiel die URL der zu testenden Softwareinstallation.

Mit diesen Werten lässt sich ein Test initialisieren, der ein *Webdriver-Objekt* korrekt konfiguriert. Dadurch findet allerdings noch keine Abstraktion statt, da der Core stellt nun lediglich ein *Webdriver-Objekt* zur Verfügung stellt. Dieses kann nun aber im weiteren Verlauf verwendet werden, um eine Abstraktion von Selenium zu erreichen.
Dieser Vorgang wird im Folgenden Abschnitt beispielhaft am Wechsel von Frames gezeigt. Bei der Verwendung von HTML kann durch Frames eine Seite in Teilbereiche unterteilt werden. Ein Frame grenzt Bereiche der Seite in Bezug auf Element-IDs vollständig ab. Wenn man also mit mehreren Elementen der Seite interagieren will, die sich in unterschiedlichen Frames befinden, wird dazwischen ein Kontextwechsel notwendig. Selenium stellt dafür die Funktion **webdriver.switch.to.frame(frame)** zur Verfügung. Fehler werden durch diese Funktion aber nicht behandelt. Versucht ein Benutzer also einen Kontextwechsel in einen nicht vorhandenen Frame, schlägt der Test sofort fehl. In Plato e1ns gab es in Bezug auf dieses Problem zum Zeitpunkt dieser Arbeit zusätzlich inkonsistentes Verhalten in der Software. In Abbildung 5.4 ist die Struktur der Anwendung in Bezug auf die Frames dargestellt. Diese Struktur findet allerdings nicht in allen Modulen der Software Anwendung. Das hat zur Folge, dass es leicht zu Fehlern beim Wechseln der Frames kommen kann. Befindet man sich

innerhalb eines Frames, muss man beispielsweise in das übergeordnete Frameset wechseln, bevor ein benachbarter Frame überhsupt sichtbar ist.

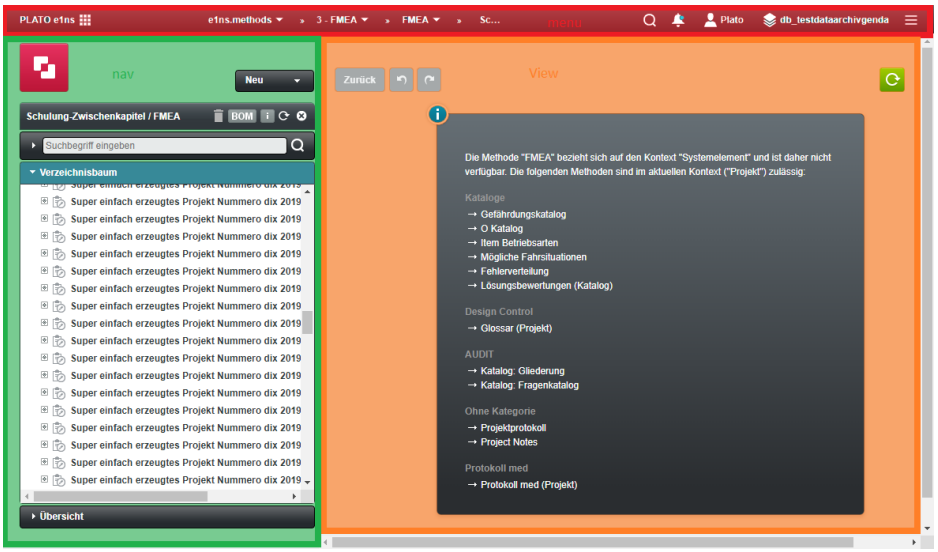

Abbildung 5.4: Frames in Plato e1ns

Wechselt man in nicht vorhandene oder nicht sichtbare Frames, schlägt der Test fehl. Das heißt, jeder Tester müsste zu jedem Zeitpunkt des Tests wissen, in welchem Frame er sich gerade befindet. Um die Komplexität dieses Sachverhalts zu reduzieren, stellt das Framework einem Tester an dieser Stelle eine Funktion zur Verfügung, die eine Fehlerbehandlung beinhaltet. Diese ist in Abbildung 5.5 dargestellt.

```
def switch_frame_if_present(self, name, by=By.NAME):
    self.screenshot_if_set("switch_frame_if_present" + name + "_before", 2)
    if not name == "default":
        self.wait_page()
        if self.is_element_present(By.XPATH, ".//frameset"):
            frame = self.driver.find_element(by, name)
            self.driver.switch_to.frame(frame)
            Core.logger.debug("Switched to Frame " + name)
            self.screenshot_if_set("switch_frame_if_present" + name + "_after", 2)
            return True
        else:
            Core.logger.debug("No Frameset present. Didn't switch to:" + name)
            self.screenshot_if_set("switch_frame_if_present" + name + "_after", 2)
            return False
    elif name == "default":
        Core.logger.debug("Switched to default Frame")
        self.driver.switch_to.default_content()
```

Abbildung 5.5: Konkrete Implementierung eines Wechsels des Frames

Ein Benutzer muss mit dieser Implementierung zwar immer noch wissen, in welchem Frame sich ein Objekt befindet, aber es wird dafür gesorgt, dass der Test nicht fehlschlägt, sollte sich der Tester in einem Bereich der Software befinden in dem beispielsweise durch Inkonsistenz keine Frames vorhanden sind. Auch wird dem Tester der Schritt abgenommen, zuerst in dem sogenannten *Default-Frame* zu wechseln, der das gesammte Frameset enthält. In einer ähnlichen Weise wird mit allen, von Selenium benötigten Funktionalitäten verfahren.
Zusätzlich stellt der *Core* dem Benutzer das Webdriver-Objekt von Selenium zur Verfügung, damit dieser bei Bedarf direkt auf Selenium zugreifen kann.

Bereitstellung von Plugins

Die Bereitstellung von Plugins zur Erweiterung der Funktionalität stellt die größte Herausforderung aus Sicht der Implementierung dar. Dafür gibt es verschiedene Gründe. Das Laden von Erweiterungen kann auf mehrere Arten geschehen. Es kann eine Schnittstelle mit vordefinierten Befehlen zur Verfügung gestellt werden oder Programmcode zur Laufzeit nachgeladen werden. Beide Vorgehensweisen haben sowohl Vor- als auch Nachteile. Vordefinierte Schnittstellen schränken die Funktionalität von Erweiterungen sehr stark ein, bieten dadurch aber Sicherheit, denn der Entwickler behält die Kontrolle über den ausgeführten Code. Das direkte Nachladen von Programmcode ist hingegen ein großes Sicherheitsrisiko, denn hier bietet sich eine Angriffsfläche. Angreifern wäre es ein Leichtes Schadcode auf dem Zielsystem auszuführen. Gleichzeitig fallen Einschränkungen bei der Erweiterbarkeit weg.

Pyrrhon verfolgt aufgrund der Anforderungen an die Software den letzten der Beiden

Ansätze. Python 2.7 bietet zum Nachladen von Code eine einfache Möglichkeit. Abbildung

```
import imp
loadedClass = imp.load_source(className, filename)
```

Abbildung 5.6: Laden von Klassen zur Laufzeit in Python 2.7

5.6 zeigt, wie einfach das Laden einer Klasse ist. Nach Ausführung dieses Befehls enthält *loadedClass* die Klasse. Um Objekte dieser Klasse zu erzeugen, muss nun nur noch der Konstruktor dieser Klasse aufgerufen werden. Der Sicherheitsaspekt ist bei dieser Vorgehensweise natürlich ein Nachteil, der in diesem speziellen Fall aber weniger schwer wiegt, da durch die Art der Nutzung des Frameworks eine Grundsicherheit gegeben ist. Ein Nutzer des Frameworks hat selten einen Grund, Code von externen Entwicklern als Plugin zu laden. Code der zum Testen von Software in einer Firma entsteht, wird dieser nur in den seltensten Fällen auch selbst schaden und bleibt meist innerhalb der Firma. Privatpersonen, die mit höherer Wahrscheinlichkeit externen Code verwenden, so wie natürlich auch Firmen, bietet sich als Gegenmaßnahme natürlich aber auch das Lesen eines Plugins vor dessen Ausführung. Dennoch besteht diese Angriffsmöglichkeit und sollte bei der Verwendung des Frameworks beachtet werden.
Aus dem in Abbildung 5.6 gezeigten Vorgehen ergeben sich noch weitere Konsequenzen. Der Name der zu ladenden Klasse muss bekannt sein. Es stellt sich das Problem, dass Erweiterungen dem System nicht im Voraus bekannt sind.
Dies wird im Framework durch eine Vorgabe bei der Erstellung von Plugins gelöst:
Geladen wird immer die Klasse, die genauso heißt, wie die Python-Datei des Plugins. Nennt ein Entwickler die Datei also beispielsweise „*Login.py*“, so wird versucht, eine Klasse mit dem Namen „*Login*“ aus eben dieser Datei zu laden. Daraus wiederum ergibt sich die Einschränkung, dass nur eine Klasse pro Plugin geladen werden kann. Dies wird im weiteren Verlauf bei der Implementierung der verschiedenen Typen von Plugins berücksichtigt. Ein Entwickler sollte dies jedoch beim Entwurf eines Plugins ebenfalls beachten.
In *Pyrrhon* gibt es 4 unterschiedliche Typen von Plugins: Extension, Login, Navigation, Verification. Plugins erben von diesen Klassen, damit einerseits ein Initialisierungsvorgang möglich wird und andererseits Anforderungen an die unterschiedlichen Plugin-Typen gestellt werden können. So erwartet der *Core* bei der Benutzung eines Plugins vom Typ Navigation z.B. das Vorhandensein einer Methode „*navigate()*“. Die Ausnahme bildet hier der Typ „*Extension*“. Dieser erwartet keine bestimmten Methoden und ist dazu gedacht, alles zu übernehmen, das nicht zu den anderen Typen passt. Die Definition eines Plugin-Typen ist in 5.7 zu sehen. Es ist nicht nur möglich, vorhandene Typen anzupassen, sondern auch gänzlich neue zu definieren. Der abgebildete Typ stellt die simpelste Form der Definition eines Plugin-Typen dar. Wichtig ist hier das Erben von der Klasse Plugin. Dies ermöglicht der Klasse Plugin die geladenen Plugins zu koodinieren, da alle Plugins von der Klasse Plugin

```
from ..Plugin import Plugin
class Custom(Plugin):

    @staticmethod
    def gettype():
        return "custom"

    def __init__(self):
        self.type = "custom"
```

Abbildung 5.7: Definition eines Plugin-Typen

erben. Die Flexibilität der Typendefinitionen wird auf diese Weise nicht eingeschränkt.
Es wäre möglich gewesen, das gleiche Verhalten mithilfe von Metaklassen zu implementieren. Für die geforderten Funktionalitäten hätte dies keine Vorteile gehabt und daher wurde entschieden, die Umsetzung auf Basis von Vererbung durchzuführen.
Eine Metaklasse ist eine Klasse deren Instanzen Klassen sind. Dies zöge unnötige Komplexität nach sich. Die Klasse Plugin besitzt eine Liste aller geladenen Plugin-Klassen, so kann sie verhindern, dass Plugins doppelt initialisiert werden und die in den Metadaten definierten Voraussetzungen gegeben sind. Dazu zählt z.B. ob der Test mit einer passenden Version von Plato E1ns ausgeführt wird. Die Klasse Plugin überprüft jedoch nicht Abhängigkeiten der Plugins untereinander oder Abhängigkeiten von Dateien. Das übernimmt aus zwei Gründen ein Loader, der vom Core beim starten eines Tests aufgerufen wird.
Zum einen liegt dies einfach in der zwingend notwendigen Übersichtlichkeit begründet, die ohne getrennte Abhängigkeitenverwaltung nicht gegeben wäre. Zum anderen ermöglicht die Auslagerung in einen eigenen Loader auch den Umgang mit Dateien die kein Python Code enthalten. Darunter fällt z.B. Javascript. Javascript kann so von *Pyrrhon* registriert und verwendet werden. Die Verwendbarkeit von Javascript, ermöglicht z.B. das erweitertete Verifizieren zurückgegebener Daten, da diese mittels Ausführung eines Skriptes angefordert und gegengeprüft werden können. Auch der Umgang mit z.B. dem asynchronen Laden großer Datenmengen wird dadurch erheblich erleichtert.

Tabelle 5.1 zeigt, wie die einzelnen Typen über den *Core* angesprochen werden können. Die Schnittstellen führen dann den Code des jeweiligen Plugins aus. Sie sorgen außerdem dafür, dass die Referenzen auf den Core überall verfügbar sind und sie kümmern sich um die Fehlerbehandlung, falls Fehler innerhalb der Plugins auftreten. Plugins vom Typ Extension sind hier die Besonderheit. Die auszuführende Funktion wird im Aufruf der Funktion definiert. Der *Core* erzeugt bei diesem Typen außerdem **kein** Objekt der Plugin-Klasse. Das führt dazu, dass von der Funktion *call* nur statische Methoden aufgerufen werden können. Eine genauere Betrachtung der Konsequenzen daraus erfolgt in Abschnitt 5.2.2. Wird bei den anderen Typen der Parameter *Plugin* als leerer String übergeben, so verwendet der Core

Typ	Aufruf über den Core	Aufgerufene Funktion im Plugin
Navigation	navigate_menu(plugin, name, *sub_menus, **kwargs)	navigate()
Login	login(plugin,user, password, database)	login()
Verification	verify(plugin,condition, **kwargs)	verify()
Extension	call(plugin,functionName, **kwargs)	verschieden

Tabelle 5.1: Typen von Plugins und ihre Aufrufe

das erste Plugin vom entsprechenden Typen.

Mehrsprachigkeit

Die letzte große Aufgabe des Core ist es, Mehrsprachigkeit zu organisieren. Legt ein Tester beispielsweise die Sprache Englisch fest, müssen sämtliche Zeichenketten aus der Benutzeroberfläche der Anwendung, die den Ablauf des Tests beeinflussen, in der entsprechenden Sprache zur Verfügung gestellt werden.
Pyrrhon verwendet dazu die weit verbreitete Lösung *gettext* [Fou16]. *Gettext* ist eine freie, opensource Programmbibliothek für die Internationalisierung von Programmen. Python bietet zur Verwendung von *gettext* ein Modul. Das Prinzip von *gettext* ist einfach: Es wird

```
locales/
|__ de
|   |__ LC_MESSAGES
|         |__ file.po
|__ en
    |__ LC_MESSAGES
          |__ file.po
```

Abbildung 5.8: Für Gettext nötige Ordnerstruktur

eine Ordnerstruktur erwartet, welche in Abbildung 5.8 dargestellt ist. Ist diese Struktur vorhanden, sucht das Modul nach *.po* Dateien im jeweiligen Ordner *LC_MESSAGES*. Diese enthalten im Wesentlichen zeilenweise Übersetzungen aller Zeichenketten im Klartext. *Gettext* stellt dann eine Funktion zur Verfügung, mit der die Zeichenketten ersetzt werden können. Der schwierige Teil besteht hier für *Pyrrhon* darin, einem Entwickler von Plugins oder einem Tester so viel Arbeit wie möglich abzunehmen. Von Nutzern des Frameworks kann nicht erwartet werden, *gettext* in jedem Plugin neu zu initialisieren. Aus diesem Grund geschieht die Initialisierung im *Core*. Danach definiert *Pyrrhon* beim Ladevorgang für jedes Plugin eine Funktion, die einen Unterstrich als Namen trägt. Bei dieser Funktion handelt es

sich um die Ersetzungsfunktion von *gettext*.
Eine zu übersetzende Zeichenkette muss von einem Tester lediglich mit einem Unterstrich gefolgt von Klammern umgeben werden.

$$_(\text{“}Hallo\ Welt\text{“})$$

ist somit bereits eine übersetzbare Zeichenkette.

Decorator

Bei der Benutzung von *Pyrrhon* zeigte sich, dass unbegründete Fehlschläge automatisierter Abläufe häufig auftreten. Dies passiert aufgrund der großen Anzahl an verschiedenen Komponenten, die in diesem Framework miteinander kommunizieren. Taucht auf dem Testrechner z.B. ein unerwartetes Fenster von einer anderen Anwendung auf, kann dies bereits zu einem Fehlschlag führen. Nachdem die geforderte Fehlerquote bereits vereinzelt überschritten wurde, muss dem entgegen gewirkt werden. Da es nicht ratsam ist, Selenium selbst zu manipulieren, löst Pyrrhon das Problem mithilfe eines Decorators. Ein Decorator umschließt dabei ein Objekt und repräsentiert es nach außen. Der Decorator kann das Objekt dann zusätzlich erweitern. [Gei15][Seite 156]
In Python sind sämtliche Konstrukte Objekte. Das schließt auch Funktionen ein. Funktionen können also mittels eines Decorators erweitert werden. Dies hat den Vorteil, dass die Funktion selbst nicht verändert werden muss, da der Decorator diese lediglich verwendet. Der von Pyrrhon verwendete Decorator nennt sich „retry“. Verursacht die dekorierte Funktion einen Fehler, so wird sie bis zu dreimal erneut ausgeführt. Erst nach dreimaligem Fehlschlag gilt dann der Test als nicht bestanden. Dieser Ansatz verringert deutlich die Anzahl der ungerechtfertigten Fehlschläge. Weitere Voreile eines Decorators sind die Selektierbarkeit und die Verfügbarkeit für Benutzer des Frameworks. Mit dem Decorator ist es möglich, nur bestimmte Funktionen zu wiederholen. Sicherheitskritische Features beispielsweise könnten demnach weiterhin zu einem direkten Fehlschlag führen.
Plugin-Entwicklern steht die Entscheidung, wann ein Fehlschlag eintreten soll, durch die Verfügbarkeit des Decorators im *Core* ebenfalls zur Verfügung.

5.2.2 Plugins

Die Implementierung von Plugins gestaltet sich unkompliziert. Es müssen nur wenige Vorgaben, abhängig vom jeweiligen Typen, beachtet werden. Auch hier erhält der Typ *Extension* eine besondere Rolle. Während der *Core* von den Klassen aller anderen Typen beim Aufruf ein Objekt erstellt und dieses zur Ausführung verwendet, passiert dies nicht bei einer Erweiterung vom Typ *Extension*. Es wird, wie in Tabelle 5.1 gezeigt, eine angegebene Funktion aufgerufen, die als „static“ definiert sein muss.

Für Plugins dieses Typen, die weitere Objekte benötigen, bietet es sich an ein Factory-Pattern zu verwenden. So kann der Core dem Tester eine Factory der jeweiligen *Extension* zur Verfügung stellen. Das in Abbildung 5.9 dargestellte minimalistische Plugin *Cursor*, zeigt

```
class Cursor(Extension):
    """
    Plugin for manipulating the cursor. Needs show_cursor.js
    """
    plugin_info = PluginMetaData(pyrrhon_version=">=1.2.1",
                                 version="1.0",
                                 e1ns=">=2.2^",
                                 browsers=[Browser.Chrome, Browser.IE],
                                 dependencies=["show_cursor.js"]
                                 )

    @staticmethod
    def show_cursor(core):
        core.execute_js_file(file="show_cursor.js")
```

Abbildung 5.9: Ein einfaches Plugin vom Typ Extension

trotz seiner Einfachheit die Flexibilität des Ansatzes. Es ist in diesem Fall kein Factory-Pattern nötig, da dieses Plugin nicht mit eigenen Objekten arbeitet, sondern es lediglich zum Anzeigen eines Cursors an der Position der Maus dient. Erreicht wird das Anzeigen durch die Ausführung von JavaScript, das in einer zugehörigen Datei liegt. Der *Core*, der beim Aufruf der statischen Methode *show_cursor* über die Schnittstelle *call()* dem Plugin mit übergeben wird, stellt eine Funktion zur Verfügung, die der Ausführung von JavaScript dient. Diese verwendet das Plugin. Gleichzeitig bleibt das Plugin flexibel, da es nicht auf den Funktionsumfang des *Core* beschränkt ist.
Abbildung 5.9 zeigt die notwendigen Metadaten eines Plugins. Diese sind unabhängig vom Typen. Die 5 enthaltenen Parameter legen neben einer erforderlichen Minimalversion von Pyrrhon auch die Version des Plugins selbst und die benötigte Version von *Plato e1ns* fest. Ebenfalls können kompatible Browser sowie andere benötigte Plugins oder Dateien festgelegt werden. Auch bei ladbaren Plugins ist die Angabe einer Version möglich. Beispielsweise:
dependencies=["MyExtension==1.2"]
Der *Core* erkennt so eine Angabe als Versionspezifikation und berücksichtigt sie. Wird eine dieser Spezifikationen nicht eingehalten, wird das Plugin vom *Core* nicht geladen und zur Verfügung gestellt. Alle weiteren vorhandenen Plugin-Typen folgen ebenfalls diesem Prinzip, mit dem einzigen Unterschied, dass ihr Aufruf jeweils über die in Tabelle 5.1 angegebenen Funktionen passiert. Im Gegensatz zur *Extension* müssen diese Methoden nicht statisch sein, da der *Core* in der Lage ist, zuerst entsprechende Objekte zu erzeugen.

Umgang mit Javascript

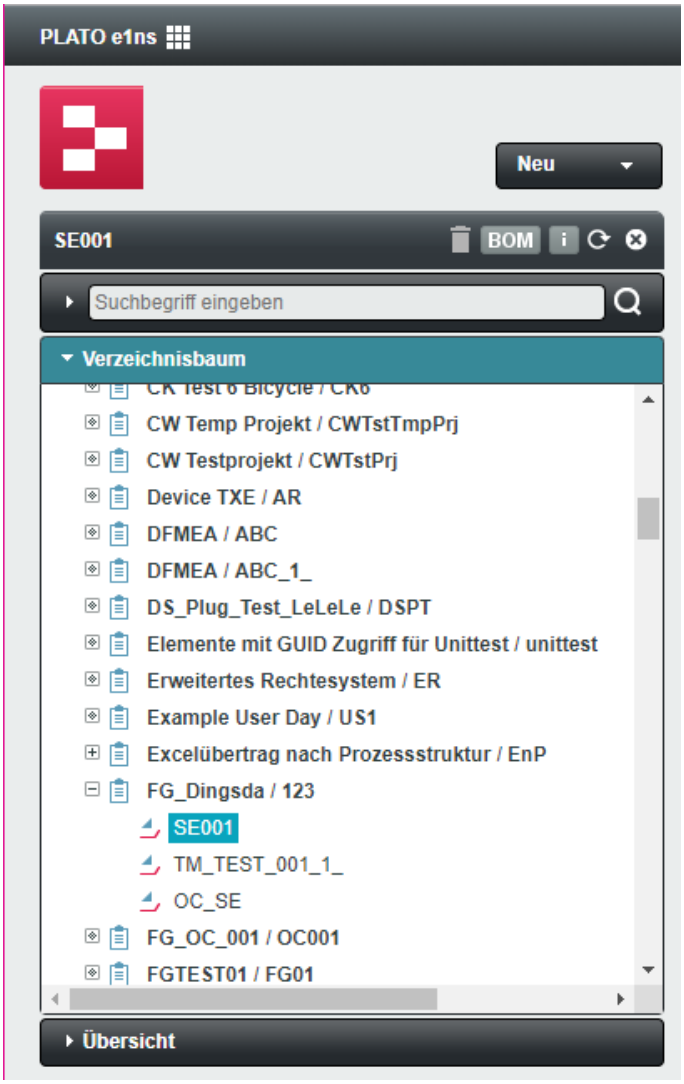

Abbildung 5.10: Der Verzeichnisbaum in *Plato E1ns*

Ein weiteres Beispiel für die Flexibilität des Plugin-Ansatzes ist der Umgang mit Javascript. In *Plato e1ns* wird für die Darstellung und Verwaltung vieler Strukturen die Bibliothek *FancyTree*[Wen19] verwendet. Ein zentrales Element ist z.B. der in Abbildung 5.10 dargestellte Verzeichnisbaum. Wie man sieht, handelt es sich um eine Liste, deren Elemente jeweils Unterelemente haben können. Die Unterelemente sind initial nicht sichtbar. Sie müssen „ausgeklappt“ werden. Aus Performancegründen werden die Daten der Unterelemente erst in diesem Moment geladen. Das macht eine Interaktion mit Unterelementen zu einem früheren Zeitpunkt unmöglich, da diese nicht im HTML-Quellcode auftauchen. Um nicht jedes Element durch Anklicken ausklappen zu müssen, ist es hilfreich, die Bibliothek FancyTree [Wen19] direkt ansprechen zu können.
Über das Plugin-System von Pyrrhon ist dies einfach möglich. Der *Core* bietet Funktionen zum Ausführen einer JavaScript Datei oder einer einzelnen Funktion in dieser Datei. Es kann also entsprechender JavaScript-Code verfasst und dann in Form eine Plugins aufgerufen werden.

Im Codeblock 5.11 wird beispielhaft gezeigt, wie *Pyrrhon* erweitert werden kann. Der Fokus liegt hier nicht auf der Benutzung von *FancyTree*. Das Besondere an dieser Funktion ist die Verwendung der asynchronen Funktion *setExpanded*. Damit die Verwendung von asynchronen Funktionen möglich wird, schreibt der *Core* bei jedem Aufruf von asynchronem Code eine Callback-Funktion in die Variable *injected*. Der Aufruf dieser Funktion signali-

```
function expand_tree(){
        callback=injected;
        var tree = $.ui.fancytree.getTree();
        var node=  tree.rootNode
        node.children.forEach(function(ele) {
                ele.setExpanded(true).done(function(){
                //do stuff when done
                        }).fail(function(){
                                // The 'fail' function is called on error
                }).always(function(){});
        });
        callback(true);
        return true;
}
```

Abbildung 5.11: Beispielhaftes JavaScript zur Erweiterung von *Pyrrhon*

siert Selenium, dass der Test fortgeführt werden kann. Es kann außerdem ein Basisdatentyp übergeben werden, der als Rückgabewert im Python Code fungiert. Das Beispiel 5.11 wartet in diesem Fall aus Performancegründen nicht auf die Beendigungen der Aktionen, da kein Rückgabewert notwendig ist. Wenn an dieser Stelle auf das Laden großer Datenmengen gewartet werden muss, sollte dies im Test-Skript passieren, da es für die Ausführung von JavaScript Code in Selenium ein Zeitlimit von 30 Sekunden gibt. Nach Möglichkeit sollten zeitintensive Aktionen also nicht in JavaScript ausgeführt werden.
Um diesen JavaScript Code in *Pyrrhon* verwenden zu können, muss zusätzlich ein kleines Python Plugin geschrieben werden.

```
class FancyTreeWorker(Extension):
        plugin_info = PluginMetaData(pyrrhon_version=">=1.2.1",
                                     version="1.4",
                                     e1ns=">=2.2^",
                                     browsers=[Browser.Chrome],
                                     dependencies=["FancyTreeWorker.js"])

        @staticmethod
        def expand_tree(core, **kwargs):
                return core.execute_js_function(file="FancyTreeWorker.js",
                                     func="expand_tree();",async=True)
```

Abbildung 5.12: Plugin zur Ausführung von JavaScript Dateien

Codeblock 5.12 zeigt eine gekürzte Version eines Plugins zur Ausführung des JavaScripts aus 5.11. Durch die angegebene Abhängigkeit von der JavaScript-Datei wird sichergestellt,

dass diese auch vorhanden ist. Dann kann eine statische Methode definiert werden, die die Ausführung von JavaScript im *Core* anfragt. Diese Methode kann im Test, wie jedes Plugin vom Typ *Extension*, einfach über die Schnittstelle *call()* aufgerufen werden.
Dieses Vorgehen erlaubt eine einfache Kontrolle über die Ausführung von JavaScript. Entsprechende Plugins können so bei Bedarf einfach entfernt werden.

5.2.3 Tests

Zuletzt wird der Aufbau eines Tests betrachtet werden. Dieser muss allerdings keiner bestimmten Struktur folgen. Es genügt, das Framework zu initialisieren. Codeblock 5.13 zeigt

```
# -*- coding: utf-8 -*-
import plato_pyrrhon
import unittest
class Pyrrhontests(unittest.TestCase):

def setUp(self):
        self.p = plato_pyrrhon.Core.from_conf()
        self.p.login(user="user", password="passwort", database="datenbank")

def tearDown(self):
        self.p.driver.quit()

# ------------------------Testcases
def test_A(self):
        self.p.wait_page(1)
        self.p.navigate_menu("MainMenu", "Actions")
        self.p.wait_page(6)
        assert(self.p.verify("VerifyActions","is_start_after_milestone",
                             project_title = "automatisiert_B",
                             milestone_title= "MS1"))
        self.p.wait_page(1)
        assert (self.p.verify("VerifyActions","is_start_after_milestone",
                             project_title="automatisiert_A",
                             milestone_title="MS1"))
```

Abbildung 5.13: Einfacher Test mit Plato Pyrrhon

einen einfachen Test, der beispielhaft als Unittest aufgebaut ist. Es wird das Framework initialisiert, dann wird ein Login ausgeführt und das Modul „e1ns.actions“ geöffnet, um dort die angezeigte Reihenfolge gegebener Daten zu verifizieren. Zu deutlich sehen ist, wie *Pyrrhon* zu verwenden ist. Nach der Initialisiering des *Core* wird dieser für alle Aktionen verwendet. Die Methode *Core.from_conf* liest die Konfiguration vor dem Aufruf des Konstruktors ein. Sichtbar sind ebenfalls verschiedene Aufrufarten der in Tabelle 5.1 gezeigten

Plugintypen. Es wird bei dem hier gezeigten Test keine Verwendung eines bestimmten Plugins vorgeschrieben. Das Framework wählt das zu verwendende Plugin deshalb selbst. Bei der Benutzung der *verify* Schnittstelle wird hingegen ein Plugin explizit angegeben.
Die Schnittstelle *verify* ermöglicht im Test außerdem die einfache Verwendung von *assert*. So können komplexere Vergleiche oder Rechnungen vom jeweiligen Plugin übernommen werden.

6 Test

Das Testen des Testframeworks gestaltet sich aufgrund der hohen Komplexität schwierig. Auch in diesem Kapitel erfolgt deshalb die Unterteilung in die Schichten *Core* und *Plugins*. Selenium, auf das *Pyrrhon* aufbaut, wird in dieser Arbeit als ausreichend getestet betrachtet, da es sich hier um ein quelloffenes Framework handelt, das eine große Benutzerbasis besitzt.

6.1 Core

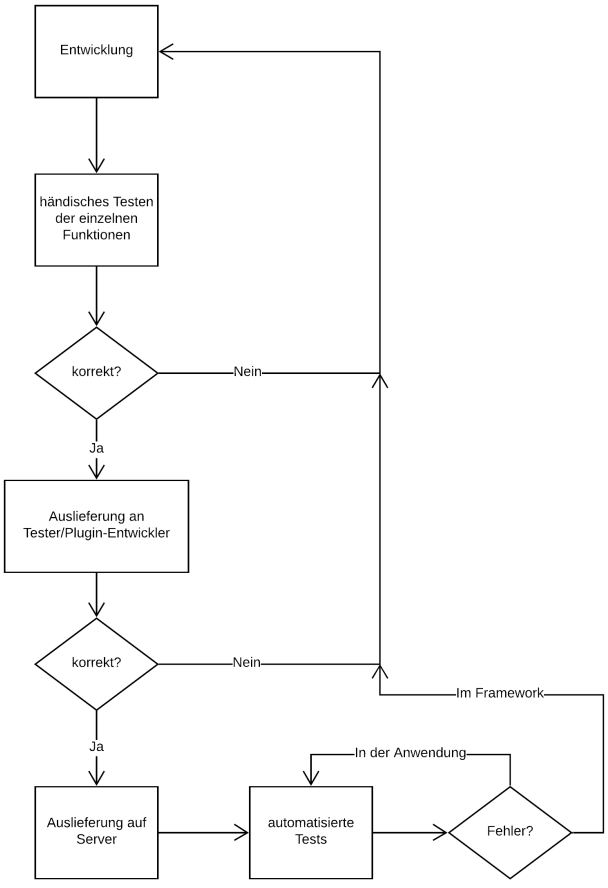

Abbildung 6.1: Ablauf vom Test des *Core*s.

Die Tatsache, dass Selenium als getestet gilt bedeutet jedoch nicht, Dass das ganze Framework korrekt funktioniert. Die einzelnen Funktionen des *Core*s verwenden zwar Selenium, bieten aber selbst Raum für Fehler.
Das Testen des Frameworks stellt eine Herausforderung dar, denn hier zeigen sich exakt dieselben Probleme, die mit dem Framework gelöst werden sollen. Eine Funktion des *Core*s muss beispielsweise ebenfalls sämtliche Browser unterstützen. Tests finden deshalb nicht automatisiert statt.

Und ein Release läuft in mehreren Stufen ab. Nach der Fertigstellung einer neuen Version werden sämtliche Funktionen an denen es Änderungen gab, möglichst zuerst einzeln und mit verschiedenen Parametern aufgerufen.
Zeigen sich die gewünschten Rückgabewerte, werden Tests, die mit älteren Versionen bekanntermaßen funktionsfähig sind, in verschiedenen Konfigurationen

ausgeführt und beobachtet. Wenn auch diese Tests erfolgreich sind, wird die Version an die Tester ausgeliefert. Diese Schreiben für einige Zeit Tests. Sollte im Laufe dieser Zeit ein Fehler bemerkt werden, findet ein Rollback zur vorherigen Version statt und der Prozess beginnt nach dem Beheben des Fehlers erneut.
Fällt den Testern bei der Automatisierung ebenfalls nichts auf, wird die Version auf den Servern installiert, die die Tests automatisiert durchführen.
Kommt es hier zu einem Fehlschlag eines Tests, sieht sich ein Tester die Ergebnisse an und sucht nach der Fehlerquelle. Wenn sich herausstellt, dass es ein Problem im Framework gibt, wird der Fehler gemeldet. Die Tester unterscheiden hier **nicht**, ob der Fehler in Plugins, dem *Core* oder dem Test selbst liegt. Sie weisen lediglich nach, dass das Problem nicht in der zu testenden Software liegt.
Unabhängig von der Schwere des Fehlers ist, dank des modularen Aufbaus, ein einfacher Rollback möglich.
Diese Art des Testens ist nur möglich, da es sich bei dem Einsatzgebiet des Frameworks um einen nicht kritischen Bereich handelt. Versagt eine automatisierte Lösung, greifen menschliche Tester ein.

6.1.1 Fälschlicherweise bestandene Tests

Es kann vorkommen, dass Tests fälschlicherweise bestanden werden, was ein Problem darstellt. Die Fehlerquelle muss dabei nicht einmal beim Framework liegen. Vergisst ein Tester z.B. die Verifizierung der korrekten Verarbeitung eingegebener Daten, so kann ein Test fälschlicherweise bestanden werden.
Die Möglichkeiten, einen Test fälschlicherweise zu bestehen, sind zahlreich und können daher kaum ausgeschlossen werden. Auch wenn dieser Fall nur sehr selten eintritt, muss also immer wenigstens eine stichprobenartige menschlichen Kontrolle stattfinden. Erläuterungen hierzu erfolgen abschließend in Kapitel 7.

6.2 Plugins und Tests

Es liegt in der Veranwortung jedes Entwicklers, sein Plugin zu testen. Das bedeutet, dass ab dieser Schicht verschiedene Testmethoden Anwendung finden können. Ein Entwickler kann sein Plugin entweder automatisiert oder auch von Hand testen. Bei den im Rahmen dieser Arbeit entwickelten Plugins findet das gleiche Verfahren Anwendung wie in Abschnitt 6.1. Hier wird ein Problem des geschichteten Aufbaus deutlich. Finden Tests von neu implementiertem Code auf einer bestimmten Schicht statt, so muss der Implementierende von der Korrektheit der darunterliegenden Schichten ausgehen. Diese muss aber nicht zwingend gegeben sein. Erläuterungen hierzu erfolgen ebenfalls in Kapitel 7.

6.3 Beurteilung des Aufwandes

Es zeigt sich durch produktiven Einsatz, dass es unter Berücksichtigung des zu erbringenden Aufwandes kaum möglich ist, auch das Framework selbst automatisiert zu testen. Auf Seiten des Frameworks entstehende Fehler sind häufig unvorhersehbar und beschränken sich nicht nur auf einen bestimmten Kontext. Fehler, die durch das Betriebssystem verursacht werden, sind im Framework ebenso möglich wie ein falsches Verhalten im Browser. Diese Bandbreite an Fehlermöglichkeiten automatisiert zu testen, erfordert sehr viel Entwicklungsarbeit, die sich gegenüber des gebotenen Nutzens nicht rentiert. Das händische Testen des Frameworks bietet eine sicherere Methode zur Fehlererkennung. Gleichzeitig kann ein manueller Test des Frameworks unabhängig von zu testenden Programmen stattfinden. Dadurch bleibt eine deutliche Ersparnis personeller und und zeitlicher Ressourcen erhalten.

7 Schlussbetrachtung

Abschließend wird die Arbeit reflektierend betrachtet und der Erfolg dieser evaluiert. Die in 7.1 aufgeführten Rückmeldungen spiegeln die subjektive Wahrnehmung der befragten Personen wider. Bei den befragten Personen handelt es sich ausschließlich um Mitarbeiter der Plato AG. Die befragten Personen haben unterschiedliche Programmierkenntnisse. Sie übernahmen hauptsächlich die Rolle eines Plugin-Entwicklers (Siehe Kapitel 2.4). Während der Entwicklung dieser Plugins entstanden jedoch auch Tests. Diese Funktion wurde also ebenfalls übernommen.

7.1 Nutzerfeedback

Das erhaltene Feedback ist fast ausschließlich positiv. Die befragten Personen loben ausdrücklich die Modularität und die daraus resultierende Flexibilität. Es fehle zwar noch einiges an Funktionen, diese könne man jedoch sehr einfach hinzufügen. Aus diesem Fehlen einiger, nicht geforderten aber gewünschten Funktionalitäten folge, dass es immer noch zu häufig nötig sei direkt auf Selenium zurückzugreifen. Beispielsweise empfanden die befragten Personen den Umgang mit den Frames (siehe Abbildung 5.4) als zu umständlich. Dieser Vorgang sollte automatisiert stattfinden.
In Bezug auf die Plugins wird am häufigsten das Fehlen eines Import-Handlers erwähnt.
Ist es nötig, dass ein Plugin ein anderes Plugin verwendet, ist vorgesehen, dass ein Objekt der gewünschten Klasse mit Hilfe der Liste *Plugins* im *Core* erzeugt wird. Stattdessen zeigt sich bei den Benutzern eine andere Tendenz: Es wird in fast allen Fällen versucht, Plugins mittels der *Import* Anweisung verfügbar zu machen. Dies führt während der Laufzeit beispielsweise aufgrund mehrerer Ordner aber zu häufigen Fehlern. Das Verfahren über die vom *Core* verwaltete Liste erscheint den Benutzern unnötig kompliziert.
Die in 2.4 erläuterte Schichtung erfüllt den Rückmeldungen zufolge den geplanten Zweck. Die parallele Arbeit an mehreren Dingen sei problemlos möglich.
Vermisst wurde weiterhin eine Funktion zum einfachen Auslösen von Tastenkombinationen auf der Tastatur.

7.2 Zusammenfassung

Zusammenfassend lässt sich *Pyrrhon*, trotz einiger Schwächen, als Erfolg bezeichnen. Bereits während der Entstehung konnten im Einsatz bei der *Plato AG* einige Fehler mit großer Zeitersparnis gefunden werden. Überwiegend positive Rückmeldungen bezüglich des Funktionsumfangs sowie der Benutzbarkeit stärken diesen Eindruck. Es ist jedoch nicht nur Gutes zu sagen. So ist das entstandene Framework nicht als autonom zu betrachten. Aufgrund der zahlreichen Fehlermöglichkeiten bedarf es in jedem Fall menschlicher Unterstützung. So muss der Test bei einem Fehlschlag des automatisierten Tests fast immer händisch wiederholt werden, um die genaue Fehlerquelle identifizieren zu können.

7.2.1 Zeitersparnis

Die Zeitersparnis fällt unterschiedlich stark aus. Die Befragung von Kollegen ergab, dass einige Abläufe automatisiert sogar mehr Zeit benötigen als manuell, andere wiederum mehrere Stunden pro Durchlauf sparen. Dabei spielen verschiedenste Faktoren eine Rollen.
Grob gilt hier: Je mehr Daten in einem Test angelegt werden sollen - also je mehr Benutzereingaben notwendig sind, desto größer die Zeitersparnis. Hierzu zählen im Fall von *Plato e1ns* beispielsweise Projekte und deren Meilensteine. Kommt es während eines Tests häufig zu Ladezeiten, sinkt die Zeitersparnis, denn aufgrund der Berücksichtigung möglicher Geschwindigkeitsschwankungen der Software kommt es häufig zu langen Wartezeiten des Frameworks. Hier liegt seine vermutlich größte Schwäche.
Für *Plato* spielt dies jedoch nur eine geringe Rolle, da die Tests über Nacht auf eigenen Rechnern stattfinden und somit keine zeitkritischen Abläufe blockieren. Die Möglichkeit eines automatisierten Ablaufs ist wichtiger als seine Geschwindigkeit. Der komplexeste Test, der während dieser Arbeit entstand umfasst knapp 200 Zeilen Code, was in etwa einem Drittel der kompletten Tests eines Moduls von *Plato e1ns* entspricht. Konkrete Zeitangaben bezüglich der Laufzeit eines Tests lassen sich jedoch nur schwer machen, da Leistungsschwankungen der Software unvorhersehbar auftreten und den Test signifikant verlangsamen können und die Art der durchgeführten Aufgaben hier ebenfalls eine Rolle spielt.

7.3 Ausblick

Nach dieser Arbeit wird die Entwicklung von *Pyrrhon* fortgesetzt. Wie bereits in 4.3 erwähnt, erreicht die verwendete Version von Python zum Ende des Jahres 2019 den Status „End of Life“. Daher ist ein Umstieg auf Python 3 geplant. Da diese Möglichkeit bereits außerhalb des Kontextes dieser Arbeit während der Entstehung des Quellcodes in Betracht gezogen wurde, sollten sich hier keine größeren Probleme ergeben. Um der gegebenen Rückmeldung des zu häufig nötigen Zugriffs auf Selenium entgegenzuwirken, wird auch weiterhin der Funktionsumfang erweitert werden.

Eine Anpassung der in Abschnitt 7.1 kritisierten Funktionalitäten wird geprüft. Eine Funktion, die es nicht in die Version geschafft hat, die für diese Arbeit eingereicht wurde, ist das automatische Aufsetzen neuer Instanzen von *Plato e1ns*. Der Grund hierfür ist die Einstellung der Unterstützung im dafür vorgesehenen Programm. Diese Funktion wird demnach zu einem späteren Zeitpunkt auf andere Weise realisiert. Erste Tests zeigten, dass die Zeitersparnis und damit der Nutzen durch eine automatisierte Installation deutlich steigt. Für die bereits erwähnte, häufig zu lange Wartezeit wird es in Zukunft Anpassungen geben, die die Performance deutlich steigern werden.

Abbildungsverzeichnis

4.1 Schichtung des Frameworks 12
4.2 Pseudo-Klassendiagramm der untersten Schicht 13

5.1 Ein einfacher Test in Selenium 17
5.2 UML-Darstellung der Klasse Sysenv 17
5.3 Bespielkonfiguration 18
5.4 Frames in Plato e1ns 19
5.5 Konkrete Implementierung eines Wechsels des Frames 20
5.6 Laden von Klassen zur Laufzeit in Python 2.7 21
5.7 Definition eines Plugin-Typen 22
5.8 Für Gettext nötige Ordnerstruktur 23
5.9 Ein einfaches Plugin vom Typ Extension 25
5.10 Der Verzeichnisbaum in *Plato E1ns* 26
5.11 Beispielhaftes JavaScript zur Erweiterung von *Pyrrhon* 27
5.12 Plugin zur Ausführung von JavaScript Dateien 27
5.13 Einfacher Test mit Plato Pyrrhon 28

6.1 Ablauf vom Test des *Core*s. 30

1 Demo-Applikation 41

Tabellenverzeichnis

2.1 Priorisierungsklassen nach Eisenhower-Prinzip 4
2.2 Beschreibung Funktionaler Anforderungen 5
2.3 Beschreibung nicht funktionaler Anforderungen 7
2.4 Beschreibung technischer Anforderungen 8

5.1 Typen von Plugins und ihre Aufrufe 23

Literatur

[AG19] Plato AG. *Plato E1ns*. 2019. URL: https://www.plato.de/e1ns/ (besucht am 07. 03. 2019).

[Fou13] Eclipse Foundation. *Jubula*. 2013. URL: https://www.eclipse.org/jubula/ (besucht am 06. 03. 2019).

[Fou16] Free Software Foundation. *Introduction to gettext*. 2016. URL: https://www.gnu.org/software/gettext/ (besucht am 27. 03. 2019).

[Gei15] Matthias Geirhos. *Entwurfsmuster. Das umfassende Handbuch*. 1. Aufl. 2015. ISBN: 978-3-8362-2762-9.

[ITW19] ITWissen.info. *Definition Plugin*. 2019. URL: https://www.itwissen.info/Plug-In-plug-in.html (besucht am 06. 03. 2019).

[JUn19] Team JUnit. *JUnit*. 2019. URL: https://junit.org/ (besucht am 06. 03. 2019).

[Kaw11] Kohsuke Kawaguchi. *Jenkins*. 2011. URL: https://jenkins.io/ (besucht am 11. 03. 2019).

[KB18] Veikko Krypczy und Olena Bochkor. *Handbuch für Softwareentwickler*. 1. Aufl. 2018. ISBN: 978-3-8362-5492-2.

[Ker15] Sascha Kersken. *IT-Handbuch für Fachinformatiker. Der Ausbildungsbegleiter*. 7. Aufl. 2015. ISBN: 978-3-8362-3473-3.

[Pel17] Jason Pellerin. *Nose*. 2017. URL: https://nose.readthedocs.io/en/latest/ (besucht am 04. 03. 2019).

[Pro18] Selenium Project. *Selenium*. 2018. URL: https://www.seleniumhq.org/ (besucht am 04. 03. 2019).

[Sta+07] Thomas Stahl u. a. *Modellgetriebene Softwareentwicklung. Techniken, Engineering, Management*. 2. Aufl. 2007. ISBN: 978-3-89864-448-8.

[Swe18] Al Sweigart. *PyAutoGUI*. 2018. URL: https://github.com/asweigart/pyautogui (besucht am 06. 03. 2019).

[Wen19] Martin Wendt. *FancyTree Framework*. 2019. URL: https://github.com/mar10/fancytree (besucht am 11. 04. 2019).

[Wik19] Wikipedia. *Pyrrhon von Elis*. 2019. URL: https://de.wikipedia.org/wiki/Pyrrhon_von_Elis (besucht am 13. 03. 2019).

Anhang

1 Voraussetzungen für die Nutzung

Die folgende Anleitung geht davon aus, dass das System auf dem das Framework verwendet werden soll einige Voraussetzungen erfüllt. Im Folgenden werden diese aufgelistet.
Es wird von der Verwendung eines neueren Microsoft Windows ausgegangen und ein Vorgehen nur für diese Plattform erläutert. Die Ausführung auf anderen Plattformen ist ungetestet, sollte aber grundsätzlich möglich sein.

- Python 2.7.16 oder neuer ist installiert. (Ein Installationsprogramm liegt der Arbeit bei.)
- Die Pythonumgebung und deren Ordner *Scripts* befinden sich in der Umgebungsvariable *PATH* des Systems. (Ist dies nicht der Fall, müssen sämtliche der folgenden Befehle manuell um die passenden Pfade erweitert werden)
- Der Paketmanager *Pip* ist funktionsfähig. (Das ist nach der Installation von Python 2.7.16 normalerweise gegeben)
- Eine Konsole ist geöffnet und es wurde bereits in den Pfad des Ordners „Code“ navigiert.
- Der Browser der den Test ausführen soll, muss in einer Version installiert sein, die zu den enthaltenen Treibern passt. Ist die Version zu neu, müssen die Driver im Ordner *tests/Browser* gegebenenfalls manuell aktualisiert werden) Die unterstützten Browser sind Google Chrome 74.X und Mozilla Firefox 66 oder höher. Die Driver für die Browser von Microsoft sind aufgrund von sehr kurzlebigen Versionen nicht enthalten, können aber manuell heruntergeladen werden.

2 SD-Karte

Dieser Arbeit liegt eine SD-Karte bei. Auf dieser SD-Karte befindet sich in dem Ordner „Code“ der Quellcode des in dieser Arbeit erstellten Frameworks. Die Version auf dieser SD-Karte wurde leicht modifiziert um die Verwendung ohne Software der *Plato AG* zu ermöglichen.

Der Ordner „Dokument“ enthält das Latex-Projekt dieses Dokumentes und eine entsprechende PDF-Datei.

2.1 Verwendung der SD-Karte

Sind die erwähnten Vorraussetzungen erfüllt, kann das Framework mit folgenden Befehlen installiert werden.

```
pip install .\plato_pyrrhon-1.4.1.BA-py2-none-any.whl
pip install nose
```

Danach kann in den Ordner „tests“ gewechselt werden. Das Beispiel lässt sich dann mittels

```
nosetests demo.py
```

ausführen.
Kommt es hier zu Fehlern, kann die Konfigurationsdatei *config.yml* geöffnet und bearbeitet werden. Hier müssen z.B. die Pfade angepasst werden, falls die Driver manuell heruntergeladen werden oder der Browser gewechselt werden soll. Als Standardeinstellung ist Google Chrome gewählt.

2.2 Die Ordner im Einzelnen

Im Ordner „docs“ befindet sich die Dokumentation, die aus den Kommentaren im Code erzeugt wurde. Durch öffnen der Datei *docs/_build/html/index.html* kann die Dokumentation angesehen werden.

Der Ordner „/plato_pyrrhonWheel“ enthält alle Skripte, die notwendig sind um den Code in ein neues Wheel zu verpacken.

Der Ordner „/pyrrhon_plugins“ enthält sämtliche Plugins.

Im Ordner „sample_app“ befindet sich die kleine Anwendung, die für Demonstrationszwecke geschrieben wurde. Sollte die vorkonfigurierte Seite nicht erreichbar sein kann das Hosting eigenständig stattfinden.

Der Ordner „src“ beinhaltet den Code des Framework-Kerns. Dieser Teil wird durch das Wheel installiert.

In „tests“ befindet sich der Test für die Demo, sowie die Browser-Driver und Log-Dateien.

3 Die Demo-Applikation

Die Demo-Applikation ist bewusst einfach gehalten, da der Fokus dieser Arbeit nicht auf ihr liegt. Es handelt sich um eine kleine Website, die Zitate anzeigt und diese zufällig wechseln kann. Der Benutzer kann auch neue Zitate hinzufügen. Diese werden aber nicht gespeichert und bleiben nur bis zur nächsten Aktualisierung der Seite erhalten.

I agree, there is no reason a human and a machine can't be friends, right? I mean... I'm glad you said yes.
-Chloe

(Detroit: Become Human)

Abbildung 1: Demo-Applikation