Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Benjamin Burchard

Automatisierte Testdatenerstellung mit Klassifikationsbäumen für Web-Testing

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Benjamin Burchard

Automatisierte Testdatenerstellung mit Klassifikationsbäumen für Web-Testing

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

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

Betreuender Prüfer: Prof. Dr. Zhen Ru Dai
Zweitgutachter: Prof. Dr. rer. nat. Stephan Pareigis

Eingereicht am: 17. Mai 2013

Benjamin Burchard

Thema der Arbeit
Automatisierte Testdatenerstellung mit Klassifikationsbäumen für Web-Testing

Stichworte
Web-Testing, Klassifikationsbäume, Test-Automatisierung, Java, JUnit, Selenium, CTE

Kurzzusammenfassung
Diese Arbeit hat zum Ziel, mit Hilfe von Klassifikiationsbäumen und den daraus entstehenden Testfällen, automatisch Testdaten für Web-Testing zu erstellen. Um dies zu erreichen, wird ein Java-Programm entwickelt welches aus den Daten eines Klassifikationsbaum-Tools (Classification Tree Editor XL) Testdaten für vorgefertigte Testfälle erstellt. Diese werden, unter Verwendung des Selenium Webdriver und JUnit, automatisch ausgeführt und die daraus resultierenden Ergebnisse dargestellt und ausgewertet.

Benjamin Burchard

Title of the paper
Automated Testdata-Generation with classification trees for Web-Testing

Keywords
Web-Testing, classification trees, test automation, Java, JUnit, Selenium, CTE

Abstract
This thesis has the goal to automatically create test data for web-testing from testcases deigned by calssification trees. To achieve this, a Java program is developed which creates test data from a classification tree tool (Classification Tree Editor XL) for pre-built testcases. These will be automatically executed using Selenium Webdriver and JUnit and the resulting findings will be graphically displayed and evaluated.

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Solides Testen im Web wird zunehmend zu einem der wichtigsten Bereiche in der Software Entwicklung sowie im Software Engineering. Um die Softwarequalität von Webseiten sicherzustellen, ist es essentiell anspruchsvolle Programme für die Testautomation zu entwickeln.

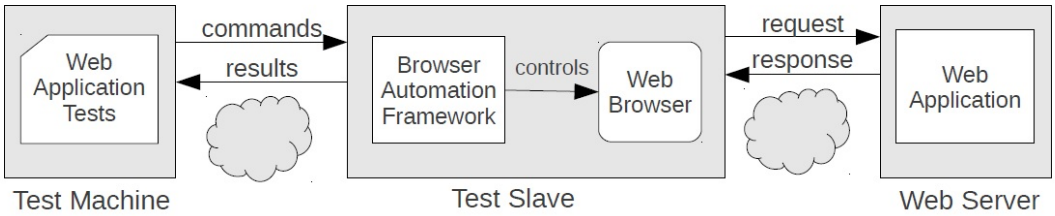

Abbildung 1.1: Ein typischer Web-Testing Aufbau [CZVO11]

Da Web-Applikationen als Client-Server Modell realisiert sind, ist der Testaufwand größer als bei traditionellen Programmen, welche nur lokal ablaufen. Ein typischer Web-Testing Aufbau ist in Abbildung 1.1 illustriert. Die Tests werden auf der Testmaschine ausgeführt welche den Test Slave ansteuert um die Tests auszuführen. Dieser besteht aus einem Browser Automation Framework (BAF) sowie dem Webbrowser, welcher durch das BAF gesteuert wird. Tests für Web-Applikationen bestehen aus mehreren Aktionen welche auf der Webseite, im Webbrowser, umgesetzt werden. Für jeden Testfall besteht der Testinput aus Daten, welche in Formulare eingegeben oder aus einer Folge von Ereignissen (z.B. Mausklick, Markierung), die auf Webelemente angewandt werden. Diese Tests werden über das BAF und den Webbrowser ausgeführt. Das BAF nimmt diverse Kommandos entgegen, welche es dann im Browser ausführt [CZVO11].

Das Design dieser Testfälle ist also eine entscheidende Testkomponente, welche in hohem Maße die Qualität eines Tests bestimmt. Die Auswahl der Testfälle legt den Grundstein für die Art sowie die Fokussierung des Tests (vgl. [HAO09]).

Wenn Testfälle auf Basis von Spezifikationen festgelegt werden, sind die daraus resultierenden Tests funktional. Funktionale Tests sind höchst wichtig für die Verifikation von Software und werden von einer breiten Masse in der Industrie eingesetzt. Allerdings gibt es nur wenige Methoden und Applikationen welche die systematische Erstellung von Testfällen für funktionale Tests unterstützen. Deshalb wird häufig auf nur teilweise anwendbare Tools zurückgegriffen, wie Microsoft Excel oder Entscheidungstabellen.

Die Klassifikationsbaum-Methode (engl. CTM - Classification Tree Method) ist eine effiziente Testmethode für funktionales Testen, welche die Möglichkeit bietet, den kompletten Eingabebereich eines Testobjekts in unabhängige Äquivalenzklassen aufzuteilen. Durch diese Einteilung kann sich die Menge der Testdaten erheblich reduzieren.

1.2 Zielsetzung

Ziel dieser Arbeit ist es eine Java-Applikation zu entwickeln welche mit Hilfe von Klassifikationsbäumen automatisiertes Web-Testing durchführt. Web-Testing beschreibt den Teil des Softwaretestens, der sich auf Applikationen aus dem Web bezieht.

Um die zu testenden Daten für die Webapplikation sinnvoll einzuschränken und damit den Testaufwand zu verringern, wird die CTM verwendet. Die dafür benötigten Klassifikationsbäume werden mit dem Programm Classification Tree Editor XL Professional (CTE XL) von Berner & Mattner[1] erstellt. Abbildung 1.2 zeigt den CTE XL. Mit diesem Editor ist es weiterhin möglich Testfälle zu generieren. Die im CTE erstellten Bäume und generierten Testfälle sollen in das zu entwickelnde Programm geladen und automatisiert abgearbeitet werden.

Mit Hilfe des Browser Automation Frameworks Selenium[2] und dessen WebDriver API (Application programming interface) für Java wird die Ansteuerung des Browsers automatisiert. Die API wirkt unterstützend bei der Kommunikation mit Webseiten, so dass auf diesen programmiertechnisch navigiert und interagiert werden kann.

[1] Berner & Mattner: http://www.berner-mattner.com
[2] Selenium Web Browser Automation: http://www.seleniumhq.org

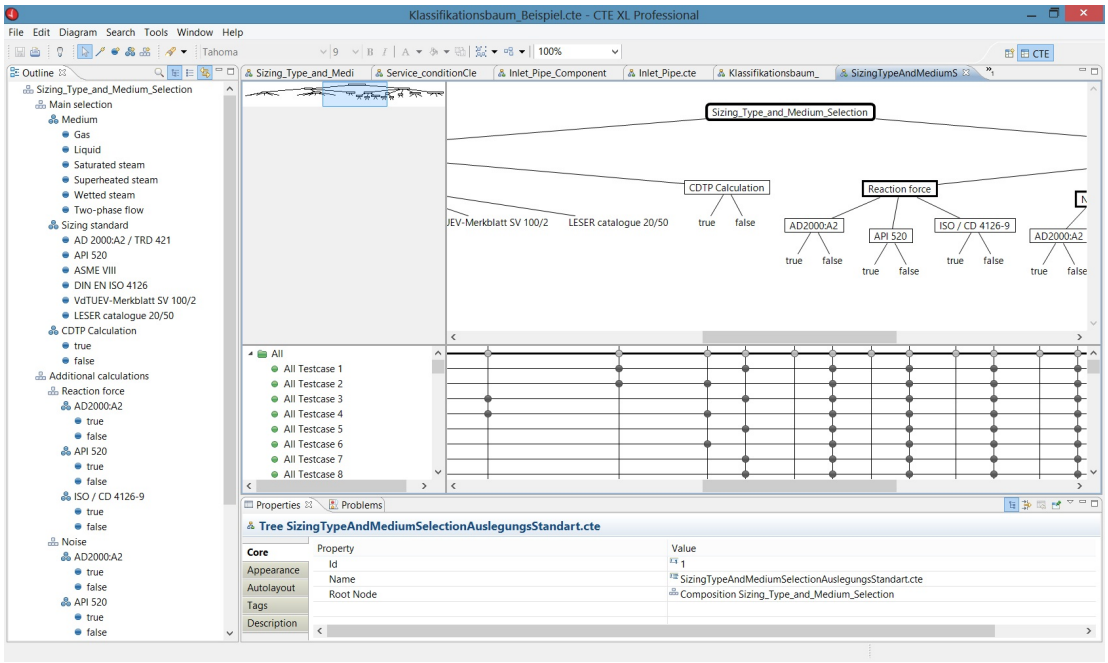

Abbildung 1.2: Classification Tree Editor XL

Zur Ausführung und Auswertung der Testfälle werden Unit Tests für die spezifischen vorgegebenen Webseiten entwickelt, um diese automatisch, mit den Daten aus den zuvor erstellten Klassifikationsbäumen und der WebDriver API auszuführen.

Im Endeffekt soll die Applikation dem Tester die Erstellung von Testfällen erleichtern sowie die Ausführung dieser automatisieren. Durch diese Vorteile reduziert sich der Testaufwand deutlich und ermöglicht eine effizientere Bearbeitung von Testobjekten.

1.3 Gliederung

Kapitel 1

Die Einleitung beschreibt die Motivation und Zielsetzung der Arbeit.

Kapitel 2

Im zweiten Kapitel wird auf die Grundlagen für die Entwicklung des Programms eingegangen.

Kapitel 3

Das dritte Kapitel analysiert die Anforderungen und behandelt das vorliegende Szenario.

Kapitel 4

Dieses Kapitel behandelt die Implementierung und geht auf die Architektur sowie die Anwendung der Grundlagen ein.

Kapitel 5

Kapitel fünf beinhaltet die Fallstudie, es behandelt hauptsächlich die Anwendung des Programms auf das Testobjekt.

Kapitel 6

Im Kapitel sechs werden die Resultate der Arbeit zusammengefasst, es wird ein Fazit gezogen sowie ein Ausblick gegeben.

2 Grundlagen

Dieses Kapitel beschäftigt sich mit den Grundlagen für die Entwicklung des Programms, der Erstellung von grundlegenden Elementen für die Arbeit sowie den in der Arbeit eingesetzten Werkzeugen. Es behandelt zunächst die theoretischen Grundlagen der Klassifikationsbaum-Methode und geht im Folgenden auf technische Details, sowie den Aufbau und die Verwendung der implementierten und eingesetzten Werkzeuge ein.

2.1 Klassifikationsbaum-Methode

Die Klassifikationsbaum-Methode ist eine weit verbreitete Methode zur Ermittlung von funktionalen Blackbox-Tests, eingeführt von Grochtmann und Grimm [GG93]. Die Blackbox-Testmethode leitet Testfälle aus der Software-Spezifikation ab ohne auf die Implementierung der Applikation Rücksicht zu nehmen.

Um große und komplexe Software-Systeme automatisch zu testen, ist eine große Menge an Testdaten nötig sowie ein gut definierter Testprozess. Die Klassifikationsbaum-Methode geht von einer funktionalen Spezifikation des Testobjekts aus. Die Grundidee hinter dieser Methode ist es, die möglichen Eingabewerte eines Testobjekts aufzuteilen. Aus diesen Eingabewerten erhält man eine Menge von Testfallspezifikationen, welche nach Möglichkeit, keine redundanten Fehlerfälle enthalten. Diese Spezifikationen sind jedoch fehler-sensitiv und decken den gesamten Eingabewertebereich ab. Durch diese methodische Herangehensweise wird sichergestellt, dass die resultierenden Äquivalenzklassen, bzw. Testfallspezifikationen, alle für die Software relevanten Testfälle enthält. Um das zu erreichen wird wie folgt vorgegangen.

Zunächst werden alle Test-relevanten Aspekte des Test Systems in Klassifikationen festgelegt. Diese müssen disjunkt und vollständig sein, so dass sie als Äquivalenzklassen im mathematischen Sinne gelten. Das bedeutet, dass keine Äquivalenzklasse mit einer anderen in Relation

steht, bzw. die Klassifikationen sich nicht überschneiden sowie dass die Menge der Testaspekte genügend ist um das Testobjekt zu beschreiben.

Beispielhaft könnte ein Testobjekt "Objekterkennung", welches das Wurzelelement des Baumes darstellt, mit den beiden Klassifikationen "Farbe" und "Form" beschrieben werden (siehe Abbildung 2.1).

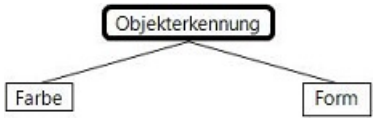

Abbildung 2.1: Testobjekt mit zwei Klassifikationen

Als nächstes werden gültige Eingabewerte für jede Klassifikation gewählt. Diese konkreten Eingabewerte oder auch Charakteristika werden als Klassen bezeichnet (Abb. 2.2). Klassen können nach Vervollständigung des Baumes mit Testfällen in der Testmatrix verbunden werden.

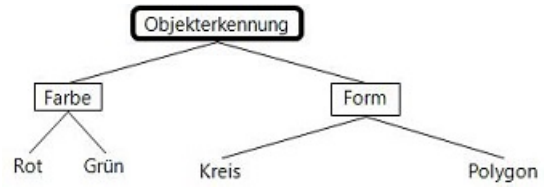

Abbildung 2.2: Testobjekt mit zwei Klassifikationen und den zugehörigen Klassen

Nun werden, wenn nötig, Klassen redefiniert. Das ist dann der Fall, wenn der mögliche Eingabewert noch zu abstrakt ist. Dieser kann dann in weitere Klassifikationen aufgeteilt werden. In dem Beispiel in Abbildung 2.3 hat die Form Polygon zwei Testaspekte, welche sich spezifisch auf Polygone beziehen. Die Regularität sowie die Anzahl der Kanten des Polygons.

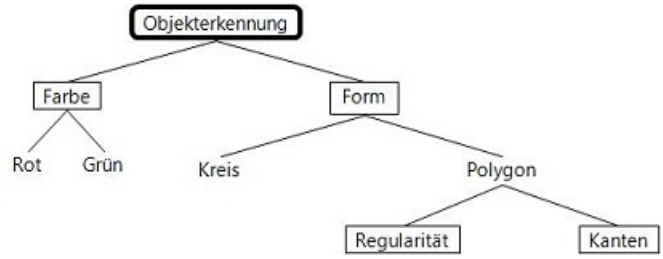

Abbildung 2.3: Die Klasse Polygon in Klassifikationen aufgeteilt

Zum Schluss werden die Testfälle mit Hilfe der Eingabewerte(Klassen) der Klassifikationen definiert. Dies wird in einer Testfall-Matrix realisiert. Der Baum gibt vor, welche Klassen dafür ausgewählt werden können. So wie dieser Klassifikationsbaum modelliert ist, können zum Beispiel die Farben Rot und Grün nicht gleichzeitig gewählt werden. Der für das Beispiel vollständige Klassifikationsbaum ist in Abbildung 2.4 dargestellt. Unter dem Baumdiagramm befindet sich die Testfall Matrix.

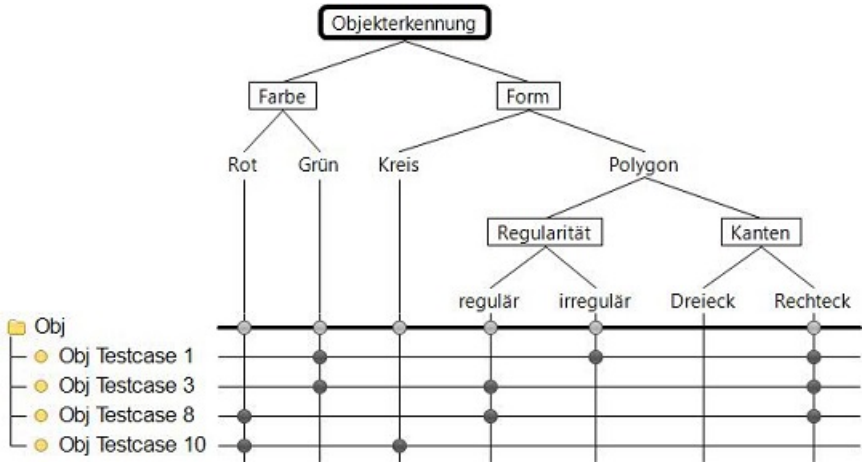

Abbildung 2.4: Vollständiger Klassifikationsbaum

Ein weiteres Element des Klassifikationsbaums ist die Komposition. Diese umfasst mehrere Klassifikationen als Kind-Elemente. Eine Komposition kann allerdings auch weitere Kompositionen als Kind-Elemente besitzen. Kompositionen leiten sich vom Wurzelelement ab. Das Wurzelelement entspricht dem Testobjekt. Kompositionen können auch von Klassifikationen oder Klassen abgeleitet werden. Abbildung 2.5 zeigt in einem grafischen Beispiel den Zusammenhang aller Elemente.

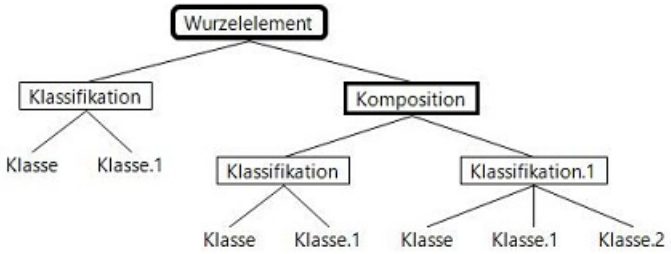

Abbildung 2.5: Ein Klassifikationsbaum mit seinen spezifischen Elementen

2.1.1 Classification Tree Editor XL Professional

Um die für das zu entwickelnde Programm benötigten Klassifikationsbäume erstellen zu können, wurde der Classification Tree Editor XL in der Professional Version von Berner & Mattner verwendet. Entwickelt wurde der Editor von DaimlerChrysler Research Berlin. Der CTE unterstützt die Erstellung und den Entwurf von Klassifikationsbäumen, als auch die Spezifikation der Testfälle in einer Matrixdarstellung.

Der CTE XL ist ein, in Java geschriebener, Graphischer Editor auf Eclipse-Basis, welcher es unter anderem ermöglicht, Testfälle und Testsequenzen zu generieren, logische und numerische Abhängigkeitsregeln festzulegen oder auch Export-Möglichkeiten zu verschiedenen weiterverarbeitenden Programmen wie Matlab bietet. Auf die wichtigsten dieser Funktionen wird später in diesem Unterabschnitt eingegangen.

Die im CTE erstellten Klassifikationsbäume werden im XML-Format gespeichert. Dieses Format wird auch als Ansatzpunkt für das zu entwickelnde Programm verwendet, so dass keine Konvertierung mehr notwendig ist. Ähnliche Ansätze, wie in dieser Arbeit, aus dem CTE konkrete Testdaten zu generieren, wurden bereits veröffentlicht [DDB+05].

Für weiterführende Informationen über Klassifikationbäume und den Classification Tree Editor sowie deren Zusammenwirken, sei auf [GWG95] und die dort referenzierten Veröffentlichungen verwiesen. Auch auf der Website des CTE XL [BM13] finden sich weitere Informationen.

Testfallgenerierung

Der CTE XL Professional ermöglicht es Testfälle aus bereits erstellten Klassifikationsbäumen zu generieren. Durch das Verbinden einzelner Klassifikationen können über logische und numerische Operatoren Verknüpfungen zwischen diesen erstellt werden. Je nach Größe und Umfang des Baumes, kann die Generierung einige Zeit in Anspruch nehmen. Nimmt man das Beispiel aus dem Abschnitt 2.1, entstehen dadurch die zehn $(2 * 5)$ möglichen Testfälle mit einem Tastendruck (Abb. 2.6).

In Abbildung 2.6 sieht man die möglichen Operatoren in der Mitte des Popup-Menüs.

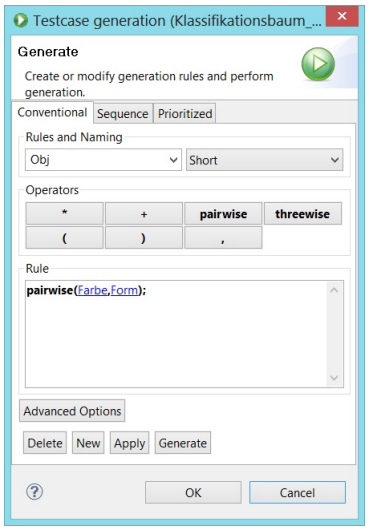

Abbildung 2.6: Testcase-generierung mit dem CTE XL

CTE Regeln

Die logischen und numerischen Abhängigkeitsregeln des Classification Tree Editors können auf Klassifikationsbäume angewendet werden, um beispielsweise Testfälle auszuschließen, welche ohnehin keine möglichen Kombinationen im System Under Test darstellen.

Betrachtet man beispielsweise Abbildung 2.4 und geht davon aus das es im SUT keine roten Polygone geben kann, so kann man dies über eine logische Regel definieren. Eine solche Regel wird im CTE XL, wie in Abbildung 2.7 zu sehen, dargestellt.

Durch das Einsetzten dieser Regel wird bei aktiviertem Regel-Checker nun die Regel direkt beim Erstellen der Testcases angewandt und alle gegen die Regel verstoßenden Testcases ausgeschlossen. Daraus folgt, dass es nur noch sechs Testfälle für diesen Baum gibt.

Numerische Regeln werden ähnlich angewandt und kommen vornehmlich bei Datensätzen mit Ziffern zum Einsatz. Diese unterscheiden sich im Aufbau nur von den hier zum Einsatz kommenden mathematischen Operatoren.

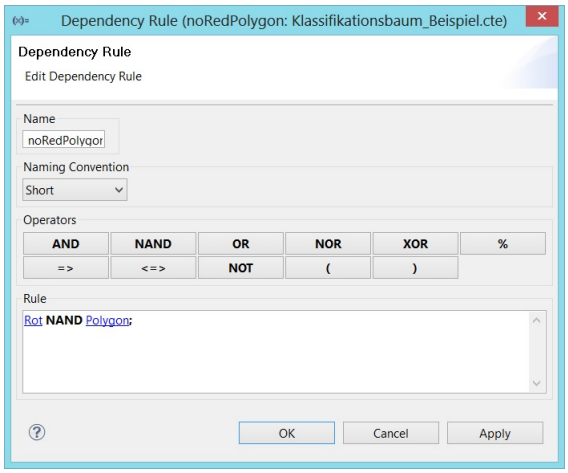

Abbildung 2.7: Erstellen einer logischen Regel im CTE XL

2.2 Selenium WebDriver

Die Selenium WebDriver API, auch Selenium 2 genannt, ist zur Automatisierung von Browsern entwickelt worden und wurde 2004 unter der Apache License 2.0 veröffentlicht. Derzeit arbeitet das World Wide Web Consortium (W3C[1]) in der Arbeitsgruppe *Browser Testing and Tools Working Group* an einem WebDriver-Entwurf, welcher sich stark an dem Selenium WebDriver orientiert und diesen Standardisieren soll [SB13].

Der primäre Fokus von Selenium liegt auf dem automatisierten Testen von Web-Applikationen. Allerdings lassen sich die Funktionen für viele weitere Gebiete einsetzten, wie zum Beispiel der Automatisierung von web-basierten Administrations-Anwendungen.

Eine einfache Methode für weniger umfangreiche und ausführliche Tests bietet Selenium mit einer im Browser integrierten IDE(Integrated development environment). Die IDE ermöglicht das Aufzeichnen von Testfällen im Browser. Dieses Werkzeug eignet sich daher vor allem zur Reproduktion von Fehlerfällen oder der Erstellung von Skripten zur Unterstützung des automatisierten Testens.

Selenium bietet eine reibungslose Anbindung an nahezu alle gängigen Betriebssysteme, Browser, Programmiersprachen sowie Testing-Frameworks. Tabelle 2.1 zeigt eine Übersicht der verwendbaren Testing-Frameworks.

[1]World Wide Web Consortium: http://www.w3.org/

Framework	Selenium IDE	Selenium 2
Bromine	Comes with template to add to IDE	Manipulate browser, check assertions via custom driver
JUnit	**Out-of-the-box code generation**	**Manipulate browser, check assertions via Java driver**
NUnit	Out-of-the-box code generation	Manipulate browser, check assertions via .NET driver
RSpec (Ruby)	Custom code generation template	Manipulate browser, check assertions via Ruby driver
Test::Unit (Ruby)	Out-of-the-box code generation	Manipulate browser, check assertions via Ruby driver
TestNG (Java)	Custom code generation template	Manipulate browser, check assertions via Java driver
unittest (Python)	Out-of-the-box code generation	Manipulate browser, check assertions via Python driver

Tabelle 2.1: Von Selenium unterstützte Testing Frameworks [pla13]

Im zu erstellenden Programm wird Selenium, wie in der Tabelle 2.1 hervorgehoben, in Verbindung mit JUnit 4.11 verwendet. Im nächsten Abschnitt wird auf dieses Framework näher eingegangen.

2.3 JUnit

JUnit[2] ist ein Testing Framework, welches dazu dient reproduzierbare, automatisierte Tests in Java zu schreiben. Es basiert auf der xUnit-Architektur, welche es erlaubt verschiedene Elemente (Units) einer Software isoliert von anderen Programmteilen zu testen. Die Auflösung der getesteten Elemente kann von Funktionen und Methoden sowie Klassen, bis zu ganzen Komponenten reichen. Die Software ist frei unter der Common Public License(CPL) veröffentlicht und im Wesentlichen von Kent Beck und Erich Gamma entwickelt. JUnit ist seit 1998 der Standard für Entwicklertests und in nahezu allen Java-Entwicklungsumgebungen integriert, siehe auch [Wes05].

Der Vorteil einer solchen Architektur ist es, dass sie eine automatisierte Lösung bietet. Es ist nicht nötig zu vermerken welches Ergebnis ein Test liefern sollte. Des Weiteren werden durch ein solches Framework redundante Tests vermieden. Eine Beurteilung und Analyse der Testergebnisse durch den Menschen ist nicht mehr nötig, da die Assert-Methoden aus JUnit diese Beurteilung übernehmen.

Um in JUnit einen Test auszuführen, muss man seit Version 4 im wesentlichen nur zwei Schritte beachten.

1. Die Testmethode muss mit entspechend als Test annotiert werden
2. In dieser Methode muss eine Assert-Methode aufgerufen werden, welche über die Assert-Klasse statisch eingebunden werden.

Dieser Test kann dann direkt über die Entwicklungsumgebung oder einen Konsolenaufruf gestartet werden. Für genauere Informationen über Unit-Testing im allgemeinen und das darauf aufbauende JUnit Framework sei hier auf entsprechende Veröffentlichungen verwiesen, wie [Mye04], [Lin05] und in der JUnit Dokumentation [BGS13].

2.4 Eclipse IDE

Für die Entwicklung der Software wurde das Programmierwerkzeug Eclipse[3] verwendet. Eclipse kann zur Entwicklung von Software verschiedenster Art verwendet werden. In dieser Arbeit

[2] JUnit: http://www.junit.org
[3] Eclipse: http://www.eclipse.org/

wird Eclipse hauptsächlich als integrierte Entwicklungsumgebung für die Programmiersprache Java verwendet. Durch die gebotene Erweiterbarkeit kann es aber auch für viele andere Entwicklungsaufgaben verwendet werden, beispielsweise wurde in dieser Arbeit auch die Versionsverwaltung der Software von Eclipse unterstützt.

Eclipse selbst basiert auf Java-Technik, seit Version 3.0 auf einem sogenannten OSGi-Framework (Open Services Gateway initiative) namens Equinox.

3 Anforderungsanalyse

Das folgende Kapitel geht auf die Ziele der zu programmierenden Software sowie das gegebene Szenario ein. In dieser Analyse werden, über die Definition der Ziele, die Anforderungen hergeleitet und das zugrundeliegende Szenario beleuchtet.

3.1 Ziele der Software

Das primäre Ziel der Software ist es die Zeit und den Aufwand zur Erstellung und Ausführung von Testfällen zu minimieren sowie die Anzahl der Testdatensätze zu reduzieren und trotz dessen voll qualifizierende Testfälle zu generieren.

Kann der Aufwand für die Ausführung von Testfällen für eine Web-Applikation verringert werden, so reduziert sich die Arbeitsbelastung des Testers durch weniger zeitintensive Tests. Wenn gleichzeitig die Erstellung von Testfällen weniger komplex wird, so vermindert dies ebenfalls das Pensum, welches für die Tests benötigt wird.

Dies wird durch die gezielte Klassifizierung von Daten mit Hilfe der Klassifikationsbaum-Methode(siehe auch Abschnitt 2.1) erreicht. Können Daten in Äquivalenzklassen aufgeteilt werden, so wird die Zahl der möglichen Eingabedaten effektiv minimiert. Jedes Element oder Datum wird dann genau einer Äquivalenzklasse zugeordnet. Alle sich in der Äquivalenzklasse befindlichen Elemente können nun als Repräsentant dieser Klasse angesehen werden. Durch diesen klassifizierenden Aufbau der Testfalldaten ist es möglich einige wenige, aber vollkommen qualifizierende, Vertreter einer Klasse zu wählen und nur mit diesen Daten Testfälle zu erstellen. Dieses Verfahren vermindert die Zahl der möglichen Testfälle in hohem Maße.

Zusammenfassend sollen durch die Klassifizierung der Daten sowie einer minimierten Zahl von Inputs und der automatischen Ausführung der daraus resultierenden Testfälle die Ziele erreicht werden.

3.2 Szenario und Anforderungen

Dieser Abschnitt behandelt im Detail das vorliegende Szenario sowie die Anforderungen an die zu entwickelnde Software. Die Software ist in dieser Arbeit so ausgelegt, dass sie nur mit diesem spezifischen Szenario arbeitet. Das Programm wurde speziell zur Testunterstützung für die gegebene Webapplikation entwickelt. Im letzten Kapitel, im Unterpunkt Ausblick, wird darauf eingegangen ob und wie sich die Software erweitern lässt, um auch andere Testobjekte und Szenarien zu unterstützen.

3.2.1 Szenario - VALVESTAR

Das für diese Arbeit verwendete Szenario, bezieht sich auf die Webseite von VALVESTAR[1]. Eine dort verfügbare Webapplikation ermöglicht dem Benutzer die individuelle Zusammenstellung von diversen Sicherheitsventilen der Firma Leser[2].

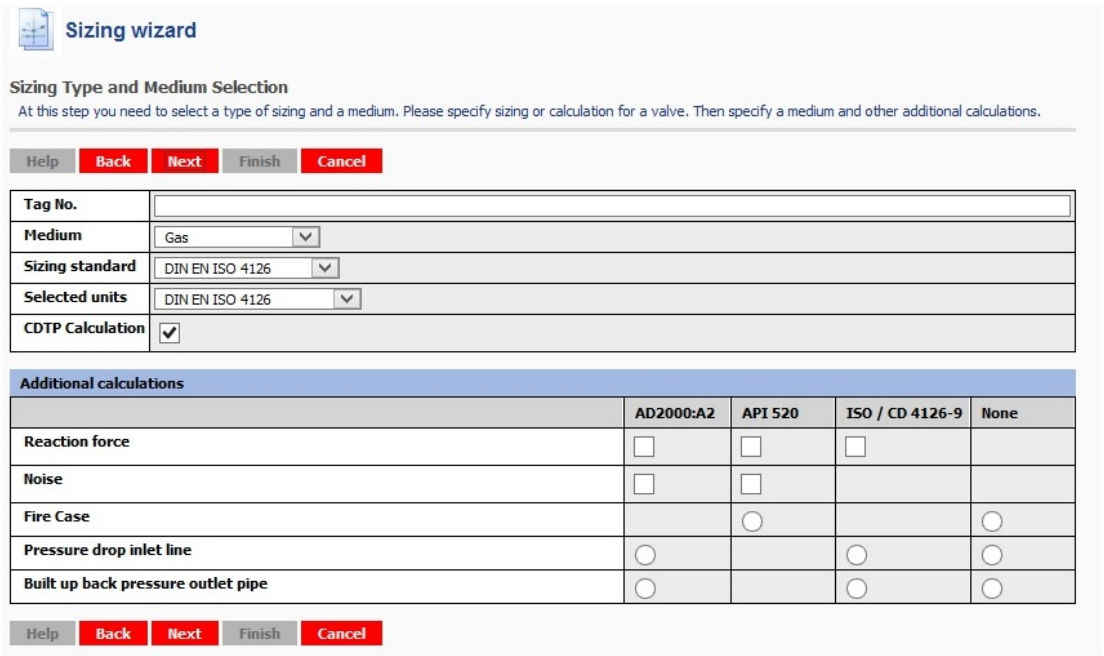

Abbildung 3.1: Auszug der erste Seite des Auslegungs-Wizards zur Erstellung eines Ventils

[1] VALVESTAR: http://www.valvestar.com
[2] Leser: http://www.leser.com

In dieser Webapplikation soll das Auslegungsprogramm, also der Entwurf und die Gestaltung eines Ventils, automatisiert getestet werden. Die Webapplikation ist als ein Wizard (Software-Assistenz Programm) realisiert, welcher den Benutzer bei der Erstellung unterstützt. Über mehrere Seiten wird man bei der Erstellung begleitet bis die Bearbeitung abgeschlossen werden kann. Ein manueller Test des gesamten Verlaufs der Website ist nur sehr mühsam zu bewerkstelligen. Schon zu Beginn des Wizards (Abb. 3.1) steht eine beträchtliche Menge an Eingabemöglichkeiten und daraus resultierende Testfälle zur Auswahl.

3.2.2 Anforderungen

Die Anforderungen an das Programm ergeben sich primär aus den Zielen, welche die Software umsetzen muss und der Art des Szenarios. Da die vorrangigen Ziele der Software einen effizienten und performanten Testablauf anstreben, sind die Anforderungen an erster Stelle daran angelehnt. Die Anzahl der möglichen Eingabedaten muss reduziert werden und der Aufwand um den Wizard zu testen muss minimiert werden.

Das vorliegende Szenario diktiert weitere Anforderungen. Da der Wizard sich aus mehreren Webseiten zusammensetzt sind Modultests eine naheliegende Variante um die Tests aufzubauen. Jede Webseite enthält mehrere minimale Funktionen, wie Eingabe, Auswahl oder Abfrage-Funktionen, welche ideal als Module getestet werden können. Um den Testablauf zu automatisieren, muss sowohl die Ansteuerung der Webapplikation als auch die des Browsers automatisiert werden.

Zusammenfassung der Anforderungen:

- Eine prägnante Reduzierung der Eingabedaten, realisiert durch Klassifikationsbäume und Abhängigkeitsregeln.
- Eine wesentliche Einsparung der Testzeit und des Testaufwandes, im Vergleich zum manuellen Testvorgang.
- Verwendung von Modul-Tests (auch Unit-Tests) für die einzelnen Webseiten des vorliegenden Szenarios.
- Die automatische Ansteuerung des Browsers sowie des System Under Test (SUT), ohne Verwendung von Benutzereingaben in beiden genannten Elementen.

- Es muss eine intuitive, grafische Benutzeroberfläche geschaffen werden, welche dem Anwender eine leichte Bedienung der Software ermöglicht.

4 Implementierung

Dieses Kapitel beschreibt den Aufbau und die Umsetzung der in dieser Arbeit entwickelten Software. Zunächst wird die Architektur der Software beschrieben. Über die wesentlichen Bestandteile des Programms wird nach einem einleitenden Abschnitt im Detail die Anbindung der verwendeten Werkzeuge und Bibliotheken erläutert.

4.1 Architektur

Die Software gliedert sich in drei Komponenten, welche auf die in Kapitel 2 beschriebenen Softwareelemente zugreifen, beziehungsweise diese verwenden. Auf die Komponenten wird in der folgenden Auflistung kurz eingegangen. In den anschließenden Unterabschnitten folgt ein detaillierter Überblick über die Komponenten und deren Zusammenwirken.

CTE

Der CTE-Programmteil analysiert die im CTE erstellten Klassifikationsbäume (XML-Format - Extensible Markup Language) mit Hilfe eines XML Document Object Model (DOM) Parsers, in Abbildung 4.1 als CTEParser dargestellt.

Die Java DOM API zum XML-Parsen arbeitet mit einer XML-Datei als einen Objektgraphen. Der Parser durchläuft das XML Dokument und erstellt die korrespondierenden DOM Objekte. Diese DOM Objekte werden in einer Baumstruktur angeordnet. Danach kann dann die DOM Struktur über die bereitgestellten Methoden durchsucht werden.

In der DOM Struktur werden die Klassifikationen und Klassen des Klassifikationsbaums per Pattern-Matching gefiltert. Die Struktur eines im CTE generierten XML-Baums ist immer gleich, weshalb sich die Verwendung dieser Methode anbietet. Der Quelltextausschnitt 4.1 zeigt einen Teil dieser Filterung.

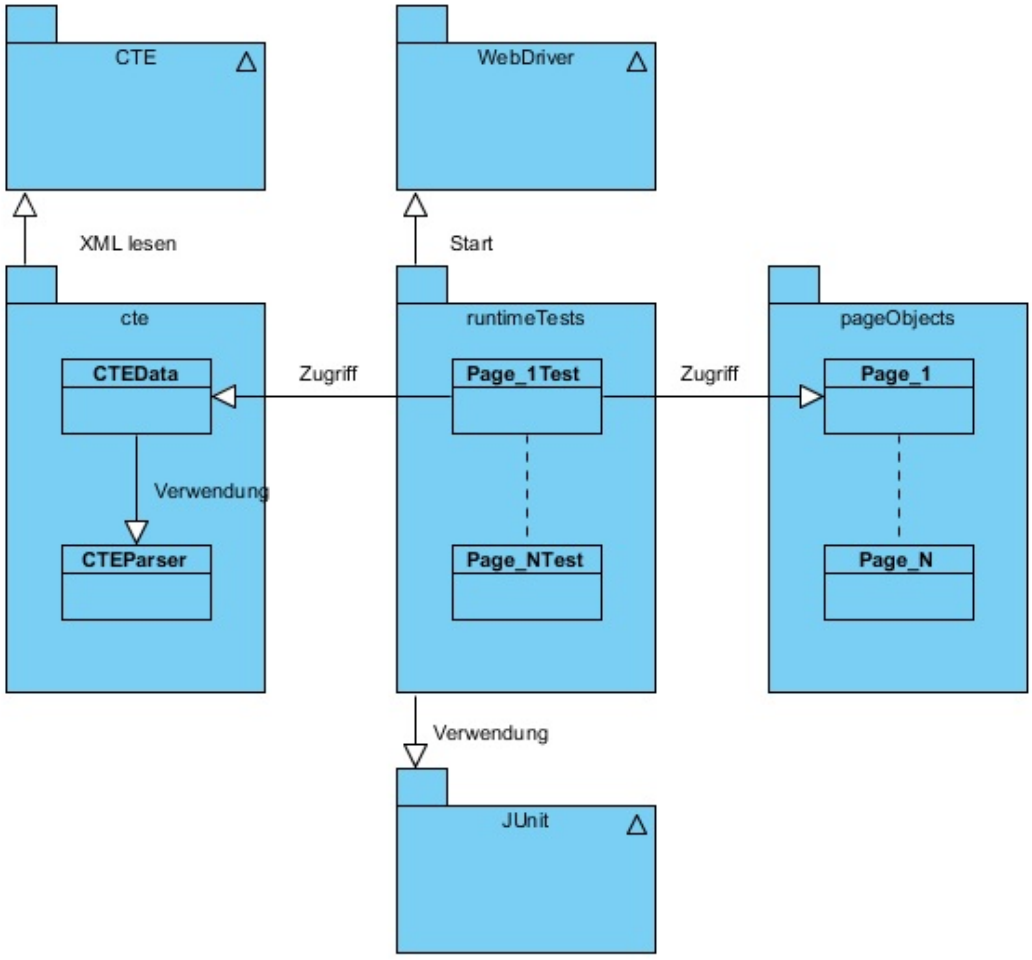

Abbildung 4.1: Der allgemeine Programmaufbau

```
public short acceptNode(Node n) {
    if (n.getNodeType() == Node.ELEMENT_NODE) {

        for (CteObj co : CteObj.values()) {
            if (n.getNodeName().equalsIgnoreCase(co.name()))
                return FILTER_ACCEPT;
        }

        if (n.getNodeName().equalsIgnoreCase("Tree")
                || n.getNodeName().equalsIgnoreCase("Tag")
                || n.getNodeName().equalsIgnoreCase("TestGroup")
                || n.getNodeName().equalsIgnoreCase("Content")
                || n.getNodeName().equalsIgnoreCase("CteObject"))
            return FILTER_SKIP;

    }

    return FILTER_REJECT;
}
```

Listing 4.1: Auschnitt aus dem CTE XML-Objekt Filter

Damit dies gelingt müssen die Klassifikationsbäume einem bestimmten, auf die zu testende Webseite angepassten, Muster folgen. Das Testobjekt (die Webseite) wird im Klassifikationsbaum als Wurzelobjekt dargestellt. Jedes Element einer Webseite, welches eine Eingabe ermöglicht, hat zumeist eine Beschriftung. Diese Beschriftungen müssen verwendet werden und im Klassifikationsbaum als Klassifikationen abgebildet sein. Ist keine Beschriftung vorhanden muss der Name oder die ID des Elements verwendet werden. Diese einheitliche Benennung erleichtert das Suchverfahren, um die entsprechenden Werte der Klassifikationen auf der Webseite verarbeiten zu können. Sind Eingabeelemente in Tabellen oder in separate Bereiche gegliedert, so können diese als Kompositionen dargestellt werden. Diese Kompositionen erhalten als Bezeichnung ebenfalls die Beschriftung, unter welcher sie auf der Webseite aufgeführt sind. Die Klassen des Baumes, welche die eigentlichen Testdaten widerspiegeln, ergeben sich aus den möglichen Eingabedaten der Webseiten-Elemente, respektive der Klassifikationen.

Runtime Tests

Im Package runtimeTests liegen die JUnit Tests, welche mit Hilfe des WebDrivers auf die Webapplikation zugreifen. Im ersten Test, welcher die erste Seite des zu testenden Wizards behandelt, wird über die Startseite zur gewünschten Seite navigiert. Dort wird über Assertions jede Eingabemöglichkeit geprüft. Dieser Vorgang wird auf allen zu testenden Seiten wiederholt. Diese Tests sind der Übersicht halber in Abbildung 4.1 als Page_XTest dargestellt, wobei des X für die Nummerierung der Tests steht. Jeder Page_XTest steht dabei für eine Seite des Wizards und beinhaltet Testmethoden für jede Test-relevante Funktion der Webseite. Der Testablauf wird detailliert in Abschnitt 4.1.3 erläutert und dargestellt.

Ein Ausschnitt des Quelltextes, des Tests für die zweite Webseite, ist im Listing 4.2 zu sehen.

```
1  @BeforeClass
2  public static void setUpBeforeClass() throws Exception {
3      tree = Cast.as(Tree.class, FileHandler.loadObjectsFromFile());
4      FileHandler.closeFile();
5      scp.setTree(tree);
6  }
7
8  @Before
9  public void setUp() throws Exception {
10     scp.setMarks(marks);
11     assumeFalse("Skipped...", newPage.isFailed());
12 }
13
14 @Test
15 public void testInputMaxAllowableWorkingPresure() {
16     assertNotNull(scp.inputMaxAllowableWorkingPresure());
17 }
```

Listing 4.2: Test-Quelltext Ausschnitt

Die Before-Methoden dienen zur Initialisierung von sogenannten Test fixtures. Test fixtures sind Objekte, welche alle Tests verwenden können und gleichbleibend sind. In diesem Beispiel sind dies der Baum und die Markierungen. Der Baum entspricht dem Klassifikationsbaum der aktuellen Webseite und die Markierungen sind die Klassen (Werte), welche der aktuelle Testfall annimmt. In der mit *@Test* annotierten Methode wird das Eingabefeld, *Maximum allowable working pressure,* einer Webseite im Wizard getestet.

Page Objects

Die Page Objects repräsentieren die Webseiten der Webapplikation. Diese Java Klassen sind nach dem Page Object Pattern erstellt worden. Ein Page Object modelliert die Bereiche einer Webseite, mit denen die Tests interagieren, als Objekte. Dadurch wird die Menge an Testcode reduziert und erleichtert das Anpassen der Tests, falls sich etwas an dem User Interface oder am Seitenaufbau ändert.

Page Objects stellen die Funktionalitäten, welche von einer bestimmten Webseite zur Verfügung gestellt werden, dar. Diese Page Objects beinhalten, als einzige Elemente des Programms, die Struktur des HTML-Inhalts (Hypertext Markup Language) einer Webseite bzw. dem modellierten Teil einer Webseite. Page Objects bieten folglich nur die Services nach außen an und umschließen sowohl die Details als auch die Mechanismen der Webseite. Die Implementation der jeweiligen Services der Webseite sind für den Test nicht von Belang.

Des Weiteren wird durch die Page Objects der Weg des Users durch die Applikation dargestellt. Jedes Page Object liefert, nach Ausführung einer Aktion, die darauf folgende Webseite zurück. Durch diesen Aufbau kann jedweder Verlauf den der User durch die Applikation nimmt abgebildet werden. Es bedeutet außerdem, dass wenn sich die Beziehungen zwischen den Webseiten ändert die angesetzten Tests automatisch nicht kompiliert werden können. Daraus folgt, dass man in der Lage ist zu beurteilen welche Tests fehlschlagen werden ohne diese ausführen zu müssen, wenn die entsprechende Änderung der Beziehung der Seiten in den Page Objects aufgenommen wurde.

In Listing 4.3 ist ein Auszug eines Page Objects abgebildet. Dieses Page Object bezieht sich auf die erste relevante Webseite des Wizards. Der spezifische Quelltextausschnitt zeigt die Methode zur Wahl eines Mediums (Gas, Dampf, Flüssigkeit) für ein Ventil. Dies geschieht über eine Auswahlbox auf der Webseite.

```
1  public SizingTypeAndMediumSelectionPage selectMedium() {
2      WebElement mediumDDL = findSelect("Medium");
3      if (mediumDDL == null) {
4          return null;
5      }
6      for (WebElement webElement : new Select(mediumDDL)
7              .getOptions()) {
8          if (medium.equalsIgnoreCase(webElement.getText())) {
9              jscript.select(mediumDDLid, webElement
10                     .getAttribute("value"));
11             return this;
12         }
13     }
14     return null;
15 }
```

Listing 4.3: Eine Methode eines Page Objects

Zunächst wird in Zeile 2 die Auswahlbox für die Medien auf der Webseite gesucht. In Zeile 6 wird über die einzelnen Elemente der Auswahlbox iteriert und diese mit dem Medium des aktuellen Testfalls abgeglichen (Zeile 8). Wurde das entsprechende Medium gefunden, wird dieses über eine Javascript-Funktion auf der Website gesetzt (Zeilen 9 und 10). Zeile 11 zeigt, dass der Rückgabewert die aktuelle Seite zurückliefert. Dies geschieht, da auf diese spezielle Funktion keine neue Seite folgt, sondern auf der aktuellen Webseite verblieben wird.

Der Aufbau der Page Objects und ihrer Methoden erlaubt sogenanntes Method chaining. Die Bedeutung dessen wird im Quelltextausschnitt 4.4 deutlich.

```
1 stams.selectMedium().selectSizingStandard()
2                 .checkCdtpBox().checkReactionForce()
3                 .selectRadioPressureDrop().clickNextButton();
```

Listing 4.4: Method-Chaining

Method chaining ermöglicht beliebig viele Methodenaufrufe aneinander zu reihen und kann so den möglichen Verlauf den ein User über die Webseite nimmt veranschaulichen.

4.1.1 CTE Anbindung

Die Anbindung des Classification Tree Editors erfolgt wie bereits erwähnt über die XML-Dateien, welche der Editor zum Anlegen der Klassifikationsbäume verwendet. Die Dateien welche getestet werden, werden im User Interface per Dateiauswahl bestimmt. Wie in Abschnitt 4.1 erläutert wird die XML-Datei analysiert und durchsucht.

Die gefundenen Kompositionen, Klassifikationen und Klassen sowie die ebenfalls im XML enthaltenen Testcases, werden in Java-Objekten gespeichert. Die aus dem Klassifikationsbaum entstandenen Java-Objekte werden, wie sich anbietet, in einer Baum-Struktur abgelegt. Die Testcases werden in einer Liste gespeichert.

4.1.2 Datenverwaltung

Die in Abschnitt 4.1.1 beschriebenen Strukturen, in welchen die Objekte abgelegt sind, werden serialisiert und in einer Datei gespeichert. So müssen bereits analysierte CTE-XML-Dateien nicht erneut durchlaufen werden. Diese Dateien werden zur Laufzeit des Programms deserialisiert und von den JUnit-Test Klassen geladen. Der Quelltextausschnitt 4.5 zeigt das einfache Serialisieren über einen File und einen Object Output Stream. Zuvor wird per regulärem Ausdruck die Dateiendung der XML-Datei (.cte) in die Programm-eigene Endung geändert und abgespeichert. In den Zeilen 7 und 8 wird erst die Liste über die Testfälle (*tcList*) und danach der Klassifikationsbaum (*cteTree*) in die Datei geschrieben.

```
1 cttcfile = cteFile.getName().replaceAll("\\..*", "") + (".cttc");
2 if (new File(cttcfile).exists() && !overwrite) {
3     throw new FileAlreadyExistsException(cttcfile);
4 }
5 fout = new FileOutputStream(cttcfile);
6 oos = new ObjectOutputStream(fout);
7 oos.writeObject(tcList);
8 oos.writeObject(cteTree);
```

Listing 4.5: Serialisierung der Daten

Der Daten-Verlauf und die daraus resultierende Änderung der Datentypen wird in Abbildung 4.2 dargestellt.

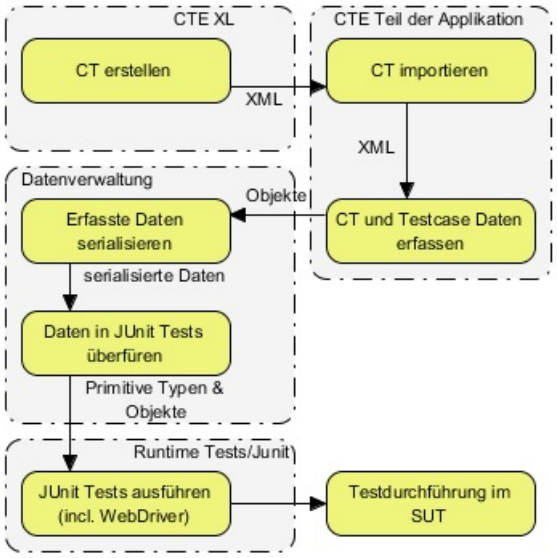

Abbildung 4.2: Datenverlauf vom CTE zum System Under Test (SUT)

4.1.3 Testablauf

Die JUnit-Test Klassen werden parametrisiert ausgeführt. Durch eine Parametrisierung ist es möglich, alle Testmethoden einer Testklasse automatisch mehrmals hintereinander mit unterschiedlichen Testdaten anzusteuern. Das bedeutet, dass pro Klassifikationsbaum-Testcase alle JUnit-Tests einer Testklasse, mit den jeweiligen Daten des aktuellen Testcases, ausgeführt werden. Dies wird wiederholt bis alle Testcases abgearbeitet wurden. Die Parameter für die jeweiligen Tests bezieht die Klasse aus den zuvor gespeicherten Dateien, auf welche im vorigen Abschnitt eingegangen wird.

Zunächst wird in einer Parameter-Methode das Testcase-Array geladen. Über dieses wird bei jedem Durchlauf eines Testcases iteriert, bis alles Testcases abgearbeitet sind. Ein veranschaulichter Ablauf der JUnit-Tests wird in Abbildung 4.3 gezeigt.

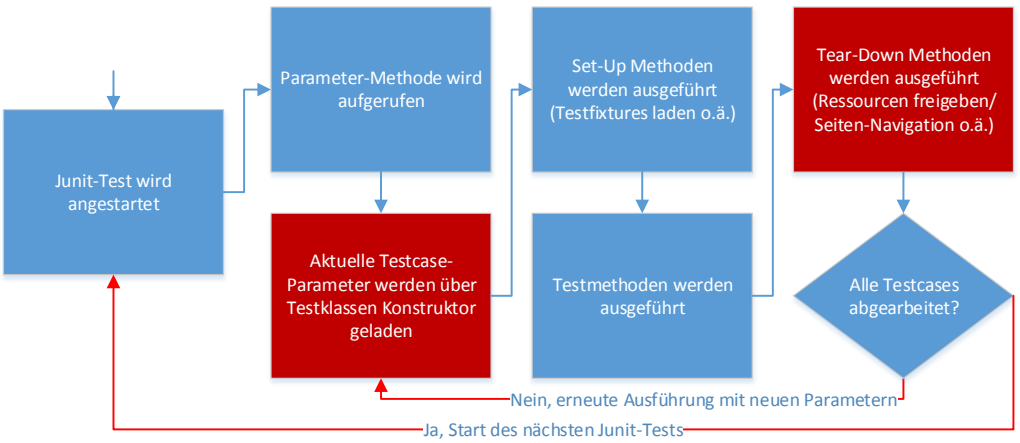

Abbildung 4.3: Junit-Testablauf

In Listing 4.6 ist ein Ausschnitt der Parameter-Methode der ersten Webseite des Wizards abgebildet. In Zeile 1 wird die Methode als Parameter-Methode annotiert, um sie dem Compiler erkenntlich zu machen. In den Zeilen 3 und 4, wird das Testcase-Array aus der zuvor erstellten Datei geladen. Die for-Schleife lädt die spezifischen Parameter des aktuellen Testfalls. Diese werden als Liste von der Methode zurückgegeben. Die Elemente dieser Liste, werden als Parameter an den Konstruktor der Testklasse übergeben.

```
1  @Parameters(name = "{index}: {1}")
2  public static List<Object[]> data() {
3      ArrayList<CteTestCase> tcs = Cast.as(ArrayList.class,
4              FileHandler.loadObjectsFromFile());
5      Object[][] data = new Object[tcs.size()][tcs.size()];
6      for (Iterator<CteTestCase> iterator = tcs.iterator(); iterator
7              .hasNext();) {
8          data[i] = iterator.next().asArray();
9      }
10     return Arrays.asList(data);
11 }
```

Listing 4.6: Beispiel einer Parameter-Methode

Im Detail betrachtet, gibt es für jede zu testende Webseite eine JUnit-Testklasse. Die erste Webseite, welche getestet werden soll, führt alle in ihr enthaltenen Testmethoden aus. Diesen Methoden werden die Daten des ersten vorliegenden Testfalls übergeben. Ist der erste Testfall

abgearbeitet, wird überprüft, ob noch weitere, auf die aktuelle Seite folgende Webseiten getestet werden sollen. Ein Testfall ist dann abgearbeitet, wenn alle JUnit-Testmethoden einer Klasse erfolgreich abgearbeitet wurden, was bedeutet, dass es keine Fehler gab oder falsche Eingabedaten vorliegen.

Wenn nach dem ersten Testcase festgestellt wird, dass noch weitere verlinkte Seiten getestet werden sollen, wird zunächst auf diese Seite navigiert. Danach ruft eine spezielle Methode, welche nach jedem erfolgreich abgeschlossenem Testcase aufgerufen wird, den nächsten JUnit-Test auf. In diesem wird die ganze Prozedur wiederholt.

Durch diese Vorgehensweise wird sichergestellt, dass jede mögliche Testcase-Kombination über zwei oder mehr Seiten ausgeführt wird. Es entsteht ein Baum-artiges durchlaufen aller Testfälle, siehe auch Abb. 4.4.

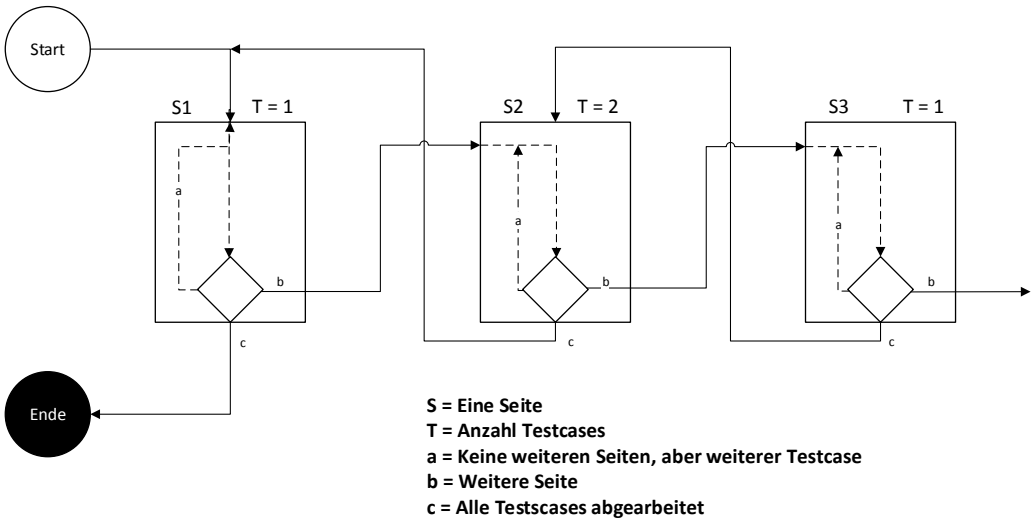

Abbildung 4.4: Verkettung der JUnit-Tests für die Webseiten

In dieser Abbildung ist veranschaulicht, wie die Testfälle durchlaufen werden. Vom Startpunkt (Teststart) ausgehend, wird die erste Webseite (S1) angesteuert. Die gestrichelte Linie mit Pfeil nach unten stellt die Durchführung der Testmethoden für diese Seite dar. Darauf folgt eine Drei-Wege-Entscheidung. In der Legende der Abbildung 4.4 werden die Bedeutungen der verschiedenen Wege erläutert. In der Abbildung sind für die Webseiten beispielhaft eine Menge von Testfällen angegeben.

Im folgenden wird anhand der vorgegeben Daten der Abbildung 4.4 ein Testdurchlauf exemplarisch dargestellt. Dafür wird eine Kurznotation verwendet. Beispielgebend steht $\mathbf{S}X\mathbf{T}X$ für **S**eite Nr. X mit **T**estcase Nr. X.

$$S1T1 \rightarrow S2T1 \rightarrow S3T1 \rightarrow S2T2 \rightarrow S3T1 \rightarrow Ende \tag{4.1}$$

Wählt man Werte mit mehr Testdurchläufen wird die Baumstruktur, welche entsteht, noch deutlicher.

Beispiel: S1 mit T=2, S2 mit T=2, S3 mit T=2

Daraus ergibt sich folgender Testablauf:

$$\begin{aligned} &S1T1 \rightarrow S2T1 \rightarrow S3T1 \rightarrow S3T2 \rightarrow S2T2 \rightarrow S3T1 \rightarrow S3T2 \rightarrow \\ &S1T2 \rightarrow S2T1 \rightarrow S3T1 \rightarrow S3T2 \rightarrow S2T2 \rightarrow S3T1 \rightarrow S3T2 \rightarrow Ende \end{aligned} \tag{4.2}$$

Anhand der Testfolgen 4.1 und 4.2 lassen sich nun die sich ergebenden Bäume darstellen, siehe Abbildungen 4.5 und 4.6.

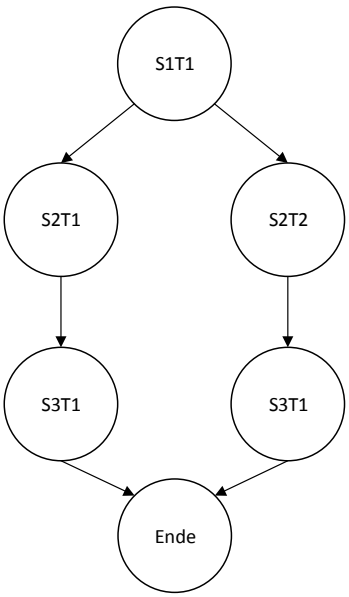

Abbildung 4.5: Der Testdurchlauf 4.1 visualisiert in einer Baumstruktur

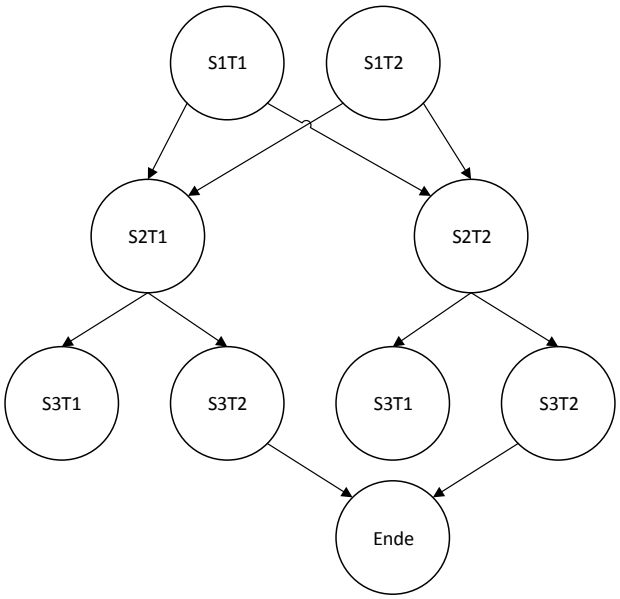

Abbildung 4.6: Testdurchlauf 4.2 in eine Baumstruktur überführt

Diese Bäume stellen gut den Zusammenhang zwischen den Testcases dar, allerdings fehlt ihnen die Rückverbindungen, wie sie in Abb. 4.4 dargestellt sind. Das ist in diesem Fall aber zu vernachlässigen, da in Abbildung 4.4 nicht deutlich wird, woher sich die Verzweigungen an den Baum-Knoten ergeben. Da das Programm aber alle verfügbaren Testcases in beliebiger Reihenfolge ausführen kann, wird dies in der Baumdarstellung verdeutlicht.

4.1.4 Ergebnisdokumentation

Die VALVESTAR-Webseite bietet eine automatisch generierte Dokumentation eines jeden Projektes an. Ein Projekt stellt im Falle des hier vorgestellten Programms eine Testreihe dar. In einem Projekt sind die einzelnen Testfälle angeordnet. Die Dokumentation ist unmittelbar nach Ausführung einer Testreihe verfügbar.

Es gibt zwei Arten von Dokumentationen welche aufgerufen werden können. Die eine ist für das gesamte Projekt. Diese kann als Excel-Tabelle gespeichert werden. Die andere ist für die jeweiligen Testfälle, respektive die erstellten Ventile. Die Dokumentation der Ventile ist in einer Webansicht sowie als PDF (Portable Document Format) verfügbar.

Jede erstellte Testfall-Dokumentation enthält alle relevanten Informationen über das erstellte Ventil sowie Details über die Korrektheit der eingegebenen Werte und gewählten Materialien. In den erstellten Dokumentationen können des Weiteren Notizen vermerkt, die Ventile korrigiert oder gelöscht werden.

Vom erstellten Programm wird zusätzlich ein Log erstellt, welches die Daten über den Verlauf der Testcases enthält. Dies geschieht im Speziellen über Erfolg oder Misserfolg der einzelnen Testfälle.

4.2 Design

Das grafische Design wurde parallel zum architektonischen Design erstellt.

4.2.1 User Interface

Es wurde ein schlichtes Design für die grafische Oberfläche (siehe Abbildung 4.7) gewählt, um den Aufwand für diesen Teil des Programms zu minimieren und trotz dessen eine für den User einfach zu bedienende und ansprechende Oberfläche zu schaffen.

Über die Menüleiste können die XML Dateien des CTE XL in das Programm geladen werden. Diese werden dann im *Files*-Fenster angezeigt. Wählt man eine geladenen XML-Datei aus, wird diese umgewandelt und in ein für die Unit-Tests verwendbares Format gebracht. Die umgewandelten Dateien stehen im *Testcases*-Fenster.

Nach der Eingabe eines Benutzernamens und des zugehörigen Passworts für die VALVESTAR-Webseite können die gewählten Unit-Tests, über den Button *Start Unit Test*, gestartet werden.

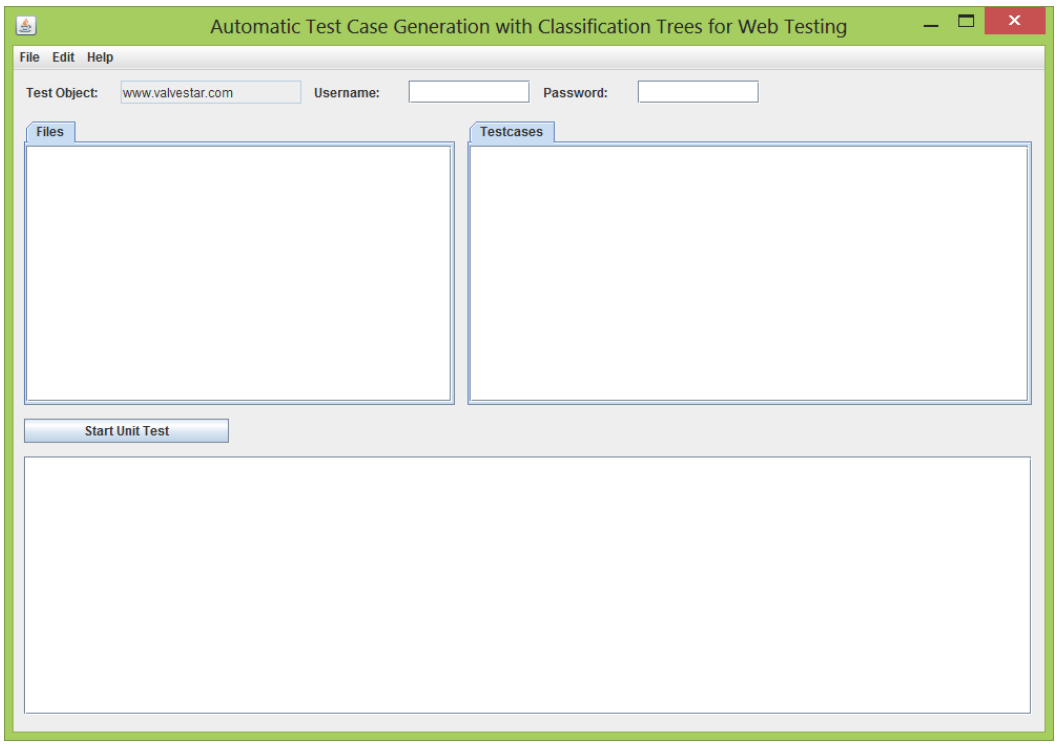

Abbildung 4.7: Das User Interface des Programms.

Nach drücken des *Start Unit Test*-Button öffnet sich der Firefox-Webbrowser, dieser öffnet die VALVESTAR-Webseite und es werden die Login-Informationen eingetragen. Im Anschluss wird der Wizard gestartet und die Tests werden abgearbeitet.

5 Fallstudie

In diesem Kapitel wird die Anwendung des realisierten Programms auf das vorliegende Testobjekt erläutert und die Ergebnisse der Tests ausgewertet.

5.1 Testsystem

In allen Tests, wie auch bei der Erstellung des Programms, wurde folgendes System verwendet:

HP EliteBook 8740w

CPU	Intel Core i7 M640 @ 2.80 Ghz
RAM	4,00 GB DDR3
Festplatte	Seagate Momentus 7200.4 250GB, SATA II (ST9250410AS)
Betriebssystem	Windows 8, 64-Bit
Java Version	1.7.0_10
Firefox Version	17.0

Tabelle 5.1: Testsystem Spezifikationen

Für das Programm werden nur Java-Versionen ab 1.7.0_10 aufwärts unterstützt. Als Browser wird nur der Mozilla Firefox in Version 17.0 oder niedrigere unterstützt, da zu Beginn der Programmierung die Selenium Bibliothek in der Version 2.25 eingebunden wurde, welche nur diese Versionen von Mozilla Firefox unterstützt.

5.2 Testobjekt

Wie in Kapitel 3 beschrieben ist das Testobjekt eine Webapplikation zur Erstellung von Sicherheitsventilen, auf der VALVESTAR Website. Die Applikation ist als Wizard realisiert.

Als Testbasis wurde von der Firma Leser ein VALVESTAR Musterauslegungsfall mit einem Auslegungsstandard vorgegeben, siehe Abbildung 5.1.

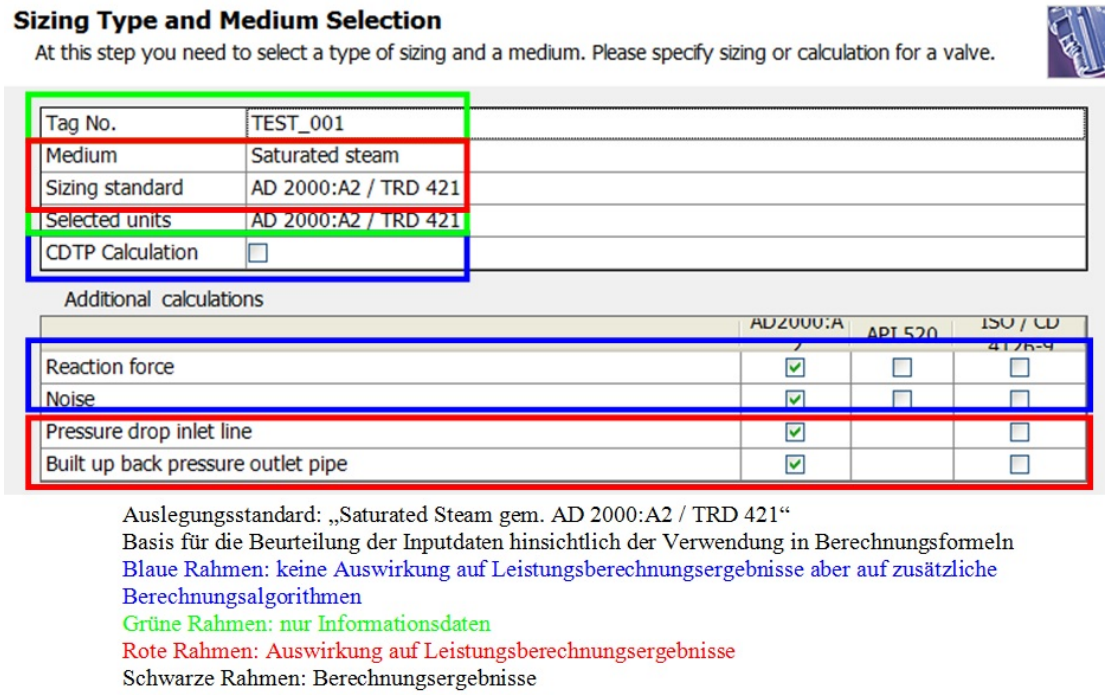

Abbildung 5.1: Auslegungsstandard

Wie in der Legende in Abbildung 5.1 zu sehen, sind nicht alle Daten für mögliche Testfälle relevant. Eine solche Vorlage wurde von der Firma Leser für alle auf diesen Standard folgenden Wizard-Seiten bereitgestellt. Durch die Markierung der Berechnungsrelevanten Testdaten (blaue und rote Rahmen) konnte schon in der Implementierung die Anzahl der abzubildenden Funktionen des Wizards eingegrenzt und teilweise ganze Seiten vernachlässigt werden.

Beispielhaft werden im Folgenden die Ersten beiden Seiten des Wizards untersucht.

Abbildung 5.1 zeigt die erste zu testende Seite des Wizards, *Sizing Type and Medium Selection*. Diese Seite wurde mit Hilfe des CTE XL als Klassifikationsbaum modelliert und zunächst wurden, ohne Angabe von Abhängigkeitsregeln (Kapitel 2 Abschnit 2.1.1), alle kombinatorisch möglichen Testcases generiert. Das Ergebnis liegt bei über 42.000 Testcases, welche abzuarbeiten wären. Durch den gezielten Einsatz der Regeln des CTE, welche bestimmte Abhängigkeiten zwischen einzelnen Inputs berücksichtigen, konnten die Testcases um ca. 33.000 auf unter 13.000 verringert werden. Im Voraus hat die Klassifikationsbaum-Methode die Testfälle stark

reduziert, durch die Verwendung von Abhängigkeitsregeln konnte der Testaufwand abermals um 60% verringert werden.

Auf der zweiten Webseite (Abbildung 5.2) des Wizards wird die Mächtigkeit der Klassifikationsbaum-Methode noch deutlicher. Diese Seite verfügt über mehrere Eingabefelder, in welchen es möglich ist, Daten in Spannen von mehreren hunderttausenden einzugeben. Das Feld *Maximum allowable working pressure* erlaubt sogar die Eingabe unendlich hoher Werte, womit unendlich viele Testfälle für diese Seite entstehen würden. Durch die Anwendung der Klassifikationsbaum-Methode für diese Seite konnte die Anzahl der Testfälle von einer unbegrenzten Menge auf 1152 Testfälle verdichtet werden.

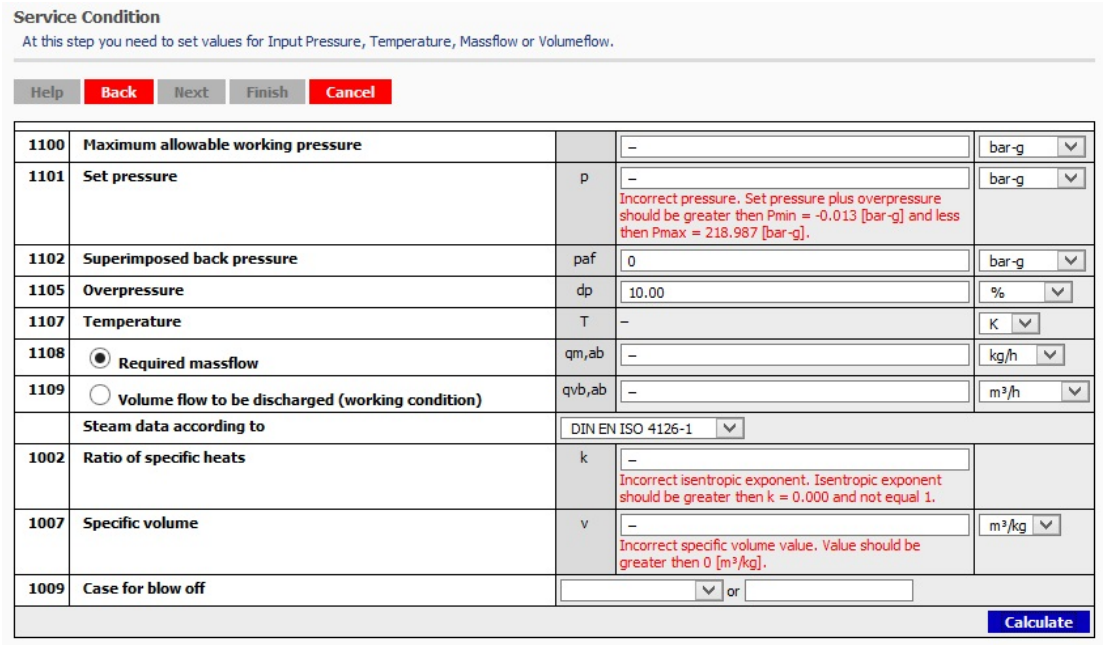

Abbildung 5.2: Service Condition Webseite

In Abbildung 5.3 wird deutlich wie dies gelingt. Über das Extrahieren der am meisten relevanten Werte, zumeist Grenzwerte, kommen die Äquivalenzklassen mit einer minimalen Anzahl an Testdaten aus. So hat das Feld *Maximum allowable working pressure* nun anstatt unendlich vieler Werte nur noch vier repräsentative Werte.

Berücksichtigt man nun noch die Abhängigkeiten zwischen den einzelnen Feldern, respektive der Äquivalenzklassen, so kann man mit den bereits vorgestellten Abhängigkeitsregelen des

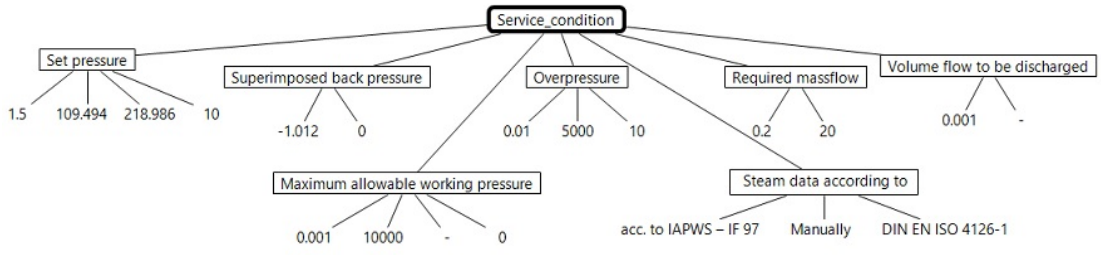

Abbildung 5.3: Klassifikationsbaum der Service Condition Webseite

CTE XL die Anzahl der Testfälle nochmal auf 216 senken. Das bedeuten eine Testfallreduktion von 81.25%.

Da diese beiden Seiten von einander abhängen, erhöht sich bei einem zusammenhängenden Test über beide Seiten die Anzahl der Testfälle allerdings enorm ($13.000 * 216 = 2.808.000$ Testfälle).

5.3 Testzeiten

Um die Geschwindigkeit der Ausführung der Tests zu ermitteln, wurde jeweils zu Beginn und nach Beendigung des Wizards die aktuelle Systemzeit gemessen.

Auf dem Testsystem wurden mehrere Tests ausgeführt, um eine Testzeitanalyse zu erstellen. Für 10 Tests, welche sich auf eine Test-Seite beziehen, benötigt das Programm im Schnitt 10.3 Sekunden. Hochgerechnet auf 13.000 Tests (siehe Abschnitt 5.2) entspricht das in etwa 3.75 Stunden. Ein Test über zwei Seiten mit zwei Testscases für die erste Seite und fünf für die zweite Seite, dauert durchschnittlich 208 Sekunden. Das bedeutet, dass 10 Tests mit einem Seitenübergang im Wizard, im Vergleich zu 10 Tests auf einer Seite, ca. 20 mal soviel Zeit benötigen.

Die Testzeiten erhöhen sich also in hohem Maße wenn mehrere Seiten getestet werden. Hauptsächlich wird dies durch die Latenz zum Server verursacht. Bei Tests über zwei Seiten erhöht sich der Abarbeitungsumfang. Es wird zusätzlich jedes mal der Wizard abgeschlossen, ein neues Sizing erstellt und wieder auf die zu testende Seite navigiert. Dieser Vorgang veranschlagt zwischen 5 und 20 Sekunden. Diese Spanne liegt so weit auseinander, da die Zeit, die benötigt

wird um wieder zur Testseite zu kommen, sich mit jedem Testcase um ein bis vier Sekunden erhöht.

5.3.1 Evaluation der Testzeiten

Die Testzeiten zeigen, dass einzelne Testobjekte sehr gut getestet werden können. Mehrere aufeinander folgende Testobjekte erhöhen die Zeiten erheblich. Dies schließt allerdings nicht ein, dass andere Testobjekte den gleichen Effekt haben. Daten zu dieser These können in einer weiterführenden Arbeit oder in einer Programmerweiterung erarbeitet werden.

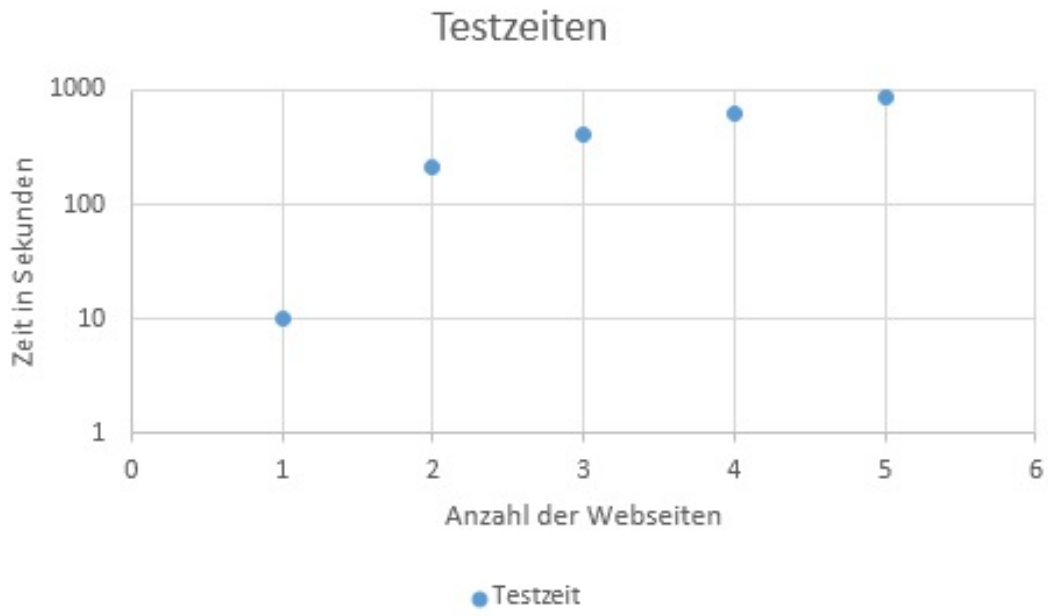

Abbildung 5.4: Dauer der Abarbeitung von zehn Testfällen über ein bzw. mehrere Webseiten

In Abbildung 5.4 ist zu erkennen, dass sich die Zeit, die ein Testdurchlauf benötigt, erhöht je mehr Webseiten abgearbeitet werden. Die sprunghafte Erhöhung von einer auf zwei getestete Seiten ist im Seitenwechsel sowie dem, bereits erwähnten, erhöhten Abarbeitungsumfang begründet. Die Erhöhung reguliert sich allerdings bei mehr als zwei Seiten, so dass nur die vermehrten Seitenübergänge ins Gewicht fallen nicht aber der Abarbeitungsumfang.

5.4 Effektivität der Methoden

Die Klassifikationsbaum-Methode zeigt eine sehr hohe Wirksamkeit bei der Reduzierung der Testfälle sowie der Testdaten. Nicht nur für große Webanwendungen ist die enorme Minderung von Testdaten interessant. Durch diese Möglichkeiten lassen sich Test schneller durchführen und haben zudem eine hohe Abdeckung der Daten.

Für bestimmte Klassifikationen ist es schwer passende Werte zu finden. Wenn eine Klassifikation keine Grenzwerte besitzt, können geeignete Werte nur durch heuristische Ansätze erfasst werden. Erfahrungswissen oder Daumenregeln können hier eingesetzt werden. Bei Abhängigkeiten zwischen einzelnen oder mehreren Klassifikationen verhält es sich ähnlich. Gibt es solche Abhängigkeiten müssen diese berücksichtigt werden. Dafür werden Regeln festgehalten. Bei gezielter Anwendung können diese Regeln den Testaufwand weiter erfolgreich drücken.

6 Fazit und Ausblick

Das abschließende Kapitel, Fazit und Ausblick, fasst die Arbeit zusammen und erörtert die Erkenntnisse, welche gewonnen werden konnten. Das Weiteren gibt es einen Einblick in die möglichen Erweiterungen und die Ausbaufähigkeit des Programms.

6.1 Fazit

In der vorliegenden Arbeit wurde ein Programm zur automatisierten Testdurchführung für eine spezielle Webanwendung entwickelt. Die einzelnen, relevanten, Teile der Webanwendung wurden auf Klassifikationsbäume abgebildet. Alle Testfälle und deren Testdaten konnten mit Hilfe dieser Klassifikationsbäume erarbeitet werden. Die Anwendung wurde in Java realisiert, wodurch diese auf allen Plattformen, welche eine virtuelle Maschine starten können, verwendbar ist.

Durch die Verwendung des Page Object Patterns konnten die Webseiten bzw. die verwendeten Teile der Webseiten voll funktional in Java abgebildet werden. Das Pattern beschreibt alle relevanten, zu testenden Elemente einer Webseite und stellt deren Funktionen zur Verfügung.

Die Ansteuerung des Browsers und der damit verbundenen Webapplikation erfolgt über den Selenium WebDriver. Der WebDriver ist ein Framework zur Automation von Browsern. Über dieses Framework kann der Browser nativ, wie von einem User, gesteuert werden. Das Ansteuern der Webseiten über den Browser erfolgt im Sourcecode über eine zur Verfügung gestellte API.

Auf der Grundlage der Page Objects, in Verbindung mit dem WebDriver Konzept, konnten JUnit Tests implementiert werden, welche die Korrektheit der Eingabedaten sowie die Funktionen der Webseiten testen.

Mit dem entwickelten Programm kann das in dieser Arbeit vorgegebene Testobjekt angesteuert und vollkommen automatisiert angesteuert und getestet werden. Das Testobjekt ist die von der Firma Leser entwickelte Applikation VALVESTAR, zur webbasierten Erstellung von Sicherheitsventilen.

Die Testergebnisse zeigen, dass durch den Einsatz von Klassifikationsbäumen effizient Testzeit eingespart werden kann. Es entstehen weniger Testfälle und der Testaufwand wird verringert.

6.2 Ausblick

Mit mehr Einblick in die Normen und Richtlinien für Ventile, wäre es möglich Heuristiken zu entwickeln welche den Testaufwand weiter verringern. Diese Heuristiken müssten auf die Klassifikationsbäume angewendet werden, so dass sich die Menge der Testdaten reduziert.

Eine weitere mögliche Erweiterung der Arbeit wäre die volle Automatisierung und generische Anwendung auf beliebige Webseiten. Der Testaufwand für Webapplikationen steigt zunehmend, nicht erst seit der erweiterten Nutzung von mobilen Geräten. Ein Programm, welches durch Klassifikation beschränkte aber vollkommen aussagekräftige Testdaten liefert und diese Automatisch auf eine Webapplikation anwendet, kann in hohem Maße die Testzeit und den Testaufwand senken und somit die Qualität von Webanwendungen erhöhen.

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Glossar

API Application programming interface, Seite 2

BAF Browser Automation Framework, Seite 1

CPL Common Public License, Seite 13

CTE XL Classification Tree Editor XL, Seite 2

CTM Classification Tree Method, Seite 2

DOM Document Object Model, Seite 19

HTML Hypertext Markup Language, Seite 21

IDE Integrated development environment, Seite 10

OSGi Open Services Gateway initiative, Seite 14

PDF Portable Document Format, Seite 29

SUT System Under Test, Seite 17

W3C World Wide Web Consortium, Seite 10

XML Extensible Markup Language, Seite 19

Literaturverzeichnis

[BGS13] Kent Beck, Erich Gamma, and David Saff. JUnit documentation. http://junit.sourceforge.net/doc/, 2013. Zugriffsdatum: 26.04.2013.

[BM13] Berner and Mattner. CTE XL Professional: Grafischer Editor für Klassifikationsbäume. http://www.cte-xl-professional.com/de/cte-xl-professional/index.html, 2013. Zugriffsdatum: 25.04.2013.

[CZVO11] Shauvik Roy Choudhary, Dan Zhao, Husayn Versee, and Alessandro Orso. Water: Web application test repair. In *Proceedings of the First International Workshop on End-to-End Test Script Engineering*, ETSE '11, pages 24–29, New York, NY, USA, 2011. ACM.

[DDB+05] Zhen Ru Dai, Peter H. Deussen, Maik Busch, Laurette Pianta Lacmene, Titus Ngwangwen, Jens Herrmann, and Michael Schmidt. Automatic test data generation for TTCN-3 using CTE. 2005.

[GG93] Matthias Grochtmann and Klaus Grimm. Classification trees for partition testing. *Software Testing, Verification and Reliability*, 3(2):63–82, 1993.

[GWG95] Matthias Grochtmann, Joachim Wegener, and Klaus Grimm. Test case design using classification trees and the classification-tree editor CTE. In *Proceedings of Quality Week*, volume 95, page 30, 1995.

[HAO09] William G.J. Halfond, Saswat Anand, and Alessandro Orso. Precise interface identification to improve testing and analysis of web applications. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, ISSTA '09, pages 285–296, New York, NY, USA, 2009. ACM.

[Lin05] Johannes Link. *Softwaretests mit JUnit: Techniken der testgetriebenen Entwicklung*. dpunkt-Verl., Heidelberg, 2., überarb. und erw. aufl. edition, 2005.

[Mye04] Glenford J. Myers. *The art of software testing*. John Wiley & Sons, Hoboken, N.J, 2nd ed. edition, 2004.

[pla13] Platforms supported by selenium. http://docs.seleniumhq.org/about/platforms.jsp, 2013. Zugriffsdatum: 26.04.2013.

[SB13] Simon Stewart and David Burns. WebDriver W3C working draft. http://www.w3.org/TR/2012/WD-webdriver-20120710/ - Work in progress, March 2013. Zugriffsdatum: 04.04.2013.

[Wes05] Frank Westphal. *Testgetriebene Entwicklung mit JUnit und FIT*. dpunkt.verlag, [s.l.], 1. aufl. edition, 2005.

Danksagung

Ich möchte mich hier bei meiner Familie bedanken. Ihr habt mich während meines gesamten, bisherigen Studiums sehr unterstützt. **D a n k e**.

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

Hamburg, 17. Mai 2013 Benjamin Burchard