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. ——–

Eingereicht am: 30. April 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 Teställen 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 evalued

# Inhaltsverzeichnis

# 1 Einleitung

## 1.1 Motivation

[TODO: cite REF REF] Solides Testen im Web wird zunehmend zu einem der wichtigsten Bereiche in der Software Entwicklung sowie im Software Engineering. Um die Softwarequalität von Websites sicherzustellen, ist es essentiell anspruchsvolle Programme für die Testautomation zu entwickeln.

Das designen von Testfällen ist 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.

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 MS Excel oder Entscheidungstabellen.

Die Klassifikationsbaum-Methode ist eine effiziente Testmethode für funktionales testen, welche die Möglichkeit bietet, den kompletten Eingabebereich eines Testobjekts in unabhängige Äquivalenzklassen aufzuteilen.

## 1.2 Zielsetzung

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

Die Klassifikationsbäume werden mit dem Programm Classification Tree Editor XL Professional von Berner & Mattner[1] erstellt. Abbildung 1.1 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 Programm geladen werden und dort in verwendbare Java-Objekte gewandelt werden.

Mit Hilfe der Selenium WebDriver API 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.

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

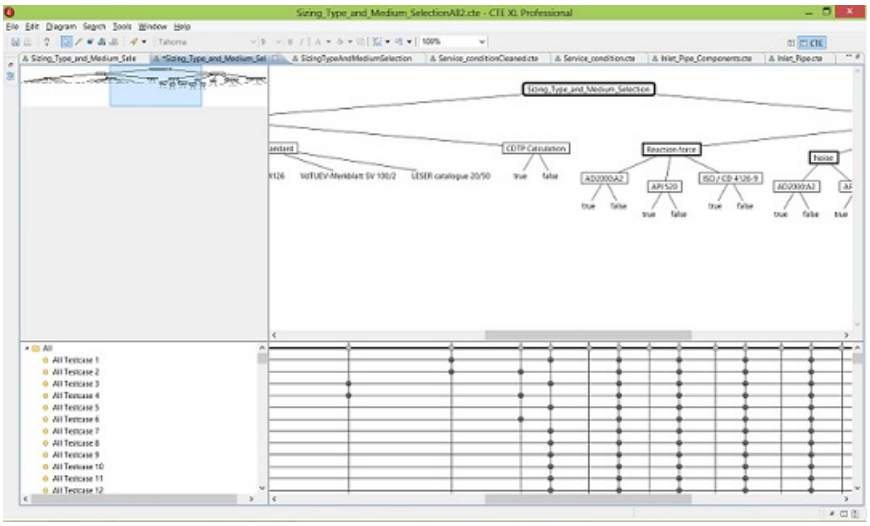

Abbildung 1.1: Classification Tree Editor XL

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.

[TODO: more more more, and ref]

[1] www.berner-mattner.com

## 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 Test Objekt.

**Kapitel 6**

Im Kapitel sechs wird ein Fazit gezogen und ein Ausblick gegeben.

[TODO: you know what]

# 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 Tools. 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 siehe [GG93]. Die Blackbox-Testmethode leitet Testfälle aus der Software-Spezifikation ab ohne auf die Implementierung 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 die jedoch fehler-sensitiv sind und den gesamten Eingabewertebereich abdecken. Durch diese methodische Herangehensweise wird sichergestellt das 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 das keine Äquivalenzklasse mit einer anderen in Relation

steht, bzw. die Klassifikationen sich nicht überschneiden sowie das 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 "Farbeünd "Form"beschrieben werden (siehe Abbildung 2.1).

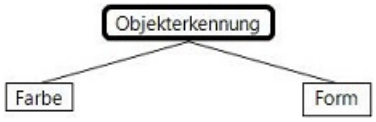


Abbildung 2.1: Test Objekt mit zwei Klassifikationen

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.

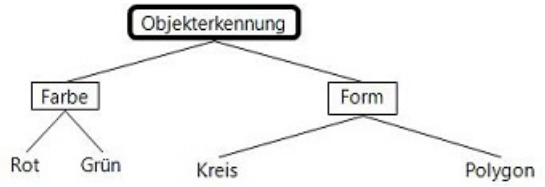


Abbildung 2.2: Test Objekt mit zwei Klassifikationen und den zugehörigen Klassen

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.

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.

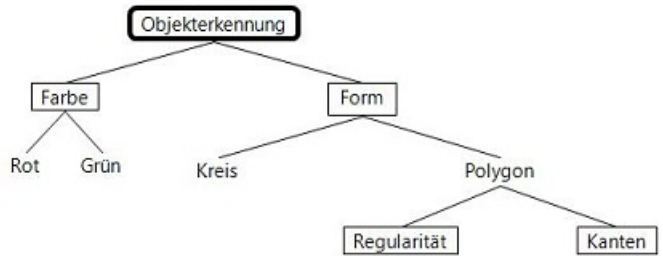


Abbildung 2.3: Die Klasse Polygon wurde in Klassifikationen aufgeteilt

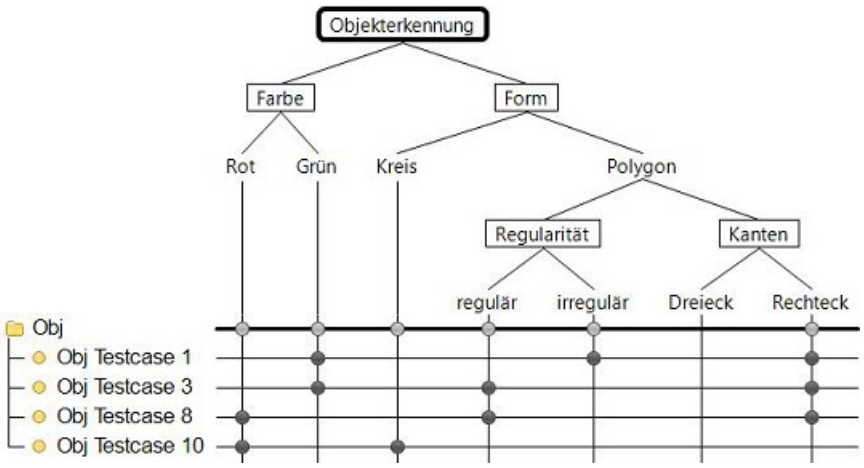


Abbildung 2.4: Vollständiger Klassifikationsbaum

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, welches das Testobjekt darstellt, ab sowie von Klassifikationen oder Klassen. Abbildung 2.5 zeigt in einem grafischen Beispiel den Zusammenhang aller Elemente.

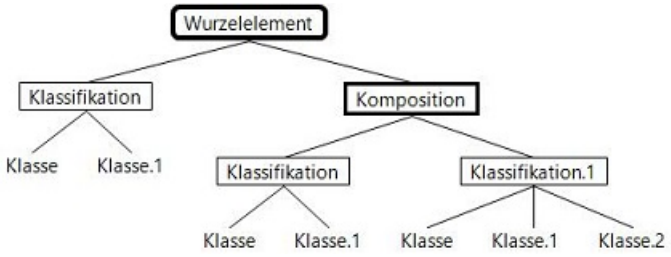


Abbildung 2.5: Ein Klassifikationsbaum mit seinen spezifischen Elementen

### 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.

### Testfallgenerierung

Der CTE XL ermöglicht es, in der Professional Version, 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).

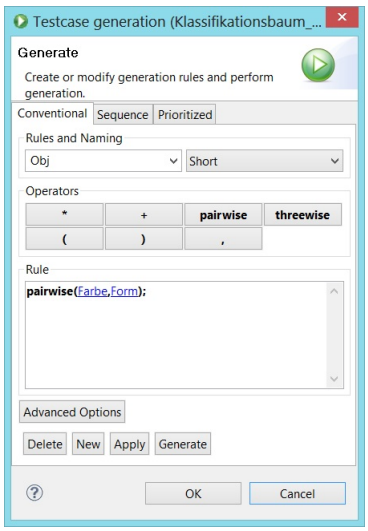


Abbildung 2.6: Testcase-generierung mit dem CTE XL

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

### 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.

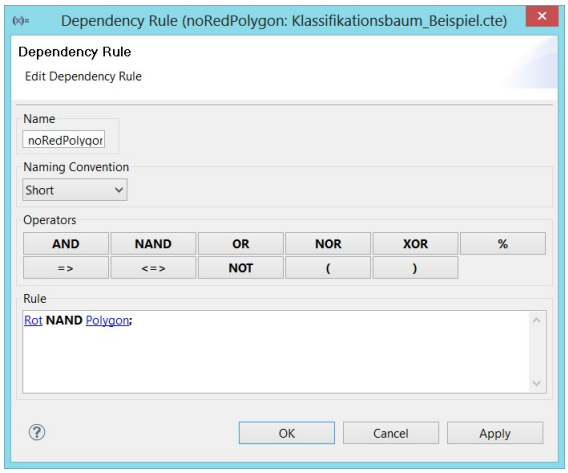


Abbildung 2.7: Erstellen einer logischen Regel im CTE XL

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.

## 2.2 Selenium WebDriver

Die Selenium WebDriver API, auch Selenium 2 genannt, ist zur Automation von Browsern entwickelt worden und wurde 2004 unter der Apache License 2.0 veröffentlicht. Derzeit arbeitet das World Wide Web Consortium (W3C) 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. Die IDE ermöglicht das Aufzeichnen von Testfällen im

| 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

Browser. Dieses Werkzeug eignet sich daher vor allem zur Reproduktion von Fehlerfällen oder der Erstellung 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.

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 diesen Framework näher eingegangen.

## 2.3 JUnit

JUnit[1] 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.

[TODO: @Before/@After Konzept SET UP TEAR DOWN] [TODO: @Rule Konzept]

[1] www.junit.org

# 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 reduzieren sowie den 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, so wie einer minimierten Zahl von Inputs und der automatischen Ausführung der daraus resultierenden Testcases 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. [TODO: say more about szenario and anforderungen]

### 3.2.1 Szenario - VALVESTAR

Das für diese Arbeit verwendete Szenario bezieht sich auf die Website www.valvestar.com. Die dort verfügbare Webapplikation ermöglicht dem Benutzer die individuelle Zusammenstellung von diversen Sicherheitsventilen der Firma Leser[1].

In dieser Webapplikation soll das Auslegungsprogramm, also der Entwurf und die Gestaltung eines Ventils, automatisiert getestet werden. Die Webapplikation ist über einen Wizard 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.

Die erste Seite dient hier als Beispiel wie mit allen Seiten verfahren wurde. Auf der Seite *Sizing Type and Medium Selection* wurde mit Hilfe des CTE XL als Klassifikationsbaum modelliert und ohne Angabe von Abhängigkeitsregeln wurden alle möglichen Testcases generiert. Das Ergebnis liegt bei über 42.000 Testcases welche abzuarbeiten wären. Durch den gezielten Einsatz dieser Regeln, welche bestimmte Abhängigkeiten zwischen einzelnen Inputs berücksichtigen, konnten die Testcases um ca. 33.000 auf unter 13.000 verringert werden. Im Voraus hat die Klassifikationsbaummethode die Testfälle stark reduziert, durch die Verwendung von Abhängigkeitsregeln konnte der Testaufwand abermals um knapp 70% verringert werden.

### 3.2.2 Anforderungen

[TODO: inital section to introduce anforderungen] Zu den Anforderungen gehören:

- Eine prägnante Reduzierung der Eingabedaten, realisiert durch Klassifikationsbäume und Abhängigkeitsregeln.

[1] www.leser.com

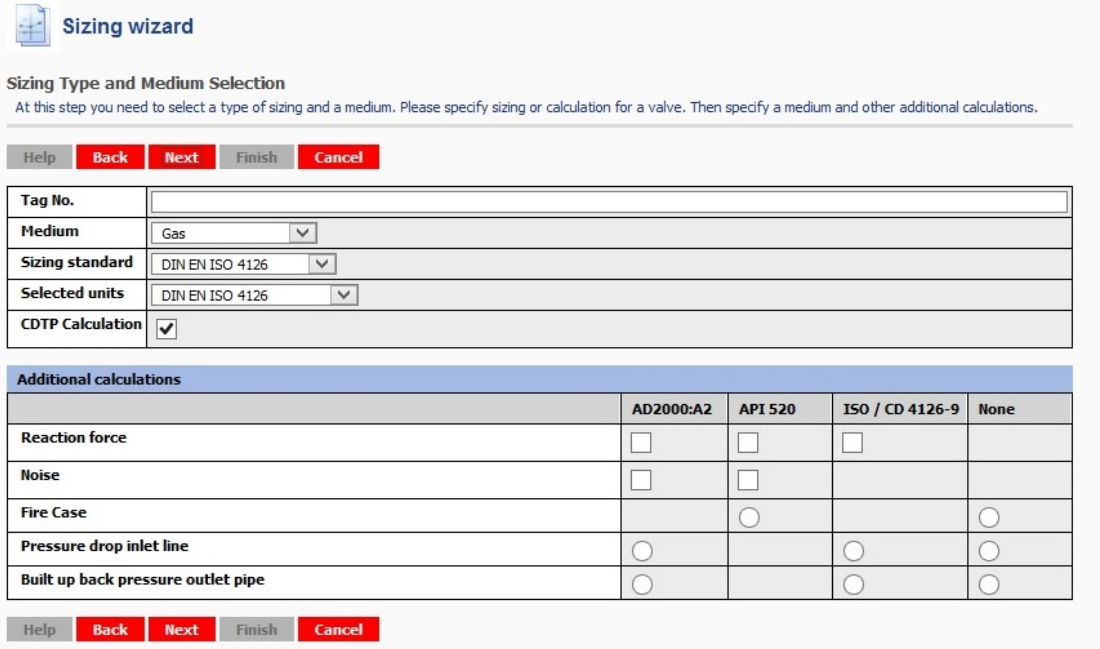


Abbildung 3.1: Auszug der erste Seite des Auslegungs Wizards zur Erstellung eines Ventils

- Eine wesentlich Einsparung der Testzeit und des Testaufwandes, im Vergleich zum manuellen Testvorgang.
- Die automatische Ansteuerung des Browsers sowie des System Under Test(SUT), ohne Verwendung Benutzereingaben in beiden genannten Elementen.
- Es muss eine intuitive Grafische Benutzeroberfläche geschaffen werden, welche dem Anwender eine leichte Bedienung der Software ermöglicht.

## 3.3 Plattformen

[TODO: Erklärung der Verwendung der Plattformen welche in Kapitel 2 beschrieben wurden??]

# 4 Implementierung

Dieses Kapitel beschreibt den Aufbau und die Umsetzung der in dieser Arbeit entwickelten Software. [TODO: more to say? i'm sure...]

## 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 XML-Dateien mit Hilfe eines XML Document Object Model 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, daher bietet sich diese Methode an.

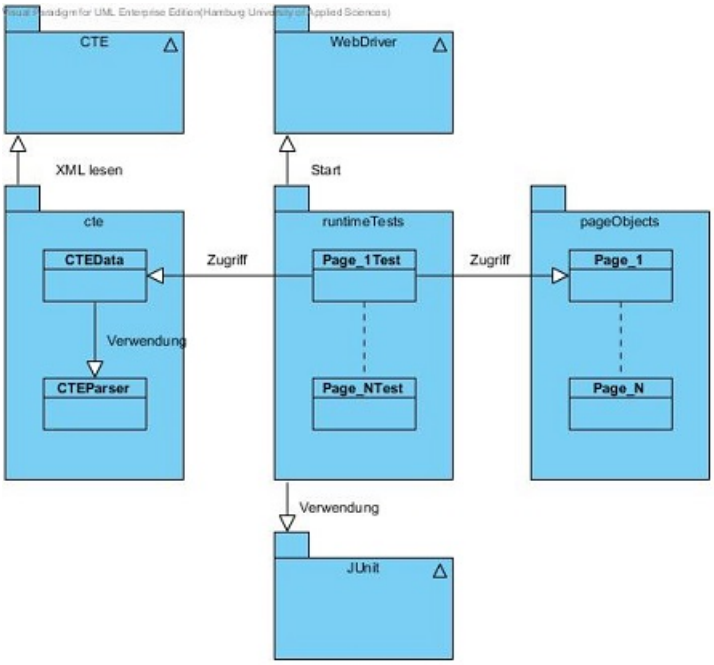


Abbildung 4.1: Der allgemeine Programmaufbau

### 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.

### 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 einer Webseite bzw. dem modellierten Teil einer Webseite. Page Objects bieten folglich nur die Services nach außen an und umschließen die Details sowie Mechanismen der Webseite.

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

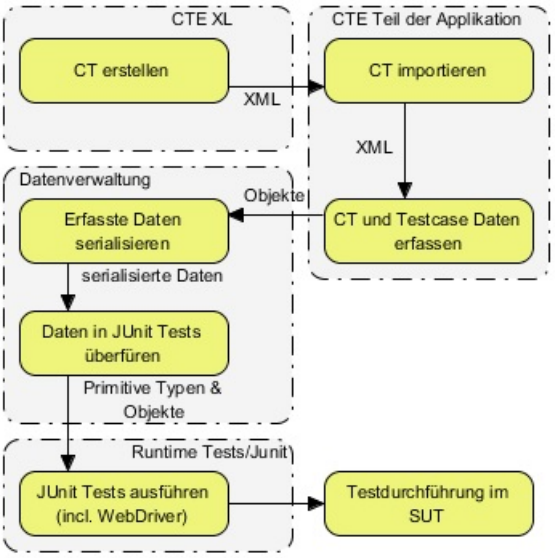


Abbildung 4.2: Datenverlauf vom CTE zum System Under Test(SUT)

### 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.

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

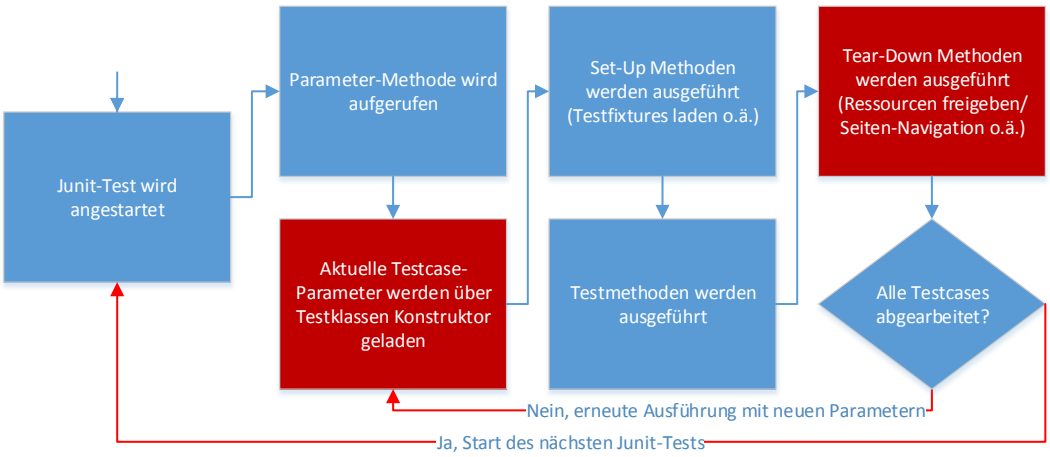


Abbildung 4.3: Junit-Testablauf

erfolgreich abgearbeitet wurden, was bedeutet das es keine Fehler oder falsche Eingabedaten vorliegen.

Wenn nach dem ersten Testcase festgestellt wird das noch weitere verlinkte Seiten getestet werden sollen, wird zunächst auf diese Seite navigiert. Danach ruft eine spezielle Methode( [TODO: ref to chapter2->junit] ), 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 das jede mögliche Testcase-Kombination über zwei oder mehr Seiten ausgeführt wird. Es entsteht ein Baum-Artiges durchlaufen aller Testfälle, siehe auch 4.4.

In dieser Abbildung sieht man sehr detailliert wie die Testfälle durchlaufen werden. Vom Startpunkt ausgehend, wird die erste Webseite(S1) angesteuert. Die gestrichelte Linie 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.

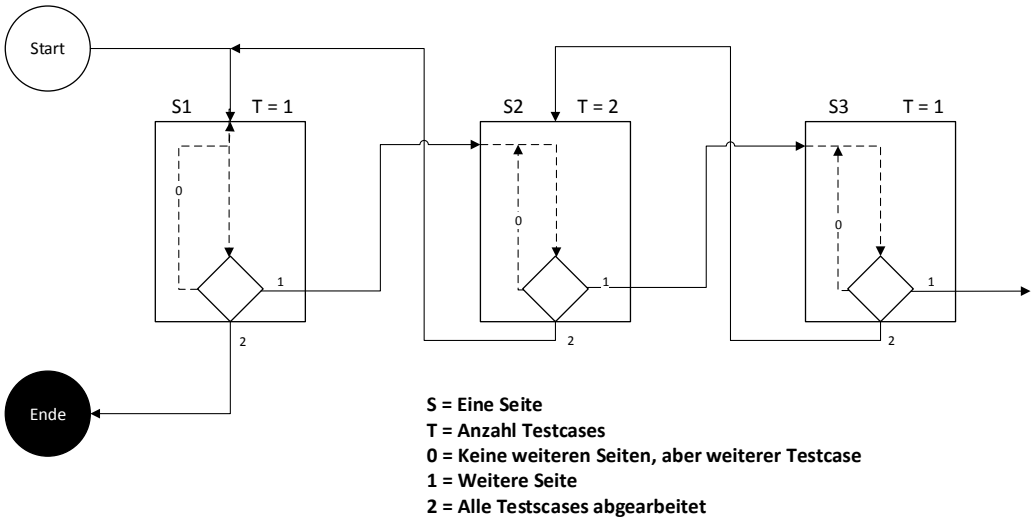


Abbildung 4.4: Verkettung der JUnit-Tests für die Webseiten

Im folgenden wird anhand der vorgegeben Daten der Abbildung ein Testdurchlauf exemplarisch dargestellt. Dafür wird eine Kurznotation verwendet. Beispielgebend steht $SXTX$ für Seite Nr. X mit Testcase 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 Bäume die diese ergeben darstellen, siehe Abbildungen 4.5 und 4.6.

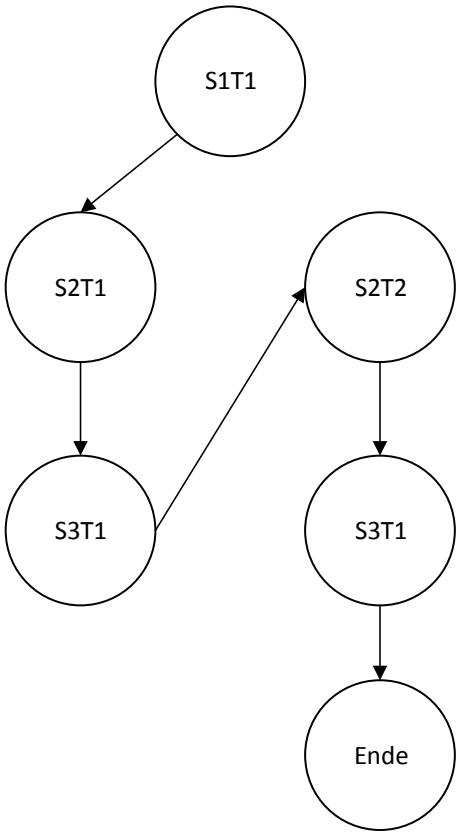


Abbildung 4.5: Der Testdurchlauf 4.1 visualisiert in einer Baumstruktur.

Diese Bäume stellen gut den Zusammenhang zwischen den Testcases dar, allerdings fehlt ihnen die Rückverbindung 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 die Verzweigungen an den Knoten ergeben. Da das Programm aber alle verfügbaren Testcases in beliebiger Reihenfolge ausführen kann, wird dies in der Baumdarstellung verdeutlicht.

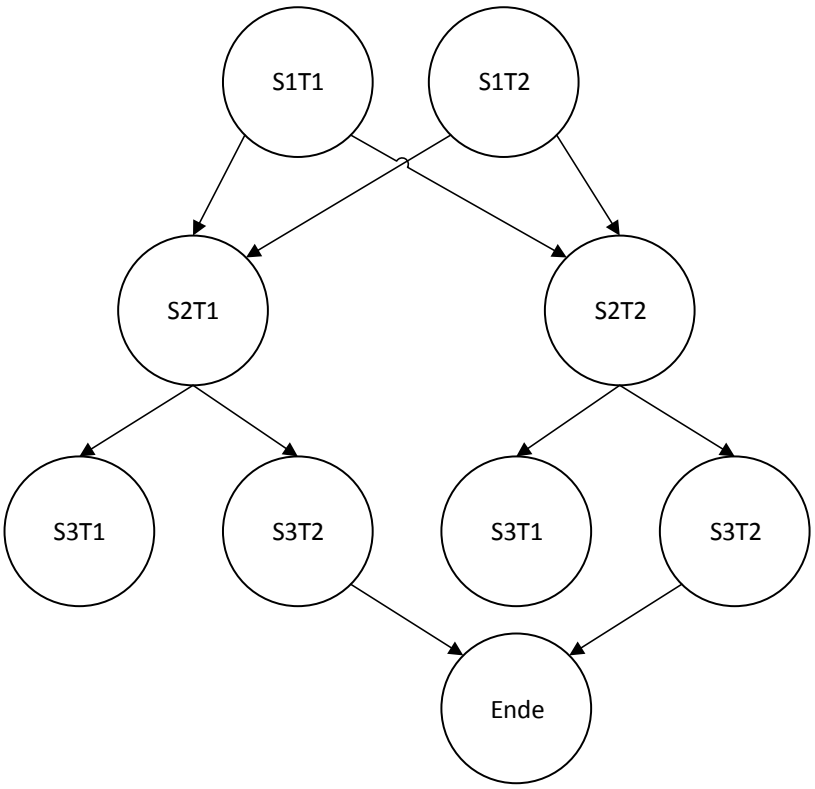


Abbildung 4.6: Testdurchlauf 4.2 in eine Baumstruktur überführt.

### 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 erste ist für das gesamte Projekt, diese kann als Excel-Tabelle gespeichert werden. Die zweite ist für die jeweiligen Testfälle, respektive die erstellten Ventile. Die Dokumentation der Ventile ist in einer Webansicht sowie als PDF 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 mit Notizen versehen, die Ventile korrigiert oder gelöscht werden.

Des Weiteren wird ein Log erstellt, welches die Daten über den Verlauf der Testcases enthält. Speziell über Erfolg oder Misserfolg der einzelnen Testfälle. [TODO: Doku auf page, Log in datei]

## 4.2 Design

Das Grafische Design wurde parallel zum architektonischen Design erstellt.

### 4.2.1 User Interface

### 4.2.2 Zusammenspiel

# 5 Fallstudie

In diesem Kapitel wird die Anwendung des Programms auf das vorliegende Test Objekt erläutert.

## 5.1 Test System

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 |

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 geringer unterstützt, da zu Beginn der Programmierung die Selenium Bibliothek in der Version 2.25 eingebunden wurde, welche nur diese Versionen von Mozilla Firefox unterstützt.

[TODO: BSP aus benchmar.tex?]

## 5.2 Test Objekt

### 5.2.1 Test Wizard

## 5.3 Testzeiten

[TODO: Vorgehensweise, Beispiel]

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(3.2.1) 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 das 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 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

## 5.4 Effektivität der Methode

# 6 Fazit und Ausblick

## 6.1 Fazit

## 6.2 Ausblick

HEURISTIKEN - VOLLE AUTOMATISIERUNG - GENERISCHE ANPASSUNG....

See also [DDB+05].

# Abbildungsverzeichnis

# Tabellenverzeichnis

# Literaturverzeichnis


[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.

[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.

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

Hamburg, 30. April 2013 Benjamin Burchard