

Masterarbeit

**Variantenspezifische Abhängigkeitsregeln und
Testfallgenerierung in TESTONA**



BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN

University of Applied Sciences

Fachbereich VI - Technische Informatik - Embedded Systems



BERNER & MATTNER
AN ASSYSTEM COMPANY

Eingereicht am: 6. Februar 2015

Erstprüfer : Prof. Dr. Macos
Zweitprüfer : Prof. Dr. Höfig
Eingereicht von : Matthias Hansert
Matrikelnummer : s791744
Email-Adresse : matthansert@gmail.com

Dankessage

Inhaltsverzeichnis

1	Einleitung	2
2	Aufgabestellung	3
2.1	Anforderungen	3
2.2	Variantenmanagement Software	4
2.3	Neue Anforderungen	5
3	Fachliches Umfeld	6
3.1	TESTONA	6
3.1.1	Klassifikationsbaum-Methode	6
3.1.2	Testfälle und Testfallgenerierung	8
3.1.3	Abhängigkeitsregeln	8
3.2	IBM Rational DOORS	11
3.3	Variantenmanagement	12
3.4	Entwicklungsumgebung und Programmiersprache	13
3.4.1	Eclipse	13
3.4.2	Plug-ins	13
3.4.3	Standard Widget Toolkit (SWT)	14
3.4.4	JFace	15
4	Lösungsansatz	16
4.1	Parameterspeicherung	17
4.2	Visualisierung	19
4.3	Testgültigkeit	20
4.4	Optimierungskriterien	22
4.5	Benutzeroberfläche	23
5	Systementwurf	24
5.1	Variantenmanagement und Parameter	24

<i>INHALTSVERZEICHNIS</i>	<i>III</i>
5.1.1 ResourceSetListener	24
5.1.2 VariantsManager	25
5.1.3 Parameter-Tag	25
5.1.4 ParameterManager	26
5.1.5 AddParameterTagCommand	27
5.1.6 Die DOORS Verbindng	27
5.2 Testfallgenerierung und Optimierung	31
6 Evaluierung	32
6.1 Chances of Failure	32
7 Zusammenfassung und Ausblick	33
A Anhang	36
A.1 CD	36
A.2 code 1	37
A.3 code 2	38
Literatur- und Quellenverzeichnis	39

Kapitel 1

Einleitung

Ziel dieser Masterarbeit ist die Erweiterung und Verbesserung des Berner & Mattner Werkzeuges TESTONA. Dieses Programm bietet Testern ein Werkzeug für eine strukturierte und systematische Ermittlung von Testszenarien und -umfänge [1]. Im Kapitel 3.1 wird weiteres zu diesem Programm und die Funktionsweise erläutert.

Die Erweiterung des Programmes besteht aus verschiedenen Themen. Ein Thema davon behandelt die Testfallgenerierung und die jeweilige Testabdeckung. Hier soll garantiert werden, dass bei einer automatischen Testfallgenerierung, eine höchstmögliche Testabdeckung erzielt wird.

Die Testfallgenerierung wird in dieser Arbeit beeinflusst, indem die Produktvarianten stärker betrachtet werden. Verschiedene Varianten beinhalten verschiedene Parameter und Produktkomponenten. Die Parameterwerte definieren auch verschiedene Produktvarianten. Durch das Add-On MERAN für die Anforderungsmanagementsoftware „IBM Rational DOORS“ können Anforderungen direkt in TESTONA importiert werden. Dabei sollen automatisch die Parameterwerte zur jeweiligen Produktvariante zugeordnet werden. Aus diesem Grund kann es zu Konflikten bei der Testfallgenerierung kommen, bzw. inkohärente Testfälle können auftreten.

Um solche Probleme zu vermeiden oder zu umgehen, gibt TESTONA den Testern die Möglichkeit Abhängigkeitsregeln anzulegen. Hier können Anfangsbedingungen sowie Sonderbedingungen definiert werden. Dabei muss wiederum beachtet werden, dass die Produktvarianten nicht verletzt werden. Weiteres zu den Themen und Begriffen wird im Kapitel 3 verdeutlicht.

Im Kapitel 2 wird die Aufgabe dieser Masterarbeit genauer erläutert und in den Kapiteln 4 und 5 jeweils eine Lösung vorgeschlagen und implementiert.

Kapitel 2

Aufgabestellung

2.1 Anforderungen

Ziel dieser Masterarbeit ist die Verbesserung der Testfallgenerierung und der Testabdeckung bei mehreren Produktvarianten, die Ersetzung von Parametern, die Prozessoptimierung sowie die Handhabung für den Benutzer in der TESTONA-Umgebung. Jedes Produkt kann unterschiedliche Produktvarianten beinhalten und jede Variante besteht aus unterschiedlichen Komponenten mit unterschiedlichen Parametern. In Abhängigkeit von der ausgewählten Variante sollen bei der Testfallgenerierung die dazugehörigen Komponenten berücksichtigt werden und die erzeugten Testfälle dargestellt werden. Besonders zu beachten sind dabei die definierten Abhängigkeitsregeln sowie die darauf bezogene Testabdeckung.

Abhängigkeitsregeln werden definiert um redundante Testfälle zu vermeiden, bzw. um Vorbedingungen für die Testfälle zu erstellen. Da Varianten verschiedene Bauelemente beinhalten, kann es dazu kommen, dass Bauelemente für Abhängigkeitsregeln nicht vorhanden sind. Dadurch könnte TESTONA bei der Testfallgenerierung die Testabdeckung verfälschen, indem die Gültigkeit eines Testfalles nicht garantiert werden kann. Um dieses Problem zu umgehen, muss bei der Erzeugung von Abhängigkeitsregeln auf mögliche Konflikte hingewiesen werden. Für den Lösungsansatz gibt es verschiedene Thesen die analysiert werden müssen, um eine optimale Prozessoptimierung zu erreichen.

Um die Handhabung der Varianten bezogen auf die Testfälle und die Testgenerierung benutzerfreundlicher und effizienter zu gestalten, soll die Benutzung des Variantenmanagements durch einen Testingenieur untersucht werden. Resultierend aus den erworbenen Erkenntnissen wird das Lösungsdesign für eine Erweiterung des bestehenden Variantenmanagements in TESTONA konzipiert.

Einer der besonderen Eigenschaften von TESTONA ist die Kopplung mit Anforderungsspezifikationen die in IBM Rational DOORS definiert worden sind. Durch das DOORS Add-On MERAN können Anforderungen die in DOORS definiert sind, mit den zugehörigen Varianten verknüpft werden. Diese Varianten können durch eine erfolgreiche Anmeldung bei DOORS (über die TESTONA Oberfläche) und ein gezieltes Auswählen der gewünschten Varianten in TESTONA eingebunden werden. Hierbei sollen die in den Anforderungen definierten Parametern (z.B. eine Geschwindigkeit oder die Anzahl der Türen eines Autos) mit gespeichert werden. Im Klassifikationsbaum soll je nach ausgewählter Variante mit dem entsprechenden Wert ersetzt werden (z.B. der Name vom Bauelement). Andere Lösungsmöglichkeiten werden noch untersucht.

Der derzeitige Varianten-Management-Ansatz in TESTONA ist nicht in der Lage für die Testfallgenerierung zwischen verschiedene Varianten zu unterscheiden. Zwar werden durch die Perspektive „Variant Management“ verschiedene Varianten unterschieden, aber die Testfälle müssen manuell mit den jeweiligen Varianten verknüpft werden. Im Falle einer automatischen Testfallgenerierung werden auch ungültige Bauelemente betrachtet (siehe Abbild 2.1 und 2.2). Um dies zu vermeiden muss der Testingenieur einzelne Generierungsregeln anlegen. Dieser Vorgang soll automatisiert und von TESTONA übernommen werden. Dabei gibt es verschiedene Betrachtungsweisen und mehrere Lösungswege. Die erworbenen Kenntnisse des Testingenieurs über die Benutzung des Variantenmanagements sind entscheidend für die Lösung. Bei der Lösung ist zu beachten, dass eine komplette Testfallabdeckung garantiert werden muss.

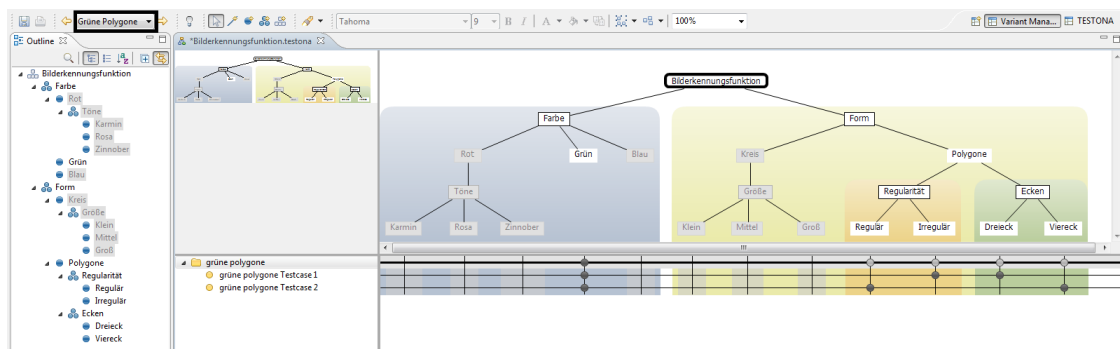


Abbildung 2.1: Richtige Auswahl der Klassen und Klassifikationen für die Testfallgenerierung

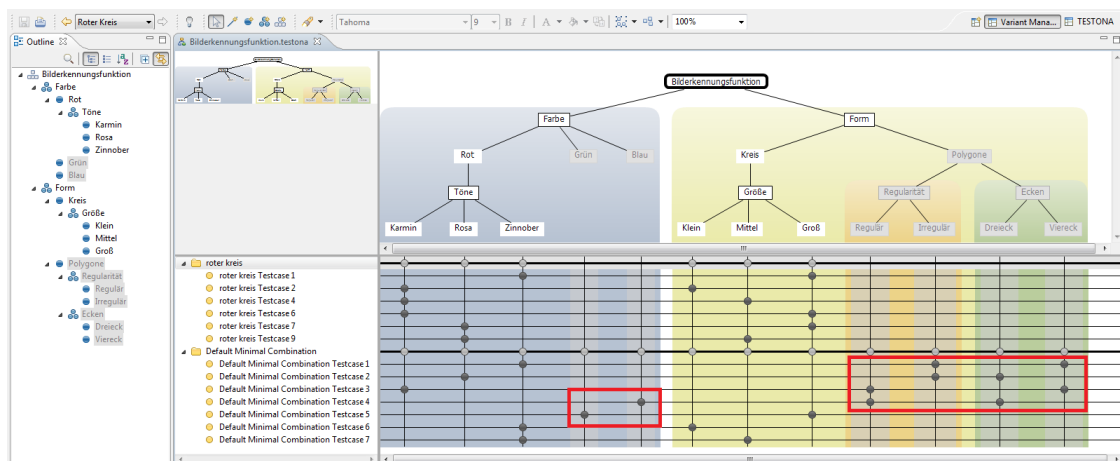


Abbildung 2.2: Ungültige Auswahl der Klassen (Grün und Blau) und Klassifikationen (Regularität und Ecken) für die Testfallgenerierung

2.2 Variantenmanagement Software

Die Erweiterung des schon vorhandenen Variantenmanagements in TESTONA wird dabei beitragen eine bessere Kopplung zwischen DOORS und TESTONA zu erzielen. Nach Recherche über schon vorhandene Lösungen zu dieser Problemfrage bin ich auf keine brauchbare Implementation

die in TESTONA angewendet werden kann gestoßen. Da es unter dem Begriff „Variante“ viel Freiraum gibt, sind vorhandenen Lösungen sehr Problemspezifisch. Es gibt Software die Varianten in ähnlicher Weise wie TESTONA unterstützt. Aber es gibt keine Verknüpfung zu einer Datenbank, in der schon Varianten definiert sind.

Die Firma „Razorcat Development GmbH“ bietet auch einen Klassifikationsbaumeditor der TESTONA sehr ähnlich ist. Nach meiner Kenntnis hat diese Software keine Schnittstelle zu einer Datenbank und kein Variantenmanagement¹.

2.3 Neue Anforderungen

Während der Programmierung haben sich die Anforderungen leicht verändert.

Die Abhängigkeitsregeln sollen das Variantenmanagement nicht mehr so sehr beeinflussen. Sie sollen enger mit der Testfallgenerierung verbunden sein als mit den einzelnen Varianten. Es wird davon ausgegangen, dass der Benutzer von TESTONA nur gültige Abhängigkeitsregeln deklariert hat. Weiterhin werden Optimierungsschritte betrachtet. Es kann dazu kommen, dass doppelte Testfälle durch gleiche Parameterwerte in einer Variante erzeugt werden.s

¹<http://www.razorcat.eu/cte.html>, 15/11/2014

Kapitel 3

Fachliches Umfeld

3.1 TESTONA

Mit TESTONA wird dem Tester ein Tool angeboten, um signifikante Testszenarien und -umfänge strukturiert zu bestimmen. Mit dem Programm können komplette Testspezifikationen schnell und einfach generiert werden und überflüssige Testfälle vermieden werden. Bei Bedarf gibt es die Möglichkeit automatische generierte Testspezifikationen und Testfälle bequem mit Anforderungen durch Add-Ons an Standard-Werkzeugen zu verlinken (siehe 3.2). Somit werden robuste Schnittstellen für ein komfortables Anforderungs- und Testmanagement erzeugt. Ein sehr wichtiger Aspekt von TESTONA ist, dass es Branchen-unabhängig ist. Das heißt ein Tester kann TESTONA im jedem Fachgebiet benutzen, nicht nur bei Software- oder Funktionstests.

Mehr zu Testfällen und der Arbeitsweise dieses Programms werden in den nächsten Kapiteln erläutert, wie zum Beispiel die anerkannte Klassifikationsbaum-Methode.

3.1.1 Klassifikationsbaum-Methode

1993 entwickelten K. Grimm und M. Grochtmann die Klassifikationsbaum-Methode zur Ermittlung funktionaler Backbox-Tests im Bereich von eingebetteter Software. Die Methode wurde im Forschungslabor von Daimler-Benz in Berlin als Weiterentwicklung der Category-Partition Method (CPM) erforscht. Gegenüber CPM hat die Klassifikationsbaum-Methode eine graphische Baum-Darstellung und hierarchische Verfeinerungen für implizite Abhängigkeiten. Als Werkzeug wurde der „Classification Tree Editor“ (CTE) ¹ programmiert und unterstützt Partitionierung und Testfallgenerierung. Das Werkzeug von CPM konnte nur Testfälle generieren ohne Bestimmung der Testaspekte[7].

Diese Methode besteht aus zwei wichtigen Schritten:

- Bestimmung der Klassifikationen (testrelevante Aspekte) und Klassen (mögliche Ausprägungen).
- Erzeugung von Testfällen aus Kombinationen von unterschiedlichen Klassen für alle Klassifikationen

¹Entwickelt von Grochtmann und Wegener[10]. Bei Berner & Mattner aus rechtlichen Gründen zu TESTONA umbenannt

Ansatzpunkt sind die funktionalen Anforderungen (siehe 3.2) eines zu testenden Objekts. Um die Testfälle zu definieren und zu erzeugen, folgt die Methode dem Prinzip des kombinatorischen Testentwurfs [15]. Dieses Prinzip hilft bei der Detektierung von Fehlern in frühen Schritten des Testvorgangs. Nicht jeder einzelne Parameter steuert einen Fehler bei, eher werden Fehler durch die Interaktion verschiedener Parameter verursacht. Betrachten wir ein einfaches Beispiel. Ein Programm soll auf Windows oder Linux laufen, unter Verwendung eines AMD oder Intel Prozessors und mit Unterstützung des IPv4 oder IPv6 Protokolls. Das ergibt intuitiv acht verschiedene Testfälle ($2^3 = 8$ Möglichkeiten, siehe Abbildung 3.1).

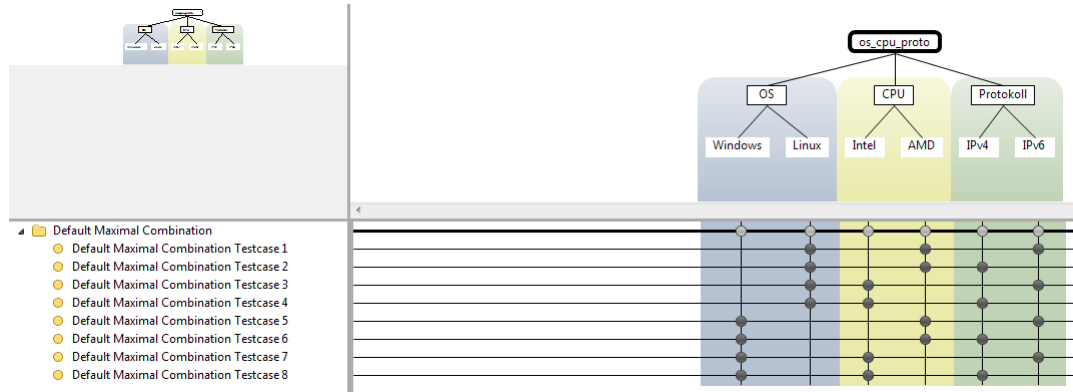


Abbildung 3.1: Baum und Testfälle ohne Kombinatorik

Verwenden wir dafür den kombinatorischen Testentwurf „paarweise Kombination“ (in Testona: *pairwise*(OS, CPU, Protokoll)), hätten wir nur vier Testfälle (siehe Tabelle 4.1). Durch diese Methode werden alle Kombinationspaare der Parameter mindestens durch einen Testfall abgedeckt[9].

No.	OS	CPU	Protokoll
1.	Linux	AMD	IPv6
4.	Linux	Intel	IPv4
5.	Windows	AMD	IPv6
8.	Windows	Intel	IPv4

Tabelle 3.1: Testfälle mittels paarweise Kombinatorik

Die Effizienz von diesem einfachen kombinatorischen Entwurf ist beim komplexeren System zu sehen. Hat ein System 20 verschiedene Schalter und jeder Schalter 10 verschiedene Einstellungen, so gibt es 10^{20} verschiedene Kombinationen. Durch Anwendung der paarweisen Kombination muss der Tester nur 180 Testfälle betrachten.

Ein Experiment hat gezeigt, dass durch die Verwendung von der paarweisen Kombinatorik die gleichen oder meistens mehrere Fehler entdeckt wurden, als mit der manuellen Testauswahl². Paarweise Kombinatorik ist am meisten verbreitet, aber man kann durchaus auch die Drei-Wege-Kombinatorik verwenden. TESTONA implementiert standardmäßig Minimalabdeckung, Paarweise-, Drei-Wege- und N-Kombinatorik (wo N die maximale Anzahl an möglichen Parametern im Klassifikationsbaum ist, auch vollständige Kombinatorik genannt)[9].

²Basierend auf funktionelle und technische Anforderungen, Use-Cases

3.1.2 Testfälle und Testfallgenerierung

Unter einem Testfall versteht man, die Beschreibung eines elementaren Zustands eines Testobjekts. Hierfür werden Eingangsdaten benötigt (Parameterwerte, Vorbedingungen) und ein erwarteter Folgezustand. Mit TESTONA werden Testfälle für eine vereinbarte Testspezifikation definiert. Laut IEEE 829 ist unter Testspezifikation zu verstehen: die Durchführung von

- **Testentwurfsspezifikation:** verfeinerte Beschreibung der Vorgehensweise für das Testen einer Software
- **Testfallspezifikation:** dokumentiert die zu benutzenden Eingabewerte und erwarteten Ausgabewerte
- **Testablaufspezifikation:** Beschreibung aller Schritte zur Durchführung der spezifizierten Testfälle.

Da TESTONA in allen Testphasen einsetzbar ist, kann die Arbeitszeit effizient reduziert werden. Dazu hilft auch die automatische Testfallgenerierung und die verschiedenen kombinatorischen Möglichkeiten (siehe 3.1.1). Somit kann der Tester einen besseren Zeitplan erzeugen und die Arbeitskräfte zielbewusst an der Ausführung und Auswertung der Testfälle beschäftigen.

Allgemein wird ein Test laut des ISTQB-Glossar³ folgendermaßen definiert:

Der Prozess, der aus allen Aktivitäten des Lebenszyklus besteht (sowohl statisch als auch dynamisch), die sich mit der Planung, Vorbereitung und Bewertung einer Softwareprodukts und dazugehörige Arbeitsergebnisse befassen. Ziel des Prozesses ist sicherzustellen, dass diese allen festgelegten Anforderungen genügen, dass sie ihren Zweck erfüllen und etwaige Fehlerzustände zu finden.[6]

Anhand der generierten Testfälle und die Benutzung von kombinatorischen Möglichkeiten kann der Tester einfach eine präzise Testtiefe erreichen. Die Testtiefe wird anhand der Durchführung einer Risikoanalyse und der Auswertung der Kritikalitätsstufe (sehr hoch, hoch, mittel und tief) des Systems vereinbart. Das soll heißen, dass die Testtiefe für ein Flugzeug (Kritikalitätsstufe = sehr hoch⁴) viel höher und genauer ist, als die Kompatibilität eines Bildschirms mit ein Graphiktreiber (Kritikalitätsstufe = tief⁵).

Anhand der Risikoanalyse und der Kritikalitätsstufe wird auch eine vereinbarte Testabdeckung und ein Testfallermittlungsverfahren erreicht. Im Falle des Flugzeugs wird eine Kombination von White und Blackbox-Methoden mit sehr hoher Testtiefe ausgeführt. Dagegen im Falle des Bildschirms wird intuitives Testen mit geringer Testtiefe angewendet.[13]

3.1.3 Abhängigkeitsregeln

Abhängigkeitsregeln werden vereinbart um überflüssige Testfälle zu vermeiden, bzw. um Vorbedingungen für bestimmte Testszenarien festzulegen. Abhängigkeitsregeln werden Mithilfe von boolische Algebra definiert wie folgende Abbildung 3.4 zeigt:

³International Software Testing Qualification Board

⁴Das Fehlverhalten kann zu Verlust von Menschenleben führen, die Existenz des Unternehmens gefährden

⁵Das Fehlverhalten kann zu geringen materiellen oder immateriellen Schäden führen

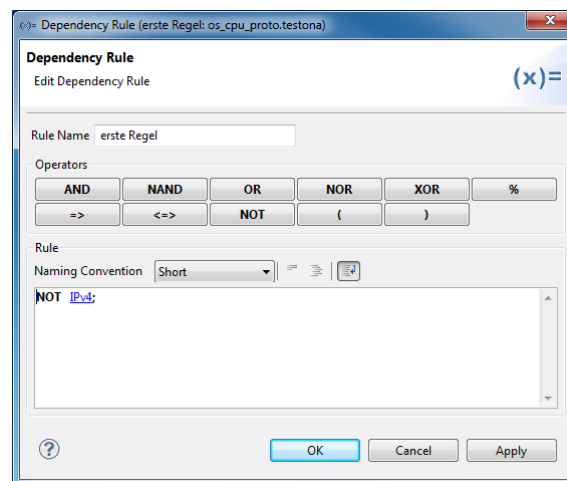


Abbildung 3.2: Abhängigkeitsregeln Bearbeitung

Boolesche Operatoren:

- AND : Konjunktion
- NAND : negierte Konjunktion
- OR : Disjunktion
- NOR : negierte Disjunktion
- XOR : ausschließende Disjunktion
- % : „don't care“ Operator
- => : vom A folgt B
- <=> : A ist gleichwertig wie B
- NOT : Negation

Durch die Verwendung dieser Operatoren werden folgende Abhängigkeitsregeln definiert:

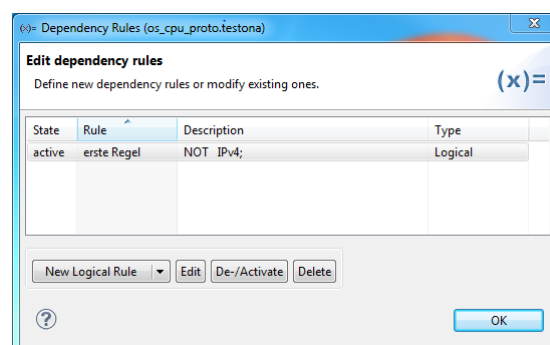


Abbildung 3.3: Abhängigkeitsregeln Übersicht

Nehmen wir das Beispiel wahr, so will der Tester die Klasse „IPv4“ für diese Tests ignorieren. Also werden nur Testfälle erzeugt, in denen dieser Parameter nicht vorkommt.

In diesem einfachen Beispiel wird nur ein Parameter ignoriert, aber durch die Abhängigkeitsregeln kann der Tester durchaus komplexere Fälle einleiten. Es kann die Reaktion eines Systems getestet werden, in dem sich die Spannung in einem niedrigen Bereich befindet. Es soll die Spannung einer Batterie geprüft werden, wenn ein Lichtschalter bedient wird. So ein Fall wird geprüft, indem verschiedene Regeln angelegt werden, die folgendermaßen aussehen:

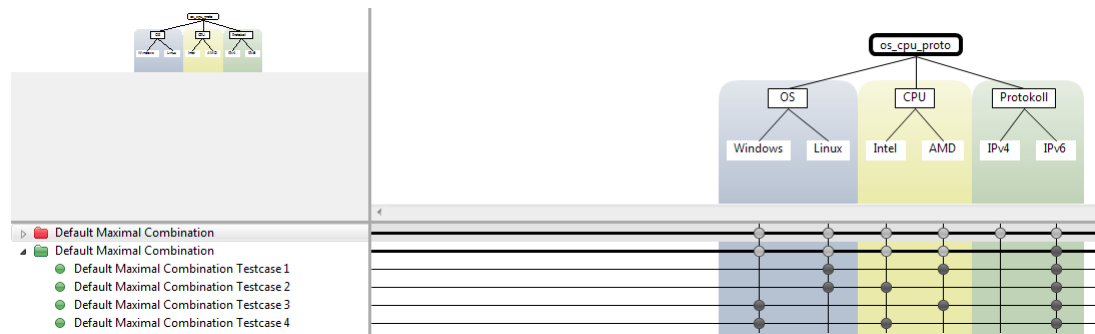


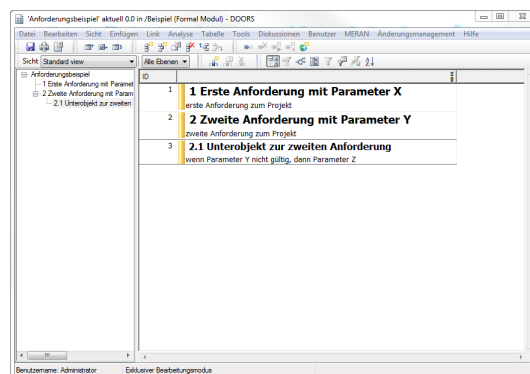
Abbildung 3.4: Testfälle mit Anwendung der Abhängigkeitsregeln aus 3.2 und 3.4

Lichtschalter1 XOR Lichtschalter2
NOT > 5V

Mit diesen Regeln, werden nur Testfälle betrachtet, in denen die Spannung der Batterie kleiner als 5V ist, und ein der beiden Lichtschalter betätigt wird.

3.2 IBM Rational DOORS

Quality Systems & Software (QSS) hat am Anfang der 90er Jahre DOORS (Dynamic Object Oriented Requirements System) entwickelt. Die Firma Telelogic kaufte im Jahr 2000 QSS, die wiederum 2008 von IBM übernommen wurde. DOORS ist eine Anforderungsmanagement Software und ermöglicht die Verwaltung und strukturierte Aufzeichnung von Anforderungen (als Objekte). Durch eine tabellarische Ansicht der Anforderungen können geordnet die Anforderungen und die zugehörige Eigenschaften abgelesen werden. Als Eigenschaften sind eine eindeutige Identifikationsnummer, sowie vom Benutzer ausgewählte Attribute.



The screenshot shows the 'Anforderungsbeispiel' window in DOORS. It contains a table with the following content:

1	1 Erste Anforderung mit Parameter X
	erste Anforderung zum Projekt
2	2 Zweite Anforderung mit Parameter Y
	zweite Anforderung zum Projekt
3	2.1 Unterobjekt zur zweiten Anforderung
	wenn Parameter Y nicht gut, dann Parameter Z

Abbildung 3.5: Beispiel einer Tabelle in DOORS

In DOORS heißt eine Tabelle „Modul“, jede Zeile innerhalb eines Moduls nennt sich „Object“ und die Spalten für jedes Object bezeichnet man als „Attribut“. Ein Object kann Unterobjekte besitzen, indem Alternativen und weitere Anforderungen beschrieben werden (siehe Abbildung 3.6).

Um Anforderungen im Laufe des Projektes zu verfolgen (Tracing), können Anforderungen miteinander verlinkt werden. DOORS basiert sich auf eine Client - Server Anwendung mit einer proprietären Datenbank.

Es werden auch Schnittstellen für den Datenaustausch zur Verfügung gestellt (Testmanagement-, Modellierungs- und Changemanagementwerkzeuge) dank der Unterstützung von RIF (Requirements Interchange Format). Durch die Skriptsprache „DXL“ (DOORS eXtension Language) erhält TESTONA Zugriff auf die gespeicherten Anforderungen in DOORS (durch die Ausführung von Skripts). [8] [14]

Um diese Aufgabe kümmert sich der TESTONA Plug-in *com.berner_mattner.testona.requirements.doors*. Dieses Plug-in beinhaltet den Java-Code für TESTONA, sowie die *dxl* Scripts für DOORS.

3.3 Variantenmanagement

Variantenmanagement, wie das Wort schon verrät, behandelt verschiedene Varianten eines Produktes. Leider gibt es oft Verwechslungen mit Feature Management. Für eine bessere Unterscheidung betrachten wir folgendes Beispiel: Ein Wagen kann in verschiedenen Modellen gebaut werden: Coupé, Limousine, Cabrio. Alle sind Wagen und haben alle groben Eigenschaften eines Wagens (4 Räder, Personenkraftwagen, etc), aber sie unterscheiden sich durch die Anzahl der Passagiere oder der Größe des Wagens. Diese sind Varianten eines Wagens der in verschiedene Modellen angeboten wird. [12]

Hier wird klar, dass durch die steigenden Produktkomplexität bzw. -vielfalt die Identifikationsmerkmale zur Definition einer Produktvariante immer schwieriger zu vereinbaren sind. Für diese Masterarbeit ist eher wichtig, Identifikationsmerkmale zu definieren, die dazu führen, dass die Tests oder der Testablauf eines Produktes geändert werden kann (mehrere Testfälle sind nötig, neue Parameterwerte). Das heißt, wenn ein Wagen als Coupé gebaut wird und danach als Cabrio angeboten wird, müssen (unter anderem) die ganze Dachfunktionen geprüft werden. So sollte man die Karosseriefarbe als ein Feature betrachten, weil theoretisch eine Änderung der Farbe keine Änderung in der Funktionalität oder sich auf die Leistung des Wagens auswirkt (somit werden die Tests oder Testabläufe nicht beeinflusst). [11]

Das Variantenmanagement wird im Fall von TESTONA dazu benutzt, um bei Produktvarianten ein besseren Überblick zu halten und eine bessere Testabdeckung zu sichern. Nennenswert ist, dass TESTONA für das Variantenmanagement die Testspezifikation als Ziel hat. Das hilft bei der Entscheidung zwischen einem Feature und einer Variante. Da wie bereits erwähnt, sind Varianten Änderungen eines Produktes, in der es mehr Gemeinsamkeiten als Unterschiede gibt. Mit dem Eintragen der Änderungen in TESTONA kann sich der Tester viel Arbeit ersparen, da neue Tests und Testabläufe besser erkannt werden.

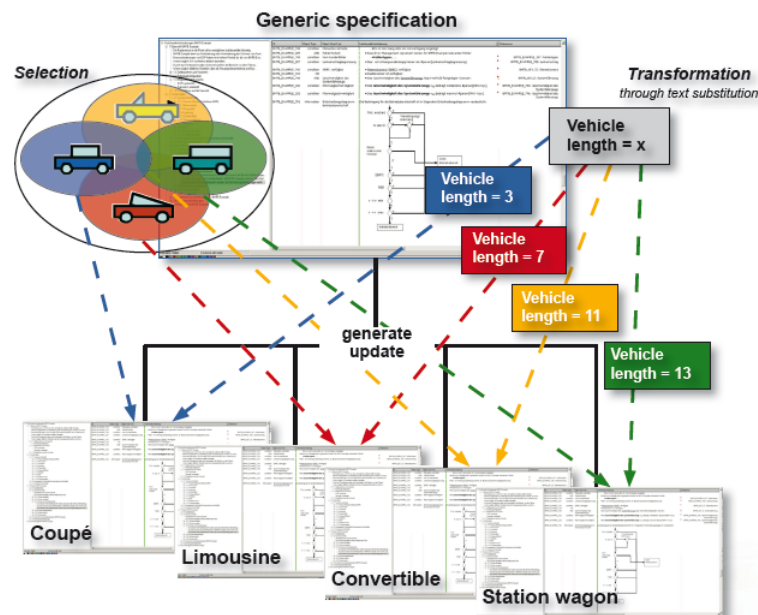


Abbildung 3.6: Betrachtung von Varianten eines Wagens von der generischen Testspezifikation zur variantenspezifische Testspezifikation aus [12]

3.4 Entwicklungsumgebung und Programmiersprache

TESTONA wurde ursprünglich in der Programmiersprache „Pascal“ entwickelt. Mit der Weiterentwicklung wurde das Programm an „C“ portiert. Durch den Kauf von Berner & Mattner in 2008 wurde TESTONA bis zum jetzigen Zeitpunkt auf Java übersetzt und wird seit 2010 mit der Entwicklungsumgebung Eclipse entwickelt.

3.4.1 Eclipse

Eclipse ist der Nachfolger von „IMB Visual Age for Java“ und ist ein quelloffenes Programmierwerkzeug zur Entwicklung verschiedener Arten von Software. Ursprünglich wurde Eclipse als integrierte Entwicklungsumgebung für Java benutzt. Dank seiner Bedienbarkeit und Erweiterung ist Eclipse mittlerweile für die Entwicklung in verschiedenen Programmiersprachen bekannt (unter anderem C/C++ und PHP). Die am 25. Juni 2014 veröffentlichte Version „Luna“ (Eclipse 4.4) ist der aktuellste Stand der Software.

Mit Eclipse 3.0 hat sich die Grundarchitektur von Eclipse geändert. Seit diesem Zeitpunkt ist Eclipse nur ein Kern, der einzelne Plug-ins lädt. Jedes Plug-in stellt eine oder verschiedene Funktionalitäten zur Verfügung. Darauf aufbauend existiert die „Rich Client Platform“ (RCP). Diese ermöglicht Entwicklern Anwendungen zu programmieren, die auf das Eclipse Framework aufbauen, aber unabhängig von der Eclipse IDE ist [5] [2]. Dies ist einer der Hauptgründe warum TESTONA zu Java und Eclipse migriert wurde. Durch die RCP können verschiedene Programmversionen (Light, Express, Professional, Enterprise) besser verwaltet werden. Durch die RCP Architektur können auch verschiedene Funktionalitäten voneinander getrennt werden. Das hilft bei der Weiterentwicklung sowie bei der Pflege des Programms, da mehrere Programmierer gleichzeitig an verschiedenen Plug-ins arbeiten können.

3.4.2 Plug-ins

Ein Plug-in ist die kleinste ausführbare Softwarekomponente für Eclipse. Um eine Anwendung mit Eclipse RCP zu schreiben, werden mindestens diese drei Plug-ins benötigt:

- Eclipse Core Plattform: steuert den Lebenszyklus der Eclipse Anwendung
- Standard Widget Toolkit: Programmierbibliothek zur Erstellung von grafischen Oberflächen
- JFace: User Interface Toolkit für komplexere Widgets

Weitere Plug-Ins, von der Eclipse Foundation implementiert, stehen den Programmierern zur Verfügung und unter marketplace.eclipse.org können auch von Privatentwicklern programmierte Plug-Ins heruntergeladen werden. [2]

Plug-ins beinhalten den Java-Code der, wie gewöhnlich, in verschiedene Pakete und Klassen strukturiert werden kann. Sinnvoll ist, dass jedes Plug-in eine Funktionalität des gesamten Programms repräsentiert, wie zum Beispiel, Variantenmanagement oder Autosave.

Testona besteht momentan aus 129 verschiedenen Plug-ins, wobei jedes Plug-in eine bestimmte Funktion oder Feature des Programms implementiert. Zum Beispiel gibt es für jede Version

(Light, Express, Professional und Enterprise) von TESTONA ein Plug-in indem die nötige Plug-ins für die gewünschte Version geladen werden.

Ein Plug-in besteht in der Regel aus folgenden Einheiten:

- **JRE System Library:** beinhaltet alle Systembibliotheken von der Java Runtime Environment, dass der jeweilige Plug-in benötigt
- **Plug-in Dependencies:** schließt die Abhängigkeiten des Plug-ins mit der Eclipse Umgebung und anderen implementierten Plug-ins ein
- **src:** bezieht die Pakete und Java-Klassen ein
- **icons:** hier befinden sich die Bilderdateien (.gif, .png, etc) die in den Klassen für die Benutzeroberfläche aufgerufen werden
- **META-INF:** gibt eine Übersicht aller Einstellungen des Plug-ins, sowie die Möglichkeit diese über eine graphische Oberfläche zu bearbeiten
- **build.properties:** beinhalten die Einstellungen für das Compilieren des Plug-in. Es kann auch über die META-INF Datei bearbeitet werden.
- **plugin.xml:** hier werden die nötigen Erweiterungen für das Plug-in definiert. Es ist möglich direkt die *xml* Datei zu bearbeiten, oder die META-INF Oberfläche benutzen.

Ein Plug-in kann durchaus mehrere Einheiten oder Elemente beinhalten, wie zum Beispiel weitere *resource* Ordner oder ein Dokumentationsordner mit wichtigen Dokumenten zum Plug-in.

3.4.3 Standard Widget Toolkit (SWT)

SWT ist eine IBM Programmierbibliothek (seit 2001) für die Programmierung grafischen Oberflächen unter Java. Die Bibliothek benutzt, im Gegensatz zu Swing⁶, die nativen grafischen Elemente des jeweiligen Betriebssystems und ermöglicht die Erstellung von Anwendungen, die optisch ähnlich wie die nativen Anwendungen des Betriebssystems aussehen. Durch die Verwendung der nativen grafischen Elemente, kann das Toolkit sofort Änderungen in das „look and feel“ des Betriebssystems in der Anwendung aktualisieren und beinhaltet ein konstantes Programmiermodell in alle Plattformen. [4]

SWT beinhaltet sehr viele komplexe Eigenschaften. Aber für eine robuste und benutzbare Anwendung zur Programmieren sind nur die Grundkenntnisse nötig. Eine typische SWT Anwendung hat folgende Struktur:

- Ein *Display* deklarieren (repräsentiert die SWT Modus)
- Erstellen eines *Shell*, welche als Hauptfenster dient
- Erzeugung eines Widgets
- Initialisierung der Widgetsparameter
- Öffnen des Fensters

⁶Programmierschnittstelle und Grafikbibliothek für Java zum programmieren von grafischen Benutzeroberflächen.

- Starten der Event-Schleife bis eine Abbruchbedingung erfüllt wird (Schließen des Fenster vom Benutzer)
- Entsorgen des Displays

```
1 public static void main (String[] args) {  
2     Display display = new Display();  
3     Shell shell = new Shell(display);  
4     Label label = new Label(shell, SWT.CENTER);  
5     label.setText("Hello_world");  
6     label.setBounds(shell.getClientArea());  
7     shell.open();  
8     while (!shell.isDisposed()) {  
9         if (!display.readAndDispatch())  
10            display.sleep();  
11     }  
12     display.dispose();  
13 }
```

Listing 3.1: Beispiel einer SWT Anwendung

3.4.4 JFace

JFace ist ein User Interface Toolkit, das auf die von SWT gelieferten Basiskomponenten setzt und stellt die Abstraktionsschicht für den Zugriff auf die Komponenten bereit. Es beinhaltet Klassen zur Handhabung gemeinsame Programmieraufgaben, wie zum Beispiel:

- Viewers: Verbindung von Oberflächenelementen zum Datenmodell
- Actions: definiert Benutzeraktionen und spezifiziert wo diese zur Verfügung stehen
- Bilder und Fonts: gemeinsame Muster für den Umgang mit Bilder und Fonts
- Dialoge und Wizards: Framework für komplexere Interaktionen mit dem Benutzer
- Feldassistent: Klassen die dem Benutzer für richtige Inhaltsauswahl bei Dialogen oder Formularen Hilfe anbieten

SWT ist komplett unabhängig von JFace (und Plattform Code), aber JFace wurde konzipiert um SWT bei allgemeinen Benutzerinteraktionen zu unterstützen. Eclipse ist wohl das bekannteste Programm das JFace benutzt.[3]

Kapitel 4

Lösungsansatz

Um den Lösungsansatz besser zu verstehen, wird erst der Ablauf von der Erstellung der Anforderung bis zu einem Testfall erläutert. Zuerst werden in DOORS alle schon vordefinierten Anforderungen eingetragen (siehe 3.2). In diesen Anforderungen können Parameter vorkommen, die für die Testfälle relevant sind. Die Parameter werden folgendermaßen in einer Anforderung eingetragen:

```
#param [Parametername]  
#param [max_Geschwindigkeit]  
Das Auto hat eine Maximalgeschwindigkeit von #param[max_geschwindigkeit] km/h
```

Die Parameter sind in einer Parameter-Tabelle definiert, in der die jeweiligen Parameterwerte eingetragen werden. Der Grund, warum es eine gesonderte Tabelle für Parameter gibt, kommt daher, dass die Parameter in der Regel pro Variante verschiedene Werte annehmen. Zum Beispiel beträgt die maximale Geschwindigkeit bei einem Cabrio 100 km/h und bei einem Kombi 150 km/h. Dank dieser Tabelle werden den Varianten (Cabrio und Kombi) die richtige Parameterwerte zugewiesen.

Da Parameterwerte und Varianten schon verlinkt sind, müssen die Anforderungen, die Parameter beinhalten, mit der Parametertabelle manuell verlinkt werden. Wenn dieser Vorgang abgeschlossen ist, kann über die TESTONA Oberfläche die Verbindung zu DOORS aufgebaut werden um die nötigen Informationen zu importieren. Voraussetzung ist, dass der Tester bereits einen Klassifikationsbaum passend zum zu testenden Produkt und die dazu gehörige Anforderungen erstellt hat. Jetzt können die in DOORS definierten Varianten importiert werden. Zunächst muss der Tester die Bauelemente der richtigen Variante zuordnen (durch ein Ausschlussverfahren wird angenommen, dass alle Bauelemente in allen Varianten gültig sind).

4.1 Parameterspeicherung

Um die Parameter erfolgreich in TESTONA zu speichern, müssen diese aus DOORS importiert werden und aus den Anforderungen gelesen werden. Durch eine gezielte Anfrage an DOORS, über eine Java API, kann die Parametertabelle in TESTONA geladen werden. Die darin bestimmten Beziehungen (Parameterwert zu Variante) müssen fest in die TESTONA Datei gespeichert werden, damit diese auch ohne eine DOORS Verbindung zur Verfügung steht. Als erstes werden die Parameterwerte in Objekten gespeichert und ein *Tag* (Kennzeichen) gegeben und nach Programmende in die TESTONA Datei im XML Format gespeichert. Dank des *Tags* kann beim Programmstart wieder der Parameterwert gelesen werden und während das Programm ausgeführt wird können Änderungen vorgenommen werden.

Einer der Besonderheiten von TESTONA in Verbindung mit MERAN ist, dass Anforderungen und Bauelemente per Drag&Drop verknüpft werden können. Mit dieser Funktion muss der Tester die Anforderung mit einem dazu gehörigen Bauelement verknüpfen (auslösendes Ereignis der Parameterspeicherung). Damit das Programm die Aktion des Benutzers mitbekommt, wird an dieser Stelle ein *Listener* implementiert. Der *Listener* bekommt verschiedene Nachrichten von Ereignissen die gefiltert werden müssen. Wenn die Nachricht empfangen wird, dass eine Anforderung an ein Bauelement verknüpft wurde, muss eine zu programmierende Methode den Anforderungstext von diesem Bauelement lesen und nach Parametern suchen. Als erstes wird davon ausgegangen, dass ein Bauelement nur eine Anforderung beinhalten kann und eine Anforderung nur ein Parameter beinhaltet. Wird in der gelesenen Anforderung ein Parameter gefunden, so müssen die möglichen Werte dieses Parameters aus der DOORS Parametertabelle gelesen und gespeichert werden. An dieser Stelle beginnt die Parameterspeicherung.

Für die Parameterspeicherung ergeben sich zwei Lösungswege. Die erste Lösung lautet, die Parametertabelle beim Import der Anforderung gleich in TESTONA zu speichern. Diese Lösung hat den Vorteil, dass später eine Verbindung zu DOORS nicht mehr notwendig ist. Somit muss während der Parametersuche auf eine Verbindung mit DOORS nicht geprüft werden, bzw. die Verbindung muss nicht wieder aufgebaut werden. Das heißt der Benutzer muss sich nur einmal mit DOORS verbinden und kann in der Zukunft problemlos die Anforderungen an einem Bauelement verlinken und gleichzeitig werden die Parameter gelesen und gespeichert. Der Nachteil ist, dass möglicherweise unnötige Daten in TESTONA gespeichert werden.

Die zweite Lösung ist gegensätzlich zu der ersten Lösung. Hier werden jeweils nur die Daten gespeichert, die der Benutzer im Moment benötigt. Wird eine Anforderung an einem Bauelement verlinkt, so muss eine Verbindung zu DOORS aufgebaut werden und die Parametertabelle aufgerufen und in TESTONA gespeichert werden. Der große Nachteil dieser Lösung ist, dass der Benutzer möglicherweise keine Verbindung zu DOORS aufbauen kann. Zum Beispiel, wenn der Server nicht vorhanden ist, keine Zugriffsmöglichkeiten, etc. Welcher der beiden Lösungen implementiert wird, wird erst noch genauer analysiert. Die Lösung ist auch abhängig von den Anforderungen verschiedener Kunden und wie diese TESTONA anwenden. Die implementierte Lösung wird in Kapitel 5 vorgestellt und begründet.

Sind die Parameterwerte in TESTONA vorhanden, müssen diese der richtigen Variante zugeordnet werden. Dafür ist vorgesehen, dass der Parameterwert als Eigenschaft des Bauelementes gespeichert wird (als ein *Tag* Objekt). Die Darstellungsstruktur für die Variantenansicht ist folgendermaßen definiert:

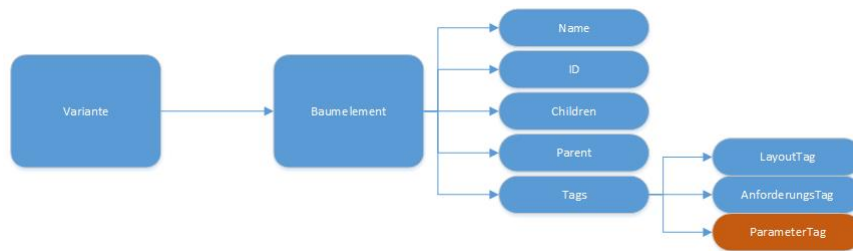


Abbildung 4.1: Darstellung der TESTONA Objekte für die Parameterspeicherung

Somit kann jetzt in TESTONA ein Parameterwert mit einer Variante und einem Bauelement verknüpft werden. Als Ergebnis werden folgenden Beziehungen erwartet:

Anforderung 1: Das Auto hat eine Maximalgeschwindigkeit von
#param[max_Geschwindigkeit] km/h.

Variante	Maximalgeschwindigkeit	Anforderung
Cabrio	100	1
Kombi	150	1
Limo	250	1

Tabelle 4.1: Beispiel für die Zuordnung zwischen Varianten und Parameterwert aus einer Anforderung

Wichtig ist auch, dass ein Bauelement verschiedene Parameter repräsentieren kann. Wenn ein Bauelement mehr als eine Anforderung mit verschiedenen Parametern beinhaltet, wird die aktive Variante entscheiden, welches Parameter ausgewertet und angezeigt wird. Dabei muss beachtet werden, dass vorhandene Parameterwerte und Anforderung (innerhalb eines Bauelementes) nicht gelöscht oder überschrieben werden.

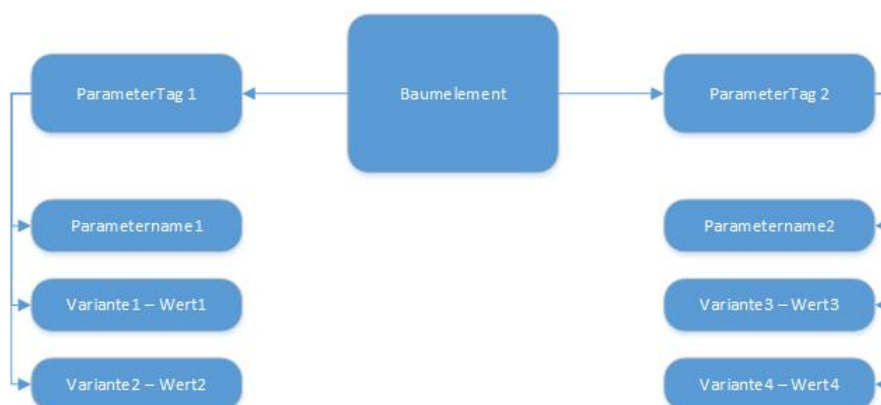


Abbildung 4.2: Darstellung eines Bauelementes mit zwei verschiedene Parametern und vier Varianten

4.2 Visualisierung

Wenn die Beziehungen zwischen Varianten, Parametern und Anforderungen erfolgreich entstanden sind, können jetzt die gespeicherten Parameterwerte angezeigt werden. Wenn der Benutzer die Variantenansicht in der Benutzeroberfläche ändert, die Beschriftung (*Label*) des Bauelementes (Maximalgeschwindigkeit von Generic = X, Cabrio = 150, Kombi = 200) muss eine Aktualisierung erfolgen.

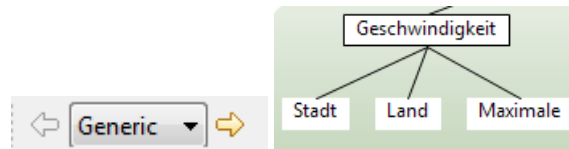


Abbildung 4.3: Aktive Variante Generic (default)

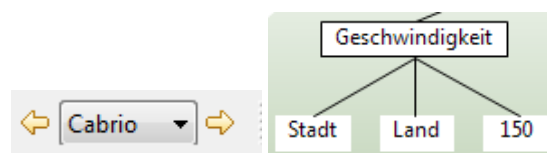


Abbildung 4.4: Aktive Variante Cabrio

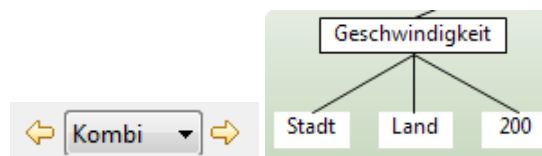


Abbildung 4.5: Aktive Variante Kombi

Dafür muss ein *Listener* beim Ändern der Variantenansicht eine Methode aufrufen. Diese aktualisiert die Werte und die Neuzeichnung der Bauelemente. Die Werte werden dynamisch aus den Inhalten der Bauelemente gelesen. Zur Erinnerung: Die Werte sind als *Tag* in jedem Bauelement gespeichert (siehe Abbildungen 4.1 und 4.2).

4.3 Testgültigkeit

Wenn TESTONA in der Lage ist, die Parameterwerte abhängig von der Variante darzustellen, soll möglicherweise nach duplizierten Testfällen gesucht werden. Es kann dazu kommen, dass sich ein Testfall dupliziert, da zwei oder mehrere Parameter den gleichen Wert besitzen. Ein einfaches Beispiel:

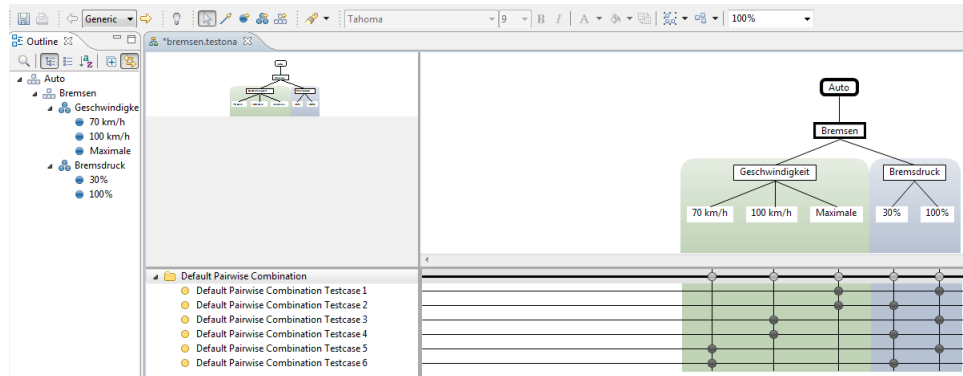


Abbildung 4.6: Das generische Baum

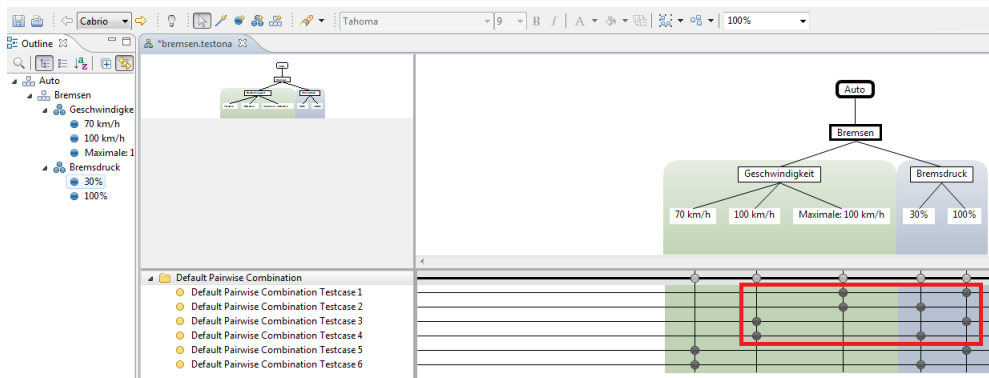


Abbildung 4.7: Aktive Variante Cabrio

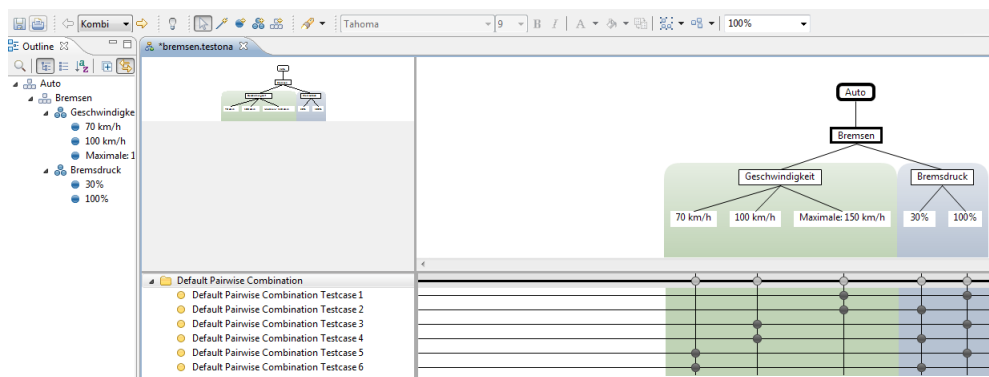


Abbildung 4.8: Aktive Variante Kombi

Es ist leicht zu erkennen, dass die Testfälle eins bis vier in der Variante „Cabrio“ dupliziert sind.

Bei einem minimalistischen Baum sind solche Fälle einfach zu erkennen. Bei einem komplexeren Baum mit sehr vielen Klassen kann es schnell zur Unübersichtlichkeit kommen. Duplizierte Testfälle könnten übersehen oder nicht erkannt werden.

Bei der Testfallgenerierung muss zusätzlich darauf geachtet werden (vorausgesetzt es gibt Varianten und Parameterwerte), dass nicht nur die Baumelemente sondern auch die Parameterwerte betrachtet werden.

Dafür müssen alle Klassen (Endknoten) unterhalb der gleichen Klassifikation überprüft werden. Nach dem obenstehenden Beispiel, müssen die Klassen „70 km/h“, „100 km/h“ und „Maximale“ verglichen werden. Da „Maximale“ bei jeder Variante einen anderen Wert annimmt, muss hier überprüft werden, dass bei jeder Variante, sich der Wert nicht wiederholt. Die Werte werden aus dem geplanten *ParameterTag* (siehe Abbildung 4.1) gelesen.

4.4 Optimierungskriterien

4.5 Benutzeroberfläche

Eine Änderung der Benutzeroberfläche von TESTONA wurde mit Kollegen in der Entwicklung offen diskutiert. Als Ergebnis konnten wir aus Sicht der Entwickler feststellen, dass die bisherige Lösung als gut und intuitiv zu bezeichnen ist.

Um weitere Meinungen und Lösungsvorschläge zu sammeln habe ich Testingenieure die TESTONA mit Kunden anwenden in der Firma Berner & Mattner befragt. Ein Ingenieur stellte fest, dass die Aktivierung der Variantenansicht unnötig ist. Er wünschte, dass die Variantenansicht als Hauptansicht definiert wird. Dass es einen Unterschied zwischen beiden Ansichten gibt liegt daran, dass verschiedene Etappen repräsentiert werden. Die Hauptansicht bietet verschiedene Features die in der Variantenansicht mit Variantenmanagement Features ergänzt werden. Dabei wird noch begründet, dass die Trennung zur Übersicht sehr viel beiträgt.

Als Ergebnis wurde festgelegt, dass für in dieser Arbeit gestellte Problemfrage nicht zur Unübersichtlichkeit der Benutzeroberfläche beiträgt. Es wird erhofft, dass außer der Erweiterung der Funktionalität, dass auch die Übersicht verbessert wird. Die Trennung zwischen der Hauptansicht und der Variantenansicht wird beibehalten ebenso die Lösung der Pfeile und des Drop-Down Menüs (siehe Abbildungen 4.3, 4.4, 4.5).

Kapitel 5

Systementwurf

5.1 Variantenmanagement und Parameter

Die Lösung zum Speichern und Darstellen der Parameterwerte abhängig der aktiven Variante wurde folgendermaßen programmiert.



Abbildung 5.1: Darstellung der Klassen

5.1.1 ResourceChangeListener

Das auslösendes Element ist, dass der Benutzer eine Anforderung mit einem Bauelement verknüpft. Um dieses Geschehen abzufragen wurde ein *Listener* programmiert, der das Interface *ResourceChangeListener*¹ implementiert. Die Aufgabe des *ResourceChangeListener* ist, dem Benutzer zu benachrichtigen, wenn sich eine Ressource ändert. Der *Listener* „hört“ auf die reinkommenden Nachrichten (*ResourceChangeEvent*) und wertet den Inhalt dieses Events aus.

Als Inhalt der Nachricht wird folgendes empfangen:

- **Event:** *ResourceChangeEvent*, heißt die Ressourcen des Objektes haben sich geändert
- **Notifier:** welches Objekt schickt die Nachricht
- **Notification:** Beschreibung von der Änderung im *Notifier*
- **OldValue:** alter Wert, wenn nicht vorhanden *null*
- **NewValue:** neuer Wert, beim löschen von Werten *null*

¹<http://download.eclipse.org/modeling/emf/transaction/javadoc/1.0.3/org/eclipse/emf/transaction/ResourceChangeListener.html>

Hier wird als erstes der *Notifier* abgefragt um zu unterscheiden, ob die Nachricht für uns relevant ist. Ist der *Notifier* eine Instanz eines Baumelementes, so wurde eine Anforderung zum ersten mal an das Bauelement verknüpft.

Lautet der *Notifier* Instanz von *RequirementTag* so wurde eine Anforderung von einem Bauelement gelöscht oder es wurde eine neue Anforderung an einen Bauelement hinzugefügt, wo schon Anforderung verknüpft waren. Für alle drei Fälle muss der *Listener* wissen:

- An welches Element wurde das Event ausgelöst?
- Welche Anforderung wurde hinzugefügt oder gelöscht?

Dafür wird aus dem *Notifier* mit der Methode `getID()` die Identifikationsnummer des zuständigen Bauelement abgefragt. Weiterhin steht im *Notifier* auch die Kennung der Anforderung (diese wurde in DOORS vergeben). Diese beide Informationen werden gespeichert und an den *VariantsManager* weiter gegeben. Diese Informationen sind für der *ParameterManager* nötig, damit er aus der Anforderung die Parameter lesen kann und mit das richtige Bauelement verknüpfen kann.

5.1.2 VariantsManager

Die Hauptaufgabe des *VariantsManager* ist die Varianten zu regeln. Hier werden Bauelemente zu Varianten hinzugefügt und gelöscht. Dabei muss das Klassifikationsbaum neu gezeichnet werden. Diese Klasse kümmert sich auch, um die Umschaltung zwischen Varianten in der Variantenansicht (siehe Abbildung 4.3). Der *VariantsManager* setzt die Bauelement-ID und die Anforderungskennung in der Klasse *ParameterManager*. Die Klasse dient als Schnittstelle zwischen der *Listener* und die Klasse *ParameterManager*

Der *VariantsManager* fragt den *Editor* (beinhaltet das Klassifikationsbaum) nach das benötigte Bauelement über die von *Listener* übergebene Identifikationsnummer. Das Bauelement wird dann in das *ParameterManager* gespeichert. Die Anforderungskennung wird auch im Objekt von *ParameterManager* gespeichert.

Requirement IndieModul im connection eigenschaften!!

5.1.3 Parameter-Tag

ParameterTag

Das Interface dient als Schnittstelle für den Zugriff auf das Inhalt von *ParameterTagImpl*. Das Interface erweitert *Tag*. Das *Tag* Interface ist ein in TESTONA allgemein implementiertes Modell, welches auf ein Interface und eine implementierende Klasse basiert. Für jedes *Tag* (Requirement-Tag, VariantTag, etc.) gibt es ein eine Klasse. In das Interface wird immer das Tagtyp definiert um *Tags* voneinander unterscheiden zu können.

```
public static final String TAGTYPE = "ParameterTag";
```

Listing 5.1: ParameterTag Interface

Das Tagmodell wird benutzt um in TESTONA überall ähnliche Objekt zu haben, die verschiedene Funktionen erfüllen, aber die gleiche Richtlinie folgen

ParameterTagImpl

Die Klasse *ParameterTagImpl* erweitert die Klasse *TagImpl* und implementiert *ParameterTag*. In dieser Klasse werden die Vorteile von das Tagmodell deutlicher. Die Klasse *TagImpl* definiert sehr nützliche Methoden und Eigenschaften die in *ParameterTagImpl* angewendet werden.

Jedes *Tag* hat als Eigenschaft einen Namen. In diese Klasse entspricht der Name, ein Parametername der aus der Parametertabelle gelesen wurde. Weiterhin ist über *ID* eine eindeutige Identifikationsnummer definiert, um *Tags* des gleichen Typs voneinander zu unterscheiden.

Der Inhalt von das *Tag* ist durch ein *EcoreEMap<String, String>* definiert. Das erste String ist ein Schlüsselwert und das zweite String das Wert. Im diesem Fall ist der Schlüssel der Name einer Variante, und der Wert ist der Parameterwert des Parameters in dieser Variante. Anhand der Methode *getContent()* kann der Inhalt eines *Tags* aufgerufen werden und mit der Methode *addEntry()* werden Inhalte hinzugefügt. Das Gegenteil kann man mit der Methode *deleteEntry()* erreichen. Wie die Methoden genauer angewendet werden, wird im Listing 5.2 gezeigt.

serialize und unserialize

5.1.4 ParameterManager

Die Klasse *ParameterManager* beinhaltet eine private innere Klasse *DoorsConnector*(siehe 5.1.6), diese kümmert sich um den Verbindungsaufbau mit DOORS.

Weiterhin baut die Klasse die vom *Listener* weitergegeben Daten (Bauelement und Anforderungskennung) auf.

Anforderung eines Bauelementes beinhaltet Modul und Interface. wichtig für nächstes Kapitel

IndieModul : interId: doors, connId: localhost;36677, moduleId: /Simple Belt Warner 2/Simple Belt Warner Spec, moduleNbr: 0, id: doors,localhost;36677,/Simple Belt Warner 2/Simple Belt Warner Spec)

```

1ParameterTag tempTag = new ParameterTagImpl();
2
3for(int i = 0; i < attributesList.size(); i++){
4
5    //An der nullte Stelle steht immer der Parametername
6    if(i == 0) {
7        tempTag.setName(req.getValue(attributesList.get(0)).
8            getValueAsString());
9    } else {
10        //entferne " Value" vom Variantenamen
11        tempTag.addEntry(sAttrList.get(i).substring(0, sAttrList.get(i).
12            lastIndexOf("Value")-1),
13            req.getValue(attributesList.get(i)).getValueAsString());
14    }
15}

```



```

13}
14paramTagList.add(tempTag);

```

Listing 5.2: Erstellung der ParameterTag

5.1.5 AddParameterTagCommand

5.1.6 Die DOORS Verbindung

DoorsConnector

Die private innere Klasse *DoorsConnector* baut die Verbindung zwischen den TESTONA und DOORS auf. Sie implementiert das Interface *IConnectionListener*, dass ein *Listener* für die Verbindung - Events umfasst. Für das Laden von Dateien aus DOORS benötigt die Klasse noch ein *Listener* (*IDataListener*) und ein *Adapter* (*IReqLoadAdapter*)

Als erstes wird von der Klasse *ParameterManager* die Methode *connectToDoors()* aufgerufen. Diese baut die Verbindung auf, indem gespeicherte Verbindungsdaten aufgerufen werden. Wie bereits in Kapitel 5.1.4 erwähnt, beinhaltet eine Anforderung die Informationen wo die Anforderung gespeichert ist (welches DOORS Modul und über welches Interface das Modul zu erreichen ist). Für den Verbindungsaufbau werden folgende Objekte benötigt:

- **DataInterface:** Über diese Klasse erfolgt die Datenanfrage an DOORS. Die Verbindung wird aufgebaut sowie getrennt. Es werden als erstes die Ordner geladen, danach einzelne Projekte und die nötige Module. Es können verschiedene Darstellungen der Module auch geladen werden (diese müssen in DOORS definiert sein). Hier werden auch direkt einzelne Anforderungen angefragt. Relevant für diese Arbeit ist, dass hiermit das Modul Parametertabelle in DOORS geladen wird.
- **PreferenceManagment:** Hier werden die in TESTONA gespeicherte Verbindungsdaten behandelt. Es können Microsoft Access Verbindungsdaten gespeichert werden, aber wir werden nur DOORS betrachten.
- **Connector:** beschreibt eine einzelne Verbindung, hat ein *DataInterface*- und *PreferenceManagmentobjekt*
- **ConnectionManager:** Singleton. Die Klasse handelt aktive und offene Verbindungen. Hier werden die *ConnectionListeners* und das *DataInterface* für den richtigen *Connector* geregelt.

Um die Verbindung mit DOORS aufzubauen muss als erstes die Instanz des *ConnecionManagers* lokal referenziert werden (weil es ein *Singleton* ist, darf kein neues Objekt erzeugt werden). Wenn die Instanz des *ConnectionManagers* geladen ist, kann jetzt der *connector* aus dem *ConnecionManager* aufgerufen werden.

```

1try {
2    connector = ConnectorManager.getInstance()
3        .getConnector(im.getInterId());
4    dataInterface = conManager.getNewDataInterface(connector, this);
5} catch (ExtensionException e) {
6    e.printStackTrace();
7}

```

Listing 5.3: Verbindungsaufbau

Um dem richtigen `connector` aufzurufen muss aus das *IndieModul* (`im.getId()`, siehe Kapitel 5.1.2) die Interfacekennung als Übergabeparameter angegeben werden. Als nächstes kann über den *ConnectionManager* eines neues *DataInterface* erzeugt werden, wo der `connector` und das aktuelle Objekt (*DoorsConnector*) übergeben werden.

Aus den in TESTONA gespeicherte Verbindungspräferenzen werden die Verbindungsparameter für DOORS geladen. An dieser Stelle braucht das *DataInterface* die nötige *Listeners* bevor die Verbindung aufgebaut wird. Durch die Methode `addListener(listener)` wird das *RequirementDataListener* und das *IConnectionListener* (von *DoorsConnector* implementiert) gesetzt. Mit dem Aufruf der Methode `connectInterface(Verbindungsparameter)` wird das *DataInterface* an DOORS verbunden.

Der Grund warum ein *IConnectionListener* implementiert wird, lautet dass der Verbindungsaufbau in einem neuen Thread stattfindet. Das *DoorsConnector* Objekt wird über das *IConnectionListener* benachrichtigt ob die Verbindung stattgefunden hat. Die Methode `interfaceConnected` (aus dem *IConnectionListener*) wird aufgerufen, wenn die Verbindung erfolgreich entstanden ist.

```
1@Override
2public void interfaceConnected() {
3    connected = true;
4    reqDataListener.setListener(reqLoadListener);
5    dataInterface.loadModule(PARAM_PATH, this, false);
6}
```

Listing 5.4: Verbindungsaufbau was erfolgreich

Der gesetzte *RequirementDataListener* benötigt ein *RequirementLoadListener*, dass eine rückmeldung gibt, wenn die Zeilen aus einer Tabelle fertig geladen worden sind. Die Tabelle wird anhand der Methode `loadModule` wird ein DOORS Modul (Tabelle) geladen. Welches Modul geladen wird, spezifiziert der Parameter `PARAM_PATH`. Es gibt an wo sich das Modul in der DOORS Datenbank befindet. Der *RequirementDataListener* erhält die Nachricht, dass ein Modul geladen worden ist. Weiter dazu wird im Kapitel 5.1.6 erläutert.

Es wird angenommen, wenn der Benutzer die Parameterersetzung für ein Parameter wahrnimmt, dass er es für weitere Parameter wahrnehmen wird. Da die Datenmenge von einer Parametertabelle relativ gering ist, wurde hier, bezüglich der offene Frage bei dem Lösungsansatz (siehe Kapitel 4.1), die Parametertabelle komplett importiert. Ein weiterer Grund lautet, dass nicht für jeder Parameter erneut eine DOORS Verbindung entstehen muss. So wird die Rechen- und Reaktionszeit von TESTONA so weit es geht gering gehalten. Die Parametertabelle wird erst bei der Verknüpfung von einer Anforderung mit einem Bauelement importiert und nicht beim Import der Anforderungen.

Wenn die Parametertabelle komplett geladen wurde, ermöglicht die Methode `closeConnection` die Verbindung mit DOORS zu beenden und das *DataInterface* zu schließen.

RequirementDataListener

Ist ein Event-Listener, dass auf die Rückmeldung vom Laden eines Moduls wartet. Relevant ist die Methode `onModuleLoad` die die Zeilen aus der Tabelle liest und speichert.

```
1@Override
2public void onModuleLoad(Module module, Object family, boolean reload){
```

```

3
4  BasicRequirement baseReq;
5  saveAttributesNames(module);
6
7  for (String reqId : module.getRequirementIds()) {
8
9      baseReq = dataInterface.getRequirement(module, reqId,
10         reqLoadListener);
11
12      if (baseReq.checkStatus(BasicRequirement.STATUS_LOADED)) {
13          paramReqList.add((Requirement) baseReq);
14          fillParamTagList((Requirement) baseReq);
15      }
16  }
17  dataInterface.flush(reqLoadListener);
18}

```

Listing 5.5: Laden der Parametertabelle nach Zeilen

Zu beachten ist, dass in DOORS jede Zeile in einer Tabelle als eine Anforderung (eng. Requirement) gesehen wird. Daher heißen Variablen und Methoden oft „Requirement“ oder Abkürzung des Wortes (req). Die Methode `saveAttributesNames` speichert die im geladenes Modul vorhandenen Attribute. Im diesem Fall (treu zum Beispiel mit dem Auto) sind es:

- Parameter Name
- Default Value
- Cabrio Value
- Kombi Value
- Limo Value

Diese Attribute repräsentieren die Varianten und ein Standardwert, sowie der Name des jeweiligen Parameter. In der Schleife werden alle Zeilen im Modul iteriert und geladen. Wenn die Zeile (baseReq) vollständig geladen wurde, wird diese in der globalen Liste `paramReqList` als ein Requirement Objekt gespeichert. Die Methode `fillParamTagList` erzeugt die *ParameterTags* und wurde in Kapitel 5.1.4 erläutert.

Wenn eine Zeile nicht vollständig geladen werden konnte, gibt es das *RequirementLoadListener*, dass sich um das vollständige laden der Zeilen kümmert. Das *RequirementLoadListener* wird in dieser Klasse instantiiert und von der *DoorsConnector* Klasse gesetzt. Weiter dazu im nächsten Kapitel.

RequirementLoadListener

Der *RequirementLoadListener* reagiert wenn eine Zeile nicht vollständig geladen worden ist, und wartet bis diese geladen wird. Die Methode `onLoad` bekommt als Eingabeparameter eine Liste der nicht geladenen Zeilen.

```

1 public void onLoad(List<BasicRequirement> requirements) {
2
3     for (BasicRequirement baseReq : requirements) {
4         paramReqList.add((Requirement) baseReq);
5         fillParamTagList((Requirement) baseReq);

```

```
6  
7 }  
8 }
```

Listing 5.6: Nachladen der Parametertabelle nach Zeilen

Diese Liste wird iteriert und wie in Kapitel 5.1.6 in der globalen Liste `paramReqList` als ein `Requirement` Objekt gespeichert.

Weiterhin meldet diese Klasse wenn alle Zeilen aus das DOORS Modul geladen wurden. Das wartenden Dialogfenster aus 5.1.4 wird benachrichtigt, dass es geschlossen werden kann. Die Benutzeroberfläche von TESTONA ist somit wieder für den Benutzer erreichbar.

5.2 Testfallgenerierung und Optimierung

Erläuterung der Lösungen zu 4.3

Kapitel 6

Evaluierung

6.1 Chances of Failure

Kapitel 7

Zusammenfassung und Ausblick

Abbildungsverzeichnis

2.1	Richtige Auswahl der Klassen und Klassifikationen für die Testfallgenerierung	4
2.2	Ungültige Auswahl der Klassen (Grün und Blau) und Klassifikationen (Regularität und Ecken) für die Testfallgenerierung	4
3.1	Baum und Testfälle ohne Kombinatorik	7
3.2	Abhängigkeitsregeln Bearbeitung	9
3.3	Abhängigkeitsregeln Übersicht	9
3.4	Testfälle mit Anwendung der Abhängigkeitsregeln aus 3.2 und 3.4	10
3.5	Beispiel einer Tabelle in DOORS	11
3.6	Betrachtung von Varianten eines Wagens von der generischen Testspezifikation zur variantenspezifische Testspezifikation aus [12]	12
4.1	Darstellung der TESTONA Objekte für die Parameterspeicherung	18
4.2	Darstellung eines Baumelementes mit zwei verschiedene Parametern und vier Varianten	18
4.3	Aktive Variante Generic (default)	19
4.4	Aktive Variante Cabrio	19
4.5	Aktive Variante Kombi	19
4.6	Das generische Baum	20
4.7	Aktive Variante Cabrio	20
4.8	Aktive Variante Kombi	20
5.1	Darstellung der Klassen	24

Listings

3.1	Beispiel einer SWT Anwendung	15
5.1	ParameterTag Interface	25
5.2	Erstellung der ParameterTag	26
5.3	Verbindungsaufbau	27
5.4	Verbindungsaufbau was erfolgreich	28
5.5	Laden der Parametertabelle nach Zeilen	28
5.6	Nachladen der Parametertabelle nach Zeilen	29

Anhang A

Anhang

A.1 CD

Inhalt:

- Quellen
- PDF-Datei dieser Arbeit

A.2 code 1

lhier kommt java code

A.3 code 2

lhier kommt auch java code

Literaturverzeichnis

- [1] Berner & Mattner, <http://www.testona.net>. *TESTONA*, Oktober 2014.
 - [2] Eclipse Foundation, <http://eclipse.org>. *Eclipse*, Oktober 2014.
 - [3] Eclipse Foundation, <http://help.eclipse.org>. *Eclipse Help*, Oktober 2014.
 - [4] Eclipse Foundation, <http://eclipse.org/swt>. *Eclipse SWT*, Oktober 2014.
 - [5] Eclipse Foundation. *Eclipse 4.0 RCP*. Someone, 2014.
 - [6] Stephan Grünfelder. *Software-Test für Embedded Systems*. dpunkt.verlag, 2013.
 - [7] M. Grochtmann and K. Grimm. *Classification Trees For Partition testing, Software testing, Verification & Reliability, Volume 3, Number 2*. Wiley, 1993.
 - [8] IBM, <http://www-03.ibm.com/software/products/de/ratidoor>. *IBM DOORS*, Oktober 2014.
 - [9] IEEE Computer Society, https://courses.cs.ut.ee/MTAT.03.159/2013_spring/uploads/Main/SWT_comb-paper1.pdf. *Combinatorial Software Testing*, 2009.
 - [10] Quality Week 1995, http://www.systematic-testing.com/documents/qualityweek1995_1.pdf. *Test Case Design Using Classification Trees and the Classification-Tree Editor CTE*, 1995.
 - [11] Christopher Robinson-Mallet. An approach on integrating models and textual specifications. *Model-Driven Requirements Engineering Workshop (MoDRE)*, 2012.
 - [12] Christopher Robinson-Mallet, Matthias Grochtmann, Joachim Wegener, Kens Köhnlein, and Steffen Kühn. Modelling requirements to support testing of product lines. *Third International Conference on Software Testing, Verification, and Validation Workshops*, 2010.
 - [13] H. Tresp and M. Ruggiero. *Application Engineering: Grundlagen für die objektorientierte Softwareentwicklung mit zahlreichen Beispielen, Aufgaben und Lösungen*. Compendio Bildungsmedien, 2011.
 - [14] Gerhard Versteegen. *Anforderungsmanagement: Formale Prozesse, Praxiserfahrungen, Einführungsstrategien und Toolauswahl*. Springer, 2004.
 - [15] Wikipedia, <http://de.wikipedia.org/wiki/Klassifikationsbaum-Methode>. *Klassifikationsbaum-Methode*, Oktober 2014.
-