

**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: 9. März 2015

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

---



## Danksagung

Zunächst möchte ich mich an dieser Stelle bei all denjenigen bedanken, die mich während der Anfertigung dieser Masterarbeit unterstützt und motiviert haben.



# Inhaltsverzeichnis

<b>1</b>	<b>Einleitung</b>	<b>2</b>
<b>2</b>	<b>Aufgabestellung</b>	<b>3</b>
2.1	Anforderungen . . . . .	3
2.2	Variantenmanagement Software . . . . .	6
2.3	Neue Anforderungen . . . . .	7
<b>3</b>	<b>Fachliches Umfeld</b>	<b>8</b>
3.1	TESTONA . . . . .	8
3.1.1	Klassifikationsbaum-Methode . . . . .	8
3.1.2	Testfälle und Testfallgenerierung . . . . .	10
3.1.3	Abhängigkeitsregeln . . . . .	11
3.2	IBM Rational DOORS . . . . .	14
3.3	Variantenmanagement . . . . .	16
3.4	Entwicklungsumgebung und Programmiersprache . . . . .	18
3.4.1	Eclipse . . . . .	18
3.4.2	Plug-ins . . . . .	18
3.4.3	Standard Widget Toolkit (SWT) . . . . .	19
3.4.4	JFace . . . . .	20
3.4.5	Tags . . . . .	21
<b>4</b>	<b>Lösungsansatz</b>	<b>23</b>
4.1	Parameterspeicherung . . . . .	24
4.2	Visualisierung der Parameterwerte . . . . .	27
4.3	Benutzeroberfläche . . . . .	28
<b>5</b>	<b>Systementwurf</b>	<b>29</b>
5.1	Variantenmanagement und Parameter . . . . .	29

---

---

5.1.1	VariantsManager . . . . .	30
5.1.2	ResourceSetListener . . . . .	31
5.1.3	Parameter-Tag . . . . .	34
5.1.4	ParameterManager . . . . .	37
5.1.5	Kommandos . . . . .	39
5.1.6	Die DOORS Verbindng . . . . .	40
5.2	Darstellung der Parameterwerte . . . . .	44
<b>6</b>	<b>Evaluierung</b>	<b>46</b>
6.1	Testaufbau und -Ablauf . . . . .	47
6.2	Ausfallrisiken . . . . .	50
6.3	Bekannte Fehler . . . . .	51
6.4	Optimierungskriterien . . . . .	52
<b>7</b>	<b>Zusammenfassung und Ausblick</b>	<b>53</b>
7.1	Zusammenfassung . . . . .	54
7.2	Persönliches Fazit . . . . .	55
<b>A</b>	<b>Anhang</b>	<b>59</b>
A.1	CD . . . . .	59
A.2	ParameterMananger Klassendiagramm . . . . .	60
	<b>Literatur- und Quellenverzeichnis</b>	<b>61</b>

---



# 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<sup>1</sup>[2]. Im Kapitel 3.1 wird dieses Programm und seine Funktionsweise erläutert.

Die Erweiterung des Programmes besteht aus verschiedenen Themen. Ein Thema behandelt die Testfallgenerierung und die jeweilige Testabdeckung.<sup>2</sup> Hier soll garantiert werden, dass bei einer automatischen Testfallgenerierung die höchstmögliche Testabdeckung erzielt wird.

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

---

<sup>1</sup> Kombination mehrerer Testfälle mit dem Ziel, komplexeren Sachverhalten zu überprüfen.

<sup>2</sup>(engl. Test Coverage bzw. Code Coverage) das Verhältnis an tatsächlich getroffenen Aussagen eines Tests gegenüber den theoretisch möglich treffbaren Aussagen. Tests werden anhand der Spezifikation einer zu testenden Software-Einheit definiert.[1]

---



# Kapitel 2

## Aufgabestellung

### 2.1 Anforderungen

Ziel dieser Masterarbeit ist die Verbesserung der Testfallgenerierung und der Testabdeckung bei mehreren Produktvarianten. Dafür soll die Ersetzung von Parametern aus den Varianten implementiert werden. Die Prozessoptimierung sowie die Handhabung für den Benutzer in der TESTONA-Umgebung soll verbessert werden. Jede Produktlinie 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 u.a. 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, da 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 An-

---

meldung bei DOORS (über die TESTONA Oberfläche) und ein gezieltes Auswählen der gewünschten Varianten in TESTONA eingebunden werden. Hierbei sollen die in den Anforderungen definierte Parametern (z.B. eine Geschwindigkeit oder die Anzahl der Türen eines Autos) mit gespeichert werden. Im Klassifikationsbaum soll je nach ausgewählter Variante der entsprechende Wert ersetzt werden (z.B. der Name des Bauelements). Andere Lösungsmöglichkeiten werden noch untersucht.

Der derzeitige Varianten-Management-Ansatz in TESTONA ist nicht in der Lage, für die Testfallgenerierung zwischen verschiedenen 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.

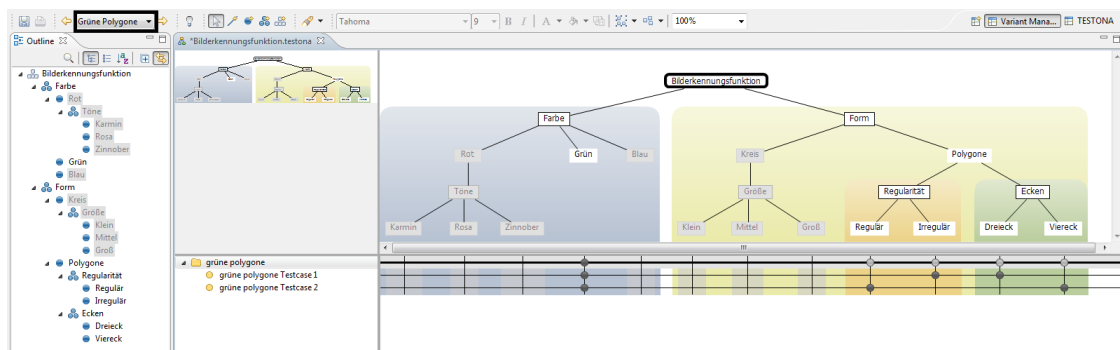


Abbildung 2.1: Richtige Auswahl der Klassen und Klassifikationen für die Testfallgenerierung

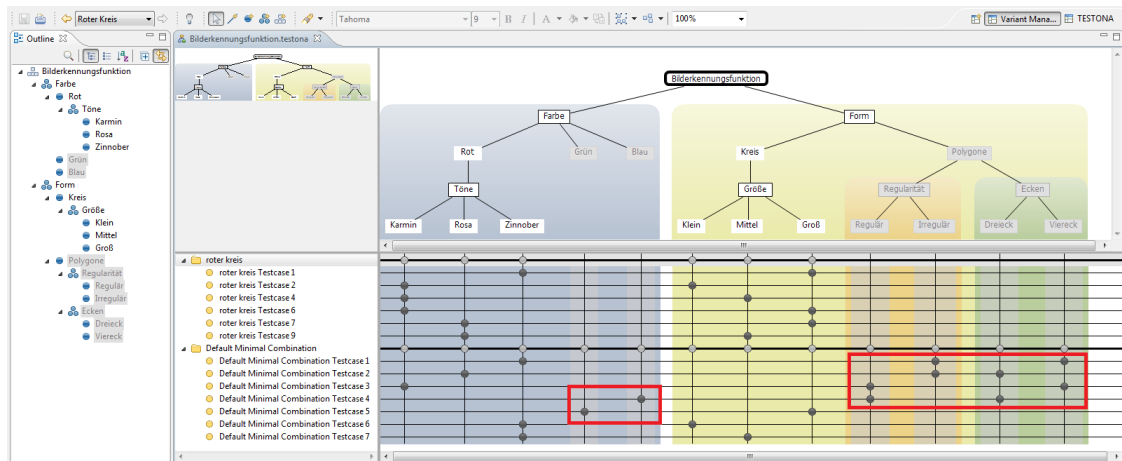


Abbildung 2.2: Ungültige Auswahl der Klassen (Grün und Blau) und Klassifikationen (Regularität und Ecken) für die Testfallgenerierung

## 2.2 Variantenmanagement Software

Die Erweiterung des schon vorhandenen Variantenmanagements in TESTONA wird dazu beitragen, eine bessere Kopplung zwischen DOORS und TESTONA zu erzielen. Nach der Recherche über schon vorhandene Lösungen zu dieser Problemfrage bin ich auf keine brauchbare Implementierung die in TESTONA angewendet werden kann gestoßen. Da es unter dem Begriff „Variante“ viel Freiraum gibt, sind vorhandenen Lösungen sehr problemspezifisch.

Im englischen Sprechraum werden die Begriffe „feature modelling“ und „domain engineering“ benutzt. Unter „domain engineering“ ist die Wiederverwendung von Wissensgebieten für die Produktion einer neuen Software zu verstehen [14]. Obwohl beide Begriffe sich auf die Softwareentwicklung spezialisieren, werden die Prinzipien allgemein in der Produktion verschiedener Produkte verwendet. Betrachtet man TESTONA, so sind die Editionen Light, Express, Professional und Enterprise Varianten.

Es existiert bereits Software, die das Variantenmanagement unterstützt. Eine davon ist *pure::variants*.<sup>3</sup> Dieses Programm ist in der Lage sich mit DOORS zu verbinden und unterstützt Variantenmanagement. Es unterstützt auch Parameterersetzung in den jeweiligen Varianten anhand so genanntes „Parametrized Text“. Hier werden Parameterwerte in das *pure::variants* Programm eingetragen, die danach in die DOORS Anforderungen ersetzt werden. Das Programm dient als reines Variantenmanagementsystem für die Anforderungen und bietet nicht die Möglichkeit Testfälle zu erzeugen.

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 implementiert kein Variantenmanagement.<sup>4</sup>

---

<sup>3</sup><http://www.pure-systems.com>, 15/11/2014

<sup>4</sup><http://www.razorcat.eu/cte.html>, 15/11/2014

---

## 2.3 Neue Anforderungen

Im Laufe des Projektes haben sich die Anforderungen leicht verändert. Die Parameterersetzung wird als Hauptpunkt dieser Arbeit betrachtet. Die in DOORS Module definierten Parameterwerte sollen in TESTONA importiert und angezeigt werden. Die Zuordnung von Parametern an Bauelemente erfolgt über die Verknüpfung von (aus DOORS importierten) Anforderungen mit einem Bauelement. In jeder dieser Varianten kann der Parameter einen anderen Wert darstellen. Bei Änderung der Variantenansicht in TESTONA soll der jeweilige Parameterwert für die aktive Variante angezeigt werden.

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 sollen Optimierungsschritte bei der Testgenerierung betrachtet werden. Anhand der Parameterwerte kann es dazu kommen, dass doppelte Testfälle erstellt werden. Wenn die Endknoten unter einen Knoten die gleichen Werte besitzen, können vorher gültige Testfälle dadurch verdoppelt werden. Dabei soll der Benutzer auf die doppelten Testfälle aufmerksam gemacht werden oder diese Testfälle sollen automatisch gelöscht werden.

---

# 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, automatisch 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 Blackbox-Tests im Bereich von eingebetteter Software [7]. 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) <sup>5</sup> programmiert und unterstützt Partitionierung und Testfallgenerierung. Das Werkzeug von CPM konnte nur Testfälle generieren ohne Bestimmung der Testaspekte [11].

Diese Methode besteht aus zwei wichtigen Schritten:

- Bestimmung der Klassifikationen (testrelevante Aspekte) und Klassen (mögliche

---

<sup>5</sup>Entwickelt von Grochtmann und Wegener[13]. Bei Berner & Mattner inzwischen zu TESTONA umbenannt

---

Ausprägungen).

- Erzeugung von Testfällen aus Kombinationen von unterschiedlichen Klassen für alle Klassifikationen.

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 [10]. 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).

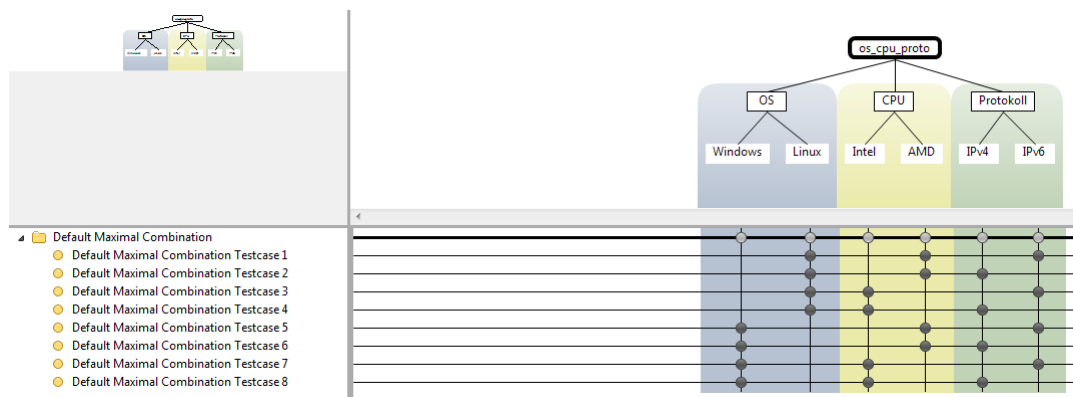


Abbildung 3.1: Baum und Testfälle ohne Kombinatorik

Verwenden wir dafür den kombinatorischen Testentwurf „paarweise Kombination“ (in TESTONA: *pairwise(OS, CPU, Protokoll)*), hätten wir nur vier Testfälle (siehe Tabelle 3.1). 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	IPv4
8.	Windows	Intel	IPv6

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.

Verschiedene Experimenten haben gezeigt, dass durch die Verwendung von der paarweisen Kombinatorik die gleichen oder meistens mehrere Fehler entdeckt wurden, als mit der manuellen Testauswahl[9].<sup>6</sup> 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).

### 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 verschiedene 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-Glossar<sup>7</sup> 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]*

---

<sup>6</sup>Basierend auf funktionelle und technische Anforderungen, Use-Cases

<sup>7</sup>International Software Testing Qualification Board

---



Anhand der generierten Testfälle und durch 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 niedrig) des Systems vereinbart. Das soll heißen, dass die Testtiefe für ein Flugzeug (Kritikalitätsstufe = sehr hoch<sup>8</sup>) viel höher und genauer ist, als die Kompatibilität eines Bildschirmes mit ein Grafiktreiber (Kritikalitätsstufe = niedrig<sup>9</sup>).

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 Bildschirmes, kann intuitives Testen mit geringer Testtiefe angewendet werden.[17]

### 3.1.3 Abhängigkeitsregeln

Abhängigkeitsregeln werden vereinbart, um überflüssige Testfälle zu vermeiden, bzw. um Vorbedingungen für bestimmte Testszenarien festzulegen. Abhängigkeitsregeln werden mithilfe von boolescher Algebra definiert wie folgende Abbildung 3.2 zeigt:

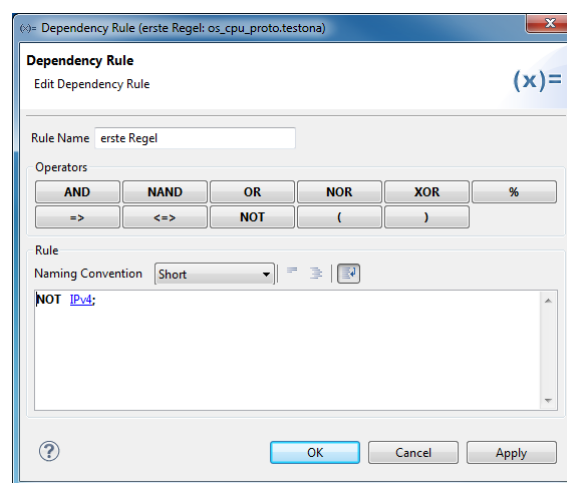


Abbildung 3.2: Abhängigkeitsregeln Bearbeitung

Boolesche Operatoren:

---

<sup>8</sup>Das Fehlverhalten kann zu Verlust von Menschenleben führen, die Existenz des Unternehmens gefährden.

<sup>9</sup>Das Fehlverhalten kann zu geringen materiellen oder immateriellen Schäden führen.

---

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 wurde folgende Abhängigkeitsregel definiert:

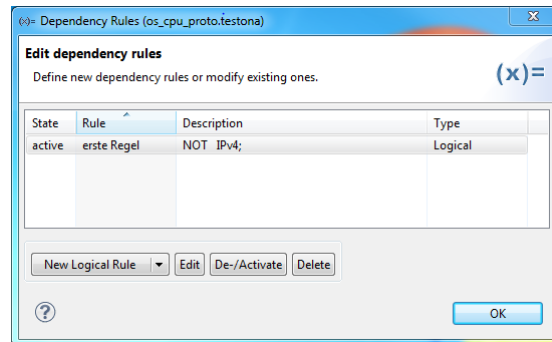


Abbildung 3.3: Abhängigkeitsregeln Übersicht

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

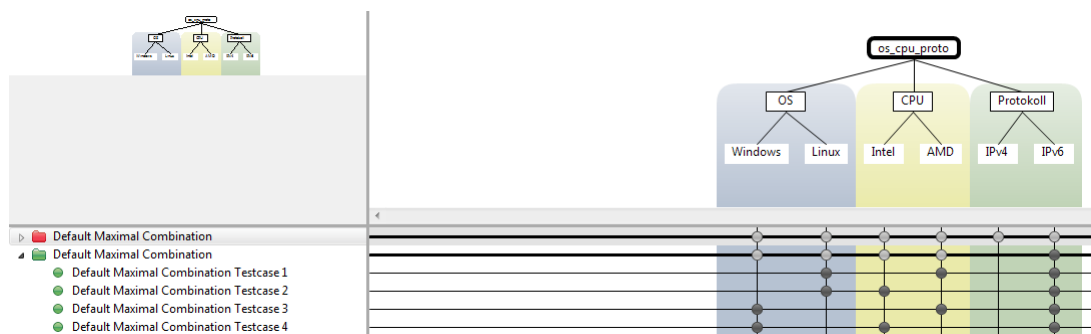


Abbildung 3.4: Testfälle mit Anwendung der Abhängigkeitsregeln aus 3.2 und 3.3

In diesem einfachen Beispiel wird nur ein Parameter ausgeschlossen, aber durch die Abhängigkeitsregeln kann der Tester durchaus komplexere Fälle einleiten. Betrachten wir folgendes Szenario: Ein Licht wird mit einer 5V Spannung durch eine Batterie versorgt. Die Spannung der Batterie befindet sich momentan im niedrigen Bereich. Das System hat zwei Lichtschaltern um das Licht zu steuern. Es soll die Reaktion des Systems getestet werden, wenn eins der beiden Lichtschalter betätigt wird. Um so ein Fall zu prüfen, werden zwei verschiedene Regeln angelegt:

1. *Lichtschalter1 XOR Lichtschalter2*

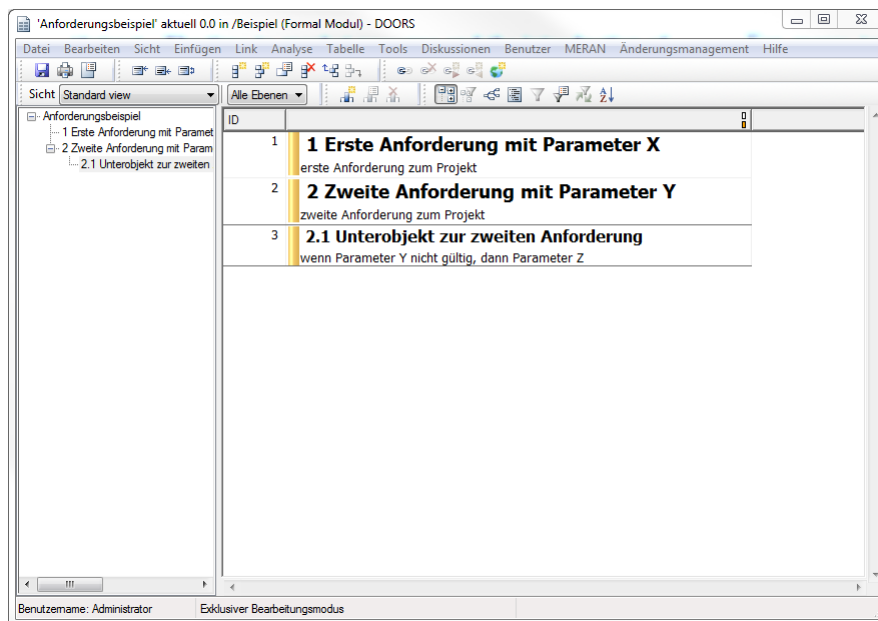
## 2. *Spannung NOT 5V*

Die erste Regel besagt, dass mindestens einer der beiden Lichtschalter betätigt werden muss, um das Licht einzuschalten. Die zweite Regel spezifiziert, dass nur der Fall betrachtet wird, wenn die Spannung der Batterie unter die 5V Grenze liegt. Daraus werden nur Testfälle erzeugt, die diese beide Kriterien erfüllen.

---

## 3.2 IBM Rational DOORS

Quality Systems & Software (QSS) hat 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 die Anforderungen und die zugehörige Eigenschaften abgelesen werden. Eigenschaften sind eine eindeutige Identifikationsnummer, sowie vom Benutzer ausgewählte Attribute.



The screenshot shows the IBM Rational DOORS application window titled "'Anforderungsbeispiel' aktuell 0.0 in /Beispiel (Formal Modul) - DOORS". The interface includes a menu bar (Datei, Bearbeiten, Sicht, Einfügen, Link, Analyse, Tabelle, Tools, Diskussionen, Benutzer, MERAN, Änderungsmanagement, Hilfe) and a toolbar. On the left, a tree view shows a project structure: "Anforderungsbeispiel" containing "1 Erste Anforderung mit Paramet", "2 Zweite Anforderung mit Param", and "2.1 Unterobjekt zur zweiten". The main area displays a table with three rows of requirements.

ID	
1	<b>1 Erste Anforderung mit Parameter X</b> erste Anforderung zum Projekt
2	<b>2 Zweite Anforderung mit Parameter Y</b> zweite Anforderung zum Projekt
3	<b>2.1 Unterobjekt zur zweiten Anforderung</b> wenn Parameter Y nicht gültig, dann Parameter Z

At the bottom of the window, it shows "Benutzername: Administrator" and "Exklusiver Bearbeitungsmodus".

Abbildung 3.5: Beispiel einer Tabelle in DOORS

In DOORS wird eine Tabelle „Modul“ genannt. Jede Zeile innerhalb eines Moduls ist ein Objekt und die Spalten für jedes Objekt bezeichnet man als Attribut. Ein Objekt kann Unterobjekte besitzen, indem Alternativen und weitere Anforderungen beschrieben werden (siehe Abbildung 3.5).

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 eXtention Language) erhält TESTONA Zugriff auf die gespeicherten Module in DOORS [8] [18].

TESTONA besitzt Klassen die DXL-Skripte ausführen, damit der Entwickler einfach und sicher Daten in DOORS abrufen kann.

### 3.3 Variantenmanagement

Variantenmanagement, wie das Wort schon verrät, behandelt verschiedene Varianten eines Produktes. Um eine Variante besser zu definieren, betrachten wir folgendes Beispiel: ein Auto „A“ soll in verschiedenen Modellen gebaut werden: Kombi, Limousine und Cabrio. Alle Modellen von „A“ haben die Eigenschaften eines Wagens (4 Räder, Personenkraftwagen, Türen, etc), aber sie unterscheiden sich untereinander durch die Anzahl der Passagiere oder der Größe des Wagens. Daher kann jedes gebaute Modell von „A“ als eine Variante betrachtet werden.

Anhand des Beispiels 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 muss (mehrere Testfälle sind nötig, neue Parameterwerte). Das heißt, wenn ein Auto als Kombi gebaut wird und danach als Cabrio angeboten wird, müssen (unter anderem) die ganze Dachfunktionen geprüft werden. Das führt dazu neue Testspezifikationen, Testabläufe und Testfälle zu erstellen, die die Funktionalität des Daches überprüfen.

Varianten werden oft mit Features verwechselt. Der Unterschied zwischen ein Feature und einer Variante ist sehr fein und oft abhängig vom Betrachtungspunkt. Um dem Unterschied deutlicher zu machen, werden Änderung in der Funktionalität oder Konstruktion des Autos als einer Variante betrachtet.

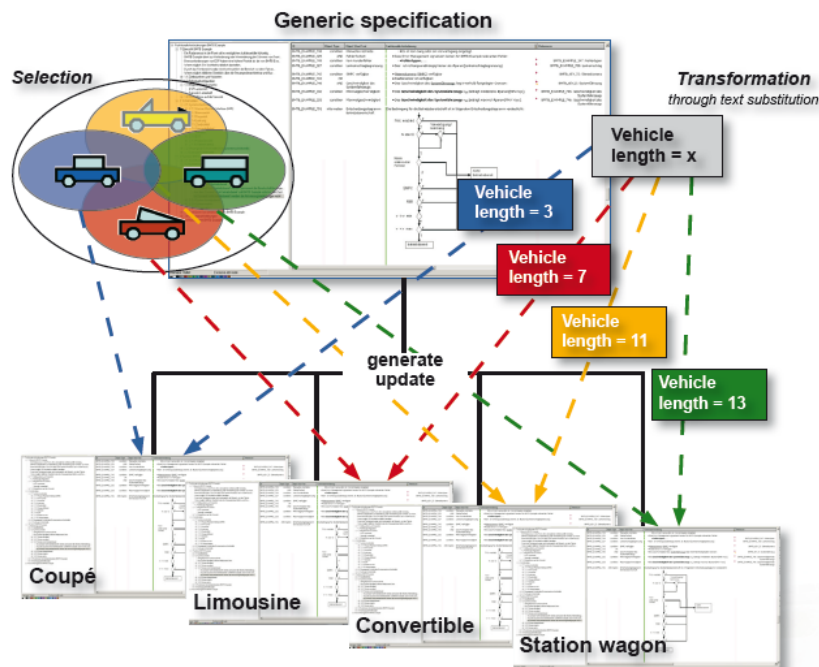


Abbildung 3.6: Betrachtung von Varianten eines Wagens von der generischen Testspezifikation zur variantenspezifische Testspezifikation (aus [16]).

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 Autos auswirkt (somit werden die Tests oder Testabläufe nicht beeinflusst). Ein Auto mit verschiedener Lackierung des selben Modells erzeugt keine Änderung im Testaufwand[15].

Das Variantenmanagement wird im Allgemeinen dazu benutzt, um bei komplexere Produkte einen besseren Überblick zu haben. Hinsichtlich das Testen können Änderung des Produktes besser erkannt werden. Somit werden die Testfälle und Testabläufe an die jeweiligen Variante angepasst. Die Abbildung 3.6 betrachtet die Länge jedes Modells und erstellt aus die generische Spezifikation einzelnen Spezifikationen für jede Variante. Die Testfälle werden mit den richtigen Werten erstellt und eine bessere Testabdeckung kann gewährleistet werden.

In TESTONA werden die in DOORS definierten Produktvarianten importiert. Bauelement werden an der jeweiligen Variante hinzugefügt. Wenn der Benutzer eine Variante auswählt, werden nur die in dieser Variante gültige Bauelemente angezeigt. So wird die Übersicht verbessert und der Tester kann Testfälle genauer erstellen, ohne dies für jede einzelne Variante wiederholen zu müssen.

## 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 „IBM 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 [19], [3]. Dies ist einer der Hauptgründe, warum TESTONA zu Java und Eclipse migriert wurde. Durch RCP können verschiedene Programmversionen (Light, Express, Professional, Enterprise) besser verwaltet werden. Durch die RCP Architektur werden auch verschiedene Funktionalitäten voneinander getrennt. Das hilft bei der Weiterentwicklung sowie bei der Pflege des Programms, da mehrere Entwickler gleichzeitig an verschiedenen Plug-ins arbeiten können. Der Aufwand wird niedrig gehalten, weil nur ein Workspace genutzt wird. Bei der Erstellung von verschiedenen Versionen werden nur die Plug-ins geladen, die nötig sind.

### 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](http://marketplace.eclipse.org) können auch von Privatentwicklern programmierte Plug-Ins heruntergeladen werden [3].

---



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

Testona besteht momentan aus viele verschiedene 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 Enviroment, 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 Swing,<sup>10</sup> 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

---

<sup>10</sup>Programmierschnittstelle und Grafikbibliothek für Java zum programmieren von grafischen Benutzeroberflächen.

---

sofort Änderungen in das „look and feel“ des Betriebssystems in der Anwendung aktualisieren und beinhaltet ein konstantes Programmiermodell in alle Plattformen [5].

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,
- 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

SWT wird in TESTONA eingesetzt, damit das Aussehen der Benutzeroberfläche automatisch aktualisiert an des jeweilige Betriebssystem angepasst wird. Es erspart auch sehr viel Entwicklungszeit, da die Grafikkomponenten des Programms nicht für jedes Betriebssystem programmiert werden müssen.

### 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, dass JFace benutzt [4].

### 3.4.5 Tags

Ein *Tag* ist ein Objekt, das auf EMF (Eclipse Modeling Framework) basiert. Es ist eine Eigenentwicklung von TESTONA und nicht teil von Eclipse. EMF ist ein Java Framework der Eclipse-Open-Source Gemeinschaft für die automatische Erzeugung von Quelltext aus Modellen [19]. Bei EMF wird die Problemstellung in einem Klassendiagramm beschrieben. Hier werden die Eigenschaften und Attribute der Klassen eingetragen um daraus wird der Quelltext generiert. Die Objekte können ohne großen Aufwand serialisiert und validiert werden und mögliche Fehlerquellen werden ausgeschlossen. In das Ecore-Modell werden die Klassen definiert und in Pakete eingeteilt. Der erzeugte Quelltext kann dann vom Benutzer ergänzt werden.

*Tags* dienen hauptsächlich dazu um Eigenschaften anderer Objekte zu beschreiben. In *VariantsTag* werden zum Beispiel Varianten gespeichert. Wenn in einem Projekt verschiedene Varianten zur Verfügung stehen, wird es dazu kommen, dass ein Bauelement nicht in allen Varianten auftritt. Das Bauelement wird ein *VariantsTag* besitzen, das spezifiziert, zu welchen Varianten das Bauelement gehört. in *RequirementTags* werden die Anforderungen gespeichert, die an einem Bauelement verknüpft worden sind. Für die farbige Trennung zwischen Knotenpunkten im Klassifikationsbaumeditor ist ein *ColoringTag* zuständig.

Die Grundstruktur eines *Tags* besteht aus:

- Name,
  - Identifikationsnummer,
  - Typ,
  - Inhalt,
  - Zugriffsrechte.
-

Je nach Zweck können die Klassen um weitere Attribute erweitert werden. Durch die Vererbung und Quelltextgenerierung wird sichergestellt, dass alle *Tags* innerhalb TESTONA kompatibel sind. Anhand der Serialisierung können die *Tags* in die TESTONA Datei im XML-Format gespeichert werden und auch ausgelesen werden. Um Parameter in Bauelementen zu speichern, wird auf diesem Modell aufgebaut. Weitere Erörterungen in Kapitel 4.1 und 5.1.3 .

# 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 eintragen:

#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. Es gibt eine gesonderte Tabelle für Parameter, da 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 richtigen 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 *Tag* Objekten gespeichert 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 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 dieser Vorgehensweise 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, 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 werden. 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 Baumelementes gespeichert wird (als ein *Tag* Objekt). Die Darstellungsstruktur für die Variantenansicht ist folgendermaßen definiert:

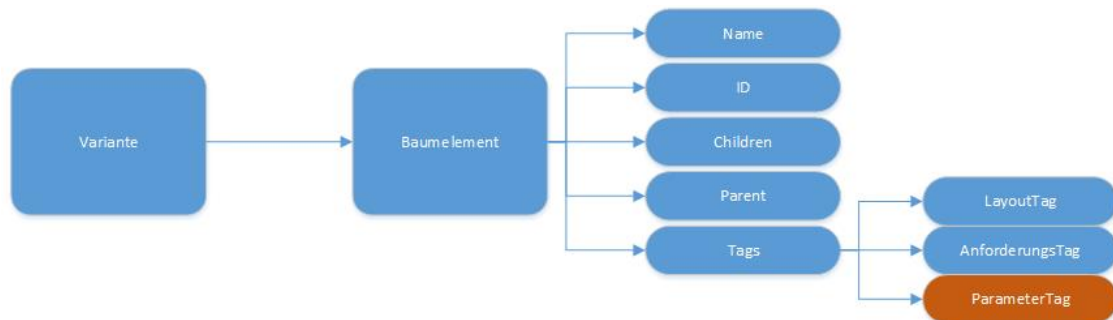


Abbildung 4.1: Darstellung der TESTONA Objekte für die Parameterspeicherung

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

**Anforderung 1:** Das Auto hat eine Maximalgeschwindigkeit von  
#param[max\_Geschwindigkeit] km/h.

Variante	max_Geschwindigkeit	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 Baumelement verschiedene Parameter repräsentieren kann. Wenn ein Baumelement 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 Anforderungen (innerhalb eines Baumelementes) nicht gelöscht oder überschrieben werden.

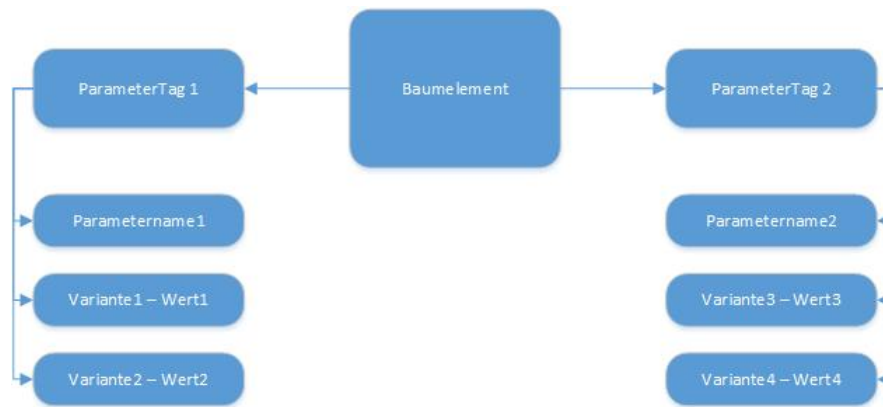


Abbildung 4.2: Darstellung eines Bauelementes mit zwei verschiedenen Parametern und vier Varianten



## 4.2 Visualisierung der Parameterwerte

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, muss die Beschriftung (*Label*) des Bauelementes (Maximalgeschwindigkeit von Generic = X, Cabrio = 150, Kombi = 200) aktualisiert werden.

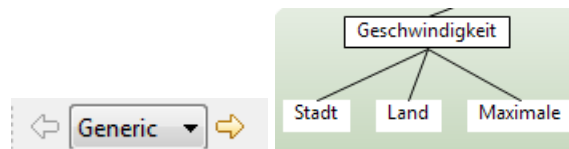


Abbildung 4.3: Aktive Variante Generic (default)

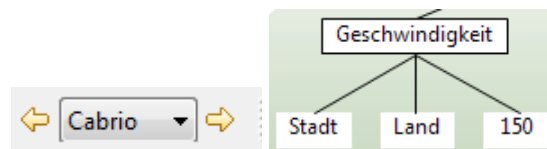


Abbildung 4.4: Aktive Variante Cabrio

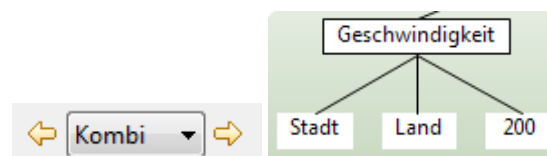


Abbildung 4.5: Aktive Variante Kombi

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

## 4.3 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. Die Trennung der Ansichten trägt sehr viel zur Übersicht bei.

Als Ergebnis wurde festgestellt, dass die in dieser Arbeit gestellte Problemfrage nicht zur Unübersichtlichkeit der Benutzeroberfläche beiträgt. Es wird erhofft, dass außer der Erweiterung der Funktionalität 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 von der aktiven Variante wird in diesem Kapitel erläutert.

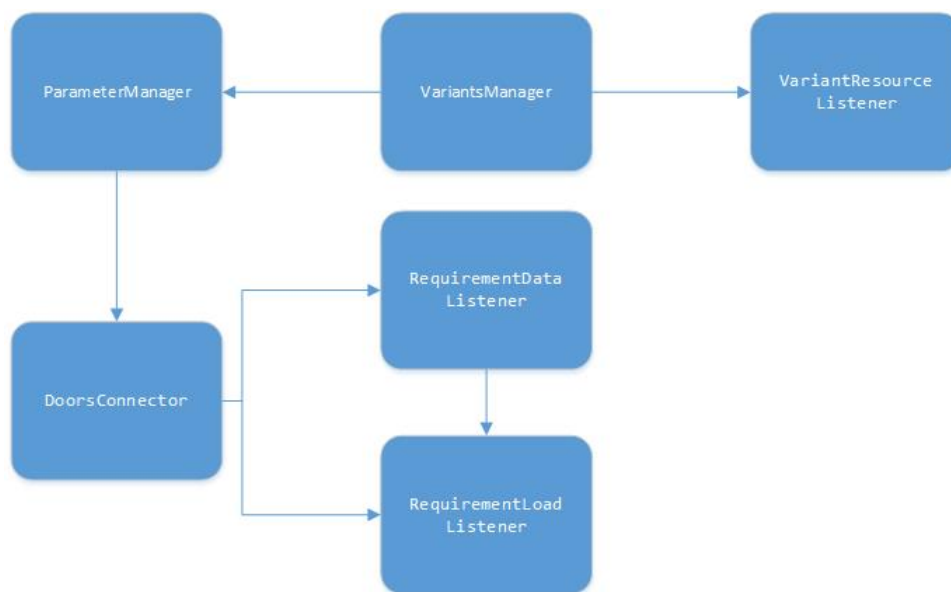


Abbildung 5.1: Einfache Übersicht der Klassen

Die genaue Darstellung der Klassen befindet sich im Anhang A.2. Ablauf für das Speichern eines Parameters in einem Bauelement:

1. Der Benutzer verknüpft eine Anforderung an ein Bauelement.
  2. Der *Listener* meldet, dass eine Änderung vorgenommen wird und gibt die Informationen weiter.
  3. Der *VariantsManager* lädt und übergibt die nötigen Informationen an den *ParameterManager*.
-

4. Der *ParameterManager* lädt, sucht und erstellt ein *ParameterTag* und gibt den Befehl, dass ein *ParameterTag* an ein Bauelement hinzugefügt werden muss.
5. Der *ParameterManager* gibt den Befehl an den *Listener* weiter.
6. Der Befehl wird an die Befehlskette angehängt.

In den nächsten Unterkapiteln wird die Aufgaben der jeweiligen Klassen erläutert und der Ablauf ausführlicher erklärt.

### 5.1.1 VariantsManager

Die Hauptaufgabe des *VariantsManager* ist die Zuordnung der Varianten. Hier werden Bauelemente zu Varianten hinzugefügt und gelöscht. Dabei muss der Klassifikationsbaum neu gezeichnet werden. Diese Klasse kümmert sich auch um die Umschaltung zwischen Varianten in der Variantenansicht (siehe Abbildung 4.3) und um die richtige Darstellung der Bauelemente mit aktuellen Informationen.

Für die Zwecke dieser Arbeit wurde die Klasse erweitert. Um die Aufgaben dieser Klasse zu erläutern, wird der Vorgang des Hinzufügen eines Parameters an einem Bauelement dargestellt. Der *VariantsManager* setzt die Bauelement - ID und die Anforderungskennung in der Klasse *ParameterManager*. Die Klasse dient als Schnittstelle zwischen dem *Listener* (siehe Kapitel 5.1.2) und der Klasse *ParameterManager* (siehe Kapitel 5.1.4).

Der *Listener* hat die Aufgabe dem *VariantsManager* zu melden, wenn eine Änderung<sup>11</sup> geschehen wird oder geschehen ist. Im *Listener* werden die benötigten Informationen gefiltert und an die Klasse weitergegeben, damit Änderung vorgenommen werden können. Im Fall der Parameterersetzung sind die Identifikationsnummer eines Bauelementes und die Anforderungskennung der verlinkten Anforderung. Diese beiden Kennungen dienen dazu, dass der *VariantsManager* die jeweiligen Objekte (ein Bauelement und eine Anforderung) laden kann. Diese Objekte werden dann der Klasse *ParameterManager* übergeben.

Das angesprochene Bauelement muss aus dem TESTONA-Diagramm geladen werden. Anhand der Identifikationsnummer werden die verschiedenen Bauelemente einzeln abgefragt bis das richtige Bauelement gefunden wurde. Danach wird das Bauelement dem *ParameterManager* Objekt übergeben.

Im Fall von der Anforderungskennung, werden mehrere Informationen übergeben. Die Anforderungskennung wird in einer lokalen Variablen vom *ParameterManager* Objekt gespeichert. Damit der Inhalt der Anforderung gelesen werden kann, müssen alle in TESTONA gespeicherte Anforderung dem *ParameterManager* Objekt übergeben werden. Die gespeicherten *Tags* des TESTONA-Objekts werden abfragt, um folgende Objekte an das *ParameterManager* Objekt übergeben zu können:

---

<sup>11</sup> Aktionen werden vom Benutzer ausgelöst anhand von Bedienelemente in der Benutzeroberfläche.

- **RequirementsConnetionTag**: beinhaltet die Information zur Datenbankverbindung, unter anderem die Verbindungsidentifikation (DOORS), das Modul(Tabelle) in der die Anforderung gespeichert ist und die Schnittstelle zur Datenbank. Anhand dieser Informationen wird später die Verbindung zur Datenbank aufgebaut.
- **RequirementsListTag**: ist eine *Map* mit allen in TESTONA gespeicherten Anforderungen. Das *Key* ist die Anforderungsidentifikationsnummer und der Wert der Inhalt der Anforderung. Eine Anforderung aus dieser Liste wurde mit dem Baumelement verknüpft.
- **VariantListTag**: ist eine Liste mit allen in TESTONA definierten Varianten.

Anhand dieser Informationen kann die Klasse *ParameterManager* ein Parameter an ein Baumelement hinzufügen. Parameter können nicht nur hinzugefügt werden, sondern auch gelöscht werden. Ein Parameter wird aus einem Baumelement gelöscht, indem die verknüpfte Anforderung vom Baumelement entfernt wird. Der Ablauf für das Löschen eines Parameter ist im Grunde der gleiche wie beim Hinzufügen. Als nächstes werden die Funktionen des *Listeners* erörtert.

### 5.1.2 ResourceSetListener

Wenn der Benutzer eine Anforderung mit einem Baumelement verknüpft, dann muss der *Listener* das Geschehen abfangen. Er muss so programmiert werden, dass das Interface *ResourceSetListener*<sup>12</sup> implementiert wird. Da der *VariantsManager* einen solchen *Listener* bereits besitzt, wurde dieser lediglich erweitert. Die Aufgabe des *ResourceSetListener* ist, den Benutzer zu benachrichtigen, wenn sich eine Ressource ändert. Der *Listener* „horcht“ auf die ankommenden Nachrichten (*ResourceSetChangeEvent*) und wertet den Inhalt dieses Events aus.

Der *Listener* implementiert zwei Methoden, die für das Abfragen der Events relevant sind. Eine davon heißt *resourceSetChanged(Event e)*. Diese Methode wird aufgerufen, wenn sich eine Ressource ändert. Am Anfang wurde diese Methode für das Abhören von Events (wenn eine Anforderung zu einem Baumelement verknüpft wird) favorisiert, um danach die Parameterersetzung zu triggern. Nach der Implementierung konnte ich feststellen, dass die Methode keine Schreibrechte auf die Baumelemente besitzt. Da sich die Ressource zu diesem Zeitpunkt schon geändert hatte.

Um die Schreibrechte zu besitzen, habe ich dann die Methode *transactionAboutToCommit(Event e)* betrachtet. Diese Methode wird aufgerufen, bevor eine Änderung vorgenommen wurde. Es erfolgt eine Benachrichtigung, dass eine Änderung erfolgt ist und welche Objekte (ein Baumelement und die angehängte Anforderung) betrachtet werden. Zu diesem Zeitpunkt besitzt die Klasse noch Schreibrechte auf die Baumelemente und kann die Änderung vornehmen. Außerdem hat diese Methode als Rückgabewert ein *Command*. So können Kommandos an der Commandstack angekettet werden. Dank der Anordnung des

---

<sup>12</sup><http://download.eclipse.org/modeling/emf/transaction/javadoc/1.0.3/org/eclipse/emf/transaction/ResourceSetListener.html>

Commadstacks wird der Befehl, ein Parameter an ein Bauelement hinzuzufügen, zum richtigen Zeitpunkt ausgeführt. Mehr Informationen zu Kommandos werden in Kapitel 5.1.5 aufgeführt.

Die empfangene Nachricht beinhaltet folgende Elemente:

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

Jetzt wird als erstes der *Notifier* abgefragt, um auswerten zu können, ob die Nachricht relevant ist. Es müssen drei Fälle unterschieden werden. Eine Anforderung wird:

1. zum ersten Mal an einem Bauelement verknüpft,
2. an einem Bauelement verknüpft, an dem schon andere Anforderungen verknüpft sind,
3. gelöscht.

In allen drei Fällen muss der *Listener* wissen, an welchem Element das Event ausgelöst wurde und welche Anforderung hinzugefügt oder gelöscht wurde. Zu diesem Zeitpunkt kann noch nicht festgestellt werden, ob in der Anforderung bereits ein Parameter definiert ist.

### Eine Anforderung hinzufügen

Ist der *Notifier* eine Instanz von *TestonaClass*, so wurde eine Anforderung zum ersten Mal an ein Bauelement verknüpft oder gelöscht. Um unterscheiden zu können, werden die Werte von *NewValue* und *OldValue* ausgewertet. Wenn *NewValue* eine Instanz von *RequirementTag* ist, dann wurde eine Anforderung hinzugefügt. Aus dem *Notifier* wird die ID (repräsentiert das Bauelement) und aus der Variable *NewValue* die Anforderungskennung (diese wurde in DOORS vergeben) abgefragt. Beide Werte werden dem *VariantsManager* weitergegeben, um danach den Befehl als Rückgabewert für das Hinzufügen eines *ParameterTags* zu bekommen.

Der Befehl wird nicht sofort als Rückgabewert der Methode *transactionAboutToCommit* weitergegeben. Der Grund für diese Maßnahme heißt, dass der Rückgabewert auch *null* sein kann. Wenn kein Parameter in der Anforderung definiert ist, so muss auch kein *ParameterTag* an das Bauelement hinzugefügt werden und es muss kein Kommando ausgeführt werden. In der Methode *transactionAboutToCommit* werden auch andere

---

*Events* ausgewertet, die ebenfalls Kommandos ausführen. Darum wird eine lokale Variable *command* definiert. Eine Eigenschaft der Klasse *Command* ist, dass Kommandos aneinander angehängt werden können.

```
1 Command command = null;  
2 command = chain(command, manager.getParameter());
```

Das Kommando für das Hinzufügen eines *ParameterTags* wird an der lokalen Variable *command* angehängt. Davor überprüft die Methode *chain*, ob einer der Parameter den Wert *null* hat. Wenn keiner der Funktionsparameter den Wert *null* hat, wird die Methode *chain(Command cmd)* der Klasse *Command* aufgerufen.

```
1 command.chain(CommandToAddParameter);
```

Der Vorgang für das Kommando wird für jedes zurückgegebene Kommando durchgeführt, unabhängig vom Kommando (hinzufügen oder löschen). Am Ende der Methode kann das Kommando (wahrscheinlich bestehend aus mehrere Kommandos) als Rückgabewerte gegeben werden.

Wenn das Bauelement mindestens eine Anforderung besitzt und eine weitere Anforderung hinzugefügt wird, ist der *Notifier* eine Instanz von *RequirementTag* und *OldValue* == *null* && *NewValue* != *null*. Aus dem *Notifier* wird die ID des Bauelements abgerufen und *NewValue* besitzt die Anforderungskennung. Bevor die neue Anforderung hinzugefügt wird, werden die alten Anforderungen gelöscht und alle Anforderung neu hinzugefügt.

### Anforderungen löschen

Ist der *Notifier* eine Instanz von *RequirementTag*, so wird eine Anforderung von einem Bauelement gelöscht oder es wird eine neue Anforderung an ein Bauelement hinzugefügt, in dem bereits eine Anforderung verknüpft ist. Wird eine Anforderung gelöscht, sind die Werte *OldValue* != *null* && *NewValue* == *null*. In *OldValue* befindet sich die Anforderungskennung, die gelöscht werden soll. Der Löschvorgang teilt sich in zwei Fälle.

Der erste Fall beschreibt, wenn eine Anforderung aus einem Bauelement gelöscht wird, die nur eine Anforderung beinhaltet. Dieses *Event* teilt sich in zwei Nachrichten. Die erste Nachricht beinhaltet die Anforderungskennung als *String*. Die Anforderungskennung wird dann in einer lokalen Variable gespeichert, die mit der zweiten Nachricht ausgewertet wird. In der zweiten Nachricht ist der *Notifier* eine Instanz von *TestonaClass*. Es unterscheidet sich vom Hinzufügen einer Anforderung, weil die Variable *OldValue* eine Instanz von *RequirementTag* ist. Zur Sicherheit wird abgefragt, ob die lokale Variable mit der Anforderungskennung ungleich *null* ist. Danach kann aus dem *Notifier* die ID des Bauelements abgefragt werden. Beide Werte werden an den *VariantsManager* übergeben und als Rückgabewert wird der Befehl für das Löschen eines *ParameterTags* erwartet.

Der zweite Fall beschreibt, dass eine oder mehrere Anforderungen aus einem Bauele-

ment gelöscht werden, wenn ein Bauelement mindestens eine Anforderung besitzt. Bevor die Anforderung hinzugefügt werden kann, werden die im Bauelement beinhaltenen Anforderungen zuerst gelöscht. Danach werden alle alten Anforderungen neu hinzugefügt und auch die neue Anforderung.

Besitzt das Bauelement mehr als eine Anforderung und eine neue wird hinzugefügt, kann aus dem *Notifier* die ID des Bauelements abgefragt werden. Der Unterschied liegt daran, dass *OldValue* jetzt eine Liste von *String*werten ist. Jeder Wert in der Liste repräsentiert eine Anforderungskennung, die gelöscht werden soll. So wird mit einer Schleife durch alle Elemente der Liste iteriert und jede Anforderung aus dem Bauelement gelöscht.

### 5.1.3 Parameter-Tag

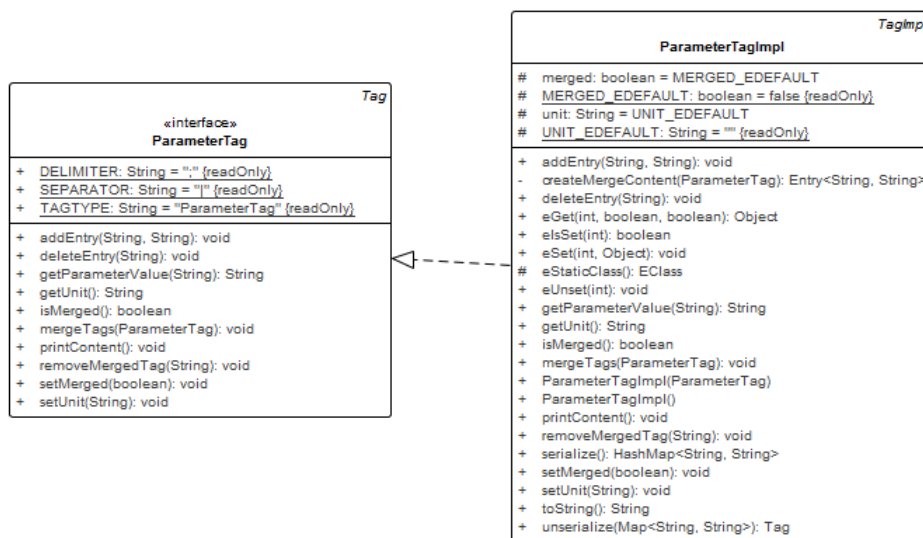


Abbildung 5.2: Darstellung der Klasse *ParameterTagImpl* und das Interface *ParameterTag*

#### ParameterTag

Der Zweck vom Interface *ParameterTag* ist, als Schnittstelle für den Zugriff auf den Inhalt von *ParameterTagImpl* zu dienen. Das Interface erweitert *Tag*. Das *Tag* Interface ist ein in TESTONA allgemein implementiertes Modell, welches auf ein Interface und eine implementierende Klasse basiert (siehe Kapitel 3.4.5). In das Interface wurde immer der Tagtyp definiert um *Tags* voneinander unterscheiden zu können.

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

Das *Tag* wurde um die in Abbildung 5.2 zu sehende Methoden erweitert. Die Konstanten *DELIMITER* (,,|“) und *SEPARATOR* (,, ;“) wurden benutzt um den Inhalt des Tags beim Speichern und Lesen zu trennen.



## ParameterTagImpl

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

Jedes *Tag* hat als Eigenschaft einen Namen. In dieser Klasse entspricht der Name einem Parameternamen, der aus der Parametertabelle gelesen wurde. Weiterhin ist über *ID* eine eindeutige Identifikationsnummer definiert, um *Tags* des gleichen Typs voneinander unterscheiden zu können. Zwei weitere Attribute wurden implementiert. Das erste Attribut ist die Einheit (*unit*) des Parameters. Das zweite Attribut ist der Status *merged*. Dieser besagt, ob der *ParameterTag* ein oder mehrere Parameter beinhaltet.

Das Attribut *merge* wurde vereinbart, weil ein Bauelement nur ein Objekt pro Tagtyp beinhalten darf. Wenn ein Bauelement zwei oder mehrere Parameter repräsentiert, müssen sich die Werte der Parameter in einen *ParameterTag* befinden. Diese Änderung wurde an den Ansatz (siehe Abbildung 4.2) angepasst.

Der Inhalt vom *Tag* ist durch ein *EcoreEMap<String, String>* definiert. Der erste String ist ein Schlüsselwert und der zweite String der Wert. Wenn das *ParameterTag* nur einen Parameter repräsentiert, ist der Schlüssel der Name einer Variante und der Wert ist der Parameterwert des Parameters in dieser Variante. Der Inhalt und Struktur eines *ParameterTags* sieht folgendermaßen aus:

ParameterTag		
ID	30	
Name	max_speed	
Unit	[km/h]	
Merged	false	
Content	Key	Value
	Generic	50
	Cabrio	100
	Kombi	200
	Limo	300

Tabelle 5.1: Beispiel eines ParameterTags

Anhand der Methode *getContent()* wurde der Inhalt eines *Tags* aufgerufen und mit der Methode *addEntry()* wurden Inhalte hinzugefügt. Die Löschung erfolgt mit der Methode *deleteEntry()*. Wie die Methoden genauer angewendet wurden, wird im Listing 5.2 gezeigt.

Wenn ein *ParameterTag* zwei oder mehrere Parameter repräsentiert, sieht die Struktur des *ParameterTags* folgendermaßen aus:

ParameterTag		
ID	31	
Name		
Unit		
Merged	true	
Content	Key	Value
	speed1 Generic;Cabrio	[km/h] 50;100
	speed2 Kombi;Limo	[km/h] 200;300

Tabelle 5.2: Beispiel eines ParameterTags mit zwei Parametern

Wenn ein Bauelement schon einen *ParameterTag* besitzt, wurden der neue Parameter mit dem aktuellen *ParameterTag* vereinigt. Dafür wurde die Methode *mergeTags* (*ParameterTag*) definiert. Diese Methode wandelt die Struktur des *ParameterTags* von Tabelle 5.1 zu der Struktur von Tabelle 5.2 um. Die Statusvariable *merge* wurde dann auf *true* gesetzt. Der Inhalt des *ParameterTags* ändert sich auf folgendes Format:

Key	Value
<i>ParameterName Variante1;Variante2</i>	<i>Einheit Wert1;Wert2</i>

Um ein Parameter aus einem *ParameterTag* mit mehreren Parametern zu löschen, wurde die Methode *removeMergedTag* (*ParameterName*) programmiert. Die Methode sucht den richtigen Eintrag anhand des Parameternames in *content* und löscht den Eintrag. Falls nur ein Eintrag übrig bleibt, bedeutet es, dass es nur ein Parameter im *ParameterTag* vorhanden ist. Wenn dieser Fall auftritt, wurde die Struktur des *ParameterTags* von Tabelle 5.2 wieder auf die Struktur von Tabelle 5.1 umgewandelt.

Sehr relevant sind die Methoden *serialize()* und *unserialize(Map<String, String>)* die für das Schreiben und das Lesen der TESTONA Datei zuständig sind. Zum Schreiben wurde ein *HashMap<String, String>* Objekt zurückgegeben. Das erste *String* beinhaltet den Namen des Parameters gefolgt von den Variantennamen. Der zweite *String* beinhaltet die Einheit des Parameters und die Parameterwerte (in einer geordneten Reihenfolge). Das Format von jedem Eintrag hält sich an das Format von der Tabelle 5.2. Der XML Eintrag für das obige Beispiel (Tabelle 5.1) wurde wie folgt aussehen:

```

1 <Tag id="30" type="ParameterTag">
2   <Content key="max_speed|Generic;Cabrio;Kombi;Limo" value="[km/
   h]|50;100;200;300"/>
3 </Tag/>

```

Listing 5.1: XML Darstellung eines ParameterTags

Beim Laden der TESTONA Datei erkennt das Programm anhand des Attributs *type* um welchen Typ von *Tag* es sich handelt. Die Methode *unserialize* erhält als Eingangsparameter die in *serialize()* erstellte Zeile als ein *Map<String, String>* Objekt. Durch die Trennzeichen (*SEPARATOR* und *DELIMETER*) werden die jeweiligen Werte voneinander unterschieden und den zugehörigen Variablen (*name*, *unit*, *content*) zugeordnet. Beide Vorgänge funktionieren durch die Vererbung automatisiert.

### 5.1.4 ParameterManager



Abbildung 5.3: Klasse ParameterManager

Die Klasse *ParameterManager* überprüft und startet die Parameterersetzung. Sie beinhaltet die private innere Klasse *DoorsConnector*. Diese kümmert sich um den Verbindungsaufbau mit DOORS und das Laden von den Parameterwerten aus der Parametertabelle. Die Klasse *DoorsConnector* wird genauer in Kapitel 5.1.6 erklärt.

Mit dem Aufruf der Methode *addParameterToTreeItem()* wird die Parameterersetzung gestartet. Die Methode hat als Rückgabewert ein Kommando, indem ein Parameter hinzugefügt oder gelöscht wird (siehe Kapitel 5.1.5). Als erstes überprüft die Methode, ob die an das Baumelement verlinkte Anforderung ein Parameter beinhaltet. Falls kein Parameter in der Anforderung gefunden wurde, gibt die Methode den Wert *null* zurück. Wenn ein Parameter gefunden wurde, wird der Name in einer Variablen gespeichert, um diese aus DOORS laden zu können. Wenn noch keine Parameter lokal zu Verfügung stehen (in Form von *ParameterTag* innerhalb einer Liste), wird die Verbindung mit DOORS gestartet. Die erforderlichen Angaben zum Verbindungsaufbau befinden sich in der Variable *im* vom Typ *IndieModul* (diese wurde von der Klasse *VariantsManager* gesetzt aus dem *RequirementsConnectionTag*). Das Objekt der Klasse *DoorsConnector* hat Zugriff auf *im* und kann die gespeicherte Daten abfragen.

Während die Verbindung entsteht und die Parametertabelle geladen wird, wird dem Benutzer ein Dialogfenster angezeigt. Die Benutzeroberfläche wird blockiert und zeigt einen Fortschrittsbalken an. Nach einem erfolgreichen Verbindungsaufbau wird die Parametertabelle geladen. Der Pfad zu der Parametertabelle in DOORS ist als Attribut des Hauptmoduls gespeichert. Im Objekt *im* befindet sich der Pfad zum Hauptmodul. Die aktuell implementierte DOORS API in TESTONA kann diese Attribute nicht abfragen. Eine in

Moment in Entwicklungszustand neue API wird in der Lage sein, die Attribute abfragen zu können. An dieser Stelle wird als Kompromiss eine Konstante angelegt. In ihr befindet sich der Pfad zur Tabelle.

Wenn ein Parameter und die zugehörige Werte geladen worden sind, werden sie in *ParameterTag* Objekt gespeichert. Die Methode `fillParamTagList(Requirement req)` agiert als Parser und wandelt der Übergabeparameter in einen *ParameterTag*. Zu beachten ist, dass innerhalb DOORS jede Zeile in einem Modul als eine Anforderung betrachtet wird. Danach wird die Namensgebung vereinbart und der Übergabeparameter ist vom Typ *Requirement*. Im *Requirement* Objekt befinden sich die Variantennamen und die Parameterwerte. Der folgende Quelltextausschnitt kümmert sich um die Speicherung der Parameterwerte in ein *ParameterTag*.

```

1 ParameterTag tempTag = new ParameterTagImpl();
2
3 for(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        //Füge ein Eintrag hinzu
11        tempTag.addEntry(sAttrList.get(i),
12            req.getValue(attributesList.get(i)).getValueAsString());
13    }
14 paramTagList.add(tempTag);

```

Listing 5.2: Auszug von der Erstellung der ParameterTags

Die Variable `attributesList` beinhaltet die in DOORS definierten Varianten.<sup>13</sup> Die Liste wird iteriert und die Werte werden im *ParameterTag* gespeichert. Zwei Attribute sind für jedes Parameter in der Liste mindestens definiert. Diese sind der Parametername und ein Standartwert. Möglicherweise ist auch die Einheit des Parameters definiert. Die geladenen Werte werden kontrolliert, ob sie ungleich einen leeren *String* sind.<sup>14</sup>, bevor sie in ein *ParameterTag* gespeichert werden.

Wenn alle vorhandenen Varianten für diesen Parameter iteriert worden sind, wird überprüft, ob der Inhalt (die Variable *content* des *ParameterTags*) der temporären Variable `tempTag` nicht leer ist. Es kann vorkommen, dass ein Parameter im Modul definiert ist, aber keine weitere Informationen ausgefüllt wurden. Wenn Inhalte vorhanden sind, wird der *ParameterTag* zur einer Liste von *ParameterTags* hinzugefügt.

<sup>13</sup>Jede Variante in das DOORS Modul wird als ein Attribut definiert und in einer Spalte angezeigt. Wobei nicht jede Spalte des Moduls ein Attribut ist.

<sup>14</sup>Im Ladevorgang einer Anforderung und ihre Attribute in DOORS wird nicht zwischen leeren und befüllte Zellen unterschieden. Der Rückgabewert ist immer ein *String* Wenn die Zelle leer ist, wird ein leerer *String* zurückgegeben.

Nachdem die Parameter geladen werden konnten und die Liste befüllt ist, wird die Verbindung mit DOORS geschlossen. Das Dialogfenster wird danach geschlossen und die Benutzeroberfläche wird freigegeben. Während dessen wird in der Liste der richtige *ParameterTag* geladen und die Methode *getCommandToAddParameter()* wird aufgerufen. Diese Methode erzeugt ein Objekt der Klasse *AddParameterTagCommand*. Das zurückgegebene Kommando wird an die Klasse *VariantsManager* weitergeleitet, die wiederum das Kommando an das *ResourceSetListener* weitergibt. Dort wird das Kommando verarbeitet.

Für den Fall, dass eine Anforderung aus einem Bauelement gelöscht wird, muss zuerst überprüft werden, ob die Anforderung ein Parameter beinhaltet. Wenn ein Parameter gefunden worden ist, wird das *ParameterTag* aus dem Bauelement geladen. Mithilfe der Klasse *RemoveParameterTagCommand* wird der *ParameterTag* aus dem Bauelement entfernt.

### 5.1.5 Kommandos

Für das Hinzufügen bzw. Löschen eines Parameters in einem Bauelement wurden zwei Klassen implementiert. Sie erben aus der Klasse *RecordingCommand*. Die Klasse *RecordingCommand* ist eine partielle Implementierung der Klasse *Command*. Diese Klasse nimmt die Manipulierung der Objekte der Subklasse auf und erzeugt daraus ein Kommando.<sup>15</sup> Eine der implementierten Klassen erzeugt ein Kommando für das Hinzufügen eines *ParameterTags* an einem Bauelement,<sup>16</sup> während der Zweite für das Löschen zuständig ist.<sup>17</sup> Der Konstruktor der beiden Klassen hat die gleichen Parameter. Es wird der aktuelle Editor-Objekt übergeben, sowie das Bauelement ID und das *ParameterTag*.

Die Methode *doExecute()* wird überschrieben um die Änderung vorzunehmen. Als erstes werden alle Bauelemente geladen und iteriert, bis das Bauelement gefunden wird, das benötigt wird (anhand der ID). Soll ein *ParameterTag* hinzugefügt werden, dann wird die Methode *addTag(ParameterTag pTag)* aufgerufen.

Soll ein *ParameterTag* gelöscht werden, dann wird die Methode *removeTag(ParameterTag pTag)* aufgerufen. Danach muss auch der Name des Bauelementes neu gesetzt werden. Wenn ein Bauelement ein Parameter besitzt, je nach aktiver Variante, wird die Darstellung geändert (siehe Kapitel 4.2). Dafür wird der Name des Bauelementes geladen und falls eine Rücksetzung nötig ist, wird diese gemacht. Genauer dazu wird in Kapitel 5.2 veranschaulicht.

---

<sup>15</sup> 14.02.2015; <http://download.eclipse.org/modeling/emf/transaction/javadoc/1.2.3/org/eclipse/emf/transaction/RecordingCommand.html>

<sup>16</sup> *AddParameterTagCommand*.

<sup>17</sup> *RemoveParameterTagCommand*.

---

### 5.1.6 Die DOORS Verbindung

#### DoorsConnector

Die private innere Klasse *DoorsConnector* baut die Verbindung zwischen TESTONA und DOORS auf. Sie implementiert das Interface *IConnectionListener*, die ein *Listener* für die Verbindungs-Events umfasst. Für das Laden von Dateien aus DOORS benötigt die Klasse einen zweiten *Listener* (*IDataListener*) und einen *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 unter anderem die Daten, 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 und auch getrennt. Es werden zuerst die Ordner geladen, danach die einzelnen 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 gespeicherten Verbindungsdaten behandelt. Es können auch 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 zunächst 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 *ConnectionManager* aufgerufen werden.

```

1 try {
2     connector = ConnectorManager.getInstance()
3         .getConnector(im.getInterId());
4     dataInterface = conManager.getNewDataInterface(connector, this
5         );
6 } catch (ExtensionException e) {
7     e.printStackTrace();
8 }

```

Listing 5.3: Verbindungsaufbau

Um dem richtigen `connector` aufzurufen muss aus dem *IndieModul* (`im.getId()`, siehe Kapitel 5.1.1) die Interfacekennung als Übergabeparameter angegeben werden. Dann kann über den *ConnectionManager* ein neues *DataInterface* erzeugt werden. Der `connector` und das aktuelle Objekt (*DoorsConnector*) werden übergeben.

Aus den in TESTONA gespeicherten Verbindungspräferenzen werden die Verbindungsparameter für DOORS geladen. An dieser Stelle braucht das *DataInterface* die nötige *Listener* 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, ist, dass der Verbindungsaufbau in einem neuen Thread stattfindet. Das *DoorsConnector* Objekt wird über den *IConnectionListener* benachrichtigt, ob die Verbindung stattgefunden hat. Die Methode `interfaceConnected` (aus dem *IConnectionListener*) wird aufgerufen, wenn die Verbindung erfolgreich zustande gekommen ist.

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

Listing 5.4: Verbindungsaufbau war erfolgreich

Der gesetzte *RequirementDataListener* benötigt einen *RequirementLoadListener*, dass es eine Rückmeldung gibt, wenn die Zeilen aus einer Tabelle fertig geladen worden sind. Die Tabelle wird anhand der Methode `loadModule` in 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. Näheres dazu wird im Kapitel 5.1.6 erläutert.

Es wird angenommen, dass, wenn der Benutzer einen Parameter ersetzen will, er auch mehrere Parameter ersetzen will. Da die Datenmenge einer Parametertabelle relativ gering ist, wurde hier die Parametertabelle komplett importiert.

Ein weiterer Grund besteht, dass nicht für jeden Parameter erneut eine DOORS Verbindung entstehen muss. So wird die Rechen- und Reaktionszeit von TESTONA so gering wie möglich gehalten. Die Parametertabelle wird erst bei der Verknüpfung einer Anforderung mit einem Bauelement importiert und nicht sofort 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

Der *RequirementDataListener* ist ein Event-Listener. Dieser Listener wartet auf die Rückmeldung eines Moduls, bis das Modul fertig geladen worden ist. Relevant ist die Methode `onModuleLoad`, die die Zeilen aus der Tabelle liest und speichert.

```

1 @Override
2 public 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 geladenen Modul vorhandenen Attribute. Im diesem Fall anhand mit dem Beispiel Auto sind es:

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

Diese Attribute repräsentieren die Varianten und einen Standardwert und den Name des jeweiligen Parameters. 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 wird in Kapitel 5.1.4 erläutert.

Der *RequirementLoadListener* wird in dieser Klasse instanziiert 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 *RequirementDataListener* in der globalen Liste `paramReqList` als ein `Requirement` Objekt gespeichert.

Weiterhin meldet diese Klasse, wenn alle Zeilen aus dem DOORS Modul geladen wurden. Das wartende 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 Darstellung der Parameterwerte

Nachdem Parameter in den Bauelementen vorhanden sind, sollten die Werte angezeigt werden. Wenn ein Pfeil oder die Kombo-Box (siehe Kapitel 4.2) für eine Änderung der Variantenansicht betätigt werden, muss jedes Bauelement überprüft werden. Gibt es einen gültigen Wert für die aktive Variante, muss dieser angezeigt werden.

ParameterTag		
Name	max_speed	
Unit	[km/h]	
Content	Key	Value
	Generic	50
	Cabrio	100
	Kombi	200
	Limo	300

Tabelle 5.3: Parameter *max\_speed*

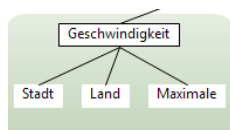


Abbildung 5.4: Der generische Baum

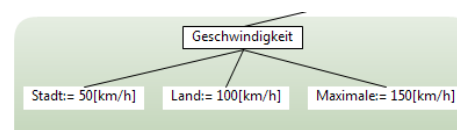


Abbildung 5.5: Aktive Variante Cabrio

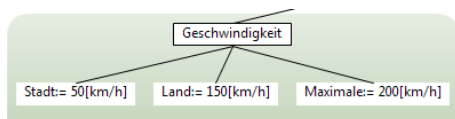


Abbildung 5.6: Aktive Variante Limo



Abbildung 5.7: Aktive Variante Kombi

Wenn die Einheit des Parameters definiert ist, wird diese auch angezeigt. Zwischen Bauelementname und dem Wert befinden sich die Zeichen „:=“ als Trennzeichen. Diese Maßnahme wurde eingeführt, da Bauelemente unter einem Knoten nicht den gleichen Namen besitzen dürfen. So kann das Programm und der Benutzer jedes Bauelemente eindeutig identifizieren.

Nachdem einer der Schaltflächen zum Ändern der Variantenansicht betätigt wird, wird die Methode *activateVariant()* in der Klasse *VariantsManager* aufgerufen. Innerhalb der Methode werden alle Bauelemente, falls es nötig ist, neu benannt.

Die Umbenennung der Bauelemente erfolgt, wie beim Hinzufügen eines *ParameterTags*, über ein Kommando. Im diesem Fall wird kein Objekt des Typs *Command* zurückgegeben, sondern die Methode *executeEMFCommand* bekommt als Parameter das Kommando. Die Methode hängt das Kommando an den Stack und verhindert, dass das

Kommando vom *Garbage Collector* gelöscht wird.

Das Kommando ist eine Klasse in der die *ParameterTags* ausgewertet werden. Wenn das Bauelement ein *ParameterTag* hat, wird überprüft ob für die aktive Variante ein Parameterwert definiert ist. Wenn ein Parameterwert gefunden wurde, wird das Bauelement immer mit dem folgenden Format neu benannt:

$$name := wert[*einheit*]$$

Wobei nur *einheit* ein optionaler Wert ist. Falls kein Parameterwert für die Variante definiert wurden ist wird der Standardname angezeigt (*name*).

Alle Bauelemente müssen aus zwei Gründen geprüft werden. Der erste Grund ist, dass nicht für alle Varianten Parameterwerte zur Verfügung stehen können. Der zweite Grund ist, dass eine Anforderung aus einem Bauelement gelöscht werden kann, nachdem die Werte schon angezeigt worden sind. In beiden Fällen muss der Name des Bauelements auf den Standardname gesetzt werden.

Nachdem alle Bauelemente überprüft worden sind, muss die Anzeige (Editor) von TE-STONA aktualisiert werden. Durch das Ändern der Namen, ändert sich auch die Breite der Bauelemente und diese müssen neu organisiert werden. Danach wird der Befehl gegeben die Anzeige neu zu zeichnen. Das Ergebnis wird in den Abbildungen 5.4 bis 5.7 dargestellt.

# Kapitel 6

## Evaluierung

Als letzter Schritt für die Beendigung der Entwicklung des Testprogrammes müssen Test durchgeführt werden und die Ergebnisse evaluiert werden. Für Begriffe und Verfahren habe ich mich auf das Buch [12] zurückgegriffen.

## 6.1 Testaufbau und -Ablauf

Um das Programm evaluieren zu können, müssen verschiedene Tests durchgeführt werden. Für die Tests wurde das Programm nach Aufgaben unterteilt:

1. Listener
2. DOORS Verbindung und Import der Parameter
3. Speichern der Parameter
4. Darstellung der Parameter
5. Löschen von Parametern
6. Lesen und Schreiben der TESTONA Datei

### Listeners

Die Erweiterung des *ResourceSetListeners* wurde getestet, indem alle ankommenden Nachrichten mit Hilfe des Debuggers überprüft worden sind. Da der *Listener* erweitert wurde, musste überprüft werden, dass die Erweiterung die anderen Funktionalitäten nicht negativ beeinflusst. Dafür wurde mit dem Debugger Schritt für Schritt überprüft, dass der neu implementierte Quelltext nur in den richtigen Fällen aufgerufen wird. Die Fälle sind das Löschen oder das Hinzufügen eines *ParameterTags* zu einem Bauelement.

### DOORS Verbindung und Import der Parameter

Da die Bibliothek für die DOORS Verbindung schon existierte und bereits getestet war, musste hier nur die Funktionalität der Verbindung überprüft werden. Es wurden Negativtests<sup>18</sup> durchgeführt. Die Negativtests bestanden aus falschen Log-in Parameter für die DOORS Datenbank. Dabei wurde der Benutzer aufmerksam gemacht, dass die Verbindung nicht entstehen konnte. Dieser Fall sollte eigentlich nicht auftreten, da die Log-in Daten in TESTONA gespeichert worden sind. Wären diese falsch, hätte der Benutzer es bereits bei dem Import der Anforderungen bemerkt.

Weiterhin wurden das Öffnen und Schließen der DOORS Module getestet und der Verbindungsaufbau und -abbau. Dabei wurde überprüft, dass die Verbindungen richtig getrennt worden sind. Nach das Öffnen der DOORS Module wurde auch überprüft, ob die Parameterwerte richtig in TESTONA gespeichert werden konnten. Hier wurde Positiv<sup>19</sup> - und Negativtest durchgeführt. Für die Positivtest wurden plausible Werten in die Parametertabelle eingetragen. In den Negativtest wurden Zellen leer gelassen oder mit Escape - Sequenzen (\n, \r oder \b) befüllt. Dabei musste das Programm erkennen, dass die Zelle kein Wert beinhaltet. Als richtige Werte wurden Buchstaben, Zahlen und Sondereichen

---

<sup>18</sup>Reaktion des Programms auf eine falsche Eingabe.

<sup>19</sup>Reaktion des Programms auf eine (nach den Anforderung) richtige Eingabe.

---

bewertet.

### Speichern der Parameter

Die Speicherung der Parameter in TESTONA musste an zwei Stellen getestet werden. Die erste Stelle ist nach dem Import der Parametertabelle und die Erstellung der *ParameterTags*. Hier wurde überprüft, dass die in die Parametertabelle eingegebenen Werte richtig in TESTONA übernommen worden sind. Weiterhin mussten die Werte in der richtigen Reihenfolge in das *ParameterTag* Objekt gespeichert und dargestellt werden. Die Escape - Sequenzen und leeren Zellen sollten ignoriert werden.

Die zweite Stelle, an der die Parameterspeicherung getestet werden musste, war bei Speichern eines Parameters in einem Bauelement. Als erstes musste getestet werden, dass das *ParameterTag* an das Bauelement hinzugefügt worden ist. Parallel dazu wurde das richtige Ausführen des *AddParameterTagCommand* Kommandos getestet. Weiterhin musste auch betrachtet werden, dass ein *ParameterTag* in einem Bauelement mehrere Parameter beinhalten kann. Hier wurde auch getestet, dass der *ParameterTag* richtig die verschiedenen Parameter mit den zugehörigen Varianten und Parameterwerten speichert.

### Darstellung der Parameterwerte

Bei der Darstellung der Parameterwerte musste beachtet werden, dass der richtige Parameterwert angezeigt wird. Je nach aktiver Variante musste das Programm den richtigen Wert aus dem *ParameterTag* lesen und darstellen. Wenn kein Wert zur Verfügung stand musste nur der Name des Bauelements angezeigt werden. Hier wurde das *ParameterTag* Objekt mit der Ausgabe im TESTONA Editor verglichen.

### Löschen von Parametern

Wie beim Speichern von Parametern gibt es hier auch zwei Fälle, die betrachten werden mussten. Der erste Fall ist, wenn einem *ParameterTag* aus ein Bauelement gelöscht werden muss. Hier wird auch parallel das Ausführen des *removeParameterTagCommand* Kommandos getestet. Dabei musste das *ParameterTag* aus das Bauelement gelöscht werden.

Der zweite Fall ist, wenn ein Parameter aus einem *ParameterTag* gelöscht werden muss (im *ParameterTag* befinden sich mindestens zwei verschiedene Parameter). Hier musste beachtet werden, dass nur der richtige Eintrag entfernt wurde. Falls nach das Entfernen des Parameters, das *ParameterTag* nur noch einen Parameter beinhaltete, musste die Struktur des *ParameterTags* umgestellt werden.

---

**Lesen und Schreiben der TESTONA Datei**

Der letzte wichtige Aspekt der noch getestet werden musste, ist das Schreiben und Lesen der TESTONA Datei. Wenn der Benutzer Parameter importiert und an Bauelement verknüpft hat, müssen diese in der TESTONA Datei gespeichert werden. Dafür wurde der Inhalt vom *ParameterTag* Objekt mit dem XML-Eintrag der TESTONA Datei verglichen.

Wenn das Speichern erfolgreich war, musste das Lesen getestet werden. Dafür wurde den Inhalt des erzeugten *ParameterTags* dem Eintrag in der XML Datei gegenüber gestellt.

## 6.2 Ausfallrisiken

Nachdem alle in Kapitel 6.1 Tests erfolgreich durchgeführt worden sind, entstand immer noch ein Ausfallrisiko. Der neue Quellcode befand sich in der *pre-Alpha* Version. Das Bestehen der durchgeführten Tests ergab die erste Alpha Version<sup>20</sup> des Programmes.

Die programmierten Erweiterungen von TESTONA basierten nicht auf dem sogenannte „vier Augen Prinzip“. Das Programm konnte noch Fehler beinhalten. Da nur der Entwickler bis jetzt die neue Erweiterung getestet hatte, konnte die Objektivität der Tests nicht gewährleistet werden. An dieser Stelle mussten genauere White-Box Tests erstellt und durchgeführt werden. Als nächster Schritt mussten die Black-Box Tests erfolgen.

Ein weiterer Punkt für das Ausfallrisiko war, dass die durchgeführten Tests auf einfachen Beispielen basierten. Das heißt, die Klassifikationsbäume und Parametertabellen waren recht simpel und klein. Es war nötig, durchaus komplexere Beispiele zu betrachten, indem der Klassifikationsbaum aus mehreren Klassen und Klassifikationen besteht. Ebenso muss die Parametertabelle viele Parameter definieren. Aus der Grundlagen der Kombinatorik (siehe 3.1.1) war klar geworden, dass Fehlverhalten nicht unbedingt aus einzelnen falschen Parametern entstanden, sondern oft aus der Interaktion verschiedener Parameter.

---

<sup>20</sup>Nicht vom Entwickler durchgeführte Tests

---



## 6.3 Bekannte Fehler

Nach der Durchführung des in Kapitel 6.1 beschriebenen Testaufbaus und -ablaufs wurden verschiedene Fehler behoben. Bisher war nur noch ein Fehler bekannt, der nicht behoben werden konnte. Nachdem sich das Programm erfolgreich mit DOORS verbinden konnte, konnte später die Verbindung nicht richtig getrennt werden. Das Programm versuchte die Verbindung zwei Mal zu trennen. Im ersten Versuch wurde die Verbindung erfolgreich getrennt und geschlossen. Aus bis jetzt unbekannten Gründen versuchte das Programm ein zweites Mal die Verbindung zu DOORS zu trennen. Hier entstand der Fehler, dass das *ConnectionInterface* für die Verbindung nicht mehr zu Verfügung stand.

Nach mühsames Suchen und Debuggen mithilfe eines Mitarbeiters, konnten wir die Ursache des Fehlverhaltens nicht finden. Dieser Fehler wurde mit einer niedrigen Priorität versehen, da in der näheren Zukunft die DOORS API durch eine neuere API ersetzt wird. Dabei wird der Quellcode für die Verbindung, Import und Trennung der DOORS Datenbank überarbeitet. Es ist sehr wahrscheinlich, dass dabei dieser Fehler behoben wird. Weiterhin brachte dieser Fehler TESTONA nicht zum Absturz.

## 6.4 Optimierungskriterien

Die geleistete Arbeit könnte noch in zwei Punkten optimiert werden. Doppelte Testfälle könnten automatisch erkannt werden. Der Benutzer könnte dabei gewarnt werden oder die Testfälle könnten sofort gelöscht werden. Diese doppelten Testfälle entstehen durch gleiche Parameterwerte innerhalb eines Knotens bei einer aktiven Variante. So konnte der Fall auftreten, dass die Landgeschwindigkeit eines bestimmten Autos zugleich die Maximalgeschwindigkeit ist. Bei dieser Konstellation würden sich einige Test wiederholen, weil der Parameterwert für Land- und Maximalgeschwindigkeit gleich ist.

Weiterhin besteht die Möglichkeit, einen Algorithmus zu entwickeln, in dem ähnliche Testfälle erkannt werden. Diese konnten nicht betrachtet werden, um die Testmenge zu verkleinern. Immerhin sollte die maximale Testabdeckung garantiert sein. Sind die Geschwindigkeiten verschiedener Varianten ähnlich, so könnten manche Testfälle übersprungen werden. Das würde zu weniger Testfälle führen, aber die Qualität des Produkts konnte immer noch gewährleistet werden.

## **Kapitel 7**

### **Zusammenfassung und Ausblick**

## 7.1 Zusammenfassung

Mit dieser Masterarbeit sollte die Erweiterung und Verbesserung des Berner & Mattner Werkzeuges TESTONA erreicht werden. Dabei soll die Testfallgenerierung optimiert werden. Nachdem Anforderungen und Varianten aus DOORS Module importiert wurden, können in DOORS definierte Parameterwerte ebenso importiert werden. Die Parameterwerte können dynamisch in Bauelementen dargestellt werden, immer in Abhängigkeit von der aktiven Produktvariante. Dabei wird der Tester unterstützt, indem eine bessere Übersicht erreicht wird. So kann eine höchstmögliche Testabdeckung einfacher erzielt werden.

Durch das Speichern und Darstellen von importierten Parameterwerten aus DOORS wurde die Testfallgenerierung in TESTONA verbessert. Mit dem automatischen Import der Parameterwerte wurde dem Benutzer viel Arbeit abgenommen. Dabei erhält der Benutzer eine bessere Übersicht von den erzeugten Testfällen und kann diese den reellen Werten gegenüberstellen. Dies führt zur Zeitersparnis und niedrige Testkosten.

Anhand der vorhandenen Parameterwerte können auch gezielt spezielle Testfälle erzeugt werden. Ist zum Beispiel ein Teil eines Produktes sehr komplex, kann der Benutzer anhand der vorhandenen Parameterwerte speziell für dieses Produkt die notwendigen Testfälle generieren.

Weiterhin besteht die Möglichkeit, doppelte Testfälle leichter zu erkennen und zu vermeiden. Diese entstehen durch gleiche Parameterwerte in verschiedenen Bauelementen mit dem gleichen Oberknoten in einer aktiven Variante.

---

## 7.2 Persönliches Fazit

Durch Verwendung der Programmiersprache Java konnte ich meine Kenntnisse und Erfahrungen in dieser Sprache bereichern. Anhand der Arbeit in einem umfangreichen Programm wie TESTONA habe ich gelernt, Teil eines größeren Entwicklungsteams zu sein. Der Ablauf von Teamabsprachen, wöchentlichen Statusmeetings sowie Vereinbarungen sind mir vertrauter geworden.

Da TESTONA auf RCP basiert, konnte ich viele neue Aspekte von Java und von Programmarchitekturen lernen. Diese Masterarbeit befasst Themen wie Testen, Qualitätssicherung und Softwareentwicklung, die meiner Interessen sehr nahe liegen. Die erworbenen Kenntnissen werde ich in meiner zukünftigen Berufsleben erfolgreich anwenden können.

Da während der Arbeit meine Betreuer aus der Firma Berner & Mattner sich geändert haben, wurden sich auch die Anforderungen etwas geändert. Die Änderung der Betreuer hat auch dazu beigetragen, dass organisatorisch nicht alles optimal lief. Ich schätze das Ergebnis als durchaus positiv und bereichernd für TESTONA. Ich glaube es wäre auch sehr interessant gewesen, die Abhängigkeitsregeln für die Testfallgenerierung genauer zu betrachten.

# 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 . . . . .	5
3.1	Baum und Testfälle ohne Kombinatorik . . . . .	9
3.2	Abhängigkeitsregeln Bearbeitung . . . . .	11
3.3	Abhängigkeitsregeln Übersicht . . . . .	12
3.4	Testfälle mit Anwendung der Abhängigkeitsregeln aus 3.2 und 3.3 . . . .	12
3.5	Beispiel einer Tabelle in DOORS . . . . .	14
3.6	Betrachtung von Varianten eines Wagens von der generischen Testspezifikation zur variantenspezifische Testspezifikation (aus [16]). . . . .	16
4.1	Darstellung der TESTONA Objekte für die Parameterspeicherung . . . . .	25
4.2	Darstellung eines Baumelementes mit zwei verschiedenen Parametern und vier Varianten . . . . .	26
4.3	Aktive Variante Generic (default) . . . . .	27
4.4	Aktive Variante Cabrio . . . . .	27
4.5	Aktive Variante Kombi . . . . .	27
5.1	Einfache Übersicht der Klassen . . . . .	29
5.2	Darstellung der Klasse ParameterTagImpl und das Interface ParameterTag .	34
5.3	Klasse ParameterManager . . . . .	37
5.4	Der generische Baum . . . . .	44
5.5	Aktive Variante Cabrio . . . . .	44
5.6	Aktive Variante Limo . . . . .	44
5.7	Aktive Variante Kombi . . . . .	44
A.1	Klassendiagramm von ParameterManager.java . . . . .	60

---

# Listings

3.1	Beispiel einer SWT Anwendung . . . . .	20
5.1	XML Darstellung eines ParameterTags . . . . .	36
5.2	Auszug von der Erstellung der ParameterTags . . . . .	38
5.3	Verbindungsaufbau . . . . .	40
5.4	Verbindungsaufbau war erfolgreich . . . . .	41
5.5	Laden der Parametertabelle nach Zeilen . . . . .	42
5.6	Nachladen der Parametertabelle nach Zeilen . . . . .	43

# Tabellenverzeichnis

3.1	Testfälle mittels paarweise Kombinatorik . . . . .	9
4.1	Beispiel für die Zuordnung zwischen Varianten und Parameterwert aus einer Anforderung . . . . .	25
5.1	Beispiel eines ParameterTags . . . . .	35
5.2	Beispiel eines ParameterTags mit zwei Parametern . . . . .	36
5.3	Parameter <i>max_speed</i> . . . . .	44



# Anhang A

## Anhang

### A.1 CD

Inhalt:

- Quellen
- PDF-Datei dieser Arbeit

## A.2 ParameterMananger Klassendiagramm

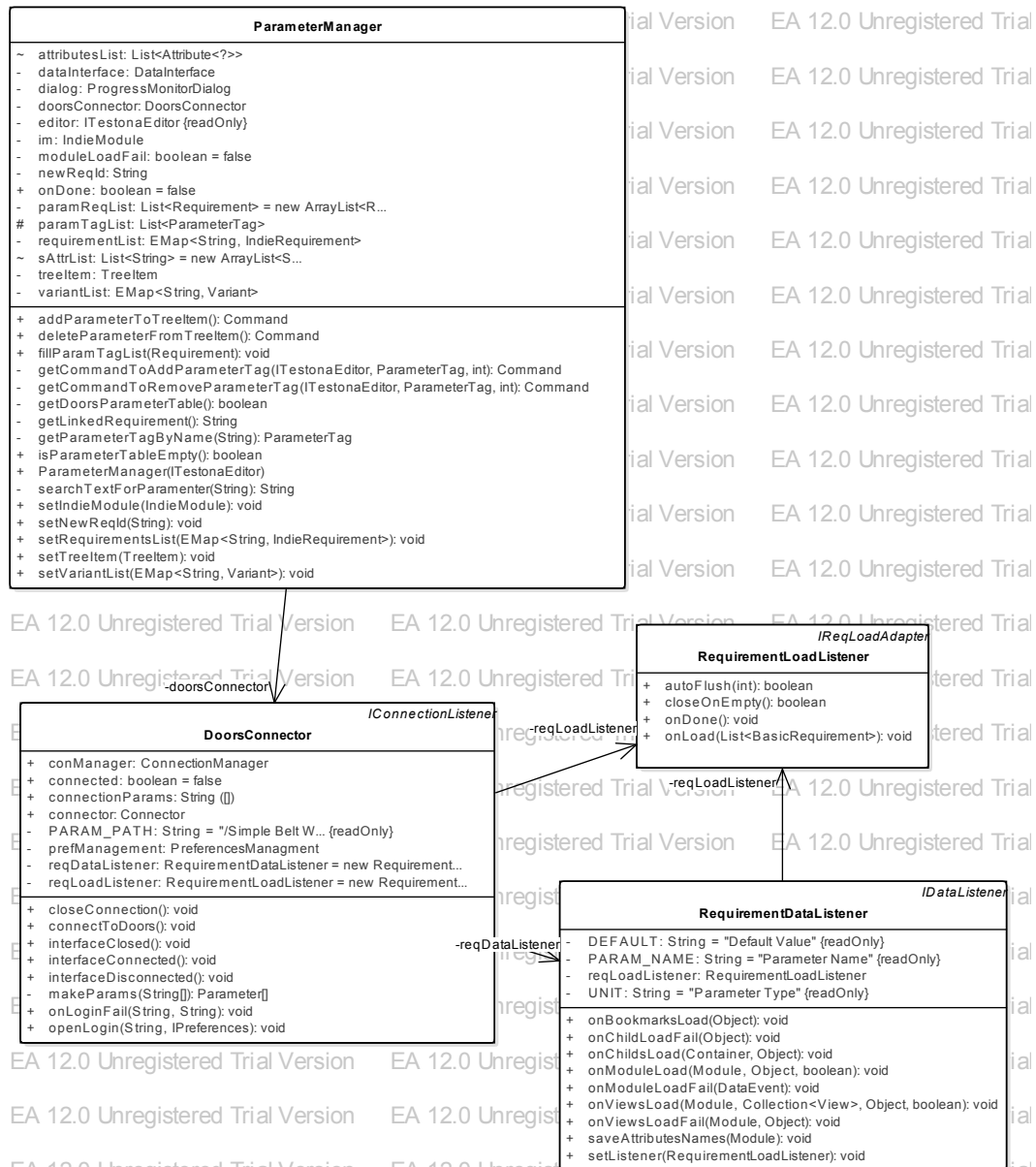


Abbildung A.1: Klassendiagramm von ParameterManager.java

# Literaturverzeichnis

- [1] Gürkan Aydin and Hartmut Pohl. Code coverage - tools. *Hakin9*, 2012.
  - [2] Berner & Mattner, <http://www.testona.net>. *TESTONA*, Oktober 2014.
  - [3] Eclipse Foundation, <http://eclipse.org>. *Eclipse*, Oktober 2014.
  - [4] Eclipse Foundation, <http://help.eclipse.org>. *Eclipse Help*, Oktober 2014.
  - [5] Eclipse Foundation, <http://eclipse.org/swt>. *Eclipse SWT*, Oktober 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] Rick Kuhn, Raghu Kacker, Yu Lei, and Justin Hunter. Combinatorial software testing. 2009.
  - [10] National Institute of Standards and Technology, <http://mse.isri.cmu.edu/software-engineering/documents/faculty-publications/miranda/kuhnintroductioncombinatorialtesting.pdf>. *Introduction to Combinatorial Testing*, 2011.
  - [11] T. J. Ostrand and M. J. Balcer. The category-partition method for specifying and generating functional tests. *Communications of the ACM*, 1988.
  - [12] Lionel Pilorget. *Testen von Informationssystemen*. Vieweg+Teubner Verlag, 2012.
  - [13] Quality Week, [http://www.systematic-testing.com/documents/qualityweek1995\\_1.pdf](http://www.systematic-testing.com/documents/qualityweek1995_1.pdf). *Test Case Design Using Classification Trees and the Classification-Tree Editor CTE*, 1995.
  - [14] Iris Reinhartz-Berger, Arnon Sturm, Tony Clark, Sholom Cohen, and John Bettin. *Domain Engineering: Product Lines, Languages, and Conceptual Models*. Springer, 2013.
  - [15] Christopher Robinson-Mallet. An approach on integrating models and textual specifications. *Model-Driven Requirements Engineering Workshop (MoDRE)*, 2012.
-

- [16] 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.
  - [17] H. Tremp and M. Ruggiero. *Application Engineering: Grundlagen für die objektorientierte Softwareentwicklung mit zahlreichen Beispielen, Aufgaben und Lösungen*. Compendio Bildungsmedien, 2011.
  - [18] Gerhard Versteegen. *Anforderungsmanagement: Formale Prozesse, Praxiserfahrungen, Einführungsstrategien und Toolauswahl*. Springer, 2004.
  - [19] Lars Vogel and Mike Milinkovich. *Eclipse 4 RCP*. Lars Vogel, 2013.
-