Beuth Hochschule für Technik Berlin

Masterarbeit

Variantenspezifische Abhängigkeitsregeln und Testfallgenerierung in TESTONA

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

Fachbereich VI - Technische Informatik - Embedded Systems

Eingereicht am: 19. Februar 2015

Erstprüfer	:	Prof. Dr. Macos
Zweitprüfer	:	Prof. Dr. 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

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[1][1]. Im Kapitel 3.1 wird weiteres zu diesem Programm und die Funktionsweise erläutert.

Die Erweiterung des Programmes besteht aus verschiedenen Themen. Ein Thema davon behandelt die Testfallgenerierung und die jeweilige Testabdeckung[2]. Hier soll garantiert werden, dass bei einer automatischen Testfallgenerierung, eine höchstmögliche Testabdeckung erzielt wird.

Die Testfallgenerierung wird in dieser Arbeit beeinflusst, indem die Produktvarianten stärker betrachten werden. Verschiedene Varianten beinhalten verschiedene Parameter und Produktkomponenten (Eigenschaften). Die Parameterwerte definieren auch 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 zur jeweiligen Produktvariante zugeordnet werden. Aus diesem Grund kann es zu Konflikten bei der Testfallgenerierung kommen, bzw. inkohärente Testfälle können auftreten.

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

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

---

[1] Kombination mehrerer Testfälle mit dem Ziel, komplexeren Sachverhalten zu überprüfen.

[2] (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.[13]

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. Jedes Produkt kann unterschiedliche Produktvarianten beinhalten und jede Variante besteht aus unterschiedlichen Komponenten mit unterschiedlichen Parametern. In Abhängigkeit von der ausgewählten Variante sollen bei der Testfallgenerierung die dazugehörigen Komponenten berücksichtigt werden und die erzeugten Testfälle dargestellt werden. Besonders zu beachten sind dabei die definierten Abhängigkeitsregeln sowie die darauf bezogene Testabdeckung.

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

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

Einer der besonderen Eigenschaften von TESTONA ist die Kopplung mit Anforderungsspezifikationen die in IBM Rational DOORS definiert worden sind. Durch das DOORS Add-On MERAN können Anforderungen die in DOORS definiert sind, mit den zugehörigen Varianten verknüpft werden. Diese Varianten können durch eine erfolgreiche Anmeldung bei DOORS (über die TESTONA Oberfläche) und ein gezieltes Auswählen der

gewünschten Varianten in TESTONA eingebunden werden. Hierbei sollen die in den Anforderungen definierten Parametern (z.B. eine Geschwindigkeit oder die Anzahl der Türen eines Autos) mit gespeichert werden. Im Klassifikationsbaum soll je nach ausgewählter Variante mit dem entsprechenden Wert ersetzt werden (z.B. der Name vom Baumelement). Andere Lösungsmöglichkeiten werden noch untersucht.

Der derzeitige Varianten-Management-Ansatz in TESTONA ist nicht in der Lage für die Testfallgenerierung zwischen verschiedene Varianten zu unterscheiden. Zwar werden durch die Perspektive „Variant Management“ verschiedene Varianten unterschieden, aber die Testfälle müssen manuell mit den jeweiligen Varianten verknüpft werden. Im Falle einer automatischen Testfallgenerierung werden auch ungültige Baumelemente 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.

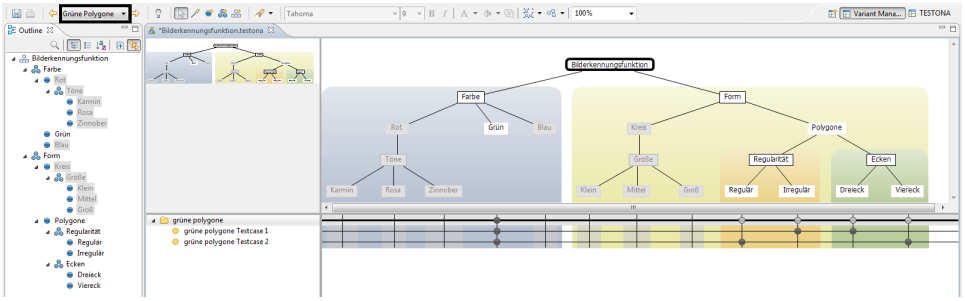

Abbildung 2.1: Richtige Auswahl der Klassen und Klassifikationen für die Testfallgenerierung

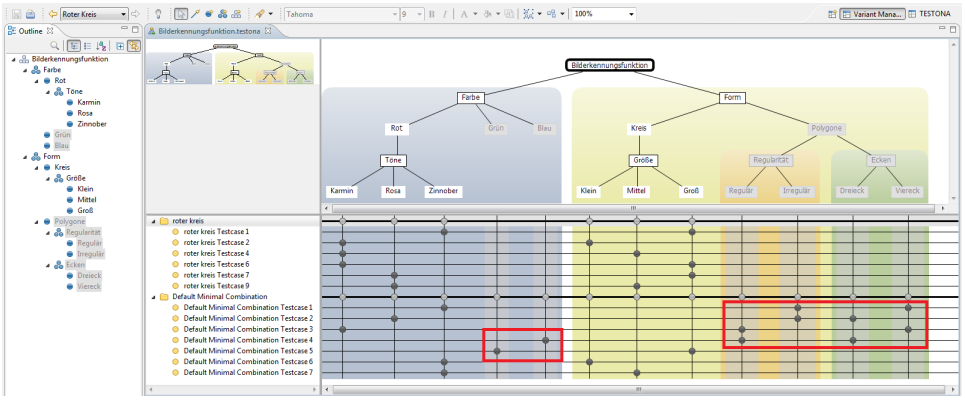

Abbildung 2.2: Ungültige Auswahl der Klassen (Grün und Blau) und Klassifikationen (Regularität und Ecken) für die Testfallgenerierung

2.2 Variantenmanagement Software

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

Im englischen Sprechraum werden die Begriffe „feature modelling“ und „domain engineering“ benutzt. Unter „domain engineering“ ist die Wiederverwendung von Wissensgebiet für die Produktion einer neuen Software zu verstehen [10]. Obwohl beide Begriffe sich auf die Softwareentwicklung spezialisieren, werden die Prinzipien allgemein in der Produktion verschiedener Produkte verwendet. Betrachten man TESTONA, so sind die Editionen Light, Express, Professional und Enterprise Varianten.

Es existiert Software die Variantenmanagement unterstützt. Aber es gibt keine Software die eine Schnittstelle zu einer Datenbank zur Verfügung stellt. Weiterhin wird das Import von Varianten aus der Datenbank nicht befördert.

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[1].

[1] 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 definierte Parameterwerte sollen in TESTONA importiert und angezeigt werden. Die Zuordnung von Parametern an Baumelemente erfolgt über die Verknüpfung von (aus DOORS importiert) Anforderungen mit ein Baumelement. In jeder Varianten kann der Parameter ein anderer 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 werden betrachtet. Anhand der Parameterwerte kann es dazu kommen, dass doppelte Testfälle erstellt werden. Wenn die Endknoten unter einen Knoten die gleichen Werten besitzen, können vorher gültige Testfälle dadurch verdoppelt werden. Dabei soll der Benutzer 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 automatische generierte Testspezifikationen und Testfälle bequem mit Anforderungen durch Add-Ons an Standard-Werkzeugen zu verlinken (siehe 3.2). Somit werden robuste Schnittstellen für ein komfortables Anforderungs- und Testmanagement erzeugt. Ein sehr wichtiger Aspekt von TESTONA ist, dass es Branchen-unabhängig ist. Das heißt ein Tester kann TESTONA im jedem Fachgebiet benutzen, nicht nur bei Software- oder Funktionstests.

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

3.1.1 Klassifikationsbaum-Methode

1993 entwickelten K. Grimm und M. Grochtmann die Klassifikationsbaum-Methode zur Ermittlung funktionaler Blackbox-Tests im Bereich von eingebetteter Software. Die Methode wurde im Forschungslabor von Daimler-Benz in Berlin als Weiterentwicklung der Category-Partition Method (CPM) erforscht. Gegenüber CPM hat die Klassifikationsbaum-Methode eine graphische Baum-Darstellung und hierarchische Verfeinerungen für implizite Abhängigkeiten. Als Werkzeug wurde der „Classification Tree Editor“ (CTE) [1] programmiert und unterstützt Partitionierung und Testfallgenerierung. Das Werkzeug von CPM konnte nur Testfälle generieren ohne Bestimmung der Testaspekte[6].

Diese Methode besteht aus zwei wichtigen Schritten:

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

[1] Entwickelt von Grochtmann und Wegener[9]. Bei Berner & Mattner aus rechtlichen Gründen zu TESTONA umbenannt

- 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 [17]. 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).

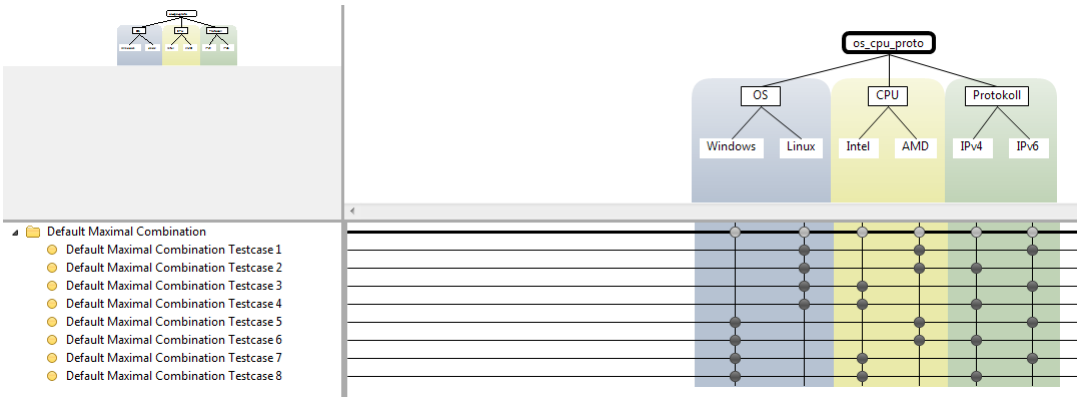

Abbildung 3.1: Baum und Testfälle ohne Kombinatorik

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[8].

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

Tabelle 3.1: Testfälle mittels paarweise Kombinatorik

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

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[2]. 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)[8].

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[3] 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.*[5]

Anhand der generierten Testfälle und die Benutzung von kombinatorischen Möglichkeiten kann der Tester einfach eine präzise Testtiefe erreichen. Die Testtiefe wird anhand der Durchführung einer Risikoanalyse und der Auswertung der Kritikalitätsstufe (sehr hoch, hoch, mittel und tief) des Systems vereinbart. Das soll heißen, dass die Testtiefe für ein

[2] Basierend auf funktionelle und technische Anforderungen, Use-Cases

[3] International Software Testing Qualification Board

Flugzeug (Kritikalitätsstufe = sehr hoch[4]) viel höher und genauer ist, als die Kompatibilität eines Bildschirmes mit ein Graphiktreiber (Kritikalitätsstufe = tief[5]).

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.[14]

3.1.3 Abhängigskeitsregeln

Abhängigkeitsregeln werden vereinbart um überflüssige Testfälle zu vermeiden, bzw. um Vorbedingungen für bestimmte Testszenarien festzulegen. Abhängigkeitsregeln werden Mithilfe von boolische Algebra definiert wie folgende Abbildung 3.3zeigt:

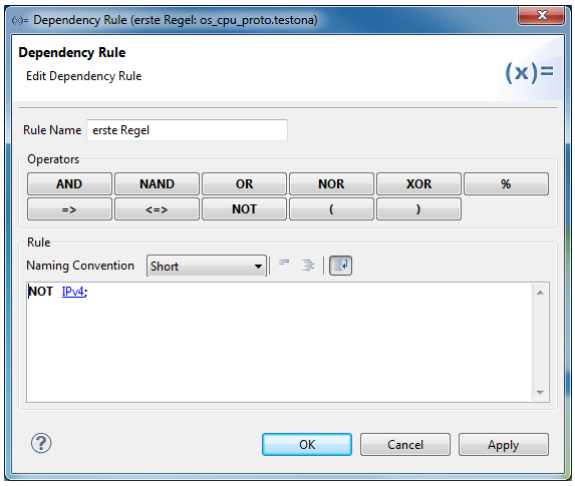

Abbildung 3.2: Abhängigkeitsregeln Bearbeitung

Boolesche Operatoren:

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

Durch die Verwendung dieser Operatoren wurde folgende Abhängigkeitsregel definiert:

[4]Das Fehlverhalten kann zu Verlust von Menschenleben führen, die Existen des Unternehmens gefährden

[5]Das Fehlverhalten kann zu geringen materiellen oder immateriellen Schäden führen

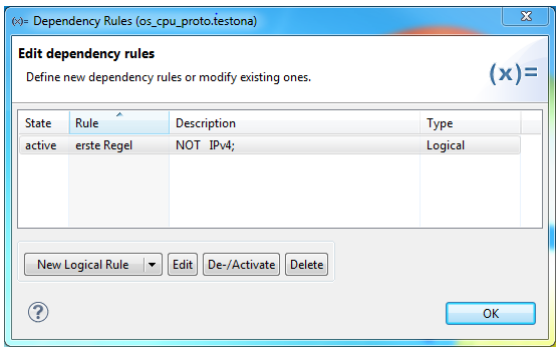

Abbildung 3.3: Abhängigkeitsregeln Übersicht

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.

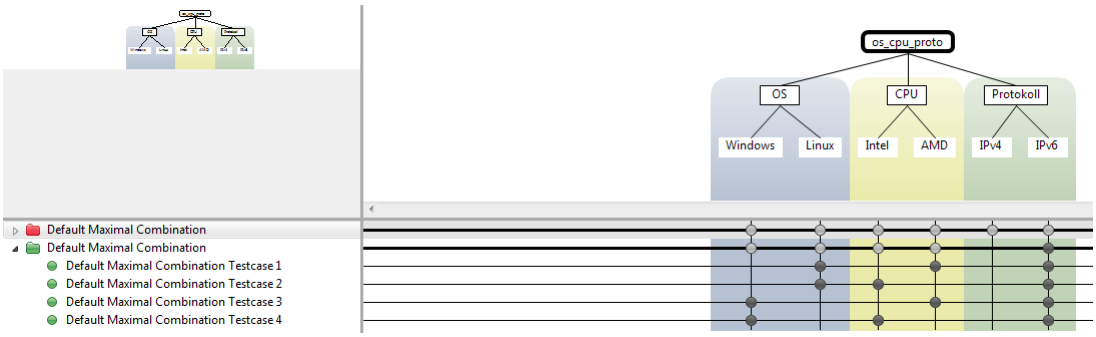

Abbildung 3.4: Testfälle mit Anwendung der Abhängigkeitsregeln aus 3.2 und 3.3

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 Lichtschaltern 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 Eingenschaften abgelesen werden. Als Eingenschaften sind eine eindeutige Identifikationsnummer, sowie vom Benutzer ausgewählte Attribute.

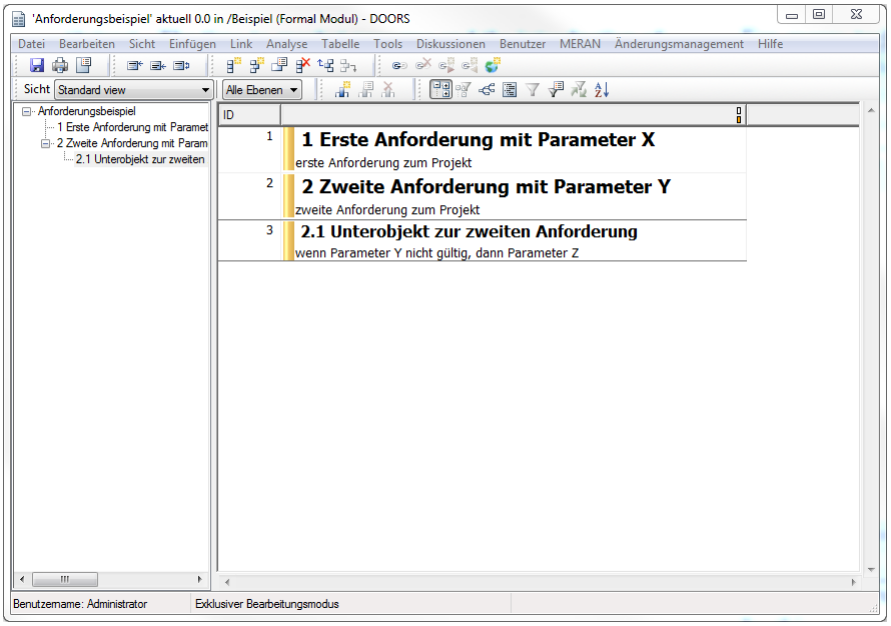

Abbildung 3.5: Beispiel einer Tabelle in DOORS

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. [7] [15]

TESTONA besitzt Klassen die DXL-Scripte 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 Beispieles 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. 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.[11]

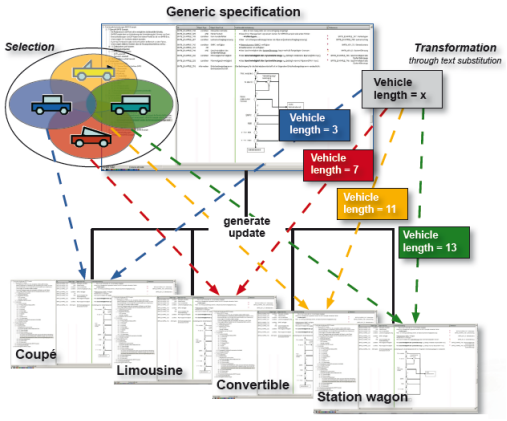

Abbildung 3.6: Betrachtung von Varianten eines Wagens von der generischen Testspezifikation zur variantenspezifische Testspezifikation aus [12]

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 der jeweiligen Variante angepasst. Die Abbbildung 3.6 betrachtet die Länge jedes Modell 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. Baumelement werden an der jeweiligen Variante hinzugefügt. Wenn der Benutzer eine Variante auswählt, werden nur die in dieser Variante gültige Baumelemente angezeigt. So wird die Übersicht verbessert und der Tester kann Testfälle genauer erstellen.

3.4 Entwicklungsumgebung und Programmiersprache

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

3.4.1 Eclipse

Eclipse ist der Nachfolger von „IMB Visual Age for Java“ und ist ein quelloffenes Programmierwerkzeug zur Entwicklung verschiedener Arten von Software. Ursprünglich wurde Eclipse als integrierte Entwicklungsumgebung für Java benutzt. Dank seiner Bedienbarkeit und Erweiterung ist Eclipse mittlerweile für die Entwicklung in verschiedenen Programmiersprachen bekannt (unter anderem C/C++ und PHP). Die am 25 Juni 2014 veröffentliche 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[16] [2]. 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 verschiedene 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
- Stardard Widget Toolkit: Programmierbibliothek zur Erstellung von grafischen Oberflächen
- JFace: User Interface Toolkit für komplexere Widgets

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

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

Testona besteht momentan aus 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[6], 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

[6]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 Programmiermodel in alle Plattformen. [4]

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

- Ein *Display* deklarieren (repräsentiert die SWT Modus)
- Erstellen eines *Shell*, welche als Hauptfenster dient
- Erzeugung eines Widgets
- Initialisierung der Widgetsparameter
- Öffnen des Fensters
- 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 das JFace benutzt.[3]

3.4.5 Tags

Ein *Tag* ist ein Objekt, dass auf EMF (Eclipse Modeling Framework) basiert. EMF ist ein Java Framework der Eclipse-Open-Source Gemeinschaft für die automatische Erzeugung von Quelltext aus Modellen [16]. Bei EMF wird die Problemstellung in einem Klassendiagramm beschrieben. Hier werden die Eigenschaften und Attribute der Klassen eingetragen um daraus wird 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 . Das erzeugte Quelltext kann dann vom Benutzer ergänzt werden.

Im Fall von *Tags*, diese dienen hauptsächlich um Eigenschaften andere Objekte zu beschreiben. Zum Beispiel gibt es unter anderem ein *Tag* indem Varianten gespeichert werden (*VariantsTag*). Wenn in einem Projekt verschiedene Varianten zur Verfügung stehen, wird es dazu kommen, dass ein Baumelement nicht in alle Varianten auftritt. Das Baumelement wird ein *VariantsTag* besitzen, dass spezifiziert zur welchen Varianten das Baumelement gehört. Ein weiterer Beispiel sind die *RequirementTags*. Hier werden die Anforderungen gespeichert, die an einem Baumelement 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

Weiterhin können die Klassen um weitere Attribute erweitert werden, je nachdem welchem Zweck gedient wird. 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 (dem entsprechend auch ausgelesen werden). Um Parameter in Baumelementen zur speichern, wird auf dieses Modell aufgebaut. Weiter dazu wird in Kapitel 4.1 und 5.1.3 erörtert.

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

Da Parameterwerte und Varianten schon verlinkt sind, müssen die Anforderungen, die Parameter beinhalten, mit der Parametertabelle manuell verlinkt werden. Wenn dieser Vorgang abgeschlossen ist, kann über die TESTONA Oberfläche die Verbindung zu DOORS aufgebaut werden um die nötigen Informationen zu importieren. Voraussetzung ist, dass der Tester bereits einen Klassifikationsbaum passend zum zu testenden Produkt und die dazu gehörige Anforderungen erstellt hat. Jetzt können die in DOORS definierten Varianten importiert werden. Zunächst muss der Tester die Baumelemente der richtigen Variante zuordnen (durch ein Ausschlussverfahren wird angenommen, dass alle Baumelemente 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 Baumelemente per Drag&Drop verknüpft werden können. Mit dieser Funktion muss der Tester die Anforderung mit einem dazu gehörigen Baumelement 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 Baumelement verknüpft wurde, muss eine zu programmierende Methode den Anforderungstext von diesem Baumelement lesen und nach Parametern suchen. Als erstes wird davon ausgegangen, dass ein Baumelement 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 Baumelement verlinken und gleichzeitig werden die Parameter gelesen und gespeichert. Der Nachteil ist, dass möglicherweise unnötige Daten in TESTONA gespeichert werden.

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

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

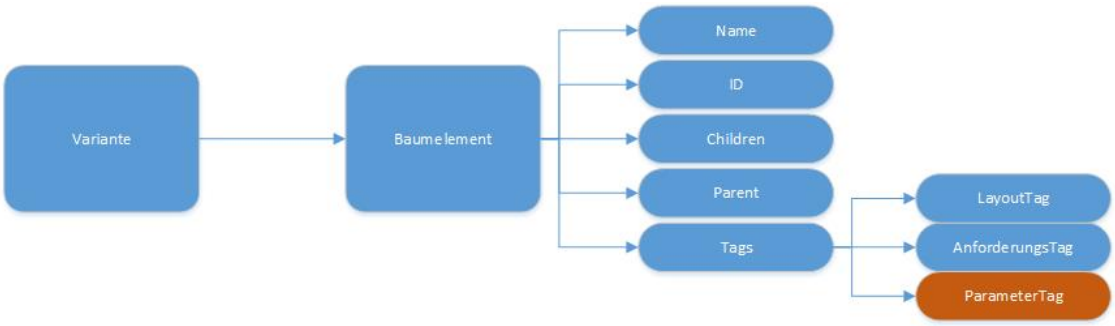

Abbildung 4.1: Darstellung der TESTONA Objekte für die Parameterspeicherung

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 Anforderung (innerhalb eines Baumelementes) nicht gelöscht oder überschrieben werden.

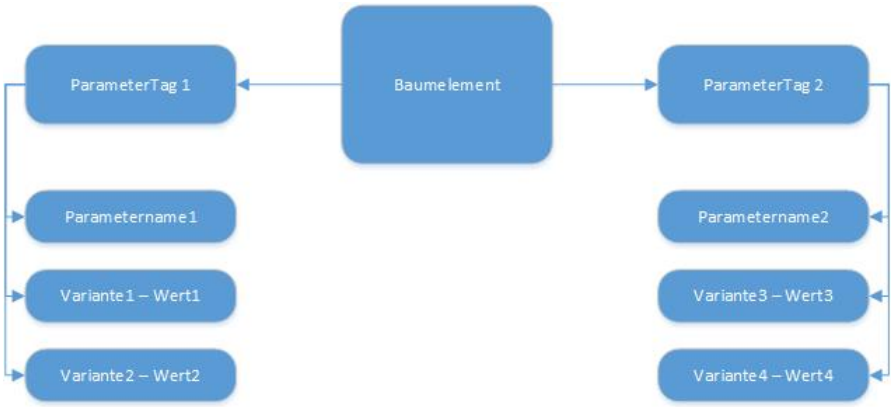

Abbildung 4.2: Darstellung eines Baumelementes mit zwei verschiedene Parametern und vier Varianten

4.2 Visualisierung

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

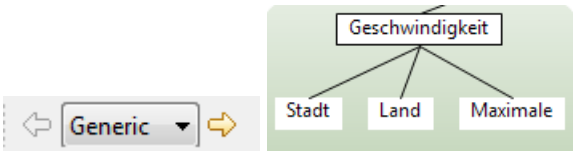

Abbildung 4.3: Aktive Variante Generic (default)

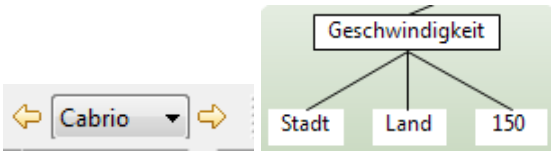

Abbildung 4.4: Aktive Variante Cabrio

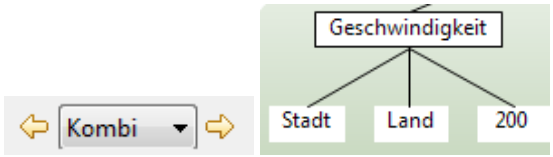

Abbildung 4.5: Aktive Variante Kombi

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

4.3 Testgültigkeit

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

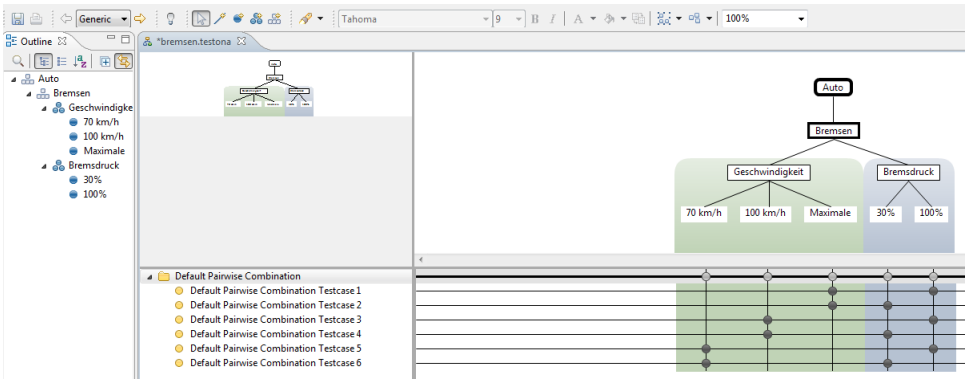

Abbildung 4.6: Das generische Baum

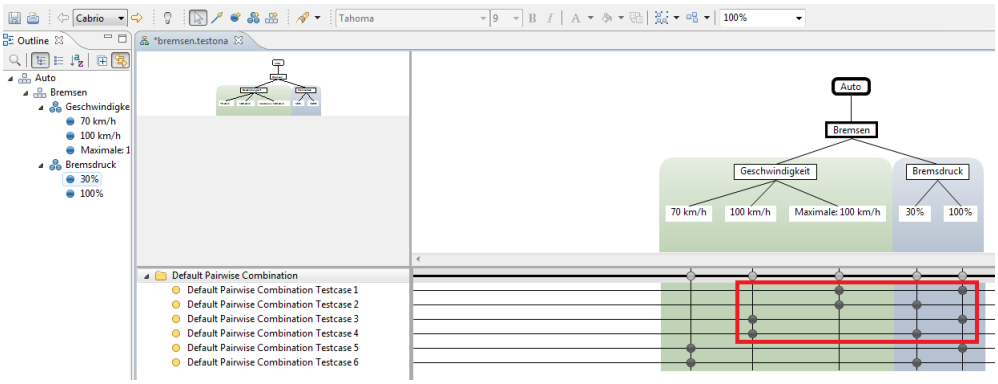

Abbildung 4.7: Aktive Variante Cabrio

Es ist leicht zu erkennen, dass die Testfälle eins bis vier in der Variante „Cabrio“ dupliziert sind. Bei einem minimalistischen Baum sind solche Fälle einfach zu erkennen. Bei einem komplexeren Baum mit sehr vielen Klassen kann es schnell zur Unübersichtlichkeit kommen. Duplizierte Testfälle könnten übersehen oder nicht erkannt werden.

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

Dafür müssen alle Klassen (Endknoten) unterhalb der gleichen Klassifikation überprüft werden. Nach dem obenstehenden Beispiel, müssen die Klassen „70 km/h“, „100 km/h“ und „Maximale“ verglichen werden. Da „Maximale“ bei jeder Variante einen anderen Wert

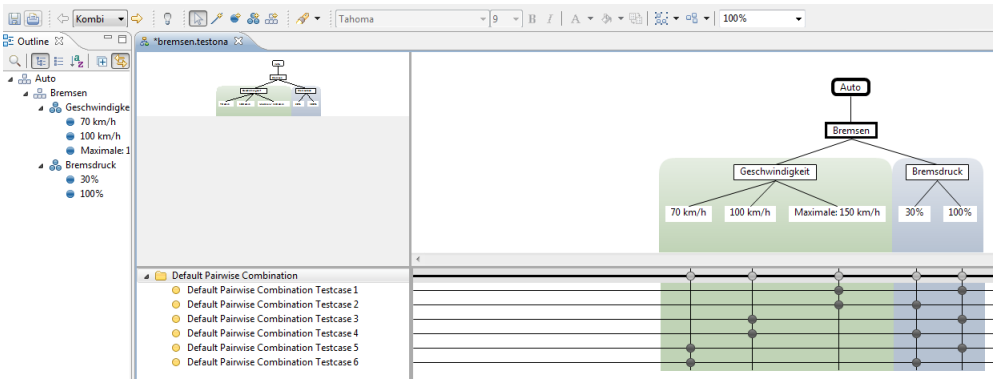

Abbildung 4.8: Aktive Variante Kombi

annimmt, muss hier überprüft werden, dass bei jeder Variante, sich der Wert nicht wiederholt. Die Werte werden aus dem geplanten *ParameterTag* (siehe Abbildung 4.1) gelesen.

4.4 Benutzeroberfläche

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

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

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

Kapitel 5

Systementwurf

5.1 Variantenmanagement und Parameter

Die Lösung zum Speichern und Darstellen der Parameterwerte abhängig der aktiven Variante wird im diesem Kapitel erläutert.

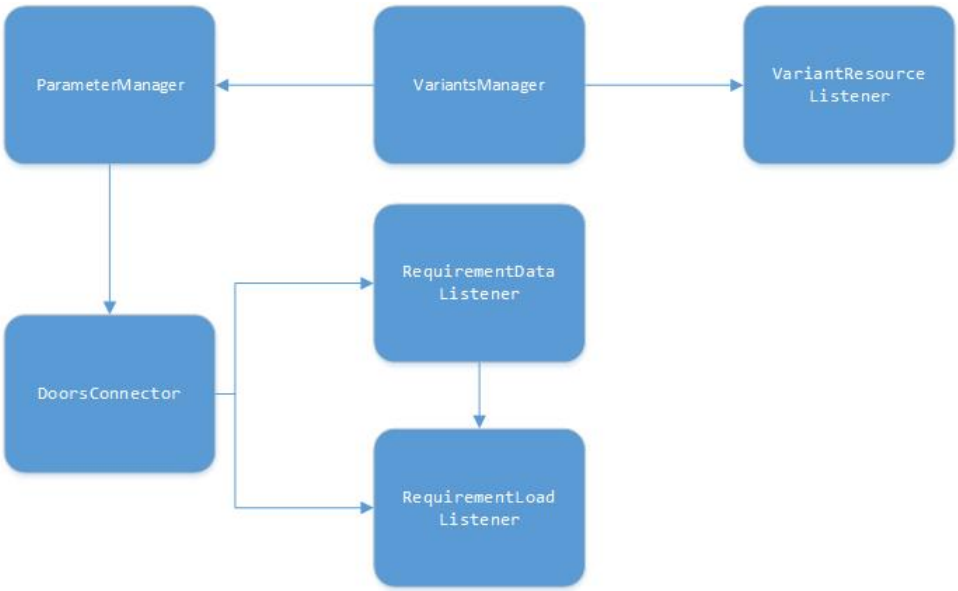

Abbildung 5.1: Einfache Übersicht der Klassen

Um eine genauere Darstellung aller Klassen, bitte siehe Anhang A.2. Der Ablauf für das Speichern eines Parameters in einem Baumelement sieht folgendermaßen aus:

1. Benutzer verknüpft eine Anforderung an einem Baumelement
2. Der *Listener* meldet das eine Änderung vorgenommen wird und gibt die Informationen weiter
3. Der *VariantsManager* lädt und übergibt die nötige Informationen an das *ParameterManager*

4. Der *ParameterManager* lädt, sucht und erstellt ein *ParameterTag* und gibt das Befehl, dass ein *ParameterTag* an ein Baumelement hinzugefügt werden muss.
5. Der *ParameterManager* gibt das Befehl an das *Listener* weiter
6. Das 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 Varianten zu Zuordnen. Hier werden Baumelemente zu Varianten hinzugefügt und gelöscht. Dabei muss das Klassifikationsbaum neu gezeichnet werden. Diese Klasse kümmert sich auch, um die Umschaltung zwischen Varianten in der Variantenansicht (siehe Abbildung 4.3) und die Richtige Darstellung der Baumelemente mit aktuelle Informationen.

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

Die Aufgabe des *Listeners* lautet, dem *VariantsManager* zu melden, wenn Änderung[1] geschehen werden oder geschehen sind. Im *Listener* werden die benötigte Informationen gefiltert und an dieser Klasse weitergegeben, damit Änderung vorgenommen werden können. Im Fall der Parameterersetzung sind es die Identifikationsnummer eines Baumelementes und die Anforderungskennung der verlinkte Anforderung. Diese beide Kennungen diesen dazu, dass der *VariantsManager* die jeweiligen Objekte (ein Baumelement und eine Anforderung) laden kann. Diese Objekte werden dann an der Klasse *ParameterManager* übergeben.

Das angesprochene Baumelement muss aus das TESTONA-Diagramm geladen werden. Anhand der Identifikationsnummer werden die verschiedene Baumelemente einzeln abgefragt bis das richtige Baumelement gefunden wurde. Danach wird das Baumelement an das ParameterManager Objekt übergeben.

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

[1] 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) wo die Anforderung gespeichert ist und die Schnittstelle zur Datenbank. Anhand dieser Informationen wird die Verbindung zur Datenbank später aufgebaut.
- **RequirementsListTag**: ist eine *Map* mit alle in TESTONA gespeicherten Anforderungen. Das *Key* ist die Anforderungsidentifikationsnummer und der Wert der Inhalt der Anforderung. Eine Anforderung aus dieser Liste wurde an das Baumelement verknüpft.
- **VariantListTag**: eine Liste mit alle in TESTONA definierten Varianten.

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

5.1.2 ResourceSetListener

Das auslösendes Element ist, dass der Benutzer eine Anforderung mit einen Baumelement verknüpft. Um dieses Geschehen abzufragen muss ein *Listener* programmiert werden, dass das Interface *ResourceSetListener*[2] implementiert. Da dass *VariantsManager* ein solcher *Listener* bereits besitzt, wurde dieser erweitert. Die Aufgabe des *ResourceSetListener* ist, dem Benutzer zu benachrichtigen, wenn sich eine Ressource ändert. Das *Listener* „hört“ auf die reinkommenden Nachrichten (*ResourceSetChangeEvent*) und wertet den Inhalt dieses Events aus.

Das *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 die Ressource zu diesem Zeitpunkt sich schon geändert hat.

Um die Schreibrechte zu besitzen, wurde dann die Methode *transactionAboutToCommit(Event e)* betrachtet. Diese Methode wird aufgerufen bevor eine Änderung geschieht. Es wird benachrichtigt, dass eine Änderung geschehen wird und welche Objekte (ein Baumelement und die angehängte Anforderung) betrachtet werden. Zu diesem Zeitpunkt besitzt die Klasse noch Schreibrechte auf die Baumelemente und kann Änderung vornehmen. Außerdem hat die Methode als Rückgabewert ein *Command*. So können Kommandos an das Commandstack angekettet werden. Das Befehl, ein Parameter an ein Baumelement hinzufügen, wird zum richtigen Zeitpunkt ausgeführt, dank der Anordnung des Commandstacks. Mehr zu Kommandos wird in Kapitel 5.1.5 erwähnt.

[2] http://download.eclipse.org/modeling/emf/transaction/javadoc/1.0.3/org/eclipse/emf/transaction/ResourceSetListener.html

Die empfangene Nachricht beinhaltet folgende Elemente:

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

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

1. zum ersten Mal an einem Baumelement verknüpft
2. an einem Baumelement verknüpft, wo schon ändere Anforderungen verknüpft sind
3. gelöscht

Für alle drei Fälle muss der *Listener* wissen, an welches Element wurde das Event ausgelöst und welche Anforderung wurde hinzugefügt oder gelöscht. Zu bemerken ist, dass zu diesem Zeitpunkt noch nicht festgestellt werden kann, ob in der Anforderung ein Parameter definiert ist.

Eine Anforderung hinzufügen

Ist der *Notifier* eine Instanz von *TestonaClass*, so wurde eine Anforderung zum ersten mal an ein Baumelement 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 Baumelement) und aus der Variable *NewValue* die Anforderungskennung (diese wurde in DOORS vergeben) abgefragt. Beide Werte werden an dem *Variantsmanager* weitergegeben um danach der 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 es *null* sein kann. Wenn kein Parameter in der Anforderung definiert ist, so muss auch kein *ParameterTag* an das Baumelement hinzugefügt werden und es muss kein Kommando ausgeführt werden. Weiterhin, in der Methode *transactionAboutToCommit* werden andere *Events* ausgewertet die auch 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());
```

So wird das Kommando für das Hinzufügen eines *ParameterTags* 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 zurückgegebenes 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 Baumelement mindestens eine Anforderung besitzt und eine weitere Anforderung wird hinzugefügt, ist der *Notifier* eine Instanz von *RequirementTag* und `OldValue == null && NewValue != null`. Aus dem *Notifier* wird die ID des Baumelements abgerufen und *NewValue* ist besitzt die Anforderungskennung. Bevor die neue Anforderung hinzugefügt wird, werden die alten gelöscht und alle neue hinzugefügt.

Anforderungen löschen

Ist der *Notifier* eine Instanz von *RequirementTag* so wurde eine Anforderung von einem Baumelement gelöscht oder es wurde eine neue Anforderung an einen Baumelement hinzugefügt, indem bereits Anforderung verknüpft sind. 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 Baumelement gelöscht wird, dass nur eine Anforderung beinhaltet. Dieses *Event* teilt sich in zwei Nachrichten. Die erste Nachricht beinhaltet die Anforderungskennung als ein *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 dem *VariantsManager* übergeben um als Rückgabewert wird das Befehl für das Löschen eines *ParameterTags* erwartet.

Der zweite Fall beschreibt, dass eine oder mehrere Anforderung aus ein Baumelement gelöscht werden, wenn ein Baumelement mindestens eine Anforderung besitzt. Bevor die Anforderung hinzugefügt werden kann, werden die im Baumelement beinhaltende Anforderung zuerst gelöscht. Danach werden alle alte Anforderungen neu hinzugefügt sowohl als die neue Anforderung.

Wenn das Baumelement bereits eine Anforderung besitzt und eine zweite wird hinzugefügt, beinhaltet das *Notifier* die ID des Baumelements und *OldValue* die Anforderungskennung als *String*. Besitzt das Baumelement mehr als eine Anforderung und eine neue wird hinzufügt, kann aus dem *Notifier* die ID des Bauelements abgefragt werden. Der Unterschied liegt daran, dass *OldValue* jetzt eine Liste von *Stringwerte* 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 Baumelement gelöscht.

5.1.3 Parameter-*Tag*

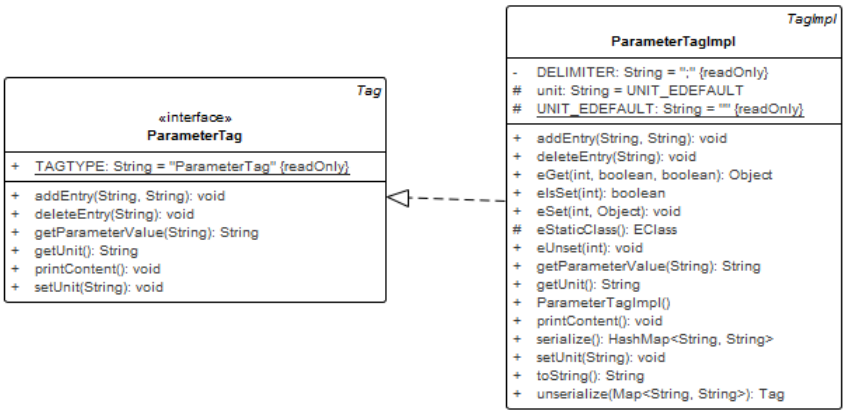

Abbildung 5.2: Darstellung der Klasse ParameterTagImplung das Interface ParameterTag

ParameterTag

Das Interface *ParameterTag* dient als Schnittstelle für den Zugriff auf das Inhalt von ParameterTagImpl. Das Interface erweitert *Tag*. Das *Tag* Interface ist ein in TESTONA allgemein implementiertes Modell, welches auf ein Interface und eine implementierende Klasse basiert (siehe Kapitel 3.4.5). In das Interface wird immer das 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.

ParameterTagImpl

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

weiter angewendet werden.

Jedes *Tag* hat als Eigenschaft einen Namen. In diese Klasse entspricht der Name, ein Parametername der aus der Parametertabelle gelesen wurde. Weiterhin ist über *ID* eine eindeutige Identifikationsnummer definiert, um *Tags* des gleichen Typs voneinander zu unterscheiden. Es wurde ein weiteres Attribut implementiert, dass die Einheit (*unit*) des Parameters definiert

Der Inhalt von das *Tag* ist durch ein *EcoreEMap<String, String>* definiert. Das erste String ist ein Schlüsselwert und das zweite String der Wert. Im diesem Fall 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]	
Content	Key	Value
	Generic	50
	Cabrio	100
	Kombi	200
	Limo	300

Tabelle 5.1: Beispiel eines ParameterTags

Anhand der Methode *getContent()* kann der Inhalt eines *Tags* aufgerufen werden und mit der Methode *addEntry()* werden Inhalte hinzugefügt. Das Gegenteil kann man mit der Methode *deleteEntry()* erreichen. Wie die Methoden genauer angewendet werden, wird im Listing 5.2 gezeigt.

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 wird ein *HashMap<String, String>* Objekt zurückgegeben. Das erste *String* beinhaltet der Name des Parameters gefolgt von die Variantennamen. Die Werte sind durch ein Vereinbartes Trennzeichen (;) getrennt. Das zweite *String* beinhaltet die Einheit des Parameters und die Parameterwerte (in einer geordneten Reihenfolge). Der XML Eintrag für das obige Beispiel (Tabelle 5.2) 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

welches Typ von *Tag* es sich handelt. Die Methode *unserialize* bekommt als Eingansparameter die in *serialize()* erstellte Zeile als ein *Map<String, String>* Objekt. Durch das Trennzeichen werden die jeweilige Werten voneinander unterschieden und zu den zugehörigen Variablen (name, unit, content) zugeordnet. Beider Vorgänge funktionieren automatisiert dank der Vererbung.

5.1.4 ParameterManager

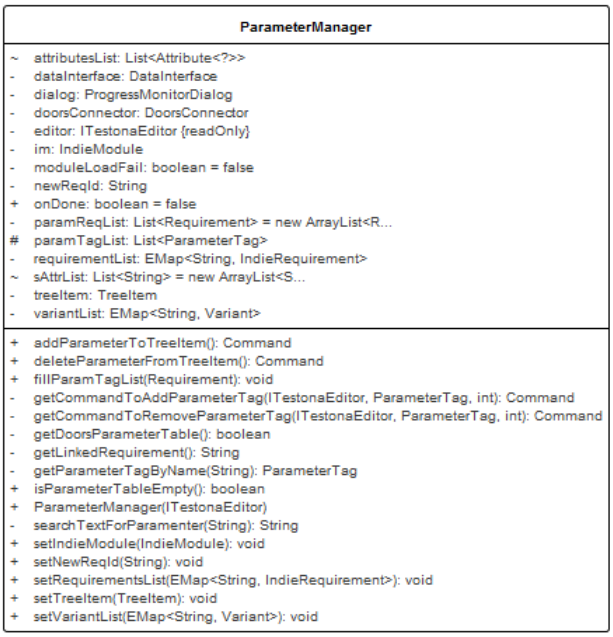

Abbildung 5.3: Klasse ParameterManager

Die Klasse *ParameterManager* überprüft und startet die Parametererersetzung. 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 siehe Kapitel 5.1.6 erklärt.

Mit dem Aufruf der Methode *addParameterToTreeItem()* wird die Parametererersetzung 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 es 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 erforderliche Angaben zum Verbindungsaufbau befinden sich in der Variable *im* vom Typ *IndieModul* (diese wurde von der Klasse *VariantsManager* gesetzt aus das *RequirementsConnectionTag*). Das Objekt der Klasse *DoorsConnetor* 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. Dieser blockiert die Benutzeroberfläche und zeigt ein Fortschrittsbalken. Nach einem erfolgreichen Verbindungsaufbau wird die Parametertabelle geladen. Der Pfad zu der Parametertabelle in DOORS ist als Attribut des Hauptmoduls gespeichert. In das Objekt *im* befindet sich der Pfad zum Hauptmodul. Die aktuell implementierte DOORS API in TESTONA kann nicht diese Attribute abfragen. Eine in Moment in Entwicklungszustand neue API wird in der Lage sein, die Attribute abfragen zu können. An dieser Stelle wurde als Kompromiss eine Konstante angelegt, in der sich der Pfad zur Tabelle befindet.

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 ein ParameterTag. Zu beachten ist, dass innerhalb DOORS jede Zeile in ein Modul als eine Anforderung betrachtet wird. Danach wurde die Namensgebung vereinbart und der Übergabeparameter ist vom Typ *Requirement*. In das *Requirement* Objekt befinden sich die Variantennamen und die Parameterwerte. Der folgende Quelltextausschnitt kümmert sich im 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)).
          getValueAsString());
8   } else {
9     //Füge ein Eintrag hinzu
10    tempTag.addEntry(sAttrList.get(i),
11        req.getValue(attributesList.get(i)).getValueAsString());
12  }
13 }
14 paramTagList.add(tempTag);
```

Listing 5.2: Auszug von der Erstellung der ParameterTags

Die Variable `attributesList` beinhaltet die in DOORS definierten Varianten[3]. Die Liste wird iteriert und die Werte werden in das *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 geladene Werte werden kontrolliert ob sie ungleich ein leerer *String* sind[4], bevor sie in ein *ParameterTag* gespeichert werden.

[3]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.

[4]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.

Wenn alle vorhandenen Varianten für dieses 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 in das Modul definiert ist, aber keine weitere Informationen wurden ausgefüllt. Wenn Inhalte vorhanden sind, wird das *ParameterTag* zur eine Liste von *ParameterTags* hinzugefügt.

Nachdem die Parameter geladen werden konnten und die Liste befühlt ist, wird die Verbindung mit DOORS geschlossen. Das Dialogfenster wird danach geschlossen und die Benutzeroberfläche wird freigegeben. Während dessen wird ein der Liste das 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 *ResourSetListener* weitergibt. Dort wird das Kommando verarbeitet.

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

5.1.5 Kommandos

Für das Hinzufügen bzw. Löschen eines Parameters in einem Baumelement wurden zwei Klassen implementiert, dass aus die Klasse *RecordingCommand* erben. 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[5]. Eine der implementierten Klassen erzeugt ein Kommando für das Hinzufügen eines *ParameterTags* an einem Baumelement[6], während der zweiten für das Löschen zuständig ist[7]. Der Konstruktor der beiden Klassen haben die gleichen Parametern. Es werden der aktuelle Editor-Objekt übergeben, sowie das Baumelement ID und das *ParameterTag*.

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

Soll eine *ParameterTag* gelöscht werden, dann wird die Methode *removeTag(ParameterTag pTag)* aufgerufen. Danach muss auch der Name des Baumelementes neu gesetzt werden. Wenn ein Baumelement ein Parameter besitzt, je nach aktive Variante, wird die Darstellung geändert (siehe Kapitel 4.2). Dafür wird der Name des Baumelementes geladen und

[5] 14.02.2015; http://download.eclipse.org/modeling/emf/transaction/javadoc/ 1.2.3/org/eclipse/emf/transaction/RecordingCommand.html

[6] AddParameterTagCommand

[7] RemoveParameterTagCommand

falls eine Rücksetzung nötig ist, wird diese gemacht. Genaueres dazu wird in Kapitel 5.2 veranschaulicht.

5.1.6 Die DOORS Verbindng

DoorsConnector

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

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

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

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

```
1 try {
2   connector = ConnectorManager.getInstance()
3         .getConnector(im.getInterId());
4   dataInterface = conManager.getNewDataInterface(connector, this
      );
```

```
5 } catch (ExtensionException e) {
6   e.printStackTrace();
7 }
```

Listing 5.3: Verbindungsaufbau

Um dem richtigen `connector` aufzurufen muss aus das *IndieModul* (`im.getInterId()`, siehe Kapitel 5.1.1) die Interfacekennung als Übergabeparameter angegeben werden. Als nächstes kann über den *ConnectionManager* eines neues *DataInterface* erzeugt werden, wo der `connector` und das aktuelle Objekt (*DoorsConnector*) übergeben werden.

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

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

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

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

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

RequirementDataListener

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

```
1 @Override
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 geladenes Modul vorhandenen Attribute. Im diesem Fall (treu zum Beispiel mit dem Auto) sind es:

- Paramenter Name
- Default Value
- Cabrio Value
- Kombi Value
- Limo Value

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

`fillParamTagList` erzeugt die *ParameterTags* und wurde in Kapitel 5.1.4 erläutert.

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

RequirementLoadListener

Der *RequirementLoadListener* reagiert wenn eine Zeile nicht völlstandigt 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 das DOORS Modul geladen wurden. Das wartenden Dialogfenster aus 5.1.4 wird benachrichtigt, dass es geschlossen werden kann. Die Benutzeroberfläche von TESTONA ist somit wieder für den Benutzer erreichbar.

5.2 Darstellung der Parameterwerte

Nachdem Parameter in Baumelemente 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 für jedes Baumelement überprüft werden. Gibt es ein gültiger 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 5.2: Parameter *max_speed*

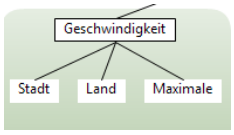

Abbildung 5.4: Das generische Baum

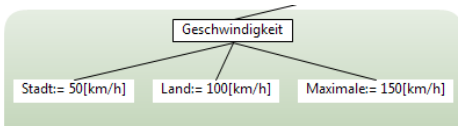

Abbildung 5.5: Aktive Variante Cabrio

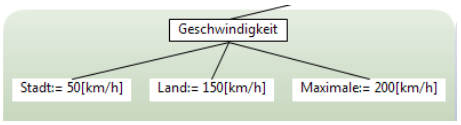

Abbildung 5.6: Aktive Variante Limo

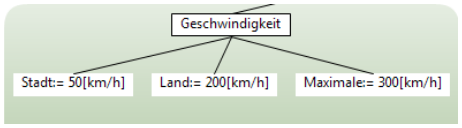

Abbildung 5.7: Aktive Variante Kombi

Wenn die Einheit des Parameters definiert ist, wird diese auch angezeigt. Zwischen Baumelementname und der Wert befinden sich die Zeichen „:=“ als Trennzeichen. Diese Maßnahme wurde eingeführt, da Baumelemente unter einem Knoten dürfen nicht den gleichen Namen besitzen. So kann das Programm und der Benutzer jedes Baumelemente 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 Baumelemente, falls es nötig ist, neu benannt.

Die Umbenennung der Baumelemente erfolgt, wie bei das 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 and 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 Baumelement ein *ParameterTag* hat, wird überprüft ob für die aktive Variante ein Parameterwert definiert ist. Wenn ein Parameterwert gefunden wurde, wird das Baumelement immer mit dem folgenden Format neu benannt:

$$name := wert[einheit]$$

Wobei nur *einheit* ein optionaler Wert ist. Falls es kein Parameterwert für die Variante definiert wurden ist, wir der Standartname angezeigt (name).

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

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

5.3 Testfallgenerierung und Optimierung

Erläuterung der Lösungen zu 4.3

Kapitel 6

Evaluierung

6.1 Chances of Failure

Kapitel 7

Zusammenfassung und Ausblick

7.1 Optimierungskriterien

Abbildungsverzeichnis

Listings

Tabellenverzeichnis

Anhang A

Anhang

A.1 CD

Inhalt:

- Quellen
- PDF-Datei dieser Arbeit

A.2 ParameterMananger Klassendiagramm

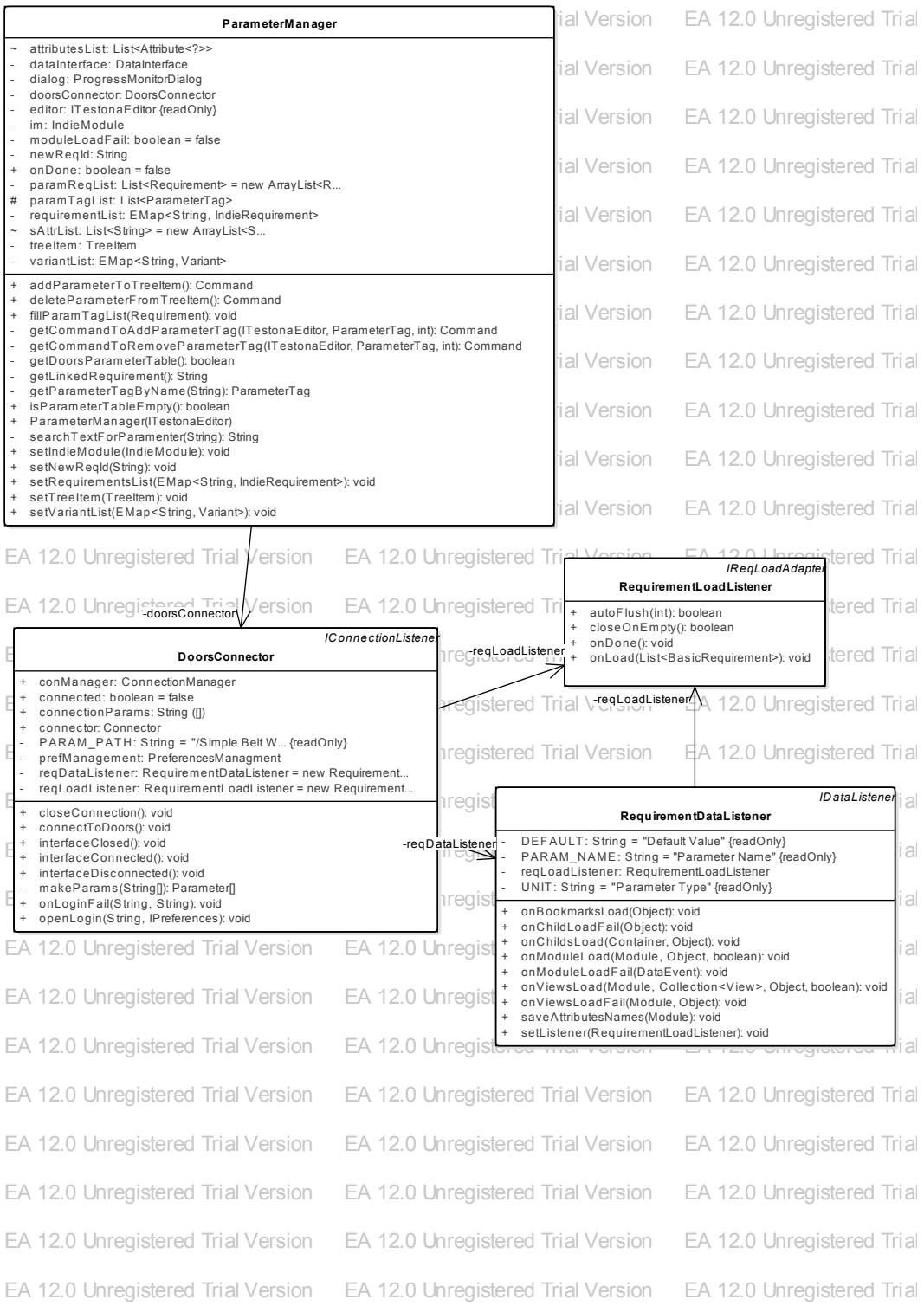

Abbildung A.1: Klassendiagramm von ParameterManager.java

A.3 code 2

```
1 hier kommt auch java code
```

Literaturverzeichnis

[1] Berner & Mattner, http://www.testona.net. *TESTONA*, Oktober 2014.

[2] Eclipse Foundation, http://eclipse.org. *Eclipse*, Oktober 2014.

[3] Eclipse Foundation, http://help.eclipse.org. *Eclipse Help*, Oktober 2014.

[4] Eclipse Foundation, http://eclipse.org/swt. *Eclipse SWT*, Oktober 2014.

[5] Stephan Grünfelder. *Software-Test für Embedded Systems*. dpunkt.verlag, 2013.

[6] M. Grochtmann and K. Grimm. *Classification Trees For Partition testing, Software testing, Verification & Reliability, Volume 3, Number 2*. Wiley, 1993.

[7] IBM, http://www-03.ibm.com/software/products/de/ratidoor. *IBM DOORS*, Oktober 2014.

[8] IEEE Computer Society, https://courses.cs.ut.ee/MTAT.03.159/2013_spring/uploads/Main/SWT_comb paper1.pdf. *Combinatorial Software Testing*, 2009.

[9] Quality Week 1995, http://www.systematic-testing.com/documents/qualityweek1995_1.pdf. *Test Case Design Using Classification Trees and the Classification-Tree Editor CTE*, 1995.

[10] Iris Reinhartz-Berger, Arnon Sturm, Tony Clark, Sholom Cohen, and John Bettin. *Domain Engineering: Product Lines, Languages, and Conceptual Models*. Springer, 2013.

[11] Christopher Robinson-Mallet. An approach on integrating models and textual specifications. *Model-Driven Requirements Engineering Workshop (MoDRE)*, 2012.

[12] Christopher Robinson-Mallet, Matthias Grochtmann, Joachim Wegener, Kens Köhnlein, and Steffen Kühn. Modelling requierements to support testing of product lines. *Third International Conference on Software Testing, Verification, and Validation Workshops*, 2010.

[13] Softscheck, http://www.softscheck.com/publications/120725_Code_Coverage.pdf. *Code coverage tools*, June 2012.

[14] H. Tremp and M. Ruggiero. *Application Engineering: Grundlagen für die objektorientierte Softwareentwicklung mit zahlreichen Beispielen, Aufgaben und Lösungen*. Compendio Bildungsmedien, 2011.

[15] Gerhard Versteegen. *Anforderungsmanagement: Formale Prozesse, Praxiserfahrungen, Einführungsstrategien und Toolauswahl*. Springer, 2004.

[16] Lars Vogel and Mike Milinkovich. *Eclipse 4 RCP*. Lars Vogel, 2013.

[17] Wikipedia, http://de.wikipedia.org/wiki/Klassifikationsbaum-Methode. *Klassifikationsbaum-Methode*, Oktober 2014.