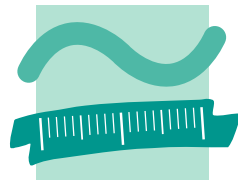


Masterarbeit

**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: 27. Januar 2015

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

Dankessage

Inhaltsverzeichnis

| | | |
|----------|---|-----------|
| 1 | Einleitung | 2 |
| 2 | Aufgabestellung | 3 |
| 3 | Fachliches Umfeld | 5 |
| 3.1 | TESTONA | 5 |
| 3.1.1 | Klassifikationsbaum-Methode | 5 |
| 3.1.2 | Testfälle und Testfallgenerierung | 7 |
| 3.1.3 | Abhängigkeitsregeln | 7 |
| 3.2 | IBM Rational DOORS | 10 |
| 3.3 | Variantenmanagement | 11 |
| 3.4 | Entwicklungsumgebung und Programmiersprache | 12 |
| 3.4.1 | Eclipse | 12 |
| 3.4.2 | Plug-ins | 12 |
| 3.4.3 | Standard Widget Toolkit (SWT) | 13 |
| 3.4.4 | JFace | 14 |
| 4 | Lösungsansatz | 15 |
| 4.1 | Parameterspeicherung | 16 |
| 4.2 | Visualisierung | 18 |
| 4.3 | Testgültigkeit | 19 |
| 4.4 | Optimierungskriterien | 20 |
| 4.5 | Benutzeroberfläche | 21 |
| 5 | Systementwurf | 22 |
| 5.1 | Variantenmanagement und Parameter | 22 |
| 5.1.1 | Listener | 22 |
| 5.1.2 | VariantsManager | 23 |
| 5.1.3 | ParameterManager | 23 |

| | |
|---|------------|
| <i>INHALTSVERZEICHNIS</i> | <i>III</i> |
| 5.1.4 DoorsConnector | 23 |
| 5.2 Testfallgenerierung und Optimierung | 25 |
| 6 Evaluierung | 26 |
| 6.1 Chances of Failure | 26 |
| 7 Zusammenfassung und Ausblick | 27 |
| A Anhang | 30 |
| A.1 CD | 30 |
| A.2 code 1 | 31 |
| A.3 code 2 | 32 |
| Literatur- und Quellenverzeichnis | 33 |

Kapitel 1

Einleitung

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

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

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

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

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

Kapitel 2

Aufgabestellung

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

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

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

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

Der derzeitige Varianten-Management-Ansatz in TESTONA ist nicht in der Lage für die Test-

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

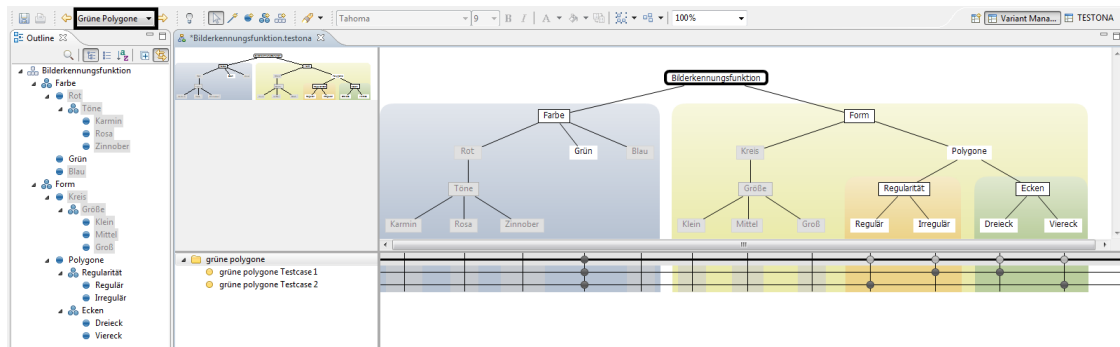


Abbildung 2.1: Richtige Auswahl der Klassen und Klassifikationen für die Testfallgenerierung

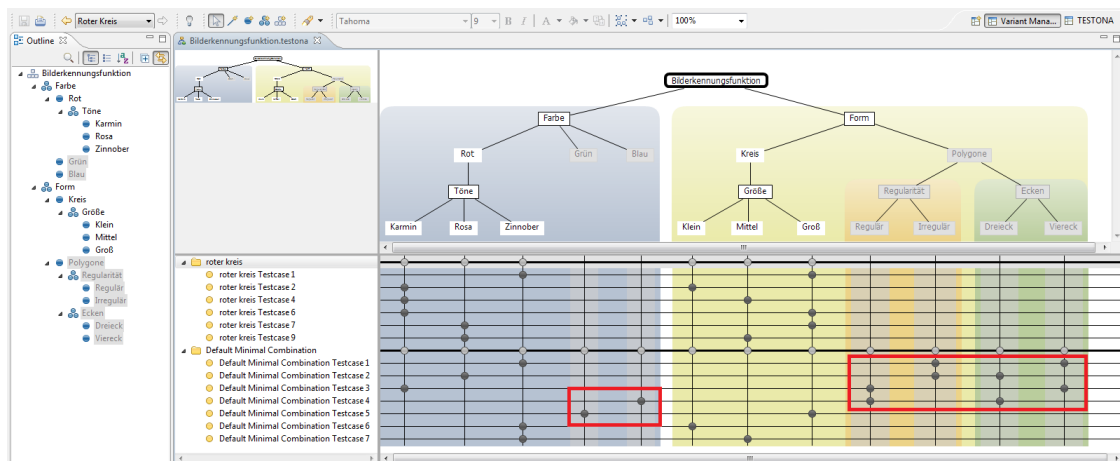


Abbildung 2.2: Ungültige Auswahl der Klassen (Grün und Blau) und Klassifikationen (Regelmäßigkeit und Ecken) für die Testfallgenerierung

Kapitel 3

Fachliches Umfeld

3.1 TESTONA

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

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

3.1.1 Klassifikationsbaum-Methode

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

Diese Methode besteht aus zwei wichtigen Schritten:

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

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

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

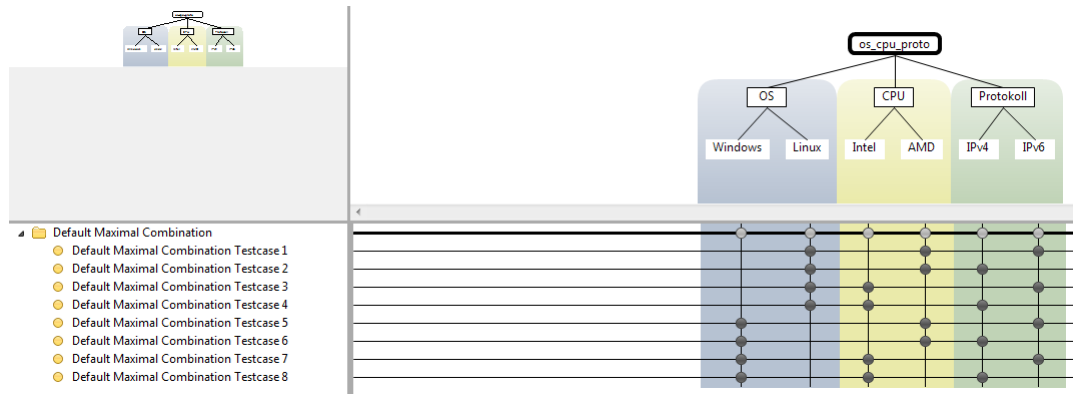


Abbildung 3.1: Baum und Testfälle ohne Kombinatorik

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

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

Tabelle 3.1: Testfälle mittels paarweise Kombinatorik

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

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

²Basierend auf funktionelle und technische Anforderungen, Use-Cases

3.1.2 Testfälle und Testfallgenerierung

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

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

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

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

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

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

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

3.1.3 Abhängigkeitsregeln

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

³International Software Testing Qualification Board

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

⁵Das Fehlverhalten kann zu geringen materiellen oder immateriellen Schäden führen

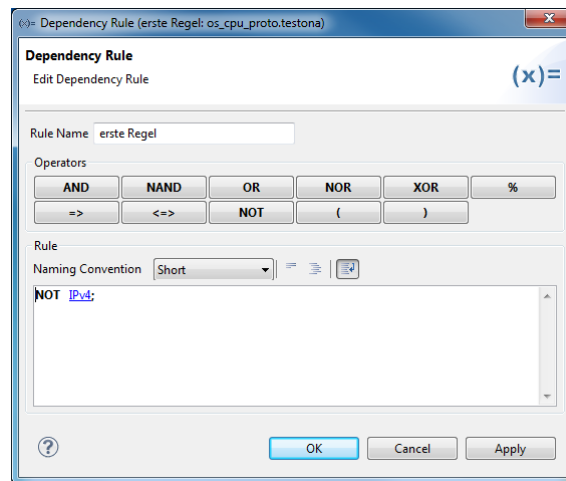


Abbildung 3.2: Abhängigkeitsregeln Bearbeitung

Boolesche Operatoren:

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

Durch die Verwendung dieser Operatoren werden folgende Abhängigkeitsregeln definiert:

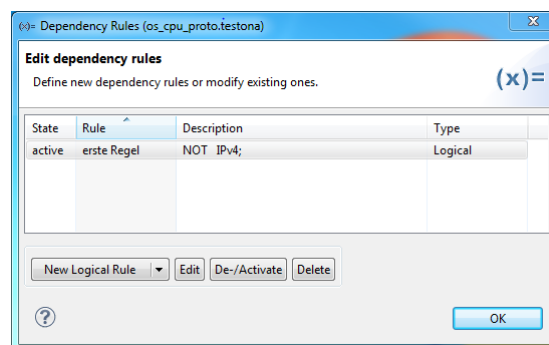


Abbildung 3.3: Abhängigkeitsregeln Übersicht

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

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

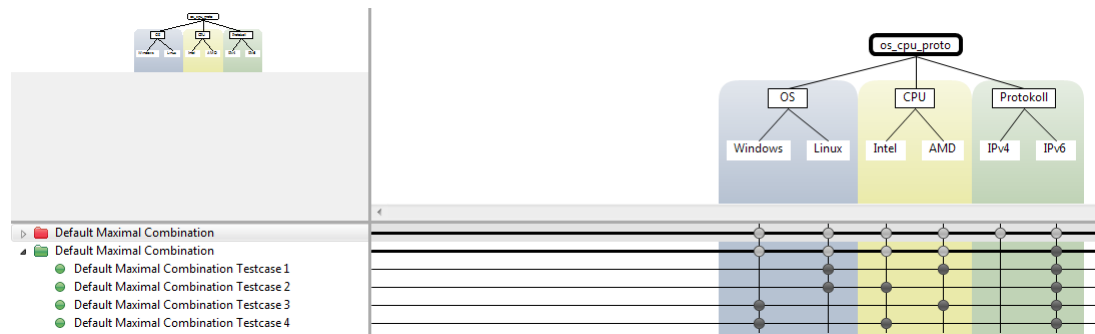


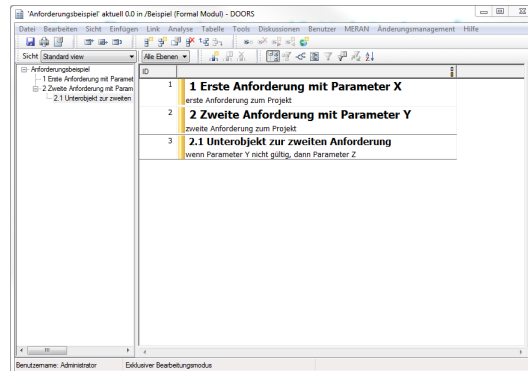
Abbildung 3.4: Testfälle mit Anwendung der Abhängigkeitsregeln aus 3.2 und 3.4

Lichtschalter1 XOR Lichtschalter2
NOT > 5V

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

3.2 IBM Rational DOORS

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



The screenshot shows the IBM Rational DOORS application window. The title bar reads 'Anforderungsbeispiel aktuell 0.0 in /Beispiel (Format Modul) - DOORS'. The menu bar includes 'Datei', 'Bearbeiten', 'Suche', 'Einfügen', 'Link', 'Analyse', 'Tabelle', 'Tools', 'Diskussionen', 'Benutzer', 'MERAN', 'Änderungsmanagement', and 'Hilfe'. The toolbar contains various icons for file operations, search, and analysis. The main window is divided into two panes. The left pane, titled 'Anforderungsbeispiel', shows a tree structure with three items: '1 Erste Anforderung mit Parameter X', '2 Zweite Anforderung mit Parameter Y', and '2.1 Unterobjekt zur zweiten Anforderung'. The right pane displays a table with three rows corresponding to these items. The first row is '1 Erste Anforderung mit Parameter X' with the description 'erste Anforderung zum Projekt'. The second row is '2 Zweite Anforderung mit Parameter Y' with the description 'zweite Anforderung zum Projekt'. The third row is '2.1 Unterobjekt zur zweiten Anforderung' with the description 'wenn Parameter Y nicht gut, dann Parameter Z'. The status bar at the bottom shows 'Benutzername: Administrator' and 'Exklusiver Bearbeitungsmodus'.

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

Abbildung 3.5: Beispiel einer Tabelle in DOORS

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

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

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

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

3.3 Variantenmanagement

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

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

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

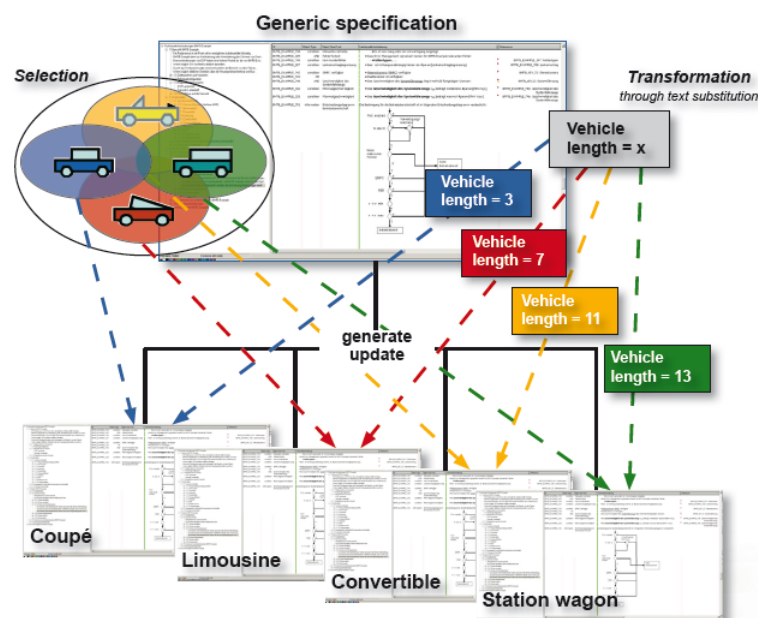


Abbildung 3.6: Betrachtung von Varianten eines Wagens von der generischen Testspezifikation zur variantenspezifische Testspezifikation aus [12]

3.4 Entwicklungsumgebung und Programmiersprache

TODO

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

3.4.1 Eclipse

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

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

3.4.2 Plug-ins

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

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

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

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

Testona besteht momentan aus 129 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 Environment, dass der jeweilige Plug-in benötigt
- **Plug-in Dependencies:** schließt die Abhängigkeiten des Plug-ins mit der Eclipse Umgebung und anderen implementierten Plug-ins ein
- **src:** bezieht die Pakete und Java-Klassen ein
- **icons:** hier befinden sich die Bilderdateien (.gif, .png, etc) die in den Klassen für die Benutzeroberfläche aufgerufen werden
- **META-INF:** gibt eine Übersicht aller Einstellungen des Plug-ins, sowie die Möglichkeit diese über eine graphische Oberfläche zu bearbeiten
- **build.properties:** beinhalten die Einstellungen für das Compilieren des Plug-in. Es kann auch über die META-INF Datei bearbeitet werden.
- **plugin.xml:** hier werden die nötigen Erweiterungen für das Plug-in definiert. Es ist möglich direkt die *xml* Datei zu bearbeiten, oder die META-INF Oberfläche benutzen.

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

3.4.3 Standard Widget Toolkit (SWT)

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

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

- Ein *Display* deklarieren (repräsentiert die SWT Modus)
- Erstellen eines *Shell*, welche als Hauptfenster dient
- Erzeugung eines Widgets

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

- 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

3.4.4 JFace

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

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

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

Kapitel 4

Lösungsansatz

Um besser den Lösungsansatz zu verstehen, wird erstmal der Ablauf von der Erstellung von Anforderung bis zu einem Testfall erläutert. Als erstes wird in DOORS alle schon vordefinierte Anforderungen eingetragen (siehe 3.2). In diesen Anforderung können Parameter vorkommen, die für die Testfälle relevant sind. Die Parameter werden als folgendes 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, wo die jeweilige Parameterwerte eingetragen werden. Der Grund, dass es eine extra 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 150 km/h und bei einem Kombi 200 km/h. Dank dieser Tabelle werden die Varianten (Cabrio und Kombi) die richtige Parameterwerte zugewiesen.

Da Parameterwerte und Varianten schon verlinkt sind, müssen die Anforderungen, die Parametern 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. Es wird davon ausgegangen, dass der Tester bereits ein Klassifikationsbaum passend zum testenden Produkt und die dazu gehörige Anforderungen erstellt hat. Jetzt können die in DOORS definierte Varianten importiert. Als erstes muss der Tester die Bauelemente der richtigen Variante zuordnen (durch ein Ausschlussverfahren, es wird angenommen, dass alle Bauelemente in alle 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 bestimmte 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 Objekte gespeichert und einen *Tag* (Kennzeichen) gegeben und nach Programmende in die TESTONA Datei im XML Format gespeichert. Dank des *Tags* kann beim Programmstart wieder der Parameterwert gelesen werden und während das Programm ausgeführt wird, Änderungen vornehmen.

Einer der Besonderheiten von MERAN in Verbindung mit 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 Benutzer 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 das Anforderungstext von dieses Bauelement lesen und nach Parameter suchen. Als erstes wird davon ausgegangen, dass ein Bauelement nur eine Anforderung beinhalten kann und eine Anforderung nur ein Parameter beinhaltet. Wird in die gelesene Anforderung ein Parameter gefunden, so müssen die möglichen Werten dieses Parameters aus die DOORS Parameter-Tabelle gelesen und gespeichert werden. An dieser Stelle beginnt die Parameterspeicherung.

Für die Parameterspeicherung ergeben sich zwei Lösungswege. Die erste Lösung lautet, die Parametertabelle beim importieren 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 nach eine Verbindung mit DOORS nicht geprüft werden, oder 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 Bauelement verlinken und zugleich werden die Parameter gelesen und gespeichert. Der Nachteil ist, dass möglicherweise unnötige Daten in TESTONA gespeichert werden.

Die zweite Lösung ist der Gegensatz zu die erste. Hier werden jeweils nur die Daten gespeichert, die der Benutzer im Moment verbraucht. Wird eine Anforderung an einem Bauelement verlinkt, so muss eine Verbindung zu DOORS aufgebaut werden (wenn sie nicht vorhanden ist) und die Parametertabelle aufrufen und in TESTONA speichern. Der große Nachteil dieser Lösung ist, dass der Benutzer möglicherweise nicht eine Verbindung zu DOORS aufbauen kann (Server nicht Vorhanden, keine Zugriff Möglichkeiten, etc.). Welcher der beiden Lösungen implementiert wird, wird erst genauer analysiert und 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 an der richtigen Variante noch zugeordnet werden. Dafür ist vorgesehen, dass der Parameterwert als Eigenschaft des Bauelementes gespeichert wird (als ein *Tag* Objekt). Die Darstellungsstruktur für die Variantenansicht ist etwa folgendermaßen definiert:

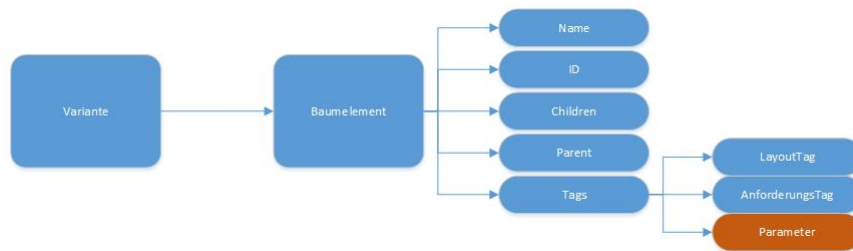


Abbildung 4.1: Darstellung der TESTONA Objekte für die Parameterspeicherung

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

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

| Variante | Maximalgeschwindigkeit | Anforderung |
|----------|------------------------|-------------|
| Cabrio | 150 | 1 |
| Kombi | 200 | 1 |
| Limo | 250 | 1 |

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

Wenn eine Baumelement mehr als eine Anforderung beinhaltet, mit verschiedene Parametern, wird die aktive Variante entscheiden, welches Parameter ausgewertet wird. Dabei muss beachtet werden, dass vorhandene Parameterwerte und Anforderung nicht gelöscht oder überschrieben werden.

TODO, Beispiel mit Baumbilder

4.2 Visualisierung

Wenn die Beziehungen zwischen Varianten, Parameter und Anforderungen erfolgreich entstanden sind, können jetzt die gespeicherte Parameterwerte angezeigt werden. Hier ist gefordert, dass wenn der Benutzer die Variantenansicht ändert (durch Betätigung an der Benutzeroberfläche) die Beschriftung (*Label*) des Bauelements (Maximaleschwindigkeit von Generic = X, Cabrio = 150, Kombi = 200) aktualisiert werden.

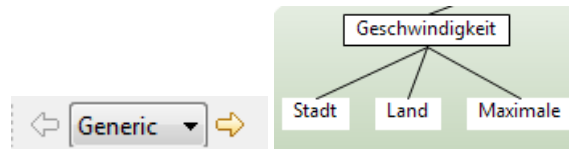


Abbildung 4.2: Aktive Variante Generic (default)

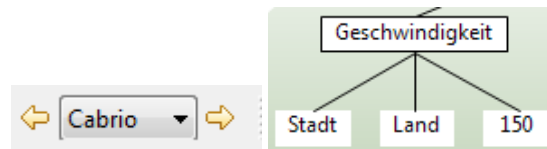


Abbildung 4.3: Aktive Variante Cabrio

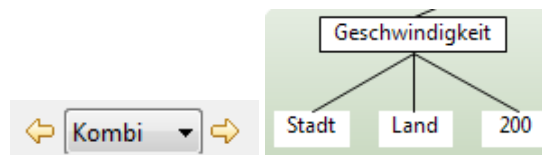


Abbildung 4.4: Aktive Variante Kombi

Dafür muss ein *Listener* beim Ändern der Ansicht eine Methode aufrufen, die sich um das aktualisieren der Werte und Neuzeichnung. Die Werten werden dynamisch aus den Inhalten der Bauelemente gelesen. Zur Erinnerung die Werte sind als *Tag* in jeden Bauelement gespeichert (siehe 4.1)

4.3 Testgültigkeit

4.4 Optimierungskriterien

4.5 Benutzeroberfläche

Kapitel 5

Systementwurf

5.1 Variantenmanagement und Parameter

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



Abbildung 5.1: Darstellung der Klassen

5.1.1 Listener

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

Als Inhalt der Nachricht wird folgendes empfangen:

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

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

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

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

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

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

5.1.2 VariantsManager

Die Hauptaufgabe des *VariantsManager* ist die Varianten zu regeln. Hier werden Bauelemente zu Varianten hinzugefügt und gelöscht. Dabei muss das Klassifikationsbaum neu gezeichnet werden. Diese Klasse kümmert sich auch, um die Umschaltung zwischen Varianten in der Variantenansicht (siehe Abbildung 4.2). Der *VariantsManager* setzt die Bauelement-ID und die Anforderungskennung in der Klasse *ParameterManager*. Die Klasse dient als Schnittstelle zwischen der *Listener* und die Klasse *ParameterManager*

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

5.1.3 ParameterManager

Die Klasse *ParameterManager* beinhaltet eine private innere Klasse *DoorsConnector*(siehe 5.1.4), diese kümmert sich um den Verbindungsaufbau mit DOORS.

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

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

5.1.4 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.

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 5.1.3 erwähnt, steht in der Anforderung in welches Modul sich die Anforderung befindet, sowie über welches Interface dieses Modul zu erreichen ist. Für den Verbindungsaufbau werden noch 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.
- **PreferenceManagement:** Hier werden die in TESTONA gespeicherte Verbindungsdaten behandelt. Es können Microsoft Access Verbindungsdaten gespeichert werden, aber wir werden nur DOORS betrachten.
- **Connector:** beschreibt eine einzelne Verbindung, hat ein *DataInterface*- und *PreferenceManagement*objekt
- **ConnectionManager:** Singleton. Die Klasse handelt aktive und offene Verbindungen. Hier werden die *ConnectionListeners* und das *DataInterface* für den richtigen *Connector* geregelt.

Wenn die Verbindung aufgebaut ist, müssen die Ordner und Projekte geöffnet werden bis zur Parametertabelle.....

muss programmiert werden

Parameter lokal in Objekt gespeichert

Datenmenge relativ gering, deswegen eine Verbindung

Da die Datenmenge von einer Parametertabelle relativ gering ist, wurde hier, bezüglich der offenen Frage bei dem Lösungsansatz (siehe Kapitel 4.1), die Parametertabelle komplett importiert. Es wird angenommen, wenn der Benutzer die Parameterersetzung für ein Parameter wahrnimmt, dass er es für die anderen Parameter auch wahrnehmen wird. Darum wird die Parametertabelle erst bei der Verknüpfung von einer Anforderung mit einem Bauelement importiert, und nicht beim Import der Anforderungen. Weiterhin, um die Rechen- und Reaktionszeit von TESTONA gering zu halten, wird die Tabelle komplett importiert, um nicht für jedes neue Parameter eine Verbindung mit DOORS aufzubauen.

Am Ende wird die Verbindung getrennt

5.2 Testfallgenerierung und Optimierung

Erläuterung der Lösungen zu 4.3

Kapitel 6

Evaluierung

6.1 Chances of Failure

Kapitel 7

Zusammenfassung und Ausblick

Abbildungsverzeichnis

| | | |
|-----|--|----|
| 2.1 | Richtige Auswahl der Klassen und Klassifikationen für die Testfallgenerierung | 4 |
| 2.2 | Ungültige Auswahl der Klassen (Grün und Blau) und Klassifikationen (Regularität und Ecken) für die Testfallgenerierung | 4 |
| 3.1 | Baum und Testfälle ohne Kombinatorik | 6 |
| 3.2 | Abhängigkeitsregeln Bearbeitung | 8 |
| 3.3 | Abhängigkeitsregeln Übersicht | 8 |
| 3.4 | Testfälle mit Anwendung der Abhängigkeitsregeln aus 3.2 und 3.4 | 9 |
| 3.5 | Beispiel einer Tabelle in DOORS | 10 |
| 3.6 | Betrachtung von Varianten eines Wagens von der generischen Testspezifikation zur variantenspezifische Testspezifikation aus [12] | 11 |
| 4.1 | Darstellung der TESTONA Objekte für die Parameterspeicherung | 17 |
| 4.2 | Aktive Variante Generic (default) | 18 |
| 4.3 | Aktive Variante Cabrio | 18 |
| 4.4 | Aktive Variante Kombi | 18 |
| 5.1 | Darstellung der Klassen | 22 |

Listings

| | | |
|-----|--|----|
| 3.1 | Beispiel einer SWT Anwendung | 14 |
|-----|--|----|

Anhang A

Anhang

A.1 CD

Inhalt:

- Quellen
- PDF-Datei dieser Arbeit

A.2 code 1

lhier kommt java code

A.3 code 2

lhier kommt auch java code

Literaturverzeichnis

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