

Masterarbeit

**Variantenspezifische Abhängigkeitsregeln und
Testfallgenerierung in TESTONA**



BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN

University of Applied Sciences

Fachbereich VI - Technische Informatik - Embedded Systems



BERNER & MATTNER
AN ASSYSTEM COMPANY

Eingereicht am: 6. Januar 2015

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

Dankessage

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 und -umfänge [?]. Im Kapitel 3.1 wird weiteres zu dieses Programm und die Funktionsweise erläutert.

Die Erweiterung des Programmes besteht aus verschiedene Themen. Eins 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 stärker die Produktvarianten 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 der jeweilige Produktvariante zugeordnet werden. Aus diesem Grund kann es zu Konflikte bei der Testfallgenerierung kommen, bzw. inkohärente Testfälle.

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

Im Kapitel 2 wird genauer die Aufgabe dieser Masterarbeit erläutert und in den Kapiteln 4 und ?? 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 Parameter, 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 in TESTONA eingebunden werden, durch eine erfolgreiche Anmeldung bei DOORS (über die TESTONA Oberfläche) und ein gezieltes Auswählen der gewünschten Varianten. Hierbei sollen die in den Anforderungen definierten Variablen (z.B. eine Geschwindigkeit oder 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. Entscheidend für die Lösung werden die erworbenen Kenntnisse über die Benutzung des Variantenmanagements durch einen Testingenieurs. Bei der Lösung ist zu beachten, dass eine komplette Testfallabdeckung garantiert werden muss.

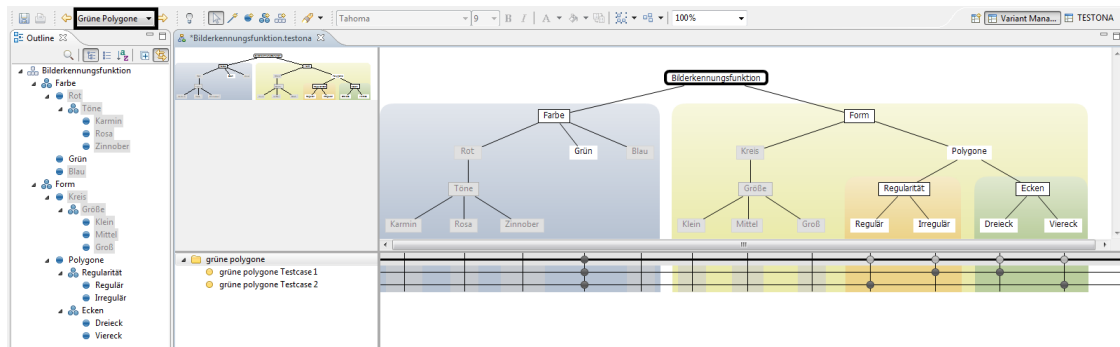


Abbildung 2.1: Richtige Auswahl der Klassen und Klassifikationen für die Testfallgenerierung

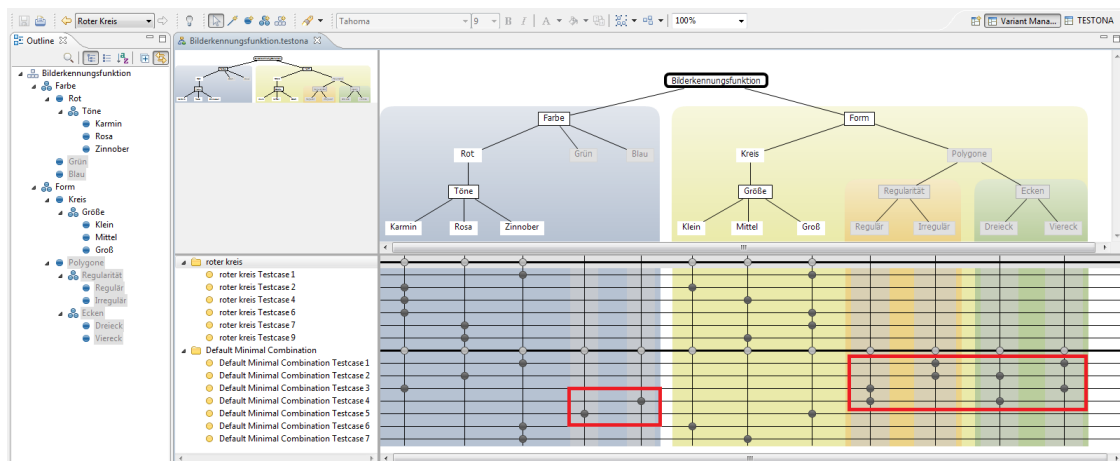


Abbildung 2.2: Ungültige Auswahl der Klassen (Grün und Blau) und Klassifikationen (Regularität und Ecken) für die Testfallgenerierung

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 in jedem Fachgebiet benutzen, nicht nur bei Software- oder Funktionstests.

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

3.1.1 Klassifikationsbaum-Methode

In 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[?].

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[?]. Bei Berner & Mattner aus rechtlichen Gründen zu TESTONA umbenannt

Ansatzpunkt sind die Funktionale Anforderungen (siehe 3.2) eines zu testendes Objekt. Um die Testfälle zu definieren und erzeugen, folgt die Methode das Prinzip des kombinatorischen Testentwurfs [?]. Dieses Prinzip hilft bei der Detektierung von Fehler in frühe Schritte des Testvorgangs. Nicht jeder einzelne Parameter steuert ein Fehler bei, eher werden Fehler durch die Interaktion verschiedene Parameter verursacht. Betrachten wir ein einfaches Beispiel, indem ein Programm auf Windows oder Linux laufen soll, 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).

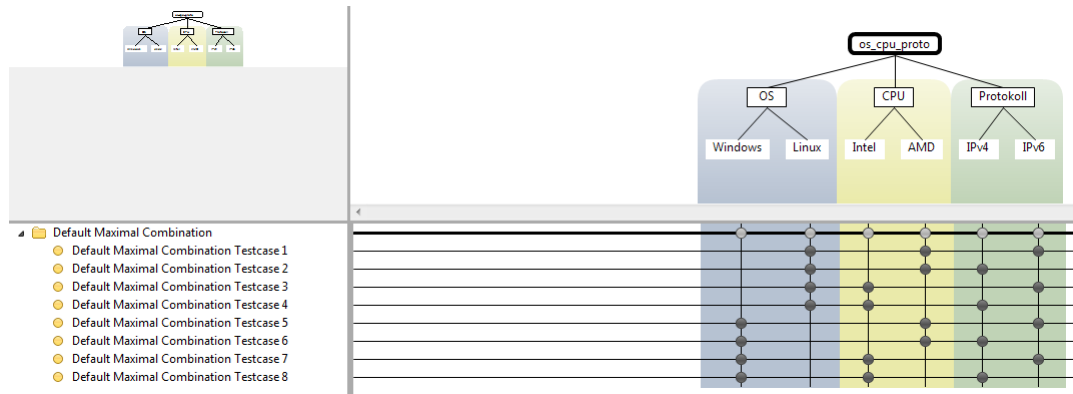


Abbildung 3.1: Baum und Testfälle ohne Kombinatorik

Verwenden wir dafür der kombinatorische 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 ein Testfall gedeckt[?].

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 diesen einfachen kombinatorischen Entwurf ist bei 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 paarweise Kombination muss der Tester nur 180 Testfälle betrachten.

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

²Basierend auf funktionelle und technische Anforderungen, Use-Cases

3.1.2 Testfälle und Testfallgenerierung

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

zu verstehen. Da TESTONA in allen Testphasen einsetzbar ist, kann effizient die Arbeitszeit 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 dem 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.[?]

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 vereinbart und die Auswertung der Kritikalitätsstufe (sehr hoch, hoch, mittel und tief) des Systems. Das soll heißen, dass die Testtiefe für ein Flugzeug (Kritikalitätsstufe = sehr hoch⁴) viel höher und genauer ist, als die Kompatibilität eines Bildschirms mit einem Graphiktreiber (Kritikalitätsstufe = tief⁵).

Anhand der Risikoanalyse und der Kritikalitätsstufe wird auch eine vereinbarte Testabdeckung und Testfallermittlungsverfahren erreicht. Im Fall vom Flugzeug wird eine Kombination von White und Blackbox-Methoden mit sehr hoher Testtiefe ausgeführt. Dagegen im Fall vom Bildschirm wird intuitives Testen mit geringer Testtiefe angewendet.[?]

3.1.3 Abhängigkeitsregeln

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

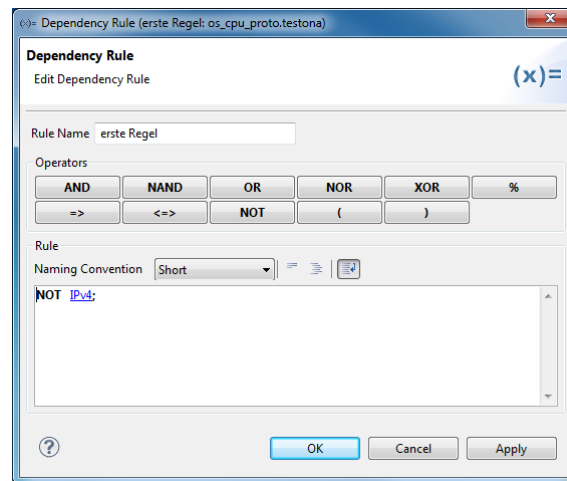


Abbildung 3.2: Abhängigkeitsregeln Bearbeitung

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 diese Operatoren werden wie folgt Abhängigkeitsregeln definiert:

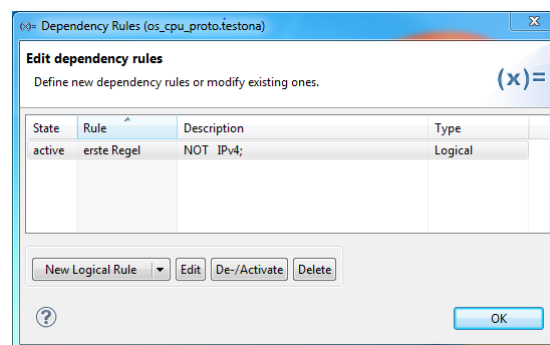


Abbildung 3.3: Abhängigkeitsregeln Übersicht

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

In diesen 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, wo die Spannung sich 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:

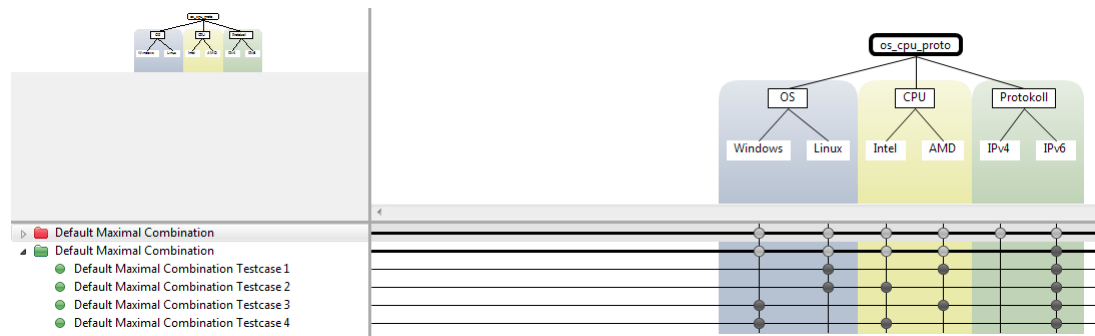


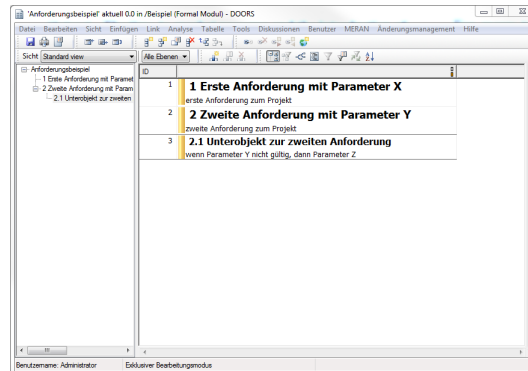
Abbildung 3.4: Testfälle mit Anwendung der Abhängigkeitsregeln aus 3.2 und 3.4

Lichtschalter1 XOR Lichtschalter2
NOT > 5V

Mit diesen Regeln, werden nur Testfälle betrachtet, wo 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 indicates '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 Changemanagementwerkzeugen) dank der Unterstützung von RIF (Requirements Interchange Format). Durch die Skriptsprache „DXL“ (DOORS eXtention Language) erhält TESTONA zugriff auf die gespeicherten Anforderungen in DOORS (durch die Ausführung von Skripts). [?] [?]

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. Um es besser zu unterscheiden soll folgendes Beispiel betrachtet werden. Ein Wagen kann in verschiedene Modelle gebaut werden: Coupé, Limousine, Cabrio. Alle sind Wagen und haben alle grobe Eigenschaften eines Wagens (4 Räder, Personenkraftwagen, etc), aber unterscheiden sich durch die Anzahl der Passagiere oder die Größe des Wagens. Diese sind Varianten eines Wagens der in verschiedene Modelle angeboten wird. [?]

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 sich ändert (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 Leistung des Wagens auswirkt (somit werden die Tests oder Testabläufe nicht beeinflusst). [?]

Das Variantenmanagement wird im Falle 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 ein Feature und eine Variante. Da wie bereits erwähnt, sind Varianten Änderungen eines Produktes, wo es mehr Gemeinsamkeiten als Unterschiede gibt. Mit dem Eintragen der Änderungen in TESTONA kann der Tester viel Arbeit sparen, da neue Test und Testabläufe besser erkannt werden.

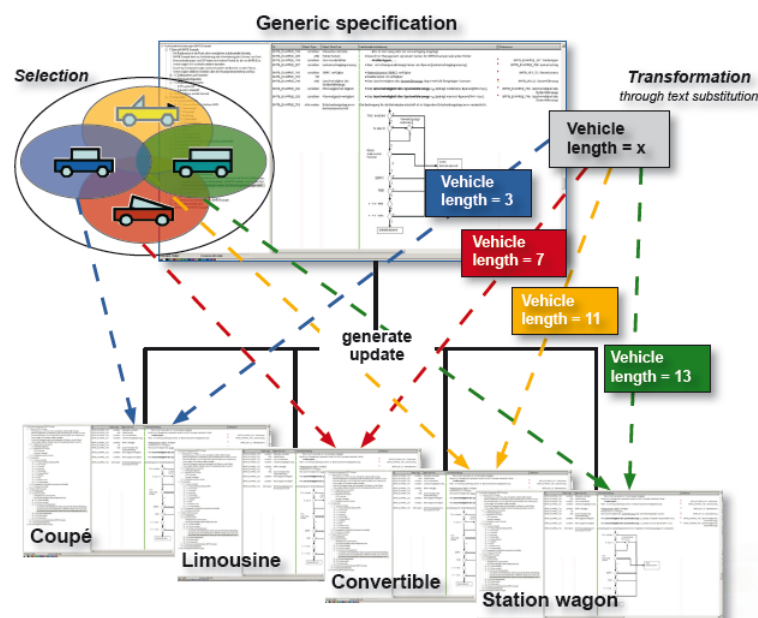


Abbildung 3.6: Betrachtung von Varianten eines Wagens von der generischen Testspezifikation zur variantenspezifische Testspezifikation aus [?]

3.4 Entwicklungsumgebung und Programmiersprache

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 Art. Ursprünglich war Eclipse als integrierte Entwicklungsumgebung für Java benutzt, aber dank seiner Bedienbarkeit und Erweiterung ist mittlerweile für die Entwicklung in verschiedener Programmiersprachen bekannt (C/C++ und PHP, unter anderen). Die am 25 Juni 2014 veröffentlichte Version „Luna“ (Eclipse 4.4) ist die 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[?] [?]. 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 Programmieren 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 grafischen Oberflächen
- JFace: User Interface Toolkit für komplexeren Widgets

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

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, wo 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 wo die nötige Plug-ins für die jeweilige Version geladen werden.

Ein Plug-in besteht in der Regel aus folgende 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 andere implementierte 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:** beinhaltet die Einstellungen für das Compilieren des Plug-in. Es kann auch über die META-INF Datei bearbeitet werden.
- **plugin.xml:** hier werden die nötige 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 seit 2001 von IBM Programmierbibliothek 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 Anwendung 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 behaltet ein konstantes Programmiermodell in alle Plattformen. [?]

SWT beinhaltet sehr viele komplexe Eigenschaften, aber um eine robuste und benutzbare Anwendung zur Programmieren sind nur die Grundkenntnissen nötig. Eine typische SWT Anwendung hat folgende Struktur:

- Ein *Display* deklarieren, dieser Repräsentiert die SWT Modus
- Erstellen eines *Shell*, welche als Hauptfenster dient
- Erzeugung eines Widgets
- Initialisierung der Widgetparameter
- Öffnen des Fensters

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

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

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

Listing 3.1: Beispiel einer SWT Anwendung

3.4.4 JFace

JFace ist ein User Interface Toolkit, dass 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 GUI-Elementen 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 Hilfe an den Benutzer anbieten für richtige Inhaltsauswahl bei Dialoge oder Formulare

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

Kapitel 4

Lösungsansätze

4.1 Parameterspeicherung und Visualisierung

Parameter aus DOORS in TESTONA importieren und speichern
Je nach ausgewählte Variante der richtige Parameterwert anzeigen

4.2 Testgültigkeit

Innerhalb einer Variante auf Testfallduplikate überprüfen

4.3 Optimierungskriterien

Testablauf optimieren in Betrachtung auf die Varianten und mögliche Testduplikate

Kapitel 5

Systementwurf

5.1 Variantenmanagement und Parameter

Erläuterung der Lösungen zu

5.2 Testfallgenerierung und Optimierung

Kapitel 6

Evaluierung

Als letzter Schritt für die Beendigung der Entwicklung des Testprogrammes müssen Test durchgeführt werden und die Ergebnisse evaluiert werden.

6.1 Chances of Failure

4 Augenprinzip
Komplexere Bäume wurden nicht betrachtet

Kapitel 7

Zusammenfassung und Ausblick

Was war wirklich wichtig bei der Arbeit?
Wie sieht das Ergebnis aus?
Wie schätzen Sie das Ergebnis ein?
Gab es Randbedingungen, Ereignisse, die die Arbeit wesentlich beeinflußt haben?
Gibt es noch offene Probleme?
Wie könnten diese vermutlich gelöst werden?

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 [?]	11

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.
-