WIRTSCHAFTSUNIVERSITÄT WIEN

DIPLOMARBEIT

Titel der Diplomarbeit:
Ableitung funktionaler Testfälle aus dem Konzept am Beispiel von CMF und UML - Use Cases

Verfasserin/Verfasser: Ferdinand Nest

Matrikel-Nr.: H8951445

Studienrichtung: Handelswissenschaften

Beurteilerin/Beurteiler: Priv. Doz. Dr. Michael Hahsler

Ich versichere:

dass ich die Diplomarbeit selbstständig verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und mich auch sonst keiner unerlaubten Hilfe bedient habe.

dass ich dieses Diplomarbeitsthema bisher weder im In- noch im Ausland (einer Beurteilerin/ einem Beurteiler zur Begutachtung) in irgendeiner Form als Prüfungsarbeit vorgelegt habe.

dass diese Arbeit mit der vom Begutachter beurteilten Arbeit übereinstimmt.

Datum _____ Unterschrift _____

Abstrakt

Softwaretest als Teil des Softwareprozesses wird oft unterschätzt und entweder nicht umfangreich genug, unstrukturiert oder zu spät durchgeführt. Diese Studienarbeit möchte auf die Notwendigkeit von organisierten Tests hinweisen, Techniken zur Ableitung von Testfällen von Softwarespezifikationen darstellen, erfolgreiche Teststrategien aus der Praxis beschreiben und diese Schritte mit Vorlagen, Mustern und selbst entwickelten Toolprogrammen erleichtern.

Abstract

Softwaretesting often is underestimated during the software development process. Either it lacks structure, comprehension or the 'right moment' to start with. This paper tries to point out the necessarity of organised Softwaretesting, it tries to provide certain techniques to derive Testcases from specifications, it describes succsessful practices executed in daily business and provides Patterns, Templates and Tools which are more than helpful in the process.

Danksagung

Ich danke primär der Firma SDS – Software Daten Service – T-Systems für die Erlaubnis Firmendaten und –material zu verwenden, ich danke Hr. Professor Harry M. Sneed, der mich zu dieser Studienarbeit inspirierte, dem Institut Janko und Hr. Dr. Hahsler, Hr. Martin Adamiker für die Umsetzung und rege Kooperation bei der Entwicklung des Programms ‚Workflowvalidator', Hr. Harald Liebetegger für die Umsetzung des Programms ‚Konzeptpattern'. Und natürlich danke ich weiters noch allen Freunden, Bekannten und Verwandten, die mich während der Erstellung dieser Studienarbeit unterstützt haben.

Inhaltsverzeichnis:

1. Einleitung

***Testen**... bedeutet, eine gegebene Software zu einem gegebenen Zeitpunkt zu nehmen und überzeugend zu zeigen, dass sie den gegebenen Anforderungen entspricht* [EBTE06].

Testen ist die Ausführung eines Programms bzw. eines Programmsystems mit dem Ziel, Fehler zu finden.

Testen ist die Ausführung eines Programms mit dem Ziel, seine Funktion zu bestätigen.

Testen ist die Ausführung eines Programms mit dem Ziel, seine Zuverlässigkeit zu messen [EBTO06].

„Testen“ als Tätigkeit ist, wie diese Statements ausdrücken, nicht gleich Testen. Es gibt unterschiedliche Philosophien oder Ansätze, ob Fehler gefunden oder die Funktionalitäten des Programms positiv nachgewiesen werden sollen. Ebenso können je nach der Problemstellung, der Phase im Entwicklungsprozess oder dem zu testenden Objekt unterschiedliche Testtechniken angewendet werden. Softwaretest als Teil des Softwareprozesses wird oft unterschätzt und entweder nicht umfangreich genug, unstrukturiert oder zu spät durchgeführt.

Das Erstellen von Testfällen, die geeignet sind, spezifizierte Funktionen eines Software - Systems zu testen, war immer eine kostspielige Angelegenheit, abhängig von der Quelle, von der die Testfälle abgeleitet werden. Werden die Testfälle aus dem Source Code oder einer Design Sprache wie z.B. SDL oder UML hergeleitet, kann die Generierung durchaus automatisch erfolgen. Robert Poston und andere haben sich mit solchen Tools befasst. Struktogramme, Zustandsübergangsdiagramme und Aktivitätsdiagramme sind

alle dafür geeignet analysiert und zu Testfällen verarbeitet zu werden. Das Selbe trifft auf formale Spezifikationssprachen wie OCL zu, die auf Basis von Regelbeschreibungen den Sollzustand einer Software darstellen. Je größer der Grad des formalen Inhalts eines Dokuments ist, desto leichter ist es Testfälle aus diesem abzuleiten. Die Testfälle stellen in diesem Sinne eigentlich die Umkehrung der Prädikatenlogik der Spezifizierungssprache dar. Mittlerweile ist das Wissen um dieses Thema sehr angewachsen.

Sind formale Spezifizierungen vorhanden, sollten auch Testfälle aus diesen generiert werden. Die Literatur zum Thema Test bietet viele Methoden hierzu, angefangen bei den Überleitungstabellen (Transition tables) von D. Harel [HARE87]. Jedoch fehlt es oft an formalen Beschreibungen, vor allem in vielen wirtschaftlichen Systemen. Sogar solch große Business Software Programme wie SAP und Oracle Financials verfügen nur über informale Dokumente, die ihre Funktionalitäten beschreiben. Von einigen Ausnahmen abgesehen, gibt es eigentlich sehr wenige Wirtschaftsapplikationen, die formale Spezifikationen vorweisen können. Der Großteil wird mittels einer natürlichen Sprache beschrieben. Sogar die zurzeit in Mode gekommenen Use-Cases, werden von Analysten in Prosa formuliert. Das ist zurückzuführen auf die Tatsache, dass die Wirtschaftsfachleute mit formalen Methodiken selten in Berührung kommen (die meisten wissen nicht einmal, was Prädikatenlogik eigentlich ist), und aus diesem Grunde ihre Anforderungen in ihrer Muttersprache definieren. Darüber hinaus ist dies oft die einzige Sprache, die die Anwender verstehen. Anwender von Wirtschaftssoftware wehren sich oftmals gegen das Erlernen von formalen Methoden und bestehen darauf, dass die bestehende Funktionalität eines solchen Systems in der ihnen verständlichen Terminologie dargestellt wird., Das führt dazu, dass Unterlagen, die die geforderten Spezifikationen eines Systems beschreiben, typischerweise in Form von strukturierten Texten einer natürlichen Sprache entworfen werden.

Sogar der ANSI-Standard-830 für Anforderungsspezifikationen unterstützt solche Praktiken [PISC93]. Die Anforderungen werden analog einem genormten Schema spezifiziert, aber die beschriebenen Inhalte sind Texte. Das stellt einen Vorteil für die

Anwender dar, bringt aber Probleme für die Tester. Diese sind verpflichtet diese informalen Texte in formale Testfälle mit Pre- und Post Zuständen überzuleiten. Anders gesagt, die Tester fungieren als Interpreter der Anwender, indem sie die ‚Anforderungsprosa' in präzise Prädikatenlogik transformieren. Das stellt eine sehr arbeitsintensive Aufgabe dar, die viel Zeit und Fachwissen des Testers verlangt. Es gibt viele Kurse, die genau darauf abzielen, die Tester dahingehend zu schulen, diese Lücke zwischen informalen Spezifikationen einerseits und formalen Testfällen andererseits zu schließen [BEIZ83]. Diese Übersetzungstätigkeit ist ähnlich der Arbeit und dem Aufwand, die ein Software Entwickler ausführt, der informal definierte Use – Cases in formale UML Diagramme oder direkt in eine formale Programmiersprache überleitet.

Automatisierte Testfälle müssen in formalen Termini ausgedrückt werden, damit man X als Ergebnis der Variablen Y und Z der Funktion foo erhält

$$X = foo(Y, Z)$$

Weiters müssen die Wertebereiche von Y und Z und ihre Relation zu X definiert werden, z.B.:

Wenn $Y > 1 \ \& \ Z < 1$

Dann $X = 0;$

Die exakte Definition der Relation zwischen Inputs und Outputs ist die Grundbedingung der Spezifikation von Testfällen. Die Ableitung dieser Beziehungen von informalen Anforderungen ist eine aufwändige Arbeit und wurde u.a. von Robert Poston [POST96] und H. Buwalda [BUWA98] beschrieben.

Der erforderliche Einsatz, der benötigt wird, diese Aufgabe zu erfüllen, ist zur Größe und zur Zahl an Anforderungstexten, d.h. Use-cases, die in Testfälle umgewandelt werden müssen, proportional. Wenn die Texte kurz und einfach sind, und es nicht zu viele von ihnen gibt, kann die Aufgabe manuell durchgeführt werden. Aber wenn die Texte lang und umfangreich sind, wird die Aufgabe sehr schnell kostspielig. Komplexe

wirtschaftliche Systemprogramme können Hunderte Use-cases haben, und jeder von ihnen kann sich über viele Textseiten erstrecken und die Aufgabe der formalen Testfallspezifikation fast zur Unmöglichkeit gestalten. Der Mangel an formalen Testfällen macht eine automatisierte Testfalldurchführung unmöglich, was wiederum dazu führt, dass eine hohe Deckung an Funktionsprüfungen nicht erreicht werden kann. Folglich ergibt sich eine definitive Notwendigkeit, die Testfallerzeugung zu automatisieren.

Bevor der Aufbau und der Inhalt dieser Studienarbeit noch kurz umrissen wird, soll grundlegendes Testvokabular erklärt werden, nämlich der Testfall, der Testlauf und das Testobjekt.

Als **Testfall** bezeichnet man die Anforderung an Testdaten zur Verfolgung eines einzelnen Testzieles. Jeder Testfall sollte ein genaues Testobjekt aufweisen können. Die aus Entscheidungstabellen abgeleiteten Testfälle bestehen jeweils aus Preconditions und Postconditions. Testfälle werden in weiterer Folge auch mit dem englischen Wort *Testcases* bezeichnet.

Ein **Testlauf** ist die Zusammenfassung von Testfällen zu einem Testfallset, dieses verfolgt ein (Teil-) Projektziel, und sollte aus einer überschaubaren Anzahl von Testfällen bestehen. Testläufe können schon auf die Testumgebung zugeschnitten sein und werden auch *Testsuites* genannt.

Das **Testobjekt** ist das Objekt das getestet werden soll. Das können Rechenergebnisse, Phasen oder Retourncodes sein. Das Testobjekt sollte sehr genau definiert sein, um nicht nur präzise Ergebnisse liefern zu können, sondern auch um das Fehlertracking zu vereinfachen und die Vergleichbarkeit zu einem späteren Zeitpunkt gewährleisten zu können. Je nach Testobjekt sind unterschiedliche Teststrategien anwendbar.

Diese Studienarbeit soll eine Anleitung zur Erstellung von Testfällen geben, sie möchte auf die Notwendigkeit von organisierten Tests hinweisen, Techniken zur Ableitung von Testfällen von Softwarespezifikationen darstellen, erfolgreiche Teststrategien aus der Praxis beschreiben und diese Schritte mit *Patterns* (Mustern), *Templates* (Vorlagen) und selbst entwickelten Tools, im konkreten Parsingprogrammen, erleichtern. Praktische Testfälle sollen am Beispiel von Konzepten der Firma SDS und ihrem Produkt GEOS erarbeitet werden. SDS (Software Daten Service) ist ein Wiener Software Unternehmen und Tochter von T-Systems und beschäftigt sich hauptsächlich mit der Entwicklung einer Finanzsoftware für die Prozessabwicklung am Wertpapiermarkt.

Im ersten Teil wurden einige für diese Arbeit grundlegende Begriffe definiert und gegeneinander abgegrenzt. Dies erscheint notwendig, da sich für diese Begriffe in der Literatur verschiedene Definitionen finden und sie in der Arbeit konstant Verwendung finden.

Im darauf folgenden Kapitel werden die Anforderungen an den Test analysiert und in ihren Anwendungsstufen erklärt.

Weiters werden die grundlegenden Arten des Tests beschrieben, aufbauend mit dem Unittest über den Integrationstest zum Systemtest, dem Funktionstest und schließlich zum Risikotest und Regressionstest.

In der Folge wird auf den Systemtest und die damit verbundenen Problematiken eingegangen. Es wird in diesem Zusammenhang auf die unbedingte Notwendigkeit des Funktionstests hingewiesen, der die absolute Grundvoraussetzung für jedes weitere Testvorgehen darstellt. Darauf folgt eine theoretische Gliederung der Arten von Funktionstestausprägungen.

Kapitel 3 beschäftigt sich mit der Methodik zur Ableitung von Testfällen. Begonnen wird mit einer Beschreibung des von der Firma SDS verwendeten Konzepttools, dem CMF – Concept Management Facility und wie Testfälle damit umgesetzt werden können. Darauf folgt eine genaue Erklärung der Geschäftsregeln, womit dann im weiteren Verlauf auf die

Darstellung der vorliegenden Fachlogik eingegangen wird.

In Kapitel 4 folgt eine Beschreibung von weiteren Quellen für die Testfallermittlung. Die UML wird in diesem Teil anhand von Use Cases und Aktivitätsdiagrammen beschrieben und dem CMF gegenübergestellt.

Im darauf folgenden Kapitel wird das CTF, die Concept Test Facility, das Testfallwerkzeug der SDS beschrieben, das sowohl über Testfallerfassungs-, Planungs- und Durchführungsmöglichkeiten verfügt.

Kapitel 6 schildert den Test der SDS in der Praxis mit Beispielen und einigen wenigen Kennzahlen.

Kapitel 7 beschreibt die Erfahrungen mit Parsingprogrammen, die für diese Arbeit erstellt worden sind.

Im letzen Kapitel werden die gesammelten Erfahrungen bewertet und Ausblicke für weiteres Vorgehen vorgenommen.

Der Anhang enthält neben dem Literaturverzeichnis und dem Abbildungsverzeichnis noch einen Glossar mit Abkürzungen, die in dieser Studienarbeit verwendet werden.

2. Softwaretest

Dieses Kapitel stellt die grundlegende Frage: „Warum testen wir?“

Bei jedem Softwareprogramm ist zu entscheiden, wie intensiv und wie umfangreich es zu testen ist. Diese Entscheidung muss in Abhängigkeit zum erwarteten Risiko bei fehlerhaftem Verhalten des Programms getroffen werden. Da ein vollständiger Test nicht möglich ist, ist entscheidend, wie die beschränkten Testressourcen eingesetzt werden. Um zu einem zufrieden stellenden Ergebnis zu kommen, müssen die Tests strukturiert und systematisch durchgeführt werden. Nur so ist es mit angemessenem Aufwand möglich, viele Fehlerwirkungen nachzuweisen und unnötige Tests, die zu keinen neuen Erkenntnissen führen, zu vermeiden.

Da es eine Reihe von unterschiedlichen Varianten und Methoden zum Testen von Software gibt, müssen viele Verfahren, die alle verschiedene Aspekte des Testobjekts prüfen, berücksichtigt werden. So steht bei kontrollflussbasierten Testverfahren der Ablauf eines Programms im Mittelpunkt der Untersuchungen, wohingegen bei datenflussbasierten Testverfahren der Gebrauch der Daten der Untersuchungsgegenstand ist. Kein Testverfahren ist für alle Aspekte gleich gut geeignet, daher ist eine Kombination von unterschiedlichen Testverfahren immer erforderlich, um Fehlerwirkungen aufzudecken, die unterschiedliche Ursachen haben.

2.1 Softwareentwicklungsprozess

Der Test ist Teil des Entwicklungsprozesses jeder Softwareentwicklung. Es gibt viele Verfahren, wie z.B. das Wasserfallmodell [FRIC95] oder das V-Modell [THAL02]. Unter dem Aspekt des ‚*Test Requirements Engineering*’ muss bei den Anforderungen auf den Test insbesondere zwischen Verifikation und Validation unterschieden werden, da diese oftmals verwechselt werden.

2.1.1 Software-Verifikation

Verifikation ist die Überprüfung einer Software am Ende einer Entwicklungsphase gegen vorgegebene Anforderungen, einen Standard oder die Ergebnisse der vorhergehenden Phase [THAL02].

D.h., die Verifikation hinterfragt, ob das System mit der Beschreibung bzw. mit der Spezifikation übereinstimmt oder ob das System so funktioniert, wie es in der Beschreibung behauptet wird.

2.1.2 System - Validation

Bei Validation handelt es sich um die Prüfung der Frage, ob ein Softwareprodukt seinen Anforderungen gerecht wird [THAL02]. Sneed [SNEE02] hat dazu gesagt: „Das System funktioniert in der Zielumgebung, oder noch simpler, es macht zumindest irgendetwas."

2.1.3 Das Wasserfall-Modell in der Softwareentwicklung

Das Wasserfall-Modell [FRIC95] ist das ursprünglichste und bekannteste Modell der Softwareentwicklung. Erst wenn eine Entwicklungsphase abgeschlossen ist, wird mit der nächstfolgenden begonnen. Es gibt nur zwischen angrenzenden Stufen Rückkopplungsschleifen, die eine gegebenenfalls notwendige Überarbeitung in der vorherigen Stufe ermöglichen. Der entscheidende Nachteil ist, dass das Testen als einmalige Aktion am Projektende vor der Inbetriebnahme aufgefasst ist [SPLI03]. Die Abbildung 2.1 stellt diesen Prozess dar.

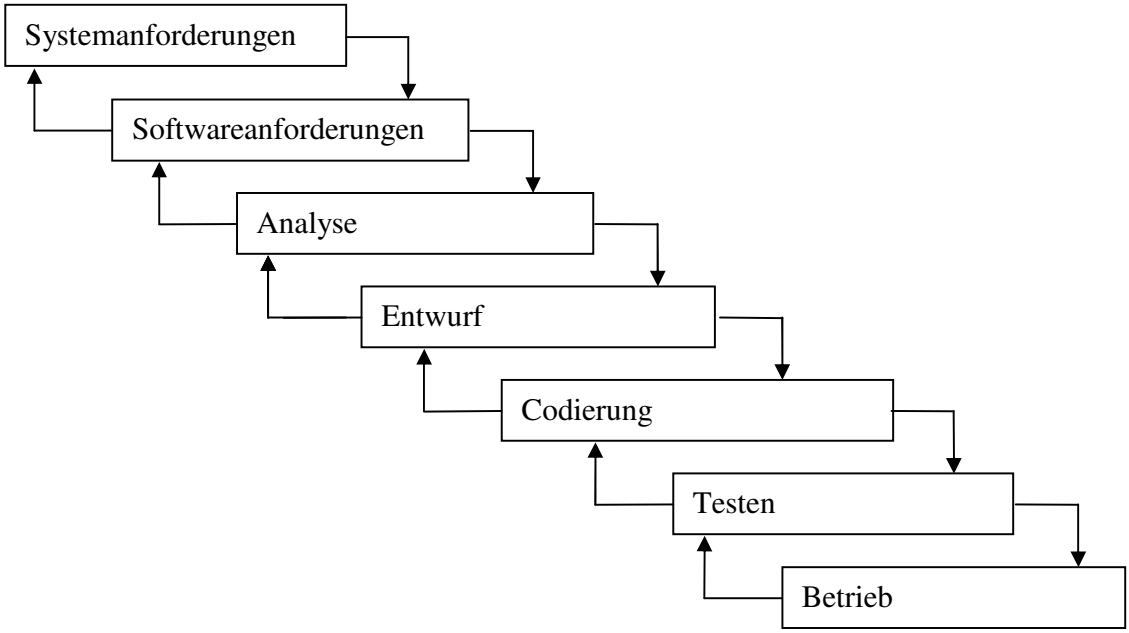


Abbildung 2.1: Wasserfall-Modell

2.1.4 Das V- Modell in der Softwareentwicklung

Das V-Modell [THAL02] ist auf der Basis der Grundidee entwickelt worden, dass Entwicklungsarbeiten und Testarbeiten zueinander korrespondierende, gleichberechtigte Tätigkeiten sind. Bildlich dargestellt wird dies durch die zwei Äste eines ‚V's, das die dem Modell innewohnende Verifikation und Validation, wie Abbildung 2.2 zeigt, noch versinnbildlicht [SPLI03]. Dies stellt einen Gegensatz zum Wasserfallmodell dar, bei dem die Testarbeiten nach den Entwicklungstätigkeiten gestartet werden.

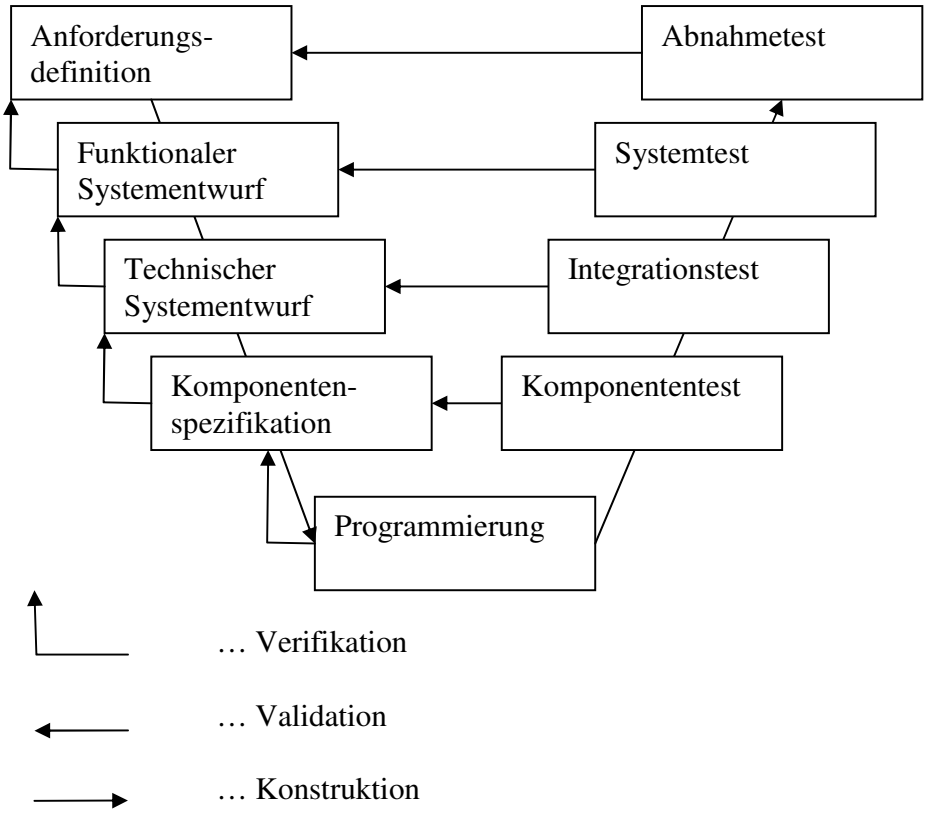


Abbildung 2.2: V-Modell

Der linke Ast steht für die immer detaillierter werdenden Entwicklungsschritte, in deren Verlauf das vom Kunden gewünschte System schrittweise entworfen und schließlich programmiert wird. Der rechte Ast steht für Integrations- und Testarbeiten in deren Verlauf Programmbausteine sukzessive zu größeren Teilsystemen zusammengesetzt (integriert) und jeweils auf die richtige Funktion geprüft werden [SPLI03].

Grob kann gesagt werden, „*bei Verifikation wird überprüft, ob das Produkt richtig ist, bei Validation prüfen wir, ob wir das richtige Produkt erstellt haben“* [THAL02].

Verifikation ist in erster Linie auf Einzelheiten (Review eines Grobkonzepts, Überprüfung der Usecases oder Durchsicht des Pflichtenheftes) ausgerichtet, während wir bei Validation das Produkt in seiner Umgebung sehen. Verifikation ist eine Methode eher

für den Entwickler geeignet, weil er detaillierte Produktkenntnisse vorweisen kann. Bei der Validation stellt sich der Test auf den Standpunkt des Kunden und der Benutzer.

2.1.5 Testen als Vertrauensmittel

Da in den meisten Fällen Software für zahlende Kunden produziert wird, hat der Test die wichtige Aufgabe, den Kontakt zu den Kunden zu intensivieren. Er soll insbesondere folgende Punkte erfüllen:

Vertrauen des Kunden gewinnen

Der Benutzer weiß, dass er durch engagierten, fundierten und qualitativen Test seitens des Produzenten gut aufgehoben sein kann. Das gewinnt das Vertrauen des Kunden und ist insbesondere bei Softwareverträgen mit längerer Laufzeit wünschenswert, oder wenn der Entwickler nachhaltig um gute Reputation bemüht ist. Des Weiteren ist es besonders bei Acquisition von Kunden von Vorteil, diesem eine funktionierende Testabteilung vorführen zu können, da der potentielle Käufer seinen eigenen Testaufwand niedriger bemessen kann.

Nachweis des Tests

Regelmäßige Testdurchführungen sind vor allem bei Weiterentwicklungen der Programme besonders wichtig. Man muss dem Benutzer nachweisen können, dass man das Programm umfangreich getestet hat. Jede Verbesserung oder Veränderung eines Programms kann neue Fehler beinhalten, und muss somit getestet werden. Verfügt man über geeignete Protokollmaßnahmen, kann dem Kunden der Beweis erbracht werden, dass das Programm mit den laufenden Änderungen funktioniert und (oftmals sehr wichtig) zumindest nicht schlechter geworden ist.

Persönlichen Kontakt

Kundenbetreuung ist insbesondere bei komplizierter Software von Bedeutung. Vor allem bei renommierten Testabteilungen (die bereits das Vertrauen des Kunden gewonnen haben), wird oftmals der Rat des Testingeneurs eingeholt, ob kundenseitig ein

Benutzerfehler begangen wird, oder ob das Programm tatsächlich fehlerhaft arbeitet, sollten Störfälle beim Endbenutzer auftreten.

Metriken

Der Nachweis der Qualität des Programms kann mit dem Einsatz von Metriken erbracht werden, die Kapitel Metriken und Fehlerstatistiken geben Beispiele für solche Kennzahlen.

2.2 Fehlerfindung

Das Grundprinzip sollte das Finden der Fehler, Mängel oder Programmschwächen sein. Die Fehler müssen beseitigt werden, und langfristig muss das Risiko reduziert werden können.

Risikoreduktion ist hierbei durch den kombinierten Einsatz von Testtechniken möglich. Beispielsweise müssen Fehler, die im Unittest (Entwicklertest) nicht auftreten, durch weitere Tests gefunden werden.

Hierbei können Fehlerprofile oder die Verwendung von Metriken die notwendige Indikation für den Einsatz von Testressourcen geben. Des Weiteren kann das Programm in risikoreiche und risikoärmere Teile unterschieden und demnach getestet werden.

2.3 Grundlagen des Testprozesses

Der Inhalt der Entwicklungsaufgabe „Testen“ muss in kleinere Arbeitsschritte gegliedert werden, nämlich in

- Testplanung,
- Testspezifikation,
- Testdurchführung,
- Testprotokollierung und
- Testauswertung.

Diese Teilaufgaben bilden einen fundamentalen Testprozess und werden im Folgenden näher beschrieben.

2.3.1 Testplanung

Eine strukturierte Abwicklung einer so umfangreichen Aufgabe wie dem Testen kann nicht planlos erfolgen. Mit der Planung des Testprozesses wird zum Beginn des Softwareentwicklungsprozesses begonnen. Wie bei jeder Planung ist während des Projektfortschritts regelmäßig eine Kontrolle der bisherigen Planung, eine Aktualisierung und Anpassung vorzunehmen.

Für den Testprozess sind die benötigten Ressourcen einzuplanen und die entsprechenden Festlegungen in einem Testkonzept festzuhalten. Es kann auch ein Testmanagement aufgebaut werden, das den Testprozess, die Testinfrastruktur und die Testware verwaltet. Eine Kernaufgabe der Planung ist die Bestimmung der Teststrategie. Da ein vollständiger Test nicht möglich ist, müssen Prioritäten anhand einer Risikoschätzung gesetzt werden. Kapitel Risikotest beschäftigt sich näher mit diesem Thema. Ziel der Teststrategie ist die optimale Verteilung der Tests auf die richtigen Stellen des Softwaresystems, um eine so genannte Testfallexplosion zu vermeiden [SPLI03].

2.3.2 Testspezifikation

Die im Testplan festgelegte Teststrategie bestimmt, welche Testmethoden zum Einsatz kommen sollen. In der Testspezifikation werden dann unter Anwendung dieser Testmethoden die entsprechenden Testfälle spezifiziert.

Neben den üblichen Systemtestfällen, die als Grundlage für die Testfälle die Systemoberfläche und externe Schnittstellen verwenden, sollte die Testfallspezifikation auch Integrationstestfälle (Testgrundlage: interne Schnittstellen), und Modultestfälle enthalten. Dies sind gängige Ansätze der prozeduralen Programmierung.

In der objektorientierten Programmierung müssten Vererbungen, etc. noch berücksichtigt werden. Das Ergebnis wäre somit eine Hierarchie von Testfällen. Die Basistestfälle

bilden die Klassentestfälle, aus denen Modultestfälle abgeleitet werden. Die Integrationstestfälle erben diese und ergänzen sie um die Komponententestfälle. Die Systemtestfälle umfassen alle bisherigen Testfälle, plus die Testfälle, die sich aus dem Nutzungsprofil des Systems ergeben, plus jene Testfälle, die für den Nachweis des Systemverhaltens notwendig sind (Robustheit, …) [SNWI02].

Die Spezifikation erfolgt in zwei Schritten. Zuerst sind logische Testfälle zu definieren. Danach können die logischen Testfälle konkretisiert werden, d.h. die tatsächlichen Eingabewerte festgelegt werden. Auch der umgekehrte Weg ist möglich, wenn ein Testobjekt unzureichend spezifiziert ist [SPLI03].

2.3.3 Testdurchführung

Nach Übergabe der zu testenden Programmteile durch die Programmierung an die Testabteilung kann mit der Testdurchführung begonnen werden. Zunächst erfolgt eine Prüfung auf Vollständigkeit durch Installations-, Start- und Ablauffähigkeit. Als grundsätzliche Teststrategie empfiehlt sich, die Testdurchführung mit der Überprüfung der Hauptfunktionen des Testobjekts zu beginnen. Sollten sich hier bereits Fehlerwirkungen oder Abweichungen von den Sollergebnissen zeigen, ist ein tiefes Einsteigen in das Testen wenig sinnvoll, da vorab die fehlerhaften Hauptfunktionen korrigiert werden sollten [SPLI03].

2.3.4 Testprotokollierung

Die Durchführung der Tests ist exakt und vollständig zu protokollieren. Zum einen muss anhand dieser Testprotokolle die Testdurchführung auch für nicht direkt beteiligte Personen, z.B. für den Kunden, nachvollziehbar sein. Zum anderen muss nachweisbar sein, ob die geplante Teststrategie tatsächlich umgesetzt wurde. Aus dem Testprotokoll muss hervorgehen, welche Teile wann, von wem, wie intensiv und mit welchem Ergebnis getestet wurden.

Zu jeder Testdurchführung gehört eine ganze Reihe von Informationen und Dokumenten:

Testrahmen, Testprotokoll, Eingabedateien usw. Die einem Testfall oder Testlauf zugehörigen Daten sind so zu verwalten, dass zu einem späteren Zeitpunkt eine Wiederholung der Tests mit den gleichen Daten und Randbedingungen möglich ist. Hierfür ist ein Konfigurationsmanagement der Testware vorzunehmen [SPLI03].

Testprotokolle sind in erster Linie Testablaufprotokolle, die die Reihenfolge der getesteten Funktionen dokumentieren, die Zustandsprotokolle, die den Zustand der erzeugten Objekte, Daten und Schnittstellen dokumentieren, und dies zu verschiedenen Zeitpunkten (nach Konstruktion, nach jeder Veränderung, und auch unmittelbar vor der Destruktion). Die Überdeckungsprotokolle geben Information darüber, welche Zweige, Methoden, Operationsaufrufe und Objekte wie oft ausgeführt werden [SNWI02].

2.3.5 Testauswertung

Tritt bei der Testausführung ein Unterschied zwischen dem tatsächlichen und dem erwarteten Ergebnis auf, ist bei der Auswertung der Testprotokolle zu entscheiden, ob tatsächlich eine Fehlerwirkung vorliegt, oder ob möglicherweise die Testspezifikation, die Testinfrastruktur oder der Testfall bzw. der Testlauf Fehler aufweisen.

Anhand der Fehlerklassen ist zu entscheiden, mit welcher Priorität ein Fehler zu beheben ist. Nach der Korrektur des Fehlerzustands ist zu prüfen, ob die Fehlerwirkung beseitigt ist und bei der Beseitigung keine weiteren Fehlerzustände hinzugekommen sind (siehe Regressionstest). Obwohl es günstiger wäre, Fehlerzustände einzeln zu korrigieren und erneut zu testen, um unbeabsichtigte gegenseitige Beeinflussungen der Änderungen zu vermeiden, allerdings ist dies in der Praxis kaum durchführbar.

Es ist in weiterer Folge zu entscheiden, ob das im Testkonzept festgelegte Kriterium zur Beendigung der Tests erfüllt ist. Für jede verwendete Testmethode ist unter Berücksichtigung des Risikos ein angemessenes Endekriterium zu bestimmen.

Ist mindestens ein Testendekriterium nach Durchführung aller Tests nicht erfüllt, müssen weitere Tests zur Anwendung kommen. Dabei ist darauf zu achten, dass die zusätzlichen Tests zu einer weiteren Annäherung an das Endekriterium führen. Sonst wird nur

zusätzlicher Aufwand aber kein Fortschritt produziert [SPLI03].

2.3.6 Testendekriterien

Je nach Testmethode gibt es unterschiedliche Testendekriterien, eine recht gute allgemeine Definition liefert Hehn und nennt folgende Kriterien [HEHI99]:

- Eine definierte Testabdeckung ist erreicht.
- Eine definierte Restfehlerzahl wurde unterschritten.
- Pro (Test-) Zeiteinheit wird nur noch eine bestimmte Fehlerzahl gefunden.
- Der Ausliefertermin ist erreicht.
- Wenn ein Entwickler mehr als n (z.B. n=20) Fehler zu korrigieren hätte, sollte dies zu einem Testabbruch führen müssen; Fortsetzung erst, wenn der Entwickler weniger als m (z.B.: 10) Fehler zu bearbeiten hat.
- Die Zahl der in einer Zeiteinheit (z.B. 1 Woche) gefundenen Fehler ist geringer als die in dieser Zeiteinheit abgeschlossenen Fehlerbearbeitungen.

Natürlich können jeder Testmethode spezielle Testendekriterien zugeordnet werden, beispielhaft sollen Binders Testendekriterien für sein Modell der *Category Partition Method*, das selbst relativ allgemein gehalten ist, beschrieben werden [BIND05]:

- Jede Kombination von Wahlmöglichkeiten muss einmal getestet werden.
- Jede *Exception* (Programmausnahmesituation, Fehler) muss einmal getestet werden.
- Die Testsuite sollte jeden Pfad oder Zweig mindestens einmal durchlaufen, dies wird *Method Scope Branch Coverage* genannt.

2.3.6.1 Metriken

Metriken sind gängige Instrumente, die mittels Kennzahlen versuchen, den Softwareprozess kontrollierter ablaufen zu lassen. Sie können selbstverständlich für die **Testüberwachung** eingesetzt werden. Nutzvolle Maße für die Verfolgung des Testprozesses sind nach Gerke beispielsweise [GERK02]:

- Testfallorientierte Metriken, z.B. Anzahl der Testläufe, Anzahl der erfolgreichen Tests, Anzahl der Testwiederholungen,
- Fehlerbasierte Metriken, z.B. Anzahl der Fehlermeldungen,
- Testobjektbasierte Metriken, z.B. Codeüberdeckung.

Gründe, die für den Einsatz von Metriken sprechen, sind [CRJA02]:

- Identifikation von risikoreichen Bereichen, die mehr Testaufwand benötigen,
- Identifikation von Schulungsbedarf,
- Identifikation von Prozessverbesserungsmöglichkeiten,
- Statusverfolgung und –kontrolle (Control/Tracking Status),
- Grundlage für Schätzungen,
- Rechtfertigung für Budget, Tools und Schulungen,
- Indikator, wann welche Aktionen gesetzt werden müssen/können.

Diese Maßzahlen können während des Testprozesses Indikationen dafür sein, ob sich die Aufwände erhöhen, bis das Testendekriterium erreicht werden kann.

Gewisse **Grundregeln** für das Arbeiten und den Einsatz von Metriken sollten eingehalten werden [CRJA02]. Bei Verwendung von Metriken soll prinzipiell sorgsam umgegangen werden, da jede Statistik ‚falsch' eingesetzt werden kann. Deswegen sollten Metriken mit dem Testpersonal gemeinsam erarbeitet, mit weiteren Metriken überprüft und statistisch

bereinigt werden. Sie sollten weiters regelmäßig wieder revalidiert werden bezüglich der Nützlichkeit. Sie sollten vertraulich behandelt werden und leicht gesammelt, interpretiert und analysiert werden können.

2.3.6.2 Programmüberdeckung

Von den Überdeckungsmaßen ist die Codeüberdeckung die bekannteste und stellt die am häufigsten verwendeten Messzahlen für *Code-Coverage*-Untersuchungen bereit [THAL02]:

Bei der Anweisungsüberdeckung wird jede Anweisung mindestens einmal getestet, diese wird auch als C0-Überdeckung bezeichnet. Bei der Zweigüberdeckung wird jeder Zweig einmal getestet. Diese auch C1-Überdeckung genannte Metrik gilt als DAS minimale Testkriterium und findet laut Zeller im Schnitt 34% der Fehler [ZELL06]. Die Pfadüberdeckung stellt das mächtigste Kontrollwerkzeug der an der Programmstruktur orientierten Metriken dar, ist aber praktisch ohne Bedeutung, da sie in der Praxis nicht durchführbar ist, weil hier die Schleifen mitgetestet werden müssen.

2.3.7 Fehlerfindungsprotokoll

Werden Softwarefehler in weiterer Folge über den Testprozess gefunden, sollten sie unbedingt in einem Formular protokolliert werden.

Typischerweise beinhaltet das **Fehlermeldungsprotokoll**

- erwartete und aktuelle Ergebnisse,
- Testumgebung,
- Identifikation der getesteten Software,
- Name des Testers,
- Schwere,
- Umfang,

- Priorisierung und
- und andere Informationen mit Relevanz für das Reproduzieren und das Lokalisieren des potentiellen Fehlers.

Fehlermeldungen sollten von ihrem Auftreten über die verschiedenen Stadien bis zum Abschluss oder der Lösung verfolgt werden.

Die Fehlerklassen [SPLI03] werden nach Spillner in 5 Fehlergruppen eingeteilt. Von Klasse 1 (Systemabsturz, Datenverlust, Produktionsstillstand), 2 (Wesentliche Funktion nicht einsetzbar), 3 (Funktionale Abweichung und Einschränkung), 4 (Geringfügige Abweichung) bis zur letzten Fehlerklasse 5 (Schönheitsfehler, Rechtschreibfehler oder Layoutfehler).

2.4 Statischer und Dynamischer Test

Es kann zwischen statischen und dynamischen Tests unterschieden werden. Statische Tests befassen sich unter anderem mit strukturierten Gruppenprüfungen, statischen Analysen und strukturellen Analysen [GERK02].

Strukturierte Gruppenprüfungen beinhalten *Code Inspections*, *Reviews* und *Walkthroughs*. Statische Analysen sind Analysen ohne Programmausführung und werden beispielsweise von einem *Compiler* durchgeführt, und strukturelle Analysen beschäftigen sich u.a. mit Datenflussanalysen, Kontrollflussanalysen, Metriken und der so genannten zyklomatischen Zahl. Spillner befasst sich sehr ausführlich mit dem Thema des statischen Tests [SPLI03].

Hier sollen aber die dynamischen Tests im Vordergrund stehen, und diese lassen sich grob in Unittest, Integrationstest und Systemtest unterteilen.

2.4.1 Unittest

Der Modultest stellt die erste unterste Stufe des computergestützten Testens bei der Softwareentwicklung dar. Zunächst werden hierbei alle elementaren Bausteine als

Fundament eines Programmsystems getestet und dann erst ihre Integration. Es werden also alle Funktionen, die selbst keine anderen Programmbausteine benötigen, getestet, dann erst ihre Integration zu einem Modul. Die Minimalanforderung für den Test eines Softwaremoduls muss dabei sein, dass mindestens alle Programmzweige einmal durchlaufen werden und das Modul gemäß seiner Spezifikation reagiert. Man bezeichnet den Unittest auch als Modultest, Klassentest (bei OO-Programmtests) oder als *Whiteboxtest*, da das Programm wie durch einen Glaskasten betrachtet werden kann [THAL02].

2.4.2 Integrationstest

Nach dem Modultest folgt in der nächst höheren Stufe der Integrationstest, auch Komponententest genannt, welcher die Integration mehrerer, bereits getesteter Module zu einem Subsystem einem Test unterzieht (man bezeichnet dies auch als "inkrementelle Integration"), bis letztlich die Integration der Subsysteme, d.h. das Gesamtsystem, getestet werden kann. Der Integrationstest wird auch als Grayboxtest bezeichnet, da manche Teile des Programms bekannt sind und manche wieder nicht. [THAL02]

2.4.3 Systemtest

2.4.3.1 Systemtest

Ein Softwaresystem ist bekanntlich mehr als nur die Summe aller Einzelteile. Es ist die Summe aller Einzelteile plus das Produkt aller Beziehungen zwischen den Teilen und dem Produkt der Effekte, die durch das Zusammenwirken der Einzelteile auftreten.

Die Systeme werden im Systemtest als schwarze Kästen, in weiterer Folge als ‚*Black Box*' bezeichnet; sie werden betrachtet als fertig geschnürte Pakete, die von außen getestet werden. Insofern ist es, was die Testmethodik anbetrifft, gleich, ob es sich um ein prozedurales oder objektorientiertes System handelt. Sie werden beide über die externen Schnittstellen getestet. d.h. von der Benutzeroberfläche bzw. von den Importschnittstellen aus. Der Unterschied liegt in der Menge der Testfälle, die zu

bewältigen ist. Üblicherweise müssen bei Tests von objektorientierten Programmen [ERRI02] mehr Testfälle aufgrund der Hierarchien, Vererbungen und Kapselungen berücksichtigt werden. Darüber hinaus tragen auch die Verteilung der Komponenten und die Gestaltung der Oberflächen dazu bei [SNWI02].

Der Systemtest beinhaltet mindestens drei Testarten [SNWI02]:

- den Umgebungstest,
- den Funktionstest und
- den Performance- und Belastungstest.

Mit dem Umgebungstest wird erprobt, ob das Anwendungssystem zu der Zielumgebung passt. Im Performance- und Belastungstest wird geprüft, ob die Anwendung die qualitativen, nicht funktionalen Kriterien ausreichend befriedigt und mit dem Funktionstest wird nachgewiesen, dass das Anwendungssystem alle spezifizierten Funktionen erfüllt.

2.4.3.2 Funktionstest

Funktionales Testen (*Black-Box*-Testen) geschieht ohne Ansehen der inneren Strukturen und ähnelt somit der Anwendung des Produkts. Für sie muss die Testdatenerzeugung von Anfang des Projekts an bedacht werden. Funktionales Testen reicht für Systeme mit komplexen Strukturen (z.B. eigen entwickelter interner Bus) allein nicht aus. Man kann dann aber Teilsysteme definieren und diese auch funktional testen.

Jeder Test braucht etwas, wogegen getestet wird: Es muss entschieden werden können, ob ein Testergebnis richtig oder falsch ist. Dieser Test sollte in der Regel von einem unabhängigen Tester durchgeführt werden. Der Funktionstest soll demonstrieren, dass das Anwendungssystem alle vereinbarten fachlichen Funktionen mit einer ausreichenden Qualität ausführt. Das ist nicht unproblematisch. Man muss davon ausgehen, dass die Funktionalität explizit festgeschrieben wurde, sei es im Fachkonzept, der Systemspezifikation, oder im Benutzerhandbuch. Man bräuchte sich nur auf das

entsprechende Dokument beziehen und die dort beschriebenen Funktionen zu testen. Dies setzt voraus, dass die gesamte Funktionalität eines Programms niedergeschrieben worden ist. Abbildung 2.3 soll demonstrieren, wie das Konzept umgesetzt werden muss, um ‚testbar' zu werden, es müssen Vorbedingungen definiert und die zu testende Funktion durch Setzen von Parametern angestoßen werden. Der damit erreichte Zustand, die so genannten Nachbedingungen, müssen wiederum mit der Spezifikation verglichen werden.

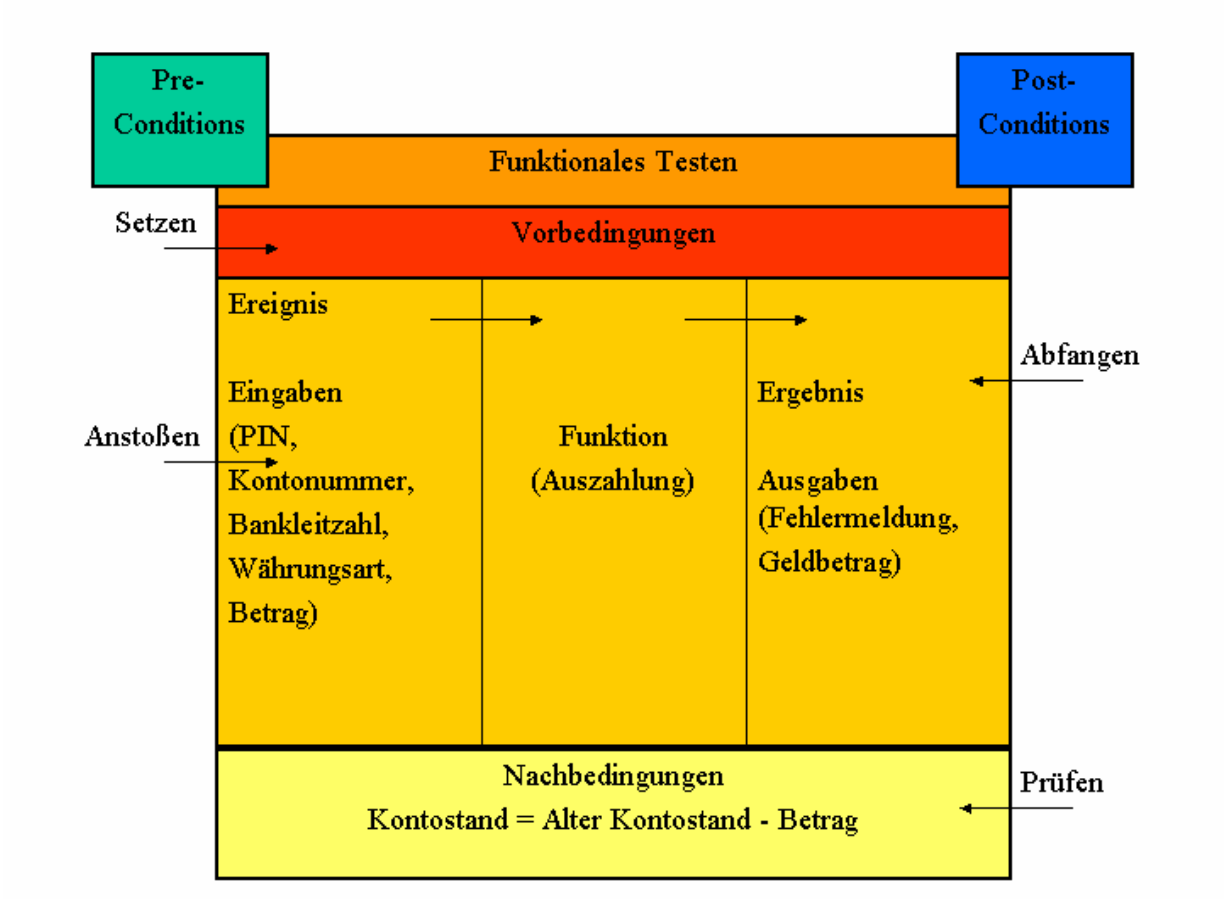


Abbildung 2.3: Funktionstest

Funktionales Testen ist typischerweise, wie oben erwähnt, ein so genannter *Black-Box*-Test. Das heißt, ein Test von Systemen, bei dem das System in seiner Gesamtheit als geschlossener Kasten betrachtet wird Es wird von außen mit Eingabedaten bzw. Ereignissen bombardiert und die Ausgaben bzw. Reaktionen werden registriert, um sie gegen die Erwartungen des Testorakels zu bestätigen. Man bräuchte nur alles einzugeben

oder anzuklicken, was einem gerade einfällt. Dies würde auf einen Zufallstest hinauslaufen, und ein Zufallstest ergibt nur zufällige Ergebnisse. Um effektiv zu sein, muss auch beim Funktionstest systematisch getestet werden, und dies bedeutet: eine Methode haben. Beizer identifiziert fünf methodische Ansätze zum Funktionstest [BEIZ83]:

- Datentest,
- Funktionsflusstest,
- Bereichstest,
- Syntaxtest und
- Zustandstest.

Bevor in Kapitel 3 die konkreten Beispiele dieser Testmethoden erarbeitet werden, sollen sie zuvor noch theoretisch und mit grafischen Beispielen umrissen werden.

Datenflusstest

Der Datenflusstest geht von den Systemeingaben und –ausgaben aus. Die Systemeingaben sind hier die Eingabefelder in der Benutzeroberfläche, die vom Benutzer gesetzt werden. Die Systemausgaben sind die Ausgabefelder sowie die Berichte. Für jede stellvertretende Ausgabemaske wird eine Kombination von Eingabewerten eingegeben, die zu diesem bestimmten Ergebnis führen soll. Im Falle von periodisch durchgeführten Ad-hoc-Berichten muss jenes Ergebnis produziert werden, das zur Generierung des Berichts führt.

Dazu müssen auch stellvertretende Parameterwerte eingegeben werden. Im Prinzip geht dieser Test von der Oberfläche aus und benutzt die Bedienungsanleitung als Grundlage [SNWI02]. Abbildung 2.4 demonstriert an einem Beispiel, wie durch das Setzen von Eingangsdaten die Ausgangsdaten ‚provoziert' werden sollen. Es werden typische Kontodaten in das System eingegeben und danach beurteilt, wie das Programm durch das Setzen der Nachbedingungen darauf reagiert. Grundlagen für den Test wären Eingabe-Ausgabe-Tabellen, Überleitungsmatrizen und Datenflussdiagramme.

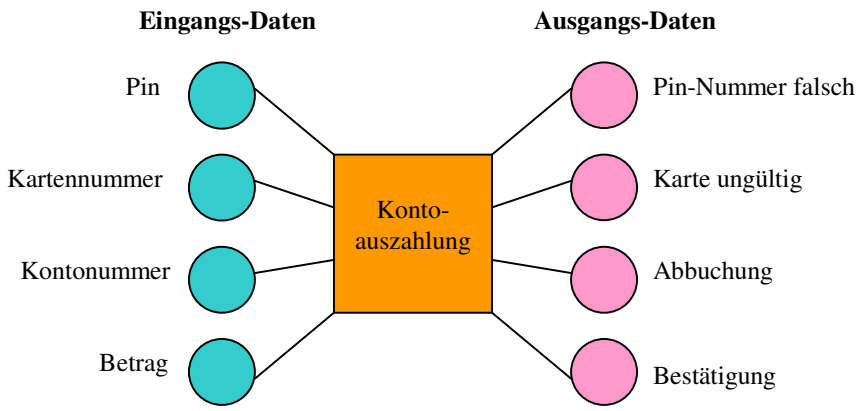


Abbildung 2.4: Datenflusstest, basierend auf Datenflussdiagrammen

Funktionsflusstest

Der Funktionsflusstest geht von den Anwendungsfällen aus. Für jeden spezifizierten Anwendungsfall wird die Summe der Bedingungen erzeugt, die zur Ausführung des Falls führt. Auf der Ausgabenseite wird dann überprüft, wie das Programm auf die Transaktionen und Prozesse reagiert. Nach diesem Ansatz gilt es, jeden Anwendungsfall in jeder praktisch relevanten Variation zu testen. Abbildung 2.5 zeigt Prozesse, die über den Funktionsfluss getestet werden können. Der Prozess ‚Kunde prüfen' wird mit verschiedenen Inputfaktoren gestartet und getestet, ob die Kreditwürdigkeit anhand der Spezifikation korrekt geprüft wird.

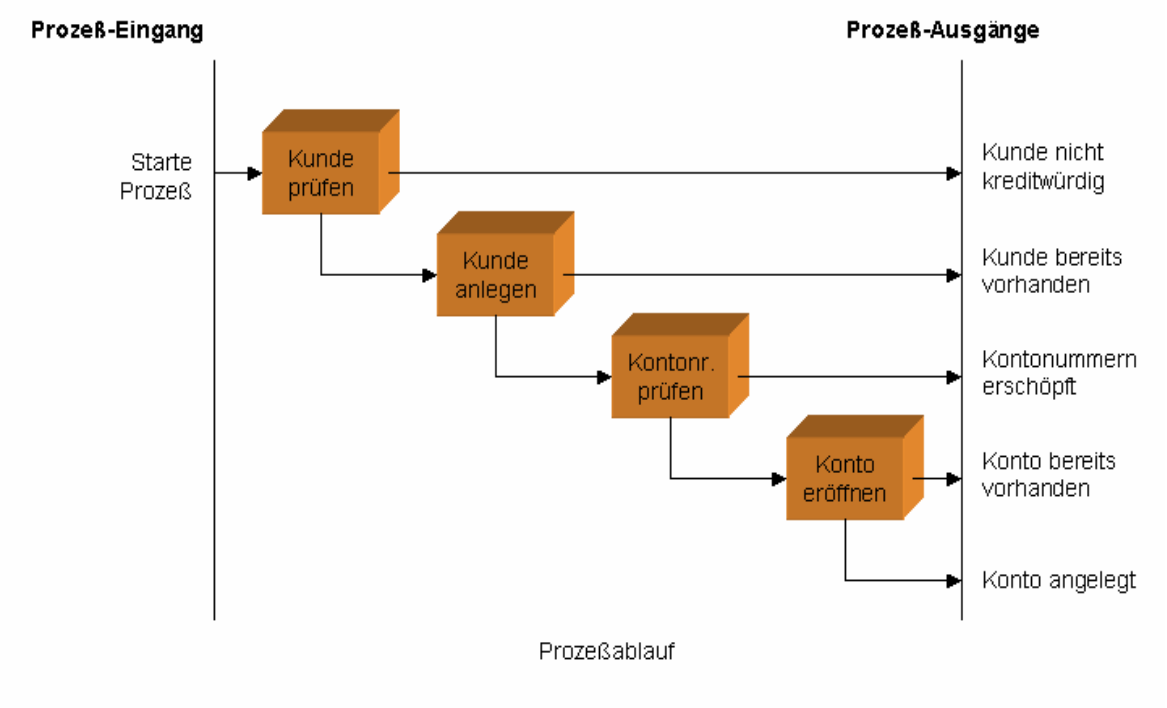


Abbildung 2.5: Prozess- oder Funktionsflusstest, basierend auf Prozessflussdiagrammen [SNEE02] (mit Einverständnis des Autors reproduziert)

Bereichstest

Beim Bereichstest werden an der Eingabeoberfläche Grenzwerte und falsche Werte eingegeben, um die Robustheit und Fehlertoleranz des Systems zu prüfen. Im Falle numerischer Werte werden die unteren und oberen Grenzwerte erprobt. Im Falle von Texten werden falsche und leere Zeichenfolgen eingegeben. Außerdem werden sowohl zulässige als auch unzulässige Tastenkombinationen und Mausklicks erprobt. Hier wird also bestätigt, ob das System auf alle stellvertretenden Impulse richtig reagiert [SNWI02]. Abbildung 2.6 zeigt Datumsprüfungen von Jahreszahlen, die innerhalb des letzten Jahrhunderts liegen (sollen).

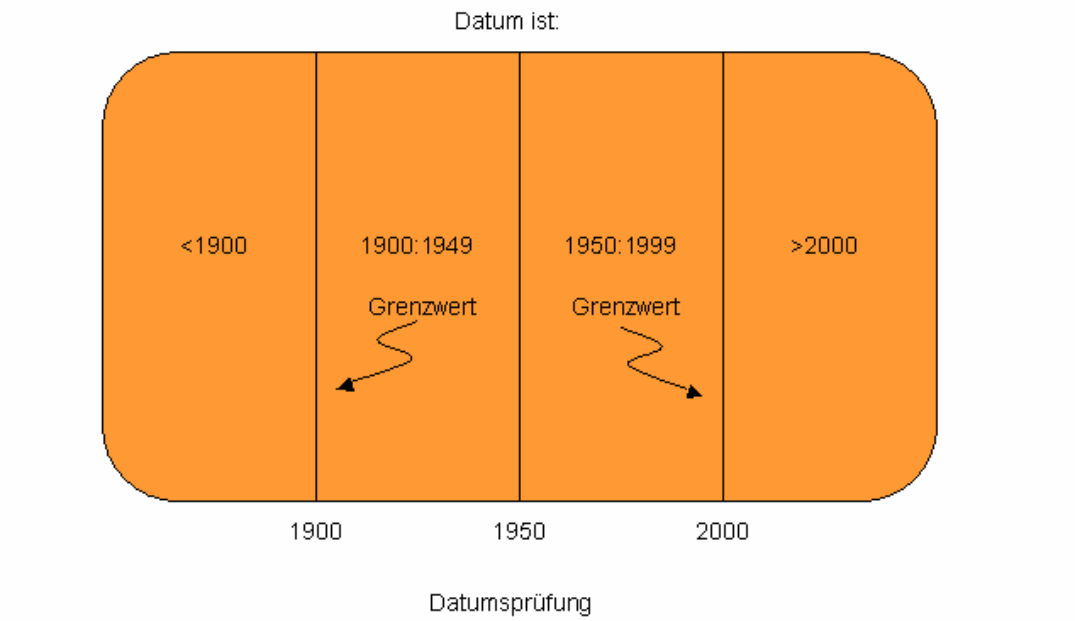


Abbildung 2.6: Bereichstest, basierend auf Datenwertebereichen

Syntaxtest

Im Syntaxtest werden alle Arten von Kommandozeilen erprobt, wobei immer andere Syntaxfehler produziert werden: einmal ein falsches Trennungszeichen, einmal ein fehlendes Wort, einmal eine falsche Wortfolge. Es wird hier also kontrolliert, ob die Syntax richtig interpretiert wird und ob bei Syntaxfehlern die korrekte Fehlermeldung erscheint [SNWI02]. Abbildung 2.7 zeigt die definitive Überprüfung jedes Syntaxteils einer Kopieranweisung in typischem DOS-Format.

DOS-Copy-Anweisung

(1) (2)(3) (4) (5) (6) (7) (8) (9)

<drive>:\[{<directory>\}][File[.ext]]

[,<drive>:\[{<directory>\}][File[.ext]]]

(10) (11) (12) (13) (14) (15)(16)(17)(18)(19)

Syntaxprüfungen

(1) drive fehlt	(10) , fehlt
(2) : fehlt	(11) drive fehlt
(3) \ fehlt	(12) : fehlt
(4) directory	(13) \ fehlt
(5) \ fehlt	(14) directory
(6) n directories	(15) \ fehlt
(7) File fehlt	(16) n directories
(8) ext fehlt	(17) File fehlt
(9) Ziel fehlt	(18) ext fehlt
	(19) Quelle fehlt

Abbildung 2.7: Syntaxtest, basierend auf Grammatik, Beispiel einer Dos-Copy-Anweisung

Zustandstest

Der Zustandstest zielt auf die Erprobung aller spezifizierten Systemzustände ab. Dazu werden Zustandstabellen, Zustandsübergangstabellen oder UML – Zustandsdiagramme verwendet, um die Testfälle an der Oberfläche zu definieren und den Endzustand an der Ausgabeoberfläche zu validieren. Abbildung 2.8 möchte demonstrieren, welche Zustände für Menschenleben möglich sind. Ausgehend von der linken Spalte kann ein Zustand einen oder mehrere Nachfolgezustände annehmen.

Zustandsübergangstabelle

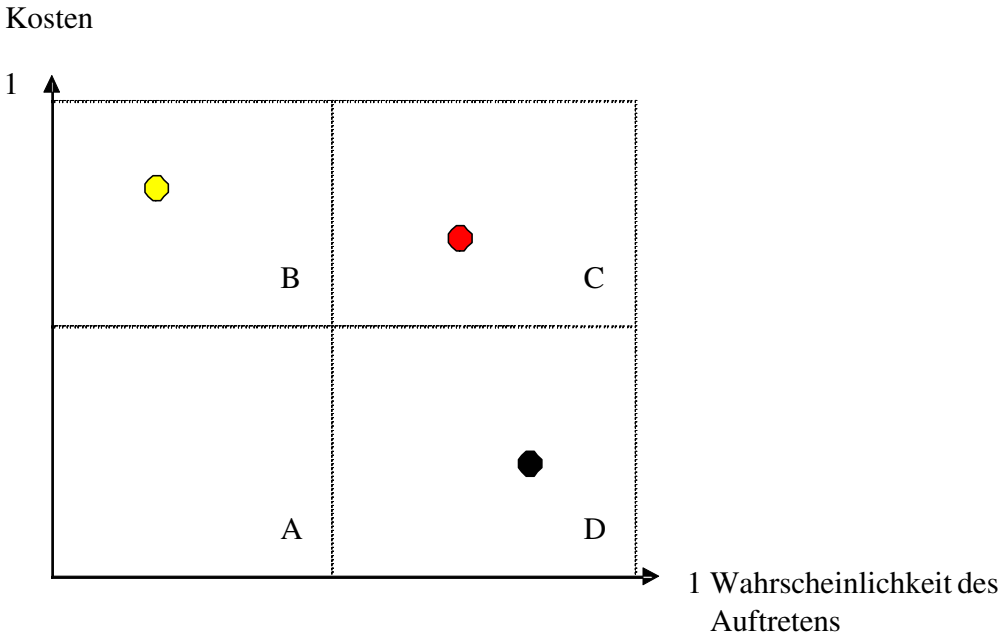
ZUSTAND	Staub	Kind	ledig	verheiratet	geschieden	verwitwet
Staub		X				
Kind	X		X			
ledig	X			X		
verheiratet	X				X	X
geschieden	X		X			
verwitwet	X		X			

Abbildung 2.8: Zustandstest, basierend auf Zustandsübergangsgraphen und Zustandsübergangstabellen

2.4.4 Risikotest

Risikobasiertes Testen ist vor allem bei Großsystemen ein wichtiger Ansatz. Da es unmöglich ist, in einem umfangreichen Softwareprodukt alle Fehler zu finden oder alle möglichen Programmteile zu testen, sollte eine Risikogewichtung vorgenommen werden, die eine grobe Unterteilung gibt, wo die Fehlerhäufigkeit am größten ist, und welche Fehler massiven Schaden bei Kunden anrichten und somit eventuell zu Schadenersatzforderungen führen können.

Abbildung 2.9 zeigt eine Matrix nach möglichem Schadensausmaß und Wahrscheinlichkeit des Auftretens, die eine Aufteilung in 4 Gruppen trifft.



A – geringes Schadensausmaß, geringe Wahrscheinlichkeit

B – hohes Schadensausmaß, geringe Wahrscheinlichkeit

C – hohes Schadensausmaß, hohe Wahrscheinlichkeit

D – geringes Schadensausmaß, hohe Wahrscheinlichkeit

Abbildung 2.9: Matrix nach möglichem Schadensausmaß und Wahrscheinlichkeit

Der Autor schlägt eine Prüfung und Behebung der möglichen Probleme in der Reihenfolge C – B – D – A vor, da die Erfahrung zeigt, dass die Zufriedenheit des Kunden eher durch seltene Produktionshemmungen als durch zahlreichere kleinere Programmfehler, auch wenn diese häufiger auftreten, beeinträchtigt wird.

2.4.5 Regressionstest

Sowohl durch Wartungsarbeiten als auch bei Weiterentwicklung werden Teile vorhandener Software geändert oder um neue Softwarebausteine ergänzt. In beiden Fällen muss die geänderte Software erneut getestet werden. Diese Tests heißen Regressionstests.

Der Regressionstest ist ein erneuter Test eines bereits getesteten Programms nach dessen Modifikation mit dem Ziel nachzuweisen, dass durch die vorgenommenen Änderungen keine neuen Defekte eingebaut oder bisher versteckte Fehlerzustände freigelegt wurden [THAL02].

Die 2 Extreme für die Durchführung wären [THAL02]:

- Alle Tests für das gesamte Programm wiederholen.
- Die Tests auf die Testfälle beschränken, bei denen die Fehler aufgetreten sind.

Sollte keine Möglichkeit bestehen, genau die Code Segmente und Programmstellen nach geänderten Modulen identifizieren zu können, liegt die Praxis irgendwo dazwischen und muss vom jeweiligen Testmanager nach einer Kosten-Nutzen-Rechnung abgeschätzt werden.

3. Methodik zur Ableitung der Testfälle nach CMF

Testfälle werden je nach effektiver Durchführbarkeit in logische und konkrete Testfälle unterteilt. Logische Testfälle beschreiben die Rahmenbedingungen einer Testdurchführung, wohingegen konkrete Testfälle auch die Werte und Parameter für die Durchführung beinhalten. Zuerst soll mit dem Aufbau von logischen Testfällen begonnen werden. Diese Arbeit versucht das Wissen um Funktionstestfälle in einem standardisierten oder standardisierbaren Format darzustellen.

Die Quelle der Testfälle ist ein semiformales Spezifikationswerkzeug, das als das *Concept-Management-Facility* - CMF bezeichnet wird. Das Ziel ist eine Datenbank für Testfälle gekennzeichnet als das *Concept-Test-Facility* - CTF. Diese zwei Toolwerkzeuge werden verwendet, um ein Standard Wertpapierhandels- und Informationssystem - GEOS - das *Global-Entitiy-Online-System* während der Entwicklung zu spezifizieren und zu prüfen.

3.1 Aufbau des CMF Konzepts

CMF – das Concept Management Facility – ist eines von der SDS entwickelten Tools und wird mittels einer Baumstruktur in die verschiedenen Systeme, Teilsysteme und Komponenten unterteilt. Die Komponenten wiederum werden in Servicefunktionen, Dialoge, Strukturen und Batches unterteilt, um die wichtigsten zu nennen. Wenn man nun eine Servicefunktion lädt, wird das Bearbeitungsfenster sofort in fachliche Kurzbeschreibung und den Ablauf unterteilt.

Die **Fachliche Kurzbeschreibung** gibt schnellen Überblick, was die betreffende Funktion für Tätigkeiten ausführt, und wird als Grundlage für das Benutzerhandbuch verwendet und in dieses übergeleitet.

Der **Ablauf** stellt den technischen Ablauf einer Funktion dar und bildet eigentlich den Grundstein des ‚zu programmierenden' Konzeptes.

Per Funktionsknopf können weitere Dialogfenster geöffnet werden, um das Konzept näher zu spezifizieren: Für das Fachkonzept der Servicefunktionen wären dies wie in der Abbildung 3.1 beispielsweise: Überleitungsmatrix, Parameter, Attribute, Nachweis, Benutzer, Journal, Detail, Prüfungen, Anhang. Reportkonzepte lassen die Pflege von den verwendeten Strukturen des Datenstroms zu, während Dialogkonzepte die Möglichkeit bieten über den Menüpunkt Pattern direkt in den Dialog zu verzweigen und per Mouseclick die jeweiligen Frontendprüfungen der Dialogfelder abzufragen.

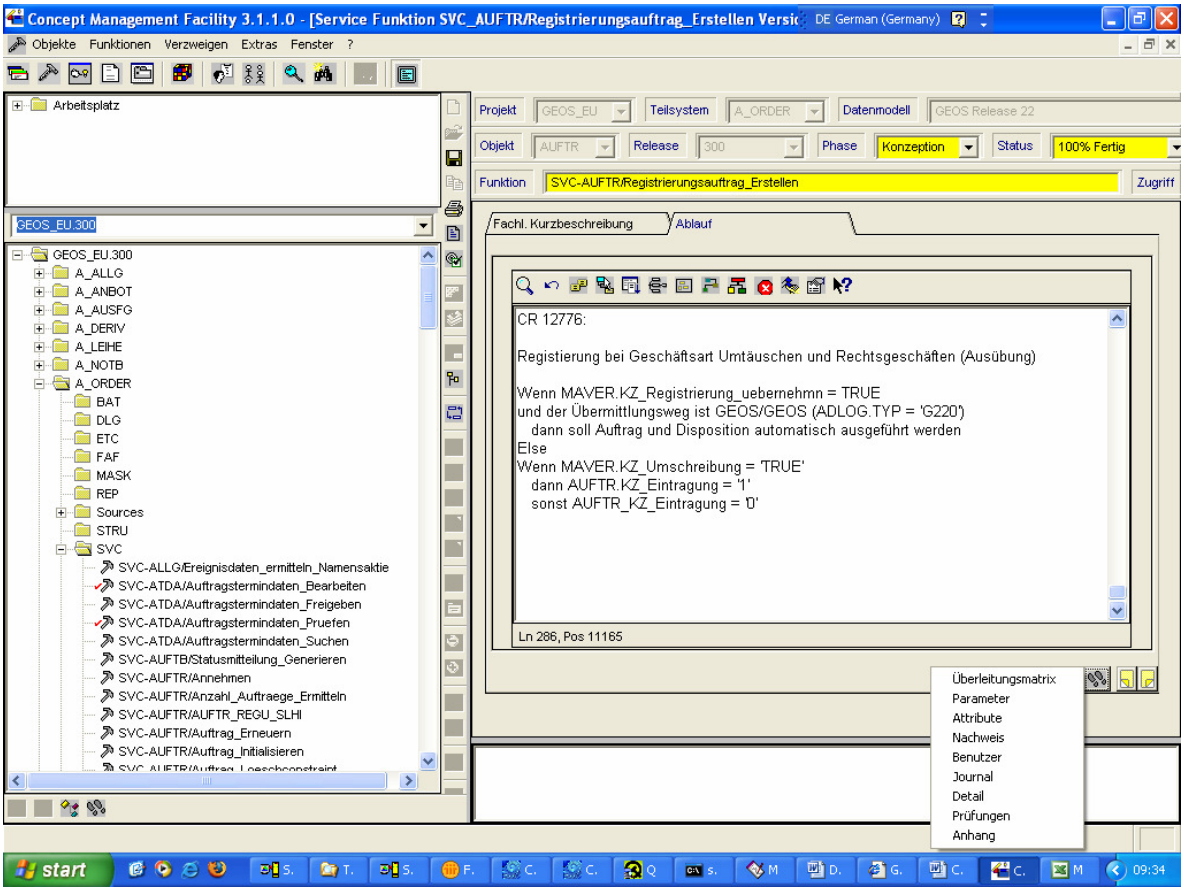


Abbildung 3.1: CMF – Dialog mit Ablaufbeschreibung, Verzweigungsmöglichkeiten und übergeordneter Baumstruktur

3.1.1 Potentielle Testfälle in CMF

Die möglichen, aus dem CMF abgeleiteten, Testfälle können folgende Formen annehmen:

- das Produzieren einer Fehlermeldung, die sich aus einer Prüfregel ergibt,
- der Aufruf einer Servicefunktion,
- das Setzen eines ungültigen Parameters,
- das Verhindern eines Datenbankzugriffs,
- das Erfüllen einer logischen Bedingung.

3.1.2 Testfallergiebigkeit des CMF

Durch Analyse der im CMF geschriebenen Konzepte wäre es möglich, ca. 67.000 Testfälle zu erzeugen, die auf Regeln, Parameter, Funktionsaufrufe, Datenzugriffe und logische Bedingungen, die in CMF spezifiziert werden, basieren.

3.2 Logische Testfälle

Testfälle sollen als Thema dieser Arbeit von semiformalen Konzepten abgeleitet werden. Logische Testfälle versuchen die Bedingungen, Regeln und Relationen einer zugrunde liegenden Problemstellung zu erfassen und innerhalb von Testgrundlagen umzusetzen. Es kann hierbei eine formale graphische Sprache als Hilfsmittel zur Wahl eines Satzes von Testfällen herangezogen werden. Beispielhaft soll ein grundlegender Arbeitsablauf für die Testfallermittlung dargestellt werden [SCHL04]:

- Zerlegung der Spezifikation in handhabbare Teile.
- Ursache und Wirkung, Vorbedingung und Nachbedingung jeder Teilspezifikation ermitteln. Ursachen in Form einer einzelnen Eingabebedingung oder einer Äquivalenzklasse von Eingabebedingungen, Wirkungen in Form von Ausgabebedingungen oder Systemtransformationen. Ursachen und Wirkungen

werden durch Analyse der Spezifikation ermittelt.

- Transformation der Spezifikation in Ursache-Wirkungsgraph o.ä..
- Eintragen der Abhängigkeiten zwischen Ursachen und Wirkungen.
- Umformung der Graphen in eine Entscheidungstabelle.
- Aus jeder Spalte der Entscheidungstabelle einen Testfall erzeugen.
- Zuweisung von Werten.

Die Schritte den Ursache-Wirkungsgraph betreffend sind optional und können durch andere graphische Konzeptsprachen ersetzt werden (z.B. Ablaufdiagramm).

Bevor wir die Testmethodik und im Besonderen die Haupttestmethoden

Funktionsbezogene Testfälle,

Zustandsbezogene Testfälle,

Datenbezogene Testfälle,

Ablaufbezogene Testfälle

näher betrachten, soll zuvor noch die Fachlogik und die Geschäftsregeln näher beleuchtet werden.

3.3 Geschäftsregeln, Fachlogik, Entscheidungstabellen

Entscheidungstabellen können logische Zusammenhänge formal darstellen, wenn regelbasierte Expertensysteme untersucht werden, oder wenn Programmlogiken größtenteils auf IF THEN Relationen aufbauen.

Eine **Entscheidungstabelle** ist ein Methodisches Hilfsmittel zur transparenten Darstellung von Bedingungskombinationen. Eine Entscheidungstabelle besteht aus je einer Liste von Bedingungen und Regeln, wahlweise ergänzt durch Relationen [BLÜM03]. Abbildung 3.2 stellt diese Bestandteile in tabellarischer Form dar.

Identifikationsteil			
Erste Bedingung . . . . Letzte Bedingung	Regel Regel … 	Bedingungsanzeiger … …	Bedingungsteil
Erste Aktion . . . . Letzte Aktion	ELSE- Aktions- anzeiger …	Aktionsanzeiger … … 	Aktionsteil
Textteil	Regelteil		

Abbildung 3.2: Bestandteile von Entscheidungstabellen

Bedingungen stellen die linke Hälfte einer Entscheidungstabelle dar. Jede Zeile enthält eine logische Bedingung, die entweder mit ja/nein beantwortet werden kann (z.B. "Kontonummer zulässig") oder einer abzählbaren Relation zuzuordnen ist (z.B. "Wohnort = " mit den drei Fällen "WIEN", "INLAND", "AUSLAND").

Regeln sind die rechte Hälfte einer Entscheidungstabelle. Eine Regel ist die Menge aller Bedingungs- und Aktionsanzeiger in einer Anzeigerspalte (Regelspalte). Jede Spalte enthält eine Kombination möglicher Antworten auf die Bedingungen der linken Hälfte.

Erlaubte Antworten sind in der Tabelle in Abbildung 3.3 ersichtlich [PLAT88]:

Bedingungsanzeiger		Bedeutung innerhalb der Regel	Anzeigerart
J	Ja	Bedingung muss erfüllt sein	Begrenzter Anzeiger
N	Nein	Bedingung darf nicht erfüllt sein	Begrenzter Anzeiger
-	Unwichtig	Bedingung ist ohne Bedeutung	Begrenzter und erweiterter Anzeiger
#	nicht definiert	Bedingung ist nicht definiert	Begrenzter und erweiterter Anzeiger
Ziffer	für Relation Nr. 1, 2, 3 ...	Bedingung und Relation muss erfüllt sein	Erweiterter Anzeiger
Beliebiger Text	für Relationen	Bedingung und Text müssen erfüllt sein	Erweiterter Anzeiger

Abbildung 3.3: Regeltabelle

Regeltabellen sammeln die Geschäftsregeln einer Spezifikation tabellarisch und dienen als Grundlage für die Umsetzung in Entscheidungstabellen, denn oftmals liegt die Schwierigkeit in der Identifikation der jeweiligen Regeln (siehe Abbildung 3.3).

Relationen sind der fakultative Teil einer Entscheidungstabelle. Einer Bedingung wird hier nicht die Antwort ja/nein zugeordnet, sondern für aufzählbare Fälle 1, 2, 3 ... je eine ergänzende Antwort.

Beispiel: Bedingung "Kontostand"

Relation Fall 1 "<0"

Fall 2 ">0" und "<100000"

Fall 3 "=0"

Zu einem Fall können mehrere Bedingungsergänzungen in je einer Zeile eingegeben

werden. Diese sind dann durch UND verknüpft.

Aktionen sind die Tätigkeiten, die durch Vorbedingungen ausgelöst werden sollen. (Siehe Abbildung 3.4)

Aktionsanzeiger	Bedeutung innerhalb einer Regel	Anzeigerart
-	Aktion nicht ausführen, wenn nur diese Regel zutrifft	Begrenzter und erweiterter Anzeiger
X	Aktion ist auszuführen, bei Zutreffen dieser Regel	Begrenzter Anzeiger
Beliebiger Text	Durch Aktionstext und Text gebildete Aktion ist auszuführen	Erweiterter Anzeiger

Abbildung 3.4: Aktionstabelle

Preconditions sind sozusagen der WENN-Teil eines Testfalls. Diese werden generiert aus jeweils einer Regel (Spalte in der Entscheidungstabelle) als die Summe der Bedingungen, die dieser Regel zugeordnet worden sind. Dabei werden die Texte von Bedingungen und eventuellen Relationen über Keywords (bsp.: *IF, IFNOT* und *CASE)* miteinander verknüpft.

Postconditions stellen den DANN-Teil eines Testfalls dar, die gewünschte Nachbedingung nach Durchführung (Sollzustand). Sie sind die Summe der gewünschten Aktionen, Zustände oder Nachbedingungen.

Folgende Regeln für Entscheidungstabellen können angewendet werden [PLAT88]:

- Bedingungsanzeigerteil enthält nur die Eintragungen ja, nein, ,-, (irrelevant), Aktionsteil enthält nur die Eintragungen X und ,-,.
- Hat eine begrenzte ET n **Bedingungen**, so hat sie 2^n **Regeln** bei einer Standardtabelle.
- Eine Regel ist die Menge aller Bedingungs- und Aktionsanzeiger in einer

Anzeigerspalte (Regelspalte).

- Bildungsgesetz für eine vollständige, begrenzte ET (Standardtabelle):

  1. Bedingung ½ ja ½ nein

  2. Bedingung ¼ ja ¼ nein ¼ ja ¼ nein

  …

  n. Bedingung j n j n j n j n …

- Zwei Regeln einer begrenzten ET lassen sich konsolidieren, (d.h. zu einer Regel unter Benutzung des Anzeigers ‚-' zusammenfassen), wenn sich bei **gleichen Aktionsanzeigern** bei genau **einer Bedingung** in den Bedingungsanzeigern die Eintragungen durch ‚ja' und ‚nein' **unterscheiden.**

- Eine Regel einer ET mit n ‚-' **Irrelevanzanzeigern** lässt sich in **2^n Regeln ohne Irrelevanzanzeiger** auflösen.

- Bei einer komplett und richtig konsolidierten (wenn sich 2^n Regeln ohne Irrelevanzanzeiger bei n Bedingungen ergeben würden (siehe vorige Regel)) begrenzten ET lassen sich alle **Regeln mit gleichen Aktionsanzeigern** in der so genannten **ELSE-Regel** zusammenfassen. Man verwendet hier die am häufigsten vorkommenden gleichen Aktionsanzeiger.

- Eine **Redundanz** bei 2 Regeln einer begrenzten ET liegt immer dann vor, wenn sie bei **gleichen Bedingungsanzeigern auch gleiche Aktionsanzeiger** haben, d.h. wenn die Regeln komplett identisch sind.

- Ein **Widerspruch** bei 2 Regeln einer begrenzten ET liegt immer dann vor, wenn sie bei **gleichen Bedingungsanzeigern unterschiedliche Aktionsanzeiger** haben.

- Die Aktion **‚UNLOGISCH'** ist einzuführen, wenn durch Anwendung der **Standardtabelle** bei voneinander abhängigen Bedingungen **unlogische Bedingungskonstellationen** auftreten. Bei unlogischen Testfällen hängt es davon ab, ob sie technisch unmöglich sind, dann müssen diese unter gewissen Voraussetzungen

nicht getestet werden (Konstellation nicht aufrufbar oder aus Effizienzgründen), oder ob sie nur fachlich „keinen Sinn“ machen, denn diese müssen sehr wohl durchgeführt werden.

3.4 Konkrete Testfälle

Sind die Testfälle einmal über Ursache und Wirkungen, Bedingungen, Regeln und Relationen oder Abläufe, etc. ermittelt worden, liegt noch ein wichtiger Schritt in der Zuweisung von Parametern, damit die Testfälle mit konkreten Werten angelegt werden können.

Für den Systemtest sollten die Testdaten alle möglichen Werte jedes Parameters der Spezifikation annehmen. Weil das jedoch unmöglich ist, sollten wenige Werte jeder Äquivalenzklasse gewählt werden.

Idealerweise sollten Testfälle, die Fehlersituationen testen, unabhängig von Funktionstestfällen geschrieben werden, und sollten Schritte beinhalten, die die Fehlermeldungen und Logfiles überprüfen. Sollten ‚Errortestfälle’ noch nicht geschrieben worden sein (was sicherlich den häufigeren Fall darstellt), ist es akzeptabel, wenn der Tester die Fehlersituationen während der Durchführung der Funktionstestfälle (mit)überprüft. Dennoch sollte klar sein, welche Testdaten Fehlersituationen provozieren [TIGR06].

3.4.1 Äquivalenzklassenbildung

Ziel und Zweck

Ziel der Äquivalenzklassenbildung ist es, durch Bildung von Äquivalenzklassen eine hohe Fehlerentdeckungswahrscheinlichkeit mit einer minimalen Anzahl von Testfällen zu erreichen.

Funktioneller Ablauf

Das Prinzip der Äquivalenzklassenbildung besteht darin, die gesamten Eingabedaten

eines Programms in eine endliche Anzahl von Äquivalenzklassen zu unterteilen, so dass man annehmen kann, dass mit jedem beliebigen Repräsentanten einer Klasse die gleichen Fehler wie mit jedem anderen Repräsentanten dieser Klasse gefunden werden [PLÖG02].

Die Definition von Testfällen mit Hilfe der Äquivalenzklassenbildung erfolgt in folgenden Schritten [PLÖG02]:

- Analyse der Dateneingabeanforderungen, der Datenausgabeanforderungen und der Bedingungen gemäß den Spezifikationen,
- Bestimmung der Äquivalenzklassen durch Einteilung der Wertebereiche für Ein- und Ausgabegrößen,
- Bestimmung der Testfälle durch Werteauswahl für jede Klasse.

Bei der Festlegung der Äquivalenzklassen werden zwei Gruppen von Äquivalenzklassen unterschieden:

- gültige Äquivalenzklassen
- ungültige Äquivalenzklassen

Bei gültigen Äquivalenzklassen werden erlaubte Eingabedaten, bei ungültigen Äquivalenzklassen fehlerhafte Eingabedaten ausgewählt. Die Bestimmung der Äquivalenzklassen ist bei Vorgabe der Spezifikationen hauptsächlich ein heuristischer Prozess [PLÖG02].

Beispiel – Gratifikation und Weihnachtsbonus

In der Anforderung findet sich folgende Textpassage

„Mitarbeiter erhalten ab einer Zugehörigkeit zur Firma von mehr als drei Jahren 50% des Monatsgehalts als Weihnachtsgratifikation. Mitarbeiter, die länger als fünf Jahre in der Firma tätig sind, erhalten 75%. Bei einer Firmenzugehörigkeit von mehr als acht Jahren wird eine Gratifikation von 100% gewährt“

Gültige Äquivalenzklassen

Parameter	Äquivalenzklasse	Repräsentant
Gratifikation	gÄK1: 0 <= x <= 3	3
	gÄK1: 3 <= x <= 5	5
	gÄK1: 5 <= x <= 8	6
	gÄK1: x > 8	10

Ungültige Äquivalenzklassen

Parameter	Äquivalenzklasse	Repräsentant
Gratifikation	gÄK1: x < 0 negative (falsche) Betriebszugehörigkeit	-3
	gÄK1: x > 73 unrealistisch lange, falsche Betriebszugehörigkeit	85

Abbildung 3.5: Äquivalenzklassen

Die Repräsentanten ungültiger Äquivalenzklassen werden nicht miteinander kombiniert. Ein ungültiger Wert soll nur mit ‚gültigen' kombiniert werden. Einerseits, weil mehrere Fehler sich gegenseitig aufheben könnten, andererseits ist die Fehleranalyse aufwendiger, weil man nicht weiß, welcher Fehler nun zur jeweiligen Ausnahmesituation geführt hat [SPLI03].

3.4.1.1 Beispiele für Äquivalenzklassen

Folgende Checkliste hat sich als nützlich erwiesen und liefert gute Beispiele für die Findung der Äquivalenzklassen [TIGR06]:

Zeichenketten / strings

- leerer string
- string bestehend nur aus Leerzeichen
- string mit Leerzeichen am Anfang oder am Ende
- syntaktisch korrekte short and long values
- syntaktisch gültig: semantisch gültige und ungütige Werte
- syntaktisch ungültige Werte: ungültige Zeichen oder Kombinationen
- sicherstellen, dass spezielle Zeichen getestet werden: #, ", ', &, and <, etc.
- sicherstellen, dass "Foreign" characters auf internationalen Tastaturen getestet werden.

Zahlen / numbers

- leerer string, wenn möglich
- 0
- positive Zahlen innerhalb des Wertebereichs / Domäne, small and large
- negative Zahlen innerhalb des Wertebereichs / Domäne, small and large
- außerhalb des Wertebereichs / Domäne: positiv
- außerhalb des Wertebereichs / Domäne negativ
- mit führenden Nullen
- syntaktisch ungültig (z.B.: string beinhaltet Buchstaben)

Identifikationen

- leere Zeichenkette
- syntaktisch gültiger Wert
- syntaktisch gültig: verweisen auf bestehende ID, und auf ungültige Referenzen
- syntaktsch ungültiger Wert

Radio Buttons

- eine Wahlmöglichkeit getroffen
- keine Wahlmöglichkeit getroffen, wenn möglich

Checkboxes

- behakt
- nicht behakt

Drop down menus

- einmal jede Möglichkeit oder Funktion wählen

Scrolling Listen

- Nichts wählen, wenn möglich
- jeden Bestandteil der Liste einmal wählen
- Mehrfachselektionen treffen, wenn möglich
- alle Bestandteile selektieren, wenn möglich

File upload

- blank
- file mit 0 byte
- long file
- short file name
- long file name
- syntaktisch ungültiger file name, wenn möglich (z.B.: "File with Spaces.tar.gz")

3.4.2 Grenzwertanalyse

Ziel und Zweck

Ziel der Grenzwertanalyse ist es, Testfälle zu definieren, mit denen Fehler im Zusammenhang mit der Behandlung der Grenzen von Wertebereichen aufgedeckt werden können.

Funktioneller Ablauf

Das Prinzip der Grenzwertanalyse besteht darin, die Grenzen von Wertebereichen bei der Definition von Testfällen zu berücksichtigen. Ausgangspunkt sind die mittels Äquivalenzklassenbildung ermittelten Äquivalenzklassen. Im Unterschied zur Äquivalenzklassenbildung wird kein beliebiger Repräsentant der Klasse als Testfall ausgewählt, sondern Repräsentanten an den Grenzen der Klassen. Die Grenzwertanalyse

stellt somit eine Ergänzung des Testfallentwurfs gemäß Äquivalenzklassenbildung dar [PLÖG02].

Binder kombiniert Grenzwertmethode und Äquivalenzklassen mit der Methode ***Invariant Boundaries*** und geht dabei wie folgt vor: Die Domänen werden erkannt und geordnet, die so genannten ON und OFF Points werden definiert, und mit den jeweiligen Werten in das Datentestset mit Echtwerten weiter übertragen, wobei in dieser speziellen von Binder gewählten Methode jedoch keine Ausgabewerte berechnet werden sollen, sondern nur die Entscheidung: OK – Reject getroffen werden soll [BIND05].

Beispiel: Tageszeiten innerhalb der gültigen Datumsdomänen der SDS

Variable	Type	Minimum	Maximum
		(ON)	***(OFF)***
sec	integer	0	59
min	integer	0	59
hour	integer	0	23
day	integer	1	31
month	integer	1	12
year	integer	1800	2800

Somit wären die ON und OFF Punkte des Monats Juni wie folgt, die ‚echten' Grenzwerte sind hervorgehoben und diese werden mit den zu testenden Randwerten ergänzt:

ON

(A0)	31.05.2006.23:59:59	Reject
(A1)	**01.06.2006.00.00.00**	**Accept**
(A2)	01.06.2006.00.00.01	Accept

OFF

(B0)	**30.06.2006.23.59.59**	**Accept**
(B1)	01.07.2006.00.00.00	Reject
(B2)	01.07.2006.00.00.01	Reject

Abbildung 3.6: Grenzwertklassen

Nach Schlingloff sollte folgendes getestet werden [SCHL04]:

die Grenzen des Eingabebereichs:

Bereich: [-1.0;+1.0]; Testdaten: -1.001; -1.0; +1.0; +1.001

Bereich:]-1.0;+1.0[; Testdaten: -1.0; -0.999; +0.999; +1.0

die Grenzen der erlaubten Anzahl von Eingabewerten:

Eingabedatei mit 1 bis 365 Sätzen; Testfälle 0, 1, 365, 366 Sätze

die Grenzen des Ausgabebereichs:

Programm errechnet Beitrag, der zwischen 0,00 EUR und 600 EUR liegt;

Testfälle: Für 0; 600 EUR und möglichst auch für Beiträge < 0; > 600 EUR

die Grenzen der erlaubten Anzahl von Ausgabewerten:

Ausgabe von 1 bis 4 Daten; Testfälle: Für 0, 1, 4 und 5 Ausgabewerte

Erstes und letztes Element bei geordneten Mengen muss beachtet werden:

z.B.: sequentielle Datei, lineare Liste, Tabelle

Bei komplexen Datenstrukturen sollen leere Mengen getestet werden:

z.B. leere Liste, Nullmatrix

Die **Vorteile** der Grenzwertanalyse wären, dass bei den Grenzen von Äquivalenzklassen häufiger Fehler zu finden sind als innerhalb dieser Klassen, sie gibt klare Regeln für den Testfallentwurf vor und kann jederzeit mit freieren Testdatenverfahren kombiniert werden.

Nachteile sind, dass die Bestimmung der Grenzwerte oftmals schwierig ist, dass Kreativität zur Findung erfolgreicher Testdaten gefordert ist, und dass die Grenzwertanalyse oftmals nicht effizient genug angewendet wird, da sie zu simpel oder einfach erscheint [SCHL04].

3.5 Testmethodik

3.5.1 Funktionsbezogener Test

Definition

Der funktionsbezogene Test - *Functional Testing* genannt - ist ein Test auf der Basis der Funktionsspezifikation. Eine Funktion ist eine Regel für die Erzeugung eines Ergebnisses oder einer Ergebnismenge aus einer Reihe von Argumenten.

x = f (y, z);

Bei einem funktionsbezogenen Testfall wird vom Ergebnis ausgegangen, d.h. man erwartet ein bestimmtes Resultat. Die Frage ist, welche Daten muss man eingeben, um zu diesem Ergebnis zu gelangen.

Um das herauszubekommen, bieten sich drei Möglichkeiten an:

erstens, es ist im Benutzerhandbuch beschrieben (Dokumentation),

zweitens, es ist in der Programmvorgabe spezifiziert (Konzept),

drittens, es steckt im Programm selbst (Code).

Trifft keine der oben genannten Vorgaben zu, bleibt nur mehr eine Möglichkeit: Jemand hat es in seinem Kopf. Ein Domänenexperte wird immer wissen, wie bestimmte Ergebnisse zustande kommen sollten. Entweder er schreibt sein Wissen in einer informellen oder einer formellen Dokumentation auf, oder er schreibt die Programme selbst.

In den prototypbasierten Entwicklungen werden die Experten gezwungen, ihr Wissen in einer Programmiersprache wie z.B. Visual-Basic festzuhalten und zu bestätigen. Anschließend werden die endgültigen Programme anhand der Prototypenprogramme gefertigt. Danach ist es möglich, die Testfälle aus dem Prototyp abzuleiten. Dies ist eine bewährte Methode für die Entwicklung funktionaler Testfälle [SNWI02].

Ableitung über Prädikatenlogik

Zuerst müssen die Regeln erkannt werden, diese können in eine Regeltabelle transformiert werden. Wenn beispielsweise in einem Dialog geprüft werden soll, dass das Eingangsdatum eines Auftrags nicht nach dem Gültigkeitsende liegen soll, kann als Prüfregel: Eingangsdatum <= Gültigkeitsdatum formuliert werden.

Editierregeln definieren gültige Wertebereiche für Eingabe- und Ausgabefelder (Input- und Outputfelder), z.B.: Feld_A = Range (3:41) oder auch Beziehungen zwischen verschiedenen solcher Felder, z.B.: Feld_A > Feld_B

Die Bedingungen müssen von den Aktionen unterschieden werden; man könnte die **Bedingungen** mit einer durchgezogenen und die **Aktionen** mit einer gestrichelten Linie unterstreichen. Die unterstrichenen Textteile sollten in einer Übersicht gesammelt werden (in der Regeltabelle), wobei Bedingungen und Aktionen getrennt werden. Ähnliche oder identische Bedingungen können zusammengefasst werden, oder je nach Problemstellung können auch Bedingungen, die sich gegenseitig ausschließen, zusammengefasst werden.

Die gesammelten Prüfregeln bilden den Bedingungsteil (Preconditions) der Entscheidungstabelle und können mit dem Bedingungsanzeiger gesteuert werden (TRUE, FALSE, usw.). Die Entscheidungstabelle muss in weiterer Folge um die Aktionen, die über die Regeln den Bedingungen zugewiesen werden sollen, erweitert werden.

Somit wäre vorerst die Basis vorhanden, um die Anzahl der erwarteten Testfälle zu prognostizieren, die Knoten für die Entscheidungsbäume zu definieren, und die konkreten Wertebereiche mit geeigneten Domänen zu versorgen.

Folgendes **Theoriebeispiel** soll nun den beschriebenen Prozess verdeutlichen, die verbale Problembeschreibung mit gekennzeichneten Prüfregeln wäre:

Wenn die vereinbarte Kreditgrenze des Ausstellers eines Schecks überschritten wird, das bisherige Zahlungsverhalten aber einwandfrei war und der Überschreitungsbetrag kleiner als 500,-DM ist, dann soll der Scheck eingelöst werden.

Wenn die Kreditgrenze überschritten wird, das bisherige Zahlungsverhalten einwandfrei war, aber der Überschreitungsbetrag über 500,-DM liegt, dann soll der Scheck eingelöst und dem Kunden neue Konditionen vorgelegt werden.

War das Zahlungsverhalten nicht einwandfrei und ist die Kreditgrenze überschritten, wird der Scheck nicht eingelöst.

Der Scheck wird eingelöst, wenn der Kreditbetrag nicht überschritten ist.

Regeltabelle

Die Umsetzung in Aktionen und Bedingungen getrennt stellt sich folgendermaßen dar:

Bedingungen

Kreditgrenze überschritten?

Zahlungsverhalten einwandfrei?

Überschreitungsbetrag <500,-DM?

Aktionen

Einlösen des Schecks

Nichteinlösen des Schecks

Vorlegen neuer Konditionen

In der Entscheidungstabelle wird nun festgelegt, wie die Aktionen von den Bedingungen abhängen.

Es wird eine vollständige Entscheidungstabelle, wie unter Abbildung 3.7 ersichtlich, angelegt, die im Aktionsteil um eine ”unlogisch” - Aktion erweitert wurde.

	R1	R2	R3	R4	R5	R6	R7	R8
Kreditgrenze überschritten J	J	J	J	J	N	N	N	N
Zahlungsverhalten einwandfrei	J	J	N	N	J	J	N	N
Überschreitungsbetrag <500,-DM	J	N	J	N	J	N	J	N
Scheck einlösen	X	X			X		X	
Scheck nicht einlösen			X	X				
Neue Konditionen vorlegen			X					
unlogisch						X		X

Abbildung 3.7: Testfalltabelle – Scheckeinlösung

Nun soll mittels eines Fallbeispiels die Testfallableitung durch ein dem CMF entnommenes Konzept verdeutlicht werden:

Fallbeispiel: Ableitung über Regeln

Hier geht es um die Registrierung von Namensaktien nach einer durchgeführten Kapitalmaßnahme. Je nachdem, wie man die Kennzeichen in den ‚Mandantenspezifischen Verarbeitungsdaten' (MAVER) kombiniert, sollen Registrierungaufträge umgeschrieben, ersteingetragen oder ausgeführt werden. Dieses wurde in der folgenden ‚technischen' Spezifikation beschrieben:

Wenn MAVER.KZ_Registrierung_uebernehmn = TRUE
Dann AUFTR.KZ_Umschreibung = ‚0'
Wenn der Übermittlungsweg = GEOS/GEOS
Dann soll Auftrag und Disposition automatisch ausgeführt werden
Sonst
Wenn MAVER.KZ_Umschreibung = TRUE
Dann AUFTR.KZ_Eintragung = ‚1'
Sonst AUFTR.KZ_Eintragung = ‚0'

Nach Ermitteln der Bedingungen und Regeln würde die Testfallmatrix wie folgt aussehen:

Regeln

Bedingungen	R1	R2	R3	R4	R5	R6	R7	R8
Registrierung übernehmen	J	J	J	J	N	N	N	N
Geos/Geos Schnittstelle	J	J	N	N	N	N	J	J
Umschreibung	J	N	J	N	N	J	N	J
Aktionen								
AUFTR.KZ_Eintragung = ‚1'					X		-	
AUFTR.KZ_Eintragung = ‚0'	X		X			X		-
Versenden und Ausführung	X							
Unlogisch		X		X				

Testfälle

Abbildung 3.8: Testfalltabelle – Registrierungen von Kapitalmaßnahmen

Dadurch, dass sich die ‚Geos/Geos Schnittstelle' nur mit ‚Registrierung übernehmen' auswirkt, können Kombinationen ohne die ‚Oberbedingung' weggelassen werden, da diese irrelevant sind (-), die unlogischen Testfälle müssen getestet werden, da hier unter Umständen Ausnahmesituationen provoziert werden könnten.

Anmerkung: Dieses Testbeispiel wird innerhalb dieser Arbeit an mehreren Stellen wiederkehren und soll auch als Beispiel für die entwickelten Parser verwendet werden.

Kriterien und Metriken für das Testende

Jede Bedingung muss einmal TRUE oder FALSE gesetzt werden, jede Aktion soll einmal aufgerufen werden, jede Postcondition soll einmal erzeugt werden. Bei Relationen sollte jeder Wert repräsentativ einmal angenommen werden.

3.5.2 Zustandsbezogenes Testen

Bei vielen Systemen muss die Reihenfolge der Eingaben berücksichtigt werden, die Systeme nehmen verschiedene Zustände ein. Beginnend mir einem Startzustand können verschiedene Zustände angenommen werden, die irgendwann in einem Endzustand terminieren. Zustandsänderungen werden durch Ereignisse ausgelöst. Die möglichen Zustände und Zustandsänderungen (Übergänge) werden in einem Zustandsmodell beschrieben [GERK02].

Zustandsmodelle können wie unter Kapitel Funktionstest/Zustandstest beschrieben nur die möglichen Zustände und deren Übergänge zeigen, sie können um Aktionen, die Zustandsveränderungen zur Folge haben, erweitert werden, oder um Ereignisse, die sowohl Zustände verändern als auch Aktionen auslösen.

Fallbeispiel: Ableitung über Transitionen

Als Beispiel soll ein Phasenmodell gezeigt werden. In der Finanzsoftware GEOS durchläuft jedes Geschäft mehrere fachliche Phasen, und diese eignen sich gut, um mittels eines Zustandsdiagramms dargestellt zu werden.

Es wird ausgehend von einem Weiterleitungsauftrag ein fachlicher Schluss (SLUSS) gebildet und die Kontrahentenausführung (KTAUS) angelegt. Dann wird die Kundenausführung (KDAUS) anhand der Kontrahentendaten und der Auftragsdaten generiert. Dann durchläuft jede der Ausführungen (AUSFG) die Phasen gültig (GULT), abgestimmt (ABGE), depotgebucht (DEBU), und abgeschlossen (ABSO). Der Schluss spiegelt dabei die jeweils niedrigste Gesamtphase wider und diese in der Reihenfolge

unstimmig (UNST), stimmig (STIM), depotgebucht (DEBU), abgeschlossen (ABSO).

Stimmt die Ausführung nicht mit dem zugrunde liegenden Auftrag überein, bleibt sie gültig, ohne Freigabe bleibt die Phase abgestimmt, danach erfolgt die Depotbuchung. Bei einem Fehler ist die Phase fehlerhaft depotgebucht (FEDE), und darauf folgt die Abrechnung. Sollten beispielsweise Kurse wegen der Abrechnung fehlen, bleibt die Phase depotgebucht (DEBU). Konnten beide Ausführungen abgerechnet werden, ist der gesamte Schluss daraufhin in der Phase abgeschlossen.

Zustandstabelle der Objekte SLUSS und AUSFG:

SLUSS	UNST	STIM	DEBU	ABSO
UNST		X		
STIM			X	
DEBU				X
ABSO				

AUSFG	GULT	ABGE	FEDE	DEBU	ABSO
GULT		X			
ABGE			X	X	
FEDE				X	X
DEBU					X
ABSO					

Abbildung 3.9: Zustandstabelle für Ausführungs-/Schlussphasen

Die Zustände und ihre möglichen Übergänge sind für sich selbst genommen nicht sehr aussagekräftig, sie werden weiters mittels einer Zustands-Transitions-Tabelle (*State Transition Diagram*) [BIND05] um Aktionen und / oder Ereignisse erweitert.

Vorbedingungen: Kundenauftrag muss angelegt und disponiert worden sein, und ein Weiterleitungsauftrag muss generiert und versendet worden sein.

Retourncodes: TRUE = 1, FALSE = 0

State			**Event**	**Action**	**Next State**		
SLUSS	AUSFG (KT)	AUSFG (KD)			SLUSS	AUSFG (KT)	AUSFG (KD)
UNST	-	-	KTAUS anlegen	KDAUS generieren	UNST	GULT	GULT
UNST	GULT	GULT	KT. Abst. 0	Abstimmfehler	UNST	GULT	GULT
UNST	GULT	GULT	KT. Abst. 1	KTAUS abstimmen	UNST	ABGE	GULT
UNST	ABGE	GULT	KD. Abst. 0	Abstimmfehler	UNST	ABGE	GULT
UNST	ABGE	GULT	KD.abst. 1	KDAUS abstimmen	STIM	ABGE	ABGE
			Freigabe				
STIM	ABGE	ABGE	KT.buchen 0	Verarbeitungsfehler	STIM	FEDE	ABGE
STIM	ABGE	ABGE	KT buchen 1	Depotbuchung	STIM	DEBU	ABGE
STIM	DEBU	ABGE	KD buchen 0	Verarbeitungsfehler	STIM	DEBU	FEDE
STIM	DEBU	ABGE	KD buchen 1	Depotbuchung	DEBU	DEBU	DEBU
DEBU	DEBU	DEBU	KT rechnen 0	Fehlermeldung	DEBU	DEBU	DEBU
DEBU	DEBU	DEBU	KT rechnen 1	Abrechnung	DEBU	ABSO	DEBU
DEBU	ABSO	DEBU	KD rechnen 0	Fehlermeldung	DEBU	ABSO	DEBU
DEBU	ABSO	DEBU	KD rechnen 1	Abrechnung	ABSO	ABSO	ABSO

Abbildung 3.10: Zustandsübergangstabelle für Ausführungs-/Schlussphasen

Jede Phase muss durchlaufen werden können. Die Tabelle wird um eine Spalte ‚Testaktion' erweitert, die den tatsächlichen Unterschied der Phasen bewirken soll.

State			Event	Testaction	Action	Next State		
SLUSS	AUSFG (KT)	AUSFG (KD)				SLUSS	AUSFG (KT)	AUSFG (KD)
UNST	-	-	KTAUS anlegen	**Anlegen**	KDAUS generieren	UNST	GULT	GULT
UNST	GULT	GULT	KT ab-stimmen 0	**Verwahr-daten ändern**	Abstimm-fehler	UNST	GULT	GULT
UNST	GULT	GULT	KT ab-stimmen 1	**-**	KTAUS abstimmen	UNST	ABGE	GULT
UNST	ABGE	GULT	KD ab-stimmen 0	**Menge ändern**	Abstimm-fehler	UNST	ABGE	GULT
UNST	ABGE	GULT	KD ab-stimmen 1	**-**	KDAUS abstimmen	STIM	ABGE	ABGE
			Freigabe					
STIM	ABGE	ABGE	KT buchen 0	**Update Position**	Verarbeitung sfehler	STIM	FEDE	ABGE
STIM	ABGE	ABGE	KT buchen 1	**-**	Depot-buchung	STIM	DEBU	ABGE
STIM	DEBU	ABGE	KD buchen 0	**Update Position**	Verarbeitung sfehler	STIM	DEBU	FEDE
STIM	DEBU	ABGE	KD buchen 1	**-**	Depot-buchung	DEBU	DEBU	DEBU
DEBU	DEBU	DEBU	KT rechnen 0	**KZ_Abrech-nung = 0**	Fehler-meldung	DEBU	DEBU	DEBU
DEBU	DEBU	DEBU	KT rechnen 1	**-**	Abrechnung	DEBU	ABSO	DEBU
DEBU	ABSO	DEBU	KD rechnen 0	**Devisen-kurs fehlt**	Fehler-meldung	DEBU	ABSO	DEBU
DEBU	ABSO	DEBU	KD rechnen 1	**-**	Abrechnung	ABSO	ABSO	ABSO

Abbildung 3.11: Zustandsübergangstabelle für Ausführungs-/Schlussphasen und Aktionen

Kriterien und Metriken für das Testende

Jede Phase / Zustand muss mindestens einmal durchlaufen worden sein. Ein zustandsbezogener Test sollte bei jedem Zustand alle bei diesem Zustand spezifizierten Funktionen mindestens einmal zur Ausführung gebracht haben.

3.5.3 Datenbezogener Test

Definition

Ein datenbezogener Testfall soll demonstrieren, wie der Testgegenstand auf seine Eingaben reagiert. Betrachtet werden Definitionen, sowie lesende Zugriffe und schreibende Zugriffe von Variablen, den so genannten Defs/Uses-Verfahren [ZELL06]:

- **def**: Wertzuweisung, auch Definitionen, Initialisierung
- **c-use**: berechnende Benutzung (*computational use*)
- **p-use**: prädikative Benutzung (*predicate use*)

Als Grundlage kann neben CMF Texten und Tabellen die UML Methodik des Kontrollflussgraphs mit Datenflussdarstellung gewählt werden.

Für die **Ableitung** der damit verbundenen Testfälle müssen die Inputdaten bereitgestellt werden und anhand von definierten Überleitungen die Sollergebnisse erstellt werden. In der Praxis stellen die zu testenden Objekte Schnittstellen und Dialoge dar.

Schnittstellen

Wenn das **Testobjekt** eine **Schnittstelle** darstellt, dann werden Daten dabei über eine definierte Schnittstelle geschickt, Input kann die Benutzeroberfläche, eine spezielle Datei oder der Output eines Prozesses sein.

Getestet wird, ob die empfangenen Daten den gesendeten Daten ‚entsprechen'. Das kann sein, dass diese 1:1 übermittelt worden sind, oder dass sie korrekt konvertiert worden sind. Konvertierung kann sowohl das Format, als auch fachliche Umschlüsselung des gleichen Formats sein. Format: Speichern der Daten auf der Attributsebene, im Varchar Format, binäres oder spezielles Format des Datenträgers.

Beim *Interface* –Test gibt es viele Möglichkeiten der Definition von **Testfallobjekten:**

- Vergleich der Attributswerte in den Datenbanken mittels einer Überleitungstabelle.
- Prüfung der Dialogbefüllungen.
- Prüfung der Umschlüsselung von Währungen mit Datenbanksurrogaten.
- Befüllung von Schlüsselverzeichnissen und Werten.
- Erfolgen die Verknüpfungen und Bezeichnungen über Entitäten und Surrogate richtig.
- Werden Löschregeln (CASCADE, RESTRICT, …) eingehalten.
- Prüfregeln auf Vorhandensein, Nichtvorhandensein, Domänenverletzungen, Muss/Kannfeld Prüfungen.
- Sind online oder auf der Datenbank Werte möglich, die die Domäne verletzen (Long Integer auf Double).
- Prüfen, ob bei Neuanlage die jeweils abhängigen Entitäten richtig befüllt werden.

Beispiel für Konvertierungen: Stammdaten werden über den WM – Meldedienst über die Schnittstelle nach GEOS geschickt und in ‚Geos-Format' konvertiert. Hierbei sind aufwendige Tests wieder notwendig, da vor allem fachlich gewährleistet werden muss, dass diese Daten richtig umgeschlüsselt werden. Für die Tests stellen Überleitungsmatrizen o.ä. die Grundlage dar.

Beispiel für fachliche Umschlüsselungen: Der servicierende Mandant schickt über die Schnittstelle von seinem Echtdepot der Lagerstelle seines Depotmandanten Geschäfte. Diese müssen um die unrelevanten Informationen bereinigt und mit den entsprechenden Lagerstelleninformationen versehen werden. Fremde Spesen und Steuern werden mitgeschickt und müssen mit umgekehrtem Vorzeichen abgerechnet werden, usw.

Wenn die Schnittstellendaten weiter verwendet werden können, dann ist **Testfallermittlung aus der Schnittstelle** möglich. Wenn Geschäftsdaten über eine

Schnittstelle ins System gelangen, können nicht nur die fachlichen und technischen Prüfungen der Schnittstelle getestet werden, gleichzeitig können die generierten Datensätze als Basis für weitere z.B.: funktionale Tests herangezogen werden. Schnittstellen stellen also hierbei ein geeignetes Mittel zur Massentestfallgenerierung dar. Hier ist der Einsatz von Generierungstools möglich, wie später noch mittels praktischer Beispiele demonstriert werden soll.

Dialogtest

Neben dem Schnittstellentest sind des Weiteren vor allem GUI – Oberflächen geeignete Objekte für den Datenflusstest: Mit dem **Testobjekt** eines **Dialogs** (eine GUI-Eingabemaske) soll ein typischer CMF – Testansatz beispielhaft an einem Auftragsdialog gezeigt werden.

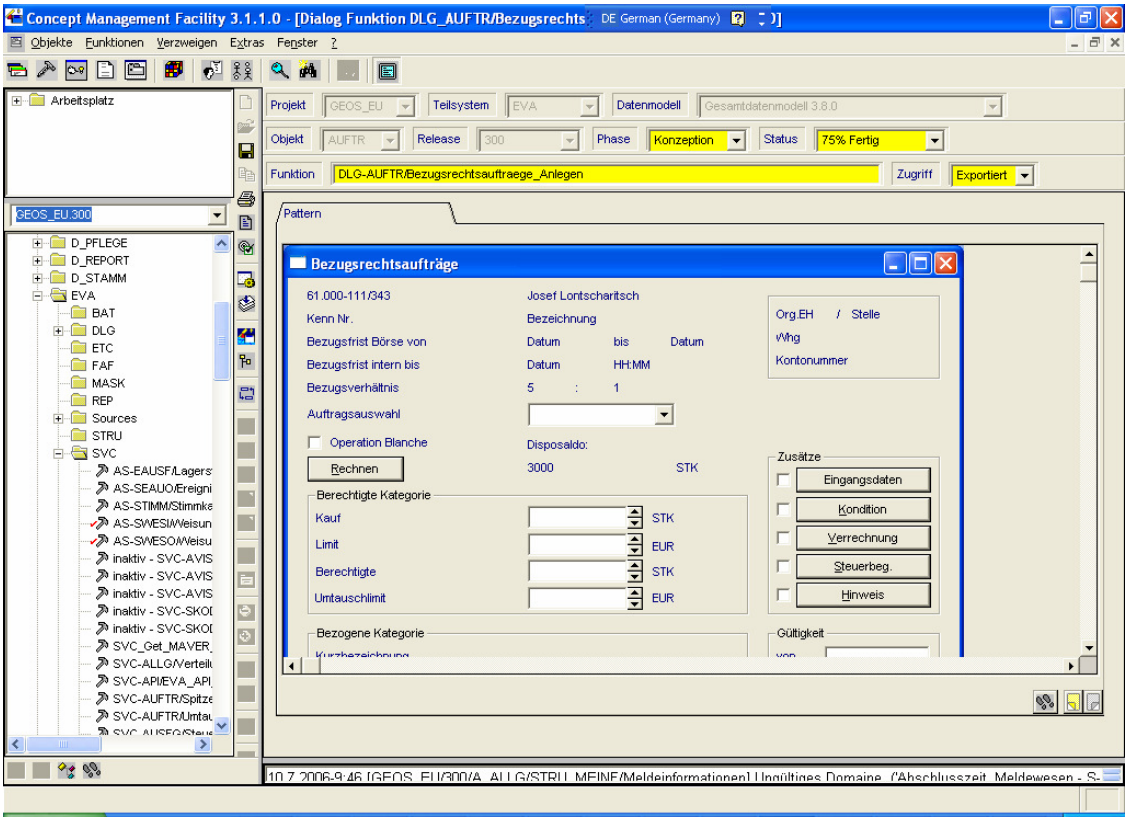


Abbildung 3.12: Bezugsrechtsauftragsdialog über CMF

Fachliche Kurzbeschreibung:

Diese Funktion dient zur Schnellanlage von Bezugsrechtsaufträgen und kann entweder aus dem Hauptobjektrahmen Auftrag oder Hauptobjektrahmen Position aufgerufen werden.
In diesem Dialog können jene Aufträge in einem Arbeitsschritt angelegt werden, die am Aviso dem Kunden vorgeschlagen werden. Voraussetzung für die Erfassung der Aufträge ist das Vorhandensein einer freiwilligen Kapitalmaßnahme mit Bezugsrecht. Stehen mehrere bezogene Kategorien zur Auswahl, so kann die Auswahl in einem vorgelagerten Dialog vorgenommen werden.
In der Auftragsauswahl kann festgelegt werden, welche Aufträge erfasst werden sollen.
Folgende Möglichkeiten stehen zur Auswahl:
- Kauf
- Verkauf
- Halten
- Bezug
- Kauf und Bezug
- Verkauf und Bezug
- Halten und Bezug.

Für den Bezug kann die beauftragte Menge sowohl als Menge der BZR angegeben werden, die jeweils andere Menge wird auf Grund des Umtauschverhältnisses in dem entsprechenden Mengenfeld automatisch eingetragen.
Weiters kann eine Operation Blanche (Bezug mit geringstem Kapitaleinsatz) berechnet werden.
Die Bearbeitung der einzelnen Aufträge ist nur über die Standardauftragsdialoge (Handel, Umtausch/Bezug) möglich.

Für folgende Testobjekte sollen Testfälle erstellt und als erster Arbeitsschritt Äquivalenzklassen definiert werden:

Depotnummer:

Negativ	-3
String	Depottests
Überlang	12345678901234567890
Gültig	61130300604

Wertpapier

Wertpapier_exisitiert=false	WKN / 0000001
Bezugsrecht=false	WKN / 071600
Bezugsrecht=true	WKN / 035630

Auftragsauswahl

Kauf (Kf), Verkauf, Halten, **Bezug (B)**, Kauf und Bezug, **Verkauf und Bezug (VB)**, Halten und Bezug. (Nur die speziell markierten Aufträge sollen getestet werden).

Kaufmenge / Umtauschmenge /Handelslimit / Umtauschlimit

Negativ, String, Überlang, Gültig, 0

Operation Blanche

Nicht gesetzt, Gesetzt, Setzen ohne Geschäftsart ‚Verkauf und Bezug'

Gültigkeiten

Gilt_von > Tagesdatum

Gilt bis Datum < Tagesdatum

Gilt_von > Gilt_bis

Nachdem Depotnummer und Wertpapier für die Initialisierung entscheidend sind, sollen die Testfälle nach Initialisierungskriterien und Verarbeitungskriterien unterteilt werden. Des Weiteren können folgende dem CMF entnommene Plausibilitäten getestet werden. Für dieses Beispiel werden die 4 besonders hervorgehobenen getestet:

Initialisierung:		Verarbeitung:	
Titel_ungueltig	**5194**	Bezugsmenge_nicht_Zaehler_teilbar	5077
Disposaldo_zu_klein	5299	Bis_groesser_Von	9208
nicht_kleinste_Stueckelung	5137	von_in_Vergangeneit	23005
Depot_nicht_vorhanden	**8439**	groesser_Maximum	5079
kein_Recht	**2552**	kleiner_Minimum	3351
interne_Bezugsfrist	5385	Bezugsfrist_zukunft	5069
Notierung_vorhanden	5084	keine_Kapitalmassnahme	3298
OB_zu_wenig_BZR	5085	letzter_handelstag	5285
Datum_Von_Bis	9208	Bezugsfrist_abgelaufen	5081
Depotnummer_ungueltig	**4399**		

Abbildung 3.13 zeigt wie die Initialisierung des Dialogs anhand einer Variation der Inputvariablen Depot und Wertpapier, die über den Hauptobjektrahmen eingegeben werden müssen, getestet werden kann.

Die **Testfälle der Dialoginitialisierung** sind:

Attribut	**Werte / Relationen**						
Depotnummer	- 3	Depottests	1234567890 1234567890	6000	61230300604	61230300604	61230300604
Wertpapier					0000001	071600	035630
Aktion	**Fehlermeldung / Postcondition**						
Initialisierung							X
FM			4399				
FM				8439			
FM					5194		
FM						2552	
Nicht möglich	X	X					

Abbildung 3.13: Testfalltabelle für Dialoginitialisierung

Nach der Dialoginitialisierung kann der Test mit der Dialogmaske ‚Bezugsrechtsaufträge' fortgeführt werden. Die für diesen Testfall gesammelten Vorbedingungen sind wie folgt, und werden mit Testbedingungen in Abbildung 3.14 erweitert und durchgeführt:

Preconditions: Depot 61230300604,
Wertpapier 035630
Dialoginitialisierung
Umtauschverhältnis: 3:1
Bezugsfrist = 30.06.
Handelsfrist = 29.06

Testfälle für Auftragsauswahl: Bezug

Input																	
Auftragsauswahl	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B
Kaufmenge	-	-	-	-	300	-	-	-	-	-	-	-	-	-	-	-	-
Umtausch menge	-300	Test	10^{30}	0	10	10	10	10	10	10	10	10	10	10	10	10	10
Handelslimit	-	-	-	-	-	-	-	-	-	-	-	3					
Umtauschlimit	-	-	-	-	-	-	- 3	Test	10^{30}	0	3						
Operation Blanche	-	-	-	-	-	-	-	-	-	-	-	-	X				
Gültigkeit Handel	-	-	-	-	-	-	-	-	-	-	-			X			
Gültigkeit Umtausch	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Gilt_von > Tagesdat.															X		
Gilt_bis < Tagesdat.																X	
Gilt_von > Gilt_bis																	X
Output																	
Ungültig	X	X	X	X	X	-	X	X	X	X		X	X	X	X	X	X
Bezogene	?	?	?	?	?	100	-	-	-	-	100	-	-	-	-	-	-
PB_Rechnen	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Gültigkeit						30.6.					30.6.						
Auftrags-anlage						X					X						

Abbildung 3.14: Testfalltabelle für Dialogtest (Bezug)

Dieses Testfallset wurde mittels Äquivalenzklassen gebildet. Umtausch- und Handelsmenge, Umtausch- und Handelslimit wurden mit den Repräsentanten -300, Test, 10, 1030 und 0 gebildet. Die Grenzwerte bleiben innerhalb dieser Suite ungetestet, und müssten noch mit ihren Domänengrößen getestet werden. Beispiel ist die Domäne der Kaufmenge *long integer* (−2.147.483.648, 2.147.483.647 und die umliegenden Werte)

Kombiniert wird das Testfallset schon komplizierter, wenn Verkauf und Bezug getestet werden sollen. Darüber hinaus soll noch die *Operation Blanche* getestet werden, d.h. es sollen möglichst so viele Bezugsrechte verkauft werden, sodass damit der Umtausch finanziert werden kann, und somit keine Kosten mehr entstehen. Abbildung 3.15 demonstriert die Testfälle für Auftragsauswahl: Verkauf und Bezug.

Die neuen Preconditions sind:

Bezugspreis = 3, Kurs des Bezugsrechts = 3.5, Positionsmenge = 600

Testfälle für Auftragsauswahl: Verkauf und Bezug

Input													
Auftragsauswahl	VB	VB	VB	VB	VB	VB	VB	VB	VB	VB	VB	VB	VB
Positionsmenge	-	-	-	-	-	-	-	-	-	-	-	-	600
Verkaufsmenge	-	0	-	-	-	300	300	-	-	-	-	-	-
Umtausch Menge	-300 Test 10^3 0	30	30	30	30	30	30	30	30	30	30	30	-
Handelslimit	-	-	-	-	-	3	-	-	-	-	-	-	-
Umtauschlimit	-	-	-	- 3 Test 10^3 0	3								
Operation Blanche	-	-	-	-	-	-							X
Gültigkeit Handel	-	-	-	-	-		X						
Gültigkeit Umtausch	-	-	-	-	-	-	-	-	-	-	-	1.7.	-
Gilt_von > Tagesdat								X					
Gilt_bis < Tagesdat									X				
Gilt_von > Gilt_bis										X	X		
Output													
Ungültig	X		-	X			X	X	X	X	X	X	X
Berechtigte	-	10	10	-			-	-	-	-	-	-	323
Handel		300				300						-	277
Bezogene	-	10	10	-	10	10	-	-	-	-	-	-	107,67
PB_Rechnen	-	-	-	-	-	-	-	-	-	-	-	-	-
Gültigkeit Handel	-	29.6.	29.6.	-	29.6.	29.6.	-	-	-	-	-	-	29.6.
Gültigkeit Umtausch	-	30.6.	30.6.	-	30.6.	30.6.	-	-	-	-	-	-	30.6.
Auftragsanlage		X	X		X	X							X

Abbildung 3.15: Testfalltabelle für Dialogtest (Verkauf und Bezug)

Darüber hinaus sollten bei Dialogtests jedes Control, jeder Fokus, Rechenoperationen, Mnemonics (Shortcuts) etc., mehrmals angewählt, durchgeführt und wieder verlassen werden, um das interaktive Dialogverhalten genauer bestimmen zu können.

Kriterien und Metriken für das Testende

Die Definitionen, Berechnungen und Prädikatsfunktionen können wie folgt auf die Vollständigkeit der Durchführung gemessen werden [ZELL06]:

- *all defs*: jede Definition muss in einer Berechnung oder Bedingung benutzt werden (identifiziert keine Kontrollflussfehler).
- *all p-uses*: jede Kombination aus Variablendefinition und deren prädikative Nutzung muss getestet werden (identifiziert sicher Kontrollflussfehler wegen der impliziten Zweigüberdeckung).
- *all c-uses*: jede Kombination aus Variablendefinition und deren berechnende Benutzung muss getestet werden (identifiziert Berechnungsfehler).
- *all uses*: sind alle *c-uses* und *p-uses* zusammen und identifizieren laut Gerke insgesamt ca. 3/4 der Programmierfehler [GERK02].

3.5.4 Ablaufbezogener Test

In seiner Ablaufstruktur besteht ein Programm aus einzelnen ausführbaren Anweisungen und deren dynamischen Beziehungen zueinander. Dargestellt wird die Struktur durch einen gerichteten Graphen. Darin bilden die Befehle die Knoten und die Beziehungen die Kanten. Dort, wo Befehle unbedingt aufeinander folgen, bilden sie eine Sequenz, d.h. wird ein Befehl ausgeführt, werden alle ausgeführt. Im Ablauf wird dies als eine Kante dargestellt. Dort, wo die Ablaufflusslinie sich in zwei oder mehr vorwärts gerichtete Linien trennt, wird eine Auswahl getroffen, welcher Weg einzuschlagen ist. Im Ablaufgraphen bilden jene Entscheidungen die Knoten. Dort, wo eine Ablaufflusslinie rückwärts verzweigt bzw. eine Kante zum Ausgangspunkt zurückkehrt, handelt es sich um eine Wiederholung bzw. um eine Schleife [SNWI02].

Anweisungen sind Befehle der jeweiligen Programmiersprache wie z.B. *WHILE, DO, IF, THEN, ADD, READ, PUT*, usw.

Zweige sind die Kanten im Ablaufgraphen, Zweige enthalten in der Regel mindestens

eine Anweisung, aber nicht immer. Z.B. aus einem IF ohne ELSE gehen zwei Zweige hervor, einer mit den anzuführenden Anweisungen und ein Leer-Zweig, der implizite Sonst-Zweig. Deshalb ist der Test aller Zweige stärker als der Test aller Anweisungen, denn dadurch werden beim Zweigtest alle Anweisungen mit getestet, nicht aber umgekehrt [SNWI02].

Pfade sind dynamische Folgen von Ablaufzweigen bzw. eine einmalige Kombination von Zweigen, oder anders ausgedrückt: „Ein Pfad ist eine endliche nicht leere Folge aufeinander folgender Kanten“ [GERK02]. Ein Pfad heißt vollständig, wenn seine erste Kante von einem Startknoten ausgeht und seine letzte Kante zu einem Endknoten führt. Je mehr Verzweigungen in der Ablauflogik, desto mehr Ablaufpfade sind zu testen. Gültige Testfälle müssen gültige Pfade erzeugen

Die Testfälle für einen ablaufbezogenen Test, sind im Grunde genommen ein Spiegelbild der Bedingungen im Testobjekt und lassen sich aus demselben ableiten.

Wie soll die **Anzahl der notwendigen Testfälle** bestimmt werden. Bei simplen Diagrammen, Datenflussdiagrammen und Prozessflussdiagrammen ist es in den meisten Fällen die Anzahl der Endknotenpunkte. Die Pfade, die diese Endpunkte anlaufen, sind jeder für sich genommen ein Prozesstestfall. Bei komplexeren Kontrollflussdiagrammen sollte jedoch wieder von der Anzahl der Bedingungen ausgegangen und mittels Prädikatenlogik die Testfälle ermittelt werden.

Abbildung 3.16 zeigt anhand des Beispiels ‚Registrierungen von Kapitalmaßnahmen’ ein Diagramm, aus dem die Testfälle anhand der Pfad- und Endknotenbestimmung ermittelt werden können.

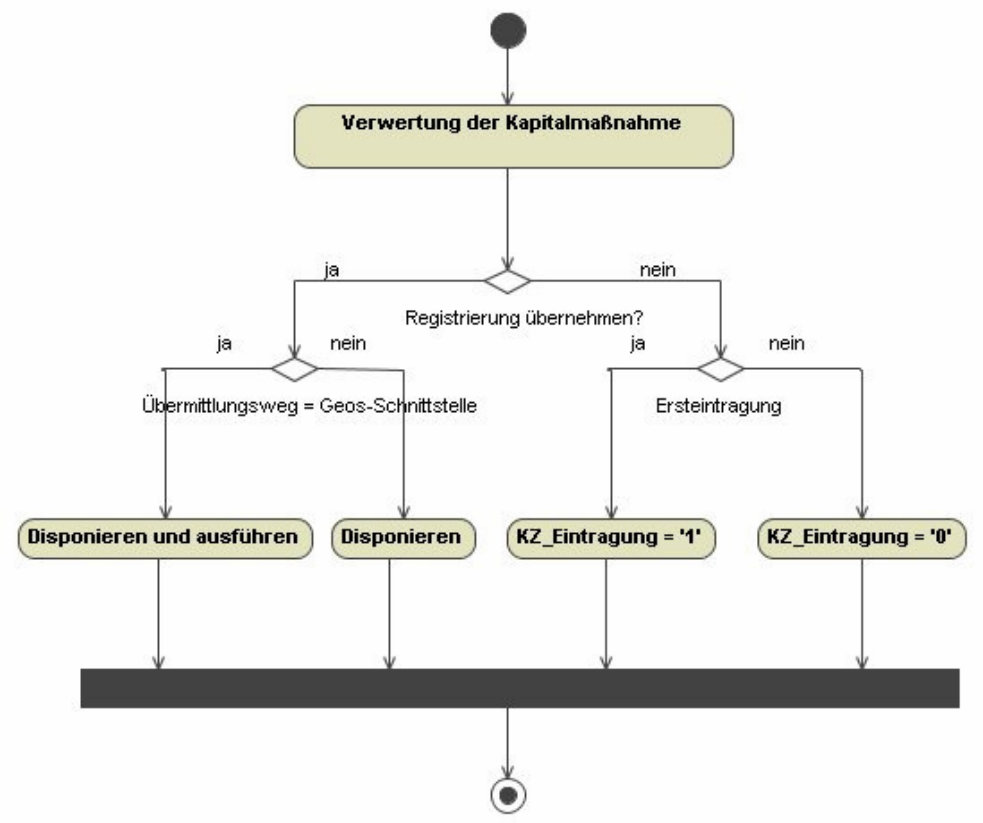


Abbildung 3.16: Ablaufdiagramm – Registrierungen von Kapitalmaßnahmen

Die Unterscheidung zwischen Prozessfluss, Kontrollfluss und Datenfluss ist oftmals nicht sehr einfach und geht aus der Datenverwendung hervor. Prozessfluss- und Datenflusstestdiagramme ergeben die Testfälle meistens aus den Endpunkten, bei Kontrollflussdiagrammen sollten Ableitungen über Regeln erfolgen wie oben dargestellt.

Der nachstehende Testfall in der Abbildung 3.17 verfolgt diesen Ansatz. Selbstverständlich kann jeder zu prüfende Pfad ermittelt und getestet werden. Nachdem es sich um ein komplexes Kontrollflussdiagramm mit wenigen Endpunkten handelt, sind die Bedingungen und Aktionen in tabellarischer Form in Abbildung 3.18 gesammelt und mittels Plausibilitätsverfahren um unlogische Pfade bereinigt worden. Diese Methodik stellt ein effizientes und relativ vollständiges Verfahren dar, wenn in dem Kontrollfluss wenig bis gar keine Schleifen durchlaufen werden sollen.

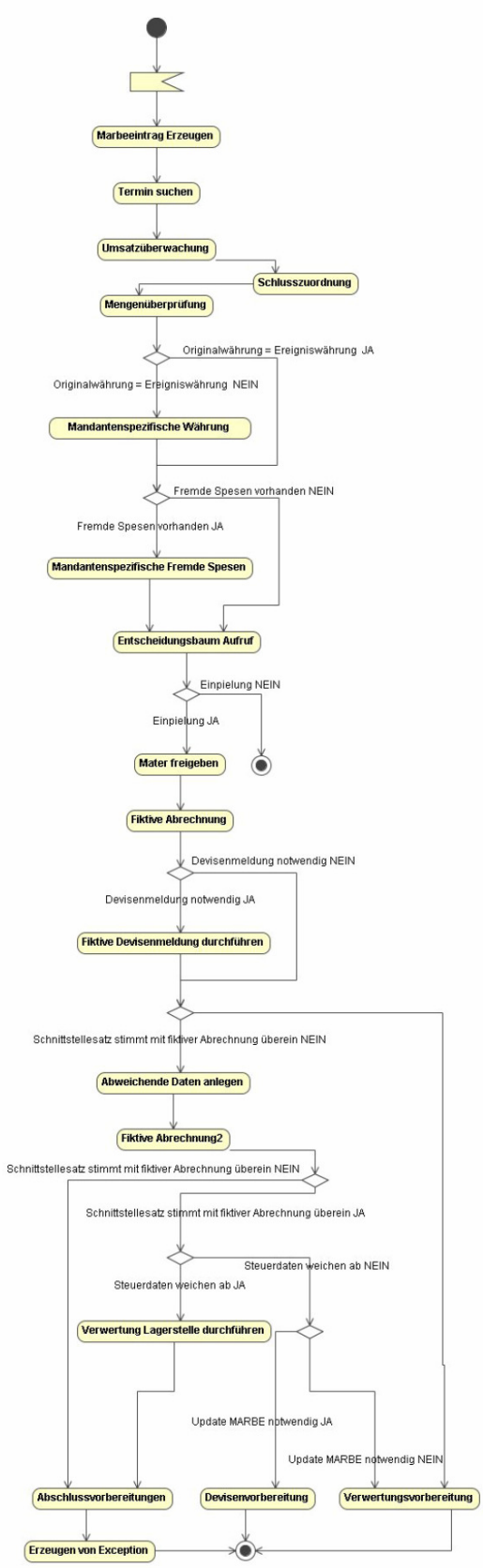


Abbildung 3.17: Ablaufdiagramm für Lagerstellenverwertung

In diesem **Fallbeispiel** soll ein SWIFT MT 566 IN Schnittstellensatz nach GEOS eingespielt werden, und je nach Übereinstimmung der Währungen, etc. sollen Zusatzdaten von GEOS angelegt und für die Verwertung herangezogen werden. In einer Schleife wird das Ergebnis mehrmals überprüft. Dies stellt einen wesentlichen Teil des STP-Prozesses dar und muss gründlich getestet werden. Aufgrund der Bedingungen des Diagramms in Abbildung 3.17 ergäben sich 64 Testfälle. (Anmerkung: Aufgrund der Komplexität sind die Steuerbedingungen bei diesem Testset weggelassen worden). Die logischen Testfälle sind unabhängig von ihrer Relevanz oder Gültigkeit in der Entscheidungstabelle in Abbildung 3.18 generiert worden, um die Vollständigkeit in weiterer Folge gewährleisten zu können. Danach werden überflüssige Testfälle entfernt. Da ein negativer iRc des Entscheidungsbaums zu einem Testfallende führt, können alle Testfälle mit Variationen nach dem Entscheidungsbaum entfernt werden. Des Weiteren gibt es unlogische Kombinationen, die gelöscht werden können: Devisenmeldung trotz gleicher Währung, Keine Meldung trotz ungleicher Währung, Fiktive Abrechnung passed und Abrechnung failed.

Originalwährung = Ereigniswährung	J	J	J	J	J	N	N	N	N	N	N	N	N	N	J
Fremde Spesen	J	J	J	N	N	J	J	J	J	N	N	N	N	N	J
Entscheidungsbaum	J	J	J	J	J	J	J	J	J	J	J	J	J	J	N
1. Fiktive Abrechnung	J	N	N	N	N	J	N	N	N	N	N	N	J	N	J	-
Devisenmeldung	N	N	N	N	N	J	J	N	J	N	N	J	J	J	J	-
2.Abrechnung	J	N	J	N	J	J	N	J	J	N	J	J	J	N	J	-
MATER.Bruttobetrag / Waehrung						X	X	X	X	X	X	X	X	X	X	-
MATER.Fremde Spesen	X	X	X			X	X	X	X							-
MATER ist gültig	X															-
Schritte sind freigegeben	X		X		X	X		X		X	X					-
Anspruch ist verwertet	X		X		X			X		X	X					-
Devisenschnittstelle asynchron						X	X		X			X	X	X	X	-
Verarbeitung nicht durchführen	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	X

Abbildung 3.18: Testfalltabelle für Lagerstellenverwertung

Beim Test von Schleifen gibt es verschiedene Ansätze und Methoden. Laut Thaller [THAL02] muss so getestet werden, dass die Schleife 1. nicht ausgeführt wird, 2. mit einem niederen Werte des Schleifenzählers getestet wird, und 3. mit einem hohen Wert des Schleifenzählers getestet wird. Bei mehreren Schleifen müssen diese geordnet werden. Die erste Schleife wird wie oben beschrieben getestet, während die anderen mit Defaultwerten versorgt werden. Mit den weiteren Tests werden die bereits getesteten mit ‚Referenzwerten' versorgt und zur Precondition und Fixparameter erklärt.

Kriterien und Metriken für das Testende

Hier können die oben erwähnten C- Überdeckungsmaße (C0, C1, C2 - Überdeckung) ihre Anwendung finden. Je nach gewählter Metrik wird jede Anweisung, jeder Knoten oder jeder Pfad getestet. Der Autor empfiehlt für dieses Beispiel die 100%ige Pfadüberdeckung durchzutesten.

3.6 Testdaten

Inputdatei

Testfälle können mit Werten über eine Inputdatei versorgt werden. Siemens beispielsweise nutzt mit seinem Testtool IDATG [SIEM06] ein Framework, in dem nur konkrete Daten den logischen Parametern zugewiesen werden. Die Testfälle, die generiert werden, werden zur Laufzeit mit den gespeicherten Werten versorgt, entweder in einer bestimmten Reihenfolge, per Zufall oder mittels fester Zuweisung.

Select über die Datenbank

Wird mit Datenbanken gearbeitet, wäre es möglich, die Testfälle mittels SQL-*Select* [SAUE98] auf der Datenbank mit Werten zu versorgen. Man könnte beispielsweise die Extremwerte, die häufigsten Werte oder zufällig gewählte Werte einer Domäne verwenden.

Zufallsgenerator

Für fachlichen Test eignen sich Zufallsgeneratoren nur bedingt. Steht jedoch bei einem Test der technische Aspekt im Vordergrund, können Testfälle mit Randomdaten generiert werden. Beispielweise für die Reorganisation von Tabellen, das später noch beispielhaft erklärt werden soll.

Baseline

Das Speichern von Testdaten, die wieder verwendet werden sollen, ist vor allem für Regressionstests wichtig. Die SDS bedient sich der *Baseline*, in der ein Stammdatenbestand gespeichert wird. Die Baseline ist prinzipiell ein ‚normales' System mit dem Datenbankstand der letzt gelieferten Version, auf der niemals Bewegungsdaten gespeichert werden sollen. Ist nun ein Testdurchlauf fertig gestellt worden, kann die Baseline jederzeit neu aufgesetzt und auf prinzipiell jeder Systemumgebung neu eingespielt werden.

3.7 Testtemplate

Nun sollen Testfallbestandteile und ein Format für die Verwaltung von Testfällen untersucht werden.

3.7.1 Bestandteile des Testfalls

Jede Testsuite sollte die folgenden Projektinformationen beinhalten [MAVE06]:

Projektname: (Sprechender) Name der Testsuite

Projektkürzel: Basis für die Testfallkürzel

Datum: Erstellungsdatum

Autor: Ersteller des Projekts

Beschreibung: Übersichtliche Ablaufbeschreibung

Testobjekte: Skizzierung der Softwareteile, auf denen der Testfokus liegt

Testgrundlage: Testentwurfsspezifikation, Anforderung, Fachkonzeptspezifikation,…

Umgebung: Systemumgebung inkl. Version, Plattform, Frontend,…

Preconditions: wenn sie für die gesamte Testsuite gelten

Die Detailinformationen jedes Testfalls der Testsuite:

Nummer: Projektkürzel und fortlaufende Nummer

Priorität: niedrig / mittel / hoch

***Preconditions*:** (Test-) unterteilt nach Kriterien z.B.: Termine, Weisungen, Stammdaten

***Aktionen*:** in der gewünschten Reihenfolge

Erwartetes Ergebnis: dieses sollte der Testgrundlage entnommen werden können

***Istergebnis*:** getestetes Ergebnis

Systemumgebung: Plattform, Version

***Status*:** passed / failed / ungetestet

Besonderheiten: wenn sie nur für diesen Testfall gelten sollen

Abhängigkeiten: von anderen Testfällen

Dies entspricht im Wesentlichen dem IEEE 829 Standard [SOKE06] und ist um Informationen von weiteren Testtemplates erweitert und / oder verfeinert worden [GERK06]. Je nach Übersichtlichkeit oder Informationsvielfalt der Testfälle können die Testfallinformationen in einer horizontalen Form dargestellt und die Details der Testfälle in Spalten unterteilt werden.

3.7.2 Testfallbeispiel

Als Beispiel soll der Prozess eines Produkts der SDS gezeigt werden. In der Terminverarbeitung von *Corporate Actions* können Weisungen angelegt werden und je nach Weisung und Prozessschritt sollen diese Weisungen zu Schlüssen verdichtet werden. Für die Durchführung werden dabei Termine, Wertpapiere und Depots benötigt, diese sind in den jeweiligen Preconditions vermerkt. Sind die Preconditions für die gesamte Testsuite gültig, sollen sie in die Projektinformationen geschrieben werden, wenn sie Bestandteil der Testvariationen sind, dann sind diese Bestandteil der Testfallinformation.

Mögliche **Testfallaktionen** sind:

GEBSWESI:	Batchprogramm, das Weisungen aus der Schnittstelle SWESI (Schnittstelle Weisung In) nach Geos übernimmt.
GEBWEIV:	Batchprogramm, das die Verarbeitung der Paketierung der Ansprüche in den jeweiligen Optionsschluss vornimmt.
GEBSWEIV:	Batchprogramm, das die Paketierungen aufhebt und alle unverarbeiteten Ansprüche wieder in den Initschluss (Urspungsschluss) umhängt.
Online-Paketierung:	Online-Dialog, der Ansprüche (um-)paketiert, gleichzeitig Terminweisungen anlegt und gegebenenfalls die Terminweisung storniert (wenn vorhanden).

Abbildung 3.19 zeigt, wie das ***Testtemplate*** mit Projektinformationen versorgt werden kann, die Testkonditionen variiert werden, und sukzessive die Testaktionen durchgeführt werden sollen.

Projektinformationen / Testsuite									
Projektname	Testprojekt								
Projektkürzel	TST								
Datum	29.06.2006								
Autor	Ferdinand Nest								
Beschreibung	Durchführung von Testfällen nach erfolgter Weisung, Paketieren der Ansprüche und Schliessen der Schlüsse								
Umgebung	Hostumgebung/Linux/Lokal								
Preconditions	Schnittstelle SWESI muss mit Standing Instructions für Wertpapier A befüllt worden sein								
Testfallinformationen / Testcase									
Testfälle	*Priorität*	*Testconditions*			*Aktionen*	*Erwartetes Ergebnis*	*Istergebnis*	*Systemumgebung*	*Status*
		A	*B*	*C*					
TST01	niedrig	Termin A	Weisung A	Depot A	GEBSWESI	Weisung ist angelegt	ok	C30600AS	passed
TST02	mittel	Termin A	Weisung B	Depot C	Online-Paketierung	Weisung ist angelegt, Anspruch ist paketiert, Schluss wurde gemoved	Schluss nicht gemoved	C30600AS	failed
TST03	hoch	Termin A	Weisung C	Depot A	GEBWEIV	Anspruch ist paketiert		C30600AS	ungetestet
TST04	…	…	…	….	…	…	…	…	…

Abbildung 3.19: Testtemplate mit horizontaler Gliederung

Die konkreten Werte der Testconditions werden entweder in einer weiteren Spalte neben jeder Testkondition festgehalten oder in einer eigenen Tabelle gespeichert, je nach Übersichtlichkeit.

3.8 Resumee Funktionaler Test

Der Funktionstest stellt einen wesentlichen Teil zur Qualitätssicherung von Programmen dar. Obwohl dieser für sich selbst genommen nicht ausreicht, um ein Programm ‚fehlerfrei' liefern zu können, ist der Funktionstest in der Lage, solche Fehler zu identifizieren, die mit Unittest u.a. nicht gefunden werden können.

Die Vorteile und Nachteile einander gegenübergestellt sind [LEVI2006]:

Vorteile sind die Identifikation von Programmteilen, die spezifiziert aber nicht implementiert wurden. Der Funktionstest ist unabhängig vom Programm, Testfälle können wieder verwendet werden, und mit der Testfallerstellung kann parallel mit der Entwicklung begonnen werden.

Nachteile stellen die ungenügende Programmabdeckung dar, und dass Redundanzen nicht gefunden werden können.

4. Weitere Quellen für Testfälle

4.1 UML

Die SDS arbeitet derzeit mit einigen verschiedenen UML Diagrammen. Obwohl nur MFL - Schnittstellen und Komponentenarchitekturkonzepte in UML modelliert werden sollen, wird die UML als unterstützendes Werkzeug für die Konzeption der Software verwendet. Für die Modellierung verwendet die SDS Klassendiagramme, Usecases und Ablaufdiagramme. Sequenzdiagramme, Struktogramme und Kollaborationsdiagramme werden praktisch nicht eingesetzt.

Innerhalb dieser Studienarbeit sollen Ablauf- / Aktivitätsdiagramme (Kapitel Ablaufbezogener Test) und UML Use Cases für die Ermittlung von Testfällen eingesetzt und erklärt werden.

Die UML bietet sehr viele Möglichkeiten über die Erstellung der Ablaufdiagramme. Diese können ‚*Stand alone'* Konzepte oder Teil von Use Case Diagrammen sein. Die UML bietet Möglichkeiten die Daten für weitere Verwendung aufzubereiten. Mit dem geeigneten Tool können durchaus Testfälle aus Diagrammen generiert werden (beispielsweise der Mercury Testdirector© [MERC06]). Mit dem UML-Tool Magic Draw© [MAGI06] ist es möglich, die Diagrammbestandteile in Textformat generieren zu lassen, und diese nach Sortierung weiter zu verwenden, um z.B. Testfälle nach den generierten Preconditions erstellen zu können. Des Weiteren können durchaus die XML-Files Grundlage für die Generierung von Testfällen sein, wenn die jeweiligen *Tags* vom *Parser* richtig interpretiert werden können.

4.1.1 Use - Case Modellierung

Use Case Diagramme eignen sich sehr gut, um an neue Problemstellungen heranzugehen, und zu zeigen, wie die verschiedenen User mit dem System interagieren. Wobei in weiterer Folge der User sowohl natürliche Personen, als auch Programmteile, die andere Stellen im Programm aufrufen, umfassen kann.

Es gibt verschiedene Definitionen von Use Cases. In vielen Softwareunternehmen versteht man unter Use Cases logische Testfälle, die z.B. von einem Konzeptionisten verfasst werden und vom jeweiligen Testteam in konkrete Testfälle umgewandelt werden sollen.

Binder beschreibt „Use Cases als eine Abstraktion von Systemreaktionen auf Inputs von außen“ [BIND05], oder allgemein formuliert ist ein Use Case „die Interaktion eines Users mit dem System“. Die einzelnen Objekte eines Use Case Diagramms stehen in einer Beziehung zueinander. Diese Bedingungen können <*extend*>, <*include*>, <*association*>, <*generalization*>, <*dependency*>, <*Package*> sein [ERRI02].

Der folgende Teil gibt Empfehlungen, wie Use Case Diagramme verfasst werden können. Ein Template soll vorgeschlagen werden, das in weiterer Folge durch die geeignete Fragestellung das Ableiten der Testfälle einfacher gestalten soll. Dann soll eine Testfallmatrix erstellt werden, die die notwendigen Variationen der Use cases widerspiegelt.

4.1.2 USE CASE Diagramm

Ein Use Case besteht aus Akteuren und Anwendungsfällen, die von Akteuren angestoßen werden. Akteure können entweder als Strichmännchen (üblich) oder textuell in einer Box als Klasse dargestellt werden. Anwendungsfälle werden als Ovale dargestellt und sind mit den Akteuren mittels Verbindungslinien verbunden.

Anwendungsfälle können mittels Generalisierungen vererbt werden. Mittels *Include*-Beziehungen werden Teilprozesse, auf die von einem anderen Anwendungsfall zugegriffen wird, gekennzeichnet. Und *Extend*-Beziehungen dienen dazu, Erweiterungen von Anwendungsfällen darzustellen, die von bestimmen Bedingungen abhängen. Es handelt sich um optionale Verzweigungen im Ablauf eines Geschäftsprozesses [ERRI02].

In Abbildung 4.1 soll von folgender Spezifikation ausgegangen werden:

„Ein Kunde möchte den verfügbaren Kontosaldo seines Kontos überprüfen und gegebenenfalls Geld abheben. Dazu benötigt der Kunde seine Karte, muss diese in den

Geldautomaten einschieben und dies mit seinem Code bestätigen. Das System interagiert mit dem Banksystem, um die notwendigen Informationen zu erhalten und muss nach erfolgter Geldbehebung den Kontostand des Kunden um die behobenen Geldmenge reduzieren.“

Akteuere: Banksystem, Kunde

Top Level Use Cases: Kontosaldo überprüfen, Geld abheben

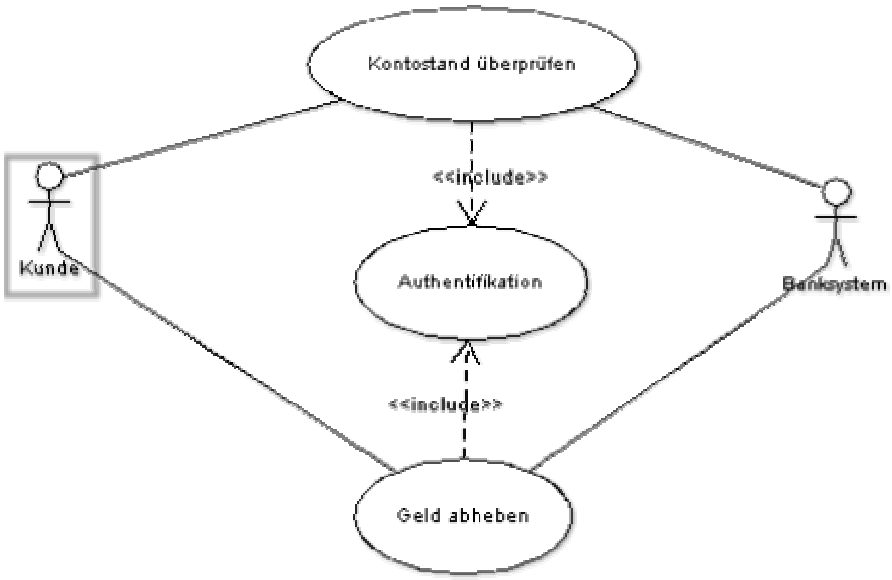


Abbildung 4.1: Use Case Diagramm: Geldautomat

Standardablauf:

1. Der Kunde schiebt die Karte in den Geldautomaten.
2. Das System überprüft die Gültigkeit der Karte.
3. Das System öffnet den Dialog zur Pineingabe.
4. Der Kunde gibt seinen Pincode ein.
5. Das System überprüft den Pin.
6. Das System öffnet das Hauptmenü.
7. Der Kunde wählt ‚Geld abheben' im Menü.
8. Das System öffnet den Dialog ‚Geld abheben'.
9. Der Kunde wählt den gewünschten Geldbetrag.
10. Das System gibt die Karte zurück.
11. Das System gibt den gewünschten Geldbetrag aus.
12. Das System ändert den Kontostand des Kunden.
13. Das System öffnet den *Welcome-Screen* für weitere Kundentransaktionen.

Diese verbal formulierten Schritte sollen / können in einem Zustandsdiagramm dargestellt werden, wie in der SCENT Methode [LISN04] vorgeschlagen. SCENT = *SCEnario based validation and Test of Software*. Werden Zustandsdiagramme verwendet, können die Alternativszenarien sehr einfach herausgearbeitet und modelliert werden.

Friske verwendet bei seiner Methodik wiederum Aktivitätsdiagramme, die er in *System Function* und *System Response* unterteilt, um den Ablauf darzustellen, der innerhalb eines UseCases durchgeführt wird [FRSL05]. Hierbei werden Eingaben und Ausgaben als Stereotypen gekennzeichnet, damit diese bei der Erstellung von Testfällen genutzt werden können. Den abstrakten Systemfunktionen und -reaktionen sind Schnittstellen und Ereignisse zuzuordnen. Für die Systemfunktionen sind implementierungsspezifische Aufrufanweisungen zu erstellen. Für die Systemreaktionen müssen

Vergleichsanweisungen zur Überprüfung der tatsächlichen mit den erwarteten Resultaten festgelegt werden [FRSL05].

Des weiteren schlägt Friske die Testfallentwicklung von Use Cases mit einem gemeinsam mit Fraunhofer FIRST entwickelten Tool, dem so genannten ‚Use Case Validator' vor, in welchem die einzelnen Abschnitte der textuellen Anwendungsfallbeschreibungen eingelesen und diese Abschnitte interaktiv typisiert werden. Anschließend wird die Entwurfsinformation importiert, d.h. Angaben über Systemfunktionen und –reaktionen einschließlich deren Parameter und Informationen über die Akteure. Der Kontrollfluss wird durch Markieren von Blöcken und Zuordnungen von Kontrollflusselementen formalisiert. Den einzelnen Schritten werden die Sytemfunktionen und –reaktionen aus der Entwicklungsinformation zugeordnet. So wird schrittweise ein formales Modell der textuellen Anwendungsfallbeschreibung erstellt, in welchem die Mehrdeutigkeit der textuellen Repräsentation interaktiv beseitig werden [FRIS05]. Die folgende Abbildung 4.2 soll diesen Prozess verdeutlichen.

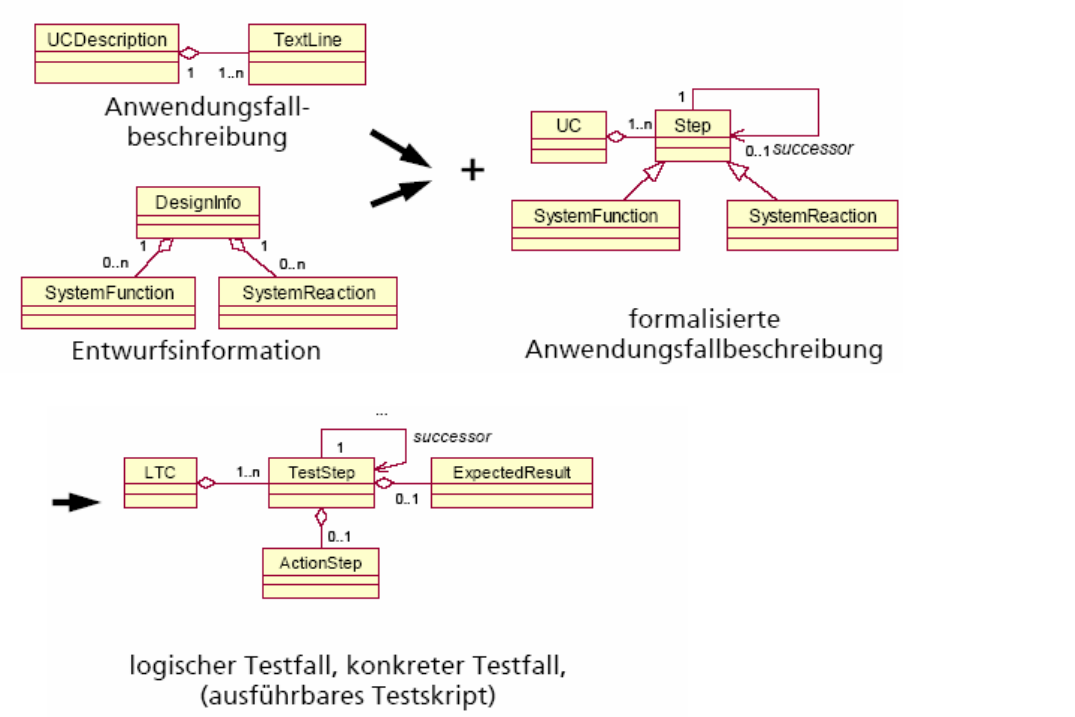


Abbildung 4.2: Interaktives Verknüpfen von Anforderungen und Entwurfsinformation [FRIS05]

4.1.3 USE CASE Template

Folgendes Template soll vorgeschlagen werden, weil sehr viele Ansätze existierten, um den verbalen Teil des Use Cases zu verfassen, und dieses Formular unter Abbildung 4.3 stellt die richtigen Fragen, die als Basis für Konzept und Test notwendig sind [COCK98].

USE CASE #	<der Name soll das Ziel formulieren als Tätigkeit in Verbform>	
Ziel	Eine längere Beschreibung des Use Case Ziels wenn notwendig	
Umfang und Level	<welches System soll die Black Box des Systems repräsentieren> <wahlweise: Gesamtsystem, Hauptaufgabe, Subfunktionalität>	
Vorbedingung	<der Status vor Durchführung des Use Cases>	
Nachbedingung	<der Status nach erfolgreicher Durchführung>	
Nachbedingung im Fehlerfall	<der Status wenn das Ziel nicht ereicht werden konnte>	
Hauptakteur Nebenakteure	<Rollenname oder Beschreibung des primären Akteurs> <andere Systeme, die für die Zielerreichung notwendig sind>	
Auslöser	<Die auf das System wirkende Aktion, die den Start des Use Cases auslöst>	
STANDARDABLAUF	Schritt	Aktion
	1	<Hier sollen die Schritte des Szenarios formuliert werden, vom Auslöser über die Zielerreichung und Nachbearbeitungen>
	2	<...>
	3	
VERZWEIGUNGEN	Schritt	Aktion
	1a	<auslösende Bedingung für die Verzweigung / Extension > : <Name oder Aktion des Sub Use Cases>
SUB-VARIATIONEN		Aktion
	1	<Liste der Variationen>

Abbildung 4.3: Use Case Template

Darüber hinaus können dem Use Case noch Informationen über die Priorisierung (kritisch oder unkritisch), den Aufwand, die Häufgkeit, Sonderinformationen (Interaktionen, Files, Datenbanken oder Timeouts), offen bleibende Punkte, Termindatum oder Managementinformationen mitgegeben werden [COCK98].

4.1.4 USE CASE Testfälle

Use Cases, so wie sie in UML und ähnlichen objektorientierten Methoden dargestellt werden, sind notwendig, jedoch nicht ausreichend für System Test Design. Folgende Parameter müssen definiert werden:

Die Domäne jeder Variable, die in dem Use Case verwendet wird, die notwendige Input / Output Relation zwischen Use Case Variablen, die relative Häufigkeit jedes Use Cases und die sequentielle Abhängigkeit der Use Cases [BIND05].

Somit werden aus Use Cases so genannte *Extended Use Cases*.

Die Methode des "*Extended Use Case Test*" kann verwendet werden, wenn der Großteil des Systems, das getestet werden soll, sinnvoll mittels Use Case abgebildet werden kann. Sinnvoll deswegen, weil in sehr komplexen Programmstrukturen der dahinter stehende Algorithmus nur ‚erraten' werden könnte.

Use Case Scenario

Use - Case	*Akteur*	*Mögliche Input/Output Kombinationen*
Bankomat Session	Kunde	Code einmal falsch eingegeben,
		richtiger Code eingegeben
		Menüanzeige
…	…	…

Welche Tests können nun daraus abgeleitet werden [UMBA01]:

- Tests der erwarteten Verläufe,
- Tests der unerwarteten Verläufe,
- Test jeder Einzelanforderung, die dem Use Case zugeschrieben werden kann,
- Test jedes Features aus der Benutzerdokumentation, das dem Use Case zugeordnet werden kann.

Abbildung 4.4 zeigt nun ein **Beispiel** für ein Use Case Szenario nach oben beschriebenen Template auf dem Level einer **Hauptaufgabe** *(main task)* mit Standardablauf und ohne konkrete Werte:

Name	**Geld abheben**	
Ziel	Der Kunde möchte Geld erhalten	
Umfang und Level	Hauptaufgabe	
Vorbedingung	Der Geldautomat ist in Betrieb, der Begrüßungsbildschirm ist aktiv, die Karte und der Code sind verfügbar	
Nachbedingung	Die Karte und das Geld werden dem Kunden ausgehändigt	
Nachbedingung im Fehlerfall	Der Kunde erhält kein Geld, die Karte wird entweder dem Kunden zurückgegeben oder vom System einbehalten	
Hauptakteur **Nebenakteure**	Bankkunde Banksystem, Servicepersonal	
Auslöser	Kunde möchte Geld abheben	
Standardablauf	Schritt	Aktion
	1	Identifikation der Karte
	2	Kunde wählt ‘Geld abheben’
	3	Dialog ‘Geld abheben’ erscheint auf dem Bildschirm
	4	Der Kunde gibt den gewünschten Geldbetrag ein
	5	Das System wirft die Karte wieder aus
	6	Das System stellt das Geld bereit
	7	Das System wechselt auf den Begrüßungsbildschirm
Verzweigungen	-	-

Abbildung 4.4: Use Case Testfall mit Standardablauf

Folgendes **Beispiel** in Abbildung 4.5 soll **Verzweigungen** eines Use Case Szenario einer Grundaufgabe (*basic task*) mit 3 möglichen Fehlerfällen ohne konkrete Werte darstellen:

Name	**Identifikation**	
Ziel	Die Karte des Kunden und den PIN überprüfen	
Umfang und Level	Grundaufgabe	
Vorbedingung	Der Geldautomat ist in Betrieb, der Begrüßungsbildschirm ist aktiv, die Karte und der Code sind verfügbar	
Nachbedingung	Der Kunde bekommt Zugang zu dem Geldautomaten	
Nachbedingung im Fehlerfall	Zugang wird verweigert, die Karte wird entweder dem Kunden zurückgegeben oder vom System einbehalten	
Hauptakteur **Nebenakteure**	Bankkunde, Banksystem Servicepersonal	
Auslöser	Kunde möchte Zugriff auf den Geldautomaten haben	
Standardablauf	Schritt	Aktion
	1	Der Kunde schiebt die Karte in den Automaten
	2	Das System überprüft die Gültigkeit der Karte
	3	Der Dialog ‚Bitte PIN eingeben' erscheint
	4	Der Kunde gibt seinen gültigen PIN ein
	5	Das System überprüft den eingegebenen PIN
	6	Das System wechselt zum Hauptmenü
Verzweigungen	Schritt	Aktion
	1a	*Der Einführungsschlitz ist verbogen* 1a. 1 Das Servicepersonal informieren 1a.2 Passende Fehlermeldung ausgeben 1a.3 Karte auswerfen
	2b	*Karte kann nicht gelesen werden* 2a. 1 Passende Fehlermeldung ausgeben 21. 2 Karte auswerfen
	2b	*Karte ist ungültig* 2b. 1 Passende Fehlermeldung ausgeben 2b. 2 Karte auswerfen
	...	

Abbildung 4.5: Use Case Testfall mit Standardablauf und Extensions

Nun müssen noch die **Werte** für die Testfälle gefunden werden, Binder schlägt folgende **Teststrategie** dafür vor [BIND05]:

Zuerst müssen die Operationsvariablen identifiziert werden, dann müssen die Wertebereiche festgelegt, die Operationsbeziehungen entwickelt und die Testfälle aufgestellt werden. Die Operationsvariablen werden identifiziert und analysiert und als Input / Output Beziehungen in einer Entscheidungstabelle dargestellt.

Operationsvariablen

Operationsvariablen können von einer Use Case Instanz zur nächsten variieren und werden verwendet um konkrete Testfälle zu spezifizieren [BIND05].

Es sind Eingaben, Ausgaben und Bedingungen, die

- unterschiedliches Verhalten bei Akteuren hervorrufen (PIN-Code 3x falsch bewirkt Karteneinzug),
- den Zustand des Testsystems abstrahieren (Bankomat-Status: außer Betrieb, kein Geld verfügbar, bereit, usw.),
- unterschiedliche Systemantworten bewirken (Kunde wählt Geldabhebung).

Wertebereiche

Die Domänen werden aufgrund der gültigen und ungültigen Werte für jede Variable definiert.

Operationale Beziehung

Relationen zwischen den Variablen werden definiert, die die unterschiedlichen Systemreaktionen in Klassen hervorrufen. Die operationalen Variablen werden mittels Entscheidungstabelle in operationale Beziehungen gebracht: Wenn alle Konditionen einer Reihe erfüllt sind (‚True'), soll die erwartete Reaktion des Use Cases produziert werden. Die operationalen Beziehungen können wieder in tabellarischer Form wie unter Abbildung 4.6 niedergeschrieben werden [BIND05].

	Operationale Variablen				Erwartetes Ergebnis	
Variante	Karten PIN	Eingabe PIN	Rückmeldung der Bank	Status des Kontos	Meldung	Aktion
1	Ungültig	DC	DC	DC	Richtige Karte …	Auswurf
2	Gültig	Gültig	Bestätigt	Geschlossen	Bank kontaktieren	Auswurf
3	Gültig	Gültig	Bestätigt	Geöffnet	Transaktion selektieren	Keine

Daraus werden konkrete Testfälle erstellt und mit den richtigen Werten versehen.

	Operationale Variablen				Erwartetes Ergebnis	
Variante	Karten PIN	Eingabe PIN	Rückmeldung der Bank	Status des Kontos	Meldung	Aktion
1T	%*@#				Richtige Karte …	Auswurf
2T	1234	1234	BEST	GESL	Bank kontaktieren	Auswurf
3T	1234	1234	BEST	OFFE	Transaktion selektieren	Keine

Abbildung 4.6: Use Case Testfälle ohne Werte und mit Werten konkretisiert

Dieses kann auch in derselben Tabelle geschehen, und unter jeder Use Case Variante kann der jeweilige konkrete Testfall geschrieben werden, somit würden logische und konkrete Testfälle untereinander gestellt werden. Das kann nach Übersichtlichkeitskriterien entschieden werden.

Welche Fehler können mit dem *Extended Use Case Test* gefunden werden [UMBA01]:

- Bereichsfehler,
- logische Fehler („und“ statt „oder“ Logik),
- inkorrektes Behandeln von DC-Termen,
- fehlerhaft Ausgabe,
- abnormales Beenden,
- weggelassene Fähigkeiten.

Binder identifiziert dabei folgende Vorteile und Nachteile [BIND05]:

Die **Vorteile** sind, dass die Use Case Methodik in jeder OOA/D Methodik vertreten ist und oft genutzt wird. Es sind keine Kenntnisse von OO-Programmierung nötig, und Use Cases stellen oftmals die einzige Anforderungsdokumentation dar.

Nachteile stellen z.B. das Fehlen von Parametern wie Performance, Fehlertoleranz, usw. dar, oder dass das Abstraktionsniveau eher sehr hoch ist. Bei komplexen Use Cases müssen die *Extend* und *Include* Beziehungen verflacht werden, um die Beziehungen zwischen den jeweiligen Fällen und so das Szenario testbar zu machen.

4.2 Testfallermittlung über Sourcecode

Auch *Whiteboxtest* genannt, bietet die Testfallermittlung aus dem Programmcode viele Möglichkeiten über Tools (beispielsweise der Workflowvalidator), über Metriken wie die C – Überdeckungen oder ganze Programm - *Frameworks* wie Eclipse© [ECLI06].

4.3 Testfälle aus dem Prosatext

Wie oben erläutert, werden vor allem fachliche Konzepte in natürlicher Sprache geschrieben und müssen vom Testingineur über Bedingungen, Regeln und Aktionen in Testfälle transformiert werden. Dabei sollte die Spezifikation schon in übersichtlicher Form niedergeschrieben werden, damit für die Testfallermittlung handhabbare Teile vorhanden wären. Dann können Bedingungen, Aktionen und Sollergebnisse jeder Teilspezifikation ermittelt und die Regeln festgelegt werden. Ob die Regelverteilung über fachliche Pfade, Sollergebnisse oder Algorithmus erfolgt, hängt sehr von der Spezifikation und der Anforderung ab. Die Zuweisung von Werten ist oftmals noch ein weiterer langwieriger Schritt, der nicht zu unterschätzen ist.

Natürlich können auch die Schritte der Ermittlung einer in natürlicher Sprache verfassten Spezifikation in einem Diagramm dargestellt werden. Die Diagramme stellen eben eine Formalisierung dar, und diese werden oftmals aus Spezifikationen abgeleitet. Ein interessanter Ansatz um einen Überblick über die Komplexität des Programms, des Konzepts und der involvierten Testfälle zu bekommen, ist, Grobkonzepte, Anforderungen

und Prozessbeschreibungen in z.B. Ablaufdiagrammen abzubilden und daraus Testfälle abzuleiten. Grobkonzepte haben einen anderen Aufbau als Feinkonzepte und ein wichtiger Schritt stellt die Formalisierung der fachlichen Beschreibungen dar. Die Zerlegung der Spezifikation in handhabbare Teile stellt einen wesentlichen Schritt dar, wenn man mittelgroße und große Anforderungen untersucht.

4.3.1 Tabelle für Keywords

Diese Keywordliste wurde mittels Analyse von vielen CMF Konzepten der Firma SDS erarbeitet, erhebt keinen Anspruch auf Vollständigkeit hierbei, da wie beschrieben, jeder Konzeptionist seinen persönlichen Stil entwickeln kann, und kann beliebig erweitert und verfeinert werden. Die Preconditions und Postconditions sind ohne zwingenden Zusammenhang in der unten stehenden Tabelle in Abbildung 4.7 gesammelt worden und können (oftmals sehr zum Leidwesen des Testers) miteinander in unterschiedlicher Form kombiniert werden. Mit dieser Liste kann der Testingineur den Text nach den Bedingungen und Aktionen für die Erstellung der Regeltabelle durchsuchen. Am Ende der Arbeit wird ein Programm ‚Konzeptpattern' vorgestellt, das Prosatexte, wie sie beispielsweise in CMF erfasst werden, nach definierten Schlüsselwörtern durchparst, und die damit verbundenen Vor- und Nachbedingungen tabellarisch aufgliedert.

Keywords – Preconditions		***Keywords – Postconditions***
Wenn	generell (SDS)	Dann
Sollte		Gilt
Könnte		Sonst
Falls		Ermittle
Insofern		Beginne
Gegebenenfalls		Weiter
Ist		Erzeuge
Für	Corporate Actions	Setze
Und		Sortiere
Oder		Check
Gibt		
Prüfe	Weisung (CA)	
Ermittle		
Bei		

Abbildung 4.7: Keywordliste

4.4 Vergleich UML - CMF

CMF ist übersichtlich in einer Baumstruktur unterteilt und macht die Suche nach der gewünschten Funktionalität projektübergreifend sehr einfach, CMF stellt die Funktionalitäten, Plausibilitäten, Parameter, Attribute,… sehr anschaulich dar, und diese können in den meisten Fällen gut gewartet werden. Verzweigungsfunktionalitäten zwischen den Servicefunktionen bieten gutes Handling, und die interaktive Steuerung der Dialogprüfungen mittels Popupfenster macht Testfallableitungen für Frontendtests einfacher.

Der Nachteil ist, dass die Softwarearchitektur des Programms als ganzes nicht gut überblickt werden kann, dies stellt wiederum die Stärken von UML dar. Die UML beschäftigt sich, wie Spezifikationen in grafischer übersichtlicher Form dargestellt werden können und bietet eine Fülle von Möglichkeiten, wie die Objekte, Phasen, Zustände, Abläufe, User und Methoden in Verbindung gesetzt werden können. Sequenzen, Abläufe und geschachtelte Prädikatsfunktionen lassen sich in grafischer Form um einiges übersichtlicher und verständlicher darstellen. UML ist Standard und darum per Plugin, sei es Opensource oder Linzenzsoftware modular erweiterbar.

Jedoch gestattet die UML dabei zu viele Freiheiten der Modellierung, und die vollautomatische Testfallgenerierung ist in den meisten Fällen ohne Regelerweiterung der UML nicht möglich.

5. Concept Test Facility – CTF

Das CTF, die so genannte Concept Test Facility, ist ein weiteres von der SDS entwickeltes Werkzeug, das unter anderem zum Erstellen, Warten, Durchführen und Verwalten von Testfällen verwendet wird.

Es ist möglich Testfälle in Textform anzulegen, und diese mit Vorbedingungen, Aktionen und Nachbedingungen zu versehen. Für Frontendtests wie z.B. für den Winrunner© Test, werden die Strukturen aus dem CMF geladen und die jeweiligen Strukturfelder mit Werten versorgt. Für Batchtests wurde der so genannte ‚Batchrunner' von der SDS entwickelt, der JCL's (*Job Control Language* – Sprache für Vorlaufkarten) teilweise aus XML Files lädt, die Retourncodes gegen vorparametrisierbare Werte prüft und nach der Durchführung das Ergebnis mittels Prüfselect in SQL gegen das Sollergebnis vergleicht. Des Weiteren wurde ein Framework für Schnittstellentests geschaffen, das Strukturen ebenfalls aus dem CMF lädt, diese in ein weiterverarbeitbares Format konvertiert und mit dem jeweiligen Schnittstellenbatch oder per Onlinefunktion in die Systemumgebung lädt.

Die Testfälle (Testcases) sind in einer Baumstruktur speicherbar und können zu Testläufen (Testsuites) verbunden werden. Diese können wiederum über die so genannten Termine zu richtigen Testsets gesammelt werden. Termine werden beispielsweise für schnelle Querchecks bei kurzfristigen Lieferungen, bei mittleren Tests für Servicepacklieferungen und bei umfangreichen Tests für größere Deltas für den entsprechenden Zweck maßgeschneidert. Die Durchführung jedes Testfalls wird speziell in einer weiteren Ansicht gegliedert und kann langfristig in komprimierter Form gespeichert werden.

Folgende Abbildung 5.1 zeigt die Hauptobjektrahmen Terminübersicht und Testfälle und die mögliche Objektauswahl auf der linken Seite.

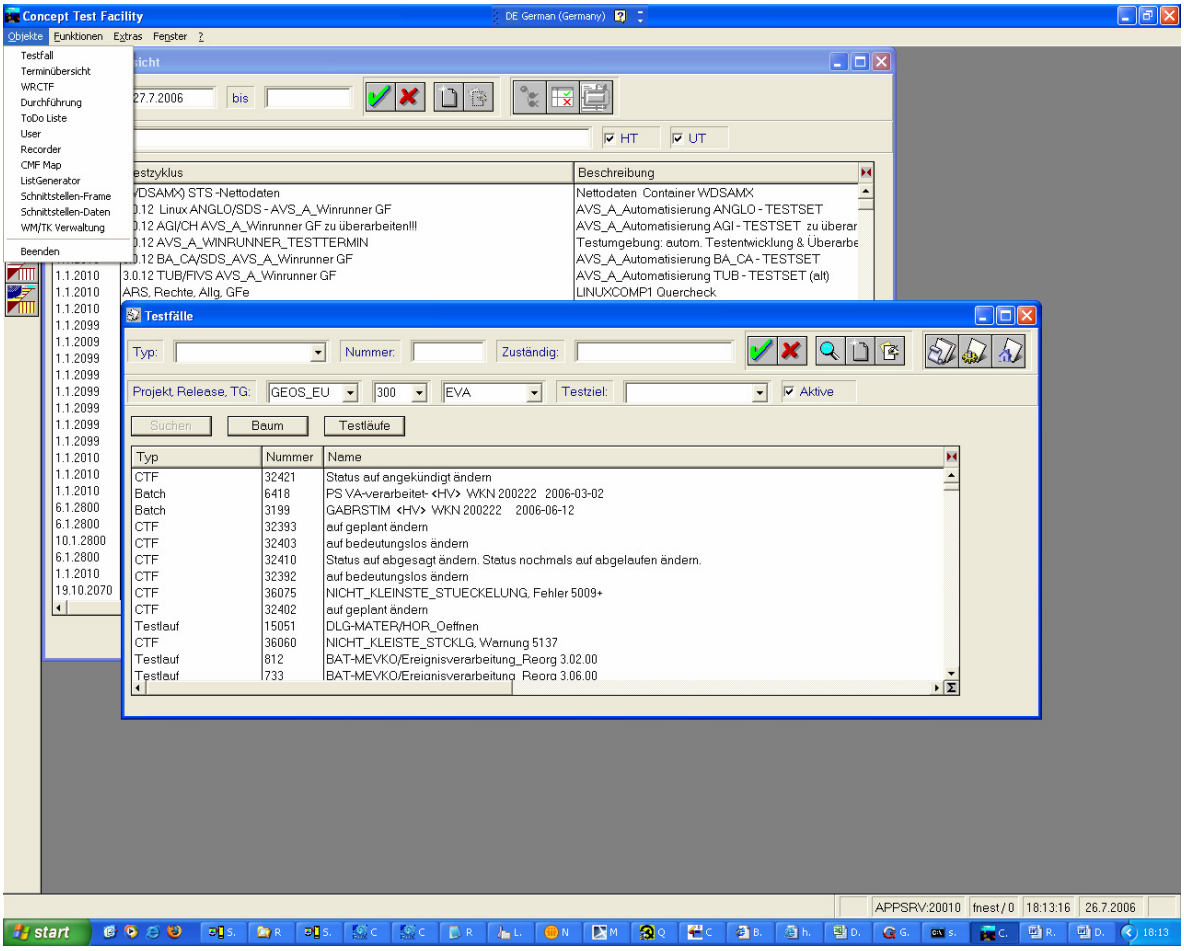


Abbildung 5.1: CTF

6. Test in der Praxis

Die SDS unterteilt nach den Vorgaben des Qualitätsmanagement [HAHA06] die Entwicklung grob in Fachbereiche, die die Planung, Umsetzung und Entwicklung von GEOS vornehmen und den Bereich Testfactory, der sich um den Regressionstest kümmert.

Die **Fachbereiche** haben neben der Entwicklung des Programms GEOS die Verantwortung über folgende Testtätigkeiten: Validierung und Test der neu entwickelten und der existierenden Serviceschnittstellen, Validierung der neu entwickelten

Prozessschnittstellen, Validierung der Use Cases der neu zu liefernden Change Requests.

Die **Testfactory** nimmt dabei den Regressionstest der Prozessschnittstellen, der Geschäftsprozesse, Frontends, Batches, sowie die Überprüfung der Systemdokumentation, der Anpassungsinformation (Mussschritte von Lieferungen), und der Releasefähigkeit vor.

Konzepte werden in den Fachbereichen validiert und der Performancetest erfolgt in der Performancegruppe in Zusammenarbeit mit den Fachbereichen und der Testfactory.

6.1 Fehlerstatistik

Kennzahlen sind, wie unter Kapitel ‚Metriken' beschrieben, nützliche Werkzeuge und werden genutzt um die Effizienz der Testtätigkeit nachzuweisen: Es ist oftmals notwendig die Quantität der Testfälle nachweisen zu können („dass fleißig getestet wird"), ebenso wie die Qualität der Testfälle („ob auch gezielt getestet wird"). Folgende Kennzahlen werden in der Praxis und auch bei SDS verwendet, um Aufschluss über den durchgeführten Testaufwand geben zu können:

- alle Fehlermeldungen
- Testerfehlermeldungen
- Kunden Fehlermeldungen
- Testerstunden
- Testfälle

Sie werden als Grundlage für die verschiedene Berechnungen verwendet und können u.a. für folgende relative Verhältnisse verwendet werden:

- Testfälle/Testerstunden
- Testerfehlermeldungen/Testfälle
- Testerfehlermeldungen/Alle Fehlermeldungen
- Kundenfehler/Alle Fehlermeldungen

Testerfehlermeldungen und Kundenfehlermeldungen, wenn einander gegenübergestellt, geben schnelle Aussagekraft über die Testeffizienz der gesamten Firma. Vor allem Kundenfehlermeldungen sollten so gering wie möglich gehalten werden, da sie einen hohen Kostenfaktor während des Test- und Softwareprozesses darstellen.

6.2 Kennzahlen in der Praxis

Beispielsweise wurde das Projekt des Produkts „Weisung“ mit folgenden Kennzahlen ermittelt, 3 Tester konnten innerhalb von 3 Wochen ca. 300 Testfälle schreiben.

Testfälle/Testerstunden:	300 / 3*5*8,5 = 13 Testfälle pro Stunde
Testerfehlermeldungen/Testfälle:	60 / 300 = 0,2 gefundene Mängel pro Testfall
Testerfehlermeldungen/Alle FM’s:	60 / 280 = 21,47% der Mängel wurden mit diesem Testfallset gefunden.
Kundenfehler/Alle FM’s:	100 / 280 = 35,8 % der Mängel wurden von Kunden über den Customer Support gemeldet.

Anmerkung: die restlichen Mängel wurden während des Entwicklungsprozesses entdeckt.

6.3 Fehlermeldungsprotokoll

Das Fehlerfindungsprotokoll wird in der SDS als ‚Mangel’ geführt und beinhaltet neben den unter Kapitel ‚Fehlerfindungsprotokoll’ angeführten Punkten noch unter anderem die Möglichkeit den Auslöser des Mangels zu vermerken, die Verknüpfung mit ähnlichen Fehlern durchzuführen, oder die Planungen der jeweiligen Liefertermine vorzunehmen. Die SDS verwendet im Großen und Ganzen 3 Klassen: Mangelgrad 3 für Oberflächenfehler, Mangelgrad 2 für Programmschwächen, für die Workarounds o.ä. existieren und Mangelgrad 1 für prozessbehindernde Mängel. In seltenen Fällen können Mängel mit Grad 0 priorisiert werden, die umgehend behoben, getestet und geliefert werden müssen.

6.4 Funktionstestfälle

Funktionstestfälle stellen das Kernstück der Testtätigkeiten dar. Geschäftsprozesstests, die über Frontendtests durchgeführt werden müssten, werden sukzessive auf Test über MFL-Schnittstellen (*Multifrontlink*), Prozessschnittstellen oder Testbatches umgestellt.

6.5 Datentestfälle

Für Überprüfungen, die größtenteils Daten testen, bieten sich diverse Tools an, die die SDS teilweise selbst entwickelt oder manchmal sogar lizenziert. Für unten stehende Testobjekte wurden von der SDS folgende Teststrategien und Tools angewandt.

6.5.1 Schnittstelle: Reports über Nettodaten

Es gibt mehr als 100 Reports (davon zwei Drittel Belege und ein Drittel Berichte; mit den untergeordneten sind es an die 170). Diese liefern so genannte ‚Nettodaten' (interner von der SDS konzipierter Datenstrom, auch genannt „allg. GEOS-Schnittstellenformat"), die von den jeweiligen Kunden mittels eigener Druckaufbereitungsprogramme formatiert werden können. Dieser Datenstrom, wie unter Abbildung 6.1 dargestellt, wird von der SDS für solche Kunden, die nicht selbst über Aufbereitungsprogramme verfügen (wollen), formatiert und in druckfähiger Version zentral oder dezentral ausgegeben. Da manuelles Überprüfen der Belege nicht in vernünftigem Ausmaß möglich wäre (weder in formatierter noch in Nettodatenform), wurden Möglichkeiten der automatischen Prüfung über den Datenstrom (Nettodaten) geschaffen und folgendermaßen durchgeführt.

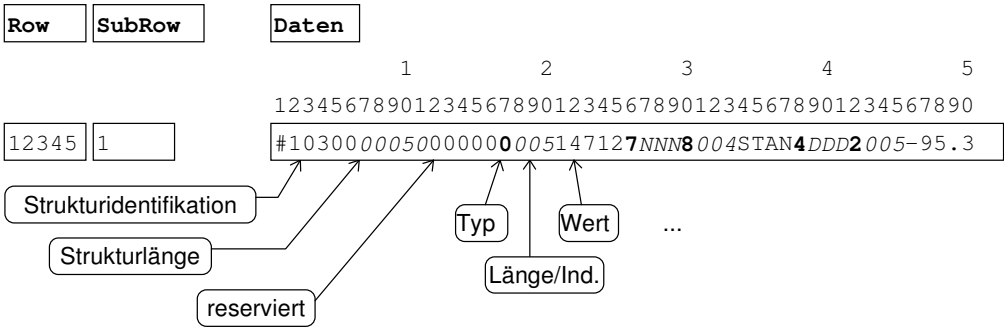


Abbildung 6.1: Nettodatenstrom

Der Datenstrom liefert die gesamten, am Beleg gedruckten Informationen. Diese werden in Strukturen und Attributen abgelegt und müssen natürlich auf ihre Richtigkeit überprüft werden: Einerseits, ob die Strukturen und Attribute vorhanden sind und in der richtigen Reihenfolge abgebildet werden; andererseits, ob die Domänen und deren Inhalte stimmen.

Diese Überprüfung muss automatisiert stattfinden, da es den zeitlichen Testzyklus-Rahmen sprengen würde, diese Daten händisch zu vergleichen.

Die **Testläufe** werden folgendermaßen gebildet. Im ersten Schritt der Automatisierung wird für die wichtigsten Reports (kundenseitig mit der höchsten Priorität) jeweils ein Referenztestlauf angelegt.

Um den automatisierten Test überhaupt möglich zu machen, müssen Testläufe erstellt werden, die pro Durchführung die gleichen Datenbestände abbilden. Somit müssen jedes Mal die gleichen Wertpapiere, Zahlungen, Kurse, Termine (am besten in der Vergangenheit liegende), Aufträge usw. vorhanden sein. Dies geschieht einerseits über die Baseline-Wartung andererseits über eingespielte Schnittstellen-Sätze.

Das Konzept der Report-Testautomatisation sieht vor, dass die erzeugten Nettodaten pro Testzyklus gegen ein Referenz-Set geprüft werden.

Das Referenz-Set wird durch das Fachkonzept oder den Report-Tester fachlich auf Richtigkeit geprüft.

Darauf folgt der **Nettodatenvergleich.** Dieser Nettodatentest erfolgt mittels Vergleich

zweier erzeugter XML-Dateien.

Die eine erzeugte XML-Datei repräsentiert dabei die Referenz-Daten, die andere die im Testzyklus erzeugten Extrakt-Daten. Die Daten sind nichts anderes als der vorhandene Nettodatenstrom abgebildet in einer XML-Strukur.

Ein Report-Testfall besteht aus einer Extrakt-Datei und einer Referenz-Datei pro Service-Pack. Außerdem, je nach genauer Prüfung der Strukturen und Attribute, aus einer oder mehreren Out-Dateien pro Service Pack.

Dies erfordert ein durchdachtes Dateisystem, um einzelne Dateien nach gewissen Kriterien abzulegen und wieder zu finden, um die unterschiedlichen Releases der Testdurchführung verwalten zu können.

6.5.2 Schnittstellen: Beispiel NOSTRO/SNUMS

Vor allem im Bereich von Schnittstellen, die Fachlogik transportieren, können (teil)automatisierte Tools geschaffen werden. Für ein Produkt der Firma SDS, das so genannte NOSTRO, das Daten für interne Bestandsführung, Buchhaltung und Accounting verwaltet, müssen diese Daten von GEOS zu NOSTRO über eine Schnittstelle, die so genannte SNUMS (Schnittstelle Nostro Umsatz Out) transportiert werden. Die Testfactory arbeitet an einem Generatortool, das die gewünschten Daten nicht nur generiert, sondern auch auf die fachliche Korrektheit überprüft.

Es werden hierbei nicht mehr Geschäfte manuell oder (teil)automatisiert durchgeführt, sondern generierte Daten über die Schnittstelle geschickt und auf fachlich korrekte Übernahme geprüft.

6.5.3 Schnittstelle: Stammdatenschnittstellen

SDS schlüsselt Formate für die Stammdatenschnittstellen des WM – Wertpapiermeldedienstes, der Schweizer VDF – Valor Data Feed und allgemeinen Formaten, wie sie die ÖWS – Österreichischer Wertpapier Service verwendet, um.

6.5.4 Reorganisation

Die Reorganisation von „alten“, zur Löschung vorgesehenen oder inaktiven Daten wird mittels Reorganisationsbatches vorgenommen. Da bei der Reorganisation vorderdringlich nicht die Qualität der Daten, sondern die Einhaltung von CASCADE, RESTRICT und IGNORE Regeln für SQL-Datenbanken geprüft werden, arbeitet die SDS mit einem Generator, der die jeweiligen zu testenden Tabellen mit Zufallswerten der Tabellendomänen, Surrogaten und Schlüsselverzeichnissen befüllt und anschließend wieder versucht, diese zu löschen, (physisch durch Entfernen von der Datenbank oder logisch durch Inaktivierungen). Abbildung 6.2 zeigt den Datengenerator mit einem Attribut und den möglichen Werten in Form von Schlüsselverzeichniseinträgen, den Löschregeln der Tabelle CHMTR und den voraussichtlich generierten Daten.

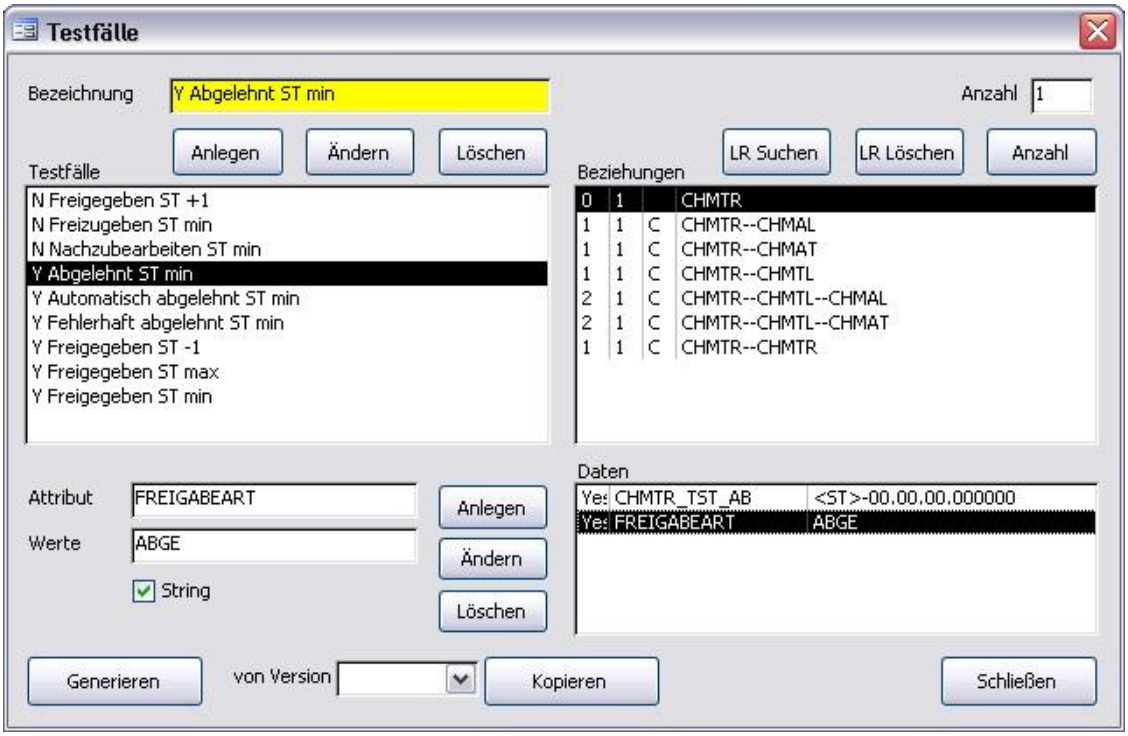


Abbildung 6.2: Datengenerator

6.5.5 Dialogtest

Für den Test von Dialogen verwendet die SDS noch den ‚Mercury WinRunner'©, dessen Lizenz jedoch nicht verlängert werden soll. Mit dem Umstieg auf die Dialogmodellierung von Starview© auf Trolltech QT© soll ein eigenes Plug-In entwickelt und eingesetzt werden. Prinzipiell werden über das Frontend so genannte Negativtests durchgeführt, d.h. man überprüft, wie sich das Programm in Ausnahmesituationen verhält und ob diese gegebenenfalls mit den richtigen Fehlermeldungen quittiert werden. Man hat vor einigen Jahren noch versucht, die Geschäftslogik ebenfalls über Winrunner zu testen, musste dann jedoch feststellen, dass der Test eines komplexen Programms wie GEOS nicht mit vertretbarem Aufwand entwickelt, durchgeführt und vor allem gewartet werden kann. Wie unter Funktionstest beschrieben, werden die Geschäftsprozesslogiken, die über Frontend getestet werden sollten, sukzessive auf Test über MFL-Schnittstellen (Multifrontlink), Prozessschnittstellen oder Testbatches umgestellt.

6.6 Ablauftestfälle

Diese stellen einen Großteil der zu testenden Prozesslogik der GEOS - Software dar und sind in den vorangegangenen Kapiteln ausführlich mit praktischen Beispielen dargestellt worden.

6.7 Fehlerprofile von Gruppen

Die Fehlerquote bei ablaufbezogenen Programm- oder Systemteilen (den ‚verarbeitenden Systemen') ist höher als bei Stammdatensystemen, die eher statische Datentransferlogik des Programms beinhalten. Und wie beschrieben ist die Testfallermittlung von datenbezogenen Systemen, wenn ausreichende Überleitungsmatrizen vorhanden sind, um einiges umfassender, übersichtlicher und leichter. Dies erhöht vor allem die Fehlermeldungsquote der selbst gefundenen Mängel gegenüber den vom Kunden gemeldeten Mängel.

6.8 Testfälle aus fachlicher Sicht und Beurteilung

Durch Erstellung von Testfällen auf funktionaler Basis konnten im Bereich der Terminverarbeitung bei der Firma T-Systems SDS bei der Entwicklung eines neuen Produktes, der so genannten Weisung, gute Erfolge erzielt werden. Es konnten innerhalb weniger Tage Testfallmatrizen mit über 130 Testfällen erstellt werden, die größtenteils aus dem Konzept abgeleitet wurden und um intuitive Testerfahrung angereichert werden konnten. So konnten innerhalb eines Monats (März 2006) Programmschwächen, die schon länger persistent waren, aufgedeckt werden, und mehr als 30 Mängel identifiziert werden. Bis Juli 2006 ist dieses Weisungs - Testfallset auf 300 Testfälle angewachsen und die damit gefundenen Fehler sind auf 60 Mängel gestiegen, das obere Ende ist demnach noch nicht erreicht.

Diese Testfallmatrizen dienen des Weiteren als Grundlage für den Bereich Regressionstest, der sich wie oben beschrieben als Ziel setzt, Geschäftsprozesstestfälle zu automatisieren und per Winrunner, Batchrunner (ein SDS Tool), oder ähnlichen Tools regelmäßig durchzuführen.

7. Parser

Während dieser Studienarbeit wurde vom Autor und einem ehemaligen Arbeitskollegen, Harald Liebetegger, ein Parsingtool, genannt ‚Konzeptpattern', umgesetzt und entwickelt, das anhand einer definierten und erweiterbaren Definitionsbibliothek Prosatexte nach Keywords, im speziellen Falle Preconditions und Postconditions durchsucht und in zwei Bildschirmhälften trennt.

Darüber hinaus wurde mit einem weiteren Kollegen, Martin Adamiker noch ein Parsingtool für das Durchparsen von Sourcecode entwickelt. Dieser so genannte Workflowvalidator durchsucht compilierbare Programme in C-Syntax.

7.1 Parser für Konzepte

Das vom Autor und Harald Liebetegger für diese Arbeit entwickelte Programm ‚Konzeptpattern' bietet Möglichkeiten zum Parsen von Prosatexten auf Bedingungen und Konditionen. Die Abbildungen 7.1 und 7.2 wollen zeigen, wie das unten stehende Inputfile über eine definierte Keywordliste in Preconditions und Postconditions geteilt werden können. Man nehme folgendes Inputfile wie in unter Kapitel ‚Funktionstest' beschrieben:

Wenn MAVER.KZ_Registrierung_uebernehmn = TRUE
Und der Übermittlungsweg = GEOS/GEOS
Dann soll Auftrag_und_Disposition automatisch ausgeführt werden
Sonst
Wenn MAVER.KZ_Umschreibung = TRUE
Dann AUFTR.KZ_Eintragung = ‚1'
Sonst AUFTR.KZ_Eintragung = ‚0'

Man definiere Konditionen über:

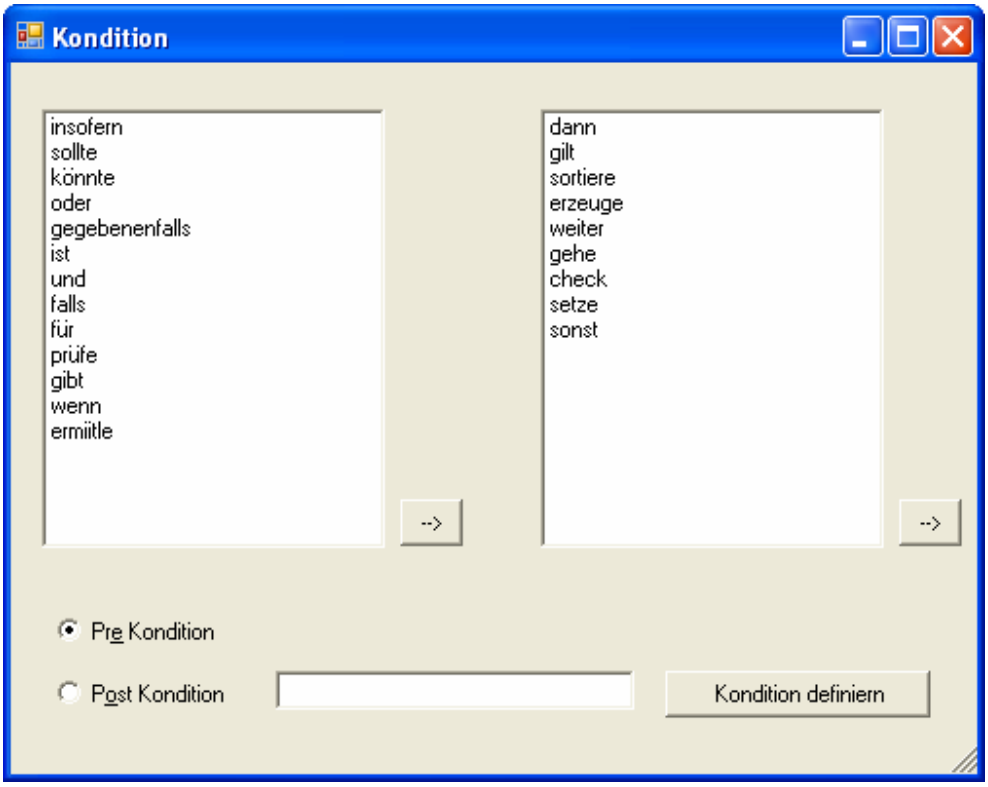


Abbildung 7.1: Konditionentabelle von Konzeptpattern

und erhalte:

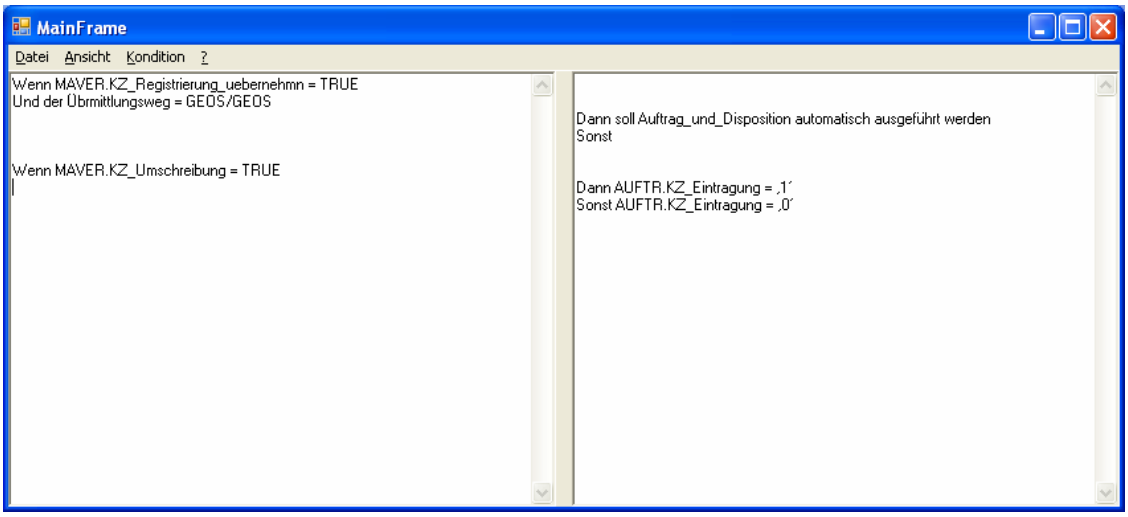


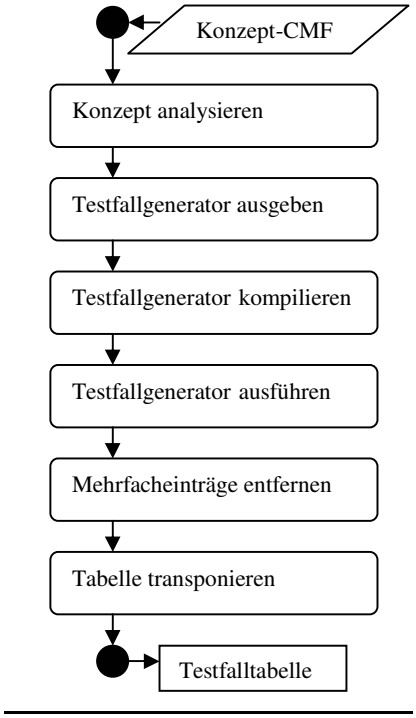
Abbildung 7.2: Hauptübersicht von Konzeptpattern mit den Vor- und Nachbedingungen

Die Schritte der Identifikation der Bedingungen und Regeln wie unter Funktionstest beschrieben, können stark vereinfacht werden. Problematisch ist es dann, wenn Begriffe sowohl für Preconditions als auch Postconditions verwendet werden. Z.B.: Und, Sonst, Prüfe, etc. Hier ist unter Umständen manueller Eingriff notwendig. Auch hier gilt: Je formaler Spezifikationen oder Konzepte formuliert worden sind, desto präziser können Testfälle, etc. daraus abgeleitet werden. Die tabellarische Outputform erleichtert eine Weiterverwendung der ermittelten Pre- und Postconditions in z.B. Testfalltabellen.

7.2 Parser für Programme

Innerhalb der Firma SDS wurde von Martin Adamiker mit Unterstützung des Autors ein Prototyp für einen Testfallgenerator für ablaufbezogene Testfälle entwickelt [ADAM06]. Der so genannte ‚Workflowvalidator’ ist ein Testfallgenerator für ablaufbezogene Testfälle. Das Ziel ist, ablaufbezogene Testfälle automatisch aus dem Konzept oder dem Sourcecode generieren zu können. Ausgangsbasis sind in C geschriebene und compilierbare Programmteile, die Arbeitsabfolge wird unter Abbildung 7.3 beschrieben.

Arbeitsabfolge



Das Konzept wird analysiert und daraus C-Quellcode erzeugt.

Der Quellcode wird compiliert und ausgeführt.

Das Ergebnis ist eine Tabelle von Testfällen.

Aus dieser Tabelle werden mehrfach vorkommende Zeilen entfernt.

Danach wird die Tabelle transponiert, damit Testfälle nebeneinander stehen.

Abbildung 7.3: Standardablauf des Workflowvalidator

Konzept analysieren

Dieser Schritt wurde zum gegebenen Zeitpunkt noch nicht umgesetzt.

Code analysieren

Dieser Schritt wird mit einem Parser [RIEH03] durchgeführt, der für die SDS-Zwecke modifiziert wurde. Ausgangspunkt ist ein ANSI-C-Parser aus dem Compilerbau. Der Parser kann Textpassagen lesen, die in ANSI-C-Syntax [LEE85, LESD87] geschrieben sind.

Die Änderungen bestehen in der Behandlung von **Zuweisungen** und von **Fallunterscheidungen** (IF, SWITCH). Alle Zuweisungen werden durch Ausgabe der Parameterliste eines Testfalls ersetzt. Alle Fallunterscheidungen werden durch

Funktionsaufrufe ersetzt. Die dazugehörigen Funktionen werden generiert.

Abbildung 7.4 zeigt, wie die C-Variablen in Testkonditionen und Testfälle umgeschlüsselt werden.

Beispiel 1

vor dem Parsen:	nach dem Parsen:

`int test()`
`{`
`if(Variable1 == 3)`
`{`
`Variable2 = 7;`
`}`
`else`
`{`
`Variable2 = 9;`
`}`
`}`
	`int test()`
	`{`
	`if(getTestCondition(0))`
	`{`
	`printTestCase(0);`
	`}`
	`else`
	`{`
	`printTestCase(1);`
	`}`
	`}`

Abbildung 7.4: Parserfunktionalität des Workflowvalidator

Die Funktionen getTestCondition() und printTestCase() greifen auf globale, vom Parser definierte und initialisierte Variable zu. Ein Rahmen ruft nun die Funktion test() für alle möglichen Kombinationen von Bedingungen (IF, SWITCH) und Zuweisungen auf.

Die generierte Funktion printTestCase() erzeugt nach Verdichtung und Transposition folgende Tabelle:

C1: (Variable1 == 3)	N	Y
A1: Variable2 = 7	-	Y
A2: Variable2 = 9	Y	-

Darin bedeutet C*n* die Bedingung Nummer *n*; A*n* bedeutet Zuweisung Nummer *n*.
Für den Test ist noch Folgendes zu beachten: es werden die Zuweisungen ausgewertet.
Die Tabelle ist so zu lesen:
Es gibt 2 Testfälle (2. und 3. Spalte).
Wenn C1 nicht erfüllt ist, dann braucht man A1 nicht zu testen. Nur A2 muss erfüllt sein.
Wenn C1 erfüllt ist, dann braucht man A2 nicht zu testen. Nur A1 muss erfüllt sein.

Beispiel 2 ist schon komplexer und beschreibt ein Inputfile mit mehreren *Switches*, Bedingungen und Zuweisungen. Abbildung 7.5 zeigt das Inputfile und Abbildung 7.6 die daraus generierte Testfalltabelle nach dem *Compile*, dem Parsen, dem Entfernen von Redundanzen und dem Transponieren in die lesbare ‚typische' Testfalltabellenform.

```
int test()
{
    if(Variable1 == 4)
    {
        Variable2 = 8;
    }
    else
    {
        Variable2 = 8;
        Variable3 = 9;
        if(Variable5 < 0)
        {
            if(Variable5 < -10)
            {
                switch(Variable5)
                {
                case -20:
                    {
                        if(Variable6 > 10)
                            Variable7 = 1;
                        else
                            Variable8 = 2;
                        break;
                    }
                case -21:
                    {
                        if(Variable9 > 12)
                        {
                            Variable10 = 1;
                        }
                        else
                        {
                            Variable11 = 2;
                        }
                        break;
                    }
                default:
                    {
                        break;
                    }
                }
            }
        }
    }
}
```

Abbildung 7.5: Inputfile in C für den Workflowvalidator

C1: (Variable1 == 4)	N	N	N	N	N	N	Y
C2: (Variable5 < 0)	–	–	Y	Y	Y	Y	–
C3: (Variable5 < – 10)	–	–	Y	Y	Y	Y	–
C4: Variable5 == – 20	–	–	N	N	Y	Y	–
C5: (Variable6 > 10)	–	–	–	–	N	Y	–
C6: Variable5 == – 21	–	–	Y	Y	–	–	–
C7: (Variable9 > 12)	–	–	N	Y	–	–	–
C8: default:	–	–	–	–	–	–	–
A1: Variable2 = 8	–	–	–	–	–	–	Y
A2: Variable2 = 8	Y	Y	Y	Y	Y	Y	–
A3: Variable3 = 9	–	Y	Y	Y	Y	Y	–
A4: Variable7 = 1	–	–	–	–	–	Y	–
A5: Variable8 = 2	–	–	–	–	Y	–	–
A6: Variable10 = 1	–	–	–	Y	–	–	–
A7: Variable11 = 2	–	–	Y	–	–	–	–

Abbildung 7.6: Generierte Testfalltabelle

Wenn man das Praxisbeispiel, wie unter Funktionstest beschrieben in eine formale C-Syntax umschreibt, ist nach erfolgreicher Kompilierung die Testfallgenerierung wie mit ‚normalem' Sourcecode möglich.

Abbildung 7.7 zeigt wieder das durchgängige Praxisbeispiel der Registrierungen von Kapitalmaßnahmen in dem die Bedingung des Übermittlungswegs richtigerweise nur bei Durchlauf der Funktion ‚Registrierung_uebernehmen' berücksichtigt wird.

```
int test()

{
	if(MAVER.KZ_Registrierung_uebernehmen == TRUE)
	{
		if(Uebermittlungsweg == GEOS_GEOS)
		{
			Regi = Auftrag_und_Disposition_automatisch;
		}
	}
	else
	{
		if(MAVER.KZ_Umschreibung == TRUE)
		{
			AUFTR.KZ_Eintragung = '0';
		}
		else
		{
			AUFTR.KZ_Eintragung = '1';
		}
	}
}
```

C1: (MAVER . KZ_Registrierung_uebernehmen == TRUE)	N	N	Y
C2: (Uebermittlungsweg == GEOS_GEOS)	-	-	Y
C3: (MAVER . KZ_Umschreibung == TRUE)	N	Y	-
A1: Regi = Auftrag_und_Disposition_automatisch	-	-	Y
A2: AUFTR . KZ_Eintragung = '0'	-	Y	-
A3: AUFTR . KZ_Eintragung = '1'	Y	-	-

Abbildung 7.7: Testfalltabelle Registrierungen von Kapitalmaßnahmen

Das Resumee, das nach dem Parsen mit diesen Tools gezogen werden kann, stellt sich durchaus positiv dar. Das ‚Konzeptpattern' erleichtert die Tätigkeiten beim Durchforsten von Konzepten und dem Erkennen von Regeln, welches oftmals keinen leichten Schritt darstellt. Der ‚Workflowvalidator' kann zum gegebenen Zeitpunkt nicht nur eingesetzt werden um *Whitebox*-Testfälle generieren zu lassen, sondern um Bedingungen aus dem Sourcecode zu überprüfen. Sollten in der Spezifikation dabei nicht einmal annähernd so viele Regeln definiert sein, wie umgesetzt worden sind, können Konzeptschwächen oder noch besser Konzeptunvollständigkeiten aufgedeckt und somit in weiterer Folge nachgetragen werden.

8. Resumee

Im Zuge dieser Arbeit wurden neben dem CMF weitere Konzepttools vor allem auf UML-Basis untersucht, wie z.B.: Rational Rose, Magic Draw oder verschiedene Eclipse Plug-Ins. Die Frage „welche Konzeptform kann's besser", oder „welches die jeweilige Spezifikationsform unterstützende Tool bringt mehr Effizienz", kann nicht leicht beantwortet werden. Jede Problemstellung kann mit verschiedenen Konzepten besser oder schlechter behandelt werden, und daraus lassen sich unterschiedliche Testfälle gestalten.

Use Cases sind fast unumgänglich, um Problemstellungen vor der Konzeptionsphase erkennen zu können, diese mit dem Kunden abzustimmen und daraus Geschäftsprozesstestfälle entwickeln zu können. Technische Konzepte in prosa eignen sich gut für Funktionstestfälle, und Ablaufdiagramme können mit Use Cases oder Funktionskonzepten kombiniert werden, um als Programmiervorlage und Testvorlage genutzt zu werden.

Diese drei Instrumente sollten Standardwerkzeuge jedes Softwareunternehmens sein, sie stellen allgemeine Techniken dar und eignen sich nicht nur für Problemstellungen der Objektorientierung.

Werden diese Techniken konsequent angewendet, um Konzepte und Testfälle zu schreiben, kann die Qualität der Software schon ohne komplexe Tools um ein Vielfaches gesteigert werden.

In der Zukunft soll der Testprozess mit strukturierteren Konzepten arbeiten, daraus vermehrt Testfälle nach den oben genannten Testtechniken verfassen und Testsuites für automatische Durchführungen innerhalb von CTF formulieren.

In weiterer Folge soll der Workflowvalidator so ausgebaut werden, dass nicht nur Programme sondern auch Konzepte wie beschrieben als Grundlage für Testfallgenerierungen herangezogen werden können.

Literaturverzeichnis

[ADAM06] Adamiker, M. D.: Testfallgenerator, SDS, Wien, 2006

[BEIZ83] Beizer, B.: Software Testing Techniques, van Nostrand Reinhold, New York, 1983

[BIND05] Binder, R.: Testing object oriented Systems, 6. Ausgabe, Addison-Wesley, 2005

[BLÜM03] Blümel: Software Engineering, 2003 (http://www.fh-bochum.de/fb6/personen/bluemel/wirtschaftsinformatik/download/scripte/sescript.pdf August 2006)

[BUWA98] Buwalda, H.: "Testing with Action Words", in Proc. of EUROSTAR 1998, München, 1998

[COCK98] Cockburn Alistair: Basic Use Case Template, 1998 (http://Members.aol.com/acockburn/papers/uctempla.htm August 2006)

[CRJA02] Craig Rick D., Jaskiel Stefan P.: Systematik Software Testing, Artech House, London, 2002

[EBTE06] Universität Koblenz, Arbeitsgruppe Ebert: Software System Testing Methods, Systemtesteinführung, 2006 (www.uni-koblenz.de/FB4/Institutes/IST/AGEbert/Teaching/SS06/SoftwareSystemTestingMethods/01-TEIN.ppt August 2006)

[EBTO06] Universität Koblenz, Arbeitsgruppe Ebert: Software System Testing Methods, Systemtestorganisation, 2006 (www.uni-koblenz.de/FB4/Institutes/IST/AGEbert/Teaching/SS06/SoftwareSystemTestingMethods/06-TORG.ppt August 2006)

[ECLI06] Eclipse – open source platform (www.eclipse.org August 2006)

[ERRI02] Erler T., Ricken M.: UML, Verlag Moderne Industrie, Landsberg, Bonn, 2002

[EXCE06] Excelsoftware: Software Models, 2006 (http://www.excelsoftware.com/statemodel.htm August 2006)

[FRIC95] Frick A.: Der Softwareentwicklungsprozess, Ganzheitliche Sicht, Hanser Verlag, München 1995

[FRIS05] Friske Mario: Interaktive Aufbereitung von Anforderungen für den

modellbasierten Test, 2006 (http://swt.cs.tu-berlin.de/~mario/2005_GI-RE_Friske_Interaktive_Aufbereitung.pdf August 2006)

[FRSL05] Friske, Schlingloff: Von Use Cases zu Test Cases: Eine systematische Vorgehensweise, 2005 (http://swt.cs.tu-berlin.de/~mario/2005_MBEES_Friske-Schlingloff_Von_Use_Cases_zu_Test_Cases.pdf Juli 2006)

[GERK02] Gerke, Michael: OO-Test, 2002 (http://www.l-ray.de/projects/internet/amacont/literatur_cache/Ger02.html August 2006)

[HAHA06] Hasitschka, Hahnl: Geos Modernisierung, Auswirkung auf die Entwicklungs- und Managementprozesse, SDS, 2006

[HARE87] Harel, D.: Statecharts – a visual formalism for complex systems, Science of Computer Programming, Nr. 8, 1987

[HEHI99] Hehn, Hindel: Test – Strategien, Publications of 3Soft, Erlangen, 1999

[LEE85] Lee, J.: ANSI C Yacc grammar, 1985: (http://www.lysator.liu.se/c/ANSI-C-grammar-y.html August 2006)

[LESD87] Lee J., Stockfisch T., Degener J.: ANSI C grammar, Lex specification, 1987 (http://www.lysator.liu.se/c/ANSI-C-grammar-l.html August 2006)

(LEVI06] Levi Lúcio: Software Verification – An Introduction, 2006 (http://smv.unige.ch/tiki-download_file.php?fileId=663 August 2006)

[LISN04] Lichter, Schröder, Naumann: Objektorientierte Analyse – Use Cases, 2004 (http://www.stce.rwth-aachen.de/classes/SS05/SWE/03_OO_Analyse.pdf August 2006)

[MAGI06] Magic Draw (www.nomagic.com August 2006)

[MAVE06] Maverickconsulting: System Test Case Template, 2006 (http://www.mavericksoftwareconsulting.com/Downloads/SystemTestCasesTemplate.xls August 2006)

[MERC06] Mercury Testdirector (www.mercury.com August 2006)

[PISC93] Piscataway: ANSI-IEEE Std. 830 – Guide to Software Requirement Specification, IEEE Standards, New.Jersey, 1993

[PLAT88] Platz, G.: Methoden der Softwareentwicklung, Hanser Verlag, München, 1988

[PLÖG02] Plögert Klaus: Lifecycle Process Model „V-Model“, 2002

(http://www.informatik.uni-bremen.de/uniform/gdpa_d/methods_d/m-bbtd.htm August 2006)

[POST96] Poston, R.: Automating Specification based Software Testing, IEEE Computer Society Press, Los Alamitos, CA., 1996

[RIEH03] Riehl, J.: Cparser, 2003: (http://codespeak.net/svn/basil/trunk/basil/lang/c/ August 2006)

[SAUE98] Sauer, H.: Relationale Datenbanken, 4. Aufl., Addison-Wesley, Bonn, 1998

[SCHL04] Schlingloff, H.: Software – Qualitätssicherung 2004 (http://www2.informatik.hu-berlin.de/~hs/Lehre/2004-WS_SWQS/20041110_Testfallauswahl.ppt#1 August 2006)

[SIEM06] Gespräch mit Fachleuten der Firma Siemens bezüglich des Testvorgehens, Automatisierung und dem Programm IDATG, Wien, 2006

[SNEE02] Testseminar bei Prof. Harry M. Sneed und anschließende Gespräche, Wien, 2002

[SNWI02] Sneed, H., Winter, M.: Testen objektorientierter Software, Hanser Verlag, München, 2002

[SOKE06] Software-Kompetenz: IEEE Std 829 „Standard für die Softwaretestdokumentation“, 2006 (http://www.software-kompetenz.de/?18783 August 2006)

[SPLI03] Spillner, A., Linz, T.: Basiswissen Softwaretest, Dpunkt Verlag, Heidelberg, 2003

[STAH95] Stahlknecht, P.: Einführung in die Wirtschaftsinformatik, Springer Verlag, Berlin, 1995

[THAL02] Thaller, G. E.: Software-Test: Verifikation und Validation, Verlag Heinz Heise, Hannover, 2002

[TIGR06] Tigris: Open Source Software Engineering Tools, 2006 (http://readyset.tigris.org/words-of-wisdom/test-cases.html August 2006)

[UMBA03] Umbach Mathias: A Testers Guide to the UML, 2001 (www.st.inf.tu-dresden.de/ Lehre/SS01/ps/vortraege/thema10.ppt August 2006) August 2006)

[ZELL06] Zeller Andreas: Software-Test: Strukturtest, 2006 (www.st.cs.uni-sb.de/edu/se/2005/strukturtest.pdf August 2006)

Glossar

GEOS	Global Entity online System, Produkt der Firma T-Systems SDS
SDS	Software Daten Service, Tochter von T-Systems
JCL	Job Control Language, Sprache für Vorlaufkarten
SQL	Structured Query Language, Sprache für Selektionen auf Datenbanken
MFL	Multifrontlink, Schnittstellenanbindung für Dialoge
XML	Extendet Markup Language, Standardsprache für austauschbare Formate
UML	Unified Modelling Language
SWIFT	Society for Worldwide Intersettle Financial Telecommunication
STP	Straight Through Processing
GUI	Graphics User Interface
OOA	Objektorientierte Analyse
OOD	Objektorientiertes Design
CA	Corporate Actions

Abbildungsverzeichnis