Bachelorarbeit

Konzeption und Implementierung systematischer Testfallerstellung im Systemtest mit dem HP Quality Center bei Logica Deutschland GmbH & Co. KG

Jörg Hahn

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Konzeption und Implementierung systematischer Testfallerstellung im Systemtest mit dem HP Quality Center bei Logica Deutschland GmbH & Co. KG

Jörg Hahn

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang Technische Informatik
Department Informatik
in der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer : Prof. Dr. rer. nat. Bettina Buth
Zweitgutachter : Prof. Dr. rer. nat. Olaf Zukunft

Abgegeben am 18. Juli 2011

Jörg Hahn

Thema der Bachelorarbeit

Konzeption und Implementierung systematischer Testfallerstellung im Systemtest mit dem HP Quality Center bei Logica Deutschland GmbH & Co. KG

Stichworte

systematische Testfallerstellung, Systemtest, HP Quality Center, OTA-Schnittstelle, Softwaretest, Softwarequalität, Black-Box-Testentwurfsverfahren, Qualitätssicherung

Kurzzusammenfassung

Im Rahmen dieser Arbeit soll eine Werkzeugunterstützung für das Testmanagement-Werkzeug HP Quality Center zur Testfallerstellung im Sytemtest entwickelt werden. Diese Werkzeugunterstützung ist für den Einsatz bei Kunden der Firma Logica Deutschland vorgesehen. Mithilfe dieses Werkzeugs wird eine Verbesserung angestrebt in Bezug auf Zeitersparnis, verbesserter Wartbarkeit und erhöhter Aussagekraft der Testfälle.

Jörg Hahn

Title of the paper

Conceptual design and implementation of systematic test case design for system testing with the HP Quality Center at Logica Deutschland GmbH & Co. KG

Keywords

systematic test case design, system testing, HP Quality Center, OTA-API, software testing, software quality, black-box test design techniques, quality assurance

Abstract

Within this thesis a support tool for the design of test cases in software testing for the test management tool HP Quality Center is to be developed. This support tool is intended for customers of Logica Deutschland to benefit time saving, improved maintainability and an increased informative value of test cases.

Danksagung

An dieser Stelle möchte ich mich bei einigen Personen bedanken, ohne deren Unterstützung diese Arbeit in dieser Form nicht möglich gewesen wäre:

Mein Dank gilt zunächst Florian Bünte für das Thema selbst und die Möglichkeit, diese Arbeit in einem qualifizierten Umfeld bei Logica Deutschland GmbH & Co. KG zu schreiben.

Mein besonderer Dank gilt Prof. Dr. Bettina Buth für die außerordentlich guten Vorlesungen, die mein Interesse an der Thematik geweckt haben und die ausgezeichnete Betreuung sowie hilfreiche Anmerkungen und Anregungen.

Bedanken möchte ich mich auch bei meinen Eltern, die mich mein ganzes Leben lang unterstützt haben und ohne die ich es niemals bis hierher geschafft hätte.

Sehr dankbar bin ich auch Svenja Lindner, die mit geschultem Auge die gesamte Arbeit durchgegangen ist und zahlreiche sprachliche Fehler aufgedeckt hat.

Zuletzt gilt mein Dank Sabine für die moralische Unterstützung und das Verständnis während der Ausarbeitung dieser Arbeit mehr Zeit meinem Schreibtisch als ihr gewidmet zu haben.

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Bei der Entwicklung von Softwaresystemen stehen viele Unternehmen besonderen Herausforderungen gegenüber. Steigendes Qualitätsbewusstsein der Stakeholder und wachsende Komplexität bei gleichzeitigem Druck zu kürzerer Entwicklungsdauer sind einige dieser Herausforderungen. Eine Folge hieraus sind (nicht rechtzeitig erkannte) Fehler in der Software mit der Konsequenz, nicht einzuhaltender Termine, Kostenexplosionen oder das vollständige Scheitern eines Projekts. Für das Jahr 2005 nannte der Chaos Report der Standish Group folgende Kennzahlen [40]:

- 18% der in Auftrag gegebenen Entwicklungsprojekte kamen nie zum Ende
- mehr als die Hälfte der Projekte sprengte Budgets oder Zeitpläne
- nur 29 % der Projekte liefen nach Plan, wobei selbst diese zum Teil mangelhafte Qualität aufwiesen

Die Zusammenfassung des Chaos Reports 2009 weist auf eine weitere Verschlechterung hin, bezeichnet die Auswertungen des Reports sogar als Tiefpunkt aller Studien der letzten fünf Jahre [16]:

- 24% aller Projekte wurden vor Fertigstellung abgebrochen oder kamen nach Auslieferung nicht zum Einsatz
- 44% der Projekte konnten ihre Zeitpläne nicht einhalten, überstiegen die Budgetplanungen und/oder boten nicht alle geforderten Funktionen

Eine weitere Studie des National Institute of Standards and Technology (*The Economic Impacts of inadequate Infrastructure for Software Testing, 2002*) zeigt, dass in den USA jährliche Kosten in Höhe von 59 Milliarden $ entstünden, begründet durch mangelhafte Infrastrukturen für das Testen. Bei verbesserter Testvorbereitung zur früheren Erkennung und Beseitigung von Fehlern könnten 22 Milliarden $ eingespart werden [41, S.11].

Ein Grund für die o.g. Auswirkungen und Tendenzen ist die untergeordnete Bedeutung von Qualitätssicherung bei Software Projekten in vielen Unternehmen. Entstehende Kosten durch das Zurverfügungstellen von Ressourcen (z.B. Personal, Zeit) werden nicht in Relation zu möglichen Fehler- bzw. Folgekosten aufgrund mangelnder Qualität gesetzt. Hierdurch entsteht schnell der falsche Eindruck, Qualitätssicherung sei nicht wirtschaftlich.

Eine der herausragenden Disziplinen von Qualitätssicherung stellt das Softwaretesten dar.

> "Das Testen von Software dient durch die Identifizierung von Defekten und deren anschließender Beseitigung durch das Debugging zur Steigerung der Softwarequalität."[37, S.11]

1.2 Aufgabenstellung

Für die Firma Logica Deutschland GmbH & Co. KG am Standort Hamburg soll ein Werkzeug für eine systematische Testfall- und Testablaufspezifikation im Systemtest entwickelt werden. Testfälle sollen tabellarisch auf Basis von Microsoft Excel festgehalten werden. Das Werkzeug soll einen automatisierten Export dieser Testfälle in das Test Plan Modul der Testmanagement Software HP Quality Center ermöglichen. Darüber hinaus sollen verschiedene Methoden zur Testfallfindung im Systemtest evaluiert werden. Abschließend erfolgt eine Nutzenanalyse anhand eines Beispiels auf Grundlage des Ergebnisses der o. g. Evaluierung unter Anwendung des entwickelten Werkzeugs.

1.3 Umfeld

Logica bietet international IT- und Beratungsdienstleistungen an. In Deutschland ist das Unternehmen spezialisiert auf

- Management- und Technologie-Consulting,
- Systemintegration,
- Infrastruktur- und Business Process-Outsourcing und
- Application Management.

Weltweit beschäftigt das Unternehmen 39.000 Mitarbeiter in 36 Ländern. Deutschlandweit sind 2.000 Mitarbeiter an 13 Standorten beschäftigt. Im Geschäftsfeld Managed Test Services ist Logica einer der führenden Anbieter weltweit und plant den weiteren Ausbau dieses Bereichs [24, S.6, S.11].

1.4 Aufbau der Arbeit

Dieses Kapitel gibt eine Einführung in den Anwendungskontext der Bachelorarbeit. Dadurch soll der Leser die Relevanz und die Problemstellung der Thematik verstehen. Kapitel 2 beschreibt die für das Verständnis der Arbeit benötigten Grundlagen und erläutert deren Zusammenhänge. In Kapitel 3 werden verschiedene Vorgehensweisen zur Spezifikation von Testfällen im Systemtest hinsichtlich der Anwendbarkeit im Fachbereich untersucht. Kapitel 4 gibt einen Überblick über den Funktionsumfang und die Struktur der Testmanagement Software HP Quality Center sowie dessen Möglichkeiten zur Anbindung individuell entwickelter Software. Die zentralen Kapitel 5, 6 und 7 beschäftigen sich mit der Definition von Anforderungen an das in der Aufgabenstellung genannte Werkzeug (vgl. Kap. 1.2), die daraus folgende Konzeption für das zu entwickelnde Werkzeug und die Dokumentation der anschließenden Umsetzung bzw. Implementierung. In Kapitel 8 folgt eine Nutzenanalyse anhand eines Beispiels. Abschließend erfolgen im Kapitel 9 eine Zusammenfassung über die Ergebnisse der Arbeit und ein Ausblick über mögliche Weiterentwicklungen.

2 Grundlagen

2.1 Softwarequalität

Der Begriff Softwarequalität erweist sich als komplex. In Anlehnung an ISO 9126 definiert er sich über verschiedene Merkmale von Software. Diese Merkmale werden unterteilt in Gebrauchsqualität sowie äußere und innere Qualität [1]. Gebrauchsqualität wird nochmals unterschieden in:

- **Effektivität** (meint die Aufgabenerfüllung innerhalb der Genauigkeits- und Vollständigkeitsgrenzen)
- **Produktivität** (beschreibt die Aufgabenerfüllung innerhalb der Aufwandsgrenzen für Benutzer (Zeit, Kosten etc.))
- **Sicherheit** (umfasst die Aufgabenerfüllung innerhalb der Risikogrenzen (Leben und Gesundheit, Geschäftsfeld etc.))
- **Zufriedenheit** (bedeutet die subjektive Zufriedenheit der Benutzer innerhalb des Nutzungskontexts)

Bei äußerer und innerer Qualität differenziert man zwischen:

- **Funktionalität** (gliedert sich in die Teilmerkmale Angemessenheit, Richtigkeit, Interoperabilität, Ordnungsmäßigkeit und Sicherheit)
- **Zuverlässigkeit** (bezeichnet die Reife, Fehlertoleranz und Wiederherstellbarkeit eines Systems)
- **Benutzbarkeit** (beleuchtet die Aspekte Verständlichkeit, Erlernbarkeit, Bedienbarkeit und Attraktivität)
- **Effizienz** (beurteilt das Zeitverhalten und den Verbrauch an Betriebsmitteln für das Erfüllen einer Aufgabe)
- **Änderbarkeit** (bezieht sich auf die Analysierbarkeit und Modifizierbarkeit von Software, sowie deren Stabilität und Testbarkeit)
- **Portierbarkeit** (beinhaltet die Faktoren Anpassbarkeit, Installierbarkeit, Konformität und Austauschbarkeit)

Die Anzahl dieser Merkmale macht bereits deutlich, wie schwer es ist, die Qualität eines Softwaresystems zu beurteilen oder zu messen. Um eine Aussage über die Qualität eines Softwareprodukts zu treffen, muss in den Anforderungen vorher festgelegt werden, welches Qualitätsniveau ein Merkmal erfüllen soll.

> “Das Ziel ist nicht die Entwicklung einer Software mit der besten, sondern mit der richtigen Qualität.“[22, S.1]

Durch geeignete Tests muss die Erfüllung dieser Merkmale überprüft werden. Somit lässt sich vereinfacht sagen:

> “Qualität ist Leistung im Verhältnis zur Erwartung.“[45, S.17]

Die Taxonomie von Softwarequalität im Bereich des Software-Engineerings bezieht sich jedoch nicht nur auf die Produktqualität, auch die Prozessqualität und die Qualität von Projekten sind Teile der Softwarequalität. Durch hohe Prozessqualität soll eine hohe Qualität in Projekten gefördert, wenn nicht erzwungen werden [27, S.112].

2.1.1 Qualitätspolitik

Um die Qualität von Software bei Software(-Systeme) produzierenden Unternehmen nachhaltig und dauerhaft zu verbessern, bedarf es einer Qualitätspolitik, die Bestandteil einer Unter-

nehmensphilosophie sein muss und sollte in der Verantwortung der obersten Leitung liegen. Sie definert die strategische Bedeutung von Qualitätsmanagement und Qualitätssicherung und stellt gleichzeitig Grundlage der Testpolitik dar. Während sich die Qualitätssicherung um die Umsetzung der Qualitätspolitik in einem Projekt kümmert, geht der Focus des Qualitätsmanagements über ein einzelnes Projekt hinaus und vertritt die Umsetzung der Qualitätspolitik in allen Projekten. ISO 8402 beschreibt die Aufgaben wie folgt:

> „Alle Tätigkeiten der Gesamtführungsaufgabe, welche die Qualitätspolitik, Ziele und Verantwortlichkeiten festlegen sowie diese durch Mittel wie Qualitätsplanung, -lenkung, -sicherung und -verbesserung im Rahmen des Qualitätsmanagementsystems verwirklichen.“[32, S.15]

Damit Qualität auf allen Ebenen vertreten werden kann, ist eine eigene betriebliche Hierarchie parallel zur Leitungshierarchie der Softwareentwicklung nötig. Zur Sicherstellung und Umsetzung der gesetzten Anforderungen, ist es notwendig, dass Qualitätsbeauftragte als Stabstellen auf jeder Ebene agieren.

2.1.2 Testpolitik

Die Qualitätspolitik, als Grundlage der Testpolitik, beschreibt die Einstellung und Grundsätze einer Firma zur Softwarequalität, was das Testen als eine Methode der Qualitätssicherung mit einschließt. Es werden Testhandbücher erstellt, worin Teststrategien und Richtlinien für die Softwareentwicklung verschiedener Unternehmensbereiche oder Branchen festgehalten werden. Ein Testkonzept für die Tests innerhalb eines spezifischen Projekts leitet sich dann aus der jeweiligen Teststrategie ab. Die Testvorgehensweise wird vom Testmanager in der Testplanungsphase ausgehend vom Testplan festgelegt. Inhalt sind die Planungsvorgaben des Testmanagers. Hier wird beschrieben, welche weiteren Testdokumente erstellt oder überarbeitet werden müssen.

Die Testpolitik soll unternehmensweit ihre Gültigkeit besitzen und einmal erstellt sollte, sie einer stetigen Wartung unterzogen werden. Inhaltlich sollte sie folgende Themen aufgreifen [39, S.52]:

- die Definition des Begriffs „Testen“ im Unternehmen
- eine Darstellung des Testprozesses
- Vorgaben zur Bewertung des Testprozesses
- Qualitätsziele
- einen Ansatz zur Testprozessverbesserung

Diese Punkte bilden nur den Rahmen einer firmenspezifischen Testpolitik und müssen entsprechend der jeweiligen Philosophie einer Organisation gefüllt und ausgebaut werden. Implizit wird hier der Stellenwert des Softwaretestens innerhalb der Qualitätspolitik geprägt.

2.2 Testmanagement

Innerhalb der Informatik hat sich das Testen von Software zu einer spezialisierten, eigenständigen Fachrichtung und Berufsdisziplin entwickelt [39, S.1]. Auf dem Gebiet des Software Engineerings zählt das Testen zur analytischen Qualitätssicherung. Der Begriff „*Softwaretesten*“ beschränkt sich nicht nur auf das Testen von Software, wie dieses Kapitel u. a. darlegen wird und wie es das ISTQB (*International Software Testing Qualifications Board*) in seiner Ausbildung zum Certified Tester lehrt [2]. Aufgrund der umfangreichen Aufgabenstellung hat man die Rolle des Testers in Testmanager, Technical Test Analyst und Domain Test Analyst aufgeteilt und ihnen eigene Aufgabenschwerpunkte zugeordnet [4, S.7]. Einige Testarten können auch von Entwicklern selbst durchgeführt werden. Dies birgt allerdings die Gefahr einer „Blindheit“

gegenüber den eigenen Fehlern. Designfehler, die auf den Entwickler zurückzuführen sind, beispielsweise aufgrund einer falsch verstandenen Aufgabe oder fehlinterpretierten Anforderung, kann er selbst nicht herausfinden, da ihm der passende Testfall nicht in den Sinn kommt. Hinzu kommt, dass die Herangehensweise der Entwickler auf demonstrativem Testen, einer „Sieh doch, es geht"-Mentalität beruht [45, S.26]. Dies ist für die Fehlerfindung ineffektiv. Abhilfe kann das paarweise Zusammenarbeiten von Entwicklern schaffen, wie es im *Extreme Programming* vorgeschlagen wird. Für den Komponententest (s. Kap. 2.4.1) können wechselseitig die vom Kollegen erstellten Komponenten getestet werden (man bezeichnet dies auch als *buddy testing* oder *code swaps*). Test-Analysten hingegen weisen eine neutrale Haltung gegenüber dem Testobjekt auf und können so falsche Annahmen oder Missverständnisse des Entwicklers aufdecken. Ihre Herangehensweise ist destruktiver Art und „[...] zielt darauf ab, das System kaputt zu machen." [7, S.451].

Die Ziele des Testens von Software sind das Vertrauen in die Qualität eines Softwaresystems zu steigern, die Qualität eines Softwaresystems anhand der Anzahl gefundener Fehler zu messen und durch Dokumentation von Fehlern die Prozessqualität indirekt zu erhöhen. Fehler werden dokumentiert und analysiert, um Fehlhandlungen in Zukunft zu vermeiden. Testen kann jedoch nicht fehlerfreie Software garantieren, da ein erschöpfender Test meistens nicht möglich ist (*Testfallexplosion*). Hierzu folgendes Beispiel [32, S.85]:

> Eine Methode oder Prozedur habe drei Parameter, wobei jeder Parameter mit 16 Bit dargestellt werde. Jeder einzelne Parameter kann somit 2^{16} Werte annehmen. Zusammen wären dies 2^{48}, also 281.474.976.710.656 verschiedene Werte. Ginge man optimistischer Weise davon aus, das Programmstück 100 mal in jeder Sekunde aufzurufen und sein Ergebnis mit dem richtigen Resultat vergleichen zu können, dann bräuchte man für einen vollständigen Test aller Kombinationen rund 89.255 Jahre!

Dieses Beispiel zeigt die Wichtigkeit systematischen Testens - also einer Möglichkeit ohne vollständiges Testen möglichst viele Fehler zu finden. Als Fehler gilt die Nichterfüllung einer festgelegten Anforderung oder eine Abweichung zwischen dem geforderten Soll-Verhalten und dem tatsächlichen Ist-Verhalten. Wird eine Funktionalität unter Beeinträchtigung der Erwartungshaltung nicht vollkommen erfüllt, so wird dies als Mangel bezeichnet.

In Anlehnung an die Terminologie der DIN 66271 wird unterschieden zwischen Fehlerwirkung (engl. *failure*), Fehlerzustand (engl. *fault*) und Fehlhandlung (engl. *error*). Ein Fehlerzustand, auch Defekt oder innerer Fehler genannt, kennzeichnet eine inkorrekte Anweisung, die die Ursache für eine Fehlerwirkung darstellt. Fehlerwirkung oder äußerer Fehler ist die nach außen in Erscheinung tretende Wirkung eines Fehlerzustands. Als Fehlhandlung bezeichnet man die menschliche Handlung eines Entwicklers, mit dem Resultat eines Fehlerzustands oder die menschliche Handlung eines Anwenders, die eine Fehlerwirkung durch Fehlbedienung hervorruft. Defekte äußern sich in einer nach außen auftretenden Fehlerwirkung. Umgekehrt lassen Fehlerwirkungen nicht ausschließlich Rückschlüsse auf Defekte zu, auch eine falsch verstandene Anforderung kann eine Fehlerwirkung provozieren.

2.2.1 Testplanung

Grundlage systematischen Testens ist eine Vorausplanung. Dafür ist es unabdingbar, das Testen so früh wie möglich (*moment of involvement*), bestenfalls zu Anfang eines Softwareprojekts, in die Entwicklung zu integrieren. Grundlegende Fehler in den Testobjekten können so bereits während der Testanalyse und des Testdesigns erkannt und die Behebungskosten minimiert werden.

Es müssen Aufgaben, Zielsetzung und benötigte Ressourcen festgelegt werden. Hierzu zählen

das benötigte Personal zur Durchführung der Aufgaben, die zu veranschlagende Zeit sowie alle nötigen Hilfsmittel und Werkzeuge. Auch eine Priorisierung der Testfälle ist ein wichtiger Bestandteil der Testplanungsphase. Kommt es zu einer vorzeitigen Beendigung der Tests aufgrund von Zeit- oder Budgetmangel wird sichergestellt, dass die wichtigsten bzw. hochprioren Testfälle bereits durchgeführt und schwerwiegende Fehlerwirkungen (*showstopper*) möglichst frühzeitig erkannt wurden. Dies alles wird im Testkonzept (engl. *test plan*) festgehalten.

Um während eines laufenden Projekts auf Verzögerungen, Änderungen der Rahmenbedingungen, Ausfälle oder andere unvorhersehbare Situationen reagieren und eine Aktualisierung der Testplanung vornehmen zu können, gehört auch die Teststeuerung zu den Aufgaben des Testmanagers. Hierdurch wird sichergestellt, dass bei ermittelten Abweichungen von der Testplanung die Testziele eingehalten werden. Testplanung und Teststeuerung stellen wichtige Aktivitäten im fundamentalen Testprozess dar.

2.2.2 Priorisierung

Die Priorisierung dient der Minderung von Risiken wobei man Produkt- und Projektrisiko unterscheidet [37, S.185]. Projektgefährdende Faktoren bedrohen die Auslieferung des Produkts. Dazu zählen lieferantenseitige Probleme durch Ausfallen eines Unterauftragnehmers, Verzögerungen im Projektverlauf oder organisationsbezogene Risiken wie Ressourcenknappheit bedingt durch mangelndes Fachpersonal oder nicht vorhandene Kooperation zwischen Abteilungen. Auch technische Probleme stellen Risiken für ein Projekt dar. Falsche oder unrealisierbare Anforderungen, neue Technologien oder Programmiersprachen ohne benötigte Erfahrung können zum Scheitern eines Projekts führen. Um Projektrisiken zu minimieren, müssen - wie im vorangegangenen Abschnitt beschrieben (vgl. Kap. 2.2.1) - die Tests priorisiert werden. Teilbereiche mit höherem Risiko erhalten höhere Prioritäten. Diese werden somit frühzeitig und intensiver getestet [39, S.245].

Zu den Risiken, die das Produktrisiko betreffen, zählen (negative) Eigenschaften des gelieferten Produkts selbst. Dies sind nicht ausreichende Produkteigenschaften in funktionaler und nicht funktionaler Hinsicht, Verfehlung des Einsatzzwecks oder Auswirkungen beim Einsatz des Produkts, die zu Schäden an Geräten oder zur Gefährdung für Menschenleben führen [37, S.185]. Für die Minderung von Produktrisiken soll nach Spillner [39, S.245] zielgerichtet getestet werden. Mittels Einteilung einzelner Systemfunktionen nach Risikostufen, deren Abdeckung mit unterschiedlichen Testverfahren und Testtiefen sowie einer gesonderten Betrachtung bei der Fehlerbehandlung.

2.2.3 Risikoanalyse

Um eine Priorisierung vorzunehmen, müssen Risiken ermittelt und identifiziert werden. Lassen sich Eintrittswahrscheinlichkeit (W) und der entstehende Schaden (S) quantitativ beziffern, kann das Risiko durch folgende Formel ermittelt werden [39, S.236]:

$$R = W \cdot S, \text{ wobei } W \in [0.0, 1.0]$$

Die Eintrittswahrscheinlichkeit W kann auch prozentual angegeben werden, wobei 0,0 bzw. 0% den unmöglichen Eintritt und 1,0 bzw. 100% den sicheren Eintritt bedeuten. Der Wert ist abhängig von der Anzahl der nach Beendigung der Tests noch vorhandenen Fehlerzustände und der Ausführungshäufigkeit einer Funktion. Sind Eintrittswahrscheinlichkeit W und Schadenshöhe S nicht quantifizierbar, können auch qualitative Angaben erfolgen. Hierzu werden Klassen für die Eintrittswahrscheinlichkeit erstellt, z. B. „niedrig“, „mittel“, „hoch“, „sehr hoch“, und auf Werte von 1 bis 9 abgebildet. Analog erfolgt eine Klassifizierung der Schadenshöhe nach z. B. „vernachlässigbar“, „marginal“, „kritisch“, „katastrophal“. Die Klassen der Schadenshöhe kön-

nen dann auf Werte, wie z. B. 1, 5, 50, 1000, abgebildet werden. Eine nicht lineare Stufung ist hierbei nicht zwingend sondern evtl. vorteilhaft. Eine Verdeutlichung hoher Risiken wird so anschaulicher dargestellt.

In Anlehnung an IEEE-Draft-Standards 829 wird folgende Tabelle 1 zur Risikoeinschätzung vorgeschlagen [39, S.237]:

Eintritt Schadenshöhe	Oft	Wahrscheinlich	Gelegentlich	Unwahrscheinlich
Katastrophal	4	4	4 oder 3	3
Kritisch	4	4 oder 3	3	2 oder 1
Marginal	3	3 oder 2	2 oder 1	1
Vernachlässigbar	2	2 oder 1	1	1

Tabelle 1: *Risikoklassen nach IEEE-Draft-Standard 829*

Eine weitere Möglichkeit der Risikoabschätzung für einzelne funktionale Anforderungen erfolgt anhand nachfolgend beschriebener Formel [39, S.238]:

$$R = \frac{R_T + R_B + R_P}{3} + F \cdot R_B + C \cdot R_T$$

Die Einsatzfrequenz F gibt die Häufigkeit einer verwendeten Funktionalität an. Bei häufiger Verwendung ist die Wahrscheinlichkeit größer, dass auftretende Fehlerwirkungen die Programmausführung beeinflussen. C stellt die Kritikalität bei Ausfall oder fehlerhafter Ausführung einer Funktionalität dar. Das Projektrisiko R_P ergibt sich aus der Abhängigkeit anderer Funktionalitäten zu dieser Funktionalität und behandelt das Risiko des Projektfortschritts bei Wegfall eben dieser. R_T repräsentiert das technische Produktrisiko. Je komplizierter die Realisierung einer Funktionalität ist, was aus der Komplexität der Beschreibung (textuell, UML) hervorgehen sollte, desto höher ist das technische Produktrisiko einzuordnen. Das geschäftliche Produktrisiko R_B bezeichnet die Gefährdung der Akzeptanz oder des Absatzes bei Nichterfüllung der Funktionalität. Auch für diese Formel können wieder Kategorien („gering“, „mittel“, „hoch“) gebildet und mit Werten belegt werden. Laut Spillner [39, S.238] bietet es sich erfahrungsgemäß an, diese Kategorien auf die Werte 1 bis 3 abzubilden.

Es gibt noch zahlreiche weitere Methoden zur Identifizierung und Priorisierung von Risiken, wie z. B. Fehlzustandsbaumanalyse (engl. *fault tree analysis*) [39, S.256], ABC-Analyse aus der Wirtschaftslehre oder Software-FMECA (Fehlermöglichkeits-, einfluss- und -kritikalitätsanalyse, engl. *Failure Mode, Effects and Critically Analysis*) [22, S.433], [39, S.255] auf die hier aber nicht näher eingegangen werden soll.

Nachdem ein Risiko erkannt und analysiert wurde, muss ein Indikator (engl. *trigger*) ermittelt werden, der als Vorbote auf das Eintreten eines Risikos schließen lässt. Indikatoren berechnen sich aus Metriken, daher sollte stets die Berechnungsvorschrift einzelner Metrikwerte zu einem Gesamtwert angegeben werden. Vor allem die Angabe eines Schwellenwerts (engl. *threshold*) ist wichtig, der angibt, ab wann Maßnahmen zur Risikosteuerung einsetzen müssen.

Zu Beginn eines Projekts ist das Datenmaterial meist beschränkt, was die Berechnung erschwert und häufige Fehlalarme provoziert. Diese sollten im Laufe des Projekts stark abnehmen, da die Daten umfangreicher und präziser werden [39, S.239], [45, S.66]. Im Anschluss werden die Ergebnisse der Risikoanalyse, nach Schaden und Eintrittswahrscheinlichkeit sortiert, in ein Risikoinventar übernommen. Dieses lässt sich grafisch in ein Risikodiagramm oder eine Risikomatrix überführen. Entscheidungsträger erhalten hierdurch einen besseren Überblick über die Risikolage und deren Auswirkungen für ein Projekt.

2.2.4 Test- und Fehlerkosten

Wie eingangs bereits erwähnt, werden Qualitätsmaßnahmen primär mit Kosten- und Zeitaufwand in Verbindung gebracht, was den wirtschaftlichen Aspekt schnell in Frage stellt (vgl. Kap. 1.1). Um solchen (Vor-)Urteilen entgegenzuwirken, muss bzgl. des Testens eine Abwägung zwischen Test-, Fehler- und deren Folgekosten erfolgen. Die zuvor beschriebenen Maßnahmen zur Risikoanalyse dienen u.a. auch der Reduzierung der Testkosten. Kritische Bereiche, die als projekt- oder produktgefährdend eingestuft wurden, sollten intensiver getestet, so dass die Wahrscheinlichkeit unentdeckter Softwarefehler, die gegebenenfalls gewaltige Folgekosten nach sich ziehen, reduziert werden. Zudem ermöglicht dies eine Verteilung der benötigten Ressourcen unter ökonomischen Aspekten. Dennoch beträgt der Testaufwand ungefähr 25% bis 50% des gesamten Entwicklungsaufwands. Um innerhalb dieses Rahmens zu bleiben, ist eine Testaufwandsschätzung nötig (s. Kap. 2.2.5). Es gibt zahlreiche Faktoren, die Einfluss auf die Testkosten haben [37, S.179]:

- Der **Reifegrad** des Entwicklungsprozesses,
- die **Qualität** und **Testbarkeit** der Software,
- die verfügbare **Testinfrastruktur**,
- die **Mitarbeiterqualifikation**,
- die **Qualitätsziele** und
- die **Teststrategie**.

Um die durch Fehler entstehenden Kosten zu senken, müssen sie frühestmöglich gefunden werden. Aus diesem Grund soll der Testprozess auch so früh wie möglich in den Softwareentwicklungsprozess eingebunden werden. Betrachtet man den Zeitpunkt der Fehlerfindung im Verhältnis zu den Korrekturkosten, so ergibt sich eine nahezu exponentiell anwachsende Kurve. Liggesmeyer nennt hierzu folgende Beträge [22, S.29]:

- Korrekturkosten während der **Analyse**: 250 Euro
- Korrekturkosten während des **Entwurfs**: 250 Euro
- Korrekturkosten während der **Codierung**: 250 Euro
- Korrekturkosten während der **Entwickler-Tests**:1000 Euro
- Korrekturkosten während des **Systemtests**: 3000 Euro
- Korrekturkosten während der **Produktnutzung**: 12500 Euro

Bei den Fehlerkosten werden, abgesehen von den Korrekturkosten, noch direkte und indirekte Fehlerkosten unterschieden. Die direkten Fehlerkosten umfassen Zahlungen für entstandene Kosten beim Kunden durch Haftung oder Garantie. Dies betrifft Berechnungsfehler (z. B. Fehlbuchung), Datenverlust, Personen-, Maschinen-, Anlagenschäden oder deren Ausfall. Zu den indirekten Fehlerkosten zählen Umsatzverlust durch Vertragsstrafen, erhöhter Aufwand für Kundensupport, Verlust von Marktanteilen oder Marktzulassung eines Produkts.

2.2.5 Testaufwand

Damit sich Qualitätsmaßnahmen lohnen, dürfen die dadurch anfallenden Kosten keinesfalls die entstehenden Kosten möglicher Fehler übertreffen. Die Abb. 1 zeigt ein anzustrebendes Optimum bezüglich der Qualitätskosten. Für systematisches Testen ist es daher wichtig, eine Testaufwandsschätzung vorzunehmen. Man unterscheidet zwischen zwei grundsätzlichen Vorgehensweisen [39, S.85]: *Top-down* und *Bottom-up*.

Beim Top-down-Ansatz wird aus Projektparametern der Gesamtaufwand abgeleitet und auf einzelne Aktivitäten verteilt. Dieser Ansatz hat den Vorteil, dass weder Kenntnisse über die ein-

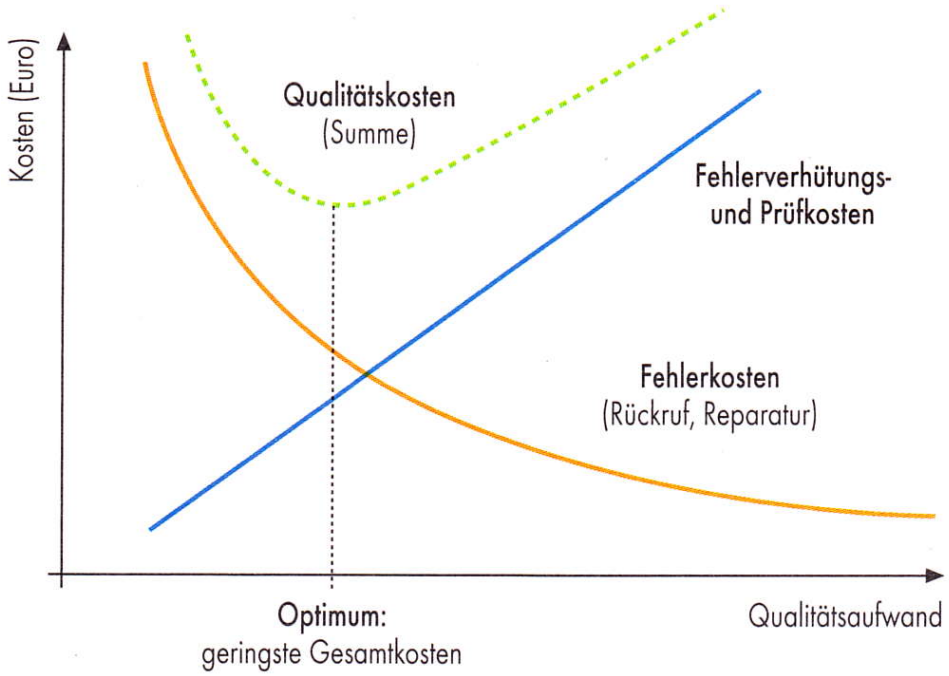

Abb. 1: *Optimierung der Qualitätsaufwände aus [33]*

zelnen Aktivitäten, noch über das zu testende Gesamtsystem nötig sind und somit eine sehr frühzeitige Schätzung vorgenommen werden kann. Demgegenüber steht die potentielle Ungenauigkeit dieses Verfahrens.

Der Bottom-up-Ansatz verlangt zunächst ein Abstecken der Einzelaktivitäten und Abschätzen der daraus resultierenden Einzelaufwände, die anschließend aufsummiert werden. Im Vergleich zum Top-down-Verfahren bietet diese Methode eine höhere Flexibilität. Für die Aufwandsschätzung der Einzelaktivitäten können verschiedene Schätzverfahren genutzt werden, sodass eine höhere Genauigkeit erzielt werden kann. Das Abstecken der Einzelaktivitäten basiert auf Grundlage eines Projektstrukturplans (engl. *work breakdown structure*). Da ein Projektstrukturplan meist nicht gleich zu Beginn eines Projekts vorliegt, kommt diese Methode erst während eines Projekts zum Einsatz.

Eine optimale Lösung stellt das Anwenden beider Verfahren dar. Eine frühzeitige Top-down Schätzung, die stetig verfeinert wird durch Aufteilen des Gesamtprojekts, um diese Teile dann Bottom-up zu schätzen.

2.2.6 Testendekriterien

Ein wichtiges Maß für die Aufwandsabschätzung sind die im Testplan festzulegenden Testendekriterien (*Definition of Done*). Sie sollen dem Testmanager bei seiner Entscheidungsfindung über die Beendigung der Testaktivitäten helfen und dienen der Sicherstellung einer Mindestproduktqualität.

Ein zu frühes Beenden der Tests, kann zur Auslieferung eines (zu) fehlerhaften Produkts führen. Ein zu spätes Beenden kann eine unnötige Verteuerung des Tests und eine Verzögerung der Produktauslieferung zur Folge haben. Beides führt zu Unzufriedenheit des Kunden. Für die Aufwandsabschätzung können die Testendekriterien als absolutes Maß herangezogen werden, für die letztliche Entscheidung zur Beendigung dienen sie zunächst nur als Indikatoren. Sie signalisieren das Erreichen eines Kriteriums, lassen aber keinen Rückschluss auf Fehlerfreiheit zu.

Ein Testendekriterium bestimmt sich aus verschiedenen Testmetriken und einer zu erreichenden Zielvorgabe. Folgende Aspekte sollen nach Spillner [39, S.143] u. a. in die Bestimmung eines Testendekriteriums einfließen:

- Der erreichte **Testfortschritt** in Bezug auf die Vorgaben im Testplan und die tatsächlich erreichte Testobjektabdeckung.
- Die **Testergebnisse** für mögliche Rückschlüsse auf die Produktqualität.
- Eine Abschätzung des **Restrisikos** mittels der Daten aus Testfortschritt und Testergebnissen oder unter Verwendung von Metriken zur Testwirksamkeit, wie z. B. Fehlerfindungsrate (engl. *Defect Detection Percentage*).
- Überprüfung der **wirtschaftlichen Rahmenbedingungen** hinsichtlich der noch vorhanden Ressourcen und dem Entwicklungsplan oder Releasetermin.

Viele Einflussfaktoren erlangen ihre Aussagekraft allerdings erst auf den fortgeschrittenen Teststufen (s. Kap. 2.4). Eine Beurteilung über die wirtschaftlichen Rahmenbedingungen z. B. während des Komponententests erscheint verfrüht und nicht zweckmäßig. Aus diesem Grund werden üblicherweise teststufenspezifische Testendekriterien festgelegt. Dies sind u. a. Metriken zum Testumfang, wie Anweisungsüberdeckung (engl. *statement coverage*) für den Komponententest oder Anforderungsabdeckung (engl. *requirement coverage*) für den Systemtest.

2.2.7 Metriken

Um den Projektfortschritt oder Eigenschaften eines Produkts, wie z. B. Qualitätsmerkmale, ermitteln zu können, müssen Metriken erhoben werden. Hierfür müssen Informationen gesammelt und ausgewertet werden. Für die Ermittlung von Wartbarkeit kann die Anzahl der Kommentare im Quellcode nützlich sein. Die Codegröße, gemessen in KLOC (*Kilo Lines Of Code*), kann ein Indiz für das Qualitätsmerkmal Wartbarkeit sein. Allerdings ist oftmals eine gesteigerte Anzahl von Codezeilen, vor allem in der objektorientierten Programmierung, eine Folge des Qualitätsmerkmals Wiederverwendbarkeit. Qualitätsmerkmale können sich also gegenseitig ausschließen. Daher sollen, wie eingangs beschrieben (s. Kap. 2.1), gewünschte Qualitätsmerkmale vorher festgelegt werden. Dementsprechend ist auch nur das zu messen, was gemessen werden soll, nicht was gemessen werden kann.

Für eine systematische und sachdienliche Bestimmung von Metriken dient die GQM-Methode (*Goal-Question-Metric*). Das Ziel (engl. *Goal*), steht dabei im Vordergrund (Was soll erreicht werden?). Anschließend werden Fragen (*Questions*) formuliert, wie das Ziel erreicht werden kann. Erst auf Grundlage dieser Fragen sind Metriken zu definieren, die entsprechende Antworten liefern.

2.3 Grundlegende Testarten

Es gibt zahlreiche verschiedene Testarten im Softwarelebenszyklus, beginnend bei der Entwicklung eines entstehenden Softwaresystems bis hin zur Wartung oder Weiterentwicklung eines bestehenden Softwaresystems. Aber auch bei Subunternehmern in Auftrag gegebene Softwarekomponenten oder hinzugekaufte kommerzielle Standardprodukte (auch COTS genannt, *Commercial Off-the-Shelf Systems*) müssen den eigenen Qualitätsansprüchen entsprechen und daraufhin getestet werden (Vertrauen ist gut, Testen ist besser).

Das Ziel der unterschiedlichen Testarten ist es, durch vielfältige Strategien und Testtechniken verschiedene Schwerunkte von Fehlern aufzudecken. Dabei ist die Gewichtung der Aspekte Validierung und Verifikation je nach Testart unterschiedlich.

Die grundlegenden Testarten sind [37, S.69]:

- funktionaler Test

- nicht funktionaler Test
- strukturbezogener Test
- änderungsbezogener Test

2.3.1 Funktionaler Test

Funktionales Testen stellt die funktionalen Anforderungen der Spezifikation eines Systems in den Mittelpunkt. Die bei der Testausführung entstehenden Ergebnisse müssen mit den aus der Spezifikation abgeleiteten Soll-Werten abgeglichen werden. Daher wird die Spezifikation auch als „*Testorakel*“ bezeichnet, das befragt werden muss, um die Soll-Ergebnisse vorherzusagen [37, S.24]. Die Umsetzung der Spezifikation ist Voraussetzung für die Nutzbarkeit eines Systems. Funktionales Testen prüft die Vollständigkeit dieser Voraussetzung. Als Techniken werden hierfür hauptsächlich die *Black-Box-Verfahren* eingesetzt (s. Kap. 2.5.2).

2.3.2 Nicht funktionaler Test

Die Kunden- oder Anwenderzufriedenheit hängt stark von den nicht funktionalen Eigenschaften eines Systems ab, auch wenn alle funktionalen Eigenschaften korrekt umgesetzt wurden. Stellt sich die Benutzbarkeit eines Systems als äußerst kompliziert dar, wird der Anwender dieses System ablehnen. Erreicht der Grad der Effizienz ein Niveau, das unter der Akzeptanz des Anwenders liegt, z. B. zu langes Zeitverhalten beim Aufruf von Kundendaten, wird er das System ebenfalls ablehnen. Solche Eigenschaften werden mit entsprechenden Testarten überprüft, vornehmlich im *Systemtest* (s. Kap. 2.4.3). Zwei typische Vertreter hierfür sind **Performanz-** und **Zuverlässigkeitstests**.

Performanztests

Die Performanz, eine nicht funktionale Eigenschaft eines Systems, ist abhängig von der Zeit, die ein System benötigt, um auf Eingaben, z. B. durch den Anwender, zu reagieren. Die Messung von Antwort- und Verarbeitungszeit spielen bei den Performanztests eine zentrale Rolle. Bei den Messungen muss das System bestimmten Bedingungen bzw. bestimmten Lasten ausgesetzt sein. Als Last bezeichnet man die Anzahl der Anwender, die das System gleichzeitig benutzen, oder andere Systeme, die über Schnittstellen mit dem System interagieren. Die folgenden Tests fallen unter die Kategorie der Performanztests:

- Beim **Lasttest** wird eine realistische und zu erwartende Last simuliert, die maximal bis an die Grenzen des erwarteten Rahmens gesteigert wird.
- Der **Stresstest** übersteigt die spezifizierten Lastgrenzen. Hiermit kann herausgefunden werden, welches das schwächste Glied in der Kette ist oder ob ein Totalausfall eintritt. Wünschenswert wäre, dass ein System seine Überbelastung erkennt und es statt zu einem unkontrollierten Totalausfall zu einem geregelten Ausfall (engl. *Graceful degradation*) kommt.
- Eine Variante des Stresstests ist der **Spitzentest**. Dabei wird die Last abrupt gesteigert, um die Systemverträglichkeit mit plötzlichen Laständerungen zu testen. Wird abwechselnd geringe Nutzlast und Spitzenlast simuliert, bezeichnet man dies als **Bounce-Test** [4, S.175].

Zuverlässigkeitstests

Zuverlässigkeit eines Systems bezieht sich auf die Fähigkeit „sein Leistungsniveau unter festgelegten Bedingungen über einen festgelegten Zeitraum oder über eine festgelegte Anzahl von Transaktionen zu bewahren“ [17, S.66]. Für die Bestimmung von Zuverlässigkeitszielen helfen Angaben über die durchschnittliche Zeit zwischen Ausfällen (MTBF = *Mean Time Bet-*

ween Failures). Sie ermittelt sich aus der konkreten Zeit in Stunden zwischen zwei Ausfällen (MTTF = *Mean Time To Failure*) und der Zeit in Stunden zur Behebung des Problems (MTTR = *Mean Time To Repair*) [39, S.237]. Im Zuge der Zuverlässigkeit muss auch die Robustheit eines Systems getestet werden. Dazu werden **Negativtests** entworfen, welche die Robustheit und Fehlertoleranz eines Systems gegenüber ungültiger Eingaben feststellen sollen.

Kritische Systeme, bei denen das Risiko des Ausfalls einer Hardware- oder Softwarekomponente als inakzeptabel gewertet wird, müssen in ihrer Architektur redundante Lösungen anbieten. Nach Bath [4, S.242] ist es Aufgabe der Technical Test Analysts darauf zu achten, dass in solchen Fällen entsprechende Maßnahmen berücksichtigt werden. Für Software kann eine redundante Softwareimplementierung gewählt werden. Dabei wird mehr als eine unabhängige Instanz eines Softwaresystems implementiert. Man bezeichnet solche Systeme als *redundant dissimilare Systeme*. Für Hardware gibt es zwei Möglichkeiten, die Zuverlässigkeit mittels Redundanz zu erhöhen. Beide Möglichkeiten sollen am Beispiel der Hardwarearchitektur des Mikrosatelliten „BIRD“ [30] erläutert werden.

Der Satellit verfügt über fünf redundante Hauptplatinen (*nodes*). Davon ist eine aktiv (*worker*), während eine zweite (*supervisor*) die erste fortlaufend auf Irregularitäten prüft. Im Falle einer festgestellten Anomalie des *worker* wird der *supervisor* zum neuen *worker* und übernimmt die Kontrolle des Satelliten. Der vorherige *worker* wird abgeschaltet und einer der verbleibenden *nodes* wird zum neuen *supervisor*. Bei dieser Art der Redundanz übernimmt eine Komponente unmittelbar nach Ausfall die Aufgaben der anderen Komponente. Eine zweite Möglichkeit ist die Verwendung eines *Voting-Systems*. Über Sensoren erfasst der Satellit Messwerte, die er zur Berechnung an mehrere *Controller* weiterreicht. Man könnte diese Ergebnisse nun an eine Instanz übergeben, die mithilfe einer Plausibilitätsprüfung falsche Ergebnisse herausfiltert. Im Falle des Satelliten aber kommt ein *voter* zum Einsatz. Dieser vergleicht die eingegangenen Ergebnisse miteinander und in einer Art demokratischer Entscheidung sendet er nur dasjenige Ergebnis weiter, das sich mehrheitlich ähnelt. Der Vorteil hierbei ist, nicht nur falsche, sondern auch ungenaue Ergebnisse filtern zu können.

Ein einzelner Fehler soll den Satelliten nicht in einen gefährdenden Zustand bringen. Dies bezeichnet man als *Single Point Failure* [45, S.144].

Sind solche Failover-Mechanismen zur Steigerung der Zuverlässigkeit, Fehlertoleranz, Wiederherstellbarkeit und Verfügbarkeit implementiert, muss deren Funktionalität durch sogenannte **Failover Tests** oder **Recovery Tests** belegt werden.

2.3.3 Strukturbezogener Test

Beim strukturbezogenen Testen liegt der Fokus auf der Kontrollstruktur bzw. auf dem Kontrollfluss des zu testenden Objekts. Ziel ist die Abdeckung (engl. *coverage*) des Quellcodes zu einem bestimmten Grad. Strukturelles Testen findet überwiegend beim Komponenten- und Integrationstest statt. Als Techniken werden hierfür hauptsächlich die *White-Box-Verfahren* eingesetzt (s. Kap. 2.5.2).

2.3.4 Änderungsbezogener Test

Werden an der vorhandenen Software Änderungen vorgenommen (Wartungsarbeiten) oder neue Softwareeinheiten hinzugefügt (Weiterentwicklung), muss das geänderte System erneut getestet werden. Ziel ist es,

> „nachzuweisen, dass durch die vorgenommenen Änderungen keine neuen Defekte eingebaut oder bisher maskierte Fehlerzustände freigelegt wurden“ [37, S.74].

Man bezeichnet dies auch als **Regressionstest**.

Regressionstests

Durch Pflege, Wartung und Weiterentwicklung werden Eingriffe in ein bestehendes System unternommen, die Änderungen zur Folge haben. Nach jeder Änderung muss das System erneut getestet werden, um nachzuweisen, dass durch die vorgenommenen Modifikationen keine neuen Defekte hinzugefügt wurden oder nicht beabsichtigte Seiteneffekte auf das bestehende System auftreten. Vor allem in Bezug auf die Seiteneffekte ist es wichtig, dass nicht nur die geänderten Programmstellen oder hinzugefügten Softwarebausteine geprüft werden, sondern ein vollständiger Regressionstest am ganzen System erfolgt. Tests bei Anpassungen oder Erweiterungen (*adaptive* und *enhanced Maintenance*) am System oder einer Designänderung ohne Verhaltensänderung (engl. *Refactoring*, *perfektive Maintenance*) nennt man **Regressionstests**, während die Wiederholung von Tests, die bereits vor einer Fehlerkorrektur (*korrektive Maintenance*) eine Fehlerwirkung erzeugt haben, als **Fehlernachtest** bezeichnet werden.

Vor allem in der agilen Softwareentwicklung, wie z. B. dem *Extreme Programming* mit seinem Konzept der permanenten Integration (engl. *continuous integration*) nach Kent Beck, sind Regressionstests unverzichtbar. Das häufige Integrieren (teilweise mehrmals täglich) und die damit verbundene Wiederholung sämtlicher Testfälle, weckt den Bedarf nach Automatisierung mittels CAST-Tools (*Computer Aided Software Testing*).

Zur Kontrolle des Fortschritts von Regressions- oder Fehlernachtests eignen sich folgende Metriken [39, S.297]:

- Anzahl Testfälle, die noch nie ohne Fehlerwirkung durchgeführt wurden (engl. *failed, never passed; FNP*)
- Anzahl Testfälle, die noch nie eine Fehlerwirkung aufdeckten (engl. *passed, never failed; PNF*)
- Anzahl fehlgeschlagener Testfälle, die vor der Modifikation keine Fehlerwirkung aufdeckten (engl. *failed, was passed; FWP*)
- Anzahl bestandener Testfälle, die vor der Modifikation eine Fehlerwirkung aufdeckten (engl. *passed, was failed; PWF*)

Alle Testfälle, die durchgehend Fehlerwirkungen aufdecken (*FNP*), offenbaren noch nicht korrigierte Abweichungen. Sind die Anzahl von stets bestandenen Testfällen (*PNF*) und die Testabdeckung hoch, kennzeichnet dies eine hohe Qualität des zu testenden Systems (engl. *system under test; SUT*). Eine geringe Testabdeckung und hohe Anzahl von stets bestandenen Testfällen könnte auf eine geringe Effektivität der Testaktivitäten hinweisen. Eine gesteigerte Anzahl von erst nach einer Modifikation fehlgeschlagener Tests ist entweder ein Indiz für ein geringes Maß des Qualitätsmerkmals Änderbarkeit oder Ausdruck mangelnder Wartungsprogrammierung. Ist die Anzahl erfolgreicher Fehlernachtests (*PWF*) hoch, lässt dies einen Rückschluss auf eine hohe Effektivität der korrektiven Maintenance zu.

2.4 Teststufen

Der Testprozess ist eng mit der Softwareentwicklung verbunden, aber als eigenständiger Prozess anzusehen. Wird der Testprozess frühestmöglich in die Softwareentwicklung eingebunden, sollten Entwicklungs- und Testaufgaben parallel verlaufen. „Für die Qualitätssicherung ist die Existenz von definierten Phasen vorteilhaft. Dies ermöglicht die Verzahnung von Konstruktions- und Prüfschritten sowie die Identifikation geeigneter Prüfreferenzen“ [22, S.346].

Das V-Modell, als international anerkannter Entwicklungsstandard für IT Systeme, zeigt die Gleichwertigkeit von Entwicklung und Testaktivitäten [37, S.39] (vgl. Abb. 2). Es zeigt die Aufteilung des Testprozesses in einzelne Testaktivitäten und die zu verwendende Testart für je-

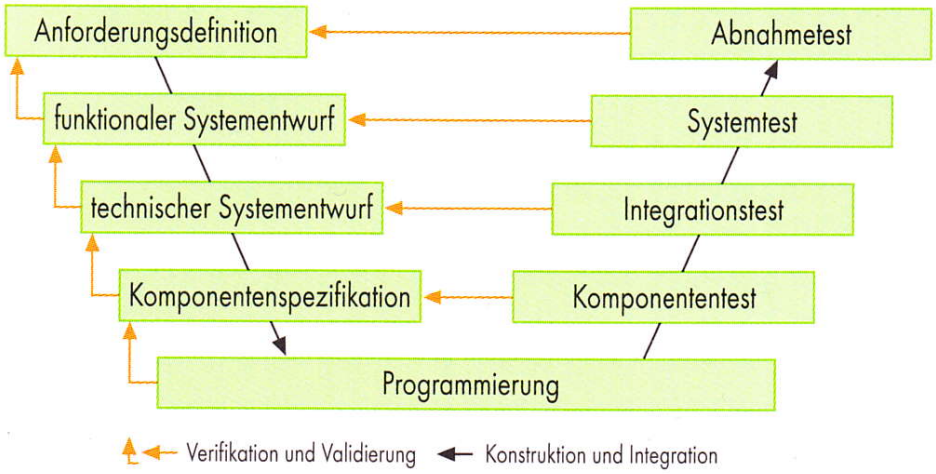

Abb. 2: *Darstellung des V-Modells aus [38]*

de Teststufe. Die dem V-Modell zu Grunde liegenden Testprinzipien lassen sich auch auf andere Vorgehensmodelle übertragen. Fehler lassen sich am einfachsten auf derselben Abstraktionsstufe finden, auf der sie entstanden. Daher ordnet der rechte Ast des Modells jedem Konstruktionsschritt eine entsprechende Teststufe zu [37, S.40]. Neben der grafischen Anordnung geben die Prüfaspekte Verifikation und Validierung dem V-Modell seinen Namen. Dass die Planung der Testaktivitäten parallel zu den Entwicklungstätigkeiten stattfindet, wird in der Darstellung des V-Modells nicht deutlich. Das W-Modell (vgl. Abb. 3) hingegen veranschaulicht dies explizit.

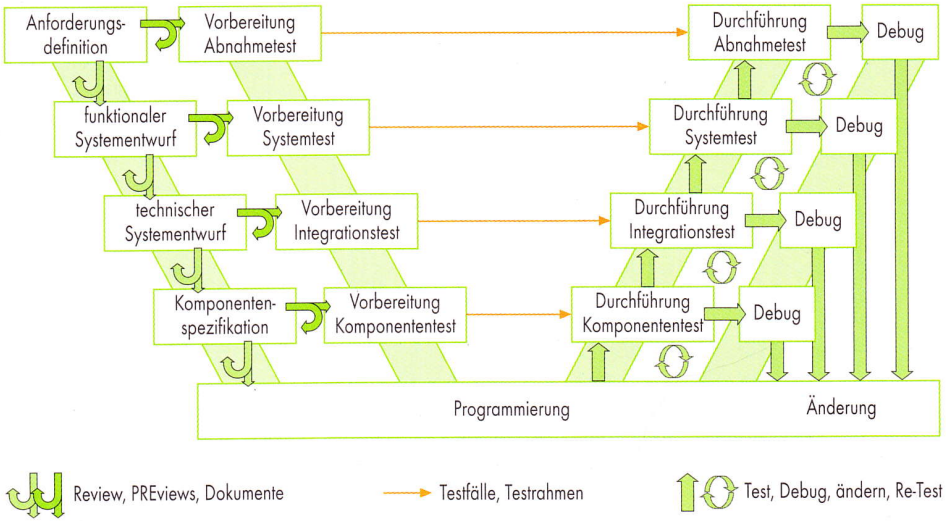

Abb. 3: *Darstellung des W-Modells aus [38]*

2.4.1 Komponententest

Der Komponententest (in der Literatur auch häufig als Modul-, Unit-, oder Klassentest bezeichnet) zählt zur Kategorie der funktionalen Tests und reiht sich auf der untersten Ebene der Teststufen ein. Die Testobjekte im Komponententest sind kleine Softwareeinheiten (Module, Units oder Klassen - je nach Programmiersprache). Direkt nach Abschluss der Programmierarbeiten unterliegen diese Einheiten durch den Komponententest einem ersten systematischen Test [37, S.42]. Die einzelnen Komponenten werden getrennt zu anderen Komponenten getestet, um

mögliche Fehler der jeweiligen Komponente zuschreiben und externe Einflüsse ausschließen zu können.

Als Testbasis dienen alle zur Verfügung stehenden Dokumente, die die zu testende Komponente betreffen. Da die einzelnen Komponenten die ersten Bausteine eines Gesamtsystems darstellen, wird für den Test häufig eine Testumgebung, auch als Testrahmen bezeichnet, benötigt. Eine Testumgebung kann aus Testtreiber (engl. *test driver*), zum Aufruf der Dienste des Testobjekts und/oder Platzhaltern (engl. *stub*) bestehen. Platzhalter mit sehr einfacher Funktionalität bezeichnet man als „*dummy*“, enthalten sie erweiterte Funktionalität, nennt man sie „*mock*“. Platzhalter dienen der Simulation später zur Verfügung stehender Dienste, die das Testobjekt importiert. Da das Aufsetzen einer Testumgebung Entwicklerwissen voraussetzt und die Testobjekte direkt von den Entwicklern zur Verfügung gestellt werden, wird der Komponententest als entwicklungsnah bezeichnet. Aus diesem Grund sind es auch meist die Entwickler selbst, die den Komponententest durchführen [37, S.45]. Neben den dynamischen Testtechniken werden hier auch statische Testtechniken angewandt (vgl. Kap. 2.5), da die Größe des Programmcodes hier noch als übersichtlich einzuschätzen ist. Als primäre Testziele sind Robustheit, Effizienz und Wartbarkeit in Betracht zu ziehen.

Für das Testen von objektorientierten Softwarekomponenten müssen nach Vigenschow [45, S.109] noch zwei weitere Eigenschaften berücksichtigt werden: Vererbung und Assoziation. Diese Eigenschaften erzwingen eine orthogonale Ausrichtung der Teststrategie. Anhand der Klassendiagramme wird vertikal (*Top-down*) entlang Vererbungshierarchie getestet, von der generalisierten Klasse hin zur spezialisierten. Auf Grundlage von Sequenzdiagrammen wird horizontal entlang der Assoziationsketten das gegenseitige Aufrufen kooperierender Objekte getestet. Bei einer Verflechtung beider Eigenschaften beginnt das Testen in der obersten Vererbungshierarchie und dort entlang der Assoziationsketten. Um auf Platzhalter verzichten zu können, empfiehlt es sich, bei unidirektionalen Assoziationen erst die unabhängige Klasse zu testen, also von der benutzten Klasse hin zur benutzenden Klasse, sofern die Fertigstellung der Klassen dies ermöglicht.

Die Testreihenfolge für die Darstellung in Abb. 4 wäre demnach *C1*, *C2*, *C3a*, *C3b*, wobei die Reihenfolge für *C3a* und *C3b* auch getauscht werden kann.

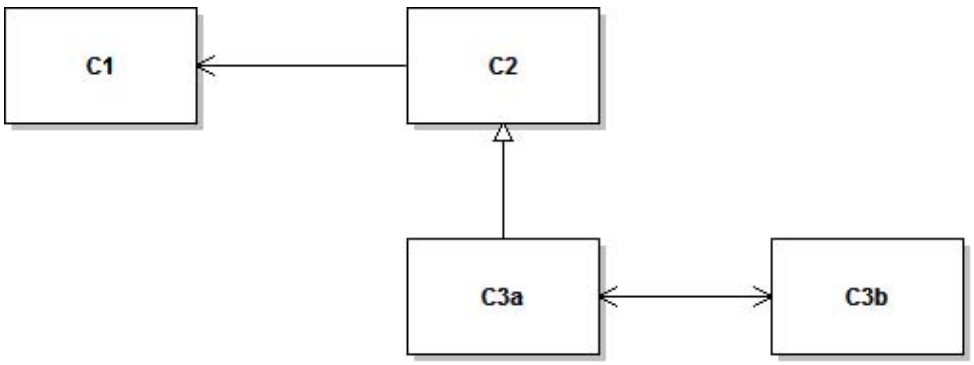

Abb. 4: *Testreihenfolge bei Vererbung und Assoziationen in Anlehnung an [45, S.115]*

2.4.2 Integrationstest

Nachdem die Komponenten den ersten systematischen Test erfolgreich durchlaufen haben, können sie zu größeren Softwareeinheiten zusammengesetzt werden. Jetzt muss geprüft werden, ob im Zusammenspiel durch Interaktion der einzelnen Komponenten Fehlerzustände auftreten. Fehler dieser Art, können durch den Komponententest nicht aufgedeckt werden. Somit rücken in den Mittelpunkt des Integrationstests die Schnittstellen der Komponenten. Charakteristische

Fehler sind zueinander inkompatible Schnittstellen von Komponenten, zeitliche Probleme beim Datenaustausch oder durch den Komponententest nicht aufgedeckte funktionale Fehler innerhalb einer Komponente. Der Integrationstest muss also funktionale und nicht funktionale Eigenschaften testen. Da beim Integrationstest nur ein Teilsystem vorliegt, wird auch hier ein Testrahmen benötigt. Meist können hierfür aber die bereits beim Komponententest verwendeten Treiber wiederverwendet werden.

Wesentlichen Einfluss auf die Vorgehensweise beim Integrationstest hat die gewählte Integrationsstrategie. Für die Integration nennt Liggesmeyer [22, S.354] drei Integrationsstrategien:

- Der **Bottom-up-Integrationstest** beginnt bei den untersten, den elementaren Komponenten und setzt sich in Richtung aufrufender Komponenten fort. Die Vorteile hierbei sind, dass ein frühes Prüfen der Interaktion mit dem Betriebssystem möglich ist und keine Platzhalter benötigt werden. Demgegenüber müssen Testtreiber erstellt werden, falls die aus dem Komponententest bereits vorhandenen nicht wiederverwendet werden können. Außerdem steht ein repräsentatives Produkt erst am Ende der Integration zur Verfügung, da die Hierarchie von unten nach oben durchlaufen wird.
- Umgekehrt verläuft der **Top-down-Integrationstest**. Der Test berücksichtigt zunächst die sich auf der obersten Schicht befindenden Komponenten und schreitet in Richtung aufzurufender Komponenten voran. Die Vorteile dieses Vorgehens sind die Möglichkeiten, bedeutende Steuerungsfunktionalität bereits zu Beginn zu prüfen, da die Hierarchie von oben nach unten durchlaufen wird sowie der mögliche Verzicht auf Testtreiber. Allerdings wird das Zusammenspiel zwischen zu testender Software und Betriebssystem erst spät getestet. Da dieser Bereich aber als besonders fehlerträchtig einzuschätzen ist, verbleibt nur wenig Zeit am Integrationsende für Korrekturen. Zudem werden Platzhalter für aufzurufende, noch nicht zur Verfügung stehende Komponenten benötigt.
- Die aus Bottom-up und Top-down abgeleitete Mischform ist der **Outside-in-Integrationstest** oder auch **Sandwich-Test-Strategie** genannt [7, S.484]. Dieser Ansatz vermengt die Vorteile der beiden Strategien und vermeidet deren Nachteile. Der einzig verbleibende Nachteil ist das Erfordernis nach Testtreibern und Platzhaltern.

Spillner [37, S.57] zählt noch zwei weitere Integrationsansätze auf:

- Bei der **Ad-hoc-Integration** werden die einzelnen Komponenten in der zufälligen Reihenfolge ihrer Fertigstellung integriert. Dadurch kann jede Komponente zum frühestmöglichen Zeitpunkt in das entstehende Gesamtsystem integriert werden, jedoch werden, wie bei der Outside-in Integration, Testtreiber und Platzhalter benötigt.
- Für die **Backbone-Integration** wird zuvor ein Programmskelett (*Backbone*) erstellt. Hier können die Komponenten nacheinander, ohne Einhaltung einer Reihenfolge, eingesetzt werden. Vorteilhaft hierbei ist die beliebige Reihenfolge und die frühestmögliche Integration, allerdings zum Preis eines evtl. aufwändigen Programskeletts.

Für die Integration kann auch eine individuelle Mischform aller genannten Strategien benutzt werden. Wichtig ist vor allem, dass die Integration schrittweise geschieht, was alle diese Strategien gemein haben. Ein Zusammenwurf nach Fertigstellung aller Komponenten (*big bang*) soll unter allen Umständen vermieden werden. Die Zeit bis zum *big bang* ist verlorene Zeit und die Fehlerbehebung findet nicht mehr schrittweise während, sondern konzentriert am Ende der Integration statt.

2.4.3 Systemtest

Als wichtiger Bestandteil dieser Arbeit soll nun der Systemtest näher erläutert werden. Nach gruppenweiser Integration der Komponenten und Beendigung aller Testaktivitäten zum Inte-

grationstest folgt der Systemtest. Wichtig hierbei ist, dass die Sichtweise der *Stakeholder* berücksichtigt wird. Es muss validiert werden, ob das System angemessen umgesetzt ist und den Ansprüchen von Kunden und Anwendern genügt [37, S.58]. Dabei müssen neben den funktionalen vor allem die nicht funktionalen Eigenschaften des Systems getestet werden. Oft werden nicht funktionale Anforderungen lückenhaft oder ungenau im Anforderungsdokument formuliert oder gelten als vorausgesetzt. Daher gilt es, nicht spezifizierte, aber dennoch relevante Eigenschaften zu validieren. Viele Projekte sind dadurch gefährdet, dass Anforderungen nicht oder ungenau definiert und dokumentiert wurden. Nicht selten kann der Systemtest dann nur das Scheitern des Projekts offiziell attestieren. Um die Umsetzung der Leistungs- und Qualitätsanforderungen zu prüfen, bedarf es entsprechender Testarten [37, S.72]:

- Lasttest
- Performanztest
- Volumentest
- Stresstest
- Test der Stabilität
- Test auf Robustheit
- Test unterschiedlicher Konfigurationen

Für die nicht funktionalen Anforderungen im Systemtest sollen Metriken für einen prüfbaren Fortschritt erhoben werden. Folgende Metriken wären hierfür denkbar [39, S.299]:

- Anzahl durch den Test abgedeckter Plattformen
- Anzahl durch den Test abgedeckter lokalisierter Versionen
- Anzahl durch den Test abgedeckter Performanzanforderungen pro Plattform

Ein realistisches Testziel für den Systemtest wäre z. B. eine Anforderungsabdeckung (engl. *requirements coverage*) von 100% sowie erfolgreiches Bestehen aller Testfälle mit Priorität 1. Für die Anforderungsüberdeckung wird das Anforderungsdokument (Lasten- oder Pflichtenheft) als Testbasis herangezogen. Für jede Anforderung muss mindestens ein Testfall erstellt worden sein. Zusätzlich sollte die Systemspezifikation durch ein Review verifiziert werden (vgl. Kap. 2.5.1). Auch hier lässt sich ein Fortschreiten der Testaktivitäten im Systemtest durch erheben von Metriken ermitteln [39, S.299]:

- Anzahl durch Testbedingungen abgedeckter Anforderungen / Anzahl aller Anforderungen
- Anzahl durch Testfälle abgedeckter Testbedingungen / Anzahl aller Testbedingungen
- Anzahl getesteter Funktionen / Anzahl spezifizierter Funktionen
- Anzahl durchgeführter Testfälle / Anzahl spezifizierter Testfälle (pro Funktion)
- Anzahl Teststunden pro Funktion

Dient das System der Automatisierung eines Geschäftsprozesses, muss geschäftsprozessbasiert getestet werden. Eine zuvor erstellte Geschäftsprozessanalyse filtert die relevanten Prozesse und untersucht diese nach Häufigkeit des Auftretens und Beteiligung von Fremdsystemen. Mithilfe dieser Analyse werden Testszenarien (Aneinanderreihung bzw. Aggregation von Testfällen, engl. *Testsuite*) zur Nachbildung typischer Geschäftsvorfälle erstellt. Häufigkeit und Relevanz der Geschäftsprozesse bestimmen die Priorität der Testszenarien.

Ähnlich ist das Verfahren für anwendungsfallbasiertes Testen. Zuvor erstellte *use cases* oder *user stories* helfen, typische Anwendungsfälle zu identifizieren. Diese Anwendungsfälle werden dann wieder in Testszenarien umgesetzt.

Obwohl sich das System noch in einer Testumgebung befindet, sollten zu diesem Zeitpunkt keine Testtreiber oder Platzhalter mehr verwendet werden, um die zukünftige Produktivumgebung optimal abzubilden. Diese Testumgebung sollte auch Basis für Fehlernachtests (*korrektive Maintenance*) und mögliche Regressionstests bei anstehenden Änderungen, Anpassungen oder Erweiterungen sein (*adaptive* und *enhanced Maintenance*).

Da die Größe des Programmcodes eines Gesamtsystems im Systemtest meist enorm ist (teilweise über eine Million LOC (*Lines of Code*)), kommen als Testtechniken dynamische Verfahren der Black-Box-Techniken in Frage. Es ist wenig sinnvoll und mit erheblichem Aufwand verbunden, erst beim Systemtest einzelne Anweisungen näher zu prüfen.

Eine Auswahl der Testarten und Testtechniken ist immer auch abhängig von dem jeweiligen Systemtyp. Jedes Produkt und Projekt ist einmalig in Bezug auf seine Anforderungen, Umstände, Betriebsmittel und Organisation. Nach Sneed [35, S.137] ist die Systemtestausführung für die nachstehenden Systemtypen zu unterscheiden:

- Tests alleinstehender Systeme
- Tests integrierter Systeme
- Tests verteilter Systeme
- Tests webbasierter Systeme
- Tests serviceorientierter Systeme
- Tests vollautomatischer Systeme
- Tests eingebetteter Echtzeitsysteme

Für die Aufwandsabschätzung des Systemtests und die Erreichbarkeit von Testzielen ist es hilfreich, die Systemtestbarkeit zu ermitteln. Sneed [35, S.16] nennt vier Testbarkeitsmerkmale (Anwendungsfälle, Benutzeroberflächen, Systemschnittstellen, Datenbank) und definiert entsprechende Metriken für deren Testbarkeit.

Die Anwendungsfälle betrachtend ist es für den Aufwand und gleichzeitig die Testwirtschaftlichkeit wünschenswert, deren Anzahl möglichst gering zu halten. Der Zusammenhang besteht darin, dass die Zahl der Testfälle proportional zur Anzahl der Anwendungsfälle und deren möglicher Ausgänge oder Pfade steigt. Weniger Anwendungsfallpfade pro Anwendungsfall begünstigen die Anwendungsfalltestbarkeit:

$$\text{Anwendungsfalltestbarkeit} = \frac{\text{Anwendungsfälle}}{\text{Anwendungsfallpfade}}$$

Für grafische Benutzeroberflächen (GUI, engl. *Graphical User Interface*) steigt der Testaufwand mit der Anzahl der enthaltenen Steuerungskomponenten (engl. *widgets*) in Relation zur Anzahl der Oberflächen. Aus der Sichtweise der Tester ist eine einfache Benutzeroberfläche daher wünschenswert:

$$\text{Benutzeroberflächentestbarkeit} = \frac{\text{Benutzeroberflächen}}{\text{Oberfächenobjekte}}$$

Benutzeroberflächen stellen für das Testen stets eine besondere Herausforderung dar. Zu selten wird eine strikte Trennung zwischen Oberflächenlogik und Verarbeitungslogik vorgenommen, wie es z. B. das MVC (*Model-View-Controller*) oder das ECB (*Entity-Control-Boundary* Pattern als Designleitlinien fordern. Sneed [35, S.18] weist auf einen Leitartikel von Robert C. Martin in der IEEE Software hin [28, S.65], in dem er den Nutzen eines Testbusses zur Umgehung der Benutzeroberfläche beim Testen bzw. im System- oder Regressionstest beschreibt. Weil sich Oberflächen während der Systementwicklung und dessen Weiterentwicklung besonders häufig ändern, müssen zahlreiche neue Tests entworfen und viele alte Tests wiederholt werden. Ein

Testbus soll den Zugriff auf die gleiche Schnittstelle, auf die auch die grafische Oberfläche zugreift, ermöglichen. So können die Testdaten unter Ausschluss der Benutzeroberfläche direkt in das Backend des Systems gelangen. Dies wäre eine Maßnahme für gut testbares Design (engl. *Design for Testability*).

Die Schnittstellentestbarkeit der Systemschnittstellen ist ein weiteres Merkmal für die Systemtestbarkeit. Für die Ermittlung spielen die Parameter bzw. Datenelemente des Systems eine zentrale Rolle. Dabei werden Datenelemente in drei Typen unterteilt: Eingaben, Prädikate und Ausgaben. Zu den Ausgaben zählen die Rückgabewerte, auch *Responses* genannt. Als Prädikate zählen Daten, die zur Steuerung im System verwendet werden. Sie bestimmen die Pfade durch das System. Eingaben sind die benötigten Argumente für die Systemfunktionen und Ausgaben sind die Ergebnisse der Systemfunktionen. Da die Ausprägungen der Prädikate verschiedene Systemverhalten auslösen, wächst die Komplexität einer Schnittstelle mit der Anzahl der möglichen Ausprägungen. Diese Tatsache berücksichtigt die Metrik, indem die Prädikate durch den Faktor 2 eine höhere Gewichtung erfahren. Ausgaben werden mit dem Faktor $1,5$ und Eingaben mit dem Faktor 1 gewichtet:

$$\text{Schnittstellentestbarkeit} = \frac{\text{Schnittstellen} + \text{Datenelemente}}{\text{Eingaben} + \text{Ausgaben} \cdot 1,5 + \text{Prädikate} \cdot 2}$$

Auch Datenbanken unterliegen einer Prüfung im Systemtest und werden beim Ermitteln der Systemtestbarkeit berücksichtigt. Als Vorbedingung (engl. *precondition*) für den Systemtest zählt das Vorhandensein geeigneter Testdaten in der Datenbank. Nach einem Test muss der Ist-Zustand mit dem Soll-Zustand der Datenbank verglichen werden. Je niedriger die Anzahl der möglichen Zustände ist, desto geringer ist der Aufwand für deren Erzeugung und Kontrolle:

$$\text{Datenbanktestbarkeit} = \frac{\text{Tabellen} + \text{Attribute}}{\text{Attribute} \cdot \text{Keys}}$$

2.4.4 Abnahmetest

Als letzter und sich auf der obersten Teststufe befindender Test folgt der Abnahmetest. Hauptakteure bei diesem Test sind der Kunde und evtl. Repräsentanten der zukünftigen Anwender. Dieser Test dient weder der Fehlerfindung, noch ist er ergebnisoffen. Vielmehr steht die Validierung des Systems durch die *Stakeholder* im Vordergrund. Es wird geprüft, ob das System den Wünschen und Ansprüchen sowie allen vertraglichen Abnahmekriterien entspricht. Daher ist es laut Spillner [37, S.62] äußerst wichtig, dass der Kunde die Testfälle im Abnahmetest selbst entwirft. Der vorherige Systemtest sollte diese Testfälle jedoch bereits berücksichtigt haben.

Die Durchführung des Abnahmetests geschieht nicht mehr in einer Testumgebung beim Hersteller, sondern in einer Abnahmeumgebung beim Kunden. Dies sollte aus Sicherheitsgründen noch nicht die zukünftige Produktivumgebung sein, ihr aber weitestgehend entsprechen.

Für manche Systeme ist es sinnvoll, nach dem Systemtest und vor dem Abnahmetest einen Feldtest durchzuführen. Hierzu werden Pilotkunden ausgewählt, um das System vor Marktfreigabe zu testen. Dies ist z. B. sinnvoll, wenn der Einsatz des Systems für verschiedene Produktivumgebungen vorgesehen ist. Geschieht der Feldtest beim Hersteller, bezeichnet man dies als *Alpha-Test*. Werden Vorabversionen bei den Pilotkunden getestet, nennt man dies *Beta-Test*. Der Vorteil des Alpha-Tests ist die Reproduzierbarkeit von Fehlern, da Fehlverhalten und Testaktivitäten direkt dokumentiert werden können.

2.5 Testtechniken

Für die zahlreichen Testarten kommen unterschiedliche Prüfmethoden zum Einsatz, wobei sich nicht jede Prüfmethode für jede Testart eignet. Die verschiedenen Testtechniken sind als Heuris-

tiken anzusehen. Sie geben eine sich bewährte Vorgehensweise vor, um nicht intuitive, sondern systematisch Testfälle ermitteln zu können.

Die Testtechniken werden grob untergliedert in statische und dynamische Testtechniken. Bei statischen Prüftechniken erfolgt eine Analyse aller mit der Software in Bezug stehender Dokumente inklusive des Quellcodes. Der Quellcode wird aber, im Gegensatz zu den dynamischen Testtechniken nicht ausgeführt. Die dynamischen Testtechniken basieren auf der Ausführung des Testobjekts auf einem Rechner.

2.5.1 Statischer Test

Der statische Test basiert auf manuellen Prüfungen von Dokumenten, wie z. B. Verträge, Anforderungsdokumente, Designspezifikationen, Quelltexte oder Testspezifikationen. Zur Prüfung semantischer Aspekte sind manuelle Verfahren unabdingbar. Diese manuellen Prüfungen werden u. a. in Form von **Reviews** durchgeführt. Die Intention statischer Testverfahren ist präventiv zu testen. Abweichungen sollen frühestmöglich, noch vor der Implementierung, erkannt werden. Da im statischen Test das Testobjekt nicht ausgeführt wird, ist die Wahrscheinlichkeit größer Fehlerzustände als Fehlerwirkungen aufzudecken.

Eine Unterkategorie des statischen Tests ist die statische Analyse. Statische Analysen prüfen automatisiert u. a. die Einhaltung von Programmierkonventionen (*Coding Rules*) durch *Coding Rule Checker* oder die Syntax des Quellcodes durch den Compiler. Dokumente wie z. B. Anforderungsdokumente lassen sich nur erschwert automatisiert prüfen. Voraussetzung hierfür ist eine formale Notation der Dokumente. Liegen Spezifikationen als formales Modell vor, z. B. als UML-Modell (*Unified Modeling Language*), können statische Model Checker Inkonsistenzen aufdecken.

Reviews

Neben den verschiedenen Reviewarten gibt es verschiedene Rollen, die in einem Review auftreten:

- Der **Autor** als Verfasser des zu prüfenden Dokuments. Er nimmt eine passive Rolle ein und seine Aufgabe besteht in der Vorbereitung des Dokuments als Eintrittskriterium und in der, wenn nötig, Nachbereitung des Dokuments als Austrittskriterium [37, S.85].
- Der **Manager** ordnet das Review an, wählt die Prüfobjekte und Teilnehmer des Reviews aus und stellt alle benötigten Ressourcen zur Verfügung.
- Der **Moderator** ist Leiter der Reviewsitzung und organisiert deren Ablauf.
- Die (max. fünf) **Gutachter** übernehmen die inhaltliche Prüfung des Testobjekts. Sie weisen auf verdächtige oder problematische, aber auch positive Stellen im Prüfling hin. Die Ernennung der Gutachter erfolgt durch den Manager oder den Moderator.
- Der **Protokollant** fasst den Verlauf und die Ergebnisse der Reviewsitzung zusammen und hält diese schriftlich fest.

Folgende Ausprägungen von Reviews werden unterschieden [37, S.87]:

- Ein **Walkthrough** weist den geringsten Aufwand aller Reviews auf, da es informellen Charakter hat und eignet sich daher für kleine Entwicklungsteams.
- Die **Inspektion**, auch **Softwareinspektion** genannt, stellt die formalste Art der Reviews dar. Der Aufwand hierfür ist hoch, da neben den zu prüfenden Dokumenten auch der Entwicklungsprozess analysiert wird.
- Beim **technischen Review** steht die Validierung im Vordergrund. Das Management und der Autor sind von der Sitzung ausgeschlossen. Der Vorbereitungsaufwand ist ebenfalls hoch und der Ablauf formal, wobei der Grad an Formalität aber sehr variieren kann.

- Das **informelle Review** ist in seinem Ablauf, wie der Name bereits andeutet, informell. Wegen seines geringen Aufwands wird es sehr geschätzt. Die in Kapitel 2.2 angesprochenen Verfahren wie *buddy testing*, *code swaps* oder *pair programming* gelten bereits als informelle Reviews.

Lassen sich die durch Reviews gefundenen Fehlerkosten beziffern, dann wird die Summe aller Fehlerkosten als *Nutzen* bezeichnet. Die Kosten für Reviews (das Produkt der Stundensätze aller Reviewteilnehmer mal deren aufgewendeter Stunden) beschreibt den *Aufwand*. Teilt man *Nutzen* durch *Aufwand*, erhält man den *ROI* (*Return on Investment*) [32, S.152]:

$$\text{ROI} = \frac{\text{Nutzen}}{\text{Aufwand}}$$

Ein ROI größer 1 gilt als positive Bilanz, da mehr Geld eingespart wurde als durch Kosten entstanden sind.

Reviews ergänzen sich gut mit statischen Analysen. Sind die für ein Review zu prüfenden Dokumente zuvor statisch analysiert wurden, verringert dies die Anzahl der Prüfaspekte im Review. Allerdings stellen Reviews keine Alternative zu anderen z. B. dynamischen Testverfahren dar. Im Unterschied zu anderen statischen Analysen, sind Reviews die einzigen Prüfmethoden die nicht automatisiert werden können.

Kontrollflussanalyse

Für die Durchführung einer Kontrollflussanalyse benötigt man die Darstellung eines Programmstücks in Form eines gerichteten Kontrollfussgraphs. Ein UML-Aktivitätsdiagramm gibt zwar auch den Kontrollfluss eines Programmstücks wieder, ist aber hierfür nicht granular genug. Für die Kontrollflussanalyse wird eine Abbildung aller Anweisungen benötigt, nicht der Aktivitäten. Dabei wird jede Anweisung als Knoten im Graph repräsentiert. Die Kanten jedes Knotens geben die Ausführungsreihenfolge wieder.

Die Überführung eines Programmstücks in einen Kontrollflussgraphen, sollte mittels eines Werkzeugs automatisiert durchgeführt werden, um eine eins-zu-eins Abbildung zu gewährleisten.

Durch die Analyse eines Kontrollflussgraphs werden Anomalien aufgedeckt, welche nicht grundsätzlich als Defekt einzustufen sind, aber Hinweise auf Irregularitäten bezüglich strukturierter Programmierung geben. Mögliche Anomalien sind z. B. das vorzeitige Herausspringen aus einer Schleife, unerfüllbare Bedingung zum Beenden einer Schleife (Endlosschleife), mehrere Ausgänge einer Funktion oder Methode oder unerreichbarer Code (engl. *dead code*).

Erweist sich der Kontrollflussgraph als unübersichtlich, weist dies auf unstrukturierten Quellcode oder sehr hohe Komplexität (vgl. Zyklomatische Komplexität) hin. In beiden Fällen sollte eine Überarbeitung des Programmstücks erfolgen.

Datenflussanalyse

Ähnlich wie die Kontrollflussanalyse befasst sich die Datenflussanalyse mit Anomalien hinsichtlich der Daten und deren Verwendung. Auch diese Anomalien sind nicht grundsätzlich als Fehler zu werten, sondern Indikatoren für mögliche Fehler. Ausgehend von den Zuständen definiert (**d**), referenziert (**r**) und undefiniert (**u**) ergeben sich drei Arten von Datenflussanomalien:

- **ur**-Anomalie: Lesender Zugriff des Programms auf eine nicht definierte Variable
- **du**-Anomalie: Gültigkeitsverlust einer definierten Variable, ohne vorherige Verwendung
- **dd**-Anomalie: Erneute Definition einer bereits definierten Variablen ohne zwischenzeitliche Verwendung

Das folgende Beispiel (Listing 1) verdeutlicht die o. g. Anomalien:

```
0  private void MinMax(int min, int max) {
           int hilf;
           if(min > max) {
           max = hilf; // ur-Anomalie der Variable hilf
           max = min;  // dd-Anomalie der Variable max
5          hilf = min;
           }
   }  // du-Anomalie der Variable hilf
      // (hilf nicht mehr im Scope von MinMax)
```

Listing 1: *Darstellung von Datenflussanomalien*

Zyklomatische Komplexität

Für die Ermittlung der Komplexität von Quellcode hilft die Berechnung der zyklomatischen Zahl, auch *McCabe-Metrik* genannt. Die McCabe-Metrik ist auch Hilfsmittel für die Analyse von Risiken, da eine (zu) hohe Komplexität ein erhöhtes Fehlerpotential in sich birgt. Aber auch Qualitätsmerkmale wie Testbarkeit und Wartbarkeit können durch sie ausgedrückt werden. Ausgangspunkt ist, wie bei der Kontrollflussanalyse (s. o.), ein Kontrollflussgraph (G). Sie berechnet sich dann nach folgender Formel [22, S.236]:

$$z(G) = e - n + 2$$

mit
e = Anzahl der Kanten des Graphen und
n = Anzahl der Knoten des Graphen

Nach McCabe gilt ein Messwert bis 10 als tolerabel. Da der erhaltene Wert auf einer Rationalskala liegt, lassen sich erhaltene Werte ins Verhältnis setzen. Erhält man als Messwert für ein Programmstück $z(G) = x$ und für ein anderes Programmstück den Messwert $z(G) = 2x$, dann ist die Komplexität des zweiten Programmstücks doppelt so hoch wie die des ersten.

Die McCabe-Metrik berücksichtigt die Anzahl an Verzweigungen im Kontrollflussgraphen. Mit zunehmender Anzahl an Verzweigungen (durch z. B. *if*-Anweisungen oder Schleifen) erhöht sich der Messwert und somit, nach McCabe, auch die Komplexität. Der erhaltene Wert repräsentiert gleichzeitig die Anzahl der unabhängigen Pfade in einem Programmstück. Soll im Zuge eines *White-Box-Tests* (vgl. Kap. 2.5.2) eine Zweigüberdeckung zu 100% erreicht werden, lässt der Messwert eine Aussage über die Testbarkeit zu.

Für die Deutung der McCabe Metrik ist wichtig, dass sich die Komplexität dabei nur aufs Testen bezieht. Je mehr Pfade getestet werden müssen, desto komplexer ist der Quellcode. Dabei ist zu beachten, dass ein nach McCabe komplexer Quellcode nicht gleichzusetzen ist mit schwer verständlichem Quellcode. Zwei Beispiele sollen dies verdeutlichen:

Die folgende *switch*-Anweisung (vgl. Listing 2) umfasst acht Kontrollflusspfade und hat somit nach McCabe eine zyklomatische-Komplexität von acht. Damit weist bereits dieses kleine Beispiel eine relativ hohe Komplexität auf. Aber trotz der relativ hohen Komplexität ist das Beispiel schnell erfassbar.

```
0  public String getWeekday(int day) {
       String weekday;
       switch(day) {
               case 1: weekday = "Monday"; break;
               case 2: weekeday = "Tuesday"; break;
5              case 3: weekday = "Wednesday"; break;
               case 4: weekday = "Thursday"; break;
               case 5: weekday = "Friday"; break;
               case 6: weekday = "Saturday"; break;
               case 7: weekday = "Sunday"; break;
10             default: weekday = null;
       }
return weekday;
}
```

Listing 2: *Switch-Anweisung mit zyklomatischer-Komplexität von 8*

Umgekehrt verhält es sich bei dem Beispiel in Listing 3 in Anlehnung aus [32, S.65] zur Berechnung eines Schaltjahres. Hier ist eine zyklomatische-Komplexität von eins realisiert. Ohne Kommentare ist die *return*-Anweisung allerdings nur schwer nachvollziehbar.

```
0  public boolean isLeapYear(int n) {
       return (((n % 4 == 0) && (n % 100 != 0)) || (n % 400 == 0));
}
```

Listing 3: *Berechnung eines Schaltjahres mit zyklomatischer-Komplexität von eins*

2.5.2 Dynamischer Test

Beim dynamischen Test wird, im Unterschied zum statischen Test, das Testobjekt zur Ausführung gebracht. Für die Ermittlung der Testfälle wird dabei zwischen zwei Strategien unterschieden. Die erste betrachtet das Testobjekt als „schwarzen Kasten“ (**Blackbox-Test**, vgl. Abb. 5) dessen innerer Aufbau und innere Struktur nicht zu erkennen ist. Die Ermittlung der Testfälle geschieht auf Basis der Anforderungen und Systemspezifikation. Das Verhalten des Testobjekts während des Tests ist nur von außen, anhand der Resultate zu beobachten (*Point of Observation; PoO*). Eine Einflussnahme auf die Steuerung des Ablaufs ist nur durch die Wahl von Eingabedaten möglich (*Point of Control; PoC*). Die zweite Strategie berücksichtigt ein Hineinblicken in das Testobjekt (**Whitebox-** oder **Glassbox-Test**, vgl. Abb. 6). Die inneren Strukturen dienen als Grundlage für die Testfallermittlung. Während der Testausführung werden die Strukturen geprüft (*Point of Observation*) und in diese gegebenenfalls auch eingegriffen (*Point of Control*).

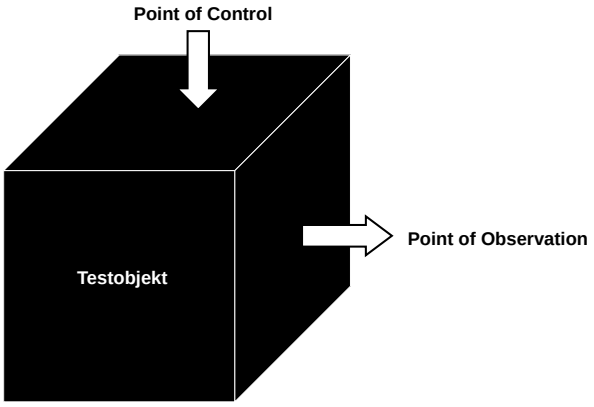

Abb. 5: *Blackbox-Test*

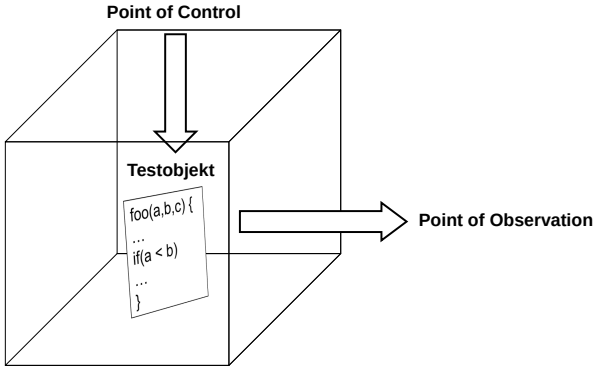

Abb. 6: *Whitebox-Test*

Blackbox-Verfahren

Zu den Blackbox-Verfahren zählen:

- Äquivalenzklassenbildung
- Grenzwertanalyse
- Zustandsbezogener Test
- Ursache-Wirkungs-Graph-Analyse
- Anwendungsfallbasierter Test

Für alle Blackbox-Verfahren gilt, dass sie keine Fehler aufgrund einer fehlerhaften Spezifikation oder Anforderung aufdecken können. Ein Verhalten eines Testobjekts, was konform zur Dokumentation ist wird immer als richtig angenommen. Auch zusätzliche Funktionalitäten, die über die Anforderungen oder Spezifikation hinaus gehen können nicht oder nur zufällig aufgedeckt werden. Da die Funktionalität eines Testobjekts im Fokus der Blackbox-Verfahren liegt, sind sie vor allem im System- und Abnahmetest unverzichtbar.

Whitebox-Verfahren

Vertreter der Whitebox-Verfahren sind:

- Anweisungsüberdeckung
- Zweigüberdeckung
- Test der Bedingungen
- Pfadüberdeckung

Die Whitebox-Verfahren eignen sich vornehmlich für die unteren Teststufen. Fehlende Anforderungen können durch sie nicht aufgedeckt werden. Da der Aufwand der Verfahren meist sehr hoch ist, sollten hierfür immer die zahlreich zur Verfügung stehenden Werkzeuge zum Einsatz kommen.

3 Evaluierung

Dieser Abschnitt beschäftigt sich mit der Evaluierung systematischer Testfallermittlungsverfahren auf Anwendbarkeit im Systemtest für den Fachbereich bei Logica. Für die Bewertung kommen nur Techniken in Betracht, die eine Testfallerstellung aufgrund einer systematischen Ermittlung von Testdaten oder deren Kombinationen zulassen, um funktionale Anforderungen zu prüfen bzw. „[...] das Verhalten des Softwaresystems hinsichtlich der Anforderungen zu falsifizieren[...]“ [7, S.454].

> „Für den Integrations- und Systemtest reicht eine rein funktionsorientierte Testplanung und Testdurchführung aus“ [22, S.476].

Wichtig für die Eignung im Systemtest ist, dass als Quelle für die Ermittlung von Testfällen nur die Anforderungsdokumentation oder Systemspezifikationen in Frage kommen. Verfahren, deren Grundlage der Programmtext oder dessen innerer Aufbau sind, werden vorab als ungeeignet für den Systemtest bewertet. Da die Testspezialisten von Logica oftmals erst während eines laufenden Projekts bzw. in einem späten Entwicklungsstadium vom Kunden involviert werden, soll in diesen Fällen eine Einarbeitung in den Quellcode für den Systemtest vermieden werden. Entsprechend kann auch nicht von einheitlichen oder typischen Formen der Systemspezifikationen ausgegangen werden.

Da der Systemtest als letzter Test vor dem Abnahmetest sich auf einer fortgeschrittenen Teststufe befindet, ist zwingend davon auszugehen, dass zu diesem Zeitpunkt ein lauffähiges Testobjekt vorliegt. Das *System under test* soll zur Ausführung gebracht werden. Daher werden für die Untersuchung ausschließlich dynamische Testverfahren gewählt, die aufgrund der o. g. Bedingungen auf Black-Box-Verfahren eingeschränkt werden können. Folgende Verfahren kommen in Betracht und werden, nach Absprache mit Logica, für die anschließende Evaluierung herangezogen:

- Äquivalenzklassenbildung
- Grenzwertanalyse
- Klassifikationsbaummethode
- Orthogonale Arrays

Die vier Verfahren erlauben keinen direkten Vergleich miteinander, da sie unterschiedliche Zwecke erfüllen. Äquivalenzklassenbildung und Grenzwertanalyse dienen vorrangig der Ermittlung von Testdaten. Die Klassifikationsbaummethode unterstützt die Ermittlung von Testaspekten. In Kap. 3.4.1 wird der *Classification Tree Editor* vorgestellt, der als unentbehrliches Werkzeug für die Klassifikationsbaummethode zusätzlich u. a. die Kombination von zu testenden Aspekten oder Eingabedaten zu Testfällen vorsieht. Dies ist auch Aufgabe der orthogonalen Arrays, wobei beide Verfahren bei der Bildung der Testfallkombinationen gleichzeitig eine Reduzierung der Testfälle ermöglichen.

Aufgrund der Heterogenität des Einsatzzwecks der Verfahren, Testdatengenerierung und Generierung von Tesfallkombinationen, folgt in Kap. 3.6 eine abschließende Bewertung in Bezug auf Kombinationsmöglichkeiten dieser Methoden.

3.1 Evaluierungskriterien

Ein wichtiges Kriterium für die Evaluierung der Verfahren ist das Verhältnis von Aufwand und Nutzen. Hierfür ist anzumerken, dass eine Bewertung dessen immer subjektiven Charakter hat und abhängig vom zu testenden System ist. Für Systeme, die im Fehlerfall eine Gefahr für Gesundheit oder Leben darstellen, kann auch ein Verhältnis von geringem Nutzen und hohem Aufwand gerechtfertigt sein. Zwei Merkmale für das Verhältnis sind die Einhaltung der Mi-

nimalforderung und des Effizienzprinzips [32, S.93]. Für die Minimalforderung gilt, dass jede Anforderung durch (mindestens) einen Testfall abgedeckt sein muss bzw. alle Testfälle jede Anforderung und somit die gesamte Spezifikation abdecken. Während das Minimalprinzip eine minimale Anforderungsüberdeckung fordert, schreibt das Effizienzprinzip eine Beschränkung der Testfälle vor. Dies bedeutet die Erstellung möglichst weniger Testfälle, mittels Abdeckung mehrerer Anforderungen durch einen Testfall. Es ist auch zu betrachten, ob der Aufwand mithilfe geeigneter Werkzeuge reduziert werden kann. Als weiteres Kriterium ist zu prüfen, ob die anhand der Verfahren zu ermittelnden Testdaten auch Testfälle für Negativtests berücksichtigen. Ferner ist die Eindeutigkeit der Verfahren zu beurteilen. Ist ein Verfahren nicht eindeutig, ist eine falsche Umsetzung wahrscheinlich und damit verbunden der Verlust der Aussagekraft. Darüber hinaus soll geprüft werden, inwiefern die Verfahren es ermöglichen, Prioritäten oder Risikofaktoren einfließen zu lassen. Außerdem ist die mögliche Kombination der Verfahren ein wichtiger Bestandteil der Beurteilung.

Anhand der o. g. Kriterien wird ein Kriterienkatalog (vgl. Tabelle 2) aufgestellt, der einer abschließenden Bewertung der einzelnen Verfahren dient. Um eine unterschiedliche Gewichtung der einzelnen Kriterien zu berücksichtigen, werden sie priorisiert. Die Prioritäten werden nach dem folgenden Abstufungskriterium ausgewählt:

- Höchste Priorität (**3**): Das Kriterium stellt eine wichtige Eigenschaft für den Systemtest dar.
- Mittlere Priorität (**2**): Das Kriterium stellt eine wünschenswerte Eigenschaft für den Systemtest dar.
- Niedrige Priorität (**1**): Das Kriterium stellt eine untergeordnete, aber nicht unbedeutende Eigenschaft für den Systemtest dar.

Die Bewertung der Erfüllung eines Kriteriums zeigt das folgende Schema:

- $+$: Kriterium sehr zufriedenstellend erfüllt (Rechnungswert: 2)
- $+ / -$: Kriterium erfüllt (Rechnungswert: 1)
- $-$: Kriterium nicht zufriedenstellend erfüllt (Rechnungswert: 0)

Kriterium	**Priorität**	**Bewertung**	**Kommentar**
Aufwand und Nutzen	3		
Einsatz von Werkzeugen	2		
Negativtests	1		
Berücksichtigung von Prioritäten	2		
Eindeutigkeit der Heuristik	3		
Ergebnis			

Tabelle 2: *Kriterienkatalog für die Evaluierung*

Die Bestimmung des Gesamtergebnisses erfolgt durch Mittelwertbildung der Einzelergebnisse, wobei der für die Erfüllung eines Kriteriums vergebene Wert mit der Priorität des Kriteriums multipliziert wird.

Definition der Kritierien

Aufwand und Nutzen beschreiben wie aufwändig (inkl. Lernaufwand) die Anwendung eines Verfahrens ist unter Berücksichtigung vorhandener Werkzeuge sowie den zu erwartenden Nutzen gegenüber intuitiver Testfallerstellung im Systemtest

Einsatz von Werkzeugen beurteilt, falls Werkzeuge zur Verfügung stehen, inwiefern die Effizienz eines Verfahrens dadurch gesteigert wird

Negativtests betrachtet die Möglichkeit, ob und wie ungültige Testdaten einbezogen werden können

Berücksichtigung von Prioritäten begutachtet inwieweit höher priorisierte Testdaten oder Testkombinationen einbezogen werden

Eindeutigkeit der Heuristik bewertet, ob eine Verfahrensbeschreibung eine konkrete Umsetzung erlaubt im Sinne einer allgemeingültigen Anleitung

Vergabe der Prioritäten

Aufwand und Nutzen stellt ein äußerst wichtiges Kriterium (Priorität 3) dar, da aufwändige Verfahren zeit- und somit kostenintensiv sind, was nur durch einen erhöhten Nutzen gerechtfertigt werden kann. Der Einsatz von Werkzeugen kann helfen Zeit und Kosten zu sparen und dadurch die Effizienz eines Verfahrens steigern. Dies wird als wünschenswertes Kriterium (Priorität 2) bewertet ist aber evtl. mit Anschaffungskosten verbunden. Negativtests als fehlerorientiertes Verfahren sind wünschenswert, besitzen im Systemtest allerdings einen geringeren Stellenwert, da hier demonstratives statt destruktives Testen Vorrang hat. „[...]Demonstration, dass keine Fehler vorhanden sind - sollte erst nach der Entwicklung des Systems benutzt werden, wenn man versucht zu zeigen, dass das gelieferte System die funktionalen und die nichtfunktionalen Anforderungen erfüllt“ [7, S.457]. Eine Berücksichtigung von Prioritäten hilft die Effektivität eines Verfahrens zu erhöhen und wird als wünschenswertes Kriterium (Priorität 2) eingestuft. Die Eindeutigkeit eines Verfahrens ist wichtig für dessen korrekte Anwendung. Daher wird diesem Kriterium, wie Aufwand und Nutzen, besonderes Gewicht beigemessen (Priorität 3).

3.2 Evaluierung Äquivalenzklassenbildung

Äquivalenzklassen können für verschiedene Werte gebildet werden. Vor allem die Durchführung für Eingabe- und Ausgabewerte eignen sich für den Systemtest, da diese Werte der Spezifikation zu entnehmen sind. Die Möglichkeit, Äquivalenzklassen auch für interne Werte, Zustände oder Schnittstellenparameter zu bilden, ist eine weitere Stärke der Äquivalenzklassenbildung [37, S.120]. Da sich die Ausführung der Tests hierdurch zum *Grey-Box-Test* wandelt, ist hierfür eher ein Nutzen im Komponententest oder Integrationstest zu sehen. Die Bildung ungültiger Äquivalenzklassen unterstützt die Ausführung von Negativtests. Hierbei ist zu beachten, dass Repräsentanten ungültiger Äquivalenzklassen nicht mit Repräsentanten anderer ungültiger Äquivalenzklassen in einem Testfall miteinander kombiniert werden. Dies birgt die Gefahr einer Fehlermaskierung oder Fehlerwirkung, die einem Repräsentanten nicht mehr eindeutig zugewiesen werden kann.

Der Testaufwand kann durch die Bildung der Klassen erheblich reduziert werden. Als Minimalkriterium sollte sichergestellt werden, „[...]dass jeder Repräsentant einer Äquivalenzklasse in mindestens einem Testfall vorkommt.“ [37, S.116]. Der Aufwand für die Einteilung der Eingabe- bzw. Ausgabeparameter in Äquivalenzklassen ist nicht zu unterschätzen. Die Anzahl der zu bildenden Klassen kann schon bei wenigen Parametern schnell steigen. Nach Spillner [37, S.118] wird die Zerlegung der Parameter oft aufgrund von Mangel an Zeit, Detailinformationen oder zur Reduzierung des Aufwands auf einer bestimmten Stufe abgebrochen. Häufig werden auch nicht alle möglichen Äquivalenzklassen erkannt. Dadurch wird provoziert, dass die einzelnen Klassen nicht disjunkt sind, und somit Repräsentanten einer Klasse evtl. doch verschiedenartiges Verhalten aufweisen. Auch eine Aussage bezüglich der Aquivalenzklassenüberdeckung wird dadurch negativ beeinflusst. Die Formel zur Berechnung der Äquivalenzklassenüberdeckung berücksichtigt lediglich die Gesamtzahl aller definierten Äquivalenzklassen. Dies

beinhaltet nicht die Erfassung der Gesamtzahl aller Äquivalenzklassen. Die Folge ist eine Verfälschung des Ergebnisses. „Werden nicht alle Äquivalenzklassen erkannt und somit zu wenige ermittelt und mit Repräsentanten getestet, wird zwar ein hoher Überdeckungsgrad erreicht, der aber aufgrund einer falschen Gesamtzahl der Äquivalenzklassen berechnet wird“ [37, S.120]. Die Äquivalenzklassenüberdeckung wird nach folgender Formel ermittelt [37, S.119]:

$$\text{ÄK-Überdeckung} = \frac{\text{Anzahl getestete ÄK}}{\text{Gesamtzahl ÄK}} \cdot 100\%$$

Die Spezifikationsabdeckung mittels Äquivalenzklassenbildung kann durch Anwendung der Minimalforderung und des Effizienzprinzips optimiert werden. Zur Veranschaulichung dient eine Methode mit drei Parametern. Für die Unterteilung der Parameter in Äquivalenzklassen ergeben sich für den ersten Parameter x, für den zweiten Parameter y und für den dritten Parameter z verschiedene Äquivalenzklassen, mit $x = 5$, $y = 5$ und $z = 6$. Für die vollständige Erfüllung der Minimalforderung ergibt sich durch Kombination aller Repräsentanten folgende Testfallanzahl:

$$x \cdot y \cdot z = \text{Anzahl Testfälle} \Rightarrow 5 \cdot 5 \cdot 6 = 150$$

Dieses Vorgehen widerspräche jedoch dem Effizienzprinzip. Durch systematisches Vorgehen kann bei geschickter Kombination die Anzahl der Testfälle drastisch reduziert werden. Ausgehend von dem Parameter mit den meisten Äquivalenzklassen (z) wird je ein Repräsentant in einen Testfall überführt. Dabei werden die Vertreter der anderen Äquivalenzklassen so variiert, dass sie mit abgedeckt werden. Daraus folgt eine Einschränkung der Testfälle auf:

$$max(x, y, z) = \text{Anzahl Testfälle} \Rightarrow max(5, 5, 6) = 6$$

So kann mithilfe des Effizienzprinzips, unter Einhaltung der Minimalforderung, die Anzahl der Testfälle theoretisch auf sechs reduziert werden. Theoretisch bedeutet in diesem Zusammenhang, dass dabei noch nicht berücksichtigt wurde, ob ein Testfall eine Kombination von Vertretern verschiedener ungültiger Äquivalenzklasssen beinhaltet, was eine Fehlermaskierung bewirken kann. Schreibt die Anforderungsdokumentation bestimmte Kombinationen vor, dürfen diese durch das Effizienzprinzip aufgrund der Minmalforderung nicht vernachlässigt werden. Undokumentierte Wechselwirkungen zwischen Äquivalenzklassen werden nicht berücksichtigt. „Falls Software-Reaktionen an eine ganz bestimmte Verknüpfung von Äquivalenzklassen gebunden sind, kann dies bei der funktionalen Äquivalenzklassenbildung nicht geeignet beschrieben werden“ [22, S.53]. Bestehende Abhängigkeiten aufgrund bestimmter Eingabeverknüpfungen müssen durch zusätzliche Testfälle abgedeckt werden.

Die Vorgehensweise der Äquivalenzklassenbildung ist eindeutig und einfach. Für das sorgfältige Auffinden aller Äquivalenzklassen sollte jedoch ein adäquater Zeitraum eingeräumt werden. Da jedes System in einer Weise auf Eingabedaten durch einen Benutzer oder anderer Systeme angewiesen ist, worauf dann eine Äquivalenzklassenbildung angewandt werden kann, ist die Methode im Systemtest als allgemeingültig einzuschätzen. Für den Einsatz im Integrationstest können schnittstellenbasierte Testdatengeneratoren die Äquivalenzklassenbildung sowie die anschließende Generierung von Testdaten automatisiert durchführen. Im Systemtest ist eine entsprechende Werkzeugunterstützung an Vorbedingungen geknüpft. Spezifikationsbasierte Testdatengeneratoren benötigen eine Spezifikation, die in einer formalen Notation, z. B. als UML-Modell oder systematischen Notation z. B. als Tabelle von Bereichen der Eigabeparameter, vorliegt.

Als schwächstes Glied der Kette kann sich eine fehlerhafte Spezifikation herausstellen. Alle Tests auf Basis der Äquivalenzklassenbildung, sind nur so gut wie die auf sie zurückzuführende Spezifikation. Fehlerfreie Tests auf Grundlage einer fehlerbehafteten Spezifikation wären die

Folge und ließen sich nicht aufdecken. Auch eine Behandlung von Werten im Quellcode, die nicht der Spezifikation entsprechen, könnte unberücksichtigte Äquivalenzklassen zur Folge haben. Daher sollte nach Bath [4, S.36] Rücksprache der Tester mit den Softwareentwicklern und ggf. Softwaredesignern und -architekten gehalten werden.

Äquivalenzklassen helfen, unnötige Testfälle zu vermeiden, sodass die Testeffizienz gesteigert und die damit verbundenen Testkosten reduziert werden. Vor allem in Verbindung mit der Grenzwertanalyse bezeichnet Spillner sie als ein „[...]sehr wirkungsvolles Verfahren.“ [37, S.121]. Sie schaffen „[...]Gewissheit, alle erkennbar verschiedenen Fälle durch je einen Testfall abgedeckt zu haben.“ [32, S.96]. Eine Bevorzugung risikoreich oder prioritär einzuschätzender Testfallkombinationen lässt das Verfahren zu, berücksichtigt dies aber nicht in seiner Heuristik. Da der Aufwand gegenüber dem Nutzen als akzeptabel einzuschätzen ist und bei formaler Spezifikation durch Automatisierung beschränkt werden kann, sollte eine Äquivalenzklassenbildung unter Beachtung der zuvor genannten Hinweise und Schwächen im Systemtest unbedingt durchgeführt werden. Hieraus ergibt sich folgende Bewertung:

Kriterium	**Priorität**	**Bewertung**	**Kommentar**
Aufwand und Nutzen	3	+ / −	Aufwand kann schnell ansteigen; hoher Nutzen
Einsatz von Werkzeugen	2	−	Werkzeugeinsatz im Systemtest nur bei formaler Notation möglich
Negativtests	1	+	durch Bildung ungültiger Äquivalenzklassen möglich
Berücksichtigung von Prioritäten	2	+	genügend Freiraum bei Auswahl der Repräsentanten
Eindeutigkeit der Heuristik	3	+	eindeutig in der Theorie, in der Praxis oft unterschätzt
Ergebnis	3.0		

Tabelle 3: *Bewertung der Äquivalenzklassenbildung*

3.3 Evaluierung Grenzwertanalyse

Die Grenzwertanalyse gehört zu den fehlerorientierten Testtechniken und sollte in Ergänzung zur Äquivalenzklassenbildung durchgeführt werden. Vor allem an den Grenzen der Definitionsbereiche von Parametern oder an den Grenzen von Äquivalenzklassen ist, entgegen der Theorie, häufig ein nicht äquivalentes Verhalten zu beobachten. Der Grund hierfür sind in vielen Fällen falsch definierte Bedingungen bei *if*-Anweisungen oder falsche Abbruchbedingungen bei Schleifen. Solche Fallunterscheidungen sind oft fehlerträchtig (*off-by-one*-Problem). In Kombination mit der Äquivalenzklassenbildung „erbt“ die Grenzwertanalyse die in Kap. 3.2 genannten Stärken und Schwächen.

Durch Ermittlung und Überprüfung von Grenzwerten, die über die Grenzen hinaus gehen, unterstützt das Verfahren Negativtests. Sind Äquivalenzklassen zuvor erstellt worden, ist der Aufwand gering, wächst aber mit steigender Anzahl ermittelter Äquivalenzklassen. Dem Effizienzprinzip und der Minimalforderung folgend, können auch hier die Grenzwerte der jeweiligen Äquivalenzklassen so kombiniert werden, dass sich die Anzahl der Testfälle reduziert und doch alle Äquivalenzklassen durch mindestens einen Repräsentanten abgedeckt werden.

In Analogie zur Formel für die Äquivalenzklassenüberdeckung lässt sich die Grenzwertüberde-

ckung folgendermaßen berechnen [37, S.128]:

$$\text{GW-Überdeckung} = \frac{\text{Anzahl getestete GW}}{\text{Gesamtzahl GW}} \cdot 100\%$$

Wurden nicht alle Äquivalenzklassen ermittelt, beeinflusst dies auch den Überdeckungsgrad der Grenzwerte. Dieser stützt sich auf die Gesamtzahl der Grenzwerte in Bezug auf zuvor ermittelte Äquivalenzklassen.

Die Vorgehensweise ist, wie bei der Äquivalenzklassenbildung, eindeutig. In der Praxis sollte die Handhabung aber nicht unterschätzt werden, auch wenn sie trivial erscheint. Ist eine Ermittlung von „Nachbarn“ bei ganzzahligen Werten noch eindeutig, kann dies bei komplexen Datenstrukturen (Bäume, Listen etc.) und Gleitkommazahlen zu Missverständnissen führen. Auch hier kann der Aufwand durch die gleichen spezifikationsorientierten Testdatengeneratoren, wie bei der Äquivalenzklassenbildung, reduziert und der Vorgang der Grenzwertermittlung automatisiert werden.

In gleicher Weise wie die Äquivalenzklassenbildung ist auch die Grenzwertanalyse von möglichen Fehlern in der Spezifikation betroffen und kann diese nicht aufdecken.

In Kombination mit der Äquivalenzklassenmethode ist die Grenzwertanalyse eine wirksame Ergänzung. Durch Überprüfung der Grenzen von Äquivalenzklassen setzt sie dort an, wo das Potential für Fehler groß ist. Da mangelnde Sorgfalt bei der Äquivalenzklassenbildung auf die Grenzwertanalyse durchschlägt, wird ihre Effizienz von der geleisteten Vorarbeit beeinflusst.

Die Grenzwertanalyse erhöht die Aussagekraft der Äquivalenzklassentests bei geringem Aufwand und sollte daher obligatorisch in Ergänzung zur Äquivalenzklassenbildung im Systemtest zum Einsatz kommen.

Daraus ergibt sich die in Tabelle 4 dargestellte Einschätzung.

Kriterium	Priorität	Bewertung	Kommentar
Aufwand und Nutzen	3	+	Aufwand gering, wächst in Kombination mit Äquivalenzklassen mit deren Anzahl
Einsatz von Werkzeugen	2	−	Werkzeugeinsatz im Systemtest nur bei formaler Notation möglich
Negativtests	1	+	durch Wahl von Repräsentanten die Grenzen über- oder unterschreiten
Berücksichtigung von Prioritäten	2	+ / −	nur wenn priorisierte Aspekte sich über die Grenzwerte testen lassen
Eindeutigkeit der Heuristik	3	+	eindeutiges Verfahren
Ergebnis	3.2		

Tabelle 4: *Bewertung der Grenzwertanalyse*

3.4 Evaluierung Klassifikationsbaummethode

Die Klassifikationsbaummethode (engl. *Classification Tree Method*; CTM) wurde im Software-Forschungslabor der ehemals Daimler-Benz AG, jetzt Daimler AG, entwickelt und stellt eine Weiterentwicklung der *Category Partition Method* dar. Sie „[...] unterstützt das systematische Design von Blackbox-Tests.“ [45, S.149] und basiert auf der Bildung von Äquivalenzklassen in einer Baumstruktur. Im Gegensatz zur Äquivalenzklassenmethode ist die Klassifikationsbaummethode nicht nur auf Ein- oder Ausgabewerte beschränkt und die Bildung dieser Äquivalenzklassen beruht auf anderen Heuristiken. Sie kombiniert Bedingungen zur Ableitung von Testfallkombinationen. Es werden Äquivalenzklassen von Fällen oder Aspekten eines Testobjekts auf visuellem Weg ermittelt.

Innerhalb eines Klassifikationsbaums werden Aspekte als Klassifikationen dargestellt. Klassifikationen werden in Klassen unterteilt, aus denen rekursiv weitere Klassifikationen mit weiteren Klassen hervorgehen können. Die Blätter des Baums bilden Klassen, denen z. B. mithilfe der Äquivalenzklassenanalyse systematisch gegliederte Eingabebereiche zugeordnet werden können. Gleichzeitig beschreiben die Blätter die Kopfzeile einer Kombinationstabelle, wie z. B. in Tabelle 5 zu sehen ist. Testfälle ergeben sich aus der Kombination von Repräsentanten unterschiedlicher Klassen, wobei aus jeder Klassifikation nur eine Klasse berücksichtigt wird. Für eine bessere Dokumentation des Testansatzes und zur Unterstützung der Systematik ist es empfehlenswert, keine konkreten Werte, sondern Wertebereiche anzugeben.

Die Klassifikationsbaummethode eignet sich hauptsächlich für Eingabewerte, obwohl auch Klassen für Ausgabewerte möglich sind. Sie unterstützt Negativtests mittels Bildung von Ausnahme- oder Fehlerfällen. Es empfiehlt sich, hierfür einen weiteren Klassifikationsbaum zu entwerfen, um die fachlich gewünschten Abläufe von den fachlichen Ausnahmen getrennt betrachten zu können [45, S.151].

Die Eindeutigkeit der Vorgehensweise ist abstrakter als bei den zuvor genannten Methoden. Vigenschow [45, S.149] beschreibt die Vorgehensweise in vier Schritten:

1. Identifizierung der Einflüsse auf die Datenverarbeitung im System, auch *Aspekte* genannt.
2. Partitionierung des Eingabebereichs für die bereits identifizierten Aspekte. Dazu werden

die Eingaben in Klassen eingeteilt. Dabei muss sich das System für jede Eingabe eines Wertes aus einer Klasse, wie von Äquivalenzklassen gefordert, gleich verhalten.

3. Spezifizierung von logischen Testfällen durch Auswahl einer Klasse für jeden Aspekt und anschließende Kombination dieser Klassen.
4. Konstruktion physischer Testfälle und Bestimmung deren Ausgangssituationen.

Vor allem der erste Schritt verdeutlicht die Abstraktionsebene dieses Verfahrens. Werden hier nicht alle Einflüsse erkannt, zieht sich dieser Mangel bis Punkt vier zur Konstruktion der Testfälle hin und mindert deren Qualität.

In Anlehnung an Bath [4, S.56] wird als Vorgehensweise empfohlen, die ermittelten Aspekte zunächst in einer Tabelle mit Eingabeparameter (engl. *Input Parameter Model*) zusammenzustellen (Tabelle 5).

Clients	**Browser**	**Sprachen**	**Datenbanken**	**Server**
Windows A	Browser A	Englisch	Datenbank A	Unix
Windows B	Browser B	Französisch	Datenbank B	Linux
Windows C	Browser C	Deutsch	Datenbank C	Solaris
Mac A	Browser D	Arabisch		
Mac B		Spanisch		
Mac C				

Tabelle 5: *Konfigurationstabelle in Anlehnung an [4, Abb. 4-10]*

Anhand dieser Tabelle kann anschließend der Klassifikationsbaum erstellt werden. Die erste Spalte bildet dabei einen Teilbaum mit der Klassifikation *Clients* und den Klassen *Windows A, Windows B, Windows C, Mac A, Mac B, Mac C*. Durch Kombination der Blätter einzelner Klassen können dann Testfälle ermittelt werden, die in einer weiteren Tabelle festzuhalten sind.

Auch die Klassifikationsbaummethode erfüllt das Effizienzprinzip und die Minimalforderung. Durch Aufnahme eines Wertes jeder Äquivalenzklasse in allen Testfällen wird die Minimalforderung eingehalten. Durch geschicktes Kombinieren und variieren der Teilbäume kann die Anzahl der Testfälle auf ein Minimum reduziert werden. Besteht ein Klassifikationsbaum aus zwei Teilbäumen T_1 und T_2, wobei T_1 vier Äquivalenzklassen und T_2 zwei Äquivalenzklassen beinhaltet, berechnet sich die minimale Anzahl an Testfällen wieder folgendermaßen:

$$max_{\text{Äquivalenzklassen}}(T_1, T_2) = \text{Anzahl Testfälle} \Rightarrow max(4, 2) = 4$$

Die Klassifikationsbaummethode hilft, geeignete Testfälle zu identifizieren. Die Stärke dieser Methode ist dabei vor allem ihre visuelle Darstellung. „Jedes Verfahren, das die bildhafte Vorstellung stärkt, kann den Entwurf von Tests erleichtern[...]“ [4, S.58]. Dies ist besonders hilfreich, wenn wenig dokumentierte Spezifikationen vorliegen. Zugleich kann sich dies auch als Schwäche herausstellen. Wenn die Bäume zu groß werden, kann dies schnell zu Unübersichtlichkeit führen. Im Zusammenhang mit der Äquivalenzklassenbildung lassen sich für die jeweiligen Klassen geeignete Testdaten ermitteln. Das Kap. 3.6 geht auf die Kombination von Äquivalenzklassenbildung, Grenzwertanalyse und Klassifikationsbaummethode genauer ein. Die Erstellung eines Baums und die Ableitung entsprechender Testfälle sollte nur mittels eines Werkzeugs geschehen (s. Kap. 3.4.1).

Im Systemtest können Klassifikationsbäume sehr hilfreich sein. Für den Nachweis der Anforderungsüberdeckung gegenüber *Stakeholdern* ist die Visualisierung sehr vorteilhaft. Die Heuristik zur Ermittlung testrelevanter Aspekte ist abstrakt. Daher sind Kreativität und Erfahrung des

Testers gefordert. Es besteht die Möglichkeit, dass aufgrund unvollständiger Bäume relevante Aspekte nicht getestet werden und Fehler evtl. unerkannt bleiben.

Eine Bewertung nach dem Kriterienkatalog erfolgt in Kap. 3.4.1, da die Anwendung der Klassifikationsbaummethode nur mit dem dort beschriebenen Werkzeug durchgeführt werden sollte.

3.4.1 Klassifikationsbaum-Editor

Ein unverzichtbares Werkzeug für die Klassifikationsbaummethode ist der ebenfalls von der Daimler AG entwickelte Klassifikationsbaum-Editor (engl. *Classification Tree Editor*; CTE). Dieser wurde bereits in zahlreichen Projekten erfolgreich angewendet [47, S.101]: elektronisches Schiffstagebuch, Leitsystem für die Flugfeldbefeuerung eines internationalen Großflughafens, Erkennungssystem für Briefverteilanlagen und Formularleser.

Der CTE ist ein graphischer Editor und unterstützt die Tester in der Entwurfsphase des Baums sowie bei der Kombination einzelner Testfälle. Im Zuge der Weiterentwicklung des CTE entstanden der CTE/ES (*Embedded System*) im Rahmen einer Dissertation an der Technischen Universität Berlin und der CTE/XL (*eXtended Logics*) bei der Daimler AG.

Während der ursprüngliche CTE nur eine manuelle Testfallgenerierung zuließ, bietet der CTE/XL eine automatisierte Lösung hierfür. Mittels Regeln können beim CTE/XL logische Abhängigkeiten gesetzt werden. Dadurch können unvereinbare Kombinationen zwischen Klassen oder unlogische Testfälle verhindert und noch nicht vorhandene Testfälle vervollständigt werden. Darüber hinaus können Kombinationsregeln (z. B. paarweise, dreierweise) angegeben werden, so dass gewisse Kombinationen von Klassifikationen intensiver bzw. vollständig getestet werden können. Dies ist hilfreich, wenn gewisse Kombinationen als risikoanfälliger eingeschätzt werden oder deren Vorkommen höher frequentiert ist [4, S.58].

Anhand eines Beispiels soll der Einsatz des CTE/XL demonstriert werden, der kostenlos zur Verfügung steht unter [5, Link CTE/XL]. Das Beispiel basiert auf der Konfigurationstabelle 5 in Anlehnung an Bath [4, S.56]. Daraus ergibt sich der in Abb. 7 dargestellte Klassifikationsbaum. Die erste Zeile der Tabelle bezeichnet die einzelnen Klassifikationen mit den spaltenweise darunterliegenden Klassen.

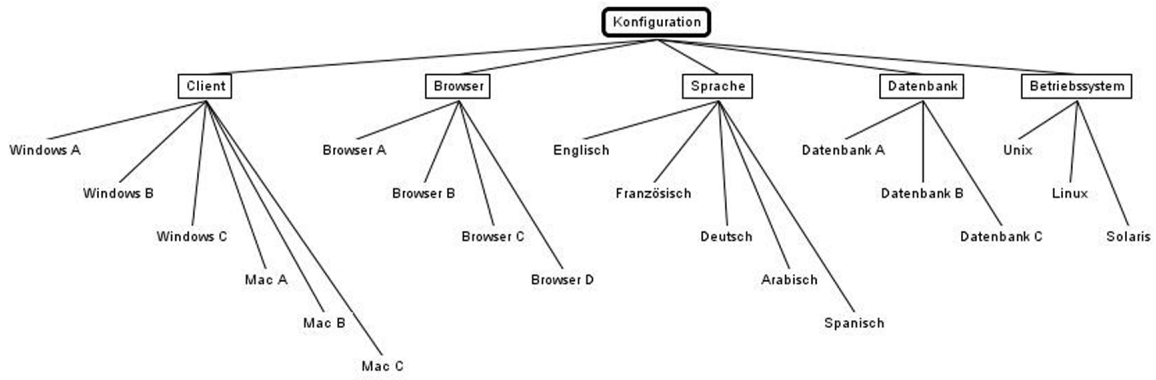

Abb. 7: *Klassifikationsbaum aus Konfigurationstabelle 5*

Mit der Kombinationsregel „*twowise*“ wird eine paarweise Testkombination gefordert. Die dafür zu erstellende Regel wird in Abb. 8 dargestellt. Die paarweise Kombination dieses Beispiels beinhaltet folgende Kombinationen:

- $Client \cdot Browser$
- $Client \cdot Sprache$

- $Client \cdot Datenbank$
- $Client \cdot Betriebssystem$
- $Browser \cdot Sprache$
- $Browser \cdot Datenbank$
- $Browser \cdot Betriebssystem$
- $Sprache \cdot Datenbank$
- $Sprache \cdot Betriebssystem$
- $Datenbank \cdot Betriebssystem$

Bei der Testfallgenerierung werden das Effizienzprinzip und Minimalforderung umgesetzt. Die verschiedenen Kombinationen werden so variiert, dass alle paarweisen Kombinationen abgedeckt und keine Testfälle mit gleicher Kombination vorhanden sind. Die automatische Generierung ergibt 30 Testfälle (vgl. Abb. 9), was dem Produkt der quantitativ stärksten Parameter (*Client* und *Sprache*) entspricht. Eine vollständige Kombination aller Klassen würde 1080 Testfälle ergeben.

Die gelben Markierungen vor den einzelnen Testfällen weisen darauf hin, dass sie nicht auf logische Abhängigkeiten, sofern vorhanden, überprüft wurden.

Dies ist eine weitere Funktionalität des CTE/XL, neben den Kombinationsregeln auch Abhängigkeitsregeln zu setzen und mittels des *Dependencymanager* prüfen zu lassen. Dies hilft, unlogische Testfälle vorab auszuschließen. Laut Wegener [21, S.5] konnten in einigen Fällen bis zu 90% aller theoretischen Testfallkombinationen hierdurch eingespart werden. Zur Demonstration soll das Beispiel um folgende Annahme erweitert werden: Der *Browser A* sei nur für *Mac A*, *Mac B* und *Mac C* konzipiert und funktioniere nicht auf Windows Betriebssystemen. Alle Testfälle mit einer Kombination aus *Windows A*, *Windows B* und *Windows C* mit *Browser A* wären somit unnötig. In solchen Fällen können logische Abhängigkeiten formuliert werden. Abb. 10 zeigt die Abhängigkeitsregel für die beschriebene Annahme. Durch Hinzufügen und Aktivieren dieser Abhängigkeitsregel werden die gelben Markierungen nun durch rote und grüne Markierungen ersetzt. Rote Markierungen verweisen auf Testfälle, die eine Verletzung der logischen Abhängigkeit darstellen (vgl. Abb. 11).

Dem Anwachsen unübersichtlicher Bäume begegnet der CTE/XL durch die Möglichkeit der Verfeinerung (engl. *Refinements*). Teilbäume können hierdurch ausgeblendet und durch Knoten ersetzt werden [31, S.51]. Dies kompensiert einen Nachteil der ursprünglichen Klassifikationsbaummethode.

Des Weiteren bietet der CTE/XL die Möglichkeit, generierte Testfälle in verschiedene Formate zu exportieren. Dieser Aspekt wird in Kap. 9 nochmal aufgegriffen.

Der CTE/XL ist ein leistungsstarkes Werkzeug für die Testfallermittlung unter Verwendung der Klassifikationsbaummethode. Seine Stärken sind, neben seiner visuellen Darstellungsmöglichkeit der Bäume und Testfälle, die automatische Generierung der Testfälle unter Berücksichtigung von Kombinations- und Abhängigkeitsregeln. Die Bewertung der Klassifikationsbaummethode unter Verwendung des CTE/XL erfolgt in Tabelle 6.

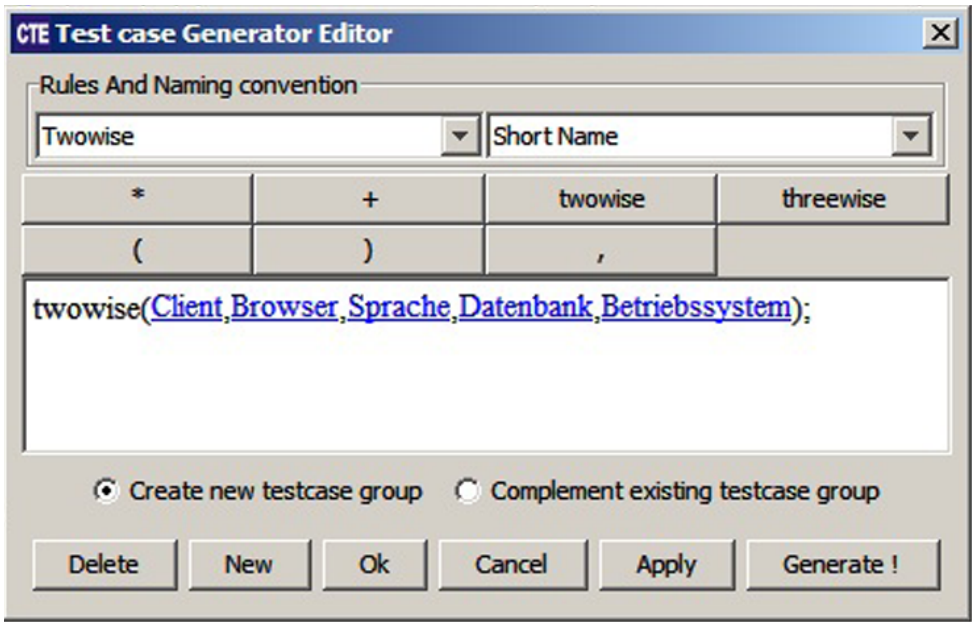

Abb. 8: *Regel für paarweises Testen*

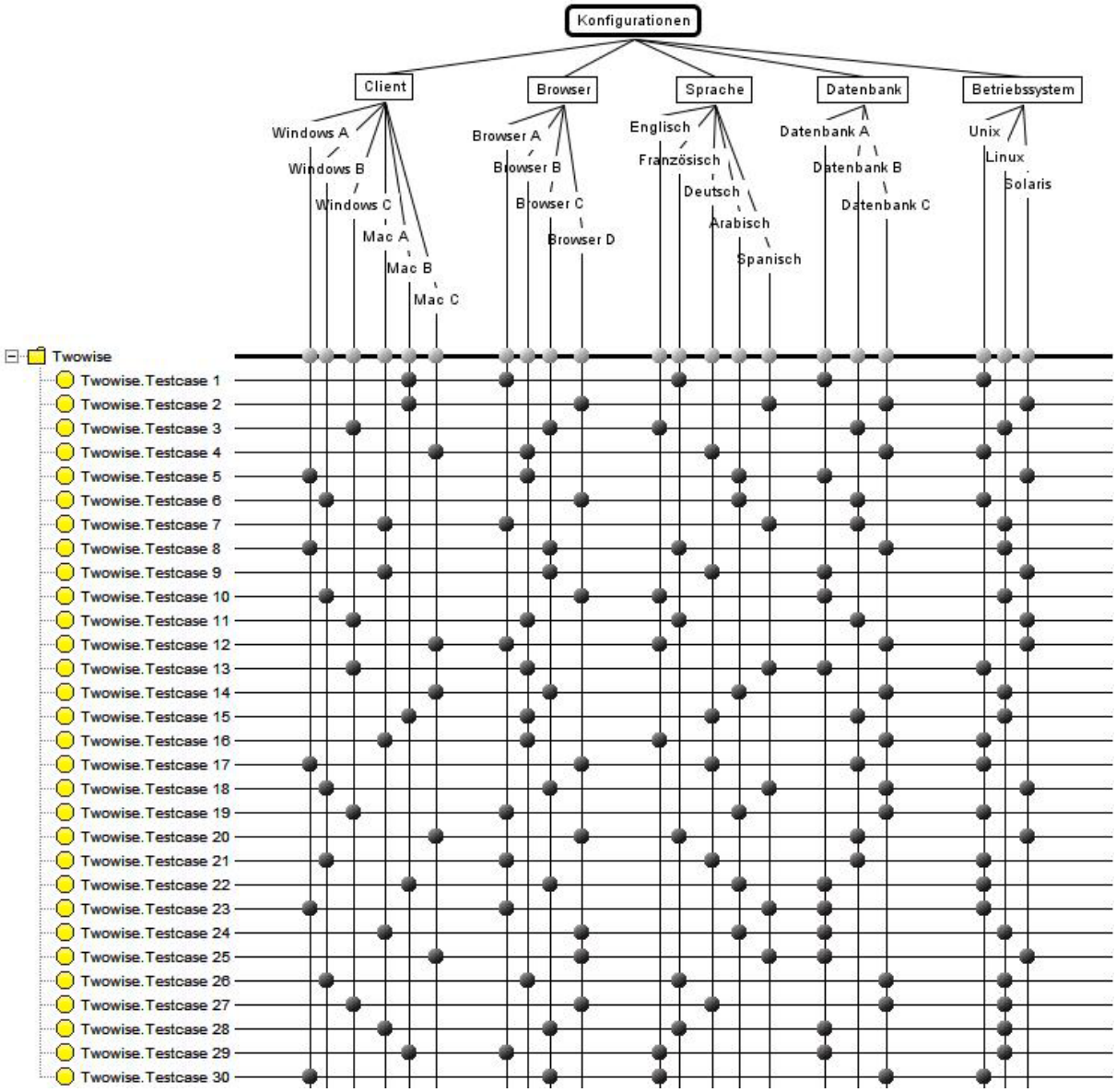

Abb. 9: *Generierte Testfälle für twowise*

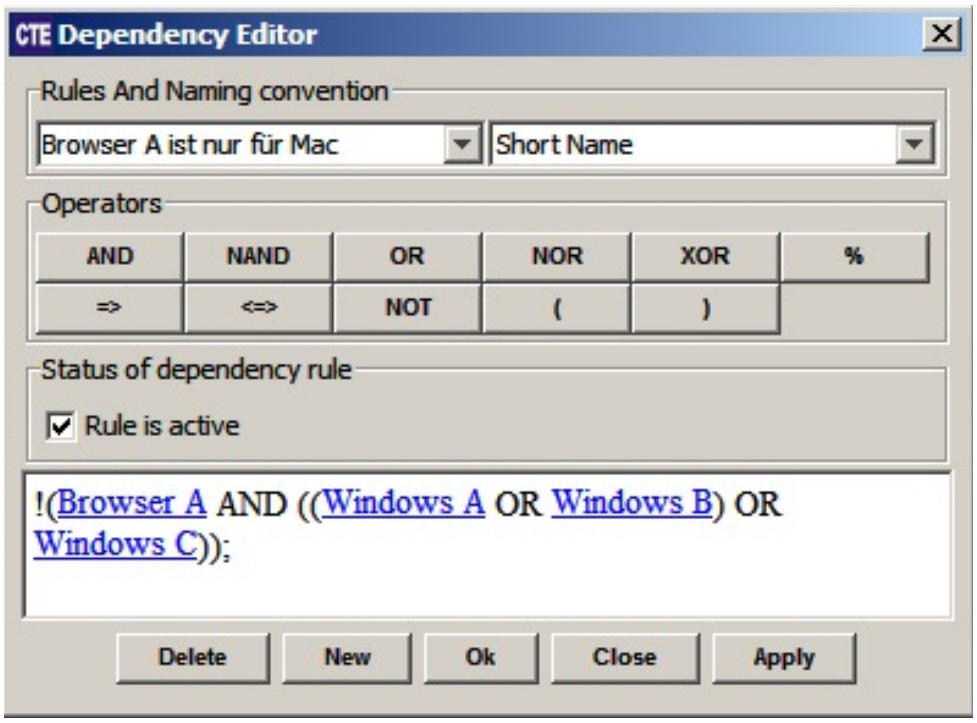

Abb. 10: *Abhängigkeitsregel: Browser A nur mit Mac*

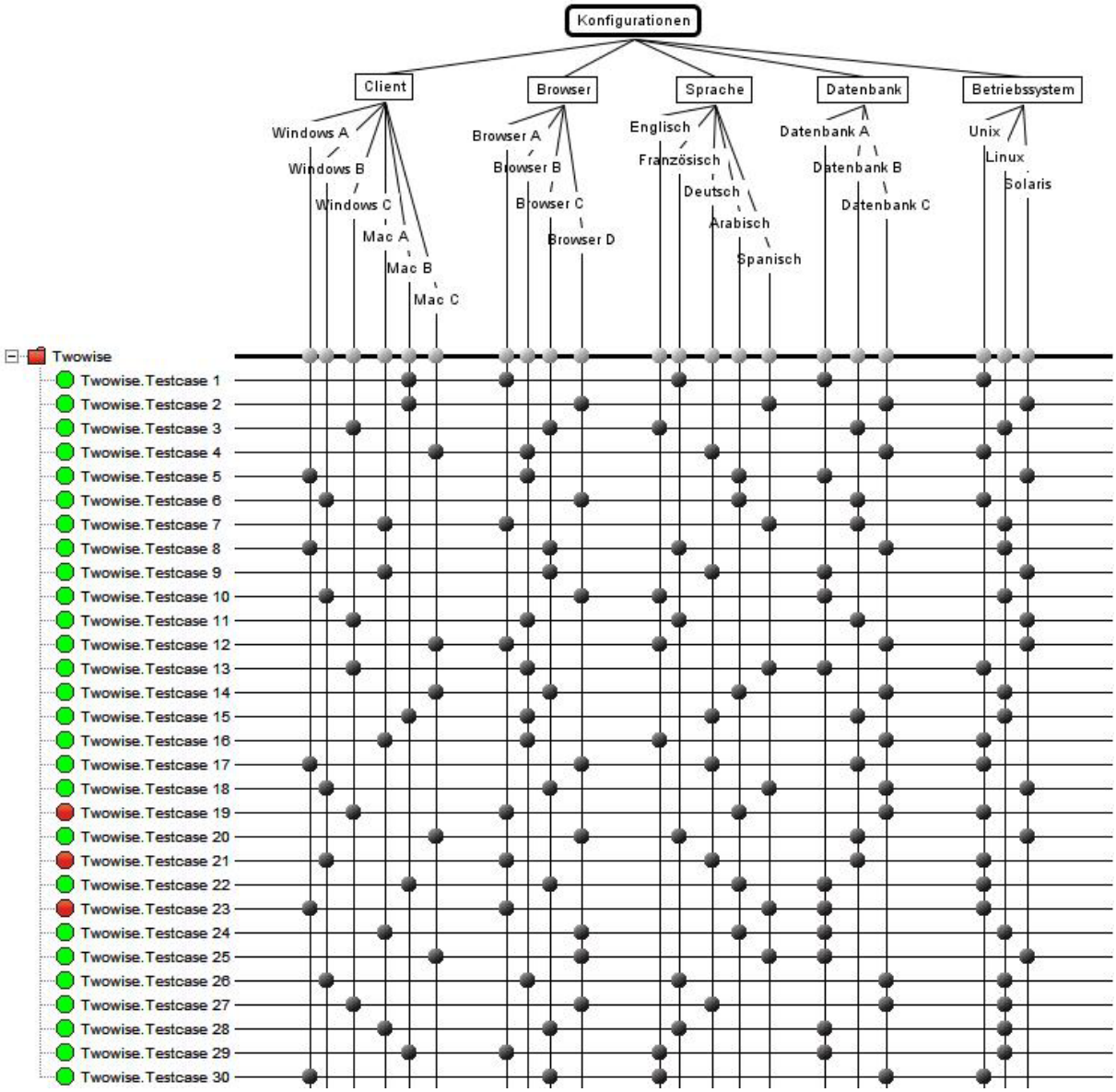

Abb. 11: *Generierte Testfälle für twowise mit Abhängigkeitsregel*

Kriterium	**Priorität**	**Bewertung**	**Kommentar**
Aufwand und Nutzen	3	$+ / -$	erhöhter Aufwand für die Erstellung des Baums bzw. der Bäume bei hohem Nutzen
Einsatz von Werkzeugen	2	$+$	sehr gute Unterstützung durch CTE/XL
Negativtests	1	$+$	mittels Äquivalenzklassenbildung und/oder Grenzwertanalyse möglich
Berücksichtigung von Prioritäten	2	$+$	kann sowohl bei der Testfallermittlung als auch bei der Testfallgenerierung berücksichtigt werden
Eindeutigkeit der Heuristik	3	$-$	Verfahrensbeschreibung sehr abstrakt
Ergebnis	2.6		

Tabelle 6: *Bewertung der Klassifikationsbaummethode mittels CTE/XL*

3.5 Evaluierung Orthogonale Arrays

Orthogonale Arrays (engl. *Orthogonal Array Testing Strategy; OATS*) greifen die mathematischen Modelle von Genichi Taguchi auf. Im Bereich des Softwaretestens dient das Verfahren der Ermittlung und Reduzierung von Testfällen auf Grundlage des paarweisen Testens. Wie das paarweise Testen geht OATS davon aus, dass Fehler überwiegend aus der paarweisen Kombination von voneinander unabhängigen Parametern resultieren.

Die Tabelle 7 zeigt ein orthogonales Array:

0	0	0
0	1	1
1	0	1
1	1	0

Tabelle 7: *Orthogonales Array*

Jedes orthogonale Array besteht aus n Zeilen und k Spalten. Die maximale Anzahl an Werten, die ein Parameter annehmen kann, wird in q ausgedrückt. Die „Stärke“ t eines orthogonalen Arrays drückt sich in der Anzahl der Spalten aus, die benötigt werden, um alle denkbaren Kombinationen gleich oft darzustellen. Das orthogonale Array in Tabelle 7 weist somit eine Stärke von zwei auf. Exemplarisch zeigen die Spalten eins und drei in Tabelle 8 (grüne Darstellung), dass alle denkbaren Kombinationen gleich oft vorkommen. Da dies für alle Spaltenpaarungen gilt, beträgt die Stärke zwei.

0	0	0
0	1	1
1	0	1
1	1	0

Tabelle 8: *Orthogonales Array der Stärke zwei*

Orthogonale Arrays werden mit folgender Notation beschrieben: OA(n, k, q, t). Das in Tabelle 8 gezeigte orthogonale Array wird entsprechend dieser Notation als OA$(4, 3, 2, 2)$ bezeichnet. Eine andere Form der Notation für dieses orthogonale Array ist nach Harrell [19, S.1]: $L_{Runs}(Levels^{Factors})$. *Runs* ist hierbei die Anzahl der Zeilen bzw. die Anzahl der Testfälle. *Levels* ist die Anzahl an annehmbaren Werten des Parameters, der die meisten Werte annehmen kann und *Factors* ist die Anzahl der Parameter. Der Faktor *Strength* wird bei dieser Notation vernachlässigt, da bei OATS die Stärke der orthogonalen Arrays immer zwei ist.

Anhand von zwei Beispielen soll die Vorgehensweise erläutert werden. In Anlehnung an Harrel [19, S.3] soll das erste Beispiel die Reduzierung der Testfälle durch Anwendung dieser Methode erläutern.

Für das Testen einer Internetseite, die in drei Sektionen eingeteilt ist (*oben*, *Mitte*, *unten*), soll deren Interaktion getestet werden. Bezüglich der einzelnen Sektionen wird unterschieden, ob sie sichtbar (1) sind oder nicht (0). Eine vollständige Kombination aller Parameter würde acht verschiedene Testfälle ergeben (s. Tabelle 9).

Wird das orthogonale Array aus Tabelle 7 darauf angewendet, reduziert sich die Anzahl der Testfälle um 50% auf nur noch vier (s. Tabelle 10).

Testfall	oben	Mitte	unten
1	0	0	0
2	0	0	1
3	0	1	0
4	0	1	1
5	1	0	0
6	1	0	1
7	1	1	0
8	1	1	1

Tabelle 9: *Vollständige Kombination aller Parameter*

Testfall	oben	Mitte	unten
1	0	0	0
2	0	1	1
3	1	0	1
4	1	1	0

Tabelle 10: *Reduzierte Anzahl an Testfällen*

Für eine verbesserte Übersicht der Testfälle werden anschließend die Eigenschaften auf die einzelnen Werte übertragen (engl. *mapping*) (s. Tabelle 11).

Testfall	oben	Mitte	unten
1	nicht sichtbar	nicht sichtbar	nicht sichtbar
2	nicht sichtbar	sichtbar	sichtbar
3	sichtbar	nicht sichtbar	sichtbar
4	sichtbar	sichtbar	nicht sichtbar

Tabelle 11: *Tabelle mit ersetzten Platzhaltern*

Die Tabellen 10 und 11 zeigen, dass die Kombination „1 1 1“ bzw. „sichtbar sichtbar sichtbar“ nicht abgedeckt wird. Die Kombinationen „1 1 0“ und „0 1 1“ sowie „1 0 1“ beinhalten bereits alle möglichen paarweisen Kombinationen mit „1“. Problematisch hierbei ist, dass das Auslassen von Testfällen auf den Kombinationsregeln der orthogonalen Arrays beruht bzw. der Annahme, dass Fehler meist aus der paarweisen Kombination von Parametern resultieren. Wäre der Testfall „1 1 1“ als fehlerträchtig oder risikoreich einzuschätzen, fände dies keine Beachtung und ein Fehler in dieser Kombination würde nicht aufgedeckt werden. Auch die Anwendung auf Basis einer Äquivalenzklassenbildung stößt hier auf Probleme. Wäre eine „1“ in Tabelle 10 ein Repräsentant einer gültigen Äquivalenzklasse (*gÄK*) und analog hierzu eine „0“ ein Repräsentant einer ungültigen Äquivalenzklasse (*uÄK*), dann würde *Testfall 1* (s. Tabelle 12) eine Kombination ungültiger Äquivalenzklassen darstellen, was aufgrund einer möglichen Fehlermaskierung zu vermeiden ist.

Testfall	oben	Mitte	unten
1	uÄK	uÄK	uÄK

Tabelle 12: *Kombination ungültiger Äquivalenzklassen*

Die Reduzierung der Testfälle ist beträchtlich und steigt prozentual mit der Anzahl an Parame-

tern und deren annehmbarer Werte, wie das zweite Beispiel verdeutlichen soll. Ausgangspunkt ist eine eingeschränkte Variante des Beispiels aus Tabelle 5:

Clients	Browser	Sprachen	Datenbanken	Server
Windows A	Browser A	Englisch	Datenbank A	Unix
Windows B	Browser B	Französisch	Datenbank B	Linux
Windows C	Browser C	Deutsch	Datenbank C	Solaris
Mac A		Arabisch		
Mac B				

Tabelle 13: *Konfigurationstabelle in Anlehnung an Bath [4, S.56]*

Ein erschöpfender Test benötigt hierfür 540 Testfälle:

$$\text{Clients} \cdot \text{Browser} \cdot \text{Sprachen} \cdot \text{Datenbanken} \cdot \text{Server} = \text{Anzahl Testfälle} \Rightarrow 5 \cdot 3 \cdot 4 \cdot 3 \cdot 3 = 540$$

Das orthogonale Array wird beschrieben durch OA(20,5,5,2). Die 20 ist das Produkt aus der Multiplikation der zwei quantitativ stärksten Parameter:

$$\text{Anzahl Clients} \cdot \text{Anzahl Sprachen} = 20$$

Die erste 5 repräsentiert die Anzahl der Parameter (*Clients*, *Browser*, *Sprachen*, *Datenbanken*, *Server*). Die zweite 5 stellt die maximale Anzahl verschiedener Werte dar (*Windows A*, *Windows B*, *Windows C*, *Mac A*, *Mac B*). Die Stärke 2 bedeutet die vollständige Abdeckung aller Paare. Die alternative Beschreibung hierfür wäre $L_?(5^1 4^1 3^3)$, ein Parameter mit fünf verschiedenen Werten (*Clients*), ein Parameter mit vier verschiedenen Werten (*Sprachen*) und drei Parameter mit drei verschieden Werten (*Browser*, *Datenbanken*, *Server*). Das „?“ ist abhängig vom zu wählenden orthogonalen Array und dessen Anzahl an $Runs$.

Es wird nun eine Tabelle eines orthogonalen Arrays benötigt mit mindestens der benötigten Beschreibung (Tabellen sind u. a. zu finden unter: `http://www.research.att.com/~njas/oadir` oder `http://support.sas.com/techsup/technote/ts723.html`). Wird keine passende Tabelle gefunden, so könnte auch z. B. ein OA(25,6,5,2) bzw. ein $L_{25}(5^6)$ der Stärke zwei benutzt werden (s. Tabelle 14). Bei einer Auswahl nach der ersten Notation ist darauf zu achten, dass kein Wert (n, k, q, t) kleiner als benötigt ist. Für eine Auswahl nach der zweiten Notation muss das orthogonale Array mindestens fünf *Levels* (quantitativ stärkster Parameter) aufweisen und mindestens fünf *Factors* (Anzahl Parameter) besitzen.

Da nur fünf Variablen benutzt werden, kann die letzte Spalte (*VAR 6*) in Tabelle 14 gelöscht werden. Die Platzhalter sind gegen die zugehörigen Werte zu ersetzen. Nicht gebrauchte Platzhalter werden durch ein „-“ ersetzt. Die Tabelle 15 verdeutlicht diesen Schritt.

Anschließend werden redundante Testfälle entfernt bzw. mit anderen Testfällen vereint (in Tabelle 16 grün dargestellt):

- Testfall 5 wird mit Testfall 3 vereint
- Testfall 9 wird mit Testfall 8 vereint
- Testfall 14 wird mit Testfall 12 vereint
- Testfall 19 wird mit Testfall 16 vereint
- Testfall 25 wird mit Testfall 21 vereint

Testfall	Var 1	Var 2	Var 3	Var 4	Var 5	Var 6
1	0	0	0	0	0	0
2	0	1	1	2	3	4
3	0	2	2	3	4	1
4	0	3	3	4	1	2
5	0	4	4	1	2	3
6	1	0	1	1	1	1
7	1	1	2	4	0	3
8	1	2	4	0	3	2
9	1	3	0	3	2	4
10	1	4	3	2	4	0
11	2	0	2	2	2	2
12	2	1	4	3	1	0
13	2	2	3	1	0	4
14	2	3	1	0	4	3
15	2	4	0	4	3	1
16	3	0	3	3	3	3
17	3	1	0	1	4	2
18	3	2	1	4	2	0
19	3	3	4	2	0	1
20	3	4	2	0	1	4
21	4	0	4	4	4	4
22	4	1	3	0	2	1
23	4	2	0	2	1	3
24	4	3	2	1	3	0
25	4	4	1	3	0	2

Tabelle 14: *Tabelle eines OA(25,6,5,2)*

Alle verbleibenden Platzhalter können nun mit Konfigurationen besetzt werden (in Tabelle 16 rot dargestellt), von denen anzunehmen ist, dass sie häufiger auftreten als andere oder deren Risiko oder Fehleranfälligkeit höher einzuschätzen ist. Dafür wird die Annahme getroffen, dass alle Variablen der ersten Zeile aus Tabelle 13 Priorität 1 haben (*Windows A*, *Browser A*, *Englisch*, *Datenbank A*, *Unix*).

Testfall	Clients	Browser	Sprachen	Datenbanken	Server
1	Windows A	Browser A	Englisch	Datenbank A	Unix
2	Windows A	Browser B	Französisch	Datenbank C	-
3	Windows A	Browser C	Deutsch	-	-
4	Windows A	-	Arabisch	-	Linux
5	Windows A	-	-	Datenbank B	Solaris
6	Windows B	Browser A	Französisch	Datenbank B	Linux
7	Windows B	Browser B	Deutsch	-	Unix
8	Windows B	Browser C	-	Datenbank A	-
9	Windows B	-	Englisch	-	Solaris
10	Windows B	-	Arabisch	Datenbank C	-
11	Windows C	Browser A	Deutsch	Datenbank C	Solaris
12	Windows C	Browser B	-	-	Linux
13	Windows C	Browser C	Arabisch	Datenbank B	Unix
14	Windows C	-	Französisch	Datenbank A	-
15	Windows C	-	Englisch	-	-
16	Mac A	Browser A	Arabisch	-	-
17	Mac A	Browser B	Englisch	Datenbank B	-
18	Mac A	Browser C	Französisch	-	Solaris
19	Mac A	-	-	Datenbank C	Unix
20	Mac A	-	Deutsch	Datenbank A	Linux
21	Mac B	Browser A	-	-	-
22	Mac B	Browser B	Arabisch	Datenbank A	Solaris
23	Mac B	Browser C	Englisch	Datenbank C	Linux
24	Mac B	-	Deutsch	Datenbank B	-
25	Mac B	-	Französisch	-	Unix

Tabelle 15: *Tabelle des OA(20,5,5,2) mit ersetzten Platzhaltern aus Tabelle 13*

Testfall	Clients	Browser	Sprachen	Datenbanken	Server
1	Windows A	Browser A	Englisch	Datenbank A	Unix
2	Windows A	Browser B	Französisch	Datenbank C	Unix
3 und 5	Windows A	Browser C	Deutsch	Datenbank B	Solaris
4	Windows A	Browser A	Arabisch	Datenbank A	Linux
6	Windows B	Browser A	Französisch	Datenbank B	Linux
7	Windows B	Browser B	Deutsch	Datenbank A	Unix
8 und 9	Windows B	Browser C	Englisch	Datenbank A	Solaris
10	Windows B	Browser A	Arabisch	Datenbank C	Unix
11	Windows C	Browser A	Deutsch	Datenbank C	Solaris
12 und 14	Windows C	Browser B	Französisch	Datenbank A	Linux
13	Windows C	Browser C	Arabisch	Datenbank B	Unix
15	Windows C	Browser A	Englisch	Datenbank A	Unix
16 und 19	Mac A	Browser A	Arabisch	Datenbank C	Unix
17	Mac A	Browser B	Englisch	Datenbank B	Unix
18	Mac A	Browser C	Französisch	Datenbank A	Solaris
20	Mac A	Browser A	Deutsch	Datenbank A	Linux
21 und 25	Mac B	Browser A	Französisch	-	Unix
22	Mac B	Browser B	Arabisch	Datenbank A	Solaris
23	Mac B	Browser C	Englisch	Datenbank C	Linux
24	Mac B	Browser A	Deutsch	Datenbank B	Unix

Tabelle 16: *Tabelle des OA(20,5,5,2) mit ersetzten Platzhaltern und Korrektur redundanter Testfälle*

Im Vergleich zum erschöpfenden Test mit 540 Testfällen sind es nach der vollständigen Transformation der Konfigurationstabelle in das orthogonale Array nur noch 20 Testfälle, also ca. 4% der vollständigen Kombination (vgl. Tabelle 16). Bath [4, S.55] bezeichnet das Verfahren der orthogonalen Arrays und das paarweise Testen als die einzigen effektiven Methoden, „[...]um mit dem explosionsartigen Wachstum an Kombinationen umzugehen, das durch mehrfache Konfigurationsmöglichkeiten verursacht wird.“. Der einzige Unterschied zwischen orthogonalen Arrays und dem paarweisen Testen liegt in der Bildung der Paare. Im Ergebnis gibt es keine erheblichen Unterschiede zwischen beiden Verfahren.

Die Minimalforderung, jeder Parameter wird in mindestens einem Testfall berücksichtigt und das Effizienzprinzip, die beachtliche Reduzierung der Testfälle, werden erfüllt. Negativtests können durch dieses Verfahren in Verbindung mit der Äquivalenzklassenbildung nur unter Vorbehalt der o. g. Problematik bezüglich einer Fehlermaskierung vollzogen werden. Die Aussagekraft bzw. Abdeckung ist hoch und ein Einsatz im Systemtest möglich.

Zu bemängeln ist jedoch, dass die Reduzierung der Testfälle auf rein mathematisch statistischen Verfahren beruht, repräsentiert durch die benötigten Tabellen. Spezifische Aspekte des zu testenden Systems werden nur bei der Ersetzung verbleibender Platzhalter (rote Darstellung in Tabelle 16) beachtet. Dies bedeutet, dass bei einer Reduzierung auf 4% der Testfälle nur 1,6% (neun Testfälle) übrig bleiben, die Aspekte, wie eine erhöhte Fehleranfälligkeit, besonderes Risiko oder Priorität berücksichtigen. Damit sind die Freiheitsgrade in der Wahl der konkreten Testdaten sehr eingeschränkt. Durch Auswahl geeigneter Repräsentanten aus einer vorangegangenen Äquivalenzklassenbildung kann die Beachtung dieser Faktoren aber auch schon früher einfließen. Eine vorherige Äquivalenzklassenbildung ist in vielen Fällen zur Reduzierung von q bzw. $Levels$ zwingend nötig. Die o. g. Beispiele weisen hierfür zwei $Levels$ (*sichtbar* und *nicht sichtbar*) und fünf $Levels$ (*Windows A, Windows B, Windows C, Mac A, Mac B*) auf. Ein Parameter der z. B. sämtliche Werte einer 32-Bit Integer Variable annehmen kann, muss durch Repräsentanten ersetzt werden, da es kein entsprechend benötigtes orthogonales Array geben wird.

Die Tatsache, dass das Verfahren auf der paarweisen Kombination unabhängiger Variablen beruht, ist eine weitere Einschränkung. „Es kann sein, dass es zu einer unerwarteten Interaktion zwischen einzelnen Komponenten kommt[...]“ [4, S.56].

Logische Abhängigkeiten, wie im Beispiel aus Kap. 3.4.1, dass *Browser A* nur für *Mac* konzipiert sei, werden nicht unterstützt. Der Aufwand ist aufgrund der Komplexität ohne Werkzeug als hoch einzuschätzen. Benötigte Tabellen müssen gefunden und ausgefüllt werden. Sofern eine Recherche im Rahmen dieser Arbeit möglich war, konnte ein kostenloses Werkzeug hierfür gefunden werden. Das Werkzeug *MktEx* (unter: `http://support.sas.com/techsup/technote/mr2010.zip`) enthält einen Katalog von orthogonalen Arrays. Bei einer Anfrage nach einem benötigten orthogonalen Array benutzt es den Katalog und einen Suchalgorithmus, um ein bestmögliches orthogonales Array zu finden.

Für die Aufwandsbetrachtung wird die selbständige Erstellung der benötigten Tabellen nicht mit einbezogen. Dies erfordert entsprechende mathematische Kenntnisse, auf die hier nicht näher eingegangen werden soll. Eine selbständige Erstellung der benötigten Tabellen würde den Aufwand aber nochmals erhöhen.

Im direkten Vergleich zur Klassifikationsbaummethode mit dem CTE/XL Editor schneidet OATS schwach ab. Der CTE/XL bietet über sein leistungsstarkes Regelwerk über paarweises Testen hinaus noch intensivere Paarungen an, die sogar nur für bestimmte risikoreiche Teilkombinationen angewandt werden können. Die Reduzierung von Testfällen im Falle einer Testfallexplosion erfolgt bei beiden Verfahren gleich gut. Der Aufwand ist jedoch erheblich geringer

beim CTE/XL als bei OATS, auch unter Verwendung von MktEx. Somit ergibt sich folgende Bewertung:

Kriterium	Priorität	Bewertung	Kommentar
Aufwand und Nutzen	3	+ / −	hoher Aufwand für Ermittlung geeigneter Tabelle und Übertragung von Werten selbst bei Nutzung des Werkzeugs
Einsatz von Werkzeugen	2	+ / −	Werkzeugunterstützung nur für die Ermittlung der Tabellen
Negativtests	1	+ / −	durch Äquivalenzklassenbildung und/oder Grenzwertanalyse möglich; Fehlermaskierung wird nicht berücksichtigt
Berücksichtigung von Prioritäten	2	−	nur bedingt möglich, da Generierung der Testfallkombinationen nicht beeinflussbar ist
Eindeutigkeit der Heuristik	3	+ / −	eindeutiges, aber umständliches Verfahren
Ergebnis	1.8		

Tabelle 17: *Bewertung der OATS*

3.6 Fazit Evaluierung

Alle vorgestellten Verfahren haben ihre Berechtigung für den Einsatz im Systemtest. Die Auswahl der Methoden sollte sich dabei nicht nur auf eine beschränken. Um das gesamte Potential jedes einzelnen Verfahrens voll auszuschöpfen, ist nur eine Kombination der Verfahren in Betracht zu ziehen. Je zahlreicher die Verknüpfungen einzelner Verfahren sind, desto höher sind Effektivität und Aussagekraft einzuschätzen. Folgende Konstellationen in jeweils angegebener Reihenfolge sind abzuwägen:

1. Äquivalenzklassenbildung und Grenzwertanalyse
2. Klassifikationsbaummethode (mittels CTE/XL), Äquivalenzklassenbildung und Grenzwertanalyse
3. Klassifikationsbaummethode (mittels CTE), Äquivalenzklassenbildung, Grenzwertanalyse und orthogonale Arrays

Die erste Konstellation hilft, unbrauchbare Testfälle auszuschließen und die Anzahl dadurch zu reduzieren. Die Reduzierung bezieht sich aber lediglich auf die Kombination einzelner Vertreter aus den verschiedenen Äquivalenzklassen. Eine große Anzahl von Äquivalenzklassen kann dennoch Grund für zahlreiche Testfälle sein. Faktoren wie Risiko, Fehleranfälligkeit oder Häufigkeit können problemlos einbezogen werden. Vor allem die Anwendung auf Ausgabeparameter ist eine Stärke dieser Konstellation. Bei begrenzter Anzahl von Äquivalenzklassen weist sie den geringsten Aufwand unter den genannten Alternativen auf und stellt vor allem bei Zeitmangel die beste Alternative dar.

Die dritte Konstellation aus Klassifikationsbaummethode (mittels CTE/XL), Äquivalenzklassenbildung, Grenzwertanalyse und orthogonale Arrays stellt die längste Verknüpfung dar. Obwohl sich alle vier Verfahren kombinieren lassen, führen sie teilweise zu gegenseitigen Ein-

schränkungen, da sie keine vollkommene Kompatibilität zueinander aufweisen. Die Unterstützung durch den CTE/XL wird auf seine Basisfunktionalität beschränkt. Eine automatisierte Generierung der Testfälle, wie es der CTE/XL bietet, steht im Widerspruch zur Methodik der orthogonalen Arrays. Lediglich die Vorzüge der graphischen Darstellung des Klassifikationsbaums zur Ermittlung von testrelevanten Aspekten können genutzt werden. Im Vergleich zur ersten Konstellation wird bei der Äquivalenzklassenbildung in Kombination mit orthogonalen Arrays die Menge an Testfällen noch deutlicher reduziert. Die Äquivalenzklassenbildung schränkt die Kombinationsmöglichkeiten annehmbarer Werte der Parameter ein und beeinflusst dadurch q bei OA(n, k, q, t) bzw. *Levels* bei $L_{Runs}(Levels^{Factors})$. Im Unterschied zur ersten Konstellation kann eine Testfallexplosion auch durch zahlreiche Äquivalenzklassen verhindert werden.

Aufgrund der in Kap. 3.5 bemängelten Eigenschaften der orthogonalen Arrays sowie der eingeschränkten Möglichkeit der Verknüpfung von CTE/XL und orthogonalen Arrays sollte diese Konstellation gemieden werden. Alle Eigenschaften von OATS werden auch bzw. noch besser und gleichzeitig komfortabler durch den CTE/XL ermöglicht. Daher ist von der Verwendung der orthogonalen Arrays im Systemtest abzuraten und stattdessen die Klassifikationsbaummethode mittels CTE/XL vorzuziehen.

Die Verbindung aus Klassifikationsbaummethode (mittels CTE/XL), Äquivalenzklassenbildung und Grenzwertanalyse stellt die effektivste Konstellation dar. Die Verfahren bauen aufeinander auf und sind absolut kompatibel zueinander. Testfallkombinationen können automatisiert generiert werden. Um hochpriore Funktionalitäten oder Bereiche mit erhöhtem Risiko intensiver zu testen, kann eine *triplewise* Kombinationsregel oder, wenn nötig, eine vollständige Kombination ausgeführt werden. Die Äquivalenzklassenbildung unterstützt eine systematische Reduzierung der Testfälle, durch Auswahl geeigneter Repräsentanten. Auch Negativtests oder Tests in Bezug auf Ausgabebewerte sind problemlos möglich.

Da der CTE/XL noch intensivere Kombinationsregeln als die paarweise Kombination erlaubt, erreicht er eine höhere Abdeckung als die auf das paarweise Testen beschränkte Technik bei OATS. Unlogische Testfälle oder die Verwendung von Repräsentanten ungültiger Äquivalenzklassen in einem Testfall können mithilfe der Abhängigkeitsregeln und des *dependencymanager*, im Unterschied zu OATS, verhindert werden.

Alle Verfahren in dieser Konstellation können ihr Potential ohne Einschränkungen optimal entfalten. Für den Systemtest empfiehlt sich diese Variante daher als beste Wahl. Unter besonderen Umständen wie Zeitmangel oder Testfallexplosion stellen die verbleibenden Konstellationen aber gute Alternativen dar.

4 HP Quality Center

Für das Geschäftsfeld Managed Test Services nutzt Logica u. a. das HP/Mercury Quality Center. HP/Mercury Quality Center ist ein webbasiertes Testmanagement System, das im Zuge der Softwareerstellung projektbegleitend den Testprozess unterstützt und dessen Automatisierung dient. Dafür gliedert es den Testprozess in folgende Phasen (s. Abb. 12) und stellt entsprechende Tools zur Verfügung:

Abb. 12: *Phasen des Testprozesses aus Quality Center Tutorial [10, S.12]*

- Entwicklung von Zeitplänen zur Freigabe von Konfigurationsobjekten in einer bestimmten Version für einen bestimmten Zweck
- Identifizierung und Validierung von funktionalen und nicht funktionalen Anforderungen für den Testprozess
- Planung und Spezifizierung von Testfällen
- Planung der Abläufe für die Ausführung von Tests und Analyse der Ergebnisse
- Festhalten von Defekten und Überprüfung der Fehlerbehebung

Werkzeuge dieser Art und wie der in Kap. 3.4.1 bereits vorgestellte CTE-Editor werden als CAST-Tools (*Computer Aided Software Testing*) bezeichnet. Ein Unterschied des HP Quality Centers gegenüber vielen anderen Werkzeugen ist die Einbindung eines breiten Spektrums von Anwendern.

> „Einbezogen werden jene Geschäftsanalytiker, die Anforderungen für das System spezifizieren, Projekt- und Testmanager, die den Testplan verfassen und die Testszenarien entwerfen, Tester, die Testfälle ausarbeiten, Testskripte schreiben und den Test anstoßen sowie der Produktmanager, der über die Freigabe des Produktes oder deren Installation entscheidet“ [35, S.209].

Alle folgenden Beschreibungen und Ausarbeitungen bezüglich des HP/Mercury Quality Center basieren auf der Version 9.2.

4.1 Komponenten

Das HP/Mercury Quality Centers besteht aus folgenden Hauptkomponenten:

- WinRunner
- QuickTest Professional
- LoadRunner
- Business Process Tester
- Test Director

WinRunner

WinRunner ist ein sogenanntes *Capture-Replay-Werkzeug*. Diese Arten von Werkzeugen wird für den Test von Benutzeroberflächen (*Graphical User Interface*; GUI) eingesetzt. Im Capture Modus werden sämtliche Interaktionen des Benutzers erfasst. Dabei identifizieren sie Gui-Elemente und deren Zustände, wie z. B. Wert, Name, Farbe und die Koordinaten der Mausposition bei Aktionen. Die Interaktionen werden in einem automatisch generierten Skript festgehalten. Durch das Setzen von *Checkpoints* im Skript können Eigenschaften von Objekten des

Bildschirm-Layouts überprüft werden. Der Replay Modus ermöglicht das Abspielen des aufgenommenen Testskripts. Veränderungen des Bildschirm-Layouts oder nicht übereinstimmende Zustände der Elemente an den Checkpoints führen zum Scheitern des Tests. WinRunner wurde 2009 vom Markt genommen und Kunden erfahren nur noch eingeschränkten Support [44].

QuickTest Professional

WinRunner und QuickTest Professional sind sich sehr ähnlich. Während WinRunner bereits seit 1995 erhältlich ist, stellt QuickTest Professional das jüngere Produkt dar, das erst seit 2002 erhältlich ist und nun dessen Nachfolge antritt. Sämtliche Funktionalitäten, die WinRunner bietet, stehen auch bei QuickTest Professional zur Verfügung. Ein großer Unterschied zu WinRunner ist es, „nicht-professionellen Testern die Möglichkeit zu geben, Tests zu erzeugen und zu wiederholen, ohne in die Testskripte eingreifen zu müssen“ [35, S.211]. Die verwendete Skriptsprache bei WinRunner nennt sich TSL (*Test Script Language*) und basiert auf der Sprache C. QuickTest Professional unterstützt die Skriptsprache TSL, sodass vorhandene WinRunner Skripte problemlos zu integrieren sind. QuickTest Professional hingegen nutzt Microsoft VBScript als Skriptsprache.

Zudem erlaubt QuickTest Professional den Funktionstest von Web-Services. Dazu wird der SOAP-Aufruf (*Simple Object Access Protocol*) eines WSDL-Service (*Webservice Description Language*) geprüft und die Auswirkung des angesprochenen Service auf andere Web-Services oder Applikationen untersucht.

LoadRunner

LoadRunner ist eine Komponente, die für Lasttests eingesetzt werden kann. Es erlaubt die Generierung von Last z. B. durch Emulieren einer großen Anzahl gleichzeitiger Benutzer. Dabei können Informationen über die Infrastruktur (Web-Server, Datenbankserver) gesammelt und anschließend analysiert werden.

Business Process Tester

Der Business Process Tester verknüpft das Wissen der Anwender aus dem Fachbereich mit dem Wissen der Testspezialisten. Anhand geschäftlicher Definitionen können die Tester eine End-to-End Verarbeitung der Prozesse nachvollziehen. Für die Tester liegt der Vorteil darin, die wesentlichen Abläufe von Geschäftsprozessen nicht mehr identifizieren zu müssen. Für die Fachexperten bietet der Business Process Tester den Vorteil, Abläufe grafisch verfolgen und Zwischenstände registrieren zu können. Ein Abgleich mit dem geforderten Ablauf deckt Fehlverhalten des Testsystems auf.

TestDirector

Der TestDirector ist die Hauptkomponente des HP Quality Center. Er bietet den Zugriff auf alle darunterliegenden Testmanagement-Werkzeuge. Diese Werkzeuge sind in nachstehende Module, die im folgenden Kap. 4.2 vorgestellt werden sollen, eingeteilt:

- Release Modul
- Requirements Modul
- Test Plan Modul
- Test Lab Modul
- Defect Modul

4.2 Module

Ausgehend vom TestDirector stellen die folgenden Module die grundlegenden Werkzeuge des Quality Centers dar.

Release Modul

Der Testprozess im HP Quality Center beginnt mit der Festlegung von Releases im Release Modul. Ein Release definiert sich hierbei als eine Gruppe von Änderungen in einer oder mehr Applikationen, die alle zu einem bestimmten Zeitpunkt abgeschlossen sind, um ein Produkt ausliefern zu können. Ein Release besteht aus einer zu definierenden Anzahl von *Cycles*. Jedem *Cycle* werden Entwicklungs- und Testaufwände zugefügt, um ein bestimmtes Ziel innerhalb des Zeitplans zu erreichen. Zusätzlich kann ein *Cycle* mit einem Start- und Endzeitpunkt versehen werden. Startzeitpunkt des ersten *Cycles* und Endzeitpunkt des letzten *Cycles* geben dann die gesamte Zeitplanung wieder. Sämtliche Aspekte der anderen Module können den einzelnen *Cycles* zugeordnet werden, wie z. B. im Requirements Modul gesetzte Anforderungen oder *Test sets* aus dem Test Lab Modul.

Anhand grafischer Diagramme werden u. a. die Anzahl ermittelter und behobener Defekte, erfolgreich bzw. noch nicht (erfolgreich) gelaufene Tests oder die Abdeckung von Anforderungen durch Tests dargestellt. Dadurch kann der Testmanager in Echtzeit den Fortschritt einzelner *Cycles* und des gesamten Projektverlaufs analysieren. Die Abb. 13 zeigt beispielhaft eine Darstellung zur Erfassung des Testfortschritts innerhalb eines *Cycle*. Von fünf geplanten Testläufen

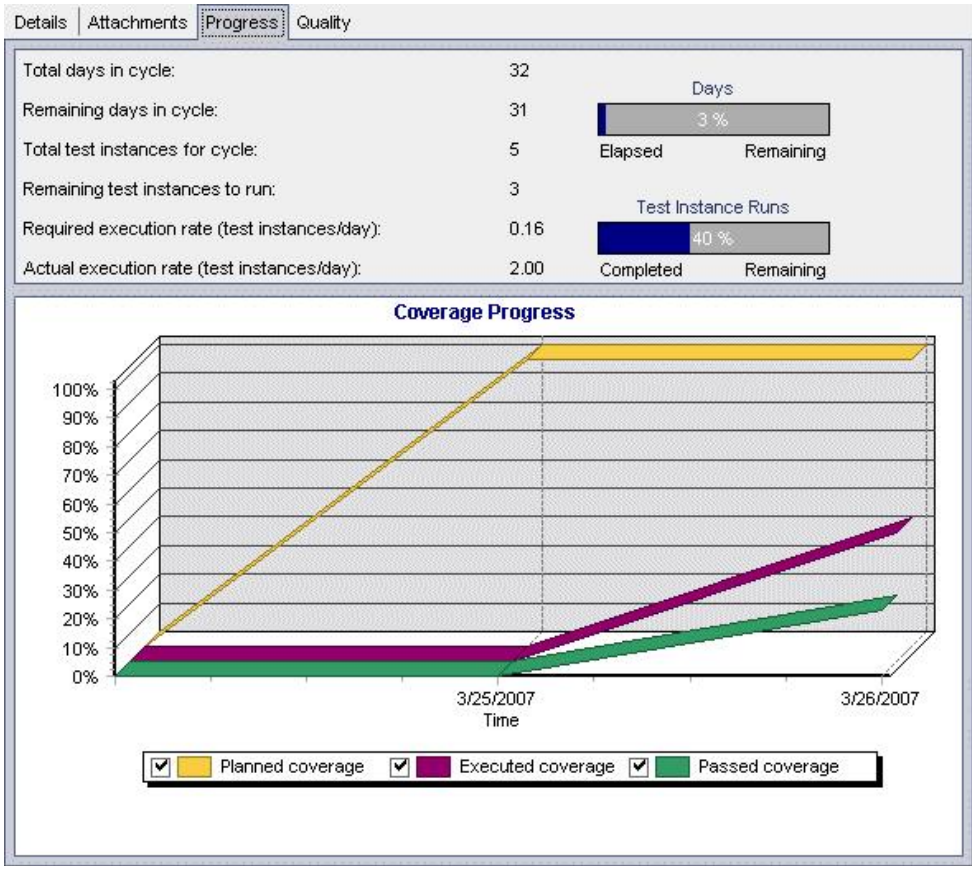

Abb. 13: *Fortschrittsdiagramm eines Cycles aus Quality Center Tutorial [10, S.100]*

sind drei noch nicht gelaufen. Alle fünf geplanten Tests innerhalb dieses *Cycle* erreichen eine Anforderungsüberdeckung von 100% (gelbe Kurve). Dies bezieht sich nur auf die dem *Cycle* zugewiesenen Anforderungen. Zwei von fünf Testläufen (40%) sind bereits durchgeführt wor-

den (lila Kurve). Einer von fünf Testläufen (20%) ist erfolgreich (*passed*) durchgelaufen (grüne Kurve).

Requirements Modul

Im Requirements Modul können Anforderungen definiert werden. Die Anforderungen werden dann in einer hierarchischen Baumstruktur oder in Tabellenform abgelegt. Eine Hauptanforderung kann dabei aus untergeordneten Anforderungen bestehen. Die wichtigsten Attribute einer Anforderung sind Priorität und die Zuordnung eines Produkts. Nachdem eine Anforderung inklusive aller optionalen untergeordneten Anforderungen definiert wurde, kann sie einem *Cycle* aus dem Release Modul zugeordnet werden.

Test Plan Modul

Das Test Plan Modul ist ein Repository von Testfällen und dient der Planung von Tests in Bezug auf Testziele und Teststrategie. Der Workflow des HP Quality Centers sieht nach der Definition aller Anforderungen im Requirements Modul die Spezifikation von Testfällen vor. Erstellte Testfälle können dann mit bestimmten Anforderungen verlinkt werden. Hierbei ist zu erwähnen, dass die systematische Ermittlung von konkreten Testfällen außerhalb von HP Quality Center stattfindet. Dies kann z. B. anhand der in Kapitel 3 evaluierten Verfahren vorgenommen werden. Tests werden attribuiert durch ein Erstellungsdatum, eine Test ID und einen Typ. Für das Attribut Typ wird z. B. unterschieden, ob der Test für die Komponenten WinRunner, QuickTest Professional oder LoadRunner erstellt werden soll oder es sich um einen manuell durchzuführenden Test handelt. Des Weiteren können eine allgemeine Beschreibung und ein Kommentar angegeben werden.

Die Tests werden in ihrer Durchführung durch Testschritte (engl. *Test Steps*) beschrieben. Testschritte werden durch Name, Beschreibung und zu erwartendem Sollwert definiert (vgl. Abb. 14).

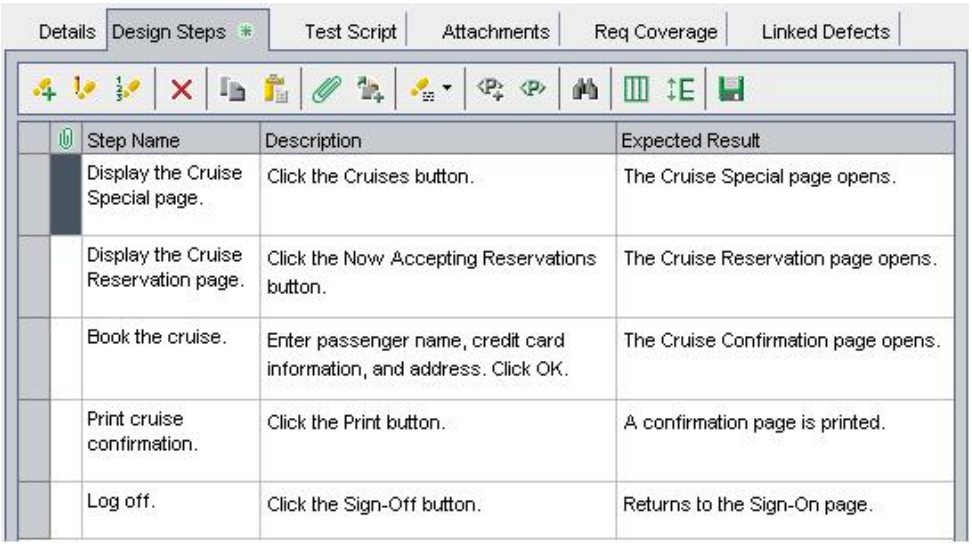

Abb. 14: *Testschritte eines Testfalls aus Quality Center Tutorial [10, S.55]*

Testfälle und deren Testschritte können mit Testschritten anderer Testfälle verknüpft werden, sodass sie als wiederverwendbare Vorbedingung genutzt werden. Ein Testfall, der z. B. das Einloggen eines Anwenders in eine Applikation abdecken soll, kann als Vorbedingung für alle weiterführenden Tests, die eine Anmeldung voraussetzen, genutzt werden. Er kann sogar als allgemeiner Testschritt für sämtliche Testobjekte, die einen Anmeldevorgang benötigen, dienen. Um einen Test für einen Anmeldevorgang zu generalisieren, können Parameter für die Anmeldedaten benutzt werden, deren Werte für jeden spezifischen Anwendungsfall geändert werden können (vgl. Abb. 15). Parameter können z. B. URL (*Uniform Ressource Locator*), Benutzername und Passwort sein. Mithilfe dieser Parameter ist es möglich verschiedene Rollen (normaler Benutzer, Administrator), unter Wiederverwendung dieses Testfalls, zu prüfen.

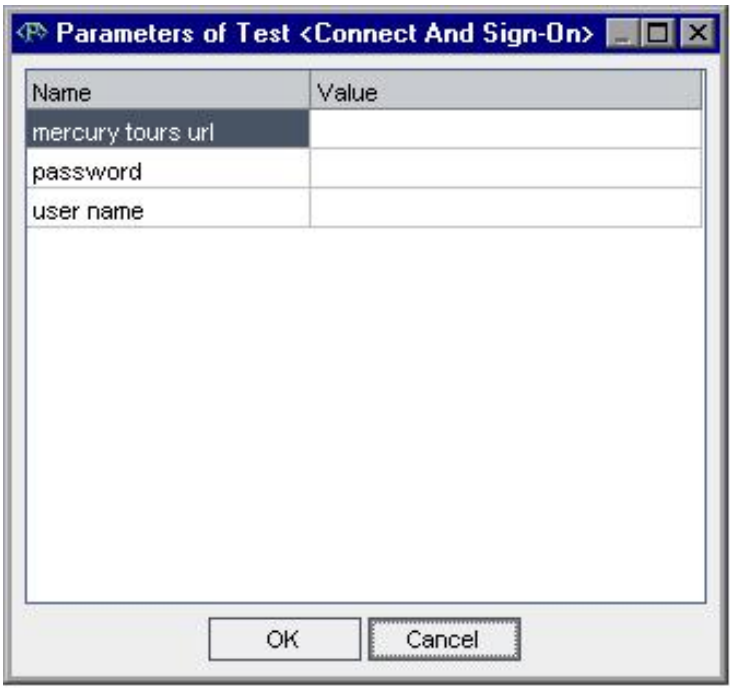

Abb. 15: *Parameter eines Testfalls aus Quality Center Tutorial [10, S.59]*

Für die Entwicklung eines Werkzeugs, das Testfälle automatisiert in das Quality Center exportiert (vgl. 1.2), spielt das Test Plan Modul eine zentrale Rolle. Erstellte Testfälle müssen hierfür im Repository dieses Moduls hinterlegt werden. Das Kapitel 6.2 geht hierauf genauer ein.

Test Lab Modul

Das Test Lab Modul ist für die Durchführung der im Test Plan Modul erstellten Tests zuständig. Hier kann bestimmt werden, welches Testdurchführungsverfahren für die erstellten Tests gewählt wird. Dies können manuell durchgeführte oder mittels geschriebener Skripte automatisiert durchgeführte Tests sein. Automatisierte Tests können mit WinRunner oder QuickTest Professional verknüpft werden. In diesem Fall startet Quality Center das ausgewählte Testwerkzeug und importiert die Ergebnisse. Des Weiteren können einzelne Testfälle zu einem *Test set*, oder nach dem German Testing Board Standardglossar [6, S.58], zu einer Testsuite zusammengeführt werden. Ein Test set kann mit einem Start-Datum und einem Ende-Datum versehen werden. Somit kann der Testmanager im Release Modul auf Verzögerungen hingewiesen werden. Für eine Menge von Test sets kann ein Ablaufplan erstellt werden, um Bedingungen für deren Abläufe festzulegen.
Der in Abb. 16 dargestellte Ablaufplan (engl. *Execution Flow*) zeigt, dass Test 2 nur durchgeführt werden darf, nachdem Test 1 beendet (*finished*, blauer Pfeil) wurde. Test 3 darf nur

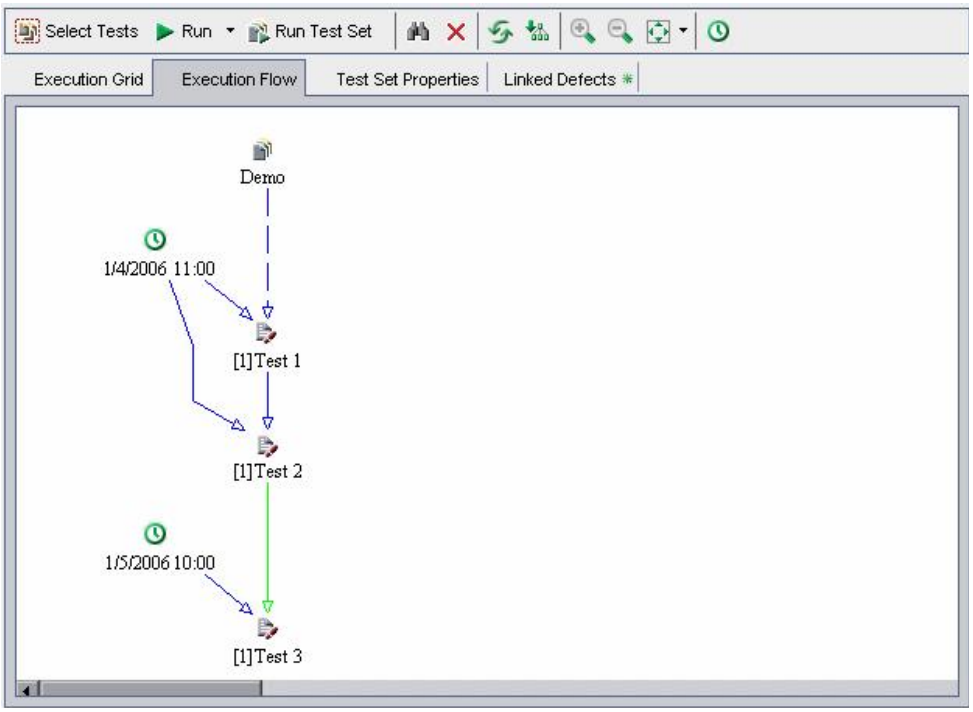

Abb. 16: *Ablaufplanung von Test sets aus Quality Center Tutorial [10, S.82]*

nach erfolgreicher Beendigung (*passed*, grüner Pfeil) von Test 2 durchgeführt werden. Zusätzlich können zeitliche Abhängigkeiten angegeben werden (grüne Uhr). In diesem Fall erfolgt die Durchführung von Test 1 und Test 2 einen Tag vor Test 3. Test 1 hat keine Vorbedingung (blauer gestrichelter Pfeil).

Defect Modul

Das Defect Modul dient der Verwaltung und Dokumentation von Defekten. Das Fehlermanagement sieht für Defekte folgende Zustände vor:

- New
- Open
- Fixed
- Closed

Aufgetretene Defekte erhalten standardmäßig den Zustand *New*. Der Projekt- oder Testmanager entscheidet über Ablehnung (engl. *Reject*) oder Annahme eines Defektes. Wird ein Defekt akzeptiert, so erhält er eine Fehlerpriorität und geht in den Zustand *Open* über. Nach der Fehlerbehebung durch einen Entwickler setzt dieser den Defekt in den Zustand *Fixed*. Durch einen Fehlernachtest wird geprüft, ob der Defekt erneut auftritt (*Reopened*) und wieder in den Zustand *Open* versetzt wird. War die Fehlerbehebung erfolgreich, wird in den Zustand *Closed* gewechselt. Sollte die Fehlerbehebung neue Defekte verursacht haben, wiederholt sich der Prozess. Der Verlauf eines Defektes wird in Abb. 17 grafisch dargestellt.

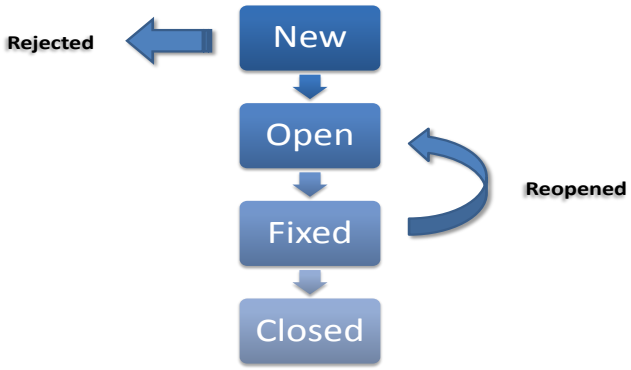

Abb. 17: *Defect tracking in Anlehnung an Quality Center Tutorial [10, S.106]*

Aufgedeckte Defekte können mit den verantwortlichen Testfällen verknüpft werden. Da die Testfälle wiederum mit bestimmten Anforderungen verbunden sind, kann sofort festgestellt werden, welche Anforderung durch Auftreten eines Defektes nicht erfüllt wird.

4.3 Architektur

Die Architektur von Quality Center ist eine 3-Tier-Architektur (vgl. Abb. 18). Auf der Client-Tier liegt die Anwendungskomponente mit ihrem GUI front-end. Sie verwendet den Browser für den Zugang zum Quality Center. Die Middle-Tier beinhaltet den Web-Server für die Kommunikation mit den Clients und den Application-Server mit der Anwendungslogik. Für die Kommunikation zwischen Clients und Web-Server wird das HTTP (*Hypertext Transfer Protocol*) Protokoll genutzt. Als Application-Server wird ein JBOSS Application-Server eingesetzt. Quality Center basiert auf Java und benötigt für die Kommunikation mit dem Application-Server das J2EE-Middleware Framework des JBOSS Application-Server. Auf der Server-Tier liegt der Datenbank-Server. Das Quality Center benutzt hierfür wahlweise Oracle oder MS SQL Server.

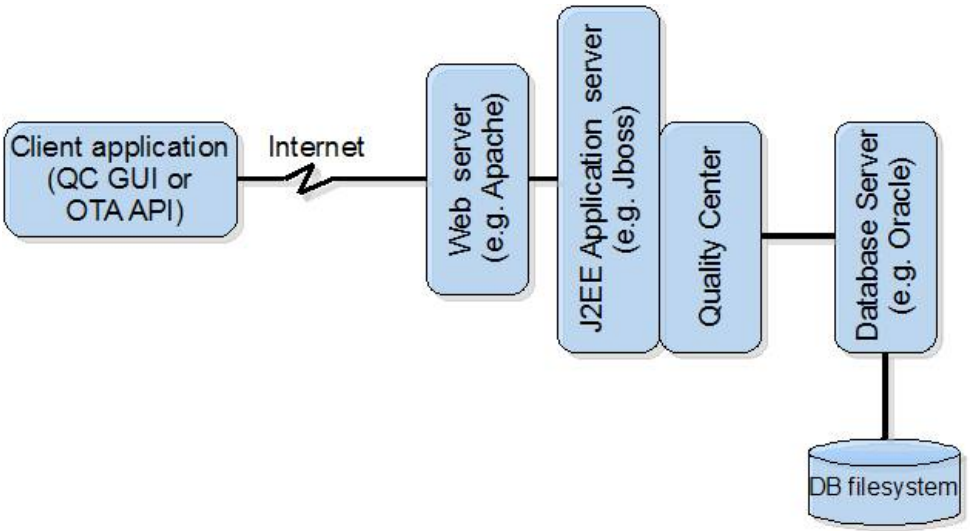

Abb. 18: *Quality Center Architektur aus Extending QC with Open Test Architecture (OTA) API [15]*

4.4 OTA-API

Für die Integration externer Applikationen wie z. B. Winrunner oder QuickTest Professional stellt Quality Center die OTA API (*Open Test Architecture Application Programming Interface*) zur Verfügung. Die OTA-API ist eine COM (*Component Object Model*) Bibliothek. Zur Kommunikation mit dem Quality Center über die OTA-API wird als Programmiersprache Microsoft Visual Basic 6.0 empfohlen [29]. Es kann aber auch jede andere COM-kompatible Sprache benutzt werden (VBScript, C++, C#, etc.). Neben der Integration externer Applikationen kann die OTA-API auch für die Entwicklung individuell erstellter Software (*third party tool*) genutzt werden. Die Schnittstelle ermöglicht eine Kommunikation mit Quality Center ohne Nutzung des GUI front-end sowie den direkten Zugriff auf dessen Datenbank. Hierfür übernimmt sie die Aufgaben einer Middleware.

Die Bibliothek befindet sich in der *OTAClient.dll* und wird nach Installation und erstmaliger Ausführung des Clients automatisch auf den entsprechenden Zielrechner heruntergeladen. Für die Entwicklung von Software zur Interaktion mit Quality Center muss die *OTA COM Type Library* als Referenz dem Projekt hinzugefügt werden.

Der erste Schritt eines Projekts ist die Herstellung einer Verbindung zum Server von Quality Center. Dies geschieht über das *TDConnection* Objekt. Über eine Instanz des TDConnection Objekts oder Objektinstanzen, die über das TDConnection Objekt erzeugt wurden, können fast alle weiteren benötigten Objekte der OTA-API erzeugt werden. Das folgende Beispiel (Listing 4) zeigt einen Verbindungsaufbau mit Visual Basic zum Server ohne zusätzliche Plausibilitätsprüfungen (engl. *Sanity Checks*):

```
0  Dim QCConnection As TDAPIOLELib.TDConnection

   QCConnection = CreateObject("TDApiOle80.TDConnection") 'Get the TDConnection object
   QCConnection.InitConnection("<server_URL>")            'Create the connection
   QCConnection.Login("<username>", "<password>")         'Log on to server
5  QCConnection.Connect("<domain>", "<project")           'Connect to the domain and project
```

Listing 4: *Verbindungsaufbau mittels TDConnection*

4.5 Microsoft Excel Add-in

Infolge der Recherche nach vorhandenen oder ähnlichen Lösungen für die Aufgabenstellung stößt man auf das Microsoft Excel Add-in für Quality Center. Dieses Add-in ermöglicht den Export von Anforderungen, Defekten und vor allem Testfällen von Microsoft Excel nach Quality Center. Die Testfälle müssen hierfür in einem zu den Test Steps im Test Plan Modul (vgl. Abb. 14) deckungsleichen Format in Microsoft Excel festgehalten werden. Die Abb. 19 zeigt beispielhaft die Formatierung von Testfällen des Microsoft Excel Add-in für das Test Plan Modul in Quality Center.

Es ist zu prüfen, inwiefern das Add-in den zu stellenden Anforderungen in Kap. 5 bereits gerecht wird. eine Bewertung hierüber erfolgt in Kap. 5.3.

	A	B	C	D	E	F
1	**Subject**	**Test Name**	**Description**	**Step Name**	**Step Description**	**Expected Results**
2	Subject_1	Test_1	Description for Test_1	Step_1	Description for Step_1	Expected results for Step_1
3	Subject_1	Test_1	Description for Test_1	Step_2	Description for Step_2	Expected results for Step_2
4	Subject_1\Subject_2	Test_2	Description for Test_2	Step_1	Description for Step_1	Expected results for Step_1
5	Subject_1\Subject_2	Test_2	Description for Test_2	Step_2	Description for Step_2	Expected results for Step_2

Abb. 19: *Erstellung von Testfällen im Microsoft Excel Add-in aus Microsoft Excel Add-in Guide [11]*

5 Anforderungsdefinition an eine Toolunterstützung

Im folgenden Abschnitt werden die Anforderungen definiert, die an das zu entwickelnde Werkzeug zu stellen sind. Dabei wird zunächst auf die Ziele eingegangen, die das Werkzeug erfüllen soll. Anschließend soll der Nutzen anhand der vorgegebenen Ziele dargestellt werden. Zusätzlich erfolgt eine Bewertung vorhandener Lösungen in Bezug auf die Anforderungen.

5.1 Ziele

Eine generische Testablaufspezifikation für konkrete Testfälle im Systemtest auf Basis von Microsoft Excel soll entworfen werden. Dafür ist eine Schablone (engl. *Template*) vorgesehen, die die Testfälle nach definiertem Schema spezifiziert aufnehmen kann. Signifikant ist hierbei die Attribuierung der Testfälle für deren Dokumentation. Das Test Plan Modul des Quality Center weist in diesem Zusammenhang deutliche Schwächen auf. Die Attribute zur Testfall- und Testschrittbeschreibung sind sehr allgemein gehalten (vgl. Kap. 4.2). Eine systematische Beschreibung der Testfälle obliegt dem Tester.

Bedienung und Erfassung der Schablone sollen größtmögliche Simplizität aufweisen, indem es sich selbsterklärend, intuitiv und ergonomisch darstellt. Dabei darf es nicht an Funktionalität einbüßen und muss einen hohen Grad an Flexibilität und Adaptabilität aufweisen. In Bezug auf die Flexibilität müssen die Tester die Freiheit haben, Attribute auszulassen. Es darf diesbezüglich kein Zwang bestehen. Zwar soll das Werkzeug die Dokumentation der Testfälle verbessern, aber es soll auch unter herrschendem Zeitdruck bei Vernachlässigung der Testfalldokumentation noch auf Akzeptanz bei den Testanalysten stoßen. Bezüglich der Adaptabilität ist ein Hinzufügen individueller Attribute vorgesehen, sodass wünschenswerte Eigenschaftsbeschreibungen ergänzt werden können. Darüber hinaus muss das Werkzeug flexibel gegenüber jeder Art von Tests im Systemtest sein. Alle für den Systemtest vorgesehenen Testarten und Testziele müssen dokumentiert werden können.

Nach Übertragung der Testfälle in die Schablone soll die Testablaufspezifikation in das Test Plan Modul von Quality Center exportiert werden. Dazu müssen die jeweiligen Attribute der Testfälle auf die Attribute des Test Plan Moduls von Quality Center übertragen werden (engl. *mapping*). Auch die Verwendung von Parametern, wie das Test Plan Modul es unterstützt (vgl. Kap. 4.2), soll zur Verfügung stehen. Zusätzlich soll die Option bestehen, zusammen mit den Testfällen die Testfallspezifikation als Anhang für den übergeordneten Order dieser Testfälle zu exportieren.

5.2 Nutzen

Der aus dem Werkzeug zu ziehende Nutzen für den Fachbereich ist u. a. Zeitersparnis. Die Vorgabe eines systematischen Schemas für die Dokumentation der Testfälle spart Zeit, da sich die Tester kein eigenes überlegen müssen.

Die durch das Template entstehende Einheitlichkeit und Systematik in der Struktur der Testfallbeschreibung fördert deren Verständlichkeit aufgrund einer wiederkehrenden Darstellung. Erstellt jeder Tester sein eigenes Schema, müssen andere sich darin erst zurechtfinden. Die Wahrscheinlichkeit des Vergessens wichtiger Merkmalbeschreibungen der Testfälle wird reduziert, die Methodik zur Testfallbeschreibung verbessert und der Nachweis eines sorgfältigen Vorgehens gegenüber Auftraggebern erleichtert.

Die Aussage von Vigenschow "Wir erfassen uns bekannte Strukturen besser und schneller als völlig neue"[45, S.41] bezieht sich zwar auf das Layout von Quellcode, die allgemeingültige Bedeutung lässt sich aber auch auf die Testfallspezifikation übertragen.

Aufgrund einer gesteigerten Systematik in der Testfallspezifikation, im Unterschied zur Dar-

stellung im Test Plan Modul, ist bezüglich der Dokumentation eine höhere Aussagekraft der Testfälle zu erwarten. Das Quality Center ermöglicht mithilfe des Release Moduls eine grafische Darstellung der Anforderungsüberdeckung durch die Testfälle. Ebenso wichtig ist es, in der Testfallspezifikation zu dokumentieren, welcher Testfall welche Anforderung abdecken soll (engl. *requirements traceability*). Auf den Systemtest bezogen betrifft dies die horizontale Rückverfolgbarkeit. Darunter versteht man „[...]das Verfolgen von Anforderungen einer Teststufe über die Ebenen der Testdokumentation (z. B. Testkonzept, Testentwurfsspezifikation, Testfallspezifikation, Testablaufspezifikation oder Testskripte)“ [17, S.23]. Auch die Kategorisierung der Testfälle soll unterstützt werden. Es muss z. B. zwischen einem „normalen“ Test, einem Regressionstest oder einem Fehlernachtest unterschieden werden.

Infolge der tabellarischen Darstellung soll auch die Wartbarkeit der Testfälle verbessert werden. Bestehende Testfälle sollen schnell änderbar sein, neue Testfälle unkompliziert hinzugefügt oder bestehende Testfälle benutzerfreundlich gelöscht werden können.

Ein weiterer Vorteil des Werkzeugs ist die Möglichkeit, die Testfälle ohne bestehende Verbindung zum Quality Center zu dokumentieren. Verbindungsprobleme z. B. infolge von Netzwerkproblemen stellen somit kein Hindernis dar. Hieraus ergibt sich ein weiterer entscheidender Vorteil. Jeder mit dem Quality Center verbundene Benutzer benötigt eine Lizenz. Diese Lizenzen müssen gekauft werden. Die Möglichkeit, Testfälle ohne Verbindung zum Quality Center zu spezifizieren und anschließend zu exportieren, beschränkt die Anzahl benötigter Lizenzen, da dies über lediglich eine Lizenz erfolgen kann.

Einige Tester bei Logica berichteten von störenden Abbrüchen ihrer Sessions nach Inaktivität innerhalb eines bestimmten Zeitraums (z. B. Unterbrechung wegen Telefonaten). Dies kann sehr ärgerlich sein, da Quality Center nicht gespeicherte Eingaben mit Abbruch der Session wieder verwirft. Zwar ist dies eine Einstellung des Servers, die der Administrator mittels des Parameters *WAIT_BEFORE_DISCONNECT* (vgl. [9, S.133]) bestimmen kann, aber auch hierfür stellt das Werkzeug eine Lösung dar.

5.3 Bewertung vorhandener Lösungen

Das in Kap. 4.5 vorgestellte Microsoft Excel Add-in folgt in seinem Ansatz den Grundzügen der Anforderungsdefinition. Es bietet die Möglichkeit einer tabellarischen Dokumentation der Testfälle auf Basis von Microsoft Excel und deren anschließender Export nach Quality Center. Das ermöglicht eine Spezifizierung der Testfälle ohne Quality Center und die Reduzierung benötigter Lizenzen. Es stellt allerdings kein Template dar, was eine intuitive Bedienung einschränkt. Es bedarf einer Anleitung, die vorgibt, wie die Testfälle zu dokumentieren sind.

Die Umsetzung der Spezifikation von Testfällen und deren Abläufe beim Excel Add-in bietet noch weniger Möglichkeiten als das Test Plan Modul. Wird im Test Plan Modul ein neuer Testfall angelegt, so werden zunächst *Test Type* und *Test Status* angegeben. Des Weiteren kann der Testfall noch mit einer *Description* und einem *Comment* versehen werden. Danach folgt die Eingabe der einzelnen Testschritte. An diesem Punkt erst setzt das Excel Add-in ein. Es erlaubt nur die Beschreibung der einzelnen Testschritte auf identische Art und Weise, so wie es das Test Plan Modul vorsieht. Somit erfolgt eine systematische Strukturierung der Testfall- und Testablaufbeschreibung auf dem gleichen Niveau wie im Test Plan Modul.

Ein weiterer Mangel besteht darin, dass das Excel Add-in die Benutzung der in Kap. 4.2 beschriebenen Parameter nicht vorsieht. Diese müssten im Test Plan Modul nachträglich hinzugefügt werden.

Die in Kap. 5.1 genannten Anforderungen, vor allem eine Verbesserung der Testfalldokumentation, kann das Werkzeug nicht erfüllen und somit auch nicht den in Kap. 5.2 beschriebenen Nutzen bringen.

6 Konzeption

Dieser Abschnitt beschreibt die Vorüberlegungen für das Konzept zur Umsetzung der Anforderungen.

Die Clients von Quality Center 9.2 können nur auf Microsoft Windows Betriebssystemen ausgeführt werden und benötigen hierfür den Microsoft eigenen Internet Explorer. Für die Entwicklung des Werkzeugs bedeutet dies, dass keine Plattformunabhängigkeit zu berücksichtigen ist.

Für eine den Anforderungen konforme Umsetzung des Werkzeugs sind zwei Ansätze zu betrachten:

Die erste Variante wäre eine vollständige Umsetzung von Testfallspezifikation und Client in Microsoft Excel. Dafür spräche die Möglichkeit, mittels der Microsoft-Office eigenen Skriptsprache VBA (*Visual Basic for Applications*) die OTA-API des Quality Centers ansprechen zu können, da VBA ein benötigtes Maß an Kompatibilität zu VB aufweist, obwohl diese seit Visual Basic 6.0 stark eingeschränkt wurde. Vonseiten Hewlett-Packard wird jedoch die Benutzung von Visual Basic 6.0 empfohlen (vgl. Kap. 4.4). Die Möglichkeiten, eine ansprechende und benutzerfreundliche GUI in VBA zu erstellen, sind beschränkt. Da das Werkzeug für den Einsatz bei den Kunden von Logica vorgesehen ist, sollte dieser Aspekt jedoch nicht vernachlässigt werden.

Die zweite Variante wäre eine Trennung zwischen einem Microsoft Excel Template (Testfall- und Testablaufspezifikation) und einer eigenständigen Applikation (Client) für den Export der Testablaufspezifikation. Diese Variante böte mehrere Vorteile gegenüber ersteren. Eine Trennung von Template und Applikation für den Export erhöht die Wartbarkeit. Diese Art der Modularisierung erleichtert Anpassungen an zukünftig mögliche Veränderungen der OTA-Api. Benötigte Funktionalitäten für das Template können als Makros in VBA realisiert werden und sind getrennt von der Kommunikation mit der OTA Schnittstelle. Der Client hingegen kann somit in Visual Basic 6.0 realisiert werden.

Die zweite Variante unterstützt zudem die Möglichkeit, verschiedene Szenariotests aus verschiedenen Templates zu exportieren. Dies würde bei der ersten Variante paradox erscheinen, da jedes Template sein eigenes Makro in einer Excel Datei beinhalten würde. Somit wäre die erste Version darauf ausgelegt, nur die sich in der eigenen Datei befindlichen Testfälle zu exportieren. Dies hätte zur Folge, dass für jeden Exportvorgang aus n verschiedenen Tabellen n mal eine Verbindung zu Quality Center hergestellt werden müsste, bei der sich der Benutzer jedes Mal authentifizieren müsste. Auch das in Kap. 4.5 vorgestellte Excel Add-in weist diesen Umstand auf.

Da die Vorteile der zweiten Variante überwiegen, empfiehlt sich diese Art der Umsetzung.

6.1 Testfallspezifikation

Um die Aussagekraft der Testfälle im Vergleich zum Test Plan Modul zu verbessern, wird eine systematische Dokumentation der Testfälle benötigt. Verfolgbarkeit und Nachvollziehbarkeit (engl. *Traceability*) spielen dabei eine zentrale Rolle. Diese Eigenschaften sind wichtig für die Reproduzierbarkeit von Fehlern und das Verständnis aller Beteiligten für die Durchführung der Testfälle.

> "It is of great importance to document the test cases that must be executed, so that all parties involved understand exactly what happens during execution of a test case. This means that you must not only document a test case during its execution, but also when you have analysed the test results." [34, S.78]

> "It must be possible to audit the testing process to check that it has been carried out correctly." [36, S.451]

Nach Schneider [32, S.87] sollten die Testfälle in einer Tabelle mit mindestens folgenden Aspekten festgehalten werden:

- eine ID für eine eindeutige Identifizierung eines Testfalls
- Testdaten, wie Eingaben und Parameter
- Vorbereitende Schritte für einen benötigten Ausgangszustand
- eine Beschreibung zur Durchführung der Tests
- Sollresultate wie Rückgabewert oder Bildschirmanzeige

Da das Test Plan Modul bereits den Testfällen eine ID zuweist, würde die Vergabe einer zusätzlichen ID Redundanz fördern. Daher wird die gleiche ID den Tesfällen in der Testfallspezifikation zugewiesen werden.

Sneed geht bei der Testfallbeschreibung noch detaillierter vor und schlägt folgende Attribuierung vor [35, S.85]:

- Testfall ID
- Testfalltyp (manuell oder automatisch)
- Testfallzweck
- Testfallquelle (wovon der Testfall abgeleitet wurde)
- Anforderung (abzudeckende Anforderung)
- Anwendungsfall (abzudeckender Anwendungsfall)
- Testobjekte (Objekte, die von diesem Testfall betroffen sind)
- Vorgängertestfall
- Auslöser (Ereignis, das diesen Test auslöst)
- Vorbedingungen
- Nachbedingungen
- Eingaben (Eingangsdaten)
- Ausgaben (Ausgabedaten)
- Testumgebung (Umgebung in der getestet wird)
- Testfallstatus (Status des Testfalls)
- Fehlermeldung (Verweis auf Fehlermeldung)

Diese Testfallbeschreibung ist wesentlich umfassender, allerdings vermengen sich hier Attribute des Test Plan Moduls mit denen des Test Lab Moduls, das für die Ausführung der Testfälle zuständig ist. Das Attribut *Fehlermeldung* ist eine Eigenschaft, die erst nach Durchführung eines Testfalls ermittelt wird. Die Durchführung der Testfälle ist Aufgabe des Test Lab Moduls. Der Workflow des Werkzeugs soll grundsätzlich nur eine Richtung haben - von der Testfallspezifikation ins Test Plan Modul. Abgesehen von der Testfall ID sollen keine Informationen zurückgeführt werden, z. B. ob eine Testfalldurchführung erfolgreich (engl. *passed*) war oder fehlgeschlagen (engl. *failed*) ist. Nachdem die Testfälle exportiert wurden, soll nicht weiter in den Arbeitsablauf von Quality Center eingegriffen werden. Dies sind Arbeitsschritte, die die anderen Module des Quality Center betreffen. Es würde ein zusätzlicher *Overhead* geschaffen und Redundanz gefördert werden. Für das Attribut *Testfallstatus* gilt dies ebenfalls. Allerdings differiert die Bedeutung dieses Attributs zwischen Sneed und Quality Center. Nach Sneed [35, S.90] kann das Attribut folgende Status annehmen:

- ermittelt

- spezifiziert
- reviewed
- getestet
  - ok
  - fehlerhaft
- validiert

Die Status *getestet* und *validiert* können erst nach Testdurchführung ermittelt werden. Das Test Plan Modul verwendet hierfür folgende Status:

- Ready
- Design
- Imported
- Repair

Zwar ist der Status *Repair* ebenfalls ein Zustand, der erst nach einer Testdurchführung angenommen werden kann, allerdings fällt die Korrektur eines Testfalls in das Aufgabengebiet des Testdesigns im Test Plan Modul zurück. Der Zustand *Repair* bezieht sich hierbei nicht auf einen fehlgeschlagenen, sondern auf einen fehlerhaften Testfall. Daher kann *Testfallstatus* mit den Attributen wie es das Test Plan Modul vorsieht für die Testfallspezifikation übernommen werden.

6.1.1 Testfälle

In Abwägung der o. g. genannten Empfehlungen soll das Konzept für die Umsetzung eines Templates folgende Attribute für einen Testfall vorsehen:

- Test ID
- Export Test Case
- Test Name
- Test Status
- Objective
- Test Priority
- Test Type
- Use-Case Coverage
- Requirement Coverage
- Version of Specification
- Version of System Under Test
- Comment

Abgesehen von den im Test Plan Modul bereits vorhandenen Attributen wie *Test Name*, *Test Status*, *Test Type* und *Comment* sollen alle hinzugekommenen Attribute auf das im Test Plan Modul für Testfälle vorhandene Attribut *Description* übertragen werden (*mapping*). Das Attribut *Description* für Testfälle wird im Template nicht mit aufgenommen.

Test ID

Die *Test ID* ist eine vom Test Plan Modul vergebene ID für einen Testfall. Nach Vergabe der ID muss diese in das Template zurückgeschrieben werden, um eine eindeutige Identifizierung des Testfalls zu ermöglichen. Außerdem zeigt die Vergabe einer ID im Template, dass ein Testfall bereits (mindestens) einmal exportiert wurde.

Export Test Case

Das Attribut *Export Test Case* dient nur der Kennzeichnung, ob ein Testfall exportiert werden soll oder nicht.

Test Name

Test Name ist ein für den Testfall eindeutig zu vergebener Name. Es ist hilfreich, den Testnamen so zu wählen, dass er bereits erste Hinweise darauf gibt, was der Test bezweckt (z. B. Login_Test, Booking_Test etc.).

Test Status

Das Attribut *Test Status* wird vom Test Plan Modul übernommen. Hier können die o. g. Werte *Ready*, *Design*, *Imported* und *Repair* gesetzt werden.

Objective

Objective soll Informationen über den Testfallzweck geben, um festzuhalten, wozu dieser Testfall ermittelt wurde. Der Zweck des Testfalls entspricht auch dem angestrebten Testziel.

Test Priority

Hier ist die Priorität des Testfalls anzugeben. Sie wird unterschieden in *High*, *Medium* und *Low*. Sie dient der groben Unterscheidung zwischen *must test* und *should test* (vgl. Kap. 2.2.2).

Test Type

Das Test Plan Modul unterscheidet hierbei, ob es sich um einen manuellen (*manual*), WinRunner (*WR_Automated*, LoadRunner (*LR_Scenario*) Test handelt. Nach Absprache mit Logica sollen alle exportierten Testfälle den Status *manual* erhalten. Es besteht nachträglich im Test Plan Modul die Möglichkeit, diesen Status nochmal zu ändern. Für die Implementierung ist daher vorgesehen, diesen Status im Quellcode fest zu vergeben.

Im Template soll dem Attribut *Test Type* eine andere Bedeutung zukommen. Hier soll festgehalten werden, um welche Art von Test es sich handelt. Dafür sind folgende Werte vorgesehen:

- Functional
- Functional (Regression)
- Functional (re-testing)
- Non-Functional

Functional stellt hierbei einen funktionalen Test, den Normalfall im Systemtest, dar. *Non-Functional* bezieht sich auf einen nicht funktionalen Test wie z. B. einen Lasttest oder Stresstest. *Regression* weist auf einen Regressionstest hin.

Use-Case Coverage

Use-Case Coverage soll eine Angabe darüber machen, welcher Anwendungsfall mit dem Testfall abgedeckt werden soll. „Wie bei der Verknüpfung des Testfalles mit der Anforderung ist es [...] erforderlich, den Testfall mit dem Anwendungsfall zu verknüpfen“ [35, S.86].

Requirements Coverage

Wie bereits o. g. und in Kap. 5.2 beschrieben ist es wichtig, dass in der Dokumentation eines Testfalls festgehalten wird, welche Anforderung durch ihn abgedeckt werden soll. Dies geschieht mithilfe des Attributs *Requirements Coverage*.

Version Of Specification

Hier soll eine Angabe über die Version der Systemspezifikation, gegen die getestet wird gemacht werden. Damit wird sichergestellt, dass die aktuellste Version getestet wird. Darüber hinaus ist es bei Regressionstests wichtig nachvollziehen zu können, ob der Test auf Grundlage einer neuen Systemspezifikation überhaupt noch erfolgreich ausgeführt werden kann bzw. ob der Test in seiner Durchführung von den Änderungen der Spezifikation betroffen ist.

Version Of System Under Test

Ebenso wichtig wie die Angabe zur Version der Systemspezifikation ist die Angabe zur Version des *SUT*. Die Reproduzierbarkeit von Fehlern ist oftmals nur bei gleicher Version des Testobjekts möglich. Daher ist es für die Verwaltung von Defekten bei fehlgeschlagenen Testfällen unentbehrlich, eine Angabe zur Version des Testobjekts zu machen.

Comment

Dieses Attribut ist bereits im Test Plan Modul für Testfälle enthalten und wird übernommen. Jegliche Art von Zusatzinformationen zum Testfall können hier hinterlegt werden.

6.1.2 Testschritte

Für die Testschritte sind folgende Attribute vorgesehen:

- Export Design Step
- Step Name
- Description
- Precondition
- Postcondition
- Boundary Condition
- Expected Result
- Parameters
- Parameters Values
- Customized Step Attributes

Gemäß dem Vorgehen bei den Testfällen sollen auch hier, abgesehen von den im Test Plan Modul bereits vorhandenen Attributen, alle hinzugekommenen Attribute auf das für Testschritte vorhandene Attribut *Description* übertragen werden. Im Unterschied zum Vorgehen bei den Testfällen wird das Attribut *Description* für die Testfälle weiterhin übernommen.

Export Design Step

Analog zu *Export Test Case* wird bei *Export Design Step* eine Angabe gemacht, ob der jeweilige Testschritt ins Test Plan Modul exportiert werden soll.

Step Name

Dieses Attribut wird vom Test Plan Modul zur Vergabe eines Namens für einen Testschritt übernommen. Wie bei *Test Name* soll auch hier ein Name verwendet werden, der auf den Kontext des Testschritts schließen lässt.

Description

Description dient der Beschreibung der Vorgehensweise in einem Testschritt. Für einen Testfall zur Anmeldung an einem System könnte für den Testschritt z. B. folgende Beschreibung

gewählt werden: Starten der Applikation, Eingabe des Benutzernamens und Passwortes und anschließende Bestätigung durch Drücken des *OK-Buttons*.

Precondition

Hier sollen alle vor Ausführung des Testschritts benötigten Vorbedingungen festgehalten werden. Falls bestimmte Datensätze in einer Datenbank vorhanden sein müssen, ist dies hier zu dokumentieren. Auch zuvor erfolgreich gelaufene Testfälle können eine Vorbedingung darstellen, wie z. B. die o. g. (erfolgreiche) Anmeldung an einem System.

Postcondition

Als Nachbedingung sollen alle „Zustände der betroffenen Objekte nach der Ausführung des Testfalls“ dokumentiert werden [35, S.87]. Dies können u. a. veränderte Datensätze in der Datenbank sein, z. B. eine Verringerung der Artikelmenge nach einem Bestellvorgang. Nachbedingungen repräsentieren aber nicht das zu erwartende Resultat eines Testfalls bzw. Testschritts.

Boundary Condition

Auch das Dokumentieren von Randbedingungen ist zu empfehlen. Hier könnte z. B. erwähnt werden, dass das Netzwerk zum Zeitpunkt der Testdurchführung eine normale Last aufweist. Randbedingungen sind nicht immer leicht von Vorbedingungen zu Unterscheiden. Sie bleiben meist unerwähnt, da sie vorausgesetzt werden und für eine Vorbedingung zu „schwach“ sind.

Expected Result

Hier werden die für einen Testschritt zu erwartenden Sollwerte oder Reaktionen notiert. Hierbei kann es sich um konkrete Werte handeln, es kann aber auch eine Reaktion des Systems sein, z. B. Ausdruck im Drucker oder Ausgabe am Bildschirm.

Parameters

Unter *Parameters* können die in Kap. 5.1 und Kap. 4.2 beschriebenen Parameter verwendet werden.

Parameters Values

Werden die o. g. Parameter verwendet, können die für diesen Testfall zu verwendeten Werte angegeben werden. Dies dient der Übersicht und Kontrolle mit welchen Werten die Parameter benutzt werden. Werden keine Parameter benutzt, kann hier angeführt werden mit welchen Eingabewerten der Testschritt auszuführen ist.

Customized Step Attributes

Um größtmögliche Flexibilität und Adaptabilität zu fördern, soll die Option bestehen, eigene Attribute für die Testschritte ergänzen zu können. Sind die vorhandenen Attribute nicht ausreichend oder zutreffend können weitere hinzugefügt werden.

6.1.3 Ergonomie

Die Darstellung der Testfallspezifikation soll eine schnelle Erfassung der Testfälle ermöglichen. Dafür ist eine selbstbeschreibende, übersichtliche und benutzerfreundliche Struktur vorgesehen. Im Unterschied zum Excel Add-in soll keine Anleitung benötigt werden, um ein benötigtes Format zu erstellen oder einzuhalten. „Der Anwender soll die gewünschte Aufgabe lösen können (Effektivität), dabei nicht unnötig viel Zeit aufwenden (Effizienz) und mit der Software zufriedenstellend arbeiten können (Zufriedenstellung).“ [48, S.24]. Das Hinzufügen individueller Attribute erfüllt neben einer umfassenden Dokumentation auch den ergonomischen Zweck der

Individualisierbarkeit. „Ein hochoptimiertes und praxisgerechtes, aber starres und unflexibles Benutzungsmodell wird es immer schwer haben, vom Anwender genügend akzepiert zu werden. Lassen sie ihm grundsätzlich Möglichkeiten, seine Software an seine Bedürfnisse anzupassen.“ [48, S.68].

Bezüglich der *Usability* ist für wiederkehrende Abläufe eine Benutzeroberfläche mit Bedienelementen vorgesehen, die den Anwender bei der Interaktion mit dem Werkzeug unterstützen soll. Das Anlegen neuer und Löschen vorhandener Testfälle und -schritte soll dadurch intuitiv geschehen.

6.2 Client Applikation

Die Client Applikation dient dem Export der Testfallspezifikation in das Test Plan Modul des Quality Center. Dafür muss der Client zunächst eine Verbindung zum Server von Quality Center herstellen. Dazu müssen die Benutzerdaten erfragt werden. Anschließend wählt der Benutzer die zu exportierende Testfallspezifikation aus und macht eine Angabe über das Zielverzeichnis. Vor einem Export kann der Benutzer optionale Einstellungen vornehmen. Er hat die Möglichkeit, wie in der Anforderungsdefinition Kap. 5.1 gefordert, einen Anhang für den übergeordneten Testfallordner auszuwählen sowie einen ergänzenden Kommentar hinzuzufügen. Wichtig ist auch eine Option, die den Benutzer entscheiden lässt, ob bereits vorhandene Testfälle im Test Plan Modul, die identisch sind mit den zu exportierenden Testfällen in der Testfallspezifikation, durch beim Exportieren ersetzt werden sollen. Identische Testfälle sind an ihren Namen festzumachen. Es dürfen keine Testfälle mit gleichen Namen innerhalb eines Ordners vorkommen.

Beim Auslesen der Testfallspezifikation soll der Client verwendende Parameter erkennen können. Im Test Plan Modul werden Parameter wie folgt angegeben: <<<*Parameter*>>>. Es ist vorgesehen, dem Benutzer das Setzen von „<<<“ und „>>>“ in der Testfallspezifikation zu ersparen. Daher muss der Client Parameter aus der Testfallspezifikation erkennen und als Parameter im Test Plan Modul ablegen.

Während eines Exportvorgangs sollen die vom Test Plan Modul für die Testfälle vergebenen Identifikationsnummern in die Testfallspezifikation zu den entsprechenden Testfällen zurückgeschrieben werden. Dies ist wichtig, um eine Rückverfolgbarkeit (engl. *tracing*) von bereits exportierten Testfällen zu den Testfällen in der Spezifikation zu gewährleisten.

Nachdem alle Testfälle übertragen wurden, sollen noch allgemeine Informationen in die Testfallspezifikation zurückgeschrieben werden. Hierfür ist vorgesehen, die Lokalisierung der exportierten Testfälle zu protokollieren, beschrieben durch *Domain*, *Project* und *Path*. Auch der Autor (*Author*) soll festgehalten werden. Der Autor ist hierbei derjenige Benutzer, der sich für den Export der Testfallspezifikation beim Quality Center angemeldet hat. Zusätzlich sollen noch Datum und Zeitpunkt des Exportvorgangs dokumentiert werden.

7 Implementierung

Dieser Abschnitt beschreibt die Umsetzung der in Kap. 5 genannten Anforderungen auf Grundlage der in Kap. 6 dargestellten Konzeption.

7.1 Testfallspezifikation

Die Implementierung der Testfallspezifikation erfolgt auf Grundlage von Microsoft Excel. Automatisierte Abläufe sind mithilfe der Microsoft Office 2007 eigenen Entwicklungsumgebung in VBA (*Visual Basic for Applications*) realisiert.

7.1.1 Darstellung der Testfallspezifikation

Die Darstellung der Testfallspezifikation weist eine vertikale Anordnung für die Attribute der Testfälle und deren Testschritte auf. Testfälle werden somit ebenfalls in vertikaler Anordnung dargestellt. Diese Anordnung ist vorteilhaft, da Testfälle zahlreiche Testschritte aufweisen können und Excel wesentlich mehr Zeilen als Spalten zur Verfügung stellt. Sind alle Spalten für Testfälle verbraucht, kann problemlos eine weitere Testfallspezifikation hinzugezogen werden. Würden sich die Spalten auf die einzelnen Testschritte beziehen, wäre die Anzahl der Testschritte sehr begrenzt. Bei Überschreitung der Spalten könnte nicht problemlos eine weitere Testfallspezifikation hinzugezogen werden, da eine Verknüpfung der Testschritte zum zugehörigen Testfall über zwei Testfallspezifikationen hinaus nötig wäre.

Eine farbliche Gestaltung der Elemente soll helfen, Testfälle, Testschritte, Attribute und Bedienelemente voneinander abzugrenzen. Das leere Template ohne hinzugefügte Testfälle oder Testschritte ist in der Abbildung 20 dargestellt.

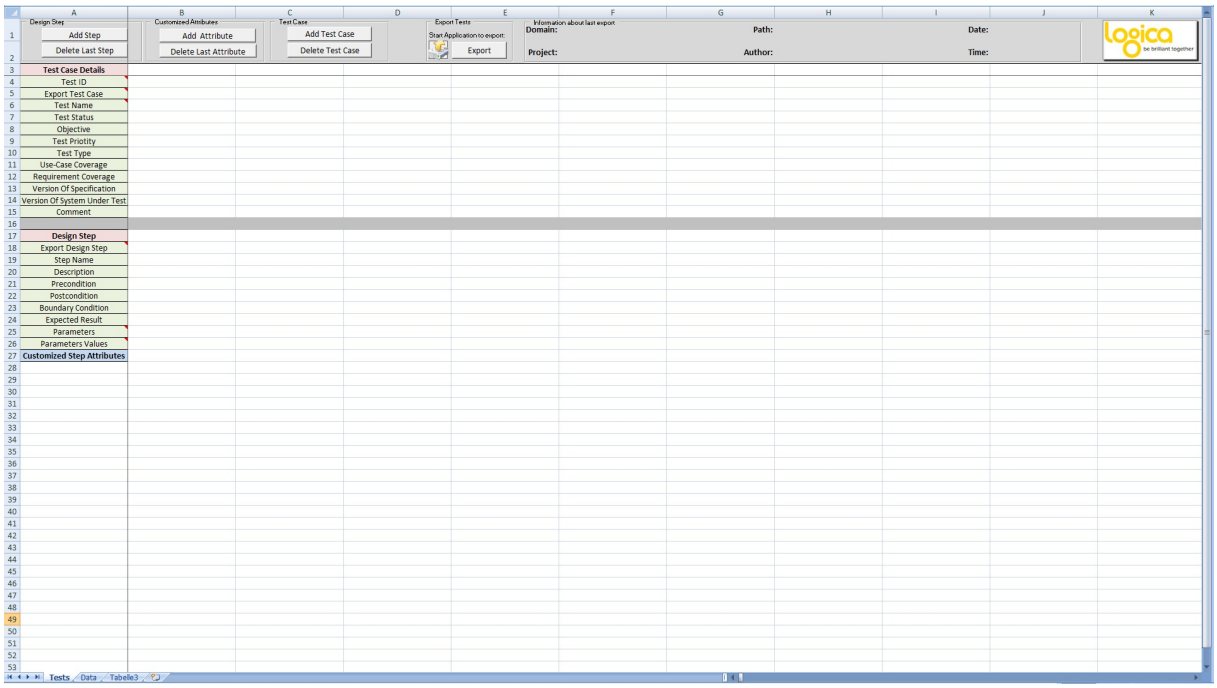

Abb. 20: *Darstellung der leeren Testfallspezifikation*

Die erste Spalte ist für die Attribute der Testfälle und Testschritte vorgesehen. Die Zeilen eins und zwei enthalten die Bedienelemente. Zeile drei ist für das Anlegen der Testfälle bestimmt.

Für den Erhalt der Formatierung sind alle Zellen, in denen keine Einträge durch den Anwender vorgesehen sind, vor Veränderungen geschützt. Zur Aufhebung des Schutzes ist die Eingabe Passwortes nötig. Alle benötigten Eingriffe in die Formatierung werden über die Bedienele-

mente vollzogen. Bei Attributen, die vordefinierte Auswahlmöglichkeiten erwarten, sind Auswahllisten (engl. *Listbox*) implementiert. Für die Testfälle betrifft dies die Attribute *Export Test Case*, *Test Status*, *Test Priority* und *Test Type*.

Für *Export Test Case* besteht die Auswahl zwischen:

- leere Zeile bedeutet, dass der entsprechende Testfall nicht exportiert werden soll
- „*X*“ signalisiert eine Exportanforderung

Für *Test Status* stehen die Status des Test Plan Modul zur Auswahl:

- *Ready*
- *Design*
- *Imported*
- *Repair*

Test Priority ermöglicht die Auswahl zwischen:

- *High*
- *Medium*
- *Low*

Für *Test Status* wird, wie in Kap. 6.1.1 bereits erwähnt, als Standardwert *manual* gewählt. Für die Beschreibung der Testfälle in der Spezifikation wird der *Test Status* für folgende Unterscheidung benutzt:

- Functional
- Functional (regression)
- Functional (re-testing)
- Non-Functional

Bei den Testschritten ist das Attribut *Export Design Step* mit einer Auswahlliste versehen und verwendet hierfür, in Analogie zu *Export Test Case*, die gleiche Auswahl mit entsprechender Bedeutung. Zur Unterstützung einer intuitiven Bedienung ist u. a. für diese beiden Attribute eine Erklärung hinterlegt. Wird der Cursor der Maus über die Attribute bewegt, erscheint automatisch ein Benutzerhinweis.

7.1.2 Bedienelemente der Testfallspezifikation

Zum Schutz der Formatierung und Steigerung der Usability stehen für typische Interaktionen mit der Testfallspezifikation Bedienelemente zur Verfügung. Die Aktionen dieser Bedienelemente werden durch Makros ausgeführt. Mithilfe der Bedienoberfläche können Testfälle, Testschritte und individuelle Attribute gelöscht oder hinzugefügt und der Client gestartet werden. Zusätzlich umfasst die Bedienoberfläche noch Informationsfelder für die zurückzuschreibenden Daten wie *Domain*, *Project*, *Path*, *Author*, *Date* und *Time*.

Hinzufügen eines Testfalls

Der Button *Add Test Case* fügt einen neuen Testfall mit mindestens einem Testschritt hinzu, da ein Testfall aus mindestens einem Testschritt besteht. Die Anzahl der Testschritte orientiert sich hierbei nach dem Testfall mit den meisten Testschritten. Ist z. B. ein Testfall mit drei Testschritten bereits vorhanden, so erhält jeder neu hinzugefügte Testfall ebenfalls drei Testschritte. Da Testschritte für den Export mit einem „*X*“ gekennzeichnet sein müssen, verhindert dies leere Testschritte im Test Plan Modul. Das Makro analysiert die bisherige Anzahl an Testfällen sowie den Testfall mit den meisten Testschritten und fügt den neuen Testfall mit entsprechen-

der Testfallnummer und Testschritten ein. Die Abb. 21 zeigt das Template nach hinzugefügtem Testfall.

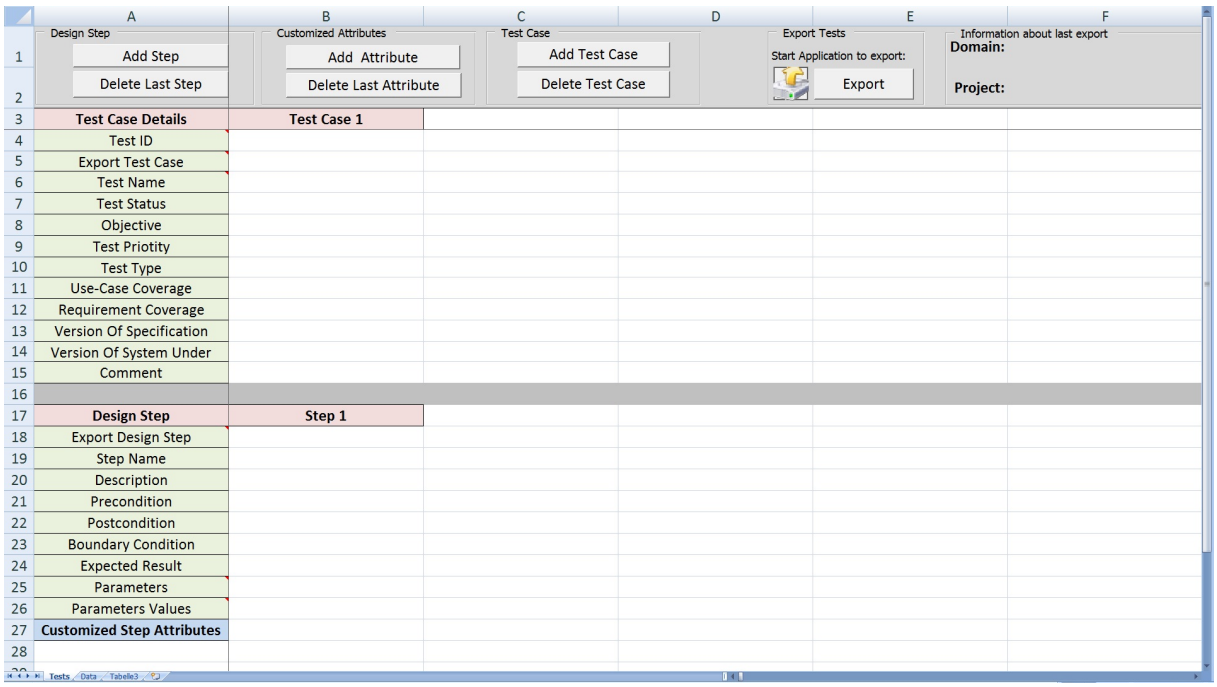

Abb. 21: *Darstellung der Testfallspezifikation nach hinzugefügtem Testfall*

Löschen eines Testfalls

Bei der Benutzung von *Delete Test Case* muss der zu löschende Testfall angegeben werden. Hierzu öffnet sich ein kleines Fenster, das alle verfügbaren Testfälle auflistet. Das Makro identifiziert und löscht den angegebenen Testfall. Handelt es sich dabei nicht um den letzten oder einzigen Testfall, werden alle verbleibenden Testfälle aufgerückt und erhalten eine neue Testfallnummer. Wären drei Testfälle vorhanden und Testfall zwei würde gelöscht, würde Testfall drei der neue Testfall zwei werden. Testfall eins bliebe davon unberührt.

Hinzufügen eines Testschritts

Add Step fügt allen vorhandenen Testfällen einen weiteren Testschritt hinzu. Die Anzahl vorhandener Testschritte richtet sich nach dem Testfall, mit den meisten Testschritten.

Löschen eines Testschritts

Analog zu *Add Step* löscht *Delete last Step* bei allen Testfällen den letzten vorhandenen Testschritt. Es werden alle Attribute des letzten Testschritts gelöscht, auch alle evtl. vorhandenen *Customized Step Attributes*, sowie alle dazugehörigen Eingaben.

Hinzufügen eines Attributs

Mit *Add Attribute* können für die jeweiligen Testschritte individuale Attribute (*Customized Step Attributes*) hinzugefügt werden. Das entsprechende Makro erfragt zunächst, welchem Testschritt das Attribut hinzugefügt werden soll. Dann erfolgt eine Beschreibung des Attributs. Sind alle Eingaben getätigt, sorgt das Makro für die richtige Platzierung des neu hinzugekommenen Attributs und rückt das verschobene Template wieder zusammen, falls es sich nicht um den letzten Testschritt gehandelt hat.

Löschen eines Attributs

Delete last Attribute löscht das letzte vorhandene *Customized Step Attribute* im letzten Testschritt. Wird ein zuvor hinzugefügtes *Customized Step Attribute* aus darüberliegenden Testschritten nicht mehr gebraucht, so kann es nicht gelöscht werden. Sind aber bei einem Testschritt keine Daten für ein *Customized Step Attribute* hinterlegt, so wird das Attribut beim Exportieren der Testfälle nicht beachtet.

Starten des Clients

Der *Export*-Button dient zum Starten der Client Applikation. Über einen File-Browser wird die Datei der Client Applikation ausgewählt und gestartet. Gleichzeitig wird die Testfallspezifikation gespeichert und geschlossen. Für den Fall, dass die Testfallspezifikation als Anhang zusammen mit den Testfällen exportiert werden soll, kann es Probleme bereiten, wenn diese noch geöffnet ist.

7.2 Client Applikation

Die Implementierung des Clients basiert auf Visual Basic 6.0 unter Verwendung von Microsoft Visual Basic 2010 Express als frei und kostenlos zur Verfügung stehende Entwicklungsumgebung [12, Link]. Für die Kommunikation mit dem Server von Quality Center wird die in Kap. 4.4 vorgestellte OTA-Api benutzt. Die Client Applikation stellt dem Anwender zwei Masken zur Verfügung. Die erste Maske dient der Eingabe aller benötigten Verbindungsdaten durch den Benutzer sowie der anschließenden Herstellung der Verbindung zum Quality Center. Die Abb. 37 im Anhang A zeigt den Vorgang der Anmeldung in Form eines Sequenzdiagramms. Die zweite Maske ermöglicht das Exportieren der Testfallspezifikation nach Quality Center unter Auswahl gewisser Optionen. Eine vereinfachte Darstellung eines Exportvorgangs zeigt das Sequenzdiagramm in Abb. 38 im Anhang A.

Das Design der Masken sieht vor, Edit-Felder zu vermeiden und wenn möglich diese durch *Listboxen* zu substituieren. Somit werden Fehlermöglichkeiten reduziert, was sich auch auf den Testumfang auswirkt [45, S.78] und einer der „8 goldenen Regeln des Interface Designs“ nach Shneiderman entspricht nämlich ein Design zu wählen, dass so gestaltet ist, dass der Benutzer kaum Fehler machen kann [20, S.7]. Gleichzeitig sind im Falle von Fehlern oder Fehleingaben Fehlermeldungen in Form von Messageboxen implementiert, die Hinweise zur Behebung eines Fehlers geben.

Für die Verteilung (*mapping*) der Attribute der Testfallspezifikation auf die zur Verfügung stehenden Attribute des Test Plan Moduls wird für die Umsetzung ein Kompromiss eingegangen. Da die vom Administrator vergebenen Rechte verschiedene Sichten erlauben, wird von einer minimalen Sicht ausgegangen. Dies betrifft das Attribut *Priority*. Die Erstellung neuer Testfälle im Test Plan Modul erlaubt bei beschränkter Sicht keine Vergabe einer Priorität für einen Testfall. Daher wird die Priorität in das Textfeld zur Beschreibung (*Description*) eines Testfalls hinzugefügt.

7.2.1 Darstellung der Login-Maske

Die Abb. 22 zeigt die Login-Maske nach Start der Applikation und Abb. 23 nach erfolgreicher Anmeldung beim Quality Center.

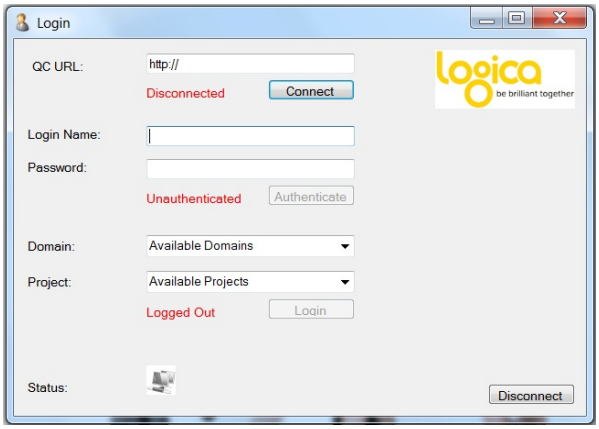

Abb. 22: *Login-Maske nach Start der Applikation*

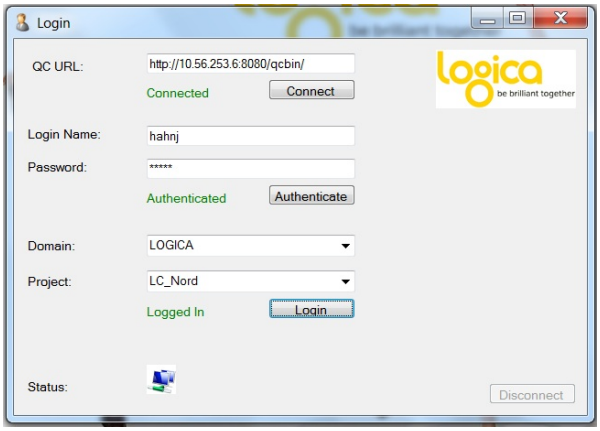

Abb. 23: *Login-Maske nach erfolgreicher Anmeldung*

7.2.2 Bedienelemente der Login-Maske

Nach jeder erfolgreichen Anmeldung wird eine Log-Datei, mit zuletzt angegebener *QC URL* und zuletzt angegebenem *Login Name* erstellt. Bei Eingabe dieser Daten wird die Log-Datei automatisch ausgelesen und als Auswahl zur Vervollständigung eingeblendet. Dies erspart dem Anwender, sich die URL merken zu müssen, sofern sich diese nicht geändert hat, oder ein anderer Server eines Quality Centers angesprochen werden soll. Gleiches gilt für den Benutzernamen bzw. *Login Name*.

Die Implementierungen der Verbindung (Listing 7), Authentifizierung (Listing 8) und Anmeldung am Quality Center Server (Listing 9) sind im Anhang B dargestellt.

Verbindungsherstellung

Die *QC URL* ist die URL zum Server von Quality Center und hat folgende Notation:

http://<Quality Center server name>[:<port number>]/qcbin

Nach korrekter Eingabe und Drücken des *Connect*-Buttons, wird eine Verbindung zum Server von Quality Center hergestellt. Das zugehörige Label wechselt von „Disconnected“ auf „Connected“.

Authentifizierung

Der *Login Name* ist ein vom Administrator vergebener Benutzername und dient zusammen mit dem *Password* der Authentifizierung am Server von Quality Center.

Nach korrekter Eingabe und Drücken des *Authenticate*-Buttons, der erst nach erfolgreich hergestellter Verbindung aktiviert wird, erfolgt eine Authentifizierung des Anwenders am Server von Quality Center. Das zugehörige Label wechselt von „Unauthenticated“ auf „Authenticated“.

Domainauswahl

Abhängig vom angemeldeten Anwender stehen verschieden viele Domains zur Verfügung. Nach erfolgreicher Verbindung und Authentifizierung werden alle dem Benutzer zur Verfügung stehenden Domains ermittelt und das Feld *Domain* mittels einer Listbox mit den zur Auswahl stehenden Domains gefüllt.

Projektauswahl

Abhängig vom angemeldeten Benutzer und der Wahl einer Domain stehen verschieden viele Projekte zur Verfügung. Nach erfolgreicher Verbindung, Authentifizierung und Auswahl der Domain werden alle dem Benutzer zur Verfügung stehenden Projekt ermittelt und das Feld *Project* mittels einer Listbox mit den zur Auswahl stehenden Projekten gefüllt.

Nach Auswahl von *Domain* und *Project* und Drücken des *Login*-Buttons, der erst nach erfolgreicher Authentifizierung und nach Auswahl eines Projekts aktiviert wird, erfolgt die Anmeldung am Server von Quality Center. Das Label wechselt von „Logged Out“ auf „Logged In“ und das Netzwerk-Icon beim Label *Status* wird farbig. Die Login-Maske wird geschlossen und es öffnet sich die Maske zum Exportieren der Testfallspezifikation.

7.2.3 Darstellung der Export-Maske

Die Abb. 24 zeigt die Export-Maske nach erfolgreichem Einloggen in das Quality Center.

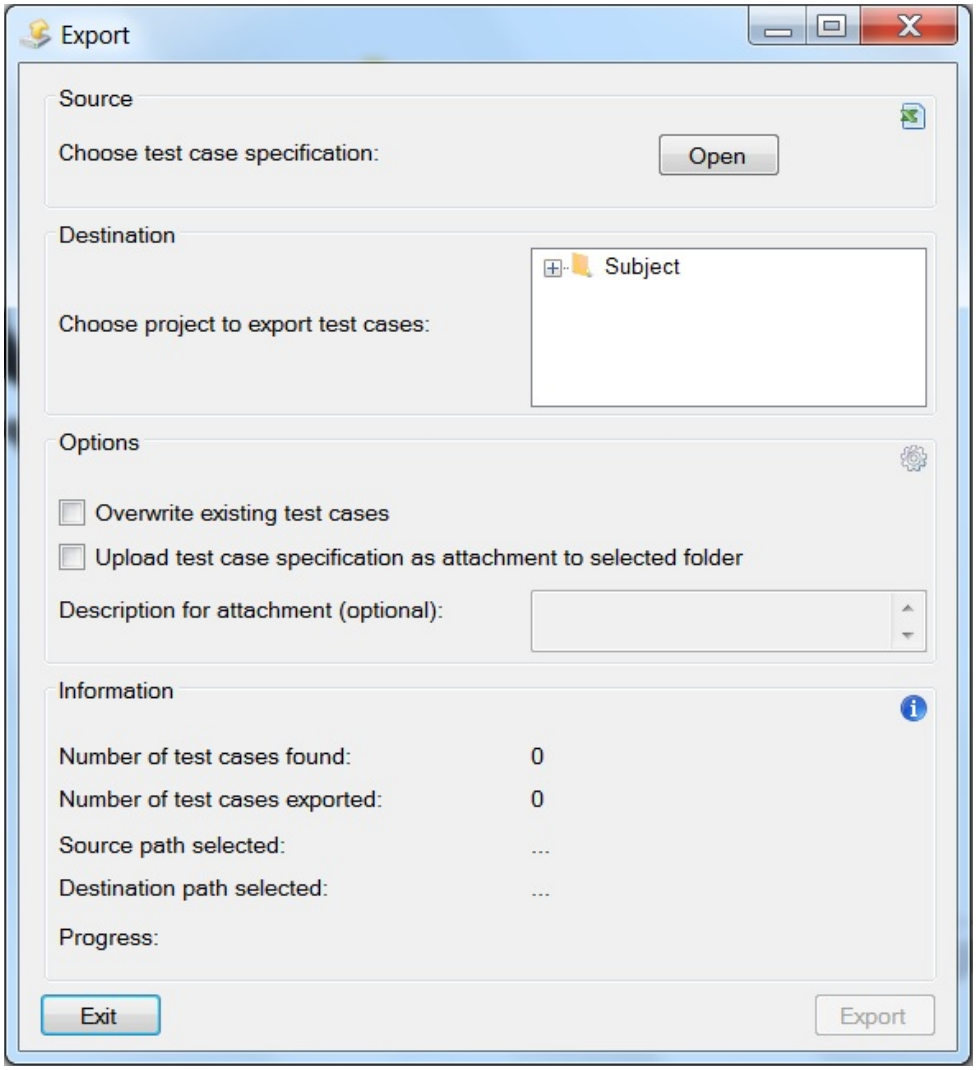

Abb. 24: *Export-Maske nach erfolgreichem Login*

7.2.4 Bedienelemente der Export-Maske

Bevor sich die Export-Maske öffnet, wird ein Verzeichnisbaum erstellt. Die Verzeichnisstruktur vom Quality Center wird rekursiv ausgelesen und anschließend abgebildet.

Die relevanten Implementierungen für das Hinzufügen von Testfällen (Listing 10), Testschritten (Listing 11) und eines Anhangs (Listing 12) sind im Anhang B dargestellt.

Quellpfadangabe

Durch Drücken des *Open*-Buttons im *Source*-Bereich der Maske wird ein Datei-Browser geöffnet, worüber der Quellpfad für die zu exportierende Testfallspezifikation angegeben wird.

Zielpfadangabe

Im *Destination*-Bereich der Maske befindet sich der nachgebildete Verzeichnisbaums von Quality Center. Durch Selektieren eines Ordners erfolgt die Zielpfadangabe. Innerhalb des gewählten Ordners werden die Testfälle der Testfallspezifikation erstellt.

Ersetzen von vorhandenen Testfällen

Im Bereich *Options* der Maske kann mittels eines Hakens in der *Checkbox* bei *Overwrite existing test cases* gewählt werden, ob ein Ersetzen identischer Testfälle durch den Export erwünscht ist. Testfälle werden als identisch angesehen, wenn sie den gleichen Namen aufweisen.

Hinzufügen der Spezifikation als Anhang

Um die gewählte Testfallspezifikation als Anhang zu exportieren, kann die *Checkbox* bei *Upload test specification as attachment to selected folder* im Bereich *Options* mit einem Haken versehen werden.

Starten des Exportvorgangs

Der Button *Export*, der nach Auswahl von Quellpfad und Zielpfad aktiviert wird, startet den Exportvorgang unter Beachtung der gewählten Optionen. Da der Client von HP Quality Center ausschließlich den Internet-Explorer für die Interaktion mit dem Server nutzt, können für eine strukturierte Darstellung der Testfälle im Test Plan Modul die Testfallbeschreibungen in *HTML* (*Hypertext Markup Language*) eingebettet werden. Dies ermöglicht eine Hervorhebung von Schlüsselwörtern, wie z. B. die Attribute, oder eine in rot dargestellte hohe Priorität.

Verlassen der Export-Maske

Der Button *Exit* oder das Symbol „Schließen“ sorgen für ein Verlassen der Export-Maske zurück zur Login-Maske. Hierbei wird automatisch ein Ausloggen des Benutzers und ein Verbindungsabbruch initiiert.

7.2.5 Informationselemente der Export-Maske

Im Bereich *Information* der Export-Maske kann der Benutzer aktuelle Informationen zum Exportvorgang ablesen.

Anzahl vorhandener Testfälle

Nach Starten des Exportvorgangs wird dem Anwender über das Label *Number of test cases found* die Anzahl der in der Testspezifikation gefundenen Testfälle mitgeteilt.

Anzahl exportierter Testfälle

Die aktuelle Anzahl bereits exportierter sowie die Gesamtanzahl nach Abschluss des Exportvorgangs exportierter Testfälle wird über das Label *Number of test cases exported* angegeben. Die Gesamtanzahl aller exportierten Testfälle kann von der Gesamtanzahl gefundener Testfälle differieren, wenn:

- Testfälle mit gleichen Namen in der Testfallspezifikation und im Zielordner des Test Plan Moduls vorhanden sind und die Option identische Testfälle zu überschreiben nicht gewählt wurde,
- Testfälle in der Testfallspezifikation vorhanden sind, bei denen *Export Test Case* nicht mit einem „X“ versehen ist und sie somit nicht für den Export vorgesehen waren.

Quellpfadangabe

Über das Label *Source path selected* erfolgt nach Auswahl der zu exportierenden Testfallspezifikation eine Angabe über deren Verzeichnispfad.

Zielpfadangabe

Über das Label *Destination path selected* erfolgt nach Auswahl eines Zielordners, zur Ablage der Testfälle, eine Angabe über deren Verzeichnispfad.

Fortschrittsanzeige

Über das Label *Progress* wird ein aktueller Fortschrittsbalken dargestellt.

8 Nutzenanalyse

Folgender Abschnitt stellt eine Nutzenanalyse der Testfallspezifikation im Zusammenhang mit der Client Applikation zum Export der Testfälle ins Test Plan Modul vor.

8.1 Durchführung eines Beispiels

Die Nutzenanalyse erfolgt anhand eines Beispiels unter Verwendung der in Kap. 3 evaluierten Testtechniken in Kombination von Klassifikationsbaummethode (mittels CTE/XL), Äquivalenzklassenanalyse und Grenzwertanalyse. Als Testobjekte dienen für das Beispiel die entwickelte Testfallspezifikation mit der Client Applikation.

Klassifikationsbaummethode (mittels CTE/XL)

Für die systematische Erstellung von Testfällen im CTE/XL müssen zunächst relevante Aspekte des Testobjekts ermittelt werden. Für das Exportieren von Testfällen aus der Testfallspezifikation mit der Client Applikation werden folgende Aspekte ermittelt:

- Komposition: *Testfälle*
- Komposition: *Exportanforderung*
- Klassifikation: *Ersetzen identischer Testfälle*
- Klassifikation: *Anhang hinzufügen*
- Klassifikation: *Identische Testfälle in QC*

Der Aspekt *Testfälle* setzt sich als Komposition aus den Klassifikationen *Anzahl Testfälle*, *Anzahl Testschritte pro Testfall* und der weiteren Komposition *Exportanforderung* zusammen. Die Klassen der Klassifikationen *Anzahl Testfälle* und *Anzahl Testschritte pro Testfall* müssen im weiteren Verlauf einer Äquivalenz- und Grenzwertanalyse unterzogen werden. Damit stellt sich die Komposition *Testfälle* wie folgt dar:

- **Komposition**: *Testfälle*
  - Klassifikation: *Anzahl Testfälle*
  - Klassifikation: *Anzahl Testschritte pro Testfall*
  - **Komposition**: *Exportanforderung*

Die Komposition *Exportanforderung* besteht aus den Klassifikationen *Exportanforderung Testfälle* und *Exportanforderung Testschritte* mit den Klassen *Alle*, *Keiner* und *Testfälle / 2* bzw. *Testschritte / 2*.

- **Komposition**: *Exportanforderung*
  - Klassifikation: *Exportanforderung Testfälle*
    - Klasse: *Alle*
    - Klasse: *Keiner*
    - Klasse: *Testfälle / 2*
  - Klassifikation: *Exportanforderung Testschritte*
    - Klasse: *Alle*
    - Klasse: *Keiner*
    - Klasse: *Testschritte / 2*

Die Klassifikationen *Ersetzen identischer Testfälle*, *Anhang hinzufügen* und *Identische Testfälle in QC* erhalten jeweils die Klassen *Ja* und *Nein*:

- Klassifikation: *Ersetzen identischer Testfälle*

  - Klasse: *Ja*
  - Klasse: *Nein*
- Klassifikation: *Anhang hinzufügen*
  - Klasse: *Ja*
  - Klasse: *Nein*
- Klassifikation: *Identische Testfälle in QC*
  - Klasse: *Ja*
  - Klasse: *Nein*

Anhand der bisherigen Einteilung ergibt sich der in Abb. 25 dargestellte Klassifikationsbaum.

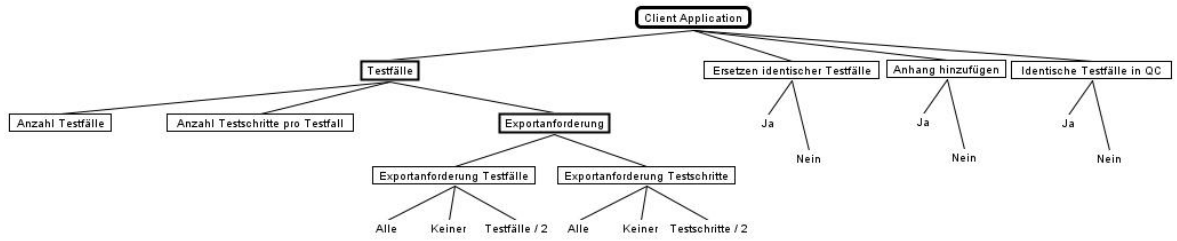

Abb. 25: *Klassifikationsbaum mit Testaspekten für Client Applikation und Testfallspezifikation*

Äquivalenzklassenanalyse

Für die Klassifikationen *Anzahl Testfälle* und *Anzahl Testschritte pro Testfall* sind zunächst Äquivalenzklassenanalysen durchzuführen. Die minimale Anzahl an Testfällen beträgt eins. Allerdings sollte die Client Applikation keine Fehler aufweisen, wenn eine leere Testfallspezifikation ohne Testfälle angegeben wird. Die maximale Anzahl an Testfällen wird durch die Anzahl an zur Verfügung stehenden Spalten bestimmt. In Excel beträgt die maximale Spaltenzahl 256. Dabei muss bedacht werden, dass die Attribute bereits eine Spalte benötigen, somit verbleiben 255 Spalten. Die Anzahl gefundener Testfälle wird in einer Variable vom Typ *long integer* gespeichert. In Visual Basic belegen Variablen von diesem Typ vier Bytes im Speicher, damit ergibt sich ein Wertebereich von -2147483648 bis 2147483647. Es ergeben sich somit folgende gültige (s. Tabelle 18) und ungültige (s. Tabelle 19) Äquivalenzklassen für die Anzahl der Testfälle:

Klasse	Gültige Äquivalenzklassen	Repräsentanten
Anzahl Testfälle	${}_g\text{Äk}_1$: $0 <$ Anzahl Testfälle $\leq$ max. Anzahl Spalten -1 $[1, \ldots, 255]$	10

Tabelle 18: *Gültige Äquivalenzklassen für Anzahl Testfälle*

Klasse	Ungültige Äquivalenzklassen	Repräsentanten
Anzahl Testfälle	${}_u\text{Äk}_1$: Anzahl Testfälle ≤ 0 [MIN_LONG, . . . , 0]	-10
	${}_u\text{Äk}_2$: Anzahl Testfälle $\geq$ max. Anzahl Spalten [256, . . . , MAX_LONG]	260

Tabelle 19: *Ungültige Äquivalenzklassen für Anzahl Testfälle*

Ähnlich verhält es sich mit den Testschritten. Die minimale Anzahl an Testschritten beträgt, wie bei den Testfällen, null. Beim Anlegen eines neuen Testfalls in der Testfallspezifikation wird immer mindestens ein Testschritt hinzugefügt. Hat dieser Testschritt aber keine Exportanforderung gesetzt, bedeutet dies für das Exportieren, dass kein Testschritt vorhanden ist. Die maximale Anzahl an Testschritten wird durch die in Excel zur Verfügung stehenden Zeilen beschränkt. Excel stellt 2^{16} (2 Byte), also 65536 Zeilen zur Verfügung. Die Funktionsleiste und die Testfallbeschreibung benötigen bereits 16 Zeilen, daher verbleiben nur noch 65520 Zeilen. Die Beschreibung für einen Testschritt benötigt mindestens 11 Zeilen, sofern keine *Customized Attributes* hinzugefügt wurden. Somit ergibt sich eine maximale Anzahl von 5956 Testschritten.

$$\text{max. Anzahl Testschritte} = \frac{65520}{11} = 5956.36$$

Wie bei den Testfällen, wird die Anzahl der benutzten Zeilen in einer Variable vom Typ *long integer* gespeichert. Für die Anzahl der Testschritte ergeben sich folgende gültige (s. Tabelle 20) und ungültige (s. Tabelle 21) Äquivalenzklassen:

Klasse	Gültige Äquivalenzklassen	Repräsentanten
Anzahl Testschritte	${}_g\text{Äk}_1$: $0 \leq$ Anzahl Testschritte ≤ 5956 [0, . . . , 5956]	10

Tabelle 20: *Gültige Äquivalenzklassen für Anzahl Testschritte pro Testfall*

Klasse	Ungültige Äquivalenzklassen	Repräsentanten
Anzahl Testschritte	${}_u\text{Äk}_1$: Anzahl Testschritte < 0 [MIN_LONG, . . . , -1]	-10
	${}_u\text{Äk}_2$: Anzahl Testschritte > 5956 [5957, . . . , MAX_LONG]	6000

Tabelle 21: *Ungültige Äquivalenzklassen für Anzahl Testschritte pro Testfall*

Grenzwertanalyse

Für die ermittelten Äquivalenzklassen folgt nun eine Grenzwertanalyse. Die Tabelle 22 zeigt die Grenzwerte der gültigen Äquivalenzklasse für die Testfälle ${}_g\text{ÄK}_1$:

Äquivalenzklasse	Untere Grenzwerte	Obere Grenzwerte
${}_g\text{Äk}_1$	$0, 1, 2$	$254, 255, 256$

Tabelle 22: *Grenzwerte der gültigen Äquivalenzklasse ${}_g\text{ÄK}_1$ für Testfälle*

Die Grenzwerte der ungültigen Äquivalenzklassen für Testfälle ${}_u\text{ÄK}_1$ und ${}_u\text{ÄK}_2$ zeigt die Tabelle 23:

Äquivalenzklasse	Untere Grenzwerte	Obere Grenzwerte
${}_u\text{Äk}_1$	MIN_LONG, MIN_LONG + 1	$-1, 0, 1$
${}_u\text{Äk}_2$	$255, 256, 257$	MAX_LONG − 1, MAX_LONG

Tabelle 23: *Grenzwerte der ungültigen Äquivalenzklassen ${}_u\text{ÄK}_1$ und ${}_u\text{ÄK}_2$ für Testfälle*

Für die Testschritte zeigt die Tabelle 24 die Grenzwerte der gültigen Äquivalenzklasse ${}_g\text{ÄK}_1$:

Äquivalenzklasse	Untere Grenzwerte	Obere Grenzwerte
${}_g\text{Äk}_1$	$-1, 0, 1$	$5955, 5956, 5957$

Tabelle 24: *Grenzwerte der gültigen Äquivalenzklasse ${}_g\text{ÄK}_1$ für Testschritte*

Die Grenzwerte der ungültigen Äquivalenzklassen für Testschritte ${}_u\text{ÄK}_1$ und ${}_u\text{ÄK}_2$ zeigt die Tabelle 25:

Äquivalenzklasse	Untere Grenzwerte	Obere Grenzwerte
${}_u\text{Äk}_1$	MIN_LONG, MIN_LONG + 1	$-2, -1, 0$
${}_u\text{Äk}_2$	$5956, 5957, 5958$	MAX_LONG − 1, MAX_LONG

Tabelle 25: *Grenzwerte der ungültigen Äquivalenzklassen ${}_u\text{ÄK}_1$ und ${}_u\text{ÄK}_2$ für Testschritte*

Fortführung Klassifikationsbaummethode (mittels CTE/XL)

Mit den ermittelten Repräsentanten der Äquivalenzklassen- und Grenzwertanalyse lassen sich die Klassifikationen *Anzahl Testfälle* und *Anzahl Testschritte pro Testfall* um die folgenden Klassen vervollständigen:

- **Klassifikation**: *Anzahl Testfälle*
  - Klasse: MIN_LONG
  - Klasse: MIN_LONG + 1
  - Klasse: -10
  - Klasse: -1
  - Klasse: 0
  - Klasse: 1
  - Klasse: 2
  - Klasse: 10
  - Klasse: 254
  - Klasse: 255
  - Klasse: 256
  - Klasse: 257
  - Klasse: 260
  - Klasse: MAX_LONG $-$ 1
  - Klasse: MAX_LONG
- **Klassifikation**: *Anzahl Testschritte pro Testfall*
  - Klasse: MIN_LONG
  - Klasse: MIN_LONG + 1
  - Klasse: -10
  - Klasse: -2
  - Klasse: -1
  - Klasse: 0
  - Klasse: 1
  - Klasse: 10
  - Klasse: 5955
  - Klasse: 5956
  - Klasse: 5957
  - Klasse: 5958
  - Klasse: 6000
  - Klasse: MAX_LONG $-$ 1
  - Klasse: MAX_LONG

Die Anzahl der Testfälle lässt sich reduzieren. Ein Test mit negativer Anzahl an Testfällen bzw.-Schritten ist nicht oder nur unter erschwerten Bedingungen zu realisieren. Mittels *fault injection* könnte die zur Ermittlung der Testfälle und Testschritte benutzte Variable manipuliert werden, um negative Werte zu erhalten. Dazu müsste in den Quellcode eingegriffen werden. Solche Schritte sollten auf früheren Teststufen geschehen, da es sich hierbei um *Grey-Box* Tests handelt. Im Systemtest sollten jedoch nur *Black-Box* Tests durchgeführt werden. Das gleiche Problem stellen die Tests mit mehr Testfällen als Spalten bzw. mehr Testschritten als Zeilen in der Testfallspezifikation vorhanden sind dar. Auch hier müsste manipulierend in den Quellcode eingegriffen werden. Somit kann auf alle Testfälle mit Repräsentanten aus den ungültigen Äquivalenzklassen verzichtet werden, was entsprechende Grenzwerte mit einbezieht. Bezüglich der Testfälle ist die Null hiervon nicht betroffen. Eine leere Testfallspezifikation lässt sich problemlos testen. Um den Klassifikationsbaum eine übersichtliche Struktur zu verleihen, werden die auszuschließenden Repräsentanten der Äquivalenzklassen und Grenzwerte nicht in den Klassifikationsbaum übernommen. Die Abb. 26 zeigt den erweiterten Klassifikationsbaum.

Für die automatische Generierung der Testfälle mit dem CTE/XL wird eine paarweise Kombination angefordert (vgl. Abb. 27). Die Generierung ergibt 36 Testfälle, die in Abb. 28 zu sehen sind.

Die Äquivalenzklassenabdeckung für *Anzahl Testfälle* liegt somit bei:

$$\text{ÄK-Überdeckung} = \frac{2}{3} \cdot 100\% = 66,6\%$$

Für *Anzahl Testschritte pro Testfall* liegt die Äquivalenzklassenabdeckung bei:

$$\text{ÄK-Überdeckung} = \frac{1}{3} \cdot 100\% = 33,3\%$$

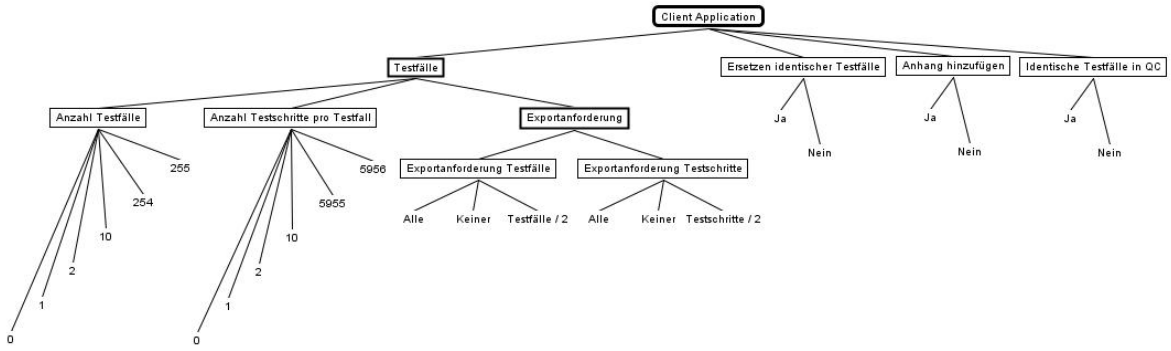

Abb. 26: *Klassifikationsbaum mit Klassen für die Klassifikationen Anzahl Testfälle und Anzahl Testschritte pro Testfall*

Die Grenzwertüberdeckung für *Anzahl Testfälle* liegt bei:

$$\text{GW-Überdeckung} = \frac{6}{15} \cdot 100\% = 40\%$$

Die Grenzwertüberdeckung für *Anzahl Testschritte pro Testfall* liegt bei:

$$\text{GW-Überdeckung} = \frac{6}{15} \cdot 100\% = 40\%$$

Die Überdeckungswerte sind nicht hoch, sollen aber für die weitere Durchführung des Beispiels reichen. Es können noch weitere Einschränkungen mittels der Abhängigkeitsregeln des CTE/XL gemacht werden:

- Die Durchführung eines Testfalls mit maximaler Anzahl an Testfällen und Testschritten erfordert, dass alle Testfälle und Testschritte der Testfallspezifikation eine Exportanforderung stellen (vgl. Abhängigkeitsregel in Abb. 29).
- Bei einer leeren Testfallspezifikation können nicht vorhandene Testfälle und Testschritte keine Exportanforderung stellen (vgl. Abhängigkeitsregel in Abb. 30).
- Bei einer Testfallspezifikation mit Testfällen ohne Testschritte, können die nicht vorhandenen Testschritte keine Exportanforderung stellen (vgl. Abhängigkeitsregel in Abb. 31).
- Ein optionales Ersetzen von identischen Testfällen ist nur interessant, wenn auch identische Testfälle im Test Plan Modul wirklich vorhanden sind (vgl. Abhängigkeitsregel in Abb. 32).
- Eine Kombination mit einer Exportanforderung von der nur die Hälfte aller vorhandenen Testfälle oder Testschritte eine Exportanforderung hat setzt voraus, dass mehr als ein Testfall oder -schritt vorhanden sein muss (vgl. Abhängigkeitsregel in Abb. 33).

Durch Anwendung der o. g. Regeln reduziert sich die Anzahl der Testfälle auf 16, die in die Testfallspezifikation aufzunehmen sind (vgl. Abb. 34).

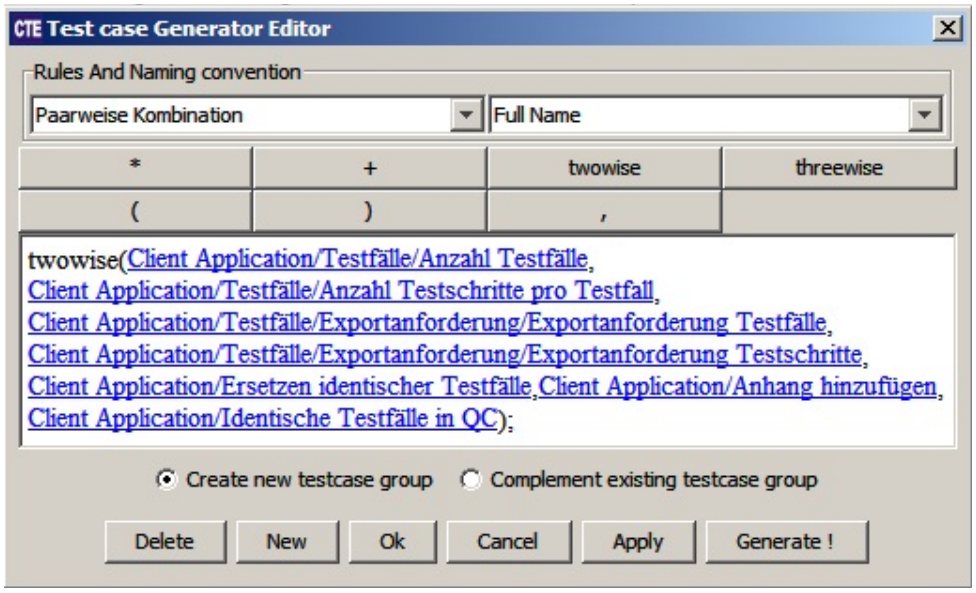

Abb. 27: *Paarweise Kombinationsregel*

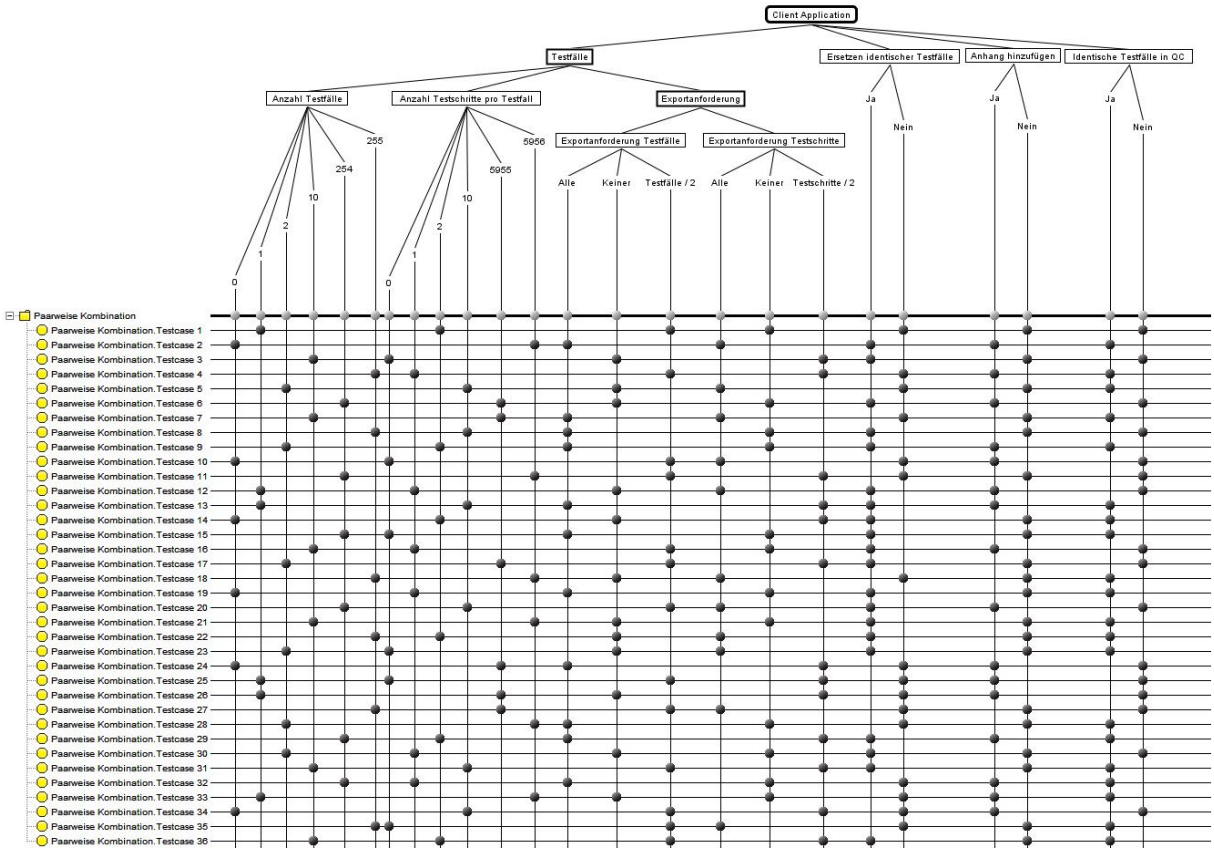

Abb. 28: *Anhand der paarweisen Kombinationsregel generierte Testfälle*

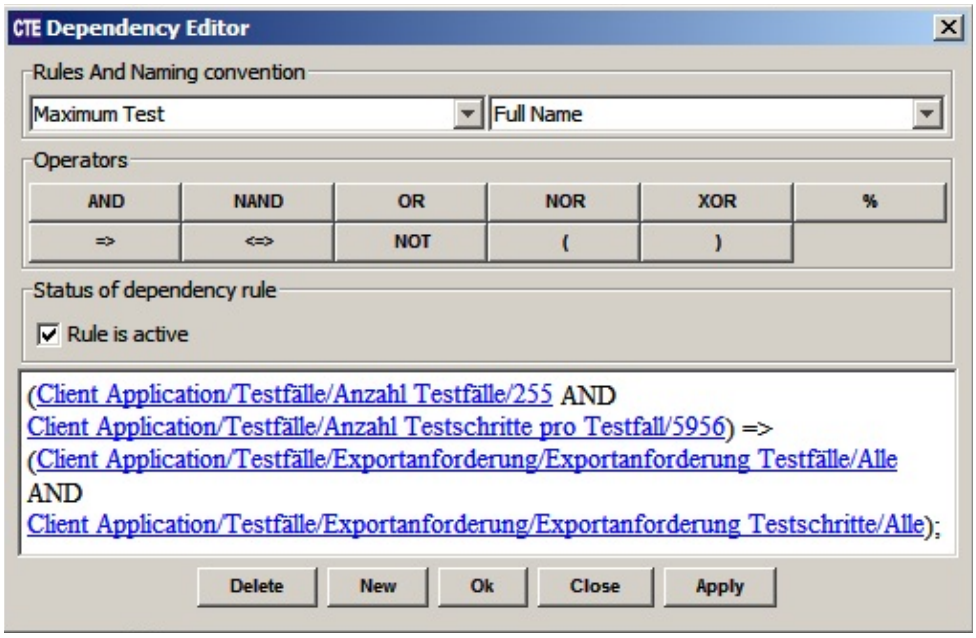

Abb. 29: *Regel: Ein Maximum-Test erfordert Exportanforderung aller Testfälle und -schritte*

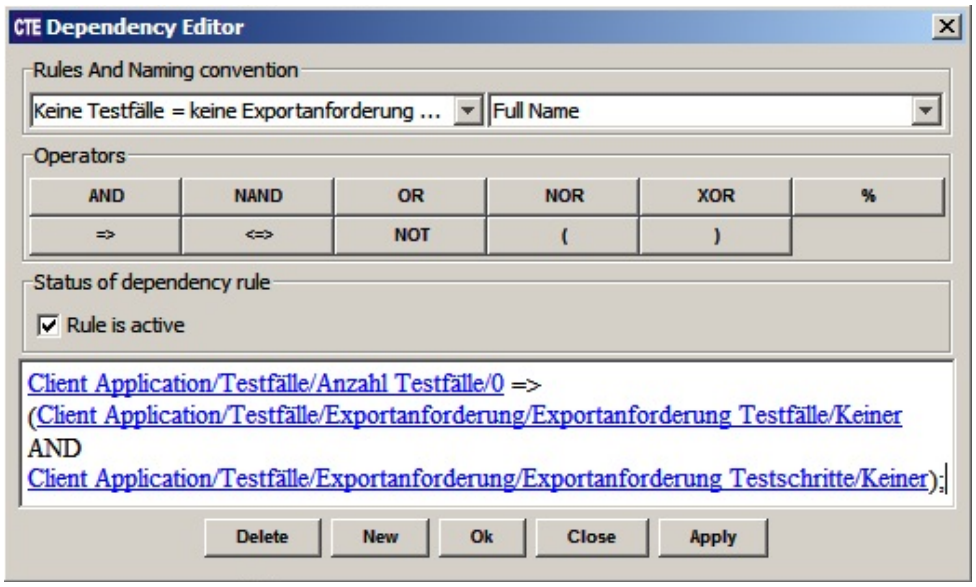

Abb. 30: *Regel: Keine Exportanforderungen bei leerer Testfallspezifikation*

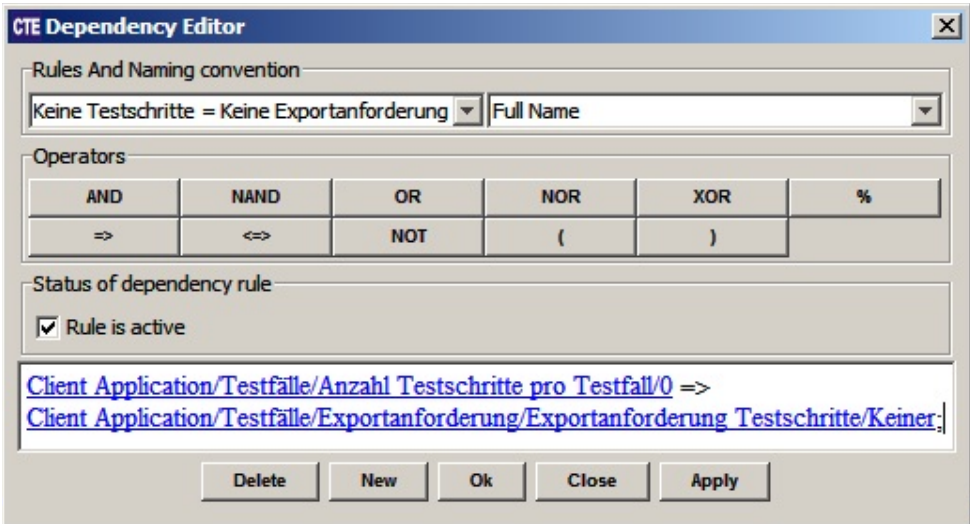

Abb. 31: *Regel: Keine Exportanforderungen von Testschritten, wenn keine vorhanden sind*

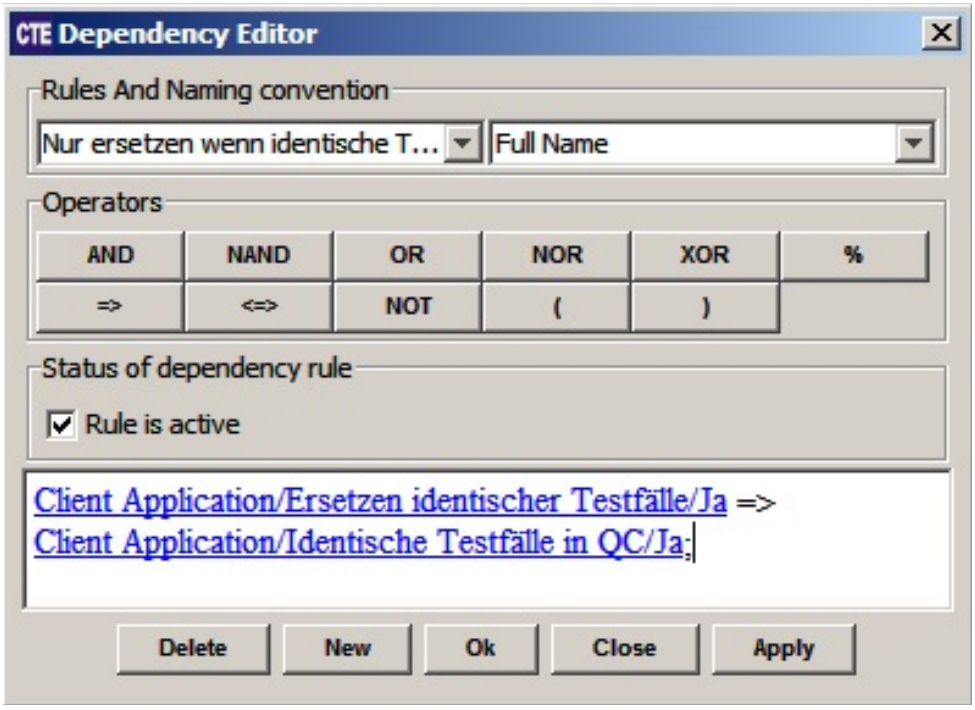

Abb. 32: *Regel: Testfälle nur ersetzen wenn identische Testfälle vorhanden sind*

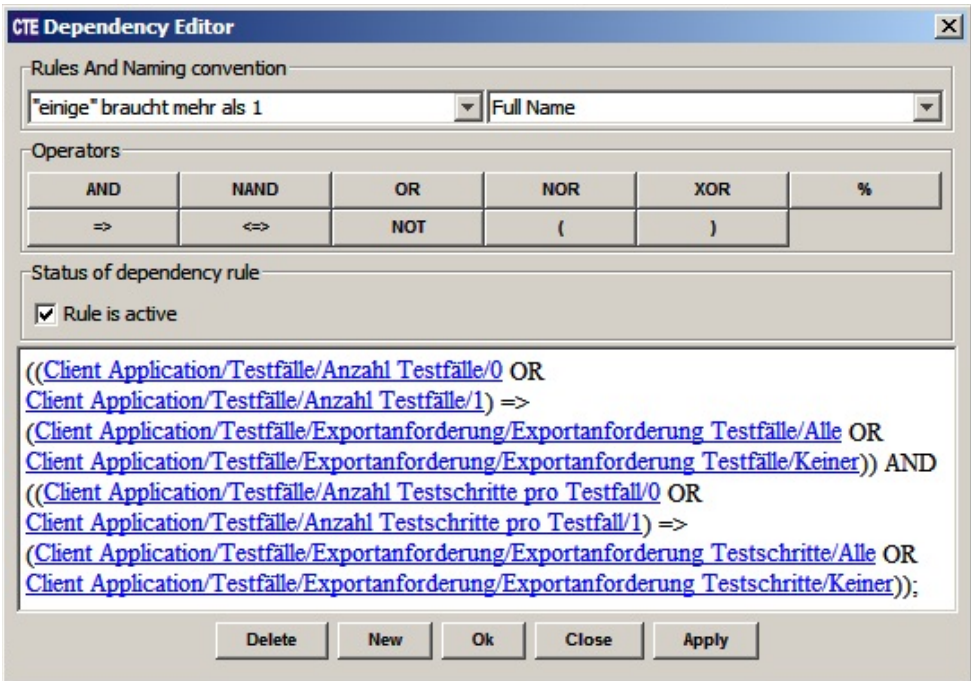

Abb. 33: *Regel: Exportanforderung für die Hälfte aller Testfälle oder -schritte benötigt mehr als einen Testfall oder -schritt*

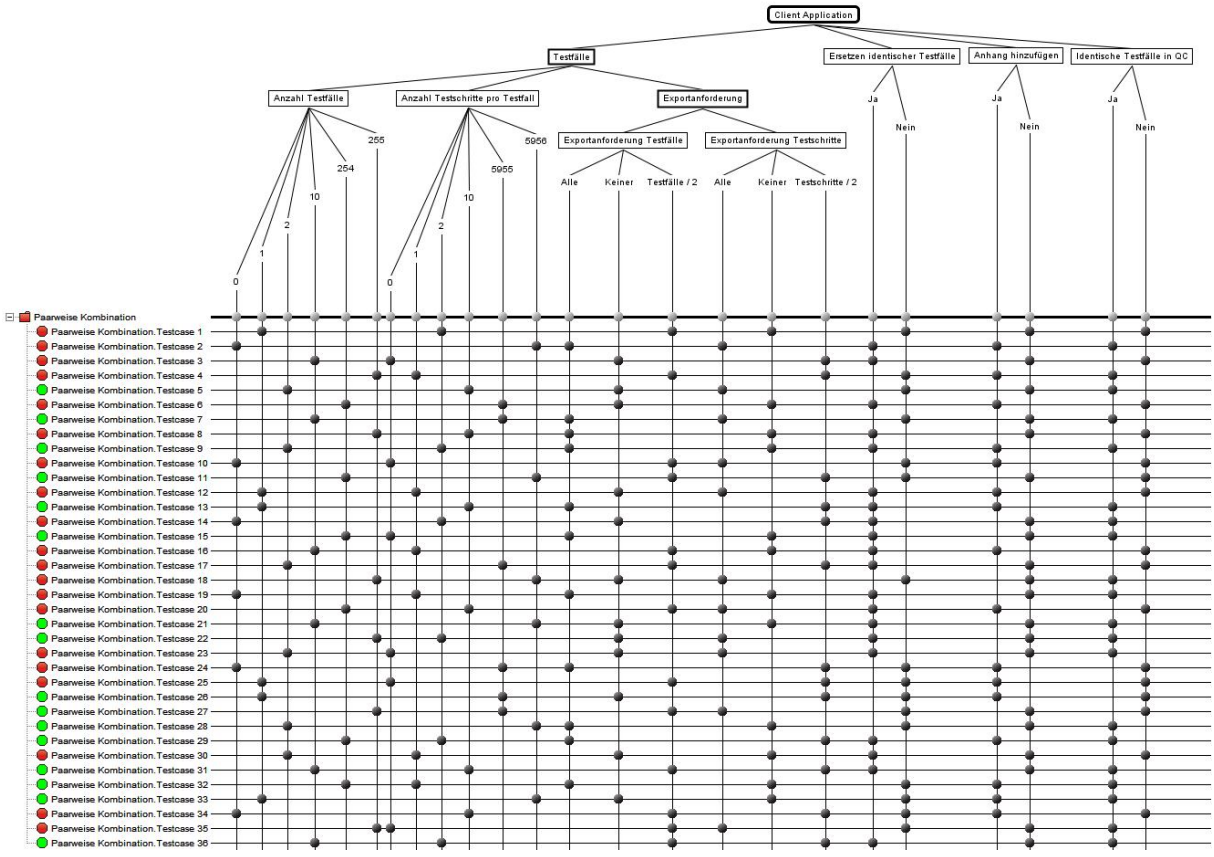

Abb. 34: *Reduzierung auf 16 gültige Testfälle*

Übertragung der Testfälle in die Testfallspezifikation

Um die Testfälle in einen Kontext einzubinden und angemessen zu dokumentieren, werden die 16 verbleibenden Testfälle nun in die Testfallspezifikation übertragen. Jeder dieser Testfälle besteht aus zwei Testschritten. Der erste Testschritt ist bei allen Testfällen gleich und behandelt das Anmelden über den Client beim Quality Center. Hierfür können *URL*, *Benutzername*, *Passwort*, *Domain* und *Projekt* als Parameter angegeben, um die tatsächlichen Eingaben variabel zu halten. Jeder zweite Testschritt beschreibt die Durchführung des Exports der 16 Testfälle zu den jeweiligen Bedingungen.

Exemplarisch folgt die Beschreibung des ersten Testfalls. Hiervon lassen sich alle weiteren Testfallbeschreibungen ableiten:

- **Test Case 1**
  - **Test ID**: kein Eintrag - wird vom Quality Center nach Export vergeben
  - **Export Test Case**: X
  - **Test Name**: CTE Test Case 5
  - **Test Status**: Imported
  - **Objective**: Testen der Exportfunktionalität, wenn kein Testfall eine Exportanforderung gesetzt hat, aber alle Testschritte eine Exportanforderung gesetzt haben
  - **Test Priority**: High
  - **Test Type**: functional
  - **Use-Case Coverage**: Exportieren von Testfällen und Testfallspezifikation
  - **Requirement Coverage**: Testfälle werden nach Quality Center exportiert; Testfallspezifikation kann als Anhang exportiert werden
  - **Version Of Specification**: 1.0
  - **Version Of System Under Test**: 1.0
  - **Comment**: Keine Exportanforderungen bei den Testfällen
- **Step 1**
  - **Export Design Step**: X
  - **Step Name**: Login
  - **Description**: Anmelden beim Quality Center mittels Client
  - **Precondition**:
  - **Postcondition**:
  - **Boundary Condition**: Normaler Netzwerkstatus
  - **Expected Result**: Bildschirm zeigt Export-Maske
  - **Parameters**: URL, Benutzername, Passwort, Domain, Projekt
  - **Parameters Values**:
- **Step 2**
  - **Export Design Step**: X
  - **Step Name**: Export
  - **Description**: Auswahl der Testfallspezifikation, Auswahl des Zielverzeichnisses, aktivieren der Check-Box bei: Anhang hinzufügen
  - **Precondition**: 2 Testfälle mit gleichen Namen im Test Plan und in der Testfallspezifikation vorhanden, die Testfälle in der Testfallspezifikation verfügen jeweils über 10 Testschritte

- **Postcondition**: Keine Einträge von Identifikationsnummern bei den Testfällen in der Testfallspezifikation, Anzeigestatus der Informationselemente beim Client: Number of test cases found = 2, Number of test cases exported = 0
- **Boundary Condition**: Normaler Netzwerkstatus
- **Expected Result**: Kein Testfall ins Test Plan Modul exportiert
- **Parameters**:
- **Parameters Values**: Anzahl Testfälle: 2, Anzahl Testschritte pro Testfall: 10, Exportanforderung Testfälle: Keiner, Exportanforderung Testschritte: Alle, Ersetzen vorhandener Testfälle: Nein, identische Testfälle im Test Plan Modul vorhanden: Ja, Anhang hinzufügen: Ja

Die in die Testfallspezifikation übertragenen Testfälle zeigt ausschnittweise die Abb. 39 im Anhang C.

Exportieren der Testfälle ins Test Plan Modul

Nachdem alle Testfälle in die Testfallspezifikation übertragen sind, können sie nach Quality Center ins Test Plan Modul exportiert werden. Die Abb. 35 und die Abb. 36 zeigen das Resultat nach dem Export.

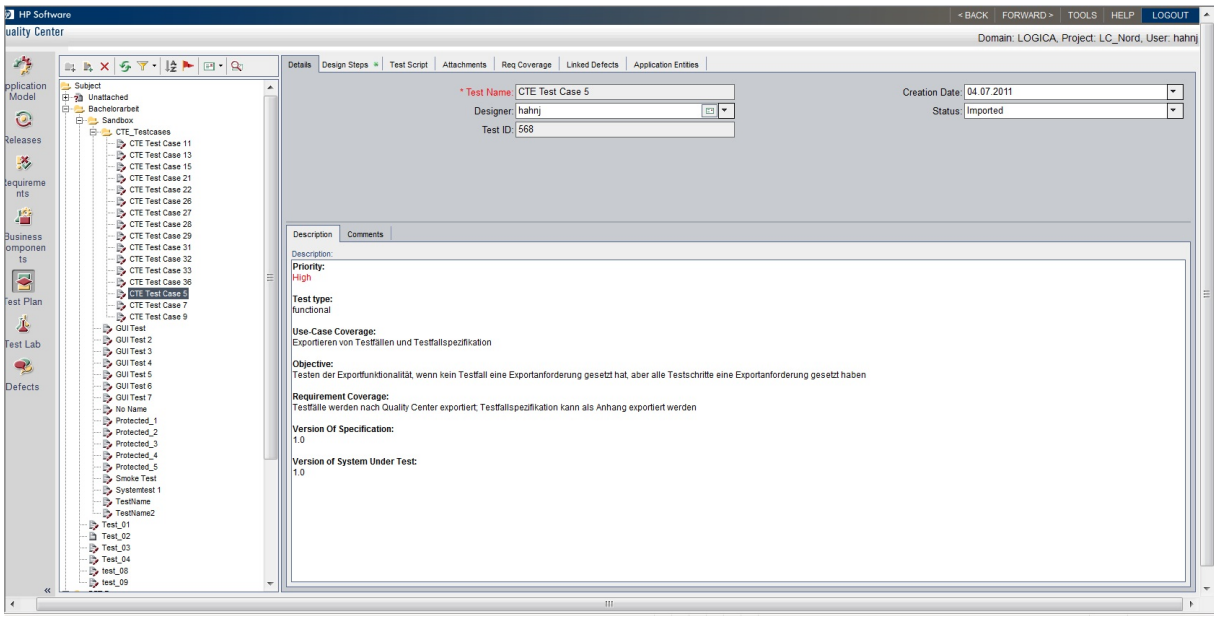

Abb. 35: *Exportierte Testfälle im Test Plan Modul*

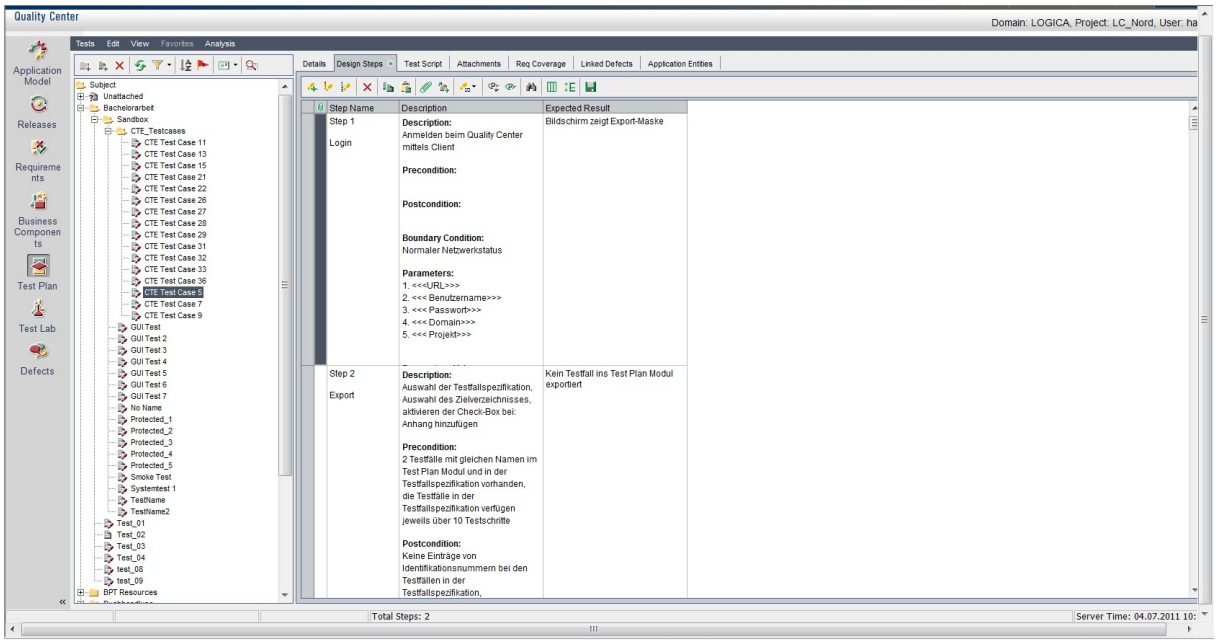

Abb. 36: *Exportierte Design Steps im Test Plan Modul*

8.2 Bewertung des Nutzens

Das Beispiel zeigt, dass die Kombination von Klassifikationsbaummethode, Äquivalenzklassenanalyse und Grenzwertanalyse im Systemtest eine effiziente und dank des CTE/XL Editors, auch eine komfortable Möglichkeit der Testfallermittlung darstellt. Dabei erweisen sich alle drei Techniken in ihrer Kombination nutzenmaximierend. Die Klassifikationsbaummethode mittels CTE/XL unterstützt die Ermittlung zu testender Aspekte und deren Untergliederung sowie die letztliche Testfallgenerierung. Auch die visuelle Darstellung zur Dokumentation von Testfällen ist von großem Nutzen wie die Durchführung des Beispiels und dessen Erläuterung aus Kap. 8.1 unter Beweis stellt. Die Äquivalenzklassenbildung erhöht die Aussagekraft des Klassifikationsbaums durch systematische Einteilung der Klassifikationen in disjunkte Klassen. Die Grenzwertanalyse unterstützt die Wahl wichtiger Repräsentanten dieser Klassen für den Test. Der Aufwand ist gering bei gleichzeitigem Erhalt qualitativ hochwertiger Testfälle.

Als Anmerkung soll erwähnt werden, dass im Zuge der Ausführung der Testfälle, was nicht Bestandteil dieser Arbeit ist, tatsächlich ein Fehler aufgedeckt wurde. Der Testfall 7 der mittels CTE/XL generiert wurde aus Abb. 34 bzw. der Testfall 2 in der Testfallspezifikation prüft den Export von 10 Testfällen mit Exportanforderung aller Testfälle und zusätzlichen Optionen. Bei der Durchführung stellte sich heraus, dass der Client nur drei vorhandene Testfälle in der Testfallspezifikation zählte. Der Fehler konnte schnell eingegrenzt werden auf die in Listing 5 gezeigte fehlerhafte Wertzuweisung der Variable *testNum*, die zur Ermittlung der Anzahl an vorhandenen Testfällen benutzt wird.

```
0  Const TEST_ROW As String = "A3"
   testNum = XLWorksheet.Range(TEST_ROW).End(Excel.XlDirection.xlToLeft).Row
```

Listing 5: *Fehlerhafte Anweisung zur Ermittlung der Anzahl an Testfällen*

Die Konstante *TEST_ROW* gibt die Zeile hinzugefügter Testfälle an. Das Problem war das Schlüsselwort *Row*. Da die Testfälle in der Testfallspezifikation spaltenweise dargestellt werden, muss die Anzahl der dafür verwendeten Spalten ermittelt werden. Die Richtungsangabe *xlToLeft* war ebenfalls falsch. Der stets ermittelte Wert drei ist wahrscheinlich darauf zurückzuführen, dass die Anzahl der Zeilen bis *TEST_ROW* ermittelt wurden. Eine Änderung der Zeile erbrachte die korrekte Ermittlung der Testfälle (vgl. Listing 6).

```
0 Const TEST_ROW As String = "A3"
testNum = XLWorksheet.Range(TEST_ROW).End(Excel.XlDirection.xlToRight).Column
```

Listing 6: *Korrigierte Anweisung zur Ermittlung der Anzahl an Testfällen*

Für die Bewertung des Nutzens bezüglich der entwickelten Testfallspezifikation inklusive Client Applikation ist zu beachten, wie die Durchführung des Beispiels in Kap. 8.1 ohne dieser Werkzeuge verlaufen wäre. Die ermittelten Testfälle wären direkt ins Test Plan Modul übertragen worden. Im Vergleich zur Testfallspezifikation, die mit ihren bereits beschreibenden Attributen einen Bezugsrahmen für die Testfallbeschreibung liefert, unterstützt das Test Plan Modul den Autor für die Beschreibung der Testfälle hauptsächlich nur mit den Attributen *Description* und *Expected Result*. Um eine Testfallbeschreibung herzuleiten, muss ein Schema entworfen werden, was unter gewissem Zeitaufwand geschieht. Somit kann der erhoffte Aspekt der Zeitersparnis aus Kap. 5.2 bestätigt werden. „Damit Dokumente übersichtlich, leicht zu durchsuchen und zu bearbeiten sind, sollten sie nicht in freier Form verfasst werden, sondern einem vorgegebenem Schema folgen“ [26, S.130]. Hinzu kommt, dass ad hoc entworfene Patterns dazu neigen, nicht immer gleich zu sein und evtl. wichtige Attribute zu vernachlässigen bzw. auszulassen. So aber „[...] garantieren die vorgegebenen Strukturen eine gewisse Vollständigkeit, denn alle wichtigen Themenkomplexe sind in der Standardstruktur enthalten und können nicht vergessen werden.“ [26, S.129]. Alle 16 Testfälle des Beispiels sind in ihrer systematischen Darstellung in der Testfallspezifikation und im Test Plan Modul gleich und decken alle wichtigen Attribute zur Testfallbeschreibung ab. Die einheitliche Darstellung der Testfälle unterstützt auch die Wartbarkeit von Testfällen. Werden Testfälle gewartet, z. B. für Regressionstests aufgrund eines veränderten Systems, hilft eine einheitliche und systematische Darstellung, Testfälle nach längerer Zeit nachvollziehen zu können. „Dokumente sind zweckmäßig strukturiert und geordnet, nicht individuell vom Verfasser, sondern durch Richtlinien, Vorgaben und Templates. Dann kann sich jeder schnell in eine fremde Dokumentation einarbeiten[...]“ [26, S.129]. Dies dürfte auch für Tester gelten, die Testfälle anderer Tester nachvollziehen müssen. Werden Testfälle lückenlos dokumentiert, kann sich dies auch auf zukünftige Projekte positiv auswirken. Zuser macht diesbezüglich folgende Aussage: „Ordentliches Dokumentieren von Tests. Fehler und das Erkennen der Ursachen bedeutet oft, dass bei kommenden Projekten diese vermieden oder zumindest leichter aufgespürt werden können.“ [50, S.220].

Die Erstellung der Testfälle konnte ohne Verbindung zum Quality Center erfolgen und benötigte bis zum Zeitpunkt des Exports keine Lizenz.

Ferner stellt eine detaillierte Testfallspezifikation grundsätzlich eine gute Basis für weiterführende Testautomatisierung dar. Sofern nötige Schnittstellen vorhanden sind, wäre eine Verwendung der Testfallspezifikation auch für andere Testwerkzeuge denkbar.

> “A detailed manual script contains exactly what will be input to the software under test and exactly what is expected as a test outcome for that input. All the tester has to do is what the script says. This is the level that is the closest to what the test tool does, so automation is easiest in many ways from this level.“ [13, S.39]

Resümierend ist festzustellen, dass, abgesehen von einzusparenden Lizenzen, die o. g. Nutzen auch ohne das entwickelte Werkzeug erbracht werden können, die Verwendung des Werkzeugs aber hierfür eine Garantie gewährt.

9 Zusammenfassung und Ausblick

Diese Arbeit gibt eine Einschätzung zur Unterstützung der Auswahl von spezifikationsorientierten Testverfahren bezüglich deren Anwendbarkeit im Systemtest. Darüber hinaus wird eine Lösung für eine systematische Testfall- und Testablaufspezifikation vorgestellt, die den Export von Testfällen ins Test Plan Modul von HP Quality Center impliziert.

Die Evaluierung der Testverfahren zeigt die Schwierigkeit einer allgemeinen Einschätzung. Alle in Kap. 3 vorgestellten Testverfahren ermöglichen eine Anwendung im Systemtest. Eine Entscheidung über ein zu verwendendes Verfahren zu treffen ist letzendlich immer abhängig vom Testobjekt und seiner Spezifikationsform. Daher ist das Ergebnis der Evaluierung eher als Abwägung von Vor- und Nachteilen zu betrachten. Unabhängig von der Evaluierung wäre für viele Systeme keines der vorgestellten Verfahren die optimale Lösung. So ist z. B. bei eingebetteten Systemen meistens ein zustandsbasiertes Testverfahren vorzuziehen. Auch sollte nicht nur ein einzelnes Verfahren im Systemtest zum Einsatz kommen. Es ist entweder, wie die Evaluierung zeigt, eine Kombination der Verfahren oder der Gebrauch verschiedener unabhängiger Verfahren anzuraten.

Bei der Entwicklung des Tools wurde deutlich, dass der gegenwärtige Marktführer im Bereich Testmanagement-Tools (HP Quality Center) hinsichtlich einer gut dokumentierten und systematischen Erfassung von Testfällen noch deutliche Schwächen aufweist. Auch die Möglichkeiten des Excel Add-in weisen diesbezüglich keine Verbesserung auf. Die Nachfrage eines solchen Werkzeugs ist offenbar sehr hoch, wie diverse Foren der Community zeigen. Die Beschreibung der OTA-API ist grundsätzlich zufriedenstellend, könnte aber mehr Beispiele liefern. Auch dies zeigt sich an den zahlreichen Fragen in den entsprechenden Foren. Während der Entwicklung der Client Applikation kamen oftmals Zweifel auf, ob die Art der Umsetzung richtig ist. Dabei war nicht immer festzustellen, ob die Beschreibung der Schnittstelle lückenhaft ist, oder die Schnittstelle selbst keine elegantere Lösung zulässt. Sollten zukünftige Versionen von HP Quality Center mit einer nicht abwärtskompatiblen Veränderung der OTA-API einhergehen, kann dies Auswirkungen auf die Funktionalität des Clients haben. Diese Abhängigkeit ist unumgänglich.

Es wurde auch in Erwägung gezogen Testfallergebnisse aus Quality Center zurück in die Testfallspezifikation zu schreiben. Dies wäre problemlos realisierbar, da die OTA-API auch hierfür entsprechende Möglichkeiten bereitstellt. Dadurch wären aber Testfallerstellung und Testdurchführung nicht mehr strikt voneinander getrennt. Das Quality Center trennt beide Aktivitäten durch das Test Plan und Test Lab Modul. Die Testfallspezifikation unterstützt durch einen systematischen Entwurf von Testfällen nur das Test Plan Modul. Nach Übertragung der Testfälle stehen sie dort zur Verfügung. Alle folgenden Aktivitäten sollen dann nur noch im Quality Center stattfinden. Ein Zurückschreiben der Testfallergebnisse würde Redundanz bezüglich der Testfallergebnisse zwischen Test Lab Modul und Testfallspezifikation fördern und nicht mit der für die Testfallspezifikation und dem Test Plan Modul vorgesehenen Korrelation übereinstimmen.

Auch über einen möglichen Import von Testfällen aus dem Test Plan Modul in die Testfallspezifikation wurde disktutiert. Testfälle, die zuvor ohne die Testfallspezifikation entworfen wurden, könnten dadurch die Struktur der Testfallspezifikation erhalten. Dies ist jedoch nicht möglich. Zwar können Testfälle, ebenso wie die o. g. Testfallergebnisse, unkompliziert mithilfe der OTA-API zurückgeschrieben werden, das Problem ist aber die frei entworfene Testfallbeschreibung anderer Autoren. Die Abbildung eines freien Entwurfs auf die Testfallspezifiktion würde ein Verständnis der Semantik voraussetzen, um eine allgemeine Testfallbeschreibung zu untergliedern und auf die jeweiligen Attribute der Testfallspezifikation zu verteilen.

Eine andere interessante Erweiterung des Werkzeugs wäre der Import vom CTE/XL generier-

ter Testfälle in die Testfallspezifikation. Der CTE/XL bietet den Export von Testfällen u. a. in Excel oder XML (*Extensible Markup Language*) Formaten an. Beide Formate würden sich eignen, um sie in die Testfallspezifikation zu importieren. Zwar müssten zahlreiche Attribute noch nachträglich manuell ausgefüllt werden, die der CTE/XL nicht vorsieht, aber vor allem bei einer hohen Anzahl an Testfällen wäre die Zeitersparnis gegenüber einer vollständigen manuellen Erfassung beträchtlich. Dies könnte sowohl über die Testfallspezifikation als auch den Client realisiert werden, wobei eine Implementierung in der Testfallspezifikation sinvoller wäre, da dies nicht der funktionalen Beschaffenheit des Clients entspricht und stets eine unnötige Anmeldung beim Quality Center erfolgen müsste.

Literaturverzeichnis

[1] WiBe (Veranst.): 2011. – URL http://www.wibe.de/infothek/themen/wibeiso9126/wibeiso9126.html. – Abruf: 18. Juli 2011

[2] ISTQB - International Software Testing Qualifications Board (Veranst.): 2011. – URL http://istqb.org/display/ISTQB/Home. – Abruf: 18. Juli 2011

[3] ARMBRUST, Ove ; OCHS, Michael ; SNOEK, Björn: Stand der Praxis von Software-Tests und deren Automatisierung / Fraunhofer IESE. Kaiserslautern, September 2004. – Publikation. – URL http://www.ove-armbrust.de/downloads/Armbrust-RLRLP-SoP-Testing.pdf. Abruf: 18. Juli 2011

[4] BATH, Graham ; MCKAY, Judy: *Test Analyst und Technical Test Analyst*. 1. Heidelberg : dpunkt.verlag, 2010. – ISBN 978-3-89864-591-1

[5] BERNER ; GMBH, Mattner S.: CTE XL. (2011). – URL http://www.berner-mattner.com/de/berner-mattner-home/produkte/cte/download.html

[6] BOARD, German T.: *Basiswissen Softwaretest - Certified Tester*. 2007

[7] BRÜGGE, Bernd ; DUTOIT, Allen H.: *Objektorientierte Softwaretechnik*. München : Pearson Studium, 2004. – ISBN 3-8273-7082-5

[8] BUTH, Bettina: *Software Engineering 2*, Hochschule für Angewandte Wissenschaften Hamburg, Vorlesungsunterlagen, 2008

[9] CORPORATION, Mercury I.: HP Quality Center - Administrator's Guide. (2007). – URL http://www.genilogix.com/downloads/docs/qc92/AdminGuide_qc92.pdf. – Abruf: 18. Juli 2011

[10] CORPORATION, Mercury I.: HP Quality Center - Tutorial. (2007). – URL http://shrivv.com/Technicle/QA_HP_Tutorial.pdf. – Abruf: 18. Juli 2011

[11] CORPORATION, Mercury I.: Microsoft Excel Add-in Guide. (2008). – URL http://updates.merc-int.com/qualitycenter/qc90/msoffice/msexcel/qc_9_2/QCMSExcelAddin.pdf. – Abruf: 18. Juli 2011

[12] CORPORATION, Microsoft: Visual Basic Express 2010. (2010). – URL http://www.microsoft.com/germany/express/download/

[13] FEWSTER, Mark ; GRAHAM, Dorothy: *Software Test Automation*. Edinburgh, Great Britain : Addison-Wesley Verlag, 1999. – ISBN 978-0-201-33140-0

[14] FOWLER, Martin: *UML Distilled*. 3. Boston : Pearson Education, 2004. – ISBN 0-321-19368-7

[15] GERSHOVICH, Igor: Extending QC with Open Test Architecture (OTA) API. (2008). – URL http://connectedtesting.com/Presentations/QCOTApresentation.ppt. – Abruf: 18. Juli 2011

[16] GROUP, Standish: CHAOS Summary 2009. (2009). – URL http://www1.standishgroup.com/newsroom/chaos_2009.php. – Abruf: 18. Juli 2011

[17] HAMBURG, Dr. M. ; HEHN, Dr. U.: ISTQB/GTB Standardglossar der Testbegriffe. (2010). – URL http://www.german-testing-board.info/downloads/pdf/CT_Glossar_DE_EN_V21.pdf. – Abruf: 18. Juli 2011

[18] HANNOVER, Universität: *Visual Basic 6.0*. September 2000

[19] HARREL, Jeremy M.: Orthogonal Array Testing Strategy (OATS) Technique. (2001). – URL http://arf.iyte.edu.tr/course/CENG316/files/OATS.pdf. – Abruf: 18. Juli 2011

[20] HEYDE, Ralf: Protokoll der Vorlesung Psychologische Aspekte der Software Ergonomie / Technische Universität Berlin. URL http://apa.cs.tu-berlin.de/LeiLehreSS08/LV7.pdf, Juni 2008. – Protokoll. Abruf: 18. Juli 2011

[21] LEHMANN, Eckard ; WEGENER, Joachim: Test Case Design by Means of the CTE XL. (2000). – URL http://www.systematic-testing.com/documents/eurostar2000.pdf. – Abruf: 18. Juli 2011

[22] LIGGESMEYER, Peter: *Software-Qualität - Testen, Analysieren und Verifizieren von Software*. 1. Heidelberg : Spektrum Akademischer Verlag GmbH, 2002. – ISBN 3-8274-1118-1

[23] LIM, Meike ; SADEGHIPOUR, Sadegh: Werkzeugunterstützte Verknüpfung von Anforderungen und Tests - Voraussetzung für eine systematische Qualitätssicherung / Fachbereich Softwaretechnik der Gesellschaft für Informatik. URL http://pi.informatik.uni-siegen.de/stt/28_3/01_Fachgruppenberichte/TAV/09_TAV27P9Sadeghipour.pdf, Juni 2008. – Bericht. Arbuf: 18. Juli 2011

[24] LOGICA: *Unternehmensprofil 2010*. 2010. – URL http://www.logica.de/we-are-logica/about-logica/~/media/5BE8D9FE48A4497D959288FD27F7D7B9.ashx. – Abruf: 18. Juli 2011

[25] LÜTZKENDORF, Stefan ; BOTHE, Klaus: Attributierte Klassifikationsbäume zur Testdatenbestimmung / Fachbereich Softwaretechnik der Gesellschaft für Informatik. URL http://pi.informatik.uni-siegen.de/stt/23_1/01_Fachgruppenberichte/FG217/05_Bothe1.ps, November 2003. – Bericht. Abruf: 18. Juli 2011

[26] LUDEWIG, Jochen ; LICHTER, Horst: Dokumentation in der Softwareentwicklung - Gut festhalten. In: *iX kompakt 1/2011 - Softwarequalität* 1 (2011), Januar, S. 126 – 130

[27] LUDEWIG, Jochen ; LICHTER, Horst: Mit Anspruch - Softwarequalität und wie man sie erkennt. In: *iX kompakt 1/2011 - Softwarequalität* 1 (2011), Januar, S. 112 – 114

[28] MARTIN, Robert C.: The Test Bus Imperative: Architectures That Support Automated Acceptance Testing. In: *IEEE Software* (2005), S. 65 – 67

[29] Mercury Interactive Corporation (Veranst.): *Mercury Quality Center Open Test Architecture API Reference*. 2007

[30] MONTENEGRO, Prof. Dr. S. ; HOLZKY, Felix: BOSS/EVERCONTROL OS/Middleware Targes ultra high Dependability. (2005). – URL http://www.montenegros.de/sergio/public/montenegro_boss_control.pdf. – Abruf: 18. Juli 2011

[31] NARDELLI, Nicolas: *Entwicklung eines Testkonzepts für parametrisierbare Simulationsmodelle mechanischer, hydraulischer und regelungstechnischer Systeme*, Universität Stuttgart, Diplomarbeit, 2000. – URL http://elib.uni-stuttgart.de/opus/volltexte/2000/711/pdf/DIP-1856.pdf. – Abruf: 18. Juli 2011

[32] SCHNEIDER, Kurt: *Abenteuer Software Qualität - Grundlagen und Verfahren für Qualitätssicherung und Qualitätsmanagement*. 1. Heidelberg : dpunkt.verlag, 2007. – ISBN 978-3-89864-472-3

[33] SCHNEIDER, Kurt: Grundkonzepte zum Bau hochwertiger Softwareprodukte. In: *iX kompakt 1/2011 - Softwarequalität* 1 (2011), Januar, S. 120

[34] SCHOTANUS, Chris C.: *TestFrame*. 1. Meppel, Niederlande : Springer Verlag, 2009. – ISBN 978-3-642-00821-4

[35] SNEED, Harry M. ; BAUMGARTNER, Manfred ; SEIDL, Richard: *Der Systemtest - Von den Anforderungen zum Qualitätsnachweis*. 2. München : Hanser Verlag, 2009. – ISBN 978-3-446-41708-3

[36] SOMMERVILLE, Ian: *Software Engineering*. 5. Harlow, England : Addison-Wesley Verlag, 1998. – ISBN 0-201-42765-6

[37] SPILLNER, Andreas ; LINZ, Tilo: *Basiswissen Softwaretest*. 3. Heidelberg : dpunkt.verlag, 2005. – ISBN 3-89864-358-1

[38] SPILLNER, Andreas ; LINZ, Tilo: Immer feiner - Testen im Lebenszyklus. In: *iX kompakt 1/2011 - Softwarequalität* 1 (2011), Januar, S. 65 – 71

[39] SPILLNER, Andreas ; ROSSNER, Thomas ; WINTER, Mario ; LINZ, Tilo: *Praxiswissen Softwaretest - Testmanagement*. 2. Heidelberg : dpunkt.verlag, 2008. – ISBN 978-3-89864-557-7

[40] SQS: Anforderungsmanagement, Potenziale und Trends. (2006). – URL http://www.sqs-group.com/de/group/Download/SQP_Anforderungsmanagement_DE.pdf. – Abruf: 18. Juli 2011

[41] STANDARDS, National I. of ; TECHNOLOGY: The Economic Impacts of Inadequate Infrastructure for Software Testing. (2002). – URL http://www.nist.gov/director/planning/upload/report02-3.pdf. – Abruf: 18. Juli 2011

[42] THIEMANN, Uwe ; LÖFFELMANN, Klaus: *Visual Basic 6 - Das Handbuch*. 1. Microsoft Press Deutschland, 1998. – ISBN 3-86063-137-3

[43] TIEDE, Rebecca: *Testentwurf in komplexen softwareintensiven Systemen mit der Klassifikationsbaummethode*, Freie Universität Berlin, Vortrag Diplomarbeit, Oktober 2007. – URL http://www.inf.fu-berlin.de/inst/ag-se/teaching/S-BSE/090_Tiede-testentwurf-klassifbaummethode.pdf. – Abruf: 18. Juli 2011

[44] UEBERHORST, Stefan: Umdenken bei Softwaretests. In: *Computerwoche* (2009). – URL http://www.computerwoche.de/software/software-infrastruktur/1883648/. – Abruf: 18. Juli 2011

[45] VIGENSCHOW, Uwe: *Testen von Software und Embedded Systems - Professionelles Vorgehen mit modellbasierten und objektorientierten Ansätzen*. 2. Heidelberg : dpunkt.verlag, 2010. – ISBN 978-3-89864-638-3

[46] VOSSEBERG, Karin: *Software Engineering 1*, Hochschule für Angewandte Wissenschaften Hamburg, Vorlesungsunterlagen, 2007

[47] WEGENER, Joachim ; GROCHTMANN, Matthias: Werkzeugunterstützte Testfallermittlung für den funktionalen Test mit dem Klassifikationsbaum-Editor CTE. (1993). – URL http://www.systematic-testing.com/documents/swt1993.pdf. – Abruf: 18. Juli 2011

[48] WESSEL, Ivo: *GUI-Design - Richtlinien zur Gestaltung ergonomischer Windows-Applikationen*. 1. Carl Hanser Verlag, 1998. – ISBN 3-446-19389-8

[49] WRUBEL, Boris: Einsatz der Klassifikationsbaummethode bei einem österrichischen Baumarktkonzern / INSO - Industrial Software. URL http://www.inso.tuwien.ac.at/uploads/media/INSO_Softwaretesten_Klassifikationsbaummethode_final.pdf. – Gastvortrag. Abruf: 18. Juli 2011

[50] ZUSER, Wolfgang ; BIFFL, Stefan ; GRECHENIG, Thomas ; KÖHLE, Monika: *Software-Engineering mit UML und dem Unified Process*. 1. München : Pearson Education, 2001. – ISBN 3-8273-7027-2

Abbildungsverzeichnis

Tabellenverzeichnis

A Sequenzdiagramme

A.1 Anmeldevorgang

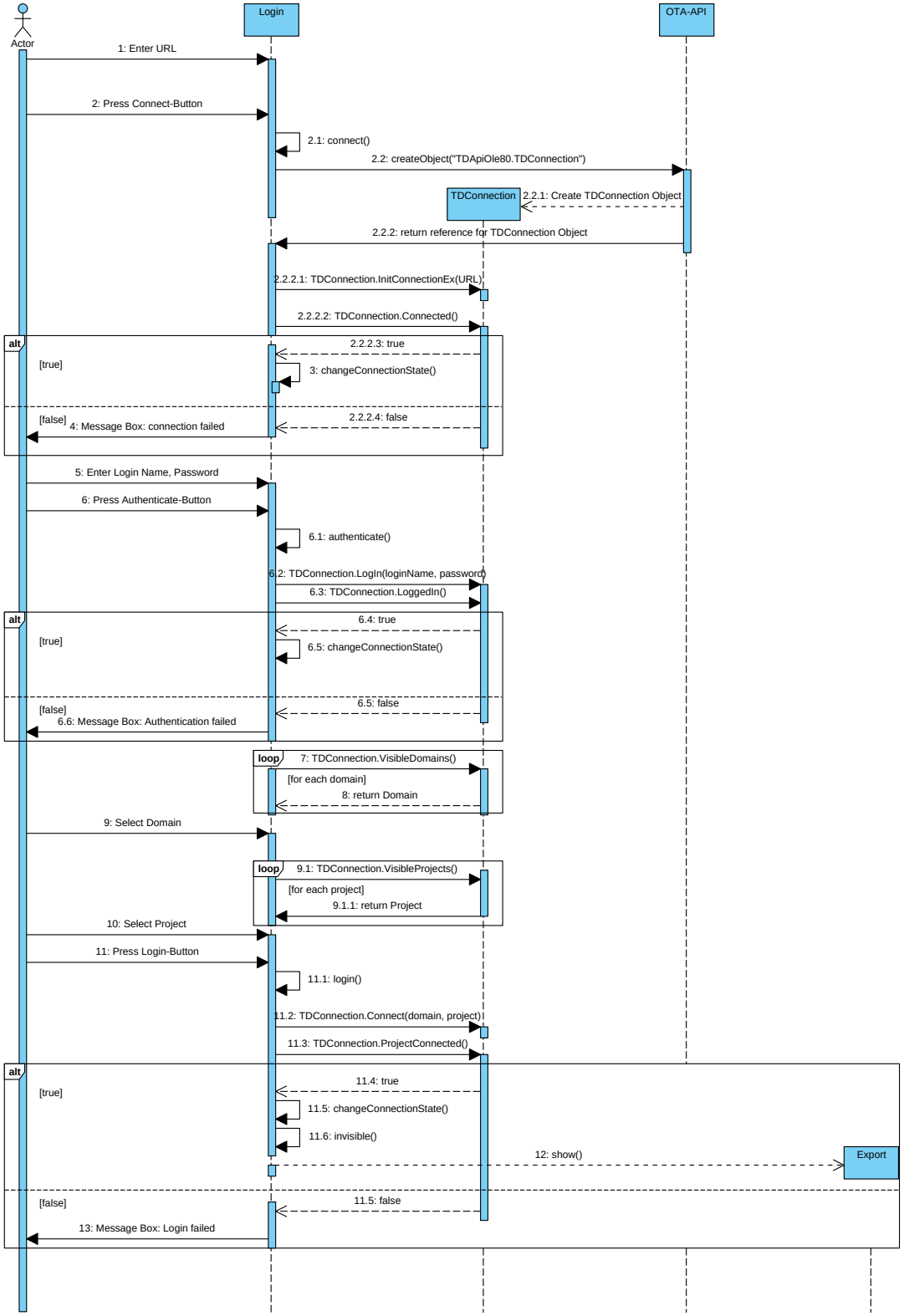

Abb. 37: *Ablauf eines Anmeldevorgangs*

A.2 Export von Testfällen

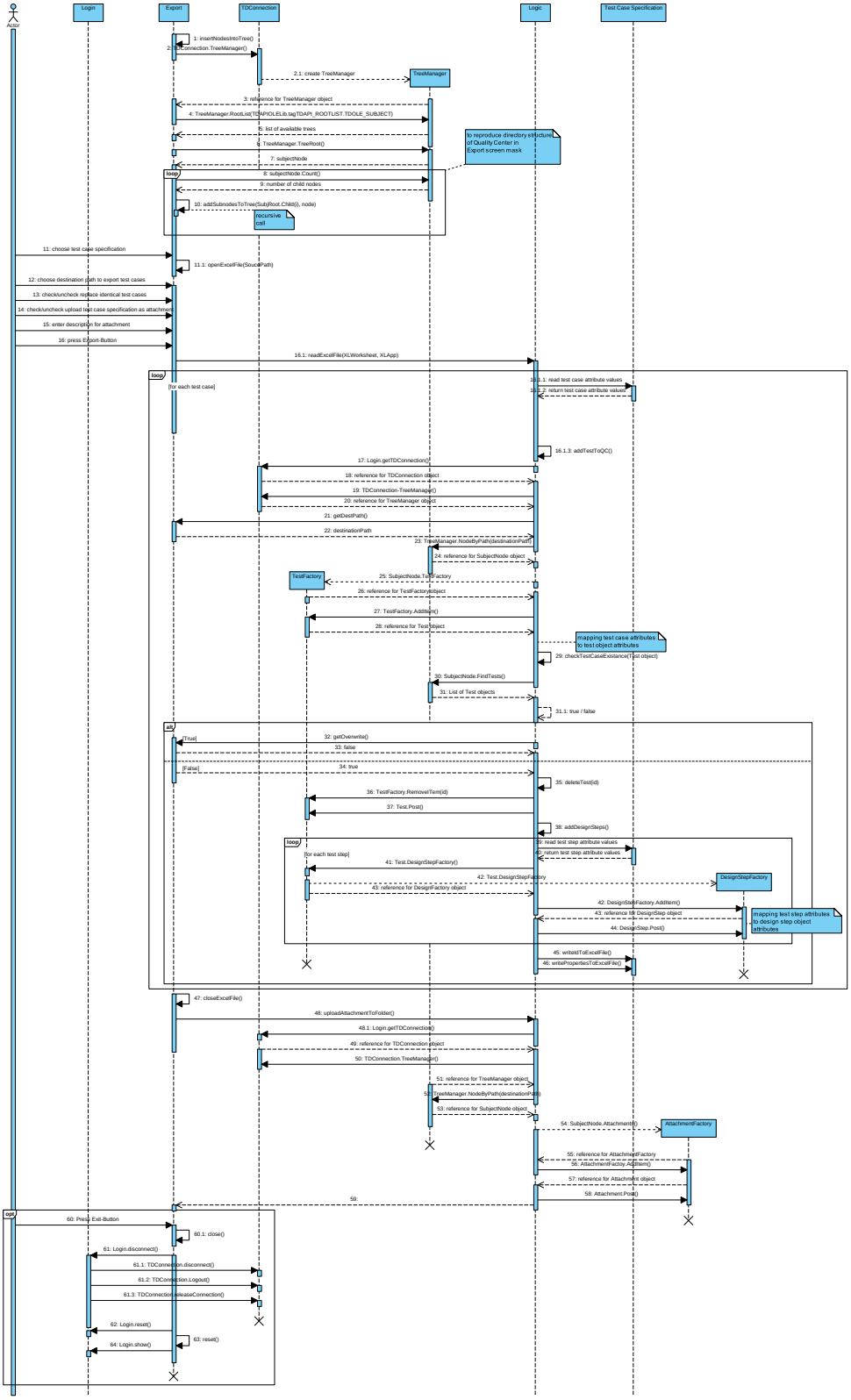

Abb. 38: *Ablauf eines Exportvorgangs*

B Relevante Implementierungen des Clients

B.1 Connect

```
'*************************************************************************
'* Name: Connect                                                         *
'* Type: Procedure                                                       *
'* Parameters: void                                                      *
'* Description: Initializes the connection to HP Quality Center and sets the URL  *
'*              for the connection.txt file.                             *
'*************************************************************************
Private Sub Connect()
    Const Connection As String = "TDApiOle80.TDConnection"
    Me.url = Me.TextBox5.Text.Trim

    Try
        ' Return the TDConnection object
        QCConnection = CreateObject(Connection)
        QCConnection.InitConnectionEx(url)

        If Not QCConnection.Connected Then
            MsgBox("Could not connect to server", vbCritical, "Error Message!")
            Exit Sub
        Else
            changeConnectionState(enmState.connected)
        End If
    Catch ex As Exception
        MsgBox("Could not connect to server, please check URL and try again", vbCritical, "Error Message!")
    End Try
End Sub
```

Listing 7: *Verbindungsaufbau zum Quality Center Server*

B.2 Authenticate

```
'*************************************************************************
'* Name: authenticate                                                    *
'* Type: Procedure                                                       *
'* Parameters: void                                                      *
'* Description: Authenticates a user by its name and password and sets the user  *
'*              for the connection.txt file                              *
'*************************************************************************
Private Sub authenticate()
    Dim Password As String

    user = TextBox1.Text.Trim
    Password = TextBox2.Text.Trim

    Try
        QCConnection.Login(Me.user, Password)

        If Not QCConnection.LoggedIn Then
            MsgBox("Could not log in", vbCritical, "Error Message!")
            Exit Sub
        Else
            changeConnectionState(enmState.authenticated)
        End If

        For Each element In QCConnection.VisibleDomains
            Debug.Print("Domains: " & element)
            ComboBox1.Items.Add(element.ToString)
        Next
    Catch ex As Exception
        MsgBox("Could not authenticate user, please verify Name and Password", vbCritical, "Error Message!")
    End Try
End Sub
```

Listing 8: *Authentifizierung am Quality Center Server*

B.3 Login

```
'*****************************************************************************
'* Name: Login                                                               *
'* Type: Procedure                                                           *
'* Parameters: void                                                          *
'* Description: Establishes a new project session for the logged-in user, starts *
'*              the timer and writes URL and user name to connection.txt file *
'*****************************************************************************
Private Sub Login()
    writeLogFile()
    Me.domain = ComboBox1.SelectedItem.ToString
    Me.project = ComboBox2.SelectedItem.ToString

    Try
        QCConnection.Connect(Me.domain, Me.project)

        If Not QCConnection.ProjectConnected Then
            MsgBox("Could not connect to project", vbCritical, "Error Message!")
            Exit Sub
        Else
            changeConnectionState(enmState.loggedIn)
            initializeTimer()
        End If
    Catch ex As Exception
        MsgBox("Could not login to Domain and Project", vbCritical, "Error Message!")
    End Try
End Sub
```

Listing 9: *Einloggen am Quality Center Server*

B.4 addTestsToQC

```
'*****************************************************************************
'* Name: addTestsToQC                                                        *
'* Type: Procedure                                                           *
'* Parameters: In -> Name of the test as string, name of the design step as string, *
'*                   description for the test as string, expected result as string *
'*                                   description for the attachment.         *
'* Description: Creates a test object for Quality Center with test name and design *
'*              steps ("Step Name", "Description" and "Expected Result" and adds it *
'*              to "Test Plan" in Quality Center.                            *
'*              If chosen by user tests with identical test names            *
'*              (not allowed in Quality Center) will be deleted and substituted *
'*              for the current test object.                                 *
'* Side effects: increases number of imported tests and updates progress bar in GUI.*
'*****************************************************************************
Public Function addTestsToQC(ByVal testName As String, ByVal comment As String, ByVal description As String, ByVal status
    As String) As Boolean
    Const NEGATIVE As Integer = -1
        'Test Type is always "Manual":
    Const TEST_TYPE As String = "MANUAL"
    Dim QCConnection As TDAPIOLELib.TDConnection
    Dim folder As TDAPIOLELib.SubjectNode
    Dim treeM As TDAPIOLELib.TreeManager
    Dim id As VariantType = NEGATIVE
    Dim addSteps As Boolean = False

    QCConnection = Login.getQCConnection
    treeM = QCConnection.TreeManager
    folder = treeM.NodeByPath(Form2.getDestPath)

    testF = folder.TestFactory
    test = testF.AddItem(DBNull.Value)
    test.Name = testName
    test.Type = TEST_TYPE
    test.Field("TS_STATUS") = status
    test.Field("TS_DESCRIPTION") = description
    test.Field("TS_DEV_COMMENTS") = comment

    'Does test case already exist?
    id = Me.checkTestCaseExistence(test)
    addSteps = True
    'if test case exists then overwrite?
    If id > -1 Then
        If Form2.getOverwrite Then
            Me.deleteTest(id)
        Else
            addSteps = False
            addTestsToQC = addSteps
            Me.qcID = NEGATIVE
            Exit Function
        End If
    End If

    test.Post()
    numOfImportedTests += 1
    'Refresh Gui elements:
    Form2.Label4.Text = numOfImportedTests
    Form2.ProgressBar1.Value += 1

    Me.qcID = Me.checkTestCaseExistence(test)
    addTestsToQC = addSteps
End Function
```

Listing 10: *Hinzufügen von Testfällen*

B.5 addDesignStep

```
'*******************************************************************************
'* Name: addDesignStep                                                         *
'* Type: Procedure                                                             *
'* Parameters: In -> last row of last design step in corresponding test case   *
'*                   column,                                                   *
'*                   column of corresponding test case,                        *
'*                   Excel worksheet object (actice sheet of an already opened *
'*                   Excel workbook)                                           *
'* Description: Reads a design step from Excel sheet and adds this design step to *
'*              the last added test case in Quality Center.                    *
'*******************************************************************************
Private Sub addDesignStep(ByVal lastRow As Long, ByVal column As Long, ByRef XLWorksheet As Excel.Worksheet)
    Const PRE_COND_TEXT As String = "Precondition"
    Const POST_COND_TEXT As String = "Postcondition"
    Const EXP_RESULT As String = "Expected Result"
    Const PARAMETERS As String = "Parameters"
    Const PARAM_VALUES As String = "Parameters Values"
    Const STEP_NAME As String = "Step Name"
    Const STEP_NUM As String = "Step "
    Const EXPORT_DESIGN_STEP As String = "Export Design Step"
    Const STEP_PROPERTIES As String = "Customized Step Attributes"
    Const BOUNDARY_COND As String = "Boundary Condition"
    Const DESCRIPTION_TEXT As String = "Description"
    Const START_ROW As Long = 16
    Const FIRST_COLUMN As Integer = 1
    Dim params As String = ""
    Dim paramValues As String = ""
    Dim description_user As String = ""
    Dim preCond As String = ""
    Dim postCond As String = ""
    Dim boundaryCond As String = ""
    Dim StepName As String = ""
    Dim row As Long
    Dim stepNumber As Long = 0
    Dim description As String = ""
    Dim expResult As String = ""
    Dim attributeContent As String = ""
    Dim attributeName As String = ""
    Dim bigAttributeString As String = ""
    Dim parametersArray As Array
    Dim paramsValuesArray As Array

    'for all rows...
    For row = START_ROW To lastRow
        If XLWorksheet.Cells(row, FIRST_COLUMN).Value = EXPORT_DESIGN_STEP Then
            If XLWorksheet.Cells(row, column).Value = "X" Then
                row += 1
                stepNumber += 1
                'for a Design Step block...
                While ((Not ("Design Step".Equals(XLWorksheet.Cells(row, FIRST_COLUMN).Value))) And (row <= lastRow))
                    If STEP_NAME.Equals(XLWorksheet.Cells(row, FIRST_COLUMN).Value) Then
                        StepName = STEP_NUM & stepNumber & vbNewLine & vbNewLine
                        StepName = String.Concat(StepName, XLWorksheet.Cells(row, column).Value)
                    ElseIf DESCRIPTION_TEXT.Equals(XLWorksheet.Cells(row, FIRST_COLUMN).Value) Then
                        description_user = "<b>" & DESCRIPTION_TEXT & ":" & "</b><br>"
                        description_user = String.Concat(description_user, XLWorksheet.Cells(row, column).Value)
                        description_user = description_user & "<br>"
                    ElseIf PRE_COND_TEXT.Equals(XLWorksheet.Cells(row, FIRST_COLUMN).Value) Then
                        preCond = "<b>" & PRE_COND_TEXT & ":" & "</b><br>"
                        preCond = String.Concat(preCond, XLWorksheet.Cells(row, column).Value)
                        preCond = preCond & "<br>"
                    ElseIf POST_COND_TEXT.Equals(XLWorksheet.Cells(row, FIRST_COLUMN).Value) Then
                        postCond = "<b>" & POST_COND_TEXT & ":" & "</b><br>"
                        postCond = String.Concat(postCond, XLWorksheet.Cells(row, column).Value)
                        postCond = postCond & "<br>"
                    ElseIf BOUNDARY_COND.Equals(XLWorksheet.Cells(row, FIRST_COLUMN).Value) Then
                        boundaryCond = "<b>" & BOUNDARY_COND & ":" & "</b><br>"
                        boundaryCond = String.Concat(boundaryCond, XLWorksheet.Cells(row, column).Value)
                        boundaryCond = boundaryCond & "<br>"
                    ElseIf PARAM_VALUES.Equals(XLWorksheet.Cells(row, FIRST_COLUMN).Value) Then
                        paramValues = "<b>" & PARAM_VALUES & ":" & "</b><br>"
                        paramsValuesArray = Me.splitParamValues(XLWorksheet.Cells(row, column).Value)
                        If Not paramsValuesArray Is Nothing Then
                            For Each value As String In paramsValuesArray
                                paramValues = paramValues & value.Trim
                            Next
                            paramValues = paramValues & "<br>"
                        Else
                            paramValues = paramValues & "<br>"
                        End If
                    ElseIf EXP_RESULT.Equals(XLWorksheet.Cells(row, FIRST_COLUMN).Value) Then
                        expResult = XLWorksheet.Cells(row, column).Value
                    ElseIf PARAMETERS.Equals(XLWorksheet.Cells(row, FIRST_COLUMN).Value) Then
                        params = "<b>" & PARAMETERS & ":" & "</b><br>"
                        parametersArray = Me.splitParameters(XLWorksheet.Cells(row, column).Value)
                        If Not parametersArray Is Nothing Then
                            For Each param As String In parametersArray
                                params = params & param.Trim
                            Next
                            params = params & "<br>"
                        Else
                            params = params & "<br>"
                        End If
                    ElseIf STEP_PROPERTIES.Equals(XLWorksheet.Cells(row, FIRST_COLUMN).Value) Then
                        'do nothing (Customized Step Attributes (blue cell))
                    ElseIf XLWorksheet.Cells(row, FIRST_COLUMN).Value = "" Then
                        'do nothing (should be grey line)
                    Else
```

```
                            attributeName = "<b>" & XLWorksheet.Cells(row, FIRST_COLUMN).Value & ":" & "</b><br>"
                            attributeContent = XLWorksheet.Cells(row, column).Value & "<br>"
100                         bigAttributeString = String.Concat(bigAttributeString, attributeName)
                            bigAttributeString = String.Concat(bigAttributeString, attributeContent)
                            bigAttributeString = bigAttributeString & "<br>"
                            If attributeContent.Equals("<br>") Then
                                                        'customized attribute can be ignored for this step
105                             bigAttributeString = ""
                            End If
                        End If
                        row += 1
                    End While
110                 description = description_user & "<br>" & preCond & "<br>" & postCond & "<br>" & boundaryCond & "<br>" &
                          params & "<br>" & paramValues & "<br>"

                    'if attributeContent is not empty then concatenate content to description String:
                    If Not attributeContent.Equals("") Then
                        description = String.Concat(description, bigAttributeString)
115                 End If

                    'convert "description" to html:
                    description = HTML_OPEN & description & HTML_CLOSE
                                        'get DesignStepFactory object
120                 desStepFact = test.DesignStepFactory
                                        'create new Design Step:
                    desStep = desStepFact.AddItem(DBNull.Value)
                    desStep.StepName = StepName
                    desStep.StepExpectedResult = expResult
125                 desStep.StepDescription = description
                    desStep.Post()
                End If
            End If
        Next
130 End Sub
```

Listing 11: *Hinzufügen von Testschritten*

B.6 addAttachment

```
0   '*************************************************************************************
    '* Name: uploadAttachmentToFolder                                                    *
    '* Type: Procedure                                                                   *
    '* Parameters: In -> path of Excel sheet as string (source),                         *
    '*                   path of project folder in Quality Center's Test Plan            *
5   '*                   (destination)                                                   *
    '* Description: Uploads the Excel sheet from source path as attachment to selected  *
    '*              project folder (destination path) in Quality Center's Test Plan if   *
    '*              chosen by user.                                                      *
    '*************************************************************************************
10  Public Sub uploadAttachmentToFolder(ByVal sourcePath As String, ByVal destPath As String)
        Dim QCConnection As TDAPIOLELib.TDConnection
        Dim folder As TDAPIOLELib.SubjectNode
        Dim treeM As TDAPIOLELib.TreeManager
        Dim attachment As TDAPIOLELib.Attachment
15      Dim attachF As TDAPIOLELib.AttachmentFactory

        If Not Form2.getUpload Then
            'upload is not selected
            Exit Sub
20      Else
            QCConnection = Login.getQCConnection
            treeM = QCConnection.TreeManager
            folder = treeM.NodeByPath(destPath)

25          'get factory object:
            attachF = folder.Attachments

            'create new attachment:
            attachment = attachF.AddItem(DBNull.Value)
30          attachment.FileName = sourcePath
            attachment.Description = Form2.getDescription
            attachment.Type = TDAPIOLELib.TDAPI_ATTACH_TYPE.TDATT_FILE
            Try
                attachment.Post()
35          Catch ex As Exception
                MsgBox("Could not upload excel file. Please make sure that selected file is not in use by any other program!",
                       vbExclamation, "Information Message!")
            End Try
        End If
    End Sub
```

Listing 12: *Hinzufügen der Testfallspezifikation als Anhang*

C Testfallspezifikation mit Tests

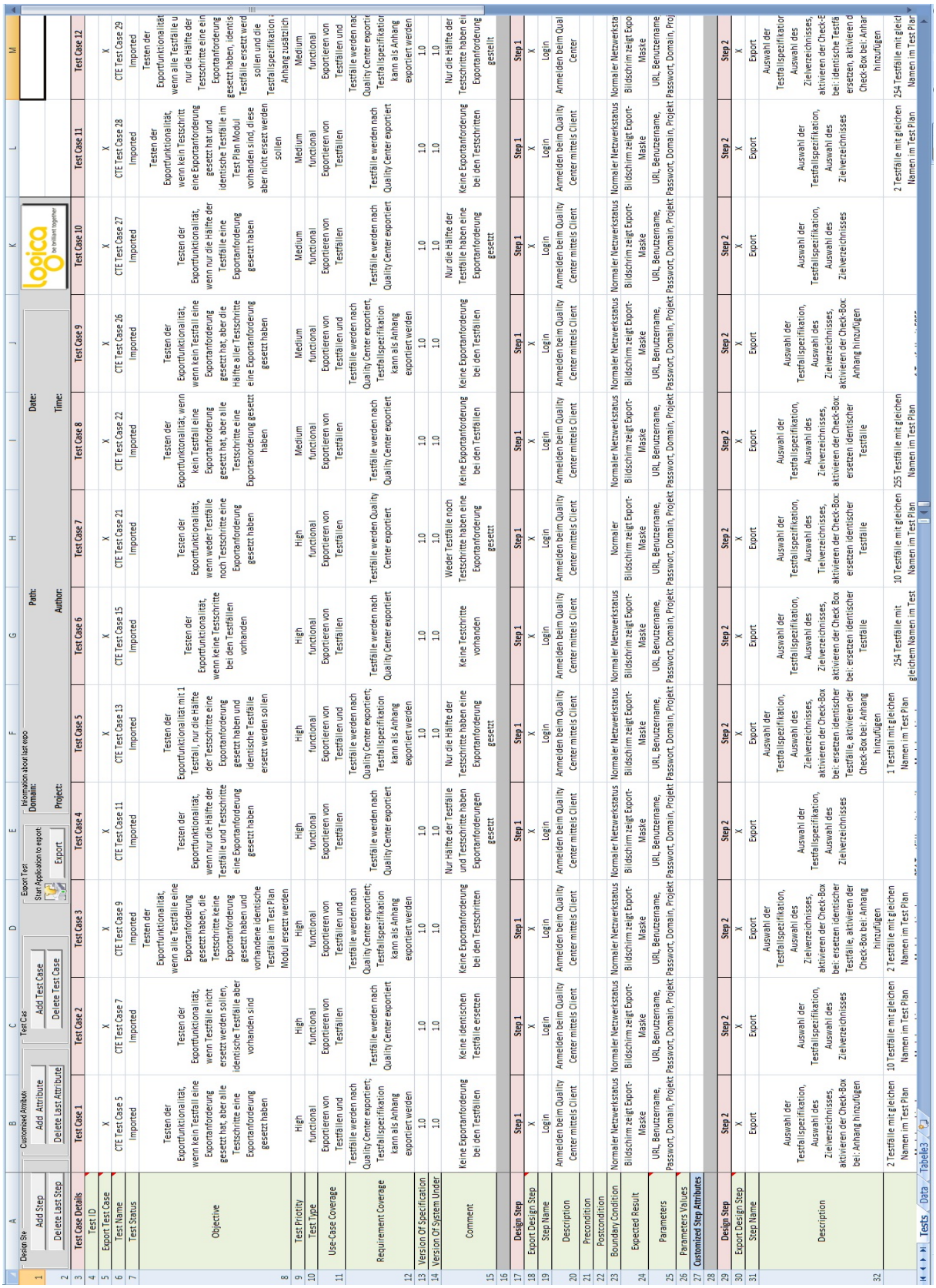

Abb. 39: *Ausschnittweise Darstellung in die Testfallspezifikation übertragener Testfälle aus Abb. 34*

D Beiliegende CD

Inhalt der CD:

1. Bachelorarbeit als PDF Datei
2. Ordner mit allen Bildern der Bachelorarbeit
3. Ordner mit Installationsroutine für den Client
4. Ordner mit Quellcode des Clients
5. Ordner mit Testfallspezifikation (beinhaltet Quellcode der Makros)

Versicherung über Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §24(5) ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, den 18. Juli 2011	
Ort, Datum	Unterschrift