Technische Universität Darmstadt
Fachbereich Informatik
Fachgebiet Theoretische Informatik
Prof. Dr. J. Buchmann

Ein Framework zur Automatisierung von Tests formularbasierter Web-Anwendungen

Diplomarbeit von Heiko Hornung

Betreuer: Markus Ruppert

März 2002

Hiermit versichere ich, die vorliegende Diplomarbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Weinheim, 5. April 2002

Heiko Hornung
`Heiko.Hornung@epost.de`

Vorwort

„Gestern hat es noch funktioniert!“ oder „Ich dachte, das hätte ich bereits korrigiert!“ sind zwei Aussprüche, die jedem geläufig sein dürften, der schon einmal mit Software-Entwicklung zu tun hatte. Diese Aussprüche fallen meistens dann, wenn das Testen der entsprechenden Software vernachlässigt wurde.

Diese Arbeit beschäftigt sich mit dem Testen von Software als einem zentralen Aspekt innerhalb der Software-Qualitätssicherung. Dazu wurde ein Framework entwickelt, in dem Konzepte und Mechanismen zum Testen formularbasierter Web-Anwendungen implementiert wurden. Die Motivation zur Erstellung dieser Arbeit lag darin, den Testprozeß der Trustcenter-Software „FlexiTrust“ zu automatisieren. Ziel war es, ein mächtiges aber einfach zu bedienendes Framework zu entwickeln, welches den Testprozeß von FlexiTrust unterstützt.

An dieser Stelle möchte ich mich bei Marcus Lippert und Alexander Wiesmaier bedanken, die sich stets Zeit für die Beantwortung meiner Fragen nahmen. Dank gilt auch Markus Ruppert, der diese Arbeit betreut und die Zusammenarbeit mit den anderen Projektmitgliedern koordiniert hat.

Inhaltsverzeichnis

Abbildungsverzeichnis **vi**

Abkürzungsverzeichnis **vii**

1 Einführung **1**
1.1 Motivation 1
1.2 Kurzübersicht zu FlexiTrust 2
1.3 Was will diese Arbeit leisten? 4
1.4 Vorgehensweise 5

2 Theoretische Grundlagen **6**
2.1 Grundlagen der Software-Qualitätssicherung 6
2.1.1 Qualität und Qualitätszielbestimmung 6
2.1.2 Qualitätsmanagement 10
2.2 Grundlagen des Software-Testens 14
2.2.1 Strukturtestverfahren 15
2.2.2 Funktionale Testverfahren 18
2.2.3 Kombinierte Funktions- und Strukturtests 19
2.2.4 Modul-, Integration-, System- und Abnahmetests 20
2.2.4.1 Modultest 21
2.2.4.2 Integrationstest 21
2.2.4.3 Systemtest 23
2.2.4.4 Abnahmetest 25
2.2.5 Testprozeß und -dokumentation 26
2.2.6 Grundsätze für den Software-Test und Probleme in der Praxis . . . 28

3 Entwurf des Frameworks **31**
3.1 Vorüberlegungen 31
3.2 Anwendungsbereich 32

3.3 Anforderungen 33
3.4 Komponenten des Frameworks 35

4 Implementierung des Frameworks **38**
4.1 Grundsätzliche Funktionsweise 39
4.2 Datenspeicherung 41
4.3 Konfiguration 42
4.3.1 Einschränkung 44
4.4 Steuerung 44
4.4.1 Einschränkung 46
4.5 Protokollierung 46
4.6 Testfallaufzeichnung 47
4.7 Testfallgenerator 50
4.7.1 Einschränkung 51
4.8 Testfallkonverter 52
4.9 Testdurchführung 53
4.10 Ergebnisüberprüfung 57

5 Ausblick **60**

A User Guide **62**
A.1 Overview 62
A.1.1 What is it? 62
A.1.2 How does it work? 62
A.2 Installation of the framework 64
A.2.1 System Requirements 64
A.2.2 Installation 64
A.3 Getting started 65
A.3.1 The build file 65
A.3.2 Generating test cases 65
A.3.3 Converting test cases 66
A.3.4 Running tests 66
A.3.5 Running single tests 67
A.3.6 Capturing client input 68
A.3.7 Customizing the framework 69
A.4 Adding new profiles (forms) 69
A.5 ToDo 75

A.6 Feedback 75

Literaturverzeichnis **76**

Abbildungsverzeichnis

2.1 Aufbau von FCM-Qualitätsmodellen (factor-criteria-metrics-models) 9
2.2 Summationseffekt von Fehlern und Mängeln 13
2.3 Vorteile verschiedener Organisationsformen der Qualitätssicherung 14
2.4 Klassifikation analytischer QM-Maßnahmen 16
2.5 Kontrollflußgraph mit 2^{20} Pfaden 17
2.6 Kategorien von Testaufgaben mit Merkmalen 20
2.7 Überblick über Integrationsstrategien 22

3.1 Informationsfluß zwischen den Komponenten des Frameworks 35

4.1 Erster Implementierungsansatz für die Testfallaufzeichnung 48
4.2 Zweiter Implementierungsansatz für die Testfallaufzeichnung 49

Abkürzungsverzeichnis

API	Application Programming Interface
CA	Certification Authority
DTD	Documenttyp-Definition
IS	Infrastructure Services
JAXB	Java Architecture for XML Binding
JCA	Java Cryptography Architecture
JDBC	Java Database Connectivity
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
LDAP	Leightweight Directory Access Protocol
PKI	Public-Key-Infrastruktur
QM	Qualitätsmanagement
QMS	Qualitätsmanagementsystem
QS	Qualitätssicherung
RA	Registration Authority
XML	Extensible Markup Language

Kapitel 1

Einführung

1.1 Motivation

Im Projekt FlexiPKI wird am Institut für Theoretische Informatik der TU Darmstadt eine Public-Key-Infrastruktur (PKI) entwickelt, deren Komponenten flexibel, skalierbar und dynamisch erweiterbar sein sollen. Mittlerweile sind aus diesem Projekt mehrere Software-Produkte hervorgegangen. Eines dieser Produkte ist FlexiTrust, eine Trustcenter-Software, welche im folgenden Abschnitt näher beschrieben wird. Die Entwicklung von FlexiTrust ist stark von dem Umstand geprägt, daß der ursprüngliche Schwerpunkt des Projekts FlexiPKI in der Erforschung neuer Technologien lag, welche sich zum Einsatz in PKIs eignen.

Neben einigen festen Mitarbeitern des Instituts für Theoretische Informatik wurde ein großer Teil der Entwicklung von Studenten im Rahmen von HiWi-Jobs, Praktika, Studien- und Diplomarbeiten realisiert. Das bedeutet, daß die meisten Mitarbeiter mitten in einer bestimmten Projektphase in das Projekt eintreten und dem Projekt anschließend nur für einen sehr begrenzten Zeitraum zur Verfügung stehen. Im Extremfall arbeiten sie an einem bereits laufenden Teilprojekt und scheiden vor dessen Abschluß aus der Projektgruppe aus. Dies hat einen erhöhten Aufwand für Koordinationsaufgaben der Projektleitung und die Einarbeitung von neuen Mitarbeitern zur Folge.

Mittlerweile ist die Entwicklung von FlexiTrust an einem Punkt angelangt, an dem absehbar ist, daß die gestiegene Komplexität nicht mehr auf unbestimmte Zeit „nebenbei“ beherrscht werden kann. Darüber hinaus gibt es bereits einige Kunden, die FlexiTrust in einem kommerziellen Umfeld einsetzen. Eine Reihe anderer potentieller Kunden erwägt den Einsatz von FlexiTrust. Um weiterhin eine effiziente Weiterentwicklung von FlexiTrust zu gewährleisten, neuen Mitarbeitern den Einstieg in das Projekt zu erleichtern und Kunden

gegenüber die Qualität von FlexiTrust dokumentieren zu können, muß daher ein Qualitätsmanagementsystem (QMS) eingeführt werden. In diesem QMS werden bereits eingesetzte Qualitätssicherungsmaßnahmen formalisiert und Qualitätssicherungskonzepte sowie verbindliche Entwicklungsrichtlinien dokumentiert. Ein besonderer Schwerpunkt liegt dabei auch auf dem Testen von FlexiTrust zur Sicherstellung der Funktionalität und Robustheit.

1.2 Kurzübersicht zu FlexiTrust

In diesem Abschnitt möchten wir dem Leser einen Überblick über FlexiTrust geben. Dabei setzen wir Grundkenntnisse der Public-Key-Kryptographie voraus. Für eine Einführung in dieses Thema sowie in die Aufgaben und Funktionsweise von PKIs siehe z.B. [Buc01].

Grundsätzlich besteht FlexiTrust aus den drei Hauptkomponenten

- Registration Authority (RA),
- Certification Authority (CA) und
- Infrastructure Services (IS).

Daneben gibt es noch Schnittstellen zu einem Krypto-Provider, welcher verschiedene Kryptoalgorithmen zur Verfügung stellt, sowie zu einer Datenbank und einem LDAP-Server. Die einzelnen Komponenten werden im folgenden nur soweit beschrieben, wie es für das Verständnis dieser Arbeit notwendig ist. Detailliertere Darstellungen finden sich in [Wie01], [Dam01] sowie [Sch01]. Eine kompakte Übersicht über FlexiTrust findet sich unter [FLE02].

Die CA ist die zentrale Dienstleistungskomponente eines Trustcenters. Hier werden die geheimen Schlüssel erzeugt und verwaltet, Zertifikate signiert oder revoziert, Revokationslisten erstellt, etc. Die RA kann als Eingangsmodul gesehen werden, in dem Anträge (z.B. auf Austellung eines Zertifikats) von Kunden abgewickelt werden. Hier wird entschieden, ob ein Kunde eine gewünschte Dienstleistung erhält oder nicht. Die IS ist für die Verwaltung wichtiger Daten wie Kundendaten und die Veröffentlichung von Zertifikaten oder Zertifikatsrevokationslisten zuständig.

Um die Anschaulichkeit der nun folgenden Beschreibung des Zusammenspiels der FlexiTrust-Komponenten zu verbessern, orientieren wir uns am Workflow eines Zertifikatsantrags. Ein solcher Antrag kann dabei entweder direkt aus einer Datenbank kommen oder z.B. von einem Benutzer über ein Servlet-basiertes Web-Formular gestellt werden. Der Aufbau des

Web-Formulars und die Arbeitsweise des Servlets werden weiter unten beschrieben. Die RA nimmt nun einen solchen Antrag entgegen und bereitet ihn so auf, daß er von der CA weiterverarbeitet werden kann. Dazu werden zunächst verschiedene Überprüfungen durchgeführt, z.B. ob die Daten vom Antragsteller korrekt eingegeben wurden oder ob die Identität des Antragstellers mit den Angaben im Antrag übereinstimmt. Nach erfolgreicher Überprüfung wird der Antrag in ein für die CA verarbeitbares Format überführt und an die CA weitergeleitet. Gleichzeitig wird ein Produktdatensatz für die IS erzeugt. Dieser Datensatz enthält u.a. Informationen über den Auslieferungsstatus eines Zertifikats oder den Auslieferungsmodus für ein Zertifikat.[1] In der CA werden dann je nach Antrag entsprechende Schlüssel erzeugt und signiert. Nachdem das Zertifikat die CA verlassen hat, wird es entsprechend der Informationen des vorher erzeugten Produktdatensatzes weiterverarbeitet.

Da in dieser Arbeit ein Framework zum Testen formularbasierter Web-Anwendungen vorgestellt wird, welches auch zum Testen von FlexiTrust benutzt wird, möchten wir nun die Arbeitsweise des Servlets beschreiben. Für jeden Antrag, der gestellt werden kann, exisitiert ein eigenes Formular, welches mit Hilfe einer XML-Datei beschrieben wird. Mit diesem Mechanismus ist es auch möglich, während der Laufzeit von FlexiTrust Formulare zu ändern oder hinzuzufügen. Ein Formular besteht aus verschiedenen Formularkomponenten, die zueinander in einer hierarchischen Beziehung stehen. Eine Formularkomponente definiert die Art eines Formularfelds. Gleichzeitig ist eine Formularkomponente mit sog. PlugIns assoziiert, die festlegen, welche Überprüfungen für die entsprechende Formularkomponente durchgeführt werden müssen, und die sequentiell ausgeführt werden können. Beispielsweise beschreibt eine Formularkomponente ein Paßwort-Feld. Diese Formularkomponente erbt von der Formularkomponente, welche ein Textfeld beschreibt. Für das Paßwort-Feld kann dann ein PlugIn definiert werden, welches eine Menge gültiger Zeichen festlegt, die in diesem Feld eingegeben werden dürfen. Zusätzlich kann dann ein PlugIn definiert werden, welches z.B. eine bestimte Mindest- und Höchstanzahl von Zeichen festlegt und vorgibt, daß das Paßwort eine bestimmte Anzahl von Ziffern sowie Groß- und Kleinbuchstaben enthalten muß.

> **Beispiel:** Wenn ein Antragsteller die gerade bearbeitete Seite des Web-Formulars abschickt, werden die Eingaben mit Hilfe der PlugIns überprüft, welche für die entsprechenden Formularkomponenten definiert wurden. War die Überprüfung erfolgreich, bekommt der Antragsteller ein Web-Formular geschickt, welches die Formularkomponenten enthält, die der nächsten Seite des Formulars entsprechen. Ansonsten erhält er ein Web-Formular, welches noch einmal

[1] Das Zertifikat kann z.B. per E-Mail verschickt werden oder auf einem LDAP-Server veröffentlicht werden.

die Formularkomponenten der gerade bearbeiteten Seite enthält, ergänzt durch Hinweise, welche Formularfelder überarbeitet werden müssen.

Die Programmierplattform für FlexiTrust ist Java, eine objektorientierte und portable Sprache. Dies ist im Kontext unserer Arbeit insofern wichtig, da es mit Java möglich ist, wiederverwendbare, modulare und robuste Programme zu entwickeln. Die Verwendung von Java hat daher einen direkten Einfluß auf die Qualität des damit realisierten Software-Produkts.

Die gesamte Anwendung ist flexibel, skalierbar und dynamisch erweiterbar. Der Krypto-Provider wurde z.B. konform zur Java Cryptography Architecture (JCA)[2] entwickelt und kann daher durch andere JCA-konforme Krypto-Provider ausgetauscht werden. Was die Skalierbarkeit betrifft, so können z.B. während der Laufzeit mehrere Instanzen der CA bestehen, so daß ein Load-Balancing betrieben werden kann. FlexiTrust ist dynamisch erweiterbar, da zur Laufzeit z.B. neue Formulare und neue Produkte hinzugefügt werden können.

1.3 Was will diese Arbeit leisten?

In Abschnitt 1.1 wurden bereits die Gründe für die Einführung bzw. Formalisierung eines QMS für FlexiTrust benannt. Ein besonderes Problem ist dabei die Frage, wie für ein bereits laufendes Software-Projekt ein QMS eingeführt werden kann und wie bereits existierende qualitätssichernde Maßnahmen und Strategien sowie verwendete qualitätssichernde Werkzeuge und Methoden integriert werden können.

Ziel dieser Arbeit ist es aber nicht, ein QMS vollständig bis auf die Ebene von Schablonen, Checklisten und Richtlinien zu beschreiben. Vielmehr soll als ein wichtiger Teilaspekt eines QMS das Testen von Software betrachtet werden. Ein Schwerpunkt liegt dabei auf der Untersuchung der Automatisierbarkeit von Softwaretests. Zu diesem Zweck wurde im Rahmen dieser Diplomarbeit ein Framework zur Automatisierung von Tests formularbasierter Web-Anwendungen entwickelt, welches in dieser Arbeit vorgestellt wird. Im Rahmen einer Diplomarbeit ist es nicht möglich, ein solches Framework vollständig zu implementieren. Es wurde jedoch Wert darauf gelegt, eine gewisse Grundfunktionalität zu implementieren, so daß bereits Tests von Web-Anwendungen durchgeführt werden können.

[2]Die JCA ist ein Java-Konzept für den Einsatz beliebiger Kryptoalgorithmen.

1.4 Vorgehensweise

In Kapitel 2 werden zunächst die Grundlagen der Software-Qualitätssicherung und die des Software-Testens beschrieben. Anschließend wird in Kapitel 3 der Entwurf des Frameworks vorgestellt. Kapitel 4 beschreibt die Implementierung des Frameworks. Kapitel 5 rundet die Arbeit mit einem Ausblick ab.

Kapitel 2

Theoretische Grundlagen

2.1 Grundlagen der Software-Qualitätssicherung

Qualitätssicherung ist nicht nur in der Informatik schon seit vielen Jahren Gegenstand der Forschung. Es existiert daher eine große Anzahl von Veröffentlichungen zu diesem Thema. Im Rahmen einer Diplomarbeit ist es unmöglich, alle Aspekte zu diesem Thema abzudecken. Da der Gegenstand unserer Untersuchungen das Testen von Software ist, soll im folgenden ein Überblick zum Thema Qualtätssicherung nur soweit gegeben werden, wie es zum Verständnis und zur Einordnung dieser Arbeit in den Gesamtzusammenhang nötig ist.

2.1.1 Qualität und Qualitätszielbestimmung

Am 4. Juni 1996 explodierte die neue Ariane-5-Rakete der Europäischen Weltraumbehörde (ESA) kurz nach dem Start. Der geschätzte Schaden betrug 500 Millionen US-Dollar. Absturzursache war ein Fehler bei der Konversion einer Zahl von der 64 Bit Gleitkommadarstellung in einen 16 Bit signed Integer, der nicht abgefangen wurde.

Im Februar 1999 fiel das Handelssystem eines größeren U.S. Online-Brokers mehrere Male während der Handelszeit aus. Ursache waren Fehler in einem Software-Upgrade, das die Bestätigung von Online-Transaktionen beschleunigen sollte.

Dies sind nur zwei Beispiele[1], die zeigen, daß unsere Gesellschaft in vielen Bereichen auf das fehlerfreie Funktionieren von Software-Systemen angewiesen ist. Software-Systeme steuern Flugzeuge und medizinische Geräte. An den Börsen der Welt werden Tag für Tag Transaktionen in Milliardenhöhe computergestützt ausgeführt. In nahezu allen Lebensberei-

[1] Vgl. [How01]

chen sind computergesteuerte Systeme zu finden. Nicht selten bedeutet der Ausfall eines solchen Systems einen enormen finanziellen Schaden oder sogar eine unmittelbare Gefahr für menschliches Leben.

Daher muß die Erstellung zuverlässiger Software Ziel jeder industriellen Software-Entwicklung sein. „Zuverlässigkeit“ ist jedoch nur ein Qualitätsaspekt eines Software-Produkts. Über den Qualitätsbegriff gibt es unterschiedliche Auffassungen, die sich grundsätzlich in fünf verschiedene Ansätze einteilen lassen:[2]

- Der *transzendente Ansatz*[3] sieht Qualität als universell erkennbar, einzigartig und vollkommen. Qualität steht für kompromißlos hohe Standards und Ansprüche an die Funktionsweise. Qualität läßt sich nach diesem Ansatz nicht exakt definieren und messen, sondern nur durch Erfahrung bewerten. Dieser Ansatz ist für die Praxis nicht geeignet.

- Nach dem *produktbezogenen Ansatz* ist Qualität eine meßbare, genau spezifizierte Größe, die das Produkt beschreibt. Anhand der gemessenen Größe kann eine Rangordnung von verschiedenen Produkten der gleichen Kategorie aufgestellt werden. Da sich dieser Ansatz nur auf das Endprodukt bezieht, kann dies zu einer mangelnden Berücksichtigung der Kundeninteressen führen.

- Im *produktbezogenen Ansatz* wird Qualität durch den Produktbenutzer festgelegt. Ein Produkt ist qualitativ hochwertig, wenn es die Wünsche und Bedürfnisse des Benutzers befriedigt. Da der Benutzer oft nur vage Qualitätsvorstellungen hat, ist es für den Hersteller schwierig, die Bedürfnisse des Benutzers im voraus zu bestimmen.

- Nach dem *prozeßbezogenen Ansatz* entsteht Qualität durch die richtige Erstellung des Produkts. Der Erstellungsprozeß wird spezifiziert, kontrolliert und permanent an sich wandelnde Kundenbedürfnisse angepaßt, um Nacharbeitungskosten zu minimieren.

- Der *Kosten/Nutzen-bezogene Ansatz* bestimmt Qualität durch eine Funktion von Kosten und Nutzen. Ein Produkt besitzt eine hohe Qualität, wenn es einen bestimmten Nutzen zu einem akzeptablen Preis erbringt.

In DIN 55350, Teil 11 wird Qualität entsprechend dem produkt- und prozeßbezogenen Ansatz definiert:[4]

---

[2] Vgl. [Bal98], S. 256f.
[3] transzendent = übernatürlich
[4] Vgl. [DIN86]

> „Qualität ist die Gesamtheit von Eigenschaften und Merkmalen eines Produkts oder einer Tätigkeit, die sich auf deren Eignung zur Erfüllung gegebener Erfordernisse bezieht.“

Die ANSI/IEEE-Norm für Software-Qualität (IEEE Std 729-1983) betont die Erwartungen der Kunden:[5]

> „**Software quality:**
>
> (1) The totality of features and characteristics of a software product that bear on its ability to satisfy given needs: for example, conform to specifications.
>
> (2) The degree to which software possesses a desired combination of attributes.
>
> (3) The degree to which a customer or user perceives that software meets his or her composite expectations.
>
> (4) The composite characteristics of software that determine the degree to which the software in use will meet the expectations of the customer.“

Diese allgemeinen Definitionen sind für die praktische Anwendung nicht ausreichend. Um den Qualitätsbegriff zu operationalisieren, verwendet man ein Qualitätsmodell.[6] In einem solchen Modell wird die Software-Qualität durch Qualitätsmerkmale beschrieben, welche in einem weiteren Schritt in Teilmerkmale bzw. Kriterien untergliedert werden. Diese Teilmerkmale werden mit Hilfe von Qualitätsindikatoren bzw. Metriken meß- und bewertbar gemacht. Ein so aufgebautes Modell bezeichnet man auch als FCM-Modell (factor-criteria-metrics-model[7]). Abbildung 2.1 zeigt den Aufbau von FCM-Modellen. Da mehrere Qualitätsmerkmale oft gemeinsame Teilmerkmale besitzen, kann ein FCM-Modell einen Baum oder ein Netz bilden. Ferner können die Teilmerkmale selbst in einer hierarchischen Beziehung stehen, so daß ein FCM-Modell auch mehr als drei Stufen enthalten kann.

FCM-Modelle können sowohl für die Produkt- als auch für die Prozeßqualität beschrieben werden. Beispiele für verschiedene FCM-Modelle sowie weitere Ansätze zur Entwicklung von Qualitätsmodellen finden sich in [Bal98], S. 258ff. Nach der Erstellung eines Qualitätsmodells erfolgt die Qualitätszielbestimmung, nach deren Durchführung sich die Qualitätsanforderungen ergeben. Diese legen fest, welche Qualitätsmerkmale für den konkreten Fall als relevant erachtet werden und inwieweit sie erreicht werden sollen. Falls sich

[5]Vgl. [ANSa]

[6]Vgl. [Bal98], S. 257f.

[7]Im englischen Sprachraum wird anstelle des Begriffs „Merkmal“ der Begriff „factor“ verwendet.

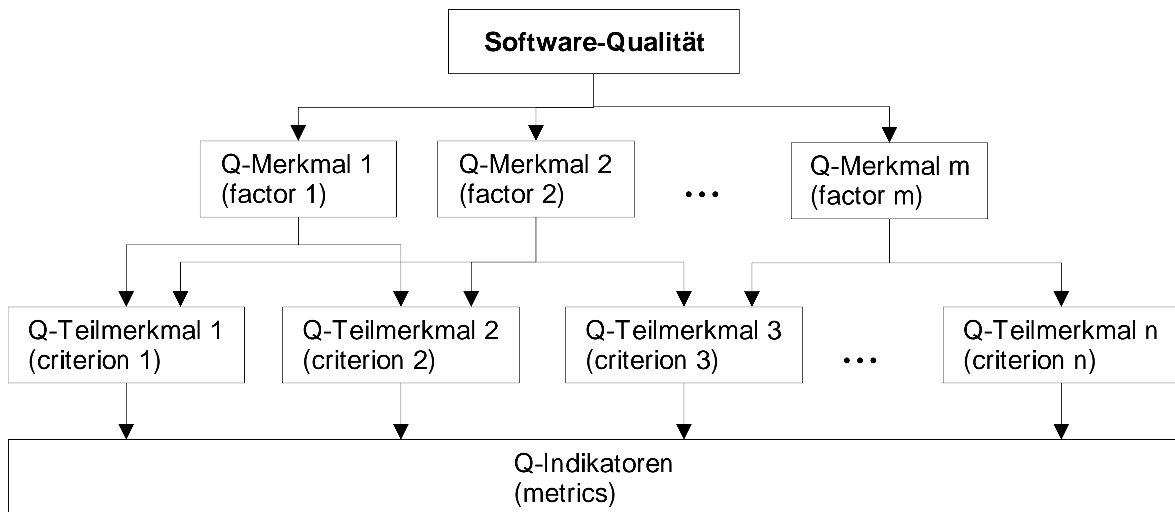

Abbildung 2.1: Aufbau von FCM-Qualitätsmodellen (factor-criteria-metrics-models). *Quelle: [Bal98], S. 257*

das benutzte Qualitätsmodell auf Indikatoren bzw. Kennzahlen stützt, können dazu für die einzelnen Qualitätsmerkmale Qualitätsstufen definiert werden. Eine Qualitätsanforderung ordnet dann einem Qualitätsmerkmal eine zu erreichende Qualitätsstufe zu. Die Qualitätszielbestimmung ist vor Entwicklungsbeginn durchzuführen und zu dokumentieren, um sicherzustellen, daß sich sowohl der Auftraggeber als auch der Entwickler auf eine definierte Qualitätsbasis stützen können.[8]

> **Beispiel:** Ein Qualitätsmodell für Software-Produkte definiert die Qualitätsmerkmale Funktionalität, Zuverlässigkeit, Benutzbarkeit, Flexibilität und Performanz. Das Merkmal Performanz ist in die Teilmerkmale Geschwindigkeit, Ressourcenverbrauch, Durchsatz und Antwortzeit untergliedert. Der Ressourcenverbrauch kann noch in Speicher- und Prozessorauslastung unterteilt werden. Die Metriken für diese Teilmerkmale ist die Auslastung in Prozent. Die Metrik für die Antwortzeit ist die Zeit in Sekunden.
>
> Je nach der Anwendungsklasse des Software-Produkts können nun unterschiedliche Qualitätsanforderungen identifiziert werden. Handelt es sich um eine Echtzeitanwendung, so beziehen sich die Qualitätsanforderungen mit der höchsten Priorität auf die Teilmerkmale des Qualitätsmerkmals Performanz. Sind mit der Anwendungsentwicklung sehr hohe Kosten verbunden, liegt die Priorität z.B. auf den Merkmalen Zuverlässigkeit und Flexibilität.

[8] Vgl. [Bal98], S. 269ff.

2.1.2 Qualitätsmanagement

Nach der Definition von Qualitätsanforderungen muß sichergestellt werden, daß diese auch erreicht werden. In diesem Unterabschnitt möchten wir daher näher auf das Qualitätsmanagement eingehen. Die Norm ISO/FDIS 9000:2000-09 definiert diesen Begriff wie folgt:[9]

> „**Qualitätsmanagement**
> Aufeinander abgestimmte Tätigkeit zur Leitung und Lenkung einer Organisation bezüglich Qualität.
> Anmerkung: Leitung und Lenkung bezüglich Qualität umfassen üblicherweise die Festlegung der Qualitätspolitik und der Qualitätsziele, die Qualitätsplanung, die Qualitätslenkung, die Qualitätssicherung und die Qualitätsverbesserung."

Man kann zwischen produktorientiertem und prozeßorientiertem Qualitätsmanagement (QM) unterscheiden.[10] Beim produktorientierten QM werden Produkte und Zwischenergebnisse auf vorher festgelegte Qualitätsmerkmale überprüft. Prozeßorientiertes QM bezieht sich auf den Erstellungsprozeß der Software. Dazu gehören Methoden, Richtlinien, Werkzeuge und Standards. Die Maßnahmen, die zur Durchführung des Qualitätsmanagements zur Verfügung stehen, lassen sich in vier Kategorien einteilen:[11]

- planerisch-administrative,
- konstruktive,
- analytische und
- psychologisch-orientierte Maßnahmen.

Diese Maßnahmen können auf verschiedenen Ebenen definiert werden, nämlich projektübergreifend, projektbezogen oder phasenbezogen. Bei planerisch-administrativen QM-Maßnahmen geht es um den Aufbau, die Einführung und die Pflege eines Qualitätsmanagementsystems, welches sowohl projektübergreifend als auch projekt- und phasenbezogen wirksam wird. Konstruktive QM-Maßnahmen können in technische Maßnahmen (Methoden, Sprachen, Werkzeuge, etc.) und organisatorische Maßnahmen (Richtlinien, Standards, Checklisten, etc.) gegliedert werden. Diese Maßnahmen sollen dafür sorgen, daß das entstehende Produkt bzw. der Erstellungsprozeß von vornherein bestimmte Eigenschaften besitzt. Im Gegensatz dazu dienen analytische QM-Maßnahmen der Prüfung und Bewertung der

[9] Vgl. [ISO00], S.15
[10] Vgl. [Bal98], S. 279
[11] Vgl. [Wal01], S. 30ff.

Qualität. In Abschnitt 2.1.2 wird auf analytische QM-Maßnahmen näher eingegangen. Die Kategorie der psychologisch-orientierten QM-Maßnahmen betrifft den Menschen als Entwickler oder Projektleiter. Hier kann man zwischen Maßnahmen unterscheiden, welche die Arbeit des Einzelnen oder die des Teams betreffen.

Grundsätzlich gilt, daß eine vorausschauende konstruktive Qualitätslenkung viele analytische Maßnahmen erspart. Gleichzeitig ist es aber auch möglich, daß analytische Maßnahmen erst durch das Ergreifen bestimmter konstruktiver Maßnahmen möglich werden. Eine Modularisierung (konstruktive Maßnahme) ist z.B. die Voraussetzung für einen umfassenden Modultest (analytische Maßnahme), in dem das Testkriterium „Jeder Zweig eines Programms soll mindestens einmal durchlaufen werden“ überprüft wird. Mit einem fertigen, nicht modular aufgebauten Gesamtprodukt ist ein solcher Test nicht mehr durchführbar. Ein generelles Ziel muß also sein, durch den Einsatz geeigneter konstruktiver Maßnahmen den analytischen Aufwand zu reduzieren.

Entsprechend der gerade zitierten ISO-Norm sind zur Durchführung des Qualitätsmanagements u.a. folgende Aufgaben erforderlich:[12]

- Qualitätsplanung,
- Qualitätslenkung und -sicherung,
- Qualitätsprüfung

Aufgabe der Qualitätsplanung ist die Auswahl eines Qualitätsmodells und die Durchführung der Qualitätszielbestimmung, d.h. die Festlegung von Qualitätsanforderungen in einer überprüfbaren Form. Darauf aufbauend sind Qualitätslenkungs- und -sicherungsmaßnahmen abzuleiten. Die Qualitätslenkung und -sicherung setzt die Qualitätsplanung um. Der Entwicklungsprozeß wird gesteuert, überwacht und korrigiert, mit dem Ziel, die vorgegebenen Qualitätsanforderungen zu erfüllen. Die Aufgaben der Qualitätslenkung sind eng mit den Software-Managementaufgaben verknüpft. Die Qualitätsprüfung führt die während der Qualitätsplanung festgelegten Maßnahmen zur Erfassung von Qualitäts-Istwerten durch und überprüft die Umsetzung der konstruktiven Maßnahmen. Die Überwachung der korrekten Durchführung der Qualitätsprüfung ist Teil der Qualitätslenkung. Die Ergebnisse der Qualitätsprüfung werden auf Basis von Soll-/Ist-Vergleichen ausgewertet. Gegebenenfalls müssen danach korrektive Maßnahmen von der Qualitätslenkung ergriffen werden. Die Ergebnisse der Qualitätsplanung werden in einem Qualitätssicherungsplan dokumentiert. Die IEEE-

[12] Vgl. auch [Bal98], S. 282f.

Norm 730 legt ein Gliederungsschema für Qualitätssicherungspläne fest.[13] Eine Beispielgliederung mit Erläuterungen für einen an diesen Standard angelehnten Qualitätssicherungsplan findet sich in [IRM02].

Um ein effizientes und effektives Qualitätsmanagement zu ermöglichen, sollten folgende Prinzipien beachtet werden:[14]

- **Prinzip der quantitativen Qualitätssicherung**
  Entscheidend für die Qualität ist, daß zu Beginn des Projekts für das zu planende Produkt die wesentlichen Qualitätsmerkmale festgelegt werden. Es ist wichtig, daß diese Qualitätsmerkmale konkret und quantifizierbar sind, da nur so objektive Vergleiche zwischen Qualitäts-Soll- und -Istwerten möglich sind. Schwierigkeiten ergeben sich dabei u.a. durch den kreativen Charakter vieler Aspekte der Software-Entwicklung. So ist es bspw. schwer möglich, ein Merkmal wie die Benutzbarkeit des Produktes zu quantifizieren.

- **Prinzip der produkt- und prozeßabhängigen Qualitätszielbestimmung**
  Die Qualitätsanforderungen an ein Produkt oder an einen Entwicklungsprozeß werden von Faktoren wie dem Verwendungszweck, der Lebensdauer oder den potentiellen Benutzern bestimmt. Daher sollte vor Entwicklungsbeginn eine explizite und transparente Qualitätszielbestimmung unter Mitwirkung von Entwicklern bzw. Lieferanten und Auftraggebern erfolgen, um für beide Seiten die notwendige Planungs- und Kalkulationssicherheit zu schaffen.

- **Prinzip der maximalen konstruktiven Qualitätssicherung**
  Dieses Prinzip wurde bereits erwähnt.[15] Durch vorausblickende konstruktive Maßnahmen sollen analytische QM-Maßnahmen reduziert und Fehler vermieden werden. Teilweise können auch analytische Maßnahmen durch konstruktive Maßnahmen erst ermöglicht werden. Durch die Anwendung dieses Prinzips wird eine direkte Verbesserung der Produktqualität erzielt.

- **Prinzip der frühzeitigen Fehlerentdeckung und -behebung**
  Untersuchungen haben gezeigt, daß die Kosten für die Entdeckung und Behebung von Fehlern exponentiell mit der Dauer zwischen Entstehung und Entdeckung des Fehlers steigen. Zudem führen Fehler und Mängel in den nachfolgenden Entwicklungsphasen

[13]Vgl. [IEE]
[14]Vgl. [Wal01], S. 17ff. und [Bal98], S. 284ff.
[15]Vgl. S. 11

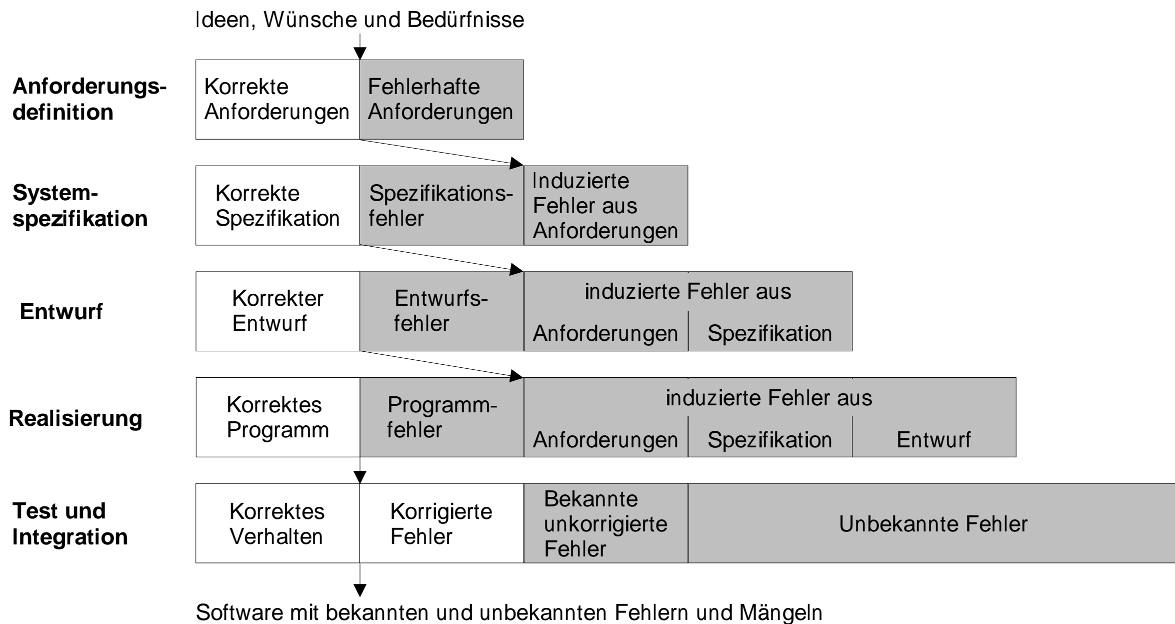

Abbildung 2.2: Summationseffekt von Fehlern und Mängeln. *Quelle: [Miz83], zitiert in [Bal98], S. 288*

zu einem Summationseffekt, wie Abbildung 2.2 zeigt. Ziel muß es also sein, Fehler zum frühestmöglichen Zeitpunkt zu erkennen und zu beheben.

- **Prinzip der entwicklungsbegleitenden, integrierten Qualitätssicherung**
  Dieses Prinzip sollte angewendet werden, um das Prinzip der frühzeitigen Fehlerentdeckung und -behebung zu realisieren. Die Qualitätssicherung findet so zu dem Zeitpunkt statt, zu dem sie im Entwicklungsprozeß angebracht ist. Dadurch wird sichergestellt, daß ein Teilprodukt der nächsten Phase erst dann zur Verfügung steht, wenn eine gewisse Qualität gewährleistet ist. Ferner ist durch die Anwendung dieses Prinzips das Qualitätsniveau zu jedem Zeitpunkt sichtbar, und der Entwicklungsfortschritt kann realistisch beurteilt werden.

- **Prinzip der unabhängigen Qualitätsprüfung**
  Das Ziel von Qualitätsprüfungen ist das Aufdecken von Fehlern und Mängeln. Qualitätsprüfungen sollten mit einer kritischen, wenn nicht sogar destruktiven, Einstellung durchgeführt werden. Aus diesem Grund sollte die Qualitätsprüfung nicht durch den Entwickler selbst erfolgen, da Außenstehende aufgrund einer größeren Objektivität und anderen Sicht- und Denkweisen mehr Fehler und Mängel finden können. Es muß jedoch sichergestellt werden, daß ein Entwickler eigene Aufgaben wie das elementare Testen der Software nicht an die Qualitätssicherung abschiebt. Organisatorisch besteht

QS unabhängig	QS als Teil der Entwicklung
kein Druck der Entwicklung auf die QS	flexibler Personaleinsatz
Neutralität bleibt gewahrt	gemeinsame Teamarbeit
klare Budgetaufteilung	vertrauensvollere Zusammenarbeit
Betonung der Bedeutung der QS	Bessere soziale Integration der QS-Mitarbeiter

Abbildung 2.3: Vorteile verschiedener Organisationsformen der Qualitätssicherung

die Möglichkeit, die Qualitätssicherung entweder unabhängig von der Entwicklung oder als Teil der Entwicklung zu plazieren. Abbildung 2.3 zeigt die Vorteile beider Organisationsformen. Vorteile der einen sind gleichzeitig Nachteile der anderen Form.

- **Prinzip der Bewertung eingesetzter QM-Maßnahmen**
  Das vorhandene Qualitätsmanagementsystem und die eingesetzten QM-Maßnahmen sollten in regelmäßigen Abständen überprüft werden. Diese Prüfung kann sowohl intern als auch im Rahmen von externen Audits[16] erfolgen und hat in der Regel Korrekturen des QMS und der eingesetzten QM-Maßnahmen zur Folge. Durch Anwendung dieses Prinzips kann sowohl eine Steigerung der Rentabilität der in ein QMS getätigten Investitionen erzielt werden als auch eine Anpassung von QM-Maßnahmen an neue Software-Entwicklungstechnologien erfolgen. Ferner werden durch eine unabhängige Qualitätssicherung Qualitätsvergleiche über mehrere Produkte hinweg ermöglicht.

2.2 Grundlagen des Software-Testens

Nachdem wir im vorigen Abschnitt allgemeine Grundlagen der Software-Qualitätssicherung dargestellt haben, möchten wir nun auf das Testen von Software eingehen, welches unter den analytischen QM-Maßnahmen einzuordnen ist. Ziel analytischer QM-Maßnahmen ist es, Fehler, Defekte, Inkonsistenzen und Unvollständigkeiten in dem zu testenden Objekt zu finden. Testobjekt kann dabei ein Software-Produkt, ein Erstellungsprozeß oder auch die Dokumentation des Produkts oder Prozesses sein. Im Rahmen dieser Arbeit soll das Testen von Software-Produkten vertieft werden. Wir bezeichnen ein zu testendes Software-Produkt im folgenden als Prüfling oder Testling, wobei Prüfling der allgemeine Begriff ist und Testling die dynamische Ausführung eines Programms impliziert. Ein anderer Begriff, der an dieser Stelle eingeführt werden soll, ist der des Regressionstests. Ein Regressionstest ist die Wiederholung der bereits durchgeführten Tests nach Änderungen des Programms. Er dient zur

[16]Ein Audit ist eine Überprüfnug, mit der festgestellt wird, ob das Vorgehen in der Praxis mit den schriftlichen Festlegungen übereinstimmt. Vgl. [Wal01], S. 198ff.

Überprüfung der korrekten Funktion nach Modifikationen wie z.B. Fehlerkorrekturen.

Prinzipiell lassen sich die analytischen QM-Maßnahmen wie in Abbildung 2.4 in analysierende, testende und verifizierende Verfahren einteilen.[17] Analysierende Verfahren vermessen oder stellen bestimmte Eigenschaften des Prüflings dar. Verifizierende Verfahren versuchen, die Korrektheit des Prüflings zu beweisen. Testende Verfahren haben das Ziel, Fehler zu erkennen. Sie lassen sich weiter unterteilen in statische und dynamische Verfahren. Statische Testverfahren führen den Prüfling nicht aus, sondern analysieren den Quellcode, um Fehler zu finden. Meistens werden dazu manuelle Verfahren eingesetzt. Beispiele für solche Methoden sind Inspektionen, Reviews und Walkthroughs. Diese drei Verfahren können in allen Phasen des Projektes eingesetzt werden und eignen sich auch zum Testen der Dokumentation. Eine detallierte Beschreibung und Gegenüberstellung dieser drei Verfahren findet sich in [Bal98], S. 301ff.

Dynamische Testverfahren werden in der Literatur meistens in Strukturtests und funktionale Tests unterteilt. Strukturtests lassen sich weiter unterteilen in kontrollflußorientierte und datenflußorientierte Tests. Allen dynamischen Verfahren gemeinsam ist, daß das übersetzte ausführbare Programm mit konkreten Eingabewerten ausgeführt wird. Der Testling kann dabei in seiner realen Umgebung getestet werden. Es handelt sich bei diesen Verfahren um Stichprobenverfahren, d.h. die Korrektheit des Testlings wird nicht bewiesen.

2.2.1 Strukturtestverfahren

Strukturtests, die auch als White- oder Glass-Box-Tests bezeichnet werden, setzen voraus, daß die Struktur des Testlings bekannt ist.[18] Zur Ableitung der Testfälle werden strukturelle Informationen wie der Feinentwurf oder der Quellcode benutzt. In einem vollständigen White-Box-Test muß im Kontrollflußgraphen bzw. Datenflußdiagramm des Testlings jeder Pfad vom Eingang bis zum Ausgang mindestens einmal durchlaufen werden. Abbildung 2.5 zeigt einen Kontrollflußgraphen mit einem zugehörigen Codebeispiel, welches eine Schleife enthält, die zwanzigmal durchlaufen wird. In der Schleife befindet sich eine if-Anweisung. Aus dieser einfachen Struktur resultieren bereits 2^{20} verschiedene Pfade. Schon aus diesem einfachen Beispiel wird deutlich, daß eine vollständige Pfadabdeckung unmöglich ist. Aber selbst wenn jeder Pfad des Testlings gestestet würde, kann dieser noch Fehler enthalten, da z.B. der Testling nicht der Spezifikation entspricht.

[17] Für eine Darstellung in dieser Arbeit nicht beschriebener Verfahren und eine ausführlichere Beschreibung der dargestellten Verfahren vgl. [Bal98], S. 395ff.

[18] Vgl. [Wal01], S. 229

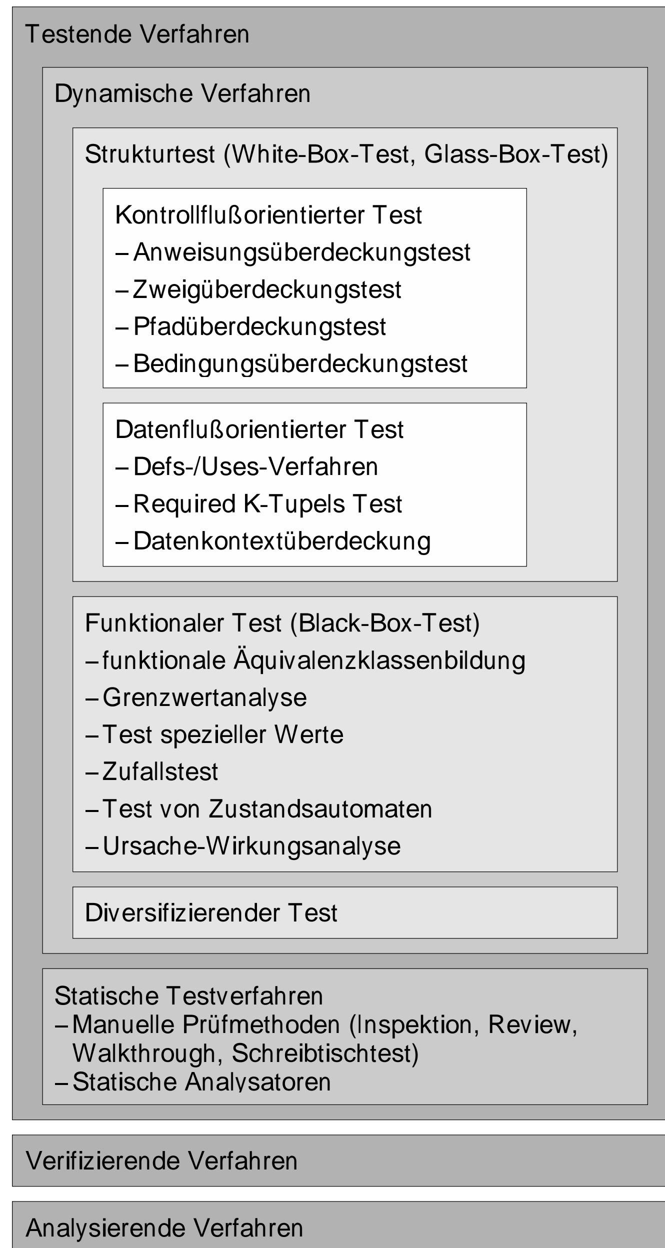

Abbildung 2.4: Klassifikation analytischer QM-Maßnahmen

Wichtige kontrollflußorientierte Testverfahren sind der Anweisungsüberdeckungstest, der Zweigüberdeckungstest, der Pfadüberdeckungstest und der Bedingungsüberdeckungstest.[19] Der Anweisungsüberdeckungstest, auch C_0-Test[20] genannt, verlangt die Ausführung aller Anweisungen, d.h. aller Knoten des Kontrollflußgraphen. Im Zweigüberdeckungstest oder

[19]Für eine detaillierte Beschreibung dieser Verfahren vgl. [Bal98], S. 400ff.
[20]C = Coverage, engl. für Überdeckung

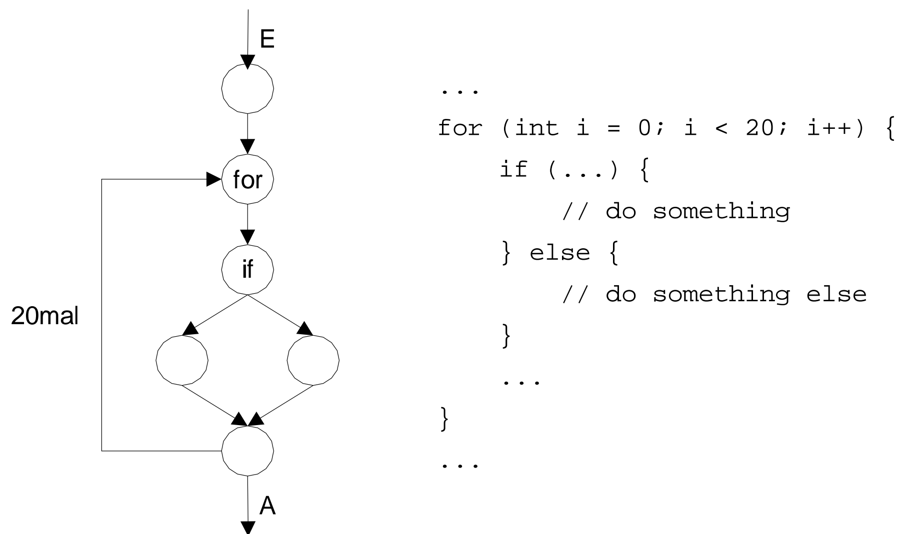

Abbildung 2.5: Kontrollflußgraph mit 2^{20} Pfaden

C_1-Test müssen alle Zweige, d.h. alle Kanten des Kontrollflußgraphen, ausgeführt werden. Der Pfadüberdeckungstest fordert die Ausführung aller unterschiedlichen Pfade des Testlings. Da diese Bedingung bei Programmen, die Schleifen enthalten, praktisch nicht zu erfüllen ist, existieren als abgeschwächte Form der boundary-interior-Pfadtest sowie der strukturierte Pfadtest. Beide Varianten schränken die Ausführung der Wiederholungen von Schleifen nach bestimmten Kriterien ein, um durchführbare Pfadüberdeckungstests zu erhalten. Bedingungsüberdeckungstests betrachten die in einer komplexen Bedingung enthaltenen atomaren Bedingungen sowie deren hierarchische Struktur.

> **Beispiel:** Die komplexe Bedingung $(((a < 5) \vee (a > 12)) \wedge (b \leq 3))$ enthält die atomaren Bedingungen $(a < 5)$, $(a > 12)$ und $(b \leq 3)$. Die atomare Bedingung $(b \leq 3)$ steht dabei auf derselben hierarchischen Stufe wie die komplexe Bedingung $((a < 5) \vee (a > 12))$.

Auch hier existieren verschiedene Varianten. Bei der einfachen Bedingungsüberdeckung muß jede atomare Bedingung mindestens einmal wahr und einmal falsch sein. Die Mehrfach-Bedingungsüberdeckung versucht, alle Variationen der atomaren Bedingungen zu bilden. Wie die komplexe Teilbedingung im gerade angeführten Beispiel zeigt, ist dies nicht immer möglich. Die minimale Mehrfach-Bedingungsüberdeckung berücksichtigt daher hierarchische Strukturen der Bedingungen. Hier muß jede Bedingung - ob atomar oder nicht - mindestens einmal wahr und einmal falsch sein.

2.2.2 Funktionale Testverfahren

Bei funktionalen Tests, die auch als Black-Box-Tests bezeichnet werden, zieht der Tester keine Informationen über die Struktur des Testlings heran.[21] Als Informationsquelle sollten dem Tester lediglich die Produktbeschreibung, die Benutzerdokumentation und die Installationsanweisungen zur Verfügung stehen.[22] Ein vollständiger Test umfaßt jede Kombination zulässiger sowie unzulässiger Eingabewerte und ist daher praktisch nicht durchführbar.

Die Begründung für die Durchführung funktionaler Tests ist, daß das Bestehen von Strukturtests nicht garantieren kann, daß die vorgegebene Spezifikation korrekt umgesetzt wurde. Ziel des funktionalen Tests ist also eine möglichst vollständige aber redundanzarme Prüfung der spezifizierten Funktionalität. Ein Problem besteht darin, daß die Spezifikation häufig nur in einer semi- oder informalen Form vorliegt, die noch einigen Interpretationsspielraum bietet. Dies erschwert die Beurteilung der Vollständigkeit eines Funktionstests und der Korrektheit der erzeugten Ausgaben sowie die Definition von Testfällen.

Die Hauptschwierigkeiten des funktionalen Testens bestehen in der Auswahl geeigneter Testfälle. Da ein vollständiger Funktionstest im allgemeinen nicht durchführbar ist, muß das Ziel sein, bei der Auswahl der Testfälle die Wahrscheinlichkeit zu erhöhen, Fehler zu finden. Folgende funktionale Testverfahren sollen im Anschluß kurz beschrieben werden:[23]

- die funktionale Äquivalenzklassenbildung,
- die Grenzwertanalyse,
- der Test spezieller Werte und
- der Zufallstest.

Bei der funktionalen Äquivalenzklassenbildung werden Definitionsbereiche der Eingabeparameter und Wertebereiche der Ausgabeparameter in Äquivalenzklassen zerlegt. Voraussetzung ist die Annahme, daß der Testling für alle Werte aus derselben Äquivalenzklasse gleich reagiert. Dieses Verfahren bildet gleichzeitig die Grundlage für die Grenzwertanalyse. Diese basiert auf der Erfahrung, daß Testfälle, welche die Grenzen der Äquivalenzklassen abdecken, besonders effektiv sind, d.h. besonders häufig Fehler finden. Bei der Auswahl von Eingabedaten kann dabei eine Näherung an die Grenzen der Äquivalenzklasse sowohl vom

[21] Vgl. [Wal01], S. 228

[22] Daneben gibt es noch sog. „Grey-Box-Tests“. Dies sind funktionale Tests, bei denen auch Informationen über Implementierungsdetails benutzt werden.

[23] Vgl. [Bal98], S. 426ff.

gültigen als auch vom ungültigen Bereich erfolgen.

„Test spezieller Werte“ beschreibt einen Ansatz, bei dem aus der Erfahrung heraus fehlersensitive Testfälle erstellt werden. Die Grundidee besteht darin, Testfälle aus einer Liste möglicher Fehler oder Fehlersituationen abzuleiten. Genau genommen ist die Unterordnung dieses Verfahrens unter die funktionalen Testverfahren nicht richtig, da die Testfälle nicht unbedingt aus der Spezifikation abgeleitet werden. Allerdings werden die Verfahren oft in Kombination mit anderen Strategien wie der Äquivalenzklassenanalyse eingesetzt. Auch die Grenzwertanalyse fällt in die Kategorie „Test spezieller Werte“.

Bei den Zufallstests handelt es sich um ein Verfahren, das aus den Wertebereichen der Eingabedaten zufällig Testfälle erzeugt. Diesem Verfahren liegt also keine deterministische Strategie zugrunde, was allerdings seine Stärke ist, da viele Tester dazu neigen, Testfälle zu erzeugen, die auch bei der Implementierung des Programms berücksichtigt wurden und für die der Testling folglich gemäß der Spezifikation reagiert. Wegen des fehlenden Determinismus sollte dieses Verfahren allerdings nur ergänzend zu anderen Verfahren eingesetzt werden.

2.2.3 Kombinierte Funktions- und Strukturtests

Sowohl Funktionstests als auch Strukturtests haben Nachteile. Es ist daher nicht ausreichend, nur ein Testverfahren einzusetzen. Mit Strukturtests kann z.B. nicht überprüft werden, ob die Funktionalität korrekt und vollständig implementiert wurde. Ein Funktionstest hingegen ist nicht in der Lage, die konkrete Implementierung zu berücksichtigen. Die Spezifikation besitzt ein höheres Abstraktionsniveau als die Implementierung. Da die Testfälle aus der Spezifikation abgeleitet werden, erfüllt ein Funktionstest in der Regel nicht die Minimalanforderungen eines Strukturtests. Daher sollten Funktions- und Strukturtestverfahren geeignet miteinander kombiniert werden.[24]

Der Test besteht dann aus zwei Schritten. Zunächst wird ein Funktionstest durchgeführt, bei dem die Überdeckung im Hintergrund mitprotokolliert wird. In diesem Schritt werden also der Funktions- und Leistungsumfang sowie funktionsorientierte Sonderfälle systematisch geprüft. Anschließend erfolgt ein Strukturtest. Dazu wird die durch den Funktionstest erzielte Überdeckungsstatistik betrachtet und ausgewertet. Die Ursachen für nicht überdeckte Zweige, Pfade oder Bedingungen müssen ermittelt und entsprechende Testfälle erstellt

[24] Vgl. [Bal98], S. 435ff.

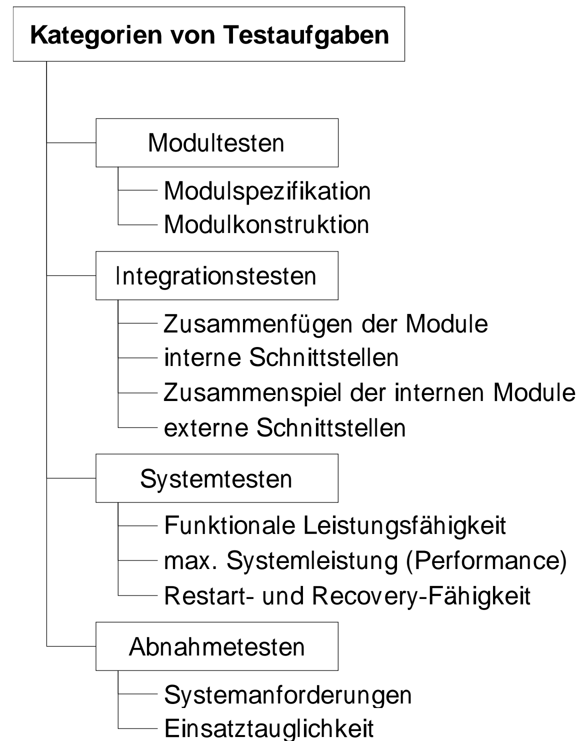

Abbildung 2.6: Kategorien von Testaufgaben mit Merkmalen. *Quelle: [Wal01], S. 249*

werden. Nicht ausführbare Zweige, Pfade oder Bedingungen sind zu entfernen. Der Strukturtest wird beendet, wenn ein vorher festgelegter Überdeckungsgrad erreicht ist. Werden anschließend Fehler im Testling korrigiert, erfolgt eine erneute Iteration der beiden Schritte.

2.2.4 Modul-, Integration-, System- und Abnahmetests

Bei allen umfangreichen Testprozessen lassen sich Kategorien von Testaufgaben unterscheiden. Man kann vom Testen im Kleinen und im Großen sprechen. Beim Testen im Kleinen werden elementare Testobjekte definiert und getestet. Die entsprechende Testaufgabe wird als Modultest bezeichnet. Beim Testen im Großen werden elementare Testobjekte zu größeren Einheiten zusammengefügt und getestet. Die erste dieser Testaufgaben wird als Integrationstest bezeichnet. Wird das System als Ganzes betrachtet und werden Systemleistung und -fähigkeiten untersucht, spricht man vom Systemtesten. Beim Abnahmetesten betrachtet man als Testobjekt das gesamte System in seiner Einsatzumgebung und konzentriert sich auf die Aspekte Systemanforderungen und Einsatztauglichkeit. Abbildung 2.6 gibt einen Überblick über die verschiedenen Kategorien von Testaufgaben und ihre jeweiligen Merkmale.

2.2.4.1 Modultest

Der Modultest, der in der Literatur auch oft als Komponententest bezeichnet wird, ist das Testen der kleinsten Programmeinheiten.[25] Dieser Test wird meistens vom Modulentwickler selbst durchgeführt. Der Testentwurf sollte von der Annahme ausgehen, daß das implementierte Modul dem Modulentwurf widerspricht. Typische Testaspekte, die für die Aufstellung von Testfällen relevant sind, umfassen die Funktionen des Moduls, die Modulstruktur, Ausnahmebedingungen, Sonderfälle sowie die Performance. Voraussetzungen für den Modultest sind die Modulspezifikation, der Feinentwurf und das Programmlisting.

2.2.4.2 Integrationstest

Nachdem die ersten Module freigegeben sind, kann mit der Integration der Module zu größeren Einheiten begonnen werden. Beim Integrationstesten geht man davon aus, daß die einzelnen Module bereits gut getestet sind. Der Schwerpunkt eines Integrationstests liegt daher auf dem Testen der Modulschnittstellen und der Modulkommunikation. Ein wichtiger Aspekt hierbei ist die gewählte Integrationsstrategie, welche einen großen Einfluß auf die Zahl der benötigten Testtreiber und Platzhalter hat. Testtreiber werden benötigt, um auf Module zugreifen zu können, die nicht direkt über die Benutzerschnittstelle aufgerufen und mit Parametern versorgt werden können. Platzhalter (sog. „dummies“ oder „stubs“) simulieren Module, die nicht für den Test benutzt werden können, da sie z.B. noch nicht implementiert sind. Platzhalter werden benötigt, falls ein zu testendes Modul ein solches Modul aufruft.

Integrationsstrategien können in vorgehens- und testzielorientierte sowie in inkrementelle und nicht-inkrementelle Strategien eingeteilt werden.[26] Bei nicht inkrementellen Integrationsstrategien werden alle oder eine größere Anzahl von Modulen gleichzeitig integriert. Der große Vorteil dieser Variante ist, daß keine Testtreiber und Platzhalter benötigt werden. Die Nachteile sind, daß zum Integrationszeitpunkt alle Module fertiggestellt und überprüft sein müssen und daß Fehler schwer zu lokalisieren sind, da viele Module zusammenwirken. Ferner ist es schwierig, Testfälle zu konstruieren, die eine bestimmte Testüberdeckung gewährleisten. Aus diesem Grund sind nicht-inkrementelle Strategien für große und umfangreiche Systeme nicht geeignet.

Bei inkrementellen Integrationsstrategien werden die Module einzeln oder in sehr kleinen Gruppen integriert. Dies bringt den Vorteil, daß Module integriert werden können, sobald sie fertiggestellt und überprüft sind. Testfälle lassen sich leichter konstruieren. Der Nachteil ist,

---

[25] Vgl. [Wal01], S. 248ff.

[26] Vgl. [Bal98], S. 505ff.

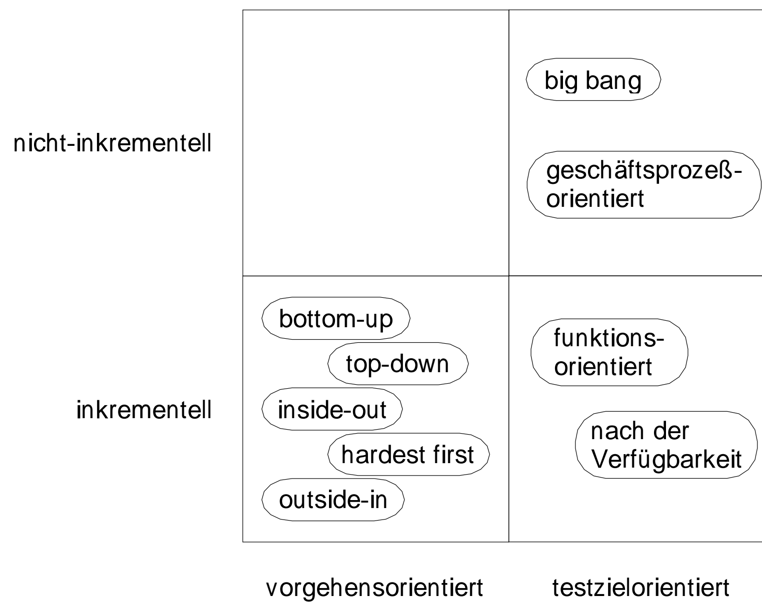

Abbildung 2.7: Überblick über Integrationsstrategien. *Quelle: [Bal98], S. 506*

daß unter Umständen eine große Anzahl von Testtreibern oder Platzhaltern erstellt werden muß.

Bei testzielorientierten Integrationsstrategien werden Testfälle ausgehend von den Testzielen erstellt. Für die Überprüfung der Testfälle werden dann die benötigten Module zu einem Subsystem integriert. Vorgehensorientierte Integrationsstrategien leiten die Integrationsreihenfolge aus der Systemarchitektur ab. Abbildung 2.7 zeigt die Einordnung üblicher Integrationsstrategien nach den gerade beschriebenen Kriterien.

Bei der big-bang-Integration werden alle Module eines Systems gleichzeitig integriert. Der Integrationsprozeß läuft dabei unstrukturiert und unsystematisch ab. Es ist schwierig, geeignete Testfälle zu erstellen und auftretende Fehler zu lokalisieren. Bei der geschäftsprozeßorientierten Integration werden alle Module integriert, die von einem Geschäftsprozeß betroffen sind. Bei der funktionsorientierten Integration werden funktionale Testfälle spezifiziert, und die betroffenen Module werden schrittweise integriert und getestet. Bei der Integration nach Verfügbarkeit werden Module nach Abschluß ihrer Überprüfung integriert. Der Integrationsprozeß wird somit durch die Implementierungsreihenfolge festgelegt.

Vorgehensorientierte Integrationsstrategien hängen von der Software-Architektur und von der Vorgehensweise im Entwurf ab. Was die Software-Architektur betrifft, so lassen sich mei-

stens Baumstrukturen oder Schichtenstrukturen identifizieren. Bei der top-down-Integration werden die Module ensprechend der Baumhierarchie oder Schichtenstruktur von oben nach unten schrittweise integriert, bei der bottom-up-Integration von unten nach oben. Bei der outside-in-Strategie wird gleichzeitig auf der höchsten und der niedrigsten Schicht begonnen. Ausgangspunkt bei der inside-out-Integration sind die mittleren Architekturschichten. Schrittweise werden dann sowohl Module höherer als auch niedrigerer Schichten integriert. Bei der hardest-first-Strategie werden zunächst die kritischsten, d.h. besonders schwer zu testenden oder potentiell fehlerbehafteten, Module integriert. Bei jedem weiteren Integrationsschritt werden diese Module indirekt mitüberprüft, so daß sie die am besten getesteten Module des Systems sind. Die vorgehensorientierten Strategien unterscheiden sich in der Anzahl der benötigten Testtreiber und Platzhalter sowie im Schwierigkeitsgrad der Testfallerstellung. Ferner können verschiedene Fehlerarten mit den verschiedenen Integrationsstrategien unterschiedlich gut und zu unterschiedlichen Zeitpunkten festgestellt werden.[27]

2.2.4.3 Systemtest

Nachdem alle Module gemäß der Intergrationsstrategie integriert und einem Integrationstest unterzogen wurden, erfolgt der Systemtest. Bei diesem ist nur noch das „Äußere“ des Systems sichtbar, d.h. die Benutzerschnittstelle und andere externe Schnittstellen. Der Systemtest ist der abschließende Test der Software-Entwickler und Qualitätsicherer in der realen Umgebung. Er findet ohne den Auftraggeber statt. Basis für den Systemtest ist die Produktdefinition. Für den Systemtest sind die folgenden Prüfziele zu berücksichtigen.[28] Für einige dieser Prüfziele wurden die in der Literatur gängigen Bezeichnungen in Klammern vermerkt.

- **Vollständigkeit**
  Es ist zu prüfen, ob das System alle funktionalen und nicht-funktionalen Anforderungen erfüllt. Die funktionalen Anforderungen werden durch einen Funktionstest geprüft.

- **Volumen**
  Das System ist mit umfangreichen Daten zu testen (Massentest, volume test). Die Größe von Dateien und Datenbanken kann eine potentielle Schwachstelle sein.

- **Last**
  Das System ist über einen längeren Zeitraum hinweg unter Spitzenbelastungen zu testen (Lasttest, load test).

[27] Vgl. [Bal98], S. 508ff.
[28] Vgl. [Wal01], S. 251ff.

- **Effizienz**
  Es ist zu prüfen, ob die Antwortzeiten und Durchsatzraten bei hoher Systembelastung den spezifizierten Anforderungen entsprechen (Zeittest).

- **Fehlertoleranz, Robustheit**
  Das System wird unter Überlast getestet (Streßtest, stress test). Beispielsweise wird ein Plattenausfall simuliert oder der Arbeitsspeicher reduziert. Es ist zu prüfen, wie das Leistungsverhalten ist und ob das System nach Rückgang der Überlast wieder in den Normalbereich zurückkehrt.

- **Benutzbarkeit**
  Die Verständlichkeit, Erlernbarkeit und Bedienbarkeit aus Sicht des Benutzers wird getestet (Benutzbarkeitstest, usability test).

- **Sicherheit**
  Datenschutzmechanismen und Datensicherheitsprüfungen durch das System beziehungsweise durch die umgebende Organisation sind zu überprüfen.

- **Konfiguration**
  Falls das System für den Einsatz auf unterschiedlichen Hard- und Softwareplattformen geeignet und konfigurierbar ist, müssen die verschiedenen Möglichkeiten gestestet werden (Konfigurationstest).

- **Interoperabilität**
  Muß das System aufgrund der Anforderungen mit anderen Systemen zusammenarbeiten, ist die Kompatibilität der entsprechenden Schnittstellen und der Daten zu prüfen (Interoperabilitätstest).

- **Dokumentation**
  Es ist das Vorhandensein, die Güte und die Angemessenheit der Benutzer- und Wartungsdokumentation zu prüfen (Dokumentenprüfung).

- **Wartbarkeit**
  Es sind Prüfungen aller Anforderungen hinsichtlich der Wartbarkeit durchzuführen.

Der Systemtest beinhaltet oft noch einen Installations- und Wiederinbetriebnahmetest.[29] Beim Installationstest wird geprüft, ob das System mit der Installationsbeschreibung installiert und in Betrieb genommen werden kann. Beim Wiederinbetriebnahmetest wird geprüft, ob das System nach einem Zusammenbruch anhand der vorliegendenden Beschreibungen

[29] Vgl. [Bal98], S. 542

wieder in Betrieb genommen werden kann und ob danach noch alle Daten verfügbar sind.

Bei allen aufgeführten Testverfahren ist darauf zu achten, daß die Durchführung von Regressionstests ermöglicht werden sollte. Die Testfälle können beim Abnahmetest und nach Wartungsarbeiten wiederholt werden. Bei einer evolutionären oder inkrementellen Softwareentwicklung werden die gespeicherten Testfälle der Vorversion mit Soll-/Ist-Ergebnisvergleich nochmals ausgeführt.

2.2.4.4 Abnahmetest

Der Abnahme- oder Akzeptanztest ist eine besondere Ausprägung des Systemtests.[30] Hier wird das System unter Mitarbeit des Auftraggebers in seiner realen Einsatzumgebung getestet. Ziel des Abnahmetests ist es, zu demonstrieren, daß das Vertrauen in das Produkt für den Einsatz beim Kunden oder Auftraggeber gerechtfertigt ist. Basierend auf dem Vertrag mit dem Auftraggeber werden dazu Testfälle erzeugt, die zeigen sollen, daß der Vertrag nicht erfüllt ist. Sind die Testfälle nicht erfolgreich, so wird das System abgenommen. Die Abnahmekriterien sind dabei bereits in der Produktdefinition bzw. im Vertrag mit dem Auftraggeber festzulegen. Die Testfälle weisen meistens folgende Charakteristika auf:

- Abnahmekriterien aus der Produktdefinition,
- Teilmengen der Testfälle des Systemtests,
- Testfälle für die Verarbeitung der Geschäftsvorgänge einer typischen Zeit- oder Abrechnungsperiode,
- Testfälle für Dauertests mit dem Ziel, einen kontinuierlichen Betrieb über eine größere Zeitspanne zu prüfen.

Bei einem Produkt für den anonymen Markt nimmt der interne Auftraggeber das System ab.[31] Das System wird zusätzlich noch einem Alpha- und/oder Beta-Test unterzogen. Beim Alpha-Test wird das System in der Umgebung des Herstellers durch Anwender erprobt. Beim Beta-Test wird das System ausgewählten Kunden in deren Umgebung zur Probenutzung zur Verfügung gestellt. Sind größere Änderungen durchzuführen, kann nach den Regressionstests der Beta-Test wiederholt werden.

[30] Vgl. [Wal01], S. 253f.
[31] Vgl. [Bal98], S. 544

2.2.5 Testprozeß und -dokumentation

Tests müssen im Rahmen der übergeordneten Qualitätssicherung mit Qualiätsplanung, Qualitätslenkung, und Qualitätsprüfung systematisch durchgeführt und geeignet dokumentiert werden. Der Testprozeß sollte mindestens aus drei Schritten bestehen:[32]

- Testplanung,
- Testdurchführung und
- Testkontrolle.

Die Ergebnisse der Testplanung sind[33]

- der Testplan,
- die Testentwurfsspezifikation,
- die Testfallspezifikation und
- die Testvorgehensspezifikation für jede Testaufgabe.

Im Testplan werden die Zielsetzungen des Testens auf einer hohen Abstraktionsstufe beschrieben. Ferner wird die Teststrategie bzw. das Testkonzept dokumentiert. Dies beinhaltet die Testobjekte sowie die zu testenden und die nicht zu testenden Merkmale dieser Objekte. Außerdem beschreibt das Testkonzept die Art der durchzuführenden Tests sowie die Vorgehensweise bei der Durchführung. Bei umfangreichen Projekten kann der Testplan zudem in einen Haupttestplan bzw. Master-Testplan und mehrere Testpläne für die verschiedenen Arten durchzuführender Tests aufgespaltet werden. Der Haupttestplan enthält dann nur allgemeine Angaben zum Testkonzept. Ein weiterer wichtiger Gliederungspunkt des Testplans ist die zeitliche und personelle Einplanung von Testaktivitäten. Außerdem enthält der Testplan eine Aufstellung der allgemeinen Anforderungen an die Hard- und Softwareumgebung und an die Dokumentation.

Testentwurfsspezifikation, Testfallspezifikation und Testvorgehensspezifikation werden in der Literatur auch unter dem Oberbegriff „Testvorschrift“ zusammengefaßt. In diesem Dokument wird die Teststrategie präzisiert. Der Zweck und Umfang des konkreten Tests müssen dokumentiert werden. Ferner sollte dieses Dokument einen Abschnitt über die Testvoraussetzungen hinsichtlich Hard- und Software, Testdaten und Personalbedarf enthalten.

[32] Vgl. [Bal98], S.548
[33] Vgl. [Wal01], S. 242

Unerläßlich ist die Festlegung von Abnahmekriterien. In diesem Abschnitt sollten Kriterien für den Erfolg, den Abbruch, die Unterbrechung und die Wiederaufnahme eines Testlaufes festgehalten werden.

Das Ergebnis der Testdurchführung ist der Testbericht, in dem die Ergebnisse der Testaktivitäten zusammengefaßt werden und in dem bewertet wird, ob die gesetzten Ziele und Anforderungen erreicht wurden. Er bildet die Basis für die Testkontrolle und besteht aus der Liste der vom Test betroffenen Software-Einheiten, der Liste der Problemmeldungen, dem Testprotokoll und der Testzusammenfassung.[34] Das Testprotokoll beschreibt in chronologischer Reihenfolge alle relevanten Details der durchgeführten Tests. Die Testzusammenfassung enthält unter anderem eine Bewertung des Erfolgs der Tests.

Für die Testdokumentation gibt es mehrere ANSI/IEEE-Normen. Diese Normen sind allerdings zur Dokumentation von Software für kritische Anwendungen gedacht und sind dementsprechend umfangreich und teilweise kompliziert. Allerdings bieten sie einen guten Anhaltspunkt für selbst zu erstellende Dokumente. Die ANSI/IEEE-Norm 829 legt den Inhalt und die Struktur von Testdokumenten fest.[35] Eine Beschreibung der nach dieser Norm zu verwendenden Dokumente findet sich in [Wal01], S. 255ff. Beispielgliederungen für Testpläne und verwandte Dokumente finden sich unter anderem in [Bal98], S. 549f. sowie in [IRM02].

Auch der Testprozeß ist im Rahmen der Qualitätssicherung zu prüfen, zu bewerten sowie einer regelmäßigen Kontrolle zu unterziehen. Daher sind geeignete Merkmale für Qualitätsprüfungen aufzustellen. Die wichtigsten Merkmale für die Qualität des Testens sind[36]

- der Spezifikationsbezug,
- die Reproduzierbarkeit und
- die Nachvollziehbarkeit für Projekt-Außenstehende.

Die Definition des Testziels und die Testfälle sind aus der Spezifikation abzuleiten. Jeder Testfall muß wiederholbar sein. Daher muß zu jedem Testfall eine exakte Vorschrift zur Durchführung und Auswertung sowie ein erwartetes Testergebnis vorliegen. Für Außenstehende muß eine verständliche Dokumentation existieren. Die Testfälle sollten zudem selbst-

[34] Vgl. [Wal01], S. 257, und [Bal98], S.548
[35] Vgl. [ANSb]
[36] Vgl. [Wal01], S. 242f.

erklärend sein, und die Testausführung muß vollständig beschrieben sein.

Eine weitere wichtige Aufgabe der Testplanung ist die Festlegung von Vollständigkeits- und Endekriterien für den Testprozeß.[37] Schlechte Kriterien sind die Begrenzung des Testprozesses auf einen bestimmten Zeitraum oder der Abbruch des Testprozesses, falls keine Fehler mehr entdeckt werden. Grundsätzlich können zwei Möglichkeiten zur Beendigung des Testprozesses in Betracht gezogen werden. Der Testprozeß kann entweder abgebrochen werden, falls eine vorgegebene Zahl von Fehlern gefunden wurde, oder falls die Fehlerrate, also die Anzahl gefundener Fehler pro Zeiteinheit, unter einen vorgegebenen Wert sinkt. Im ersten Fall orientiert man sich an Erfahrungswerten aus vergangenen Projekten, um die Gesamtzahl der Fehler abzuschätzen. Nachdem die Effektivität des Testens in Abhängigkeit von der Testaufgabe (z.B. Modultest, Integrationstest, etc.) geschätzt wurde, können für jede Testaufgabe die Anzahl der zu findenden Fehler abgeschätzt werden. Im zweiten Fall kann man die Fehlerrate in einem Diagramm eintragen, aufgrund dessen man feststellen kann, wann die Effizienz des Testens erschöpft ist. Beide Möglichkeiten setzen voraus, daß im Testfallentwurf eine geeignete Kombination der verschiedenen Testmethoden berücksichtigt wurde. Für die zweite Möglichkeit wird außerdem ein Fehlererfassungssystem benötigt.

2.2.6 Grundsätze für den Software-Test und Probleme in der Praxis

Nachdem in diesem Abschnitt die theoretischen Grundlagen des Software-Testens beschrieben wurden, möchten wir abschließend Grundsätze für die Durchführung von Software-Test anführen und auf einige häufig auftretende Probleme in der Praxis eingehen. [Wal01] formuliert auf S. 223ff. folgende Grundsätze für den Software-Test:

- Testen dient nicht dazu, die Fehlerfreiheit der Software zu zeigen, sondern Testen ist der Prozeß, ein Programm mit der Absicht auszuführen, Fehler zu finden.
- Mit jedem Test müssen sinnvolle und quantifizierbare Ziele erfüllt werden.
- Testziele sind nur dann akzeptabel, wenn sie auch quantifizierbar und meßbar sind.
- Für jeden Test muß ein Endekriterium vorgesehen werden.
- Es ist praktisch nicht möglich, ein Programm vollständig zu testen. Daher muß ökonomisch getestet werden, d.h. gerade soviel wie nötig.
- Für jeden Testfall müssen die erwarteten Testergebnisse a priori spezifiziert werden.

[37] Vgl. [Wal01], S. 246f.

- Die Ergebnisse eines jeden Tests müssen sorgfältig überprüft werden.
- Der Entwickler ist für die Tests der einzelnen Module verantwortlich.
- Integrations- und Systemtests werden von einer unabhängigen Testgruppe durchgeführt, die aber eng mit den Entwicklern zusammenarbeitet.
- Testfälle müssen immer auch unerwartete ungültige Eingabedaten berücksichtigen.
- Testfälle und ihre Ergebnisse sind zu sammeln und zu archivieren. Insbesondere sind „Wegwerf-Testfälle“ zu vermeiden.
- Ein Testfall ist dann als erfolgreich anzusehen, wenn er einen bisher unbekannten Fehler entdeckt.
- Der Testplan sollte immer unter der Annahme aufgestellt werden, daß Schwierigkeiten beim Testen auftreten.
- Testen muß reproduzierbar sein.
- Module, in denen bereits Fehler gehäuft aufgetreten sind, müssem besonders sorgfältig getestet werden.

Einige Probleme, die beim Testen von Software auftreten, können direkt aus den obigen Grundsätzen abgeleitet werden. Entwickler, die ihre Module testen, gehen oft davon aus, daß die von ihnen entwickelte Software keine oder nur sehr wenige Fehler hat. So ist es schwierig für sie, Testfälle zu definieren, die unerwartete ungültige Werte enthalten, da sie davon ausgehen, daß sie bereits alle denkbaren Eingaben berücksichtigt haben. Weitere Probleme können bei der Einbindung der unabhängigen Testgruppe entstehen. Dieser Aspekt wurde bereits unter dem Begriff „Prinzip der unabhängigen Qualitätsprüfung“ diskutiert.[38]

Ein weiteres Problemfeld ist die Quantifizierbarkeit von Testzielen. Zwar ist es leicht, für Strukturtests bestimmte Abdeckungsgrade zu definieren, für Kriterien wie „Benutzbarkeit“ ist es allergings schwierig, ein Testziel zu quantifizieren. Ferner ist es problematisch, Testendekriterien zu definieren, da gegenwärtig theoretisch bestimmbare Kriterien fehlen. Dies impliziert wiederum Probleme beim „ökonomischen Testen“. Die einzige Möglichkeit hierbei ist, heuristische Ansätze zur Quantifizierbarkeit von Testzielen und Festlegung von Testendekriterien zu wählen.

---

[38] Vgl. S. 13

Das dritte Problemfeld ist im Bereich des Testprozesses und der Testdokumentation zu sehen. Probleme treten dort insbesondere dann auf, wenn keine dedizierte Testgruppe existiert und wenn wie im Beispiel von FlexiTrust eine hohe Mitarbeiterfluktuation herrscht und neue Mitarbeiter in der Regel wenig Erfahrung im Testen von Software haben. Aufgrund des schon bestehenden Zeitdrucks sind viele Mitarbeiter nicht bereit, auch noch Zeit für die Einarbeitung in die Abläufe des Testprozesses und für das Erlernen von Qualitätssicherungsmaßnahmen aufzubringen. Daher muß der Testprozeß für diese Mitarbeiter so einfach und überschaubar wie möglich gehalten werden. Zur Unterstützung des Testprozesses empfiehlt sich der Einsatz einfach zu bedienender Werkzeuge und Checklisten. Besonders sinnvoll ist auch der Einsatz von Dokumenten, welche die Anwendung von Richtlinien, Standards, Werkzeugen und Qualitätssicherungsmaßnahmen am Beispiel von einfachen Anwendungsfällen erläutern (sog. HowTos oder Cookbooks). Was die Dokumentation betrifft, sollten zudem für alle verwendeten Dokumenttypen Schablonen existieren, welche nur noch ausgefüllt werden müssen. Hierbei ist darauf zu achten, daß neben diesen Schablonen auch bereits fertig ausgefüllte Beispiele existieren. Zudem sollte bei allen eingesetzten Maßnahmen auf eine umfassendere Funktionalität, Detailgetreue, etc. verzichtet werden, falls dadurch die Komplexität so weit steigt, daß die Akzeptanz unter den Benutzern auf ein zu niedriges Niveau fällt.

Kapitel 3

Entwurf des Frameworks

In diesem Kapitel soll der Entwurf eines Frameworks vorgestellt werden, welches dazu benutzt werden kann, formularbasierte Web-Anwendungen zu testen. Der Entwurf orientiert sich an den Bedürfnissen von FlexiTrust, ist aber so allgemein gehalten, daß auch andere Web-Anwendungen getestet werden können. Bevor der Entwurf vorgestellt wird, sollen zunächst einige Vorüberlegungen getroffen sowie der Anwendungsbereich des Frameworks genau definiert werden.

3.1 Vorüberlegungen

Grundsätzlich stellt sich die Frage, warum man ein neues Framework entwickeln soll, wenn doch bereits eine große Anzahl von Werkzeugen zur Automatisierung von Software-Tests exisistiert.[1] In Gesprächen mit Mitarbeitern des Fachgebiets sowie in Diskussionen in Newsgroups wie „comp.software.testing“ wurden jedoch immer wieder Nachteile solcher Werkzeuge genannt:

- Die meisten Werkzeuge sind nicht frei verfügbar.
- Die Einarbeitungszeit ist oft zu hoch.
- Kein Werkzeug deckt alle Bedürfnisse ab.
- Die meisten Werkzeuge sind zu unflexibel, was Änderungen in der zu testenden Anwendung betrifft.

[1]Eine Zusammenstellung verschiedener Werkzeuge zum Testen von Java-Anwendungen findet sich z.B. in [Kle01] oder [BS01].

Bei den nicht frei verfügbaren Werkzeugen fallen Lizenzkosten an, die in der Regel für ein Projekt wie FlexiTrust nicht akzeptabel sind. Für Werkzeuge, die sich vom angebotenen Funktionsumfang eignen würden, ist die Einarbeitungszeit zu hoch, da sie zu komplex sind. Viele Werkzeuge decken jedoch nur sehr spezielle Anwendungsbereiche ab, wie z.B. Lasttests, so daß der Einsatz von mehreren Werkzeugen notwendig wäre. Insbesondere Werkzeuge zum Aufzeichnen und Wiedereinspielen von Benutzereingaben (sog. Capture/Replay-Tools) sind in der Regel zu unflexibel, was Änderungen in der zu testenden Anwendung betrifft.

3.2 Anwendungsbereich

Bevor das Framework beschrieben wird, muß zunächst der Begriff der „formularbasierten Web-Anwendung“ geklärt werden. Eine solche Anwendung zeichnet sich aus durch

- die Ausführung der Anwendung in einem Web-Browser,
- dynamisch aufgebaute HTML-Formulare als Benutzerschnittstelle,
- eine Anbindung an externe Datenspeicher und andere Systeme,
- asynchrone Dienste/automatische Funktionen.

Web-Anwendungen werden in einem Web-Browser ausgeführt. Es handelt sich also um Client/Server-Anwendungen, bei denen der Anwendungsserver ein Web-Server und die Clients Web-Browser sind. Die Anwendung basiert auf Internettechnologien, d.h. die Kommunikation findet über HTTP und die Interaktion mit dem Benutzer über vom Web-Server ausgelieferte HTML-Seiten statt. Formularbasierte Web-Anwendungen in unserem Sinne sind Web-Anwendungen, bei denen die zum Client ausgelieferten HTML-Seiten aus HTML-Formularen bestehen. Die HTML-Formulare werden von Benutzern der Clients ausgefüllt und an den Web-Server zurückgeschickt, welcher danach mit einer HTML-Seite antwortet, die wiederum HTML-Formulare enthalten kann. Der Inhalt der HTML-Formulare wird dynamisch generiert, wobei zur Generierung auf die Inhalte von angebundenen externen Datenspeichern[2] zurückgegriffen wird. Auch die Benutzereingaben, d.h. die Informationen, die ein Benutzer in ein Formular eingetragen hat, werden in externen Datenspeichern abgelegt. Daneben kann auch eine Anbindung an weitere Systeme existieren, z.B. an einen Verzeichnisdienst. Ferner werden einige Funktionen der Anwendung automatisch oder asynchron

[2] Z.B. Datenbanksysteme, aber auch Dateien o.ä.

ausgeführt. Im Beispiel von FlexiTrust wird nach dem Ausfüllen eines bestimmten Formulars ein Zertifikat generiert, welches anschließend dem Benutzer, der das Formular ausgefüllt hat, per E-Mail zugeschickt oder auf einem LDAP-Server veröffentlicht wird.

Das Framework soll Mechanismen zur Verfügung stellen, die es erlauben, ein Testsystem zu implementieren, mit dem Teile einer Anwendung automatisch getestet werden können. Ziel ist es nicht, den gesamten Testprozeß bestehend aus Testplanung, -durchführung und -kontrolle[3] abzubilden. Dies ist u.a. aus folgenden Gründen nicht möglich:

- Die Testplanung kann nicht vollautomatisch erfolgen, da die Zielsetzung des Testens und das Testkonzept manuell erstellt werden müssen.
- Nicht alle Tests können oder sollen automatisiert werden. Teilweise kann die Automatisierung eines Tests zu komplex sein, so daß der Aufwand in keinem Verhältnis zum Nutzen steht.
- Kontrollmaßnahmen können nicht automatisch durchgeführt werden. Lediglich die Information, welche Kontrollmaßnahmen ausgelöst werden müssen, kann automatisch aus den Ergebnissen der Testdurchführung generiert werden.

Daher soll das Framework eine Unterstützung des Testprozesses für alle automatisierten Tests bieten. Da die Automatisierung eines Prozesses nur Sinn macht, wenn dieser sehr oft ausgeführt wird, ist ein wichtiger Anwendungsbereich des Frameworks die Durchführung von Regressionstests.

3.3 Anforderungen

Folgende Anforderungen lassen sich an ein Framework zum Testen formularbasierter Web-Anwendungen und an ein von diesem Framework abgeleitetes Testsystem stellen:

- **Das Framework muß die Ausführung aller Schritte des Testprozesses geeignet unterstützen.**
  Wie bereits gesagt ist eine vollständige Automatisierung des Testprozesses nicht möglich. Für die Testplanung bedeutet dies, daß für automatisierte Tests zumindest die Testfallspezifikation, die Testvorgehensspezifikation und die Festlegung von Abnahmekriterien vollständig unterstützt werden sollten. In der Testdurchführung müssen die automatisierten Tests mit den generierten Testfällen ausgeführt werden. Zudem muß die Analyse der Ergebnisse dieser Testläufe automatisch erfolgen.

[3] Vgl. Unterabschnitt 2.2.5

- **Das aus dem Framework abgeleitete Testsystem muß leicht bedienbar sein.**
  Dies bedeutet insbesondere, daß das Testsystem ohne größere Einarbeitungszeit benutzt werden kann und daß Testläufe auch von Mitarbeitern durchgeführt und ausgewertet werden können, die nicht der Qualitätssicherung angehören, wie im Beispiel von FlexiTrust HiWis des Fachgebietes.

- **Das Testsystem muß einfach und flexibel konfigurierbar sein.**
  Dies bedeutet, daß das Framework einen Konfigurationsmechanismus zur Verfügung stellen muß, der die Einstellung von Parametern wie die Auswahl der zu testenden Formulare, etc. zuläßt.

- **Definitionen von Formularen müssen ohne großen Aufwand hinzugefügt und geändert werden können.**
  Sollen neue Definitionen von Formularen zum Testsystem hinzugefügt oder bereits bestehende Definitionen geändert werden, muß dies möglich sein, ohne den Programmcode zu ändern.

- **Das Generieren von Testfällen muß möglich sein.**
  Das Framework muß einen Mechanismus zur Verfügung stellen, mit dem es möglich ist, Testfälle nach bestimmten Kriterien zu generieren, z.B. Testfälle für eine Grenzwert-Analyse, Testfälle zum Testen zufälliger Eingabewerte oder Testfälle zum Testen spezieller Werte.

- **Das Framework muß einen Mechanismus zum Aufzeichnen von Testfällen zur Verfügung stellen.**
  Eine solche Funktion ist sinnvoll zum Testen bestimmter Eigenschaften, für welche noch kein Testfallgenerator implementiert wurde, bzw. für welche die Implementierung eines Testfallgenerators unverhältnismäßig komplex wäre. Das Framework muß dabei auch einen Mechanismus zur Verfügung stellen, welcher die Aufzeichnung eines Testfalls protokolliert und Anmerkungen des Benutzers über den Erfolg der manuellen Testausführung zuläßt.

- **Testfälle müssen wiederverwendbar sein.**
  Dies bedeutet, daß die generierten und aufgezeichneten Testfälle so gespeichert werden müssen, daß sie bei einem erneuten Testdurchlauf wieder eingespielt werden können. Insbesondere sollen gespeicherte Testfälle auch für neue Versionen eines zu testenden Formulars wiederverwendet werden können. Das Framework muß daher einen Mechanismus zum Konvertieren von Testfällen zur Verfügung stellen.

Abbildung 3.1: Informationsfluß zwischen den Komponenten des Frameworks

- **Die Ergebnisse eines Testlaufs müssen leicht ausgewertet werden können.**
  Das Framework muß daher einen geeigneten Protokollierungsmechanismus zur Verfügung stellen.

- **Das Testsystem muß erweiterbar sein.**
  Außer durch neue Formulare muß das Testsystem insbesondere auch durch zusätzliche Tests oder zusätzliche Testfallgeneratoren erweitert werden können. Dies muß möglich sein, ohne den bestehenden Programmcode zu ändern.

3.4 Komponenten des Frameworks

Das in dieser Arbeit entwickelte Framework besteht aus den in Abbildung 3.1 dargestellten Komponenten. Der Informationsfluß zwischen den Komponenten ist durch Pfeile gekennzeichnet. Teilkomponenten, die sich bezüglich des Informationsflusses gleich verhalten, wurden der besseren Übersichtlichkeit halber durch einen größeren Kasten zusammengefaßt.

In der Komponente „**Datenspeicher**" sind Informationen über zu testende Formulare, Testfälle und zugehörige Meta-Informationen abgespeichert. Außerdem wird hier der Output der Protokollierungskomponente gespeichert. Der Datenspeicher ist eine passive Kom-

ponente. Der Zugriff erfolgt in der Regel lesend. Schreibender Zugriff erfolgt nur, wenn Informationen der Protokollierungskomponente oder neu erzeugte Testfälle gespeichert werden.

Die Komponente „**Protokollierung**“ erhält Informationen von allen Komponenten und legt sie im Datenspeicher ab. Die Konfigurationskomponente bestimmt dabei, welche Informationen wo im Datenspeicher zu speichern sind.

Neue Testfälle können mit der Komponente „**Testfallgenerator**“ erzeugt werden. Zur Laufzeit können verschiedene Implementierungsvarianten dieser Komponente existieren. Mit jeder Variante können Testfälle auf unterschiedliche Art und Weise erzeugt werden. Welche Variante verwendet werden soll, für welche Formulare welche Anzahl von Testfällen zu generieren ist und wo die generierten Testfälle zu speichern sind, wird durch die Konfigurationskomponente bestimmt.

Die Komponente „**Testfallaufzeichnung**“ ist für die Aufzeichnung von Testfällen zuständig. Wie beim Testfallgenerator bestimmt die Konfigurationskomponente, wo im Datenspeicher die aufgezeichneten Testfälle zu speichern sind.

Testfälle können mit der Komponente „**Testfallkonverter**“ in eine neue Version konvertiert werden. Die Konfigurationskomponente bestimmt, welche Testfälle von welcher Version in welche Version konvertiert werden.

Die Komponente „**Testdurchführung**“ ist für die Durchführung und Auswertung von Testläufen zuständig. Da es möglich sein muß, verschiedene Tests durchzuführen, müssen auch verschiedene Implementierungsvarianten dieser Komponente existieren. Die Konfigurationskomponente bestimmt, welche Tests für welche Formulare bzw. Testfälle durchgeführt werden sollen und wo die Ergebnisse gespeichert werden.

In der Komponente „**Ergebnisüberprüfung**“ wird überprüft, ob die Informationen, die während eines Testlaufs in ein Formular eingetragen wurden, auch korrekt weiterverarbeitet wurden. Im Fall von FlexiTrust muß z.B. überprüft werden, ob ein Zertifikat mit entsprechendem Inhalt generiert und ausgeliefert wurde. Da diese Überprüfung je nach Formular und je nach durchgeführtem Test unterschiedlich sein kann, müssen mehrere Implementierungsvarianten dieser Komponente existieren.

Die Komponente „**Steuerung**“ ist für die Ausführung der gerade beschriebenen Komponenten zuständig. Außerdem werden hier noch weitere Funktionen wie die Initialisierung der Testumgebung, das Übersetzen des Quellcodes, das Erstellen von Backups, etc. ausgeführt.

Die Aufgabe der Komponente „**Konfiguration**“ ist es, die einzelnen Komponenten mit den bereits beschriebenen Informationen zu versorgen. Daneben können in dieser Komponente noch Einstellungen vorgenommen werden, welche die Installation der Anwendung betreffen.

Kapitel 4

Implementierung des Frameworks

Das in dieser Arbeit vorgestellte Framework wurde in der Programmiersprache Java von Sun Microsystems implementiert. Gründe für diese Entscheidung sind die Vorteile dieser Sprache (z.B. Plattformunabhängigkeit, Objektorientierung, Anzahl verfügbarer Bibliotheken). Ein weiterer Grund ist, daß dieses Framework zum Testen von Anwendungen im Umfeld des FlexiTrust-Projekts eingesetzt werden soll und daß diese Anwendungen ebenfalls in Java entwickelt wurden. Erweiterungen des Frameworks und insbesondere solche Erweiterungen, welche auf FlexiTrust-Bibliotheken zurückgreifen, sind so einfacher möglich.

Dieses Kapitel gibt einen Überblick über die Implementierung des Frameworks. Im folgenden werden die Implementierung der einzelnen Komponenten, bei der Programmierung getroffene Entscheidungen, sowie Hinweise zur Benutzung der Komponenten und zur Erweiterung des Frameworks beschrieben. Im Rahmen dieser Diplomarbeit war es nicht möglich, alle Mechanismen zu implementieren, die zum Testen von FlexiTrust benötigt werden. Alle Komponenten des Frameworks wurden jedoch soweit implementiert, daß bereits verschiedene Tests sowohl für FlexiTrust als auch für andere Web-Anwendungen durchgeführt werden können. An den entsprechenden Stellen in diesem Kapitel wurde jeweils vermerkt, inwieweit der Funktionsumfang einer Komponente noch eingeschränkt ist.

Als weitere Dokumentationsquellen stehen eine Bedienungsanleitung in Anhang A, sowie die mit dem Java-Dokumentationsgenerator Javadoc erstellte Dokumentation und die entsprechenden Quelldateien selbst zur Verfügung. Die Bedienunganleitung enthält neben einer Beschreibung der Benutzung und Erweiterung des Frameworks unter anderem auch Hinweise zur Installation.

Die mit Javadoc generierte Dokumentation richtet sich an Entwickler, die das Framework

mit selbst erstellten Komponenten erweitern möchten. Die Details der Programmierung sind den entsprechenden Quelldateien zu entnehmen.

Im folgenden werden Paket-, Klassen-, Methoden-, und Dateinamen sowie Codebeispiele und Namen von Eigenschaften in `Schreibmaschinenschrift` geschrieben. Auf den Wert einer Eigenschaft `Name` beziehen wir uns mit dem Ausdruck `${Name}`. Variable Bestandteile des Namens einer Datei oder Eigenschaft werden in der Form `<variablerInhalt>` dargestellt.

4.1 Grundsätzliche Funktionsweise

Wie in Kapitel 3 beschrieben, werden Informationen über zu testende Formulare und zugehörige Testfälle in der Komponente „Datenspeicher“ gespeichert. Die Informationen werden dabei in XML-Dateien gespeichert, die im Dateisystem des Betriebssystems abgelegt werden. Die Struktur der XML-Dateien wird in vier verschiedenen Dokumenttyp-Definitionen (DTDs) definiert:

- `ParameterTypes.dtd`
  Eine Instanz dieser DTD definiert die allgemeinen Eigenschaften von Formularfeldern und kann als eine Art Vorlagenkatalog verstanden werden. Insgesamt werden vier verschiedene Arten von Formularfeldern unterschieden: Felder, die ein Datum enthalten, Felder, die eine Zeichenkette enthalten, Optionsfelder und Auswahlfelder. Der Zweck einer solchen Definition kann am besten durch ein Beispiel verdeutlicht werden.
  Beispiel: In verschiedenen Formularen einer Anwendung existieren Formularfelder, in denen ein Nachname eingetragen werden muß. Kontextabhängig kann ein solches Feld verschiedene Bezeichnungen haben, z.B. „Nachname“ oder „Name des Antragstellers“. Allerdings sollte ein Feld dieses Typs in allen Formularen der Anwendung dieselben Eigenschaften besitzen. Ein Nachname sollte z.B. eine Mindestlänge von zwei und eine Maximallänge von 30 Zeichen haben. Alle Felder dieses Typs würden dann durch ein einziges Element in der Instanz von `ParameterTypes.dtd` repräsentiert.
  Da davon ausgegangen wird, daß sich die Eigenschaften eines Formularfeldtyps zwischen zwei Versionen eines Formulars nicht ändern, und da das Framework gegenwärtig nur zum Testen der FlexiTrust-Formulare verwendet wird, existiert nur eine einzige Instanz dieser DTD.

- `Form.dtd`
  Instanzen dieser DTD definieren die zu testenden Formulare. Ein Formular besteht aus ein oder meheren Seiten, welche jeweils eine Menge von Parametern oder Formularfeldern enthält. Ein Formularfeld hat verschiedene Eigenschaften. Das Ausfüllen des Formularfeldes kann obgligatorisch oder fakultativ sein. Es kann eine Abhängigkeitsbeziehung zu einem anderen Formularfeld bestehen, d.h. das Formularfeld wird nur angezeigt, wenn ein bestimmtes anderes Formularfeld ausgefüllt oder nicht ausgefüllt wurde. Für jedes zu testende Formular und jede Version dieses Formulars existiert eine Instanz von `Form.dtd`. Die Namen der darin definierten Formularfelder entsprechen den Namen der in der Instanz von `ParameterTypes.dtd` definierten Formularfelder. Das Erstellen einer Instanz von `Form.dtd` wird anhand eines Beispiels detailliert in Anhang A.4 beschrieben.

- `TestCase.dtd`
  Instanzen dieser DTD repräsentieren konkrete Testfälle, d.h. sie enthalten die Daten, die ein Benutzer in ein entsprechendes HTML-Formular eintragen würde. Jede Instanz von `TestCase.dtd` ist mit einer bestimmten Instanz von `Form.dtd` assoziiert. Ein Unterschied zwischen Instanzen der beiden DTDs ist, daß Instanzen von `TestCase.dtd` nur Formularfelder enthalten, die auch ausgefüllt werden, während Instanzen von `Form.dtd` alle Felder eines Formulars enthalten, also auch diejenigen, die gar nicht angezeigt werden, weil eine bestimmte Abhängigkeitsbeziehung zu einem anderen Feld besteht. Ein anderer Unterschied ist, daß in Instanzen von `TestCase.dtd` die Namen der Formularfelder den Namen der Felder in dem jeweiligen HTML-Formular und nicht den Namen der in der Instanz von `ParameterTypes.dtd` definierten Formularfelder entsprechen.

- `FormToServletMapping.dtd`
  Eine Instanz dieser DTD bildet die Namen der in der Instanz von `ParameterTypes.dtd` definierten Formularfelder auf die Namen der Formularfelder in den entsprechenden HTML-Formularen ab. Somit ist eine Zuordnung von Formularfeldern aus einer Instanz von `Form.dtd` zu den Formularfeldern aus einer entsprechenden Instanz von `TestCase.dtd` möglich. Da zur Zeit nur eine Instanz von `ParameterTypes.dtd` existiert, gibt es auch nur eine Instanz von `FormToServletMapping.dtd`. Der Name eines Formularfelds, welches in der Instanz von `ParameterTypes.dtd` definiert ist, kann dabei auf mehrere Namen von Formularfeldern aus verschiedenen HTML-Formularen abgebildet werden.

Der in `FormToServletMapping.dtd` definierte Abbildungsmechanismus wurde eingeführt, da es nur so möglich ist, Anwendungen zu testen, bei denen sich die Namen der Formularfelder im HTML-Code von Version zu Version oder zwischen verschiedenen Formularen derselben Web-Anwendung ändern. Da Instanzen von `Form.dtd` die Namen der Formularfelder aus der Instanz von `ParameterTypes.dtd` verwenden, können zudem verschiedene Formulare einer Anwendung oder verschiedene Versionen eines Formulars einheitlich definiert werden. Dadurch werden Wartungsarbeiten erheblich erleichtert.

4.2 Datenspeicherung

Alle Daten werden in Form von Dateien im Dateisystem des Betriebssystems gespeichert. Neben dem Output der Protokollierungskomponente müssen auch die Instanzen der in Abschnitt 4.1 beschriebenen DTDs gespeichert werden. Auf Informationen in diesen Instanzen muß auch lesend zugegriffen werden. Die Protokollierung wurde mit dem Logging-Framework „Log4J“ des gleichnamigen Jakarta-Projekts realisiert, welches Methoden zum Speichern von Daten in Dateien bereitstellt.[1] Zum Speichern und Lesen der restlichen Daten wurde die Java Architecture for XML Binding (JAXB) benutzt.[2]

JAXB ist ein Framework zur Abbildung von XML-Dokumenten in Java-Objekte und umgekehrt. Mit Hilfe von JAXB wurde aus den DTDs Java-Quellcode erzeugt. Dazu mußte zunächst für jede DTD ein sog. Binding-Schema erstellt werden, welches in einer entsprechenden XML-Datei mit der Endung „xjs“ hinterlegt wurde. Dieses Schema legt fest, wie eine DTD in Java-Quellcode überführt wird, also z.B. ob für ein Element der DTD eine eigene Java-Klasse erzeugt wird oder ob für das Element eine Instanzenvariable in einer Java-Klasse angelegt wird, welche einem übergeordneten Element in der DTD entspricht. Daneben kann im Binding-Schema auch definiert werden, ob Attributwerte eines Elements als String-Objekte (Standardverhalten) oder als andere Java-Datentypen oder -Objekte interpretiert werden sollen. Ferner können auch noch Java-Interfaces definiert werden.

JAXB stellt Methoden zum Lesen und Schreiben von XML-Dateien zur Verfügung. Nachdem eine XML-Datei eingelesen wurde, steht ihr Inhalt als eine Hierachie von Java-Objekten zur Verfügung, auf welche mit Hilfe von Methoden zugegriffen werden kann, die vom JAXB-Compiler generiert wurden. Enthält ein Element einer XML-Datei mehrere andere Elemente, gibt es allerdings keine Möglichkeit, auf diese Elemente direkt zuzugreifen.

[1] Vgl. [LOG02]

[2] Ein kurzer Überblick über JAXB findet sich in [HU01]. Für eine umfassende Einführung vgl. [JAX01]

Daher wurden verschiedene Methoden implementiert, welche bestimmte Inhalte einer XML-Datei in einer Hash-Tabelle abbilden, so daß auf diese Informationen mit Hilfe bestimmter Schlüssel zugegriffen werden kann. Die Methoden sind in der Klasse `testtools.Utils` implementiert.

Eine Alternative zu dieser Implementierungsvariante ist das Speichern der Informationen in einer Datenbank. Dies hätte den Vorteil, daß keine Zugriffsmethoden implementiert werden müßten, da das JDBC-API[3] benutzt werden könnte. Ein Nachteil ist aber, daß Änderungen der Definition eines Formulars, Testfalls, etc. Zugriffe auf die Datenbank zur Folge hätten, wofür wieder Werkzeuge zur Verfügung gestellt werden müßten. Bei der für dieses Framework gewählten Implementierung müssen lediglich die entsprechenden XML-Dateien geändert werden, was mit Hilfe eines einfachen Texteditors erfolgen kann.

4.3 Konfiguration

Die Komponente „Konfiguration“ dient der Konfiguration der übrigen Komponenten des Frameworks. Zu diesem Zweck steht die Konfigurationsdatei `testtools.properties` zur Verfügung, in der alle wichtigen Eigenschaften der Komponenten konfiguriert werden. Ferner exisitieren Zugriffsmethoden auf diese Eigenschaften, welche in der Klasse `testtools.Configuration` implementiert wurden.

Die Konfigurationsdatei liegt als Java-Property-Datei vor. d.h sie ist zeilenorientiert, und jede Zeile hat das Format „`<PropertyName>=<Value>`“. Zeilen, die mit „`#`“ beginnen, werden ignoriert und können daher Kommentare enthalten. Die Datei `testtools.properties` ist in verschiedene Abschnitte aufgeteilt. Im ersten Abschnitt werden globale Eigenschaften definiert. Die Eigenschaft `home` beschreibt das Wurzelverzeichnis, in dem alle im folgenden beschriebenen Verzeichnisse liegen. Die Eigenschaft `applicationDir` legt das Verzeichnis fest, in dem die Dateien des Frameworks liegen und ist relativ zum Verzeichnis `${home}` definiert, d.h die Dateien des Frameworks liegen im Verzeichnis `${home}/${applicationDir}/`. Die Eigenschaft `testCaseDir` beschreibt das Verzeichnis, in dem die abgespeicherten Testfälle liegen, und ist ebenfalls relativ zu `${home}` definiert. Jeglicher Logging-Output sowie die Zusammenfassungen der Testläufe werden im Verzeichnis `${home}/${records}` gespeichert. Die Eigenschaft `datePattern` gibt das Muster an, nach dem Zeitstempel formatiert werden sollen.[4] Die Eigenschaft `default-`

[3] Java Database Connectivity; API zum Zugriff auf relationale Datenbanken

[4] Die Syntax dieses Musters entspricht der Spezifikation in der Klasse `java.text.Simple-`

ServletURL beschreibt die URL der zu testenden Web-Anwendung.

Die Eigenschaften des nächsten Abschnitts beziehen sich auf die Verarbeitung der definierten Formulare, also der Instanzen von Form.dtd. Die Eigenschaft xmlDir bezeichnet das Verzeichnis, in dem die Instanzen gespeichert sind. Dieses Verzeichnis ist relativ zum Verzeichnis ${home}/${applicationDir} definiert. Die Eigenschaft forms enthält eine Liste der Namen aller bekannten Formulare. Diese Namen stammen aus dem Attribut „name" des Wurzelelements „Form" der entsprechenden XML-Dateien. Die Eigenschaft currentVersion beschreibt den Namen der Version der zu testenden Web-Anwendung, die standardmäßig von allen Komponenten verwendet wird. versions beinhaltet eine Liste aller bekannten Versionen der zu testenden Web-Anwendung. Die Eigenschaften FormToServletMapping und ParameterTypes geben die Positionen der verwendeten Instanzen von FormToServletMapping.dtd und ParameterTypes.dtd relativ zum Verzeichnis ${home}/${applicationDir}/${xmlDir} an. Die Eigenschaften <formName>.<version> beschreiben für alle Formulare aus ${forms} und alle Versionen aus ${versions} die Position der entsprechenden Instanz von Form.dtd relativ zum Verzeichnis ${home}/${applicationDir}/${xmlDir}. Die Eigenschaften <formName>.<version>.name ordnen den Formularnamen aus der Liste ${forms}, also den Namen der Instanzen von Form.dtd, entsprechende HTML-Formulare der zu testenden Web-Anwendung zu. Diese Zuordnung wird nicht in der Instanz von FormToServletMapping.dtd definiert, da sie abhängig von der zu testenden Web-Anwendung auf unterschiedliche Weise erfolgen kann. Im Falle des FlexiTrust-Servlets wird ein HTML-Formular z.B. über einen Eintrag in der Auswahlliste auf der ersten HTML-Seite des Servlets identifiziert. In einer anderen Web-Anwendung könnte ein Formular z.B. über das Attribut „name" des öffnenden HTML-Tags „form" identifiziert werden.

Die Eigenschaften des nächsten Abschnitts, TestCase.doctype, TestCase.dtd und TestCase.encoding, beschreiben die Standardwerte für den Dokumenttyp, die Position der DTD und den Zeichensatz für Instanzen von TestCase.dtd. Da Testfälle standardmäßig im Verzeichnis ${home}/${testCaseDir} gespeichert werden, wird ${TestCase.dtd} relativ zu diesem Pfad angegeben.

Die Eigenschaften mit dem Präfix „log4j" beziehen sich auf die Protokollierungskomponente, welche in Abschnitt 4.5 beschrieben wird.

DateFormat

Der nächste Abschnitt bezieht sich auf die Testfallaufzeichnung. Die Eigenschaft `userAgent` legt fest, von welchen Web-Browser-Typen keine Eingaben aufgezeichnet werden sollen. Diese Eigenschaft wird dazu verwendet, um das Aufzeichnen von Daten zu verhindern, die von den im Framework definierten Tests abgeschickt werden. Die Eigenschaft `webAppLibDir` definiert eine Position, an der die zu testende Web-Anwendung zusätzliche Klassendateien findet, und wird für die Installation der Komponente „Testfallaufzeichnung“ benötigt.[5]

Der nächste Abschnitt der Konfigurationsdatei, welcher die Eigenschaften mit dem Präfix „TestCaseGenerator.“ enthält, bezieht sich auf die Konfiguration des Testfallgenerators, welcher in Abschnitt 4.7 beschrieben wird. Die Eigenschaften mit dem Präfix „TestCaseConverter.“ dienen der Konfiguration des Testfallkonverters, welcher in Abschnitt 4.8 beschrieben wird. Die Eigenschaften der letzten Abschnitte beziehen sich auf die Testdurchführung und die Ergebnisüberprüfung, welche in den Abschnitten 4.9 und 4.10 beschrieben sind.

4.3.1 Einschränkung

Prinzipiell lassen sich mit dem hier vorgestellten Framework zwar mehrere Web-Anwendungen testen, die Konfigurationskomponente unterstützt allerdings nicht das gleichzeitige Testen mehrerer Web-Anwendungen. Zu diesem Zweck könnte in der Konfigurationsdatei eine Eigenschaft `webApps` eingeführt werden, welche eine Liste von Bezeichnern zu testender Web-Anwendungen enthält. Namen von Eigenschaften, die abhängig von der zu testenden Web-Anwendung sind, müssen dann durch ein Präfix erweitert werden, welches den Bezeichner der entsprechenden Web-Anwendung enthält. Außerdem muß die Implementierung der Klasse `testtools.Configuration` entsprechend erweitert werden.

4.4 Steuerung

Diese Komponente wurde mit dem Build-Tool „Ant“[6] realisiert. Grundlage ist die Konfigurationsdatei `testtools.properties` und die Build-Datei `build.xml`. Der Funktionsumfang läßt sich am besten anhand der in der Datei `build.xml` beschriebenen Targets beschreiben. Ein Target wird dabei mit dem Kommando „`ant <targetName>`“ aufgerufen.

- `init`: Dieses Target generiert einen Zeitstempel, der gemäß des Musters `${date-`

[5] Vgl. Abschnitt 4.6
[6] Vgl. [ANT02]

`Pattern`} formatiert wird. Außerdem werden die Verzeichnisse erstellt, in denen die kompilierten Java-Klassen, Informationen über Abhängigkeiten zwischen Java-Klassen, die generierte Java-Dokumentation und Backups gespeichert werden. Ferner wird noch der vom JAXB-Compiler generierte Java-Quellcode gelöscht, falls die entsprechenden DTDs neuer als der entsprechende Java-Quellcode ist.

- `xjc`: Dieses Target ruft den JAXB-Compiler auf, der für alle in Abschnitt 4.1 beschriebenen DTDs Java-Quellcode erzeugt. Dieses Target wird nur ausgeführt, wenn der generierte Java-Quellcode nicht mehr auf dem neuesten Stand ist, d.h. wenn dieser während der Ausführung des Targets `init` gelöscht wurde.

- `compile`: Mit diesem Target wird der Quellcode kompiliert. Dabei wird das optionale Ant-Task „depend“ benutzt, welches Abhängigkeiten zwischen den Java-Klassen feststellt. Dadurch werden nur Quelldateien kompiliert, wenn sie sich geändert haben oder wenn sich eine Datei geändert hat, von der sie abhängen.

- `dist`: Dieses Target erzeugt aus den kompilierten Dateien eine JAR-Datei und legt diese im Distributionsverzeichnis ab.

- `deploy_filter`: Mit diesem Target wird die Benutzung der Testfallaufzeichnung vorbereitet. Die mit dem vorigen Target erzeugte JAR-Datei wird an eine Stelle kopiert, an der sie von der Web-Applikation, für die Testfälle aufgezeichnet werden sollen, gefunden wird.[7]

- `clean`: Mit diesem Target werden die kompilierten Java-Klassen und die generierte Java-Dokumentation gelöscht, indem die mit dem Target `init` erzeugten Verzeichnisse, sowie die mit dem Target `deploy_filter` kopierte JAR-Datei gelöscht werden.

- `backup`: Dieses Target erzeugt ein Backup aller Java-Quelldateien, aller DTDs, aller XJS- und XML-Dateien, sowie aller Property-Dateien.

- `run`: Dieses Target führt eine Java-Klasse aus, die als Eigenschaft „classname“ beim Aufruf von Ant übergeben wurde. Der Testfallgenerator könnte so z.B. auch mit dem Kommando „`ant -Dclassname=testtools.TestCaseGenerator run`“ gestartet werden.

- `generate`: Mit diesem Target wird der Testfallgenerator gestartet.

- `convert`: Mit diesem Target wird der Testfallkonverter aufgerufen.

[7]Vgl. Abschnitt 4.6

- test: Dieses Target führt alle implementierten Tests aus. Die Quelldateien der auszuführenden Tests müssen dabei per Konvention unterhalb des Verzeichnisses ${src} liegen. Die Dateinamen müssen dem Muster „*Test.java“ entsprechen.

4.4.1 Einschränkung

Der verwendete JAXB-Compiler liegt gegenwärtig nur in einer „Early Access“-Version vor und ist noch nicht vollständig und teilweise fehlerhaft implementiert. Falls ein Element einer DTD ein Attribut mit Identifikationsreferenzwert („IDREF“) enthält, wird eine fehlerhafte „import“-Anweisung generiert, welche entfernt bzw. auskommentiert werden muß. Im Quellcode der generierten Klasse testtools.xml.Parameter muß daher folgende Zeile entfernt werden, bevor der Quellcode kompiliert werden kann.

```
import testtools.xml.DependsVPatcher;
```

Ferner wurde die Initialisierung der Testumgebung noch nicht implementiert. Für FlexiTrust exisitiert bereits ein Werkzeug zum Zurücksetzen des Systems in seinen initialen Zustand. Dieses wurde jedoch noch nicht eingebunden. Zudem müßten noch Mechanismen geschaffen werden, mit denen das System in einen beliebigen Zustand gesetzt werden kann. Für FlexiTrust wäre z.B. ein Zustand wünschenswert, in dem die Datenbank bereits eine Anzahl bestimmter Datensätze enthält.

4.5 Protokollierung

Für die Realisierung dieser Komponente wurde das Logging-Framework „Log4J“ des gleichnamigen Jakarta-Projekts verwendet. Logging-Output wird für die Komponenten Testfallaufzeichnung, Testfallgenerator, Testfallkonverter und Testdurchführung erzeugt, wobei bei der Testdurchführung auch der Output der Ergebnisüberprüfung protokolliert wird.

Für die Komponenten Testfallaufzeichnung, Testfallgenerator und Testfallkonverter wird der Logging-Output in einem Verzeichnis namens ${home}/${recordDir}/<className>/log.<TimeStamp>.xml gespeichert, wobei <className> entsprechend der jeweiligen Komponente testtools.CaptureFilter, testtools.TestCaseGenerator oder testtools.TestCaseConverter ist. <TimeStamp> ist der gemäß dem Muster ${datePattern} formatierte Zeitstempel, welcher zum Aufrufzeitpunkt der jeweiligen Komponente erzeugt wurde.

Während der Testdurchführung wird der Logging-Output für jeden durchgeführten Test in ein Verzeichnis namens `${home}/${recordDir}/<TimeStamp>/<TestName>` geschrieben, wobei `<TestName>` der Klassenname des entsprechenden Tests ist, also z.B. `testtools.ReplayTest`. Eine Zusammenfassung der Ergebnisse des jeweiligen Testlaufs wird in das Verzeichnis `${home}/${recordDir}/<TimeStamp>/summary` geschrieben. Zusätzlich wird eine Übersicht über die Ergebnisse aller Testläufe in eine Datei `${home}/${recordDir}/<TimeStamp>/index.html` geschrieben.

Das Logging-Verhalten kann in der Konfigurationsdatei durch die Eigenschaften mit dem Präfix „log4j“ beeinflußt werden. Hier können alle Konfigurationsmöglichkeiten des Log4J-Frameworks genutzt werden. Insbesondere kann für jede Komponente das sog. „Log-Level“ eingestellt werden, welches festlegt, welche Informationen mitprotokolliert werden. Falls das Log-Level für einen durchzuführenden Tests auf „DEBUG“ gesetzt wird, wird zusätzlich der Inhalt jeder vom Web-Server verschickten HTML-Seite mitprotokolliert. Im Fehlerfall wird zudem der Inhalt des zuletzt erhaltenen HTML-Formulars protokolliert. Darüber hinaus werden auch die Dateinamen der jeweiligen Testfälle protokolliert, bzw. deren Inhalt, falls sie zur Laufzeit generiert wurden. Somit ist eine effiziente Fehlersuche möglich, da die protokollierten HTML-Seiten und das zuletzt erhaltene Formular mit dem jeweiligen Testfall verglichen werden können.

4.6 Testfallaufzeichnung

Für die Implementierung dieser Komponente gab es zwei verschiedene Ansätze. Abbildung 4.1 stellt den ersten Ansatz dar. Die Testfallaufzeichnung („CaptureTool“) fungiert als Vermittler zwischen dem Browser und der Web-Anwendung („WebApp“). Das CaptureTool besteht aus zwei Komponenten, dem sog. „SnoopUtil“ sowie dem sog. „LogUtil“. Das SnoopUtil lauscht am Browser-Port, liest die Anfragen des Browsers (Schritt 1) und leitet diese unverändert an das LogUtil weiter (Schritt 2). Dieses analysiert die empfangenen Daten und schreibt die aufzuzeichnenden Informationen in eine Datei. Die aufgezeichneten Daten werden als Testfälle gemäß der DTD in `TestCase.dtd` abgespeichert. Anschließend werden die Daten an die Web-Anwendung weitergereicht (Schritt 3). Die Antwort der Web-Anwendung wird vom LogUtil an das SnoopUtil weitergeleitet und von diesem zurück an den Browser geschickt.

Das LogUtil wurde als Servlet implementiert, welches in dem Servlet-Container Tomcat

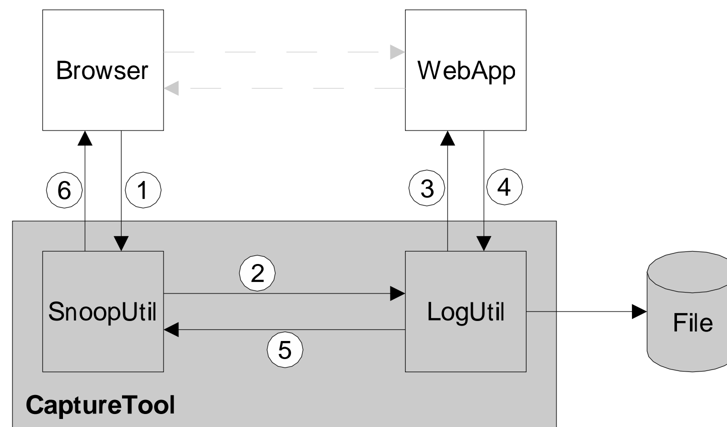

Abbildung 4.1: Erster Implementierungsansatz für die Testfallaufzeichnung

läuft.[8] Die Kommunikation zwischen dem LogUtil und der Web-Anwendung findet mittels HttpUnit statt.[9] Die Kommunikation zwischen dem SnoopUtil und dem Browser bzw. dem LogUtil findet mittels Socket- bzw. ServerSocket-Objekten aus dem Java-Paket `java.net` statt. Diese Implementierungsvariante besitzt den großen Vorteil, daß die Testfallaufzeichnung unabhängig von der zu testenden Web-Anwendung funktioniert. Ein großer Nachteil ist allerdings, daß die Implementierung fehleranfällig ist, da Daten auf einer niedrigen Abstraktionsebene (Sockets) verarbeitet werden müssen, ohne daß dabei auf ein mächtiges API zurückgegriffen werden kann. Während der Implementierung stießen wir auf einen reproduzierbaren Fehler, welcher nur auftrat, wenn es sich bei der zu testenden Web-Anwendung um ein Servlet handelte. In diesem Fall blockierte der InputStream, der die Antwort der Web-Anwendung laß, so daß diese nicht vollständig zum Browser gesendet werden konnte.

Mit der Veröffentlichung der Version 4.0 des Servlet-Containers Tomcat wurde daher die zweite Variante implementiert. Diese basiert darauf, daß in der Servlet-Spezifikation 2.3[10], für welche Tomcat 4 die Referenzimplementierung ist, sog. Filter existieren, die dazu benutzt werden können, Anfragen eines Clients zu filtern, bevor sie an ein Servlet weitergereicht werden. Anschließend kann auch die Antwort des Servlets gefiltert werden, bevor sie an den Client zurückgeschickt wird.

Abbildung 4.2 stellt die zweite Implementierungsvariante dar. Diese besitzt den großen Vorteil, daß sie einfacher und somit weniger anfällig für Fehler ist, da zur Analyse von Client-Anfragen das Servlet-API benutzt werden kann. Ein Nachteil ist jedoch, daß die Testfal-

[8] Vgl. [TOM02]
[9] HttpUnit ist ein Framework zur Automatisierung von Web-Site-Tests. Vgl. [HTT02]
[10] Vgl. [SER02]

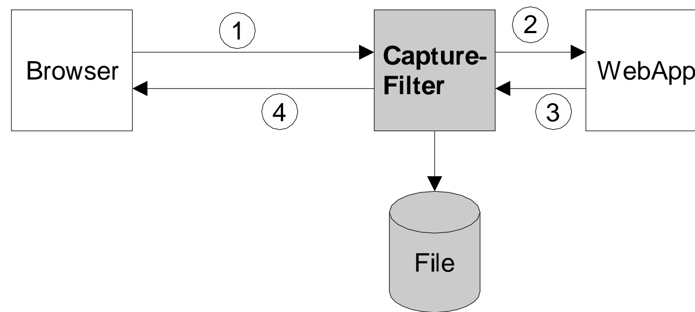

Abbildung 4.2: Zweiter Implementierungsansatz für die Testfallaufzeichnung

laufzeichnung nur für Web-Anwendungen funktioniert, die in einem Servlet-Container laufen, welcher die Servlet-Spezifikation 2.3 oder neuer implementiert. Da das hier vorgestellte Framework zum Testen des FlexiTrust-Servlets entwickelt wurde und die aktuelle Servlet-Spezifikation rückwärtskompatibel ist, ist darin keine Einschränkung zu sehen. Allerdings läuft die aktuelle Version des FlexiTrust-Servlet bisher nicht mit Tomcat 4.0 oder höher. Das Problem wurde bereits identifiziert, konnte aber zum Fertigstellungszeitpunkt dieser Arbeit noch nicht behoben werden.

Da die Implementierung der ersten Variante nicht vollständig abgeschlossen wurde, soll im folgenden beschrieben werden, wie die zweite Variante eingesetzt werden kann. Um diese Variante der Testfallaufzeichnung zu benutzen, muß die Klasse `testtools.CaptureFilter` an eine Position kopiert werden, wo sie von dem zu testenden Servlet gefunden wird. Im Falle des FlexiTrust-Servlets unter Tomcat 4.x ist dies z.B. das Verzeichnis `<TOMCAT_HOME>/webapps/openra/WEB-INF/lib`, wobei `<TOMCAT_HOME>` das Wurzel-Verzeichnis der Tomcat-Installation ist. Dieses Verzeichnis muß in der Eigenschaft `webAppLibDir` eingetragen werden, also z.B. `webAppLibDir=/home/flexitest/jakarta-tomcat/webapps/openra/WEB-INF/lib`. Die entsprechenden Klassendateien werden dann mit Hilfe der Steuerungskomponente mit dem Aufruf „`ant deploy_filter`“ kopiert. Außerdem muß dem Servlet-Container bekannt gemacht werden, daß der CaptureFilter verwendet werden soll und für welche Servlets er verwendet werden soll. Dazu müssen im Falle des FlexiTrust-Servlets unter Tomcat 4.x folgende Zeilen in die Datei `$TOMCAT_HOME/webapps/openra/WEB-INF/web.xml` direkt unterhalb des öffnenden `<web-app>`-Tags eingetragen werden:

```
<!-- Define filter -->
<filter>
    <!-- arbitrary expressive name -->
```

```
        <filter-name>CaptureFilter</filter-name>
        <!-- class name of filter -->
        <filter-class>testtools.CaptureFilter</filter-class>
    </filter>

    <!-- Define filter mappings for the defined filter -->
    <filter-mapping>
        <!-- filter name from above -->
        <filter-name>CaptureFilter</filter-name>
        <!-- pattern, for which filter will be applied;
          in this example, all servlets in this context
          will be filtered -->
        <url-pattern>/servlet/*</url-pattern>
    </filter-mapping>
```

Falls eine Web-Applikation von Tomcat 3.x ohne Veränderung kopiert wurde, ist zudem darauf zu achten, daß der Dokumenttyp der Datei `<TOMCAT_HOME>/webapps/openra/WEB-INF/web.xml` an die Servlet-Spezifikation 2.3 angepaßt wird:

```
<!DOCTYPE web-app PUBLIC
  "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
  "http://java.sun.com/j2ee/dtds/web-app_2_3.dtd">
```

4.7 Testfallgenerator

Mit dieser Komponente können Testfälle generiert werden. Das Framework wird mit einem Generator ausgeliefert, der Testfälle erzeugt, die zufällige, gültige Werte für die entsprechenden Formularfelder enthalten. Der Generator kann in verschiedenen Modi betrieben werden.[11]

Daneben kann das Famework mit selbst implementierten Testfallgeneratoren erweitert werden. Ein solcher Testfallgenerator muß die abstrakte Klasse `testtools.TestCaseGenerator` erweitern. Ferner muß die abstrakte Instanzenmethode `generateSingleTestCase` implementiert werden, welche als Aufrufparameter einen Formularnamen, eine Versionsbezeichnung und einen Modus erwartet. Diese Methode liefert einen Testfall für das entsprechende Formular in der entsprechenden Version zurück, welcher entsprechend des

[11] Vgl. den Code der Klasse `testtools.RandomTestCaseGenerator`

übergebenen Modus generiert wurde.

Der Testfallgenerator kann auf zwei verschiedene Arten genutzt werden. Eine Instanzierung des Testfallgenerators und ein Aufruf der Instanzenmethode `generateSingleTestCase` erzeugt wie gerade beschrieben einen einzigen Testfall. Die zweite Möglichkeit ist, die Klasse `testtools.TestCaseGenerator` direkt auszuführen, wie es z.B. mit Hilfe der Steuerungskomponente möglich ist. Auf diese Weise wird eine Anzahl von Testfällen erzeugt und in Dateien gespeichert. Das Verhalten des Testfallgenerators kann wie folgt über die Konfigurationsdatei gesteuert werden.

Die Eigenschaft `TestCaseGenerator.implementation` legt fest, welche Implementierung des Testfallgenerators benutzt werden soll. Diese Eigenschaft muß den vollständigen Klassennamen einer Testfallgenerator-Implementierung enthalten. Sie wird benötigt, falls eine Instanz des Testfallgenerators mit der Methode `testtools.Configuration.createTestCaseGenerator` erzeugt werden soll. Ansonsten kann die Instanzierung eines Testfallgenerators auch durch den Aufruf des entsprechenden Konstruktors erfolgen. Da die Methode `testtools.Configuration.createTestCaseGenerator` einen Testfallgenerator mit Hilfe der Mechanismen aus dem Paket `java.lang.reflect` instanziiert, muß bei der Implementierung eines eigenen Testfallgenerators der Code des Frameworks nicht angepaßt werden. Die einzige Einschränkung ist, daß der implementierte Testfallgenerator die abstrakte Klasse `testtools.TestCaseGenerator` erweitern muß. Damit soll eine einheitliche Protokollierung für alle Testfallgeneratoren sichergestellt werden.

Die folgenden Eigenschaften beziehen sich auf die direkte Ausführung des Testfallgenerators. Mit Hilfe der Eigenschaft `TestCaseGenerator.forms` kann festgelegt werden, für welche Formulare Testfälle generiert werden sollen. Die Eigenschaft `TestCaseGenerator.modes` legt fest, in welchen Modi der Testfallgenerator ausgeführt werden kann. Die Klasse `testtools.RandomTestCaseGenerator` kennt z.B. verschiedene Modi, in denen entweder nur Werte für obligatorische Formularfelder oder Werte für alle sichtbaren Formularfelder generiert werden. Die Eigenschaft `TestCaseGenerator.number` legt fest, wie viele Testfälle pro Modus für jedes Formular generiert werden sollen.

4.7.1 Einschränkung

Die Methode `testtools.TestCaseGenerator.invalidateParameterValue`, welche einen gültigen Eingabewert für ein Formularfeld invalidiert, ist nicht vollständig im-

plementiert. In der Klasse `testtools.RandomValueGenerator` wird für Formularfelder, die Zeichenketten enthalten, bei der Generierung eines Wertes nicht unterschieden, um was für eine Zeichenkette es sich handelt. Die Verwendung unterschiedlicher Zeichenkettenformate ist allerdings bereits in der Datei `ParameterTypes.dtd` vorgesehen.

4.8 Testfallkonverter

Diese Komponente ist dafür zuständig, Testfälle für ein Formular von einer Version in eine andere Version zu konvertieren. Diese Funktionalität ist wichtig, wenn gespeicherte Testfälle eines bestimmten Formulars zum Testen einer neuen oder anderen Version dieses Formulars wiederverwendet werden sollen. Bei einer neuen Version eines Formulars können folgende Fälle auftreten:

Fall 1: Formularfelder der alten Version fallen in der neuen Version weg.

Fall 2: Es kommen neue Formularfelder hinzu.

Fall 3: Die „required"-Eigenschaft eines Formularfeldes ändert sich.

Fall 4: Die Abhängigkeiten zwischen Formularfeldern ändern sich.

Fall 5: Ein Formularfeld erscheint in der neuen Version auf einer anderen Seite.

Fall 6: Eigenschaften eines Formularfeldes ändern sich.

In Fall 1 fällt das entsprechende Formularfeld in dem konvertierten Testfall weg. In Fall 2 wird ein Wert für das neu hinzugekommene Feld generiert. Die Art der Erzeugung des neuen Wertes richtet sich nach der Konfiguration des Testfallgenerators.[12] In Fall 3 wird ein neuer Wert generiert, falls das entsprechende Feld in der alten Version nicht ausgefüllt war und nun obligatorisch auszufüllen ist.

Im Falle einer geänderten Abhängigkeitsbeziehung (Fall 4) müssen verschiedene Möglichkeiten berücksichtigt werden:

- Die Art der Abhängigkeit kann sich ändern.[13]
- Das Feld, von dem das entsprechende Formularfeld abhängt, kann sich geändert haben.

[12] Vgl. Abschnitt 4.7

[13] Vgl. Abschnitt 4.9 für eine Beschreibung der verschiedenen Abhängigkeitsarten.

- Die entsprechenden Formularfelder in beiden Versionen können von unterschiedlichen Feldern abhängen.

Da Abhängigkeitsbeziehungen im FlexiTrust-Servlet einen Sonderfall darstellen und da gegenwärtig nicht damit zu rechnen ist, daß sich die existierenden Abhängigkeitsbeziehungen in späteren Versionen ändern werden, wurden die gerade beschriebenen Möglichkeiten nicht gesondert betrachtet. Stattdessen wird bei einer Änderung in der Abhängigkeitsbeziehung für das entsprechende Formularfeld ein neuer Wert entsprechend der Definition der neuen Formularversion generiert.

Erscheint ein Formularfeld in der neuen Version auf einer anderen Seite (Fall 5), wird es auf dieser Seite eingefügt. Dies impliziert, daß die Konvertierung nicht seitenweise erfolgen kann und daß die alte Version des Testfalls erst komplett eingelesen werden muß, bevor mit der Konvertierung begonnen wird. Änderungen der Eigenschaften eines Formularfeldes (Fall 6), wie z.B. die Mindestlänge der enthaltenen Zeichenkette, werden in der aktuellen Version des Frameworks nicht unterstützt. Voraussetzung für eine Unterstützung ist, daß verschiedene Versionen der Instanz von `ParameterTypes.dtd` unterstützt werden. Bei der Konvertierung eines Formularfeldes muß dann überprüft werden, ob der aktuelle Wert gemäß der neuen Definition immer noch gültig ist. Ist dies nicht der Fall, so muß überprüft werden, ob der alte Wert an die neue Definition angepaßt werden kann, oder ob ein neuer Wert erzeugt werden muß.

Zur Konfiguration des Testfallkonverters stehen in der Konfigurationsdatei drei Eigenschaften zur Verfügung: Es werden nur Testfälle konvertiert, die für eines der Formulare aus `${TestCaseGenerator.forms}` erstellt wurden, und die zu konvertierenden Testfälle werden von der Version `${TestCaseGenerator.fromVersion}` in die Version `${TestCaseGenerator.toVersion}` konvertiert.

4.9 Testdurchführung

In dieser Komponente werden die eigentlichen Testläufe durchgeführt. Für die Implementierung wurden die Frameworks JUnit und HttpUnit benutzt. JUnit ist ein Framework zur Automatisierung sog. „Unit Tests“[14] in Java. HttpUnit ist eine JUnit-Erweiterung, mit der sich Tests von Web-Anwendungen automatisieren lassen. Im folgenden setzen wir voraus, daß

[14] Der Begriff „Unit Test“ kann mit „Komponententest“ beschrieben werden und bezieht sich auf das Testen einzelner Einheiten eines Software-Systems. Eine Einheit kann dabei ein einzelnes Modul oder eine Zusammenfassung mehrerer Einheiten sein, so daß der Begriff vom Modultest bis zum Integrationstest reicht.

die grundsätzliche Arbeitsweise dieser Frameworks bekannt ist. Eine Einführung zu JUnit findet sich in [BG98], eine kurze Beschreibung zu HttpUnit in [HTT02].

Das Framework stellt bereits vier Tests zur Verfügung:

- einen Test von Abhängigkeitsbeziehungen zwischen Formularfeldern,
- einen Test von ungültigen Eingabewerten,
- einen Test von obligatorisch auszufüllenden Formularfeldern und
- einen Regressionstest.

Die Klasse `testtools.DependenceTest` implementiert den Test von Abhängigkeitsbeziehungen zwischen Formularfeldern. Dabei wird zwischen zwei verschiedenen Abhängigkeitsarten unterschieden. Wird ein Formularfeld nur angezeigt, wenn ein entsprechendes Formularfeld ausgefüllt wurde, liegt - in Anlehnung an die englische Bezeichnung des logischen Und-Operators - eine „AND-Abhängigkeit“ vor. Wird ein Formularfeld nur angezeigt, falls ein entsprechendes Formularfeld nicht ausgefüllt wurde, liegt eine sog. „XOR-Abhängigkeit“ vor. Um alle Abhängigkeiten zu berücksichtigen, wird für ein Formular der Test jeder Abhängigkeitsart mit zwei verschiedenen Testfällen durchgeführt. Der erste Testfall enthält nur Werte für obligatorisch auszufüllende Formularfelder, der zweite Testfall enthält Werte für alle sichtbaren Formularfelder. Somit sollten bei einer korrekten Formulardefinition im ersten Testfall alle Formularfelder erscheinen, die in einer XOR-Abhängigkeitsbeziehung zu einem anderen Formularfeld stehen. Im zweiten Testfall sollten analog die Formularfelder erscheinen, welche in einer AND-Abhängigkeitsbeziehung zu einem anderen Formularfeld stehen. Daraus folgt, daß mit diesem Mechanismus keine mehrstufigen Abhängigkeitsbeziehungen getestet werden können.

Mit dem in der Klasse `testtools.IllegalValuesTest` implementierten Test wird überprüft, ob sich die Web-Anwendung korrekt verhält, wenn ungültige Werte in ein Formularfeld eingetragen werden. Die Gültigkeit eines Eingabewertes hängt dabei vom Typ des Formularfelds ab. Wird ein Feld mit einem ungültigen Wert ausgefüllt (z.B. „32.01.2003“ für ein Feld vom Typ „Datum“), wird erwartet, daß die Web-Anwendung nach dem Empfang der fehlerhaft ausgefüllten Seite dieselbe Seite wieder zurückschickt, damit der Wert korrigiert werden kann. Zur Durchführung dieses Tests wird zunächst ein Testfall erzeugt, der nur gültige Eingabewerte enthält. Anschließend wird eine Seite des Formulars mit den entsprechenden Werten aus dem Testfall ausgefüllt. Vor dem Abschicken wird der Eingabewert

eines Formularfeldes invalidiert. Es wird erwartet, daß die Web-Anwendung dieselbe Seite wieder zurückliefert. Der vorher invalidierte Eingabewert wird wieder auf einen gültigen Wert gesetzt, und die gerade beschriebenen Schritte werden für die restlichen Formularfelder wiederholt. Um alle Abhängigkeiten zwischen Formularfeldern berücksichtigen zu können, wird der Test wie im vorigen Fall mit zwei verschieden generierten Testfällen ausgeführt.

Mit dem in der Klasse `testtools.RequiredTest` implementierten Test wird überprüft, ob sich die Web-Anwendung korrekt verhält, wenn ein obligatorisch auszufüllendes Formularfeld nicht ausgefüllt wurde. Das Verhalten ist analog zum Test ungültiger Eingabewerte korrekt, wenn eine Seite, bei der ein obligatorisch auszufüllendes Feld nicht ausgefüllt wurde, zur Korrektur zurückgeschickt wird. Der Test läuft ebenfalls analog zum Test ungültiger Werte ab, nur daß statt der Invalierung von Eingabewerten obligatorisch auszufüllende Formularfelder nicht ausgefüllt werden.

Die Klasse `testtools.ReplayTest` implementiert einen Regressionstest. Während dieses Tests werden die gespeicherten Testfälle wieder abgespielt. Dabei wird eine Anzahl gleichzeitig mit der Web-Anwendung arbeitender Benutzer simuliert. Die Eigenschaften `concurrentUsers` und `workload` in der Konfigurationsdatei legen fest, welche Anzahl gleichzeitiger Benutzer simuliert wird und wieviele Testfälle für jeden Benutzer abgespielt werden.

In der Konfigurationsdatei gibt es verschiedene Eigenschaften, die für die Durchführung der Tests relevant sind. Standardmäßig werden die Tests für alle Formulare in der Liste `${forms}` ausgeführt. In der Eigenschaft `Tests` können die Klassennamen von Tests eingetragen werden, die nur für eine Teilmenge der Formulare aus `${forms}` durchgeführt werden sollen. Anschließend kann für jeden Klassennamen aus der Liste `${Tests}` eine Eigenschaft `<className>.forms` definiert werden, welche eine Liste der Formulare enthält, die getestet werden sollen.

Beispiel: Die Konfigurationsdatei enthält folgende Eigenschaften:

```
forms=form1,form2
Tests=testtools.DependenceTest,testtools.RequiredTest
testtools.DependenceTest.forms=form1
testtools.RequiredTest.forms=form2
```

Dann wird der Test von Abhängigkeitsbeziehungen zwischen Formularfeldern nur für das Formular „form1“ ausgeführt. Der Test obligatorisch auszufüllender

Formularfelder wird nur für „form2“ ausgeführt. Alle anderen Tests werden mit „form1“ und „form2“ ausgeführt.

Die Eigenschaft `ResultChecker.implementation` legt fest, auf welche Art die Ergebnisüberprüfung vorgenommen wird.[15] Der Wert dieser Eigenschaft ist der vollständige Name der Klasse, welche die gewünschte Ergebnisüberprüfung implementiert.

Das Framework kann um selbst erstellte Tests erweitert werden. Dabei ist wie folgt vorzugehen: Die Klasse des implementierten Tests sollte die Klasse `testtools.TesttoolsTestCase` erweitern, um eine einheitliche Protokollierung sicherzustellen. Damit der neue Test von der Steuerungskomponente gefunden wird, muß der Klassenname auf „Test“ enden. Ferner muß die Quelldatei unterhalb des Verzeichnisses gespeichert werden, in dem die Quelldateien des Frameworks gespeichert sind. In der Regel werden Formulare einer Web-Anwendung getestet. Die Implementierung sollte dann wie folgt aussehen:

```
 1 public class XYZTest extends TesttoolsTestCase {
 2     // get instance of Configuration component
 3     static Configuration c = Configuration.getInstance();
 4     // get names of forms that have to be tested
 5     static String[] formNames =
 6         c.getTestForms(XYZTest.class.getName());
 7
 8     protected void setUp() {
 9         // for this test testObjects correspond to
10         // the forms that have to be tested
11         super.testObjects = formNames;
12         super.setUp();
13     }
14
15     public XYZTest(String name) {
16         super(name);
17     }
18
19     public static void main (String[] args) {
20         junit.textui.TestRunner.run(suite());
21     }
```

[15] Vgl. Abschnitt 4.10

```
22
23     public static Test suite() {
24         TestSuite suite = new TestSuite();
25         // add a test for each form
26         for (int i = 0; i < formNames.length; i++) {
27             suite.addTest(new XYZTest("testMethod"));
28         }
29         return suite;
30     }
31
32     public void testMethod() {
33         ...
34     }
35 }
```

In der `for`-Schleife ab Zeile 26 wird für jedes Formular ein Test zur Testsuite hinzugefügt. Bei der Ausführung des Testfalls wird gemäß der Implementierung von JUnit für jeden dieser Tests einmal die Methode `setUp()` aus Zeile 8 ausgeführt. Die Zuweisung in Zeile 11 und der Aufruf des Konstruktors in Zeile 12 sorgen dafür, daß die Protokollierung so erfolgt, wie in Abschnitt 4.5 beschrieben wurde. Soll der Test nicht formularbasiert erfolgen, so ist darauf zu achten, daß das Array `super.testObjects` für jeden Test, der mit `suite.addTest(...)` in Zeile 27 hinzugefügt wurde, einen Eintrag enthält. Dieser Eintrag bestimmt, in welches Verzeichnis der Logging-Output geschrieben wird. Ferner wird diese Information in den bisher implementierten Tests dazu benutzt, um das zu testende Formular zu bestimmen. Die eigentliche Implementierung des Tests erfolgt in der Methode `testMethod(...)` (Zeile 32). Damit die Protokollierung wie beschrieben funktioniert, darf zudem in jeder Klasse nur eine Test-Methode implementiert werden. Die Klasse `testtools.TesttoolsTestCase` stellt eine Methode `fillInForm(...)` zur Verfügung, die ein von der Web-Anwendung geliefertes HTML-Formular ausfüllt. Beispiele zur Verwendung dieser Methode finden sich in den Quelldateien der oben beschriebenen Tests.

4.10 Ergebnisüberprüfung

Mit dieser Komponente wird überprüft, ob die Informationen, die in ein Formular eingetragen wurden, von der zu testenden Web-Anwendung korrekt weiterverarbeitet werden. Da diese Überprüfung von der zu testenden Web-Anwendung abhängt, stellt das Framework nur eine Schnittstelle für die Implementierung bereit, welche in dem Java-Interface

`testtools.ResultChecker` definiert wird.

Die Weiterverarbeitung von Formularen kann prinzipiell synchron oder ansynchron erfolgen. Analog dazu muß die Ergebnisüberprüfung ebenfalls synchron oder asynchron erfolgen. Die asynchrone Ergebnisüberprüfung sollte mit Hilfe einer Warteschlange unter Verwendung verschiedener Threads erfolgen. Die Warteschlange enthält die zu überprüfenden Testfälle, bzw. die zur Überprüfung benötigten Informationen über einen Testfall. Bei der Instanziierung der Ergebnisüberprüfung sollte ein Thread gestartet werden, welcher die Warteschlange während der Laufzeit eines Tests abarbeitet. Innerhalb der Testmethode werden dann zwei Operationen benötigt: Nachdem ein Testfall verarbeitet wurde, muß er in die Warteschlange eingefügt werden. Wenn alle Testfälle abgearbeitet wurden, muß das Gesamtergebnis überprüft werden, welches ebenfalls abhängig von der zu testenden Web-Anwendung ist. Der folgende Pseudo-Code verdeutlicht die Benutzung der Ergebnisüberprüfung innerhalb einer Testmethode:

```
public void testMethod() {
   Configuration c = Configuration.getInstance();
   ResultChecker rc = c.createResultChecker(...);
   ...
   // for each testcase
   for (...) {
       // process testcase
       ...
       // enqueue testcase
       rc.enqueueTestCase(...);
   }
   // get results
   Object o = rc.collectResults();
   // analyze results
   ...
}
```

Für eine synchrone Ergebnisüberprüfung kann derselbe Mechanismus verwendet werden. Eine Alternative wäre, eine Methode `checkResult(...)` bereitzustellen, welche einen Wahrheitswert zurückliefert. Im Falle einer fehlerhaften Weiterverarbeitung durch die Web-Anwendung müßte man zur Fehleranalyse allerdings wieder auf eine Methode wie `collectResults()` zurückgreifen oder die Analyse in die Implementierung von `testtools.ResultChecker` verlagern. Im Hinblick auf einen sauberen Entwurf und eine

einheitliche Implementierung haben wir uns dazu entschlossen, für die synchrone und die asynchrone Weiterverarbeitung denselben Überprüfungsmechanismus zu verwenden.

Die Auswahl einer konkreten Implementierung der Ergebnisüberprüfung erfolgt über die Eigenschaft `ResultChecker.implementation`. Da hierzu wie auch schon bei der Auswahl des Testfallgenerators die Java-Reflection-Mechanismen aus dem Paket `java.lang.reflect` verwendet wird, ist es möglich, das Framework mit eigenen Implementierungen zu erweitern, ohne den übrigen Programmcode zu ändern.

Kapitel 5

Ausblick

Das in dieser Arbeit vorgestellte Framework kann für verschiedene Testaufgaben eingesetzt werden. Je nach Art der verwendeten Ergebnisüberprüfung kann mit diesem Framework der Integrations- oder der Systemtest unterstützt werden: Wenn im Beispiel von FlexiTrust die produzierten Zertifikate mit den Zertifikatsanträgen verglichen werden, kann ein Systemtest als Black-Box-Test durchgeführt werden. Wird hingegen nur überprüft, ob ein entsprechender Datensatz für die RA erzeugt wurde, handelt es sich um einen Integrationstest. Da zur Überprüfung eines Datensatzes Wissen über die Implementierung benötigt wird, handelt es sich dabei um einen Grey-Box-Test. Ferner können je nach verwendeter Ergebnisüberprüfung verschieden stark integrierte Subsysteme getestet werden. Neben klassischen Regressionstests können die in der Klasse `testtools.ReplayTest` implementierten Mechanismen dazu benutzt werden, Massen- und Lasttests durchzuführen.

Im begrenzten Rahmen dieser Diplomarbeit konnten nicht alle Funktionen realisiert werden, die zum automatischen Testen von FlexiTrust benötigt werden. Im folgenden möchten wir daher einige Ansatzpunkte für Erweiterungen anführen. Die folgende Aufzählung erhebt dabei weder Anspruch auf Vollständigkeit, noch impliziert sie eine Priorisierung.

- In die Komponente „Testfallgenerator“ müssen Methoden integriert werden, die es erlauben, Zeichenketten gemäß eines bestimten Formats zu generieren.
- Darauf aufbauend kann ein Testfallgenerator implementiert werden, welcher Testfälle erzeugt, die sich zur funktionalen Äquivalenzklassenbildung bzw. zur Grenzwertanalyse eignen.
- Es muß eine Ergebnisüberprüfung implementiert werden. Dazu können eventuell Teile eines bereits bestehenden Testtreibers von Marcus Lippert verwendet werden.

- Da damit zu rechnen ist, daß in naher Zukunft eine Reihe neuer Formulare für FlexiTrust erstellt werden, sollte ein Mechanismus geschaffen werden, der die Instanzen der in Abschnitt 4.1 beschriebenen DTDs hinsichtlich ihrer Konsistenz überprüft. Die Struktur dieser Instanzen wird bereits mit Hilfe der von JAXB bereitgestellten Methoden überprüft. Die Überprüfung des Inhalts ist bisher jedoch nur rudimentär implementiert.

- Es müssen zusätzliche Tests implementiert werden. In einem Szenario, in dem nur ein Zertifikat pro Benutzer erlaubt ist, kann z.B. ein Test implementiert werden, bei dem überprüft wird, wie sich die Anwendung verhält, wenn ein Benutzer, der bereits ein Zertifikat besitzt, einen Antrag auf ein neues Zertifikat stellt.

- Der Logging-Output der Protokollierungskomponente wird zur Zeit in XML-Dateien gespeichert. Diese sollten in HTML-Dateien transformiert werden, um sie in einem Web-Browser betrachten zu können und so ihren Wert für die Fehlersuche zu steigern.

- Prinzipiell können zwar bereits verschiedene Web-Anwendungen mit dem Framework getestet werden, allerdings müssen noch Mechanismen implementiert werden, die ein gleichzeitiges Testen mehrerer Anwendungen ermöglichen.

Wie man sieht, gibt es noch einige Ansatzpunkte zur Erweiterung des in dieser Arbeit vorgestellten Frameworks. Es ist daher wünschenswert, daß die hier begonnene Arbeit fortgesetzt wird und somit einen Beitrag zur Sicherung der Qualität von FlexiTrust leistet.

Anhang A

User Guide

A.1 Overview

A.1.1 What is it?

This framework is an application for testing form based web applications. It has been developed for testing the FlexiTrust servlet, but it can be adapted for testing any other web application that is based on HTML forms. With this framework you can generate test cases for the forms that have to be tested. Stored test cases can be used for replay testing. You can run tests that check the structure and the contents of your forms. Test cases are reusable, i.e they can be converted to run with new versions of your application. If you have a servlet based application that runs with tomcat 4 or higher, you can capture the client input. This enables you to perform replay testing of special values. New forms can be defined and added to the test runs at runtime. The test case generator can be extended. This opens new use cases some of which aren't related to testing at all. A generator for performing boundary analysis could be implemented for example. With a generator that reads input from a database it would be possible to process huge numbers of "real" requests, the input data for which can be extracted from existing ressources (e.g. generating certificates for all staff members using the staff database).

A.1.2 How does it work?

The information about the forms that have to be tested is stored in several XML files the structure of which is defined by four DTDs:

1. `ParameterTypes.dtd`
   An instance of this DTD defines the general attributes of form fields. It can be

understood as a template catalogue.
Example: Across different forms there exist form fields where a client has to fill in her last name. Depending on the context those fields could be differently labelled (e.g. "last name" or "subject name"), but should always have the same attributes (e.g. minimum length of 2, maximum length of 20, valid characters in {a-z, A-Z}). All those fields would then be represented in the instance of `ParameterTypes.dtd` by a single element called "firstName" that has the above attributes.
Since this framework is only used for testing the FlexiTrust servlet by now, and since attributes of parameters are not expected to change between different versions of FlexiTrust, there exists only one instance of `ParameterTypes.dtd`.

2. `Form.dtd`
   Instances of this DTD define the forms that have to be tested. A form consists of one or more pages each of which contains a set of parameters, i.e. form fields. Those parameters have different attributes. A parameter can be required, i.e. it has to be filled in, or it can depend on another parameter, i.e. it is only visible, if the parameter it depends on has been filled in (called "and-dependence") or if the parameter it depends on has not been filled in (called "xor-dependence"). The names of this parameters come from the instance of `ParameterTypes.dtd`. There exists one instance of `Form.dtd` for each form and each version of this form.

3. `TestCase.dtd`
   Instances of this DTD represent the test cases. They contain the data a user would fill into the corresponding HTML form of the application. Each instance of `TestCase.dtd` refers to a certain instance of `Form.dtd`. One difference to an instance of `Form.dtd` is, that instances of `TestCase.dtd` contain only parameters that would be visible in the corresponding HTML form. The other difference is that the names of the parameters in instances of `TestCase.dtd` come from the HTML form.
   Example: Let there be a xor-dependence relationship between two parameters "a" and "b" on two subsequent pages. Both parameters have to be defined in the corresponding instance of `Form.dtd` as well as in the instance of `ParameterTypes.dtd`. If in the current instance of `TestCase.dtd` parameter "a" has been "filled in", parameter "b" must not be contained in this test case.

4. `FormToServletMaping.dtd`
   An instance of this DTD maps names of parameters from the instance of `ParameterTypes.dtd` to names of instances of `TestCase.dtd`. Since there

exists only one instance of ParameterTypes.dtd there also exists one instance of FormToServletMapping.dtd, but one parameter name in the instance of ParameterTypes.dtd can map to several parameter names of instances of TestCase.dtd representing different forms or different versions of a form. The reason why instances of Form.dtd use parameter names from the instance of ParameterTypes.dtd rather than names from corresponding instances of TestCase.dtd is the better comparability of different forms and different versions of forms.

A.2 Installation of the framework

A.2.1 System Requirements

- JDK 1.2.2_008 or later
- Ant 1.4 or later and the corresponding optional tasks (both available at http://jakarta.apache.org/builds/ant/release/v1.4/bin/).

A.2.2 Installation

1. Unpack and install the gzipped tar.
   Let $HOME represent the extraction directory. Unpacking the archive will then create a directory $HOME/testtools
2. Update the java classpath.
   To be able to run the tests, the following jars in the directory $HOME/testtools/lib have to be added to the classpath:
   - junit.jar
   - xalan.jar
   - xml-apis.jar
   - xercesImpl.jar
3. Adjust the properties file.
   In the file $HOME/testtools/testtools.properties set the property "home" to the extraction directory:
   - home=$HOME

A.3 Getting started

This framework uses the build tool Ant to invoke the different parts of the application. Necessary properties are configured in the property file `$HOME/testtools/testtools.properties`.

A.3.1 The build file

The build file `$HOME/testtools/build.xml` defines several targets. To invoke a target simply change to the directory `$HOME/testtools` and type

- `ant <target-name>`

at the command prompt (assuming you have got a working Ant installation). The following targets are important for users:

- `javadoc`: generate javadocs
- `deploy_filter`: install the filter for capturing client input
- `run`: run a single class; invoke with property classname;
  example: `ant -Dclassname=testtools.ReplayTest run` will run the class `testtools.ReplayTest`
- `convert`: convert test cases
- `generate`: generate testcases
- `test`: run all tests
- `replay`: run the replay test

A.3.2 Generating test cases

The following properties have to be set:

- `TestCaseGenerator.forms`: comma separated list of names of the forms for which test cases will be generated
- `TestCaseGenerator.modes`: modes in which test cases will be generated (cf. javadocs)

- TestCaseGenerator.number: number of test cases that will be generated for each form and in each mode

The command ant generate will generate test case files named testcase_<formName>_v<version>_<mode>_<dateOfGeneration>_<n>.xml where n is in [0.. ${TestCaseGenerator.number}-1]. The test cases are stored in the directory ${home}/${testCaseDir}. The logging output is stored in the directory ${home}/${recordDir}/testtools.TestCaseGenerator.

A.3.3 Converting test cases

The following properties have to be set:

- TestCaseConverter.forms: comma separated list of names of the forms on which test cases that will be converted have to be based
- TestCaseConverter.toVersion: version from which test cases will be converted
- TestCaseConverter.fromVersion: version to which test cases will be converted

The command ant convert will scan the directory ${home}/${testCaseDir} for test cases with versionTestCaseConverter.toVersion and convert them to version TestCaseConverter.fromVersion. The converted test cases are also stored in ${home}/${testCaseDir}. The file names of the converted test cases are the same as the old file names except for the version part (..._v<version>_...). The logging output is stored in the directory
${home}/${recordDir}/testtools.TestCaseConverter.

A.3.4 Running tests

The following property may be set:

- Tests: comma separated list of test names for which a list of forms may be specified. Test name is the complete name of the respective class including the preceding package name testtools.

For each item `<Test>` in `${Tests}` a property `<Test>.forms` may be defined which contains a comma separated list of form names for which the test `<Test>` should be run. For tests not contained in `${Tests}` or tests for which no property `<Test>` is specified, the forms in `${Forms}` are tested.
For running the replay test the following properties have to be set:

- `currentUsers`: number of users filling in forms concurrently
- `workload`: number of forms each user has to fill in

The command `ant test` will run all tests in the package `testtools` the classnames of which end with "`Test`". The tests are run subsequently. For each test all specified forms (see above) will be tested except for `testtools.ReplayTest`. For this test the test cases in the directory `${home}/${testCaseDir}` the version of which matches the property `currentVersion` are processed, but at most `${workload} * ${currentUsers}` test cases. The logging output of a test is stored in the directory `${home}/${recordDir}/<date>/<testName>` where `<date>` date is the current date formatted according to the property `datePattern` and `<testName>` is the name of the test (e.g. `testtools.RequiredTest`). For each test a summary of the results is written to the directory `${home}/${recordDir}/<date>/summary`. Additionally a summary of the results of all tests is generated and stored to the file `${home}/${recordDir}/<date>/index.html`. Currently the following tests are implemented:

- `testtools.RequiredTest`: tests the required fields in a form
- `testtools.DependenceTest`: tests dependence relationships between form fields
- `testtools.IllegalValuesTest`: tests illegal values in form fields
- `testtools.ReplayTest`: replays stored test cases

A.3.5 Running single tests

Single tests can be run with the command `ant -Dclassname=<testName>`. The logging output is stored as described above, but no summary is generated.

A.3.6 Capturing client input

Capturing of client input currently only works for web applications that run as servlets in a servlet container that implements the servlet specification 2.3 or later (e.g. Tomcat 4). To use this feature, you have to set the property `webAppLibDir` to an absolute path, where your servlet container is looking for libraries for your web application (e.g. the `WEB-INF/lib`-direcory of your web application. Additionaly you have to insert the following lines into the web.xml-file of the web application (directly beneath the opening web-app tag):

```
<!-- Define filter -->
<filter>
    <!-- arbitrary expressive name -->
    <filter-name>CaptureFilter</filter-name>
    <!-- class name of filter -->
    <filter-class>testtools.CaptureFilter</filter-class>
</filter>

<!-- Define filter mappings for the defined filter -->
<filter-mapping>
    <!-- filter name from above -->
    <filter-name>CaptureFilter</filter-name>
    <!-- pattern, for which filter will be applied;
      in this example, all servlets in this context
      will be filtered -->
    <url-pattern>/servlet/*</url-pattern>
</filter-mapping>
```

Finally you have to make sure, that the doctype-tag of the web.xml file looks like that:

```
<!DOCTYPE web-app PUBLIC
  "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
  "http://java.sun.com/j2ee/dtds/web-app_2_3.dtd">
```

All you have to do now is restaring your servlet container or reloading the web application (via the manager web app in tomcat 4). Captured testcases will be stored in the directory `${home}/${testCaseDir}`.

A.3.7 Customizing the framework

The framework can be customized by changing the properties in `${home}/${applicationDir}/testtools.properties`. In addition to the properties explained above you can change some directory names to your needs (don't change the property `applicationDir` since the application won't find the property file anymore!) and adjust the logging behaviour, e.g the log levels or the layout of the logging output. All properties are explained in the file.

A.4 Adding new profiles (forms)

Sooner or later you'll want to add new profiles that should be tested by the framework. This section walks you through writing a new form definition and adjusting the mapping, the parameter types and the configuration. The framework is shipped with a definition of the "Softtoken" application profile. The definition (an instance of `Form.dtd`) is stored in `${home}/${applicationDir}/${xmlDir}/Softtoken.2.5.xml`. The instances of `ParameterTypes.dtd` and `FormToServletMapping.dtd` are stored in `${home}/${applicationDir}/${xmlDir}/ParameterTypes.xml` and `${home}/${applicationDir}/${xmlDir}/FormToServletMapping.xml`.
In this example we'll create a definition for the "Chipkarte" application profile. The definition of this new profile will be stored in the file `${home}/${applicationDir}/${xmlDir}/Chipkarte.2.5.xml`. The file names are arbitrary, but to be better able to distinguish different form definitions the naming convention should be something like `<profileName>.<version>.xml`. The best way to create the new profile is to simultaneously edit the files `${home}/${applicationDir}/${xmlDir}/ParameterTypes.xml` and `${home}/${applicationDir}/${xmlDir}/FormToServletMapping.xml` and to execute the servlet.
At the beginning our empty form definition should look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE Form SYSTEM "../Form.dtd">

<Form name="form2" version="2.5">
</Form>
```

In general the values for the name and the version attributes are arbitrary, but since we already defined the "Softtoken" profile (`Form.name="form1"`) for this servlet, we have

to use the same version for the new profile. The first page in the servlet only contains the drop down box for choosing the profile. The name of the drop down box in the form (cf. HTML source) is `DBChooserFormName`. Drop down boxes and choice fields correspond to `OptionType` elements in `${home}/${applicationDir}/${xmlDir}/ParameterTypes.xml`. There is already an `OptionType` element defined the attribute `parameterName` of which has the value "profile". The `enum` attribute defines the valid values for the corresponding field. Since this attribute already contains the value "Chipkarte" we don't have to update `${home}/${applicationDir}/${xmlDir}/ParameterTypes.xml`. Now we add the first `Page` element which contains only one `Parameter` element to the `Form` element:

```
<Page name="Choose profile" method="GET">
  <Parameter
    name="profile"
    required="true"
  />
</Page>
```

The name of the `Page` element is arbitrary. It is just used to increase the "readability" of the form. The value of the atribute `method` has to be equal to the method of the corresponding HTML form. The attribute `name` of the `Parameter` element comes from the corresponding `ParameterType` element in `${home}/${applicationDir}/${xmlDir}/ParameterTypes.xml`. If the parameter is required, the value of the required attribute has to be set to "true", else it has to be set to "false". After adding a `Parameter` element to the Form, the mapping for this parameter has to be added to `${home}/${applicationDir}/${xmlDir}/FormToServletMapping.xml`. There already exists a mapping for the `Parameter` with the name "profile", so you just have to insert another `ServletParameter` element to the corresponding `FormParameter` element:

```
<FormParameter name="profile">
  ...
  <ServletParameter
    form="form2"
    version="2.5"
    name="DBChooserFormName"
```

```
  />
</FormParameter>
```

Proceed with the next page. Add a new Page element and add the Parameter elements `firstName`, `lastName`, `organizationalUnit`, `organizationName`, `country`, `eMail`, `caCreatesPIN` and `caCreatesRevocPass`. All parameters except for `organizationalUnit`, `caCreatesPIN` and `caCreatesRevocPass` are required. Have a look at the corresponding `ParameterType` Objects in `${home}/${applicationDir}/${xmlDir}/ParameterTypes.xml`. `StringType` parameters can have optional attributes `min` and `max` that define the minimum and maximum length of the string. The default values are defined in `ParameterTypes.dtd`. Furthermore they must have an attribute `parse` that defines the type of the string (i.e. the valid characters for the corresponding form field). `DateType` parameters have optional attributes `min` and `max` that define the minimum and maximum number of days until this date from the day a corresponding form field is filled in. `CheckboxType` parameters have an attribute that defines the value of the corresponding form field, if the checkbox is checked. `OptionType` parameters have an attribute `enum` that defines the different options for this form field. All parameter types have an attribute `number` that defines the number of values the corresponding form fields accepts. At the moment, the attributes of parameter types are not used to validate test cases but only for generating test cases.
Now add the mappings

- `firstName=>Subject_FirstName`,
- `lastName=>Subject_LastName`,
- `organizationalUnit=>Subject_OrganizationalUnit`,
- `organizationName=>Subject_Organization`,
- `country=>Subject_Country`,
- `eMail=>subject_email`,
- `caCreatesPIN=>PINCreatesCA` and
- `caCreatesRevocPass=>RevocCreatesCA`

to the mapping. The form fields labelled "PIN" and "Revokations Passwort" on the next page are only visible, if the checkboxes labelled "CA erzeugt PIN" and "CA erzeugt

Revokationspasswort" have not been checked. Now we have got a dependence relationship. In this case you have to enter the parameter name of the parameter this parameter depends on into the depends attribute. The `dependsAnd` attribute is true if the dependence is an and-dependence and false if it is an xor-dependence. There is another special case in this page. The drop down box labelled "Key length" would match with the `ParameterType keyLength`. But this parameter type accepts the values "1024", "1536" and "2048" where the field in this page only accepts the value "1024". Now we have to add a new `ParameterType` element:

```
<OptionType
  parameterName="keyLengthChipCard"
  enum="1024"
/>
```

The `Page` element should look like this:

```
<Page name="Key parameters" method="POST">
    <Parameter
      name="keyLengthChipCard"
      required="true"
    />

    <Parameter
      name="pin"
      required="true"
      depends="caCreatesPIN"
      dependsAnd="false"
    />

    <Parameter
      name="revocPass"
      required="true"
      depends="caCreatesRevocPass"
      dependsAnd="false"
    />
  </Page>
```

Finally you have to add the mappings for `pin` and `revocPass` and to insert a new `FormParameter` element:

```
  <FormParameter name="keyLengthChipCard">
    <ServletParameter
      form="form2"
      version="2.5"
      name="Keylen"
    />
  </FormParameter>
```

After you are done, you can save the file. It should look similar to this:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE Form SYSTEM "../Form.dtd">

<Form name="form2" version="2.5">
  <Page name="" method="GET">
    <Parameter
      name="profile"
      required="true"
    />
  </Page>

  <Page name="Personal Data" method="POST">
    <Parameter
      name="firstName"
      required="true"
    />

    <Parameter
      name="lastName"
      required="true"
    />

    <Parameter
      name="organizationalUnit"
      required="false"
    />
```

```
    <Parameter
      name="organizationName"
      required="true"
    />

    <Parameter
      name="country"
      required="true"
    />

    <Parameter
      name="eMail"
      required="true"
    />

    <Parameter
      name="caCreatesPIN"
      required="false"
    />

    <Parameter
      name="caCreatesRevocPass"
      required="false"
    />
  </Page>

  <Page name="Key parameters" method="POST">
    <Parameter
      name="keyLengthChipCard"
      required="true"
    />

    <Parameter
      name="pin"
      required="true"
      depends="caCreatesPIN"
```

```
    dependsAnd="false"
  />

  <Parameter
    name="revocPass"
    required="true"
    depends="caCreatesRevocPass"
    dependsAnd="false"
  />
 </Page>
</Form>
```

Now all you have to do is inserting the properties `form2.2.5=Chipkarte.2.5.xml` and `form2.2.5.name=Chipkarte` to testtools.properties and adding form2 to the property forms. If you have defined any `<testName>.forms` properties you have to add `form2` to those properties to include this form in the tests. If you would have defined a profile for a new version, you would have had to add the version number to the property `versions` and you might have wanted to change `currentVersion` to the new version.

A.5 ToDo

- Check if requests are processed by FlexiTrust (only implemented rudimentarily).
- Check the consistency of instances of `ParameterTypes.dtd`, `FormToServletMapping.dtd`, `Form.dtd` and `TestCase.dtd` (not fully implemented).
- Integrate tool for initializing the test environment (i.e. reset FlexiTrust to a defined state).
- Transform logging output to HTML files.
- Implement additional test case generators.
- Implement additional tests.

A.6 Feedback

Please send your feedback to `Heiko.Hornung@epost.de`.

Literaturverzeichnis

[ANSa] Glossary of Software Engineering Terminology (ANSI/IEEE STD 729-1983).

[ANSb] IEEE Standard for Software Test Documentation (ANSI/IEEE STD 829-1983).

[ANT02] Homepage des Jakarta-Ant-Projekts. http://jakarta.apache.org/ant/ (2002).

[Bal98] HELMUT BALZERT. „Lehrbuch der Software-Technik: Software-Management, Software-Qualitätssicherung, Unternehmensmodellierung“. Spektrum, Akad. Verlag, Heidelberg; Berlin (1998).

[BG98] KENT BECK UND ERICH GAMMA. Test-Infected: Programmers Love Writing Tests. http://www.junit.org/junit/doc/testinfected/testing.htm (1998).

[BS01] AXEL BARTRAM UND UWE STRICKER. Testverfahren - (Web-)Anwendungen testen. *iX Magazin für professionelle Informationstechnik* Seiten 46–50 (November 2001).

[Buc01] JOHANNES BUCHMANN. „Einführung in die Kryptographie“. Springer, Berlin u.a., 2., erw. Auflage (2001).

[Dam01] JENS DAMBRUCH. „OpenRA - Framework zur flexiblen Formularverarbeitung: Formulare, Verarbeitungslogik und Ergebnisse“. Diplomarbeit, Technische Universität Darmstadt (2001). ftp://ftp.informatik.tu-darmstadt.de/pub/TI/reports/dambruch.diplom.OpenRA.pdf.

[DIN86] Begriffe der Qualitätssicherung und Statistik: Grundbegriffe der Qualitätssicherung, DIN 55350, Teil 11 (1986).

[FLE02] FlexSecure Trustcenter-Software FlexiTrust. http://www.flexsecure.de/ojava/flexitrust.html (2002).

[How01] RICK HOWER. Software QA and Testing Frequently-Asked-Questions, Part 1. http://www.softwareqatest.com/qatfaq1.html (2001).

[HTT02] HttpUnit-Homepage. http://www.httpunit.org (2002).

[HU01] ANDREAS HOLUBEK UND FRANK ULBRICHT. Die Java Architecture for XML Binding (JAXB) in der Praxis. *Java Magazin* Seiten 27–29 (Oktober 2001).

[IEE] IEEE Standard for Software Quality Assurance Plans (IEEE STD 730-1989).

[IRM02] IRMC Approved Principles, Policies, and Standards. http://irmc.state.nc.us/documents/approvals/ (2002).

[ISO00] Deutsche Rohübersetzung zu ISO/FDIS 9000:2000-09 mit englischer Originalfassung. In „Qualitätsmanagementsysteme“. Beuth, Berlin (2000).

[JAX01] The Java Architecture for XML Binding User’s Guide - Early Access Draft. http://java.sun.com/xml/jaxb/jaxb-docs.pdf (Mai 2001).

[Kle01] MANUEL KLEIN. Test-Tools für Java im Überblick. *Java Magazin* Seiten 42–47 (Oktober 2001).

[LOG02] Homepage des Jakarta-Log4J-Projekts. http://jakarta.apache.org/log4j/ (2002).

[Miz83] YUKIO MIZUMO. Software Quality Improvement. *IEEE Computer* Seiten 62–72 (März 1983).

[Sch01] MARKUS SCHUSTER. „OpenRA - Framework zur flexiblen Formularverarbeitung: Darstellung, Transport und Produktion“. Diplomarbeit, Technische Universität Darmstadt (2001). ftp://ftp.informatik.tu-darmstadt.de/pub/TI/reports/schuster.diplom.pdf.

[SER02] Java Servlet 2.3 and JavaServer Pages 1.2 Specifications. http://www.jcp.org/jsr/detail/53.jsp (2002).

[TOM02] Homepage des Jakarta-Tomcat-Projekts. http://jakarta.apache.org/tomcat/ (2002).

[Wal01] ERNEST WALLMÜLLER. „Software-Qualitätssicherung in der Praxis: Software-Qualität durch Führung und Verbesserung von Software-Prozessen“. Carl Hanser Verlag, München; Wien, 2., völlig überarb. Auflage (2001).

[Wie01] ALEXANDER WIESMAIER. „FlexiTrust CA - Ein flexibles, skalierbares und dynamisch erweiterbares Trustcenter“. Diplomarbeit, Technische Universität Darmstadt (2001). ftp://ftp.informatik.tu-darmstadt.de/pub/TI/reports/wiesmaier.diplom.flexiTrustCa.ps.gz.