Fachhochschule der Wirtschaft
-FHDW-
Paderborn

**Diplomarbeit**

**Thema:**

# Konzeption und Realisation eines Tools zur automatisierten Durchführung von Softwaretests am Beispiel der Programmiersprache Java

Prüfer:
Herr Prof. Dr. Carsten Weigand
Herr Prof. Dr. Werner Oertmann

Verfasser:

Daniel Becker
Im Obstgarten 5
32657 Lemgo

6. Hochschulquartal
Studiengang Wirtschaftsinformatik

Eingereicht am:
04. Oktober 2001

*Wenn man als einziges Werkzeug einen Hammer besitzt, neigt man dazu, jedes Problem als Nagel zu betrachten.*

- Abraham Maslow -

# Inhaltsverzeichnis

**Seite**

# Abbildungsverzeichnis

**Abbildung** **Seite**

## Abkürzungsverzeichnis

| | |
|---|---|
| **dtsch.** | deutsch |
| **engl.** | Englisch |
| **evtl.** | eventuell |
| **etc.** | et cetera |
| **f.** | folgend |
| **ff.** | fortfolgend |
| **ggf.** | gegebenenfalls |
| **ggT** | größter gemeinsamer Teiler |
| **i.a.** | im Allgemeinen |
| **i.d.R.** | in der Regel |
| **o.a.** | oben angeführt |
| **s.** | siehe |
| **S.** | Seite |
| **s.a.** | siehe auch |
| **sog.** | sogenannt |
| **u.a.** | unter anderem |
| **usw.** | und so weiter |
| **u.U.** | unter Umständen |
| **vgl.** | vergleiche |
| **z.B.** | zum Beispiel |
| **z.T.** | zum Teil |

# Glossar

**3-Tier**: Bezeichnung einer 3-Schichtenarchitektur. Beschreibt die funktionale Unterteilung eins Softwareprojektes in Schichten (horizontale Trennung). Im Gegensatz dazu ist auch eine fachliche Unterteilung möglich (vertikale Trennung).

**API**: Akronym für „Advanced Programmers Interface“. Unter mit diesem Begriff gekennzeichneten Routinen- und Methodensammlungen finden sich i.d.R. Möglichkeiten, über den Rahmen der Standardfunktionalität einer Programmiersprache hinaus Systemfunktionen (z.B. über das Windows-API) oder spezielle Funktionen der Programmiersprache (z.B. über das Reflection-API) aufzurufen.

**ASCII**: Abkürzung für „American Standard Code for Information Interchange“. Dieser 7-Bit-Code dient der Darstellung von Buchstaben, Ziffern und Sonderzeichen. Das im Byteformat freie 7. Bit (mit Wert 128) wird entweder als Paritätsbit verwendet oder dient der Erweiterung auf einen 8-Bit-Code. Da der ursprüngliche ASCII-Code nicht mehr zur Darstellung aller auf der ganzen Welt verbreiteten Zeichen ausreicht, wurde er auf einen 32-Bit-Code erweitert, der Unicode genannt wird. Damit Abwärtskompatibilität gewahrt bleibt, kann Unicode aber auch mit 8 oder 16 Bit dargestellt werden. Näheres findet sich unter http://www.unicode.org.

**Atomare Datentypen**: Unter atomaren Datentypen versteht man die Datentypen einer Programmiersprache, die nicht als Objekt angesehen werden. Unter Java sind dies neun Datentypen: boolean, byte, char, short, int, long, float, double und void, wobei „void“ kein echter Datentyp ist. Da diese atomaren Datentypen eigentlich nicht in das Konzept einer objektorientierten Programmiersprache passen, gibt es in Java neben den atomaren noch entsprechende objektorientierte Datentypen, sog. Wrapper-Klassen. Statt atomar wird auch der Begriff „primitiv“ verwendet. Jedoch wird der Begriff des „primitiven Datentyps“ in dieser Arbeit anders gebraucht. Um Verwechslungen vorzubeugen, wird daher immer von „atomaren Datentypen“ gesprochen, wenn die primitiven Datentypen von Java gemeint sind.

**Callback-Function**: In manchen Fällen ist es notwendig, daß eine Klasse über das Eintreten eines Ereignisses benachrichtigt wird. Das Ereignis wird oft jedoch in einer anderen Klasse auftreten. Beispielsweise sollte ein Dialog darüber benachrichtigt werden, wenn die gerade angezeigten Daten nicht mehr aktuell sind und damit neu gelesen werden müssen. Das Daten nicht mehr aktuell sind, weiß i.d.R. jedoch nur die Klasse, welche gerade die neuen Daten in die Datenbank geschrieben hatte. Es ist daher zweckmäßig, bei der für Datenveränderungen verantwortlichen Klasse eine Methode aus der Dialogklasse zu registrieren, welche nach einer Veränderung des Datenbestandes aufgerufen wird und automatisch die angezeigten Daten aktualisiert. Diese Technik wird unter dem Begriff Model-View-Controller (MVC) zusammengefaßt.

## Glossar (Fortsetzung)

**DOM**: Akronym für „Document Object Model“. Ein auf dieser Technik basierender XML-Parser liest das komplette XML-Dokument ein und liefert es in der Form eines Baumes. Dieser kann dann von der Verarbeitungssoftware traversiert werden.

**DTD**: Akronym für „Document Type Definition“[1]. Mit dieser auf ASCII-Text basierenden Datei wird die zulässige Syntax eines XML-Dokumentes festgelegt.

**Hashtable**: Für die Programmiersprache Java existieren zahlreiche Klassen, die zum Speichern von Objekten geeignet sind, wie z.B. den Vector, der ein dynamisches Array darstellt, sowie auch die Hashtable. Diese speichert ein Objekt unter einem Schlüsselobjekt ab, das in aller Regel ein String ist. So kann unter einem eindeutigen Namen ein Objekt gespeichert und auch wieder aufgefunden werden. Um dies möglichst schnell erledigen zu können, bedient sich diese Klasse eines Hash-Verfahrens zur Ermittlung des Index eines Schlüsselobjektes.

**HTML**: Abkürzung für „Hypertext Markup Language“, eine für das Internet entwikkelte (Web-)Seitenbeschreibungssprache.

**JVM**: Akronym für „Java Virtuell Machine“, auf dtsch. „Java Virtuelle Maschine“. Da Java für sich in Anspruch genommen hat, daß mit Java erstellte Programme ohne neue Kompilierung auf jeder Plattform lauffähig sein sollen, war es nötig, einen Interpreter zu entwickeln. Dieser Interpreter ist für die jeweiligen Zielplattformen erhältlich und interpretiert den auf irgendeiner Plattform erstellten Byte-Code immer gleich. Damit wird allerdings ein großer Vorteil, nämlich die Portabilität von Programmen, durch einen nicht unerheblichen Geschwindigkeitsnachteil gegenüber Maschinencode erkauft, wie er aus Assembler- oder C/C++-Programmen entsteht.

**Multithreading**: Zu dtsch.: „nebenläufige Verarbeitungsschritte“ (engl. threads)[2]. Meint die gleichzeitige Verarbeitung von einzelnen Aufgaben in einem Programm.

**SAX**: Akronym für „Simple API for XML“. Ein auf dieser Technik basierender XML-Parser liest nicht das gesamte XML-Dokument ein, wie es der DOM-Parser tun würde, sondern liefert jedes gefundene Tag via Callback-Function an die Software, welche sofort interpretieren kann. Gerade für sehr große Dokumente ist diese Technik empfehlenswert.

**SGML**: Akronym für „Standard Generalized Markup Language“, Oberklasse von allen Markup Languages, wie z.B. XML.

---

[1] vgl. http://www.w3schools.com/dtd/dtd_intro.asp, Introduction to DTD

[2] vgl. Hansen, H.R., 1998, S. 878f.

## Glossar (Fortsetzung)

**Tag**: Tags sind in Kleiner- und Größerzeichen eingefaßte Texte und stellen die Sprachelemente der Markup Languages dar. In XML werden die erlaubten Tags durch eine DTD beschrieben. Beispiel eines Tags: `<br>`, `<p>`. Jedes öffnende Tag benötigt nach der Sprachstandardisierung der W3C immer ein schließendes Tag. Umschließt ein Tag nichts, dann kann es zu seinem eigenen schließendem Tag werden. Beispielsweise ist gutes HTML: `<br />`, `<p />`, was äquivalent zu `<br></br>`, `<p></p>` ist.

**UML**: Akronym für „Unified Modelling Language“. Ein Modellierungstool zum erstellen von Objektmodellen.

**WML**: Abbreviatur für „Wireless Markup Language“ und findet bei WAP, dem „Wireless Application Protocol“ Anwendung.

**Wrapper-Klasse**: Als Wrapper-Klassen werden Klassen bezeichnet, die einen primitiven Datentypen umhüllen. So umschließt z.B. die Klasse `java.lang.Integer` den primitiven Datentyp `int`. Auch `java.lang.String` kann man als Wrapper bezeichnen, da diese Klasse ein Array aus Charactern umhüllt.

**XML**: Akronym für „Extensible Markup Language“[3]. Eine Sprache, die wie HTML auf Tags basiert. Mit der Hilfe einer DTD ist XML eine selbsterklärende Sprachen und findet vor allem bei der Speicherung von Daten Anwendung.

**XML4J**: Dieser von IBM entwickelte XML-Parser vereint, ebenso wie sein Konkurrenzprodukt XERCES von der Apache Group, sowohl einen DOM- wie auch einen SAX-Parser. Mit ihrer Hilfe können XML-Dateien eingelesen und verarbeitet werden.

**XP**: Abkürzung für „Extreme Programming“. Ein neues Konzept für die Entwicklung von Software. Beim XP bedient man sich in aller Regel einer objektorientierten Sprache, um das Design mit Modellierungstools wie UML durchführen zu können und optimale Voraussetzungen für Arbeitsteilung vorzufinden.

---

[3] vgl. http://www.w3schools.com/xml/xml_whatis.asp, XML Introduction - What is XML

# 1 Einleitung

## *1.1 Problemstellung*

Das Testen von Software stellt eine der wichtigsten und umfangreichsten Aufgaben dar, der sich die Beteiligten eines Softwareentwicklungsprozeß stellen müssen. Die Notwendigkeit von Tests ergibt sich nicht allein aus der Sicherstellung der Funktionsfähigkeit eines Softwareproduktes, sondern dient vor allem der Sicherstellung von Qualitätsmerkmalen, an denen sich Softwarefirmen messen lassen müssen.

Im klassischen Softwareentwicklungsprozeß gilt zudem, daß jeder bereits in der Entwicklung aufgedeckte Fehler leicht und kostengünstig zu entfernen ist, wogegen sich solche, erst nach der Auslieferung in der Wartungsphase auftretende Fehler nur schwer beseitigen lassen und somit höhere Kosten verursachen.

Betrachtet man die Aufteilung der anfallenden Kosten während der Entwicklung von Software, so zeigt sich, daß über 50% der Gesamtkosten[4] allein auf die Durchführung von Tests entfallen. Das liegt unter anderem daran, daß schon die Ausarbeitung von geeigneten Testfällen sehr viel Zeit in Anspruch nimmt. Doch auch die Testdurchführung und die anschließende Überprüfung der Ergebnisse ist sehr zeit- und damit auch kostenintensiv.

## *1.2 Der Wandel im Softwareentwicklungsprozeß*

Die konventionelle Entstehung der Kosten für Redesign, Erweiterungen und die Fehlerbeseitigung im Softwareentwicklungsprozeß, welche in Abhängigkeit der Zeit stetig steigen, soll durch ein neues Konzept vermieden werden.[5] Dieses unter dem Namen „Extreme Programming“ (XP) bekannte Konzept versucht, die Kostenkurve möglichst konstant zu halten.

Im klassischen Entwicklungsprozeß wurde versucht, die Kosten dadurch zu minimieren, daß bereits in der ersten Stufe der Entwicklung einer Software Flexibilität für spätere Erweiterungen eingeplant wird. Unter der gestellten Prämisse, daß die Implementie

---

[4] vgl. Myers, G.J., 1979, S.VII, Vorwort

[5] vgl. Oestereich, B., 2001, S. 26 ff.

rung zu diesem Zeitpunkt noch relativ wenig kostet, erscheint dies vernünftig. Dadurch wurde der Code jedoch auch umfangreicher, als eigentlich notwendig.

Das Konzept des XP versucht nun, die Änderungskosten für spätere Erweiterungen zu minimieren, indem nur die Funktionen implementiert werden, welche vom Auftraggeber gefordert wurden[6]. Dabei soll die Struktur des Codes so einfach gehalten werden, daß die gestellte Aufgabe gerade erfüllt wird.

Dieses Vorgehen scheint auf den ersten Blick widersprüchlich zu sein. Bei näherer Betrachtung wird dagegen deutlich, daß unkomplizierter Code wesentlich leichter zu warten ist und zudem schneller einem nachträglichen Redesign unterzogen werden kann.

Da beim XP der Code sukzessive erweitert wird und damit ständig einem Redesign unterworfen ist, welches auch eine komplette Umstrukturierung des dem Projekt zugrunde liegenden Objektmodels zur Folge haben kann, ist es äußerst wichtig, alle bisher durchgeführten Tests ständig zu wiederholen. Damit wird gewährleistet, daß sich die Änderungen nicht auf das bisher getestete Verhalten der Software auswirken.

Um diese aufwendigen Tests kostengünstig durchführen zu können, ist die Automation von Softwaretests beim XP unumgänglich.

## *1.3 Gründe für die Automation von Softwaretests*

Das XP sieht vor, daß für jede zu implementierende Funktion erst die Testfälle zu entwickelt sind, welche den Code überprüfen werden. Während der Entwicklung des neuen Codes zur Erfüllung der gestellten Aufgabe wird nun regelmäßig mit den entsprechenden Testfällen überprüft, ob die Implementierung fehlerfrei erfolgt. Nach dem erfolgreichen Durchlauf der Tests werden die Testfälle jedoch nicht gelöscht, sondern für spätere Erweiterungen aufbewahrt.

Mit dieser Technik sammelt sich im Laufe des Entwicklungsprozesses ein großes Testfallarchiv an, mit dem die gesamte Software getestet werden kann. Nach jeder Veränderung des Codes kann nun das gesamte Testfallarchiv abgearbeitet werden, um nicht nur die Änderung selbst, sondern auch potentielle Seiteneffekte zu testen und zu finden.

---

[6] s. Oestereich, B., 2001, S.28, das „Story"-Konzept des XP

Unter diesem Aspekt ist es sinnvoll, die Durchführung eines solchen umfangreichen Tests möglichst automatisiert ablaufen zu lassen. Andernfalls ist davon auszugehen, daß ein Entwickler, sei es aus Bequemlichkeit oder weil er meint, es besser zu wissen, nur die Testfälle erneut ausführen wird, die seine Änderungen direkt tangieren. Bei diesem Vorgehen ist es jedoch unmöglich, Seiteneffekte zu finden.

## *1.4 Vorgehensweise in dieser Arbeit*

An dieser Stelle wird die vorliegende Diplomarbeit ansetzen. Ziel ist es, ein Tool zu entwickeln, daß dem Entwickler die Durchführung der Tests und den anschließenden Vergleich zwischen Soll- und Ist-Daten abnimmt.

Durch die computergestützte Testdurchführung ist es möglich, die Tests wie ein Experiment jederzeit zu wiederholen, um die noch bestehende Fehlerfreiheit der Software zu überprüfen und Seiteneffekte durch Ergänzungen und Weiterentwicklungen an der Software auszuschließen.

Im Verlauf der Arbeit wird an kleinen Beispielen demonstriert, wie Tests prinzipiell automatisiert werden können. Dabei werden mögliche Probleme, welche durch die Automation entstehen können, aufgezeigt und einer Lösung zugeführt. Sukzessive soll gleichzeitig auf die Programmierung eines Tools hingearbeitet werden, das den Entwickler bei der Durchführung von automatischen Tests unterstützen kann.

Unter anderem wird die Entwicklung entsprechender Vergleichsmethodiken für die jeweils zu überprüfenden Ergebnisdaten eine Rolle spielen, wie auch die Realisation von flexiblen Übergabeparametern.

Dabei wird darauf verzichtet, sowohl die üblichen Techniken zur Ermittlung von sinnvollen Testfällen zu erläutern, als auch auf die Vielzahl möglicher Verfahren für die Testdurchführung im Detail einzugehen. In dieser Arbeit wird ein eher praktischer Ansatz zur Entwicklung eines Tools verfolgt werden.

# 2 Prinzipien der Automation von Softwaretests

Die wichtigste Grundlage für das weitere Vorgehen bildet die Definition des Begriffes „Testen“.

*„Testen ist der Prozeß, ein Programm mit der Absicht auszuführen, Fehler zu finden.“*[7]

In diesem Zusammenhang soll folgende begriffliche Definition gelten: ein Test wird als „nicht erfolgreich“ bzw. „negativ“ bezeichnet, wenn kein Fehler gefunden wurde, und als „erfolgreich“ bzw. „positiv“, wenn Fehler aufgedeckt wurden.

Normalerweise würde man Testen als einen Prozeß definieren, der geeignet ist, die Fehlerfreiheit eines Programmes oder die Erfüllung der gestellten Aufgabe zu demonstrieren. Jedoch wäre diese Definition in der Praxis kaum zu erfüllen, denn bereits bei kleinen Programmen ist die notwendige Anzahl an Testfällen, die benötigt würden, um die Fehlerfreiheit eines Programmes zu beweisen, nahezu unendlich groß.

Auch die Erkenntnis, daß die Anzahl der Fehler in einer Anwendung proportional zu ihrer Komplexität steigt, kann nicht dazu dienen, Programmen mit geringer Komplexität bereits Fehlerfreiheit zu attestieren. Dennoch wird die Reduzierung der Komplexität gerne als vorbeugende Maßnahme verwendet, um bereits im Vorfeld Fehler zu vermeiden.

Auf der anderen Seite behaupten einige Pessimisten, daß es das fehlerfreie Programm gar nicht gibt. Diese Aussage führt zu der Konsequenz, daß sogar das kleine, meist Programmieranfängern als erstes Übungsprogramm dargebotene „Hallo Welt“-Programm nicht fehlerfrei ist, obwohl doch gerade dieses Programm ein gutes Beispiel für geringe Komplexität und daraus resultierende geringe Fehleranfälligkeit darstellt.

Beide Definitionen sind in der Realität zutreffend, denn sowohl das Entwickeln kleiner überschaubarer Methoden und Klassen und die daraus resultierende lokale geringe Komplexität, wie auch das Wissen um die trotz der Durchführung umfangreicher Tests verbleibenden Fehler, kann bei der Entwicklung von Testfällen und der gebotenen Vorsicht bei der Aussage, das Produkt sei fehlerfrei, sehr hilfreich sein.

---

[7] s. .Myers, G.J., 2001, S. 4

Sehr oft werden Fehler erst durch ein Redesign der Software aufgedeckt, da sich bereits getestete Funktionalitäten einer anderen und neuen Situation gegenüber sehen, die bisher noch nicht getestet wurde. Gerade im Falle des Redesigns soll die Automation von Softwaretests helfen, kostengünstig und schnell ein weiterhin lauffähiges Produkt zu liefern.

Dennoch darf nicht daran geglaubt werden, daß die Automation ein „Allheilmittel“ ist, welches keine manuellen Eingriffe mehr benötigt und sofort nach dem Einsatz bereits Kosten spart[8]. Bisher ist es immer noch so, daß zumindest die Testfälle durch die Entwickler oder einer speziellen Abteilung erstellt werden müssen. Ferner sind die Kosten für die Erstellung eines kompletten Testskriptes nicht zu unterschätzen. Auch wird ein automatisch ablaufendes System nie wirklich alle notwendigen bzw. geplanten Tests an einer Software vornehmen können.

Bevor ein Testwerkzeug eingesetzt wird, ist es zudem sinnvoll zu überprüfen, für welche Aufgaben dieses Werkzeug eingesetzt werden kann. Viele bisher erhältlichen Werkzeuge unterstützen lediglich den Test einer graphischen Oberfläche. Dies ist aber im Hinblick auf die Zurechenbarkeit von Fehlern erst dann sinnvoll, wenn die Basis des Programmes ebenfalls getestet wurde. Des Weiteren sollte sich das verwendete Tool in den bisherigen Softwareentwicklungsprozess integrieren lassen. Auf diese Punkte soll jedoch erst im späteren Verlauf der Arbeit näher eingegangen werden.

Bevor das Prinzip der Automation von Softwaretests, deren Einsatzgebiete und notwendigen Konsequenzen für den Softwareentwicklungsprozeß näher beleuchtet werden sollen, ist es zweckmäßig, noch einige wichtige Grundlagen zu besprechen. Daher wird im folgenden Abschnitt auf verschiedenen Techniken eingegangen, die bei einem manuell durchgeführten Softwaretest entscheidend sind, um daraus die nötigen Vorbereitungen für einen automatisierten Test abzuleiten.

## *2.1 Abgrenzung White-Box-, Black-Box-, und Grey-Box-Verfahren*

Es gibt verschiedene Arten von Softwaretests, welche sich nach dem Grad des Eingriffs in den zu testenden Code, bzw. nach dem Grad der Kenntnis von dem verwende

[8] vgl. Dustin, E., 2000, S. 26 f.

ten Algorithmus unterscheiden[9]. Diese Tests werden in White-Box, Grey-Box und Black-Box-Verfahren unterteilt.

### 2.1.1 Die White-Box-Verfahren

Als White-Box-Tests versteht man die bereits auf der Entwicklungsebene durchgeführten Tests. Beispielsweise kann der Programmierer an geeigneten Stellen Code für die Ausgabe von Variableninhalte einstreuen, um das Programmverhalten zu untersuchen. Ebenso fällt das absichtliche *Einfügen von Fehlern* in diese Kategorie, welches das Programmverhalten bei Fehlern wie „Datei ist schreibgeschützt“, „Platte ist voll“, usw. untersuchen soll.

Eine populäre Technik ist es auch, an zentralen Stellen Fehler per Zufall zu erzeugen, um die Reaktion des Programmes zu untersuchen. So wird z.B. in der Sprache „C“ die für das Reservieren von Speicher zuständige `malloc()`-Methode oft so verändert, daß diese in zehn Prozent aller Aufrufe meldet, es sei kein Speicher mehr frei. Mit einer solchen Technik ist es u.a. möglich zu verifizieren, ob vernünftige und auch für den Anwender verständliche Fehlermeldungen erzeugt werden.

Jeder dieser White-Box-Tests wird in die Programmausführung integriert und stellt damit naturgemäß einen automatisch durchgeführten Test dar. Das Einfügen solcher Testanweisungen wird als „Instrumentieren“ bezeichnet. Es ist möglich, die für einen bestimmten Test nötige Instrumentierung von einer Software übernehmen zu lassen[10]. Auf dieses Verfahren wird hier jedoch nicht eingegangen.

Wird die Instrumentierung des Codes von einem Tool übernommen, wird i.d.R. regelmäßig eine reine „Testversion“ des Programmes erzeugt, so daß der zur Auslieferung bestimmte Code davon unberührt bleibt. Wird dagegen die Instrumentierung manuell durchgeführt, muß sie oft im originalen Quellcode verbleiben, da es zu aufwendig wäre, sie für einen Release zu entfernen oder mit zwei Versionen, einer Testversion und einer Lieferversion, zu arbeiten. Aus diesem Grund bedient man sich eines boolschen Schal

[9] vgl. Dustin, E., 2000, S. 273 ff.

[10] vgl. Hasemann, M., 2001

ters oder, wie in der Sprache „C“ möglich, einer Compilerdirektive[11], um den Instrumentierungs-Code je nach Bedarf zu aktivieren oder zu deaktivieren.

### 2.1.2 Die Black-Box-Verfahren

Bei den Black-Box-Verfahren ist die zu testende Funktion eines Programms nur prinzipiell bekannt. Der Entwickler der Testfälle hat keinen Einblick und auch keine Eingriffsmöglichkeit in den Code. Er weiß nur, welche Eingaben zu welchen Ergebnissen führen sollen.

Somit ist das Entwickeln von Testfällen äußerst schwierig, und man kann sich nur auf Erfahrungen stützen. So finden bei mathematischen Funktionen als mögliche Eingaben häufig die Null, negative Zahlen und sehr große Zahlen Anwendung.

Im Gegensatz zu den White-Box-Tests entwickeln bei den Black-Box-Verfahren nicht die Programmierer die Testfälle. In vielen Softwarefirmen existiert hierfür eine spezielle Testabteilung, deren einzige Aufgabe es ist, Fehler in Programmen aufzudekken.

Dadurch ergeben sich Nachteile, die, wie bereits angemerkt, vor allem in der Unkenntnis über den verwendeten Algorithmus und die wirklichen Schwachstellen des Codes begründet sind. Schließlich weiß ausschließlich der Entwickler selbst, wo sein Code nicht sauber konstruiert ist und Angriffen durch ein Black-Box-Verfahren möglicherweise nicht standhalten würde. Er wird jedoch nie versuchen, diese Fehler wirklich aufzudecken, da er als Urheber des Codes bestrebt sein dürfte, dessen Fehlerfreiheit zu demonstrieren.

Alle Black-Box-Tests erfordern die explizite Ausführung des Programms. Grundsätzlich funktionieren die meisten Black-Box-Tests nach dem gleichen Schema: es wird eine Funktion des Programmes mit ausgewählten Parametern aufgerufen und das Ergebnis überprüft. Da der Black-Box-Test somit nicht prinzipiell automatisch abläuft, kann hier ein Tool entwickelt werden, welches die gesamte Prozedur automatisch durchführt und die Ergebnisse protokolliert.

---

[11] dazu gehören #define, #ifdef, etc. Diese Präprozessorbefehle erlauben in C/C++ eine bedingte Compilierung. In Java würde man sich eines boolschen Schalters bedienen müssen, der erst zur Laufzeit abgefragt wird.

Neben dem direkten Test von Funktionalitäten einer Software gehören auch Messungen der Leistung, der Belastung und der Last zu den Black-Box-Verfahren. Auch Verfahren zum Überprüfen der Netzwerklast oder Verfahren zur Prüfung der Datenkonsistenz bei Multithreading- oder Mehrbenutzersystemen zählen dazu. Auf diese soll aber in dieser Arbeit nicht weiter eingegangen werden.

### 2.1.3 Die Grey-Box-Verfahren

Der Begriff des Grey-Box-Testen hat sich durch das XP etabliert. Bei dem XP wird schon vor der Implementierung der eigentlichen Aufgabe mit der Entwicklung geeigneter Testfälle begonnen. Das hat den Vorteil, daß nicht blind drauflos programmiert wird, sondern sich die Entwickler erst einmal Gedanken über die Schnittstellen machen müssen.

Gegner des XP führen oft an, daß durch die fehlende Trennung zwischen Tester und Entwickler die Qualität der Testfälle stark angezweifelt werden muß. Diesem Urteil wird mit einem Grundprinzip des XP begegnet: es sollen immer zwei Entwickler an einem Problem arbeiten. Durch diesen als „pair-programming“ bezeichneten Stil beobachten und korrigieren sich die Entwickler gegenseitig[12].

Doch obwohl die Entwickler selbst die Testfälle für den eigenen noch zu implementierenden Code entwickeln und damit äußerst dicht an dem zu testenden Code sind, machen sie weiterhin von den Verfahren Gebrauch, die eigentlich für das Black-Box-Testen entwickelt wurden. Das heißt, es wird grundsätzlich nicht der eigentliche Code instrumentiert oder mit „Debugging“-Informationen angereichert, wie es bei einem White-Box-Verfahren üblich wäre, sondern es werden vielmehr die zu testenden Methoden mit kritischen Testparametern aufgerufen und die Ergebnisse überprüft.

Dabei haben die Grey-Box-Verfahren einen entscheidenden Vorteil gegenüber den Black-Box-Verfahren, denn hier kann der Programmierer einen Nutzen aus seiner Kenntnis über den verwendeten Algorithmus ziehen. Hat er z.B. für die Implementierung eines Algorithmus eine Methode zur Berechnung der Funktion $f(x) = 1/(x-5)$ schreiben müssen, so wird er für $x=5$ eine Abfrage eingebaut haben, um eine Division

---

[12] vgl. Westphal, F., Seite „XPueberdieSchultergeschaut.html“

durch Null zu vermeiden. Mit den Black-Box-Verfahren wäre es möglicherweise übersehen worden, die fünf als kritischen Wert zu testen. Bei den White-Box- wie auch bei den Grey-Box-Verfahren kann das nicht passieren.

Wie bereits bei den Black-Box-Verfahren angedeutet, bietet sich auch hier Potential, um den Entwicklern mit der Hilfe der Automation Arbeit abzunehmen und die Kosten gering zu halten.

## *2.2 Modultests und inkrementelle Tests*

Werden Programme sehr groß, wird die Entwicklung geeigneter Testfälle immer komplizierter. Es gibt daher einige Möglichkeiten, Testfälle statt für die gesamte Software nur für ausgewählte Bereiche zu entwickeln. Diese Bereiche können ein komplettes Unterprogramm, eine Unterroutine oder eine einzelne Prozedur bzw. Methode sein.

Die Zerlegung eines Programmes in seine Module hilft dabei, die kombinatorischen Probleme beim Testen in den Griff zu bekommen, es kann das parallele Testen mehrerer Module gestatten und es erleichtert die Fehlerbehebung, da ein aufgetretener Fehler einem bestimmten Modul zugeordnet werden kann.

Diese Verfahren sind unter dem Begriff „Modultest“ zusammengefaßt. Da die Module eines Systems getestet werden, ist es ebenfalls entscheidend, in welcher Reihenfolge die Module getestet werden. Wie noch gezeigt wird, sind die inkrementellen Verfahren i.d.R. die effektiveren Verfahren.

### 2.2.1 Modultests

Bei den modernen, objektorientierten Sprachen ist die Unterteilung des Softwareprojektes in kleine Module Teil der Planung. Hier werden mit Modellierungswerkzeugen wie UML einzelne Klassen und deren Abhängigkeiten beschrieben. Ferner wird bei der Planung eines großen Softwareprojektes eine Schichtenarchitektur angestrebt, die voneinander trennbare Probleme in verschiedene Unterprojekte aufteilt. Diese Faktoren sind bei der Vorbereitung eines Softwaretests sehr hilfreich.

Da in der objektorientierten Entwicklung ein Grundprinzip die Wiederverwendbarkeit des Codes ist, kann sich der Tester damit sein Aufgabe erleichtern. Schließlich braucht er den wiederverwendeten Code nur bei der erstmaligen Erstellung zu testen.

Bei der weiteren Verwendung einer Klasse durch Ableitung sind nur noch Tests an den von der Kindklasse überlagerten Methoden der Vaterklasse und neu hinzugekommen Methoden notwendig.

Dem Modultest kommt ferner der Umstand zugute, daß gerade bei großen Projekten oft die sog. 3-Tier-Architektur Anwendung findet, die sich in die Schichten „Presentationlayer“ (Darstellungsschicht), „Applicationlayer“ (auch Businesslayer oder „Anwendungsschicht“) und „Databaselayer“ (Datenbankschicht) unterteilt.

Jede dieser Schichten kann als einzelne Komponente bzw. Modul angesehen werden und kann somit auch einzeln getestet werden. Hilfreich ist dabei die Tatsache, daß diese Module nicht zwingend aufeinander aufbauen, sondern üblicherweise als austauschbare Komponenten designed werden.

Sollte z.B. die Datenbankschicht auf eine Oracle-Datenbank spezialisiert sein, so soll sie leicht gegen eine neue, auf DB2-Datenbanken spezialisierte Schicht ausgetauscht werden können, ohne daß das restliche Projekt dafür geändert oder angepaßt werden muß.

Das Austauschen der Implementierung einer der Schichten ist jedoch nur dann möglich, wenn eine Schnittstellendefinition für die entsprechende Schicht vorliegt, die auch eingehalten wird. Diese Schnittstelle kann verwendet werden, um einheitliche Tests an den beiden Implementierungsvarianten durchzuführen.

Bis auf ein paar spezielle Tests, die sich aus möglichen gravierenden Unterschieden zwischen den Implementierungen ergeben, ist damit eine Wiederverwendbarkeit von Testfällen möglich, was weitere Kosten spart.

Für das Testen der einzelnen Schichten werden oft nichtinkrementelle Testverfahren angewendet, wogegen die interne Klassenstruktur einer Schicht mit den inkrementellen Verfahren getestet wird.

### 2.2.2 Nichtinkrementelle und inkrementelle Tests

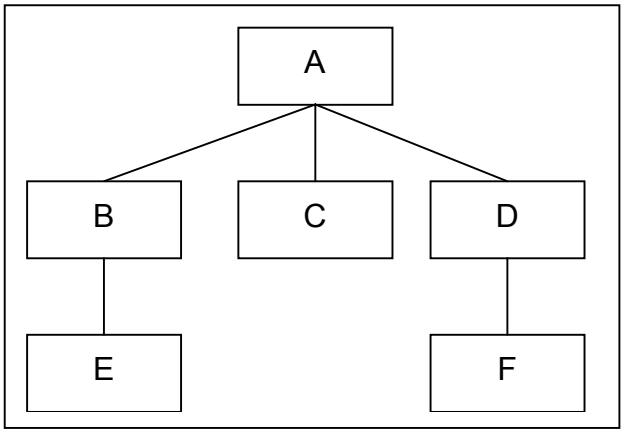
Ob bei einem Modultest ein nichtinkrementelles oder ein inkrementelles Testverfahren angewandt wird, entscheidet sich nach der späteren Reihenfolge der Zusammenführung der Module.

Bei den nichtinkrementellen Verfahren wird jedes Modul einzeln getestet, ohne die Implementierungsreihenfolge zu berücksichtigen. Bei den inkrementellen Verfahren wird die Reihenfolge der Implementierung auch als Testreihenfolge gewählt. Daraus ergeben sich für beide Verfahren sowohl Vor-, als auch Nachteile.

Die nichtinkrementellen Verfahren finden vor allem dann Anwendung, wenn mehrere Entwicklerteams mit der Implementierung einzelner Module beauftragt werden, denn hier ist es oft durch Zeitverschiebungen nicht möglich, die Module in der Reihenfolge zu entwickeln, in der sie hinterher auch zusammenarbeiten.

Betrachten wir hierzu das Beispiel in der folgenden Abbildung. Wird das Modul B von einer anderen Abteilung entwickelt, als das Modul E, so wird neben der Entwicklung selbst auch das Testen des Moduls B durch die Abhängigkeit von Modul E problematisch, wenn für den Test das Modul E noch nicht fertiggestellt ist.

**Abbildung 1: Programmbeispiel mit sechs Modulen**




(Quelle: eigene Darstellung in Anlehnung an Myers, G.J., 2001, S.89)

Dies stellt die Vorgehensweise bei den nichtinkrementellen Testverfahren dar, bei welchem die Module einzeln getestet werden, ohne die in Abhängigkeit stehenden Module mit zu verwenden. Der Zweck bei einem solchen Vorgehen ist die eindeutige Zurechenbarkeit der entdeckten Fehler auf das gerade getestete Modul.

Um nun ein normalerweise abhängiges Modul ganz für sich zu testen, müssen die Funktionen der abhängigen Module simuliert werden. Für den Fall, daß z.B. das Modul B getestet werden soll, ist es notwendig, die Funktionalität des Moduls E zu simulieren,

indem man eine eigene Implementierung schreibt. Ein solches nur für Testzwecke entwickeltes Simulationsmodul nennt man „STUB“. Um den Aufruf von Testfällen an das Modul B abzuwickeln, bedient man sich eines „Treibermoduls“, welches die Parameter an B übergibt und das Ergebnis zurück liefert[13].

Der nichtinkrementelle Modultest kann dadurch sehr aufwendig werden. Für das oben angeführte Beispiel wären für einen kompletten Test aller Module nach dem nichtinkrementellen Verfahren elf zusätzliche Module zu entwickeln, die aus drei STUBs für das Modul A und jeweils einen STUB für die Module B und D, plus fünf Treibermodulen bestehen würden, wenn davon ausgegangen wird, daß für das Modul A kein Treiber entwickelt wird.

Den nichtinkrementellen Verfahren ist es somit als negativ auszulegen, daß sie durch die zusätzlich notwendigen Implementierungen von STUBs und Treibermodulen nicht nur sehr zeit- und damit kostenintensiv sind, sondern diese neuen Module wieder fehlerhaft sein können.

Damit wäre nicht klar, ob zum einen alle Fehler des getesteten Moduls aufgedeckt werden, und ob auf der anderen Seite ein eventuell gefundener Fehler nicht auf eine Fehlfunktion des STUBs zurückzuführen ist. Damit ist es grundsätzlich notwendig, daß diese extra für den Test implementierten Module ebenfalls überprüft werden müssen.

Diese Tests wiederum würden jedoch mindestens die Implementierung von Treibermodulen für die Aufrufe der zu testenden STUBs erfordern, welche erneut Fehler enthalten könnten, somit wieder zu testen wären, und so weiter. Dieser Rekursion und dem damit verbundenen Aufwand kann man nur Herr werden, indem man die Tests der Treibermodule und STUBs schlicht wegläßt.

Da die Treibermodule normalerweise nicht sonderlich komplex sein werden, mag es genügen, sie nur einer Codeinspektion oder einem Schreibtischtest[14] zu unterziehen. Es bleibt jedoch das Problem, daß auch auf diesem Weg Fehlerfreiheit nicht garantiert werden kann. Das Gleiche gilt für die STUBs.

Ein gänzlich anderes Vorgehen verfolgen die inkrementellen Tests. Diese Verfahren werden angewandt, wenn bereits die Codierung der einzelnen Module in der Reihenfol

---

[13] s. Myers, G.J., 2001, S. 89

[14] s. Myers, G.J., 2001, S. 16 ff.

ge abgewickelt wird, in der sie implementiert werden sollen. Somit werden auch die Tests an den jeweilig fertiggestellten Modulen in genau dieser Reihenfolge durchgeführt.

Es ergeben sich nun zwei Möglichkeiten, in welcher Reihenfolge implementiert wird. Entweder von oben nach unten (Topdown) oder von unten nach oben (Bottomup). Dem o.a. Beispiel folgend würde demnach entweder erst das Modul A entwickelt, oder erst die Module E, C und F.

Wird nun nach der Fertigstellung eines Moduls der Test durchgeführt, so sind bereits in Abhängigkeit stehende Module fertig entwickelt und ebenfalls getestet worden. Daraus ergibt sich ein Vorteil der inkrementellen Methoden im Gegensatz zu den nichtinkrementellen Verfahren.

Bei dem Topdown-Verfahren wurden die übergeordneten Module bereits implementiert, so daß die Entwicklung eines Treibermoduls nicht notwendig ist. Es müssen nur noch die STUBs entwickelt werden, um die tiefer liegenden Module zu simulieren.

Bei den Bottomup-Verfahren verhält es sich genau umgekehrt. Hier müssen nur die Treibermodule als Ersatz für höher liegende Module entwickelt werden, wogegen die STUBs wegen den bereits vorhandenen Modulen nicht mehr entwickelt werden müssen.

Für das obige Beispiel ergibt sich daraus, daß für einen Topdown-Ansatz genau fünf STUBs entwickelt werden müssen. Für einen Bottomup-Ansatz sind es fünf Treibermodule, wenn man davon ausgeht, daß für das Modul A kein Treiber mehr implementiert werden muß.

Die aus dem wesentlich geringerem Aufwand resultierende Zeit- und Kostenersparnis ist offensichtlich. Daher sollte den inkrementellen Methoden grundsätzlich der Vorzug gegeben werden.

#### *2.2.2.1 Die Topdown-Methode*

Bei den Topdown-Methoden werden die einzelnen Module eines Projektes von oben nach unten getestet. Dabei müssen die Funktionalitäten von aufgerufenen Modulen teilweise oder gänzlich simuliert werden.

Die Implementierung der dazu notwendigen STUBs ist jedoch keineswegs trivial. Schließlich muß das Verhalten der simulierten Module möglichst exakt nachempfunden

werden. Beispielsweise sollte die Simulation bei falschen Übergabewerten auch so reagieren, wie es das originale Modul getan hätte. Nur so kann das getestete Modul „lebensecht“ reagieren. Wäre die korrekte Reaktion z.B. die Rückgabe eines Null-Wertes, so muß dies auch der STUB zurückgeben.

Um sich das Leben etwas zu erleichtern, werden die STUBs normalerweise so programmiert, daß sie statische Werte zurück liefern. Damit sind sie aber nur für einen bestimmten Test oder eine begrenzte Anzahl Tests geeignet. Das führt zu der Konsequenz, daß für viele unterschiedliche Tests ein neuer STUB erarbeitet werden muß, der entsprechend des Tests reagiert. Soll z.B. die Fehlertoleranz eines Moduls getestet werden, so muß der STUB auch Fehler provozieren.

#### *2.2.2.2 Die Bottomup-Methode*

Es ist äußerst zweckmäßig, die in der Hierarchie an unterster Stelle stehenden Funktionen einer Software, wie z.B. Routinen zum lesen einer Textdatei, als erstes zu testeten. Dies als Bottomup-Ansatz bezeichnete Verfahren erleichtert die Fehlersuche, da man sich beim Testen höher gelegener Stufen auf die Funktionalität bereits getesteter Serviceroutinen verlassen kann und mögliche Fehler nur in den neu implementierten Methoden suchen muß.

Dieses Vorgehen setzt selbstverständlich voraus, daß die für die Servicemethoden ausgesuchten Testfälle ausreichend waren und somit ausgeschlossen werden kann, daß sie noch Fehler enthalten. Andernfalls schlägt dieses Prinzip genau in dem Augenblick fehl, wenn die neu implementierten Funktionen die Serviceroutinen mit Parametern verwenden, die vorher nicht überprüft wurden, jedoch Fehler auslösen.

Ein weiterer Vorteil der Bottomup-Methode besteht darin, daß hier statt der STUBs für die Topdown-Methode nur Treibermodule implementiert werden müssen. Diese Treibermodule sind i.d.R. wesentlich einfacher zu entwickeln, da ihre einzige Aufgabe darin besteht, die zu testenden Methoden eines Moduls bzw. einer Klasse mit entsprechenden Parametern aufzurufen. Zudem kann ein Treibermodul mehrere Testfälle abdecken, was bei den STUBs nicht immer möglich ist.

Bei der Bottomup-Methode müssen jedoch immer dann auch STUBs entwickelt werden, wenn bei der Ausführung des zu testenden Moduls nicht vorhersehbare Daten

benötigt werden. Einer solcher Umstand würde die Überprüfung des Resultates auf Korrektheit schwer oder sogar unmöglich machen. Die Problematik wird im Zusammenhang mit der „Definition der Testumgebung“ angesprochen werden. Da sie bei der Automation eine besondere Rolle spielt, wird dort noch einmal explizit darauf eingegangen.

## *2.3 Anwendungsgebiete und Voraussetzungen für automatisiertes Testen*

Die von Managern oder Projektleitern oft gemachte Assoziation „automatisch“ gleich „Rationalisierung“ gleich „Kostenersparnis“ ist im Falle der Automation von Softwaretests ein Trugschluß. Oft ist es sogar in der Realität üblich, daß direkt nach der Einführung eines automatisierten Testsystems die Kosten der Entwicklung zuerst ansteigen, statt zu sinken[15].

Dieser Kostenanstieg liegt sowohl in der Anpassung des bestehenden Softwareentwicklungsprozeß an das neue Tool, als auch in der Umstellung und möglicherweise notwendigen Schulung der Mitarbeiter begründet. Trotzdem wird oft von Projektleitern die Umstellung auf ein automatisiertes Testsystem gefordert, da auch Softwarehersteller von Personaleinsparungen und anderen Rationalisierungsmaßnahmen betroffen sind, jedoch im Gegenzug immer schneller und billiger ein neues Softwareprodukt entwikkeln müssen. Gerne wird daher versucht, mit automatisierten Testmethodiken Zeit und Geld zu sparen.

Die Stärken eines automatisierten Testverfahrens zeigen sich jedoch nie oder selten bei der Erstimplementierung von Funktionalitäten, sondern vor allem bei Erweiterungen bereits getesteter Funktionen oder bei einem Redesign der kompletten Software. In diesen Fällen wird ein bereits bestehendes Testfallarchiv auf Knopfdruck abgearbeitet und kann zeigen, ob die programmierten Änderungen Auswirkungen auf bereits getestete Funktionalitäten haben.

Bei der Einführung eines automatisierten Testverfahrens sind einige Punkte zu berücksichtigen, die auch für den zu erwartenden Kostenanstieg verantwortlich sind.

---

[15] s. Dustin, E., 2000, S. 26 ff.

### 2.3.1 Die verschiedenen Arten von Testwerkzeugen

Es gibt eine Vielzahl automatisierter Testtools auf dem Markt, was eine Kaufentscheidung notwendig macht. Um sich überhaupt für ein Produkt entscheiden zu können, müssen alle in Frage kommenden Produkte erst auf ihre Stärken und ihre Einsatzmöglichkeit hin geprüft werden. Die Ansätze von Testtools sind sehr unterschiedlich. Grob lassen sie sich in zwei Kategorien unterteilen, nämlich in die Klasse der White-Box-Teststools und der Black-Box-Testtools.

Unter die Werkzeuge für White-Box-Tests fallen beispielsweise Tools zur Codeinspektion, die Debugger, welche heutzutage normalerweise bereits in der Entwicklungsumgebung integriert sind, Tools zur statischen und dynamischen Analyse und die Werkzeuge zur Überdeckungsmessung.

Zu den Black-Box-Tests zählende Tools sind beispielsweise Werkzeuge zum Aufspüren von Speicherproblemen und Laufzeitfehlern und Testwerkzeuge zum Testen graphische Benutzeroberflächen und zur Bestimmung von Last, Leistung und Belastung. Dabei sind insbesondere die GUI-Testwerkzeuge auch in der Lage, funktionale Fehler zu finden.

Die Aufgabe eines GUI-Tests ist es, die Funktionalität einer graphische Oberfläche eines Programmes zu überprüfen. Einige dieser Tools sind dabei nicht von der verwendeten Programmiersprache abhängig. Sie steuern die Maus und die Tastatur über die Betriebssystemebene fern, um Aktionen eines Benutzers zu simulieren.

Solche Tools sind oft nicht sehr hilfreich, da sie zwar sehr wohl Tests durchführen und Fehler entdecken, jedoch die Ortung eines Fehlers nicht erleichtern. Dazu kommt, daß i.d.R. nur die Tests durchgeführt werden, die sich aus der Funktionsbeschreibung der Oberfläche ableiten lassen, und nicht aus den verwendeten Algorithmen oder Aufgaben einzelner Methoden oder Module.

Ebenso nachteilig ist die Erstellung des für die simulierte Bedienung der zu testenden Oberfläche notwendigen Skriptes. Meistens finden hier sog. Aufzeichnungstools Anwendung, welche eine zu testende exemplarische Bedienung der Oberfläche aufnehmen und dann im Rahmen eines Tests wieder abspielen.

Auf die gesamte Palette an verfügbaren Testwerkzeugen soll an dieser Stelle nicht weiter eingegangen werden. Im praktischen Einsatz wird immer eine Kombination aus

verschiedenen Werkzeugen sinnvoll sein. So kann beispielsweise die Überdeckungsmessung mit einem funktionalen Test der Software kombiniert werden, um eine Aussage über die Qualität der verwendeten Testfälle zu erhalten. Wurde beispielsweise nur eine geringe Überdeckung erreicht, so ist zu vermuten, daß die Testfälle nicht ausreichend waren, um die gesamte Funktionalität zu überprüfen.

### 2.3.2 Voraussetzungen für den Einsatz eines Testwerkzeuges

Das in dieser Arbeit entwickelte Werkzeug wird einen Funktionstest der Software ermöglichen. Dabei wird es nicht die GUI als Schnittstelle zwischen Testtool und Testkandidat wählen, sondern direkt Methoden aus Klassen aufrufen und Ergebnisse vergleichen. Es handelt sich somit um ein in die Klasse der Black-Box-Tools einzuordnendes Werkzeug.

Soll ein solches Tool zum Einsatz kommen, sind zum einen vom Entwickler verschiedene Voraussetzungen zu erfüllen, damit eine Methode oder eine Klasse erfolgreich getestet werden kann, auf der anderen Seite muß sich ein solches Tool in den Softwareentwicklungsprozeß eingliedern. Zusätzlich ergeben sich durch die heutigen objektorientierten Ansätze weitere Besonderheiten.

#### *2.3.2.1 Die Implementierungsreihenfolge*

Im Rahmen des Softwareentwicklungsprozeß muß entschieden werden, welche Module bzw. Klassen in welcher Reihenfolge von welchem Team implementiert werden. Dabei ist es im Hinblick auf die durchzuführenden Tests wesentlich günstiger, inkrementelle Methoden zu verwenden, da sowohl bei den Topdown-, als auch bei den Bottomup-Verfahren weniger Aufwand durch zusätzlich zu implementierende Treibermodule bzw. STUBs entsteht. Im Gegensatz dazu müssen bei den nichtinkrementellen Verfahren immer beide Module implementiert werden.

Gerade die Bottomup-Verfahren haben zusätzlich den entscheidenden Vorteil, daß im Rahmen eines automatisierten Testverfahrens bei der erneuten Verwendung einer Basisklasse in einem anderen Projekt bereits Testfälle dafür entwickelt wurden.

Existiert z.B. ein sehr umfangreiches Datenbank-Framework, welches in zahlreichen Projekten zum Einsatz kommt, so braucht trotzdem nur einmal ein komplettes Testfal

larchiv dafür erzeugt zu werden. Bei folgenden Veränderungen oder Erweiterungen dieses Frameworks kann das Archiv zum einen dazu dienen, die weitere Funktionsfähigkeit zu zeigen, und auf der anderen Seite helfen, auftretende Fehler in einem bisher lauffähigen Projekt auf Änderungen im Framework zurückzuführen[16].

Diese Vorteile zeigen, daß das Bottomup-Verfahren im Hinblick auf durchzuführende Funktionstests die bessere Wahl zu sein scheint. Daher wird im weiteren Verlauf der Arbeit auch immer so argumentiert werden, als wäre dieses Verfahren angewendet worden.

#### *2.3.2.2 Besonderheiten der Objektorientierung*

In den prozeduralen Sprachen stellen Funktionstests an einer Funktion oder Prozedur in aller Regel keine besondere Schwierigkeit dar. Der Aufruf einer Prozedur oder Funktion allein genügt. Das zu überprüfende Ergebnis wird entweder zurückgeliefert oder aber ist in anderer Form aus Variablen direkt ablesbar. Bei den objektorientierten Sprachen ergeben sich aus den damit eingeführten Techniken einige Besonderheiten, die hier besprochen werden sollen.

Durch das Kapselungsprinzip sind sowohl Klassen- und Instanzvariablen, als auch Methoden oft nicht von außen erreichbar und können somit auch nicht überprüft bzw. aufgerufen werden. Ist jedoch das zu überprüfende Ergebnis gerade in einer solchen Variablen untergebracht oder es ist das Ergebnis einer privaten Methode, muß entweder der Programmierer im Vorfeld eine Möglichkeit geschaffen haben, um diese Inhalte auszulesen, oder aber das verwendete Testtool ist in der Lage, die Sicherheitsbarriere zu durchbrechen, um gegen jede Konvention die Inhalte selbst zu ermitteln[17].

Bei den Funktionstests ist weiterhin zu beachten, daß durch die Verwendung von Polymorphismus die Möglichkeit besteht, daß ein Methodenname mehrfach in einer Klasse vorkommt und sich nur durch die verwendeten Parameter unterscheidet. Ein ähnliches Problem besteht auch durch das Prinzip der Ableitung. Es ist durchaus möglich, daß der selbe Methodenname sowohl in der Vaterklasse, als auch in der Kindklasse existiert.

---

[16] s.a. „Smoke-Test“, Dustin, E., 2000, S. 396 f.

[17] was in Java durch das Reflection-API durchaus möglich ist.

Dadurch ist es einerseits denkbar, bereits vorhandene und möglicherweise bereits getestete Funktionalität mitzubenutzen, jedoch wird bei der Vererbung oft bereits vorhandene Funktionalität durch neue ersetzt. Dies hat Konsequenzen für einen erfolgreichen Funktionstest einer Methode oder ganzen Klasse.

Es muß darauf geachtet werden, daß der Test einer Methode auch ihre gesamte Funktionalität abdeckt. Das folgende Beispiel zeigt, wie Polymorphismus nicht aussehen sollte. Die beiden Methoden „`getElementString()`“, die sich nur durch ihre Parameter unterscheiden, sollen sich in der gleichen Klasse befinden.

**Abbildung 2: Beispiel von schlechtem Polymorphismus**

```
public String getElementString(int Element)
{
    if (Element==0)
        return "Null";
    else
        return Integer.toString(Element);
}

public String getElementString(int [] Element)
{
    String result = "";
    for (int i=0; i<Element.length; i++)
    {
        if (i!=0) result+=", ";
        //statt:
        // result += getElementString(Element[i]);
        if (Element[i]==0)
            result += "Null";
        else
            result += Integer.toString(Element[i]);
    }
}
```

(Quelle: eigene Darstellung)

Für die durchzuführenden Tests macht es keinen Unterschied, ob in einer Methode redundanter Code vorhanden ist oder nicht. Grundsätzlich wird jedoch sowohl das Testen selbst, sowie auch die spätere Korrektur fehlerhaften Codes vereinfacht, wenn Redundanz vermieden wird. Was in diesem Beispiel durch den Spezialfall des Polymorphismus angedeutet wird, trifft auch für andere Methodenaufrufe zu. Grundsätzlich gilt es als Prinzip der Objektorientierung, das Rad nie neu zu erfinden, sondern immer auf vorhandenen und damit möglicherweise auch bereits getesteten Strategien und Al

gorithmen aufzubauen. In dem Fall des Polymorphismus kann es jedoch als Problem angesehen werden, daß durch die gleichen Methodennamen Verwirrung über die zu erstellenden Testfälle besteht. Schließlich unterscheiden sich solche Methoden grundsätzlich nur durch ihre Parameterliste, was die Erstellung korrekter Testfälle erschweren kann.

Etwas anders sieht es aus, wenn eine Methode durch Ableitung überlagert und möglicherweise sogar von der neuen Methode in der Kindklasse aufgerufen wird, um die bereits vorhandene Funktionalität weiter zu nutzen. In der Regel sollte die Vaterklasse bereits vollständig getestet worden sein, setzt man ein Bottomup-Verfahren voraus.

Arbeiten jedoch mehrere Entwickler an einer Klasse und konsolidieren ihre jeweiligen Implementierungen, kann es bei einer unsauberen Durchführung vorkommen, daß die Implementierungen eines Entwicklers komplett gelöscht werden. Sollte dabei eine ganze Methode verschwinden, muß dennoch nicht zwingend ein Compilerfehler auftreten. Überlagerte die verschwundene Methode nämlich eine Methode der Vaterklasse, so wird diese nun einspringen, was mit Sicherheit zu Fehlern führen wird. Daraus folgt, daß in objektorientierten Systemen wesentlich genauer getestet werden muß, also sollte auch nach Komplettierung eines Systems erneut ein vollständiger Test durchgeführt werden, um hierdurch entstandene Fehler aufzuspüren, selbst wenn die im Vorfeld durchgeführten Tests alle negativ waren.

Die Möglichkeit der Ableitung sorgt ferner dafür, daß keine große Wahl bei der Implementierungsreihenfolge besteht. Schließlich müssen die Vaterklassen zu erst entwikkelt werden, um danach erfolgreich die Kindklassen erzeugen zu können. Dem entsprechend sollte auch getestet werden.

Sowohl bei der Menge der objektorientierten, wie auch bei der Menge der prozeduralen Sprachen sind Abhängigkeiten vorhanden, die das Vorgehen bei der Implementierung und bei den Testläufen grundsätzlich vorschreiben, doch sind bei den objektorientierten Sprachen durch das Prinzip der Ableitung wesentlich bessere und auch häufiger genutzte Möglichkeiten geschaffen worden. Daher ist gerade bei den objektorientierten Sprachen das Testen nach dem Bottomup-Verfahren sinnvoll, da meistens auch nach diesem Verfahren entwickelt werden dürfte.

### 2.3.3 Die Definition der Testumgebung

Wie oben angeführt, wird nur die Automation von Funktionstests im Bereich der Black-Box- sowie Grey-Box-Verfahren angestrebt. Um erfolgreich ein Prinzip zur Automation von Tests aus diesem Bereich vorzuschlagen, ist es notwendig, in allen Testverfahren eine gemeinsame Basis zu finden, welche sich als Standardimplementierung anbietet. Es ist daher hilfreich, daß sich, trotz der Vielzahl an möglichen Tests, das Vorgehen bei einem Black-Box- oder Grey-Box-Test auf eine bestimmte Tätigkeit reduzieren läßt.

Es wird immer die Aufgabe des Testers sein, zu einem bestimmten Input das zu erwartende Ergebnis eines Testlaufs mit dem tatsächlich aufgetretenen zu vergleichen. Hierfür ist es unumgänglich, den zu erwartenden Output vorher zu definieren, um einen echten Vergleich durchführen zu können. Oft resultieren in einer Software verbliebene Fehler auf der Mißachtung dieser Regel.[18]

Doch gerade unter der Prämisse, die Tests automatisch ablaufen zu lassen, muß einem solchen Programm vorher mitgeteilt werden, welcher Output als korrekt angesehen werden soll, und was als Fehler gewertet werden muß. Um die Vergleichbarkeit des Ergebnisses zu gewährleisten, muß sicher gestellt sein, daß das erwartete Ergebnis auftreten wird, wenn der Code korrekt arbeitet, bzw. es *nur dann nicht* auftreten wird, wenn der Code fehlerhaft ist.

Dies führt zu einer wichtigen Erkenntnis im Zusammenhang mit automatisierten Softwaretests. Die Tests müssen in einer wohl definierten Umgebung ablaufen, so daß die zu prüfenden Ergebnisse vorhersehbar sind. Diese Umgebung setzt sich aus dem Input und dem Output des Tests zusammen; es muß klar festgelegt sein, welche Eingaben zu welchen Ausgaben führen werden bzw. sollen.

Für eine Methode, welche z.B. einen Funktionswert ausrechnen soll, würde die Definition der Testumgebung aus den Übergabeparametern und dem zugehörigen Ergebnis bestehen. Dies stellt keine besonders schwere Aufgabe dar.

Ungleich komplizierter wird es dagegen, wenn die zu testende Funktion zeitabhängige Daten oder Zufallswerte heranzieht, die aus Serviceroutinen stammen, oder aber al

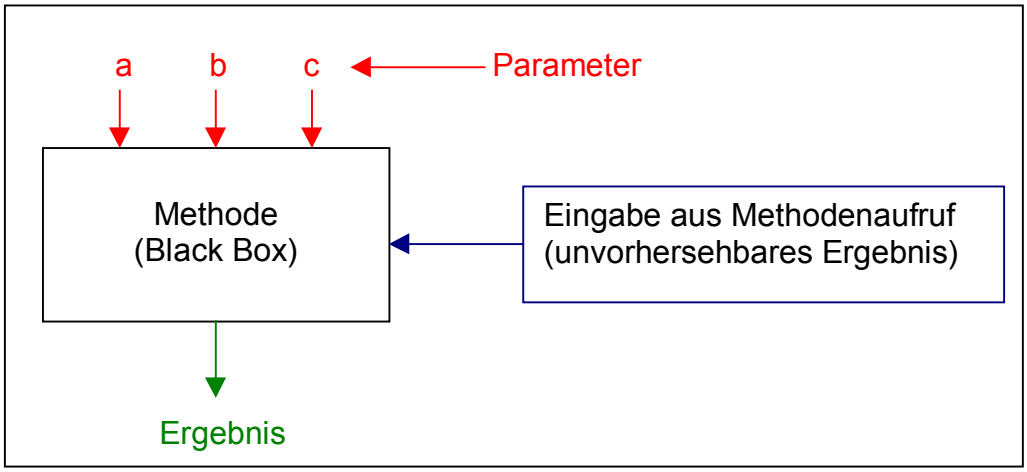
---

[18] vgl. Myers, G.J., 2001, S. 11

lein auf diesem Weg ihre Parameter erhält, statt diese übergeben zu bekommen. Hat die zu testende Funktion z.B. die Aufgabe, die aktuelle Uhrzeit zu ermitteln und in einer Zeichenkette zurück zu liefern, so ist zum Zeitpunkt des Testlaufs diese Zeichenkette nicht festgelegt, da normalerweise die Zeit angehalten werden müßte.

Um auch bei solchen nicht vorhersehbaren Werten erfolgreich testen zu können, ist es daher notwendig, für die zu testende Software einen „Testmodus“ zu implementieren, so daß entsprechende Funktionen nicht die echten und zufälligen Werte liefern, sondern vorher festgelegte Werte.

**Abbildung 3: Beispiel einer Parameterherkunft durch Methodenaufruf**




(Quelle: eigene Darstellung)

In dem obigen Beispiel würde das bedeuten, daß der Entwickler bei der Implementierung der Funktion die Möglichkeit schaffen muß, daß bei einem Test nicht die echte aktuelle Uhrzeit verwendet wird, sondern eine vom Test vorgeschriebene Zeit. Nur so ist die Prämisse der wohldefinierten Umgebung aufrecht zu halten.

Bei diesem Problem zeigt sich eine Schwäche der Automation, die aber gleichzeitig auch als eine Stärke interpretiert werden kann. Während ein Mensch bei einem manuell durchgeführten Test die zurückgelieferte Zeichenkette schlicht betrachtet und damit verifiziert, ist ein vom Computer durchgeführter automatischer Test auf ein Muster angewiesen, das er für den Vergleich heranziehen kann.

Bei dem angeführten Beispiel ist die Uhrzeit gar nicht relevant, da sie wahrscheinlich von einer Systemfunktion ermittelt wird, die in aller Regel auch funktioniert. In diesem Fall ist es wesentlich wichtiger zu überprüfen, ob der Ergebnisstring auch korrekt nach den Anforderungen formatiert ist, also z.B. der Form „HH:MM:SS“ genügt und auch eine korrekte Uhrzeit darstellt.

Diese für einen Menschen mit einem Blick durchgeführte Form der Ergebnisvalidierung wäre jedoch schlecht zu automatisieren. Schließlich sollen die Kriterien für ein korrektes Ergebnis leicht und in allgemeingültiger Form formuliert werden können. Es ist daher bei der Testautomation wesentlich einfacher, die Software schlichte Vergleiche zwischen Ergebnis und Muster durchführen zu lassen, zur Not auch Byte für Byte.

Da der Entwickler in den oben angeführten Fällen sein Programm mit zusätzlichem Code ausstatten muß, wenn er die automatische Testdurchführung ermöglichen will, kann dieser Umstand als Schwäche der Automation ausgelegt werden. Schließlich werden so höhere Kosten bei der Entwicklung verursacht.

Auf der anderen Seite ist bekannt, daß bei nicht definierten Ergebnissen und somit manueller Verifizierung, schnell Fehler übersehen werden. Dies ist in der Psychologie des Menschen begründet, der bestrebt sein wird, keine Fehler zu entdecken[19]. In solchen Fällen ist die unbestechliche Aussage eines automatisch durchgelaufenen Tools wesentlich mehr wert.

Unter der Berücksichtigung der Definition von Testen ist es somit eine Stärke der Automation von Softwaretests, Fehler auch als solche zu erkennen. Ein Programm zur automatischen Durchführung von Tests wird die ihm aufgetragen Tests durchführen und die Resultate unbeirrt vergleichen. Die Entwicklung geeigneter Testfälle, also solche, die auch wirklich geeignet sind, Fehler aufzudecken, kann dagegen nicht von einem Tool durchgeführt werden.

Um nun auch Methoden testen zu können, welche unvorhersehbare Eingabewerte verwenden, wodurch auch das Ergebnis unvorhersehbar wird, ist es die Aufgabe des Entwicklers, entsprechende Vorkehrungen zu treffen. Für dieses Problem gibt es jedoch keine grundsätzliche und einheitliche Lösung. In der Regel handelt es sich hierbei um Methodenaufrufe, die im Testfall auf eine andere Methode mit festem Rückgabewert

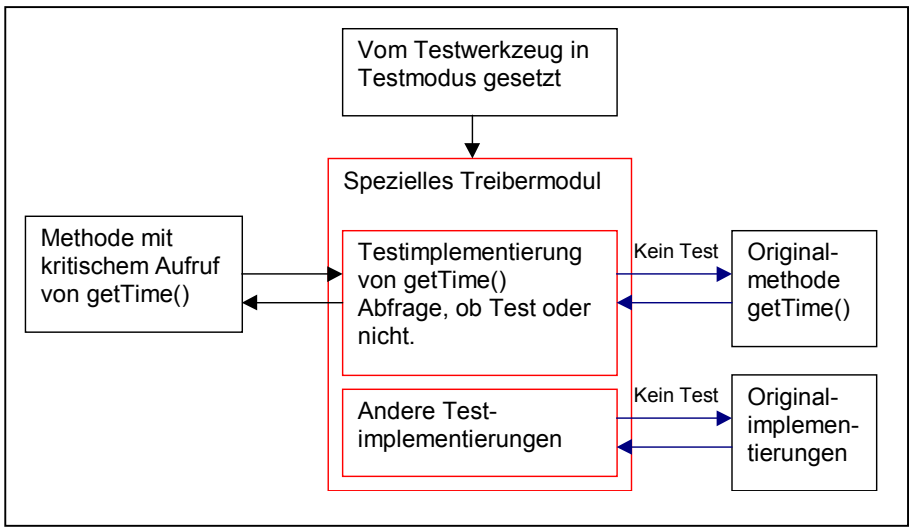
---

[19] s. Myers, G.J., 2001, S.11

umgeleitet werden müssen. Bei der Entwicklung einer Technik für diese Umleitung sind drei Dinge wünschenswert. Die Umleitung sollte möglichst automatisch erfolgen, wenige oder im besten Fall gar keine Änderungen am Code zur Folge haben und möglichst flexibel sein, so daß die festen Rückgabewerte für jeden Testlauf wechseln können, wie es bei den Parametern einer Methode auch möglich ist.

Es ist möglich, alle kritischen Methodenaufrufe in eine nur für diesen Zweck existierende Klasse umzuleiten, welche für jede dieser Methoden eine gesonderte Implementierung bereit hält. Dieses Modul kann dann global bei dem Start des Tests in den Testmodus geschaltet werden, so daß die Aufrufe an die kritischen Methoden nicht durchgereicht, sondern entsprechend feste Rückgabewerte geliefert werden. Das folgende Schaubild soll diesen Gedanken verdeutlichen.

**Abbildung 4: Struktur des Treibermoduls für kritische Methodenaufrufe**




(Quelle: eigene Darstellung)

Die ursprüngliche beispielhafte Codezeile `myDate=myCalendar.getTime();`, wobei `myCalendar` eine Instanz der Klasse `java.util.GregorianCalendar` sei, müßte somit ersetzt werden in `myDate=KritischeMethoden.getTime(myCalendar);`. Wie die Implementierung dieser neuen `getTime(..)`-Methode aussehen muß, ist in der nächsten Abbildung dargestellt.

Diese Form der Umleitung ist selbstverständlich nicht die optimale Lösung. Besser wäre es, wenn die Umleitung der kritischen Methoden automatisch durch das Testwerk

zeug realisiert würde. Dazu müßte allerdings dafür gesorgt werden, daß die JVM kritische Methodenaufrufe erkennt und bei Bedarf selbständig umleitet. Eine derartige Technik ist derzeit in Java jedoch nicht möglich.

**Abbildung 5: Beispiel einer Umleitungsmethode für getTime()**

```
public class KritischeMethoden
{
    public static boolean isTest = false;

    public static java.util.Date getTime(java.util.GregorianCalendar instanz)
    {
        if (!isTest)
            return instanz.getTime();
        else
            // Ist Testmodus aktiv, immer den 01.05.2001 zurückliefern!
            return (new java.util.GregorianCalendar(2001,05,01)).getTime();
    }
}
```

(Quelle: eigene Darstellung)

Die Schwäche dieser Technik ist zum Einen in der schwierigen und keinesfalls automatisierten Anpassung des Codes zu sehen, zum Anderen müssen die neuen Methodenaufrufe zur Kapselung kritischer Methodenaufrufe vor der Auslieferung der Software entweder umständlich entfernt werden, oder aber die Klasse `KritischeMethoden` wird mit in das Produkt integriert. Beide Möglichkeiten stellen keine echte Alternative dar.

Die für die Kapselung notwendigen Methoden sind nicht komplex. Sie leiten entweder ohne jegliche Überprüfung weiter oder geben einen festen Wert zurück, der auf irgendeine Art zu erzeugen ist. Daher ist es vorstellbar, daß vom Entwickler nur beschrieben wird, welche Methoden umgeleitet werden sollen, und was im Testfall zurückzugeben ist. Bevor eine Methode oder ein ganzes Projekt schließlich dem Test unterzogen wird, kann vorher ein Parser über den Quellcode schauen und alle umzuleitenden Methodenaufrufe durch den neuen Aufruf ersetzen. Der durch die Veränderung entstandene Code wird dann dem Test unterzogen.

Dieses Verfahren hat jedoch erneut entscheidende Nachteile. Werden Fehler gefunden, so sind diese im Originalcode zu beheben, was eine erneute Erzeugung des testfähigen Codes notwendig macht. Ferner ist die Umstellung des Codes keineswegs trivial, da die zu ersetzenden Methodenaufrufe durch einen Automatismus nicht unbedingt

leicht zu identifizieren sind. Schließlich kann dieser nicht mit einer simplen „suchen und ersetzen“-Strategie über den Namen der Methode arbeiten. Es muß sichergestellt sein, daß der Methodenaufruf an eine Instanz der richtigen Klasse gerichtet wird, und ob die Methode auch die korrekte Parameterliste verwendet. Andernfalls würde bei der Verwendung von Polymorphismus die falschen Methodenaufrufe umgeleitet. Ferner sollte es möglich sein, auch statische Methodenaufrufe umzuleiten.

Die weitere Besprechung dieses Parsers würde den Rahmen dieser Arbeit sprengen, daher soll darauf nicht weiter eingegangen werden. Ungeachtet der umständlichen Anwendung scheint es dennoch ratsam, über ihren Einsatz noch genauer nachzudenken. Trotzdem wird im weiteren Verlauf der Arbeit davon ausgegangen, daß die kritischen Methodenaufrufe manuell ersetzt werden.

## *2.4 Grenzen der Automation von Softwaretests*

Die bisher gezeigten Einsatzmöglichkeiten von automatisierten Softwaretests dürfen nicht darüber hinweg täuschen, daß solche Werkzeuge keine Patentlösung darstellen, deren Einsatz automatisch ein fehlerfreies Programm garantieren. Vielmehr haben alle Werkzeuge feste Grenzen, in denen sie nur funktionieren.
Gerade das obige Kapitel hat deutlich gemacht, daß für die Durchführung eines automatisierten Tests wichtige Voraussetzungen zu treffen sind, damit der Test überhaupt funktionieren kann. Diese Voraussetzungen zu schaffen, stellt nicht nur eine hohe Anforderung an den Entwickler, sondern ist zudem noch aufwendig und anspruchsvoll.

Auch die Aussagen dieser Werkzeuge über die Fehleranfälligkeit oder die Wahrscheinlichkeit für Restfehler muß immer äußerst vorsichtig interpretiert werden. Hierzu sei das folgende Beispiel angeführt.

Es gibt Werkzeuge auf dem Markt, die durch das Parsen einer Funktion bzw. Methode auf mögliche Testfälle schließen. Dabei sind diese erzeugten Testfälle mit der Absicht gewählt, eine vollständige Pfad- oder Zweigüberdeckung zu erzeugen. Durch eine mit diesen Testfällen generierte hundertprozentige Pfadüberdeckung kann jedoch auf gar keinen Fall darauf geschlossen werden, daß damit auch der Algorithmus sicher und komplett getestet oder die prinzipielle Fehlerfreiheit einer Methode nachgewiesen wurde.

Die Aussage einer Pfadüberdeckung gilt vielmehr nur im Umkehrschluß. Sicher ist, sollte eine vom Entwickler vorher vorgegebene prozentuale Pfadüberdeckung durch die bisherigen Testfälle nicht erreicht werden, dies andeutet, daß noch weitere Testfälle entwickelt werden sollten, um das Programm, die Prozedur bzw. die Methode oder den Algorithmus komplett zu testen.

Diese neuen Testfälle können jedoch nicht von einem Tool entwickelt werden, da damit eine falsche Voraussetzung für die neuen Testfälle zugrunde gelegt wird. Schließlich deutet eine nicht komplette Pfadüberdeckung nur an, daß es noch nicht getestete Bereiche in der Software gibt. Werden nun Eingaben für eine Methode erzeugt, die zu einer kompletten Pfadüberdeckung führen, so sieht das Ergebnis zwar sehr gut aus, jedoch wurden damit nicht auch automatisch alle notwendigen Tests durchgeführt. Bei den funktionalen Tests steht der verwendete Algorithmus im Vordergrund. Daher sollte es eher die Aufgabe des Entwicklers oder eines seiner Kollegen sein, sich den verwendeten Algorithmus genau anzusehen, um den Grund für die nicht komplett erreichte Pfadüberdeckung zu erkennen. Dieses Vorgehen wurde bereits bei den Erläuterungen zu den Grey-Box-Verfahren dargestellt.

Beispielsweise könnte ein schlechtes Überdeckungsmaß dadurch entstehen, daß sowohl der Variableninhalt eines Divisors vor der Division auf Null überprüft und zusätzlich die `java.lang.ArithmeticException` abgefangen wird. Diese als übermäßige Absicherung zu bezeichnende Technik sollte definitiv vermieden und somit auch gesondert getestet werden.

Ein weiteres Problem, welches besonders für die Überdeckungsmessung im Zusammenhang mit der Objektorientierung entsteht, ergibt sich aus den Regeln des Polymorphismus. Es ist schließlich möglich, Methoden bei der Ableitung von Klassen absichtlich zu überlagern und damit auch nicht mehr zur Ausführung zu bringen. Ebenso kann es sein, daß die Methoden aus einer komplexen Serviceklasse nur teilweise wirklich benötigt werden. Dennoch werden auch solche Methoden in das Maß der Überdekkungsmessung mit einfließen und somit das Ergebnis negativ beeinflussen[20].

Die gebotene Vorsicht, mit der die Ergebnisse von automatisierten Testwerkzeugen zu interpretieren sind, gelten auch für das in dieser Arbeit zu entwickelnde Tool für

---

[20] s. Hasemann, M., 2001

funktionale Black-Box-Tests, vor allem dann, wenn ein solches Programm dem Tester auch die Interpretation der Ergebnisse abnimmt.

Ein Werkzeug zur Durchführung automatisierter funktionaler Tests wird aber auch in anderen Bereichen an seine Grenzen stoßen. Dabei sind die bereits angeführten Probleme der wohldefinierten Testumgebung nur ein Beispiel, wo der Entwickler bereits während der Implementierung von Funktionalitäten auf den späteren Test achten muß. Nicht selten ist das Problem der für den Testlauf fest zu definierenden Testumgebung mit aufwendigen und umfangreichen Lösungen verbunden.

Weitere Probleme bei der Anwendung eines automatisierten funktionalen Testwerkzeuges sind in der internen Struktur der zu testenden Methode oder Funktionalität zu vermuten. So könnte es z.B. relevant sein, ob ein iterativer oder rekursiver Algorithmus Verwendung findet oder ob es sich um eine auf Multithreading-Techniken basierende Software handelt.

Die Rekursion selbst stellt kein Problem für einen funktionalen Test dar, was sich bereits aus der Definition der Rekursion ergibt. Rekursion ist die Definition eines Problems, einer Funktion oder eines Verfahrens durch sich selbst, wobei die Iteration als Spezialfall der Rekursion angesehen wird[21], und nicht umgekehrt. Daraus ist zu schließen, daß ein Testwerkzeug, welches iterative Prozesse testen kann, mit dem allgemeineren Fall der Rekursion keine Probleme haben sollte.

Das dem so sein muß, wird deutlich, wenn man sich klar macht, daß ein funktionaler Test grundsätzlich ein Black-Box-Verfahren darstellt, so daß die interne Struktur der Methode völlig nebensächlich ist. Demnach darf es auch keine Rolle spielen, ob der verwendete Algorithmus iterativer oder rekursiver Natur ist.

Das einzige Problem liegt wiederum in der Automation der Tests begründet. Da die Rekursion im Prinzip das Pendant zu einer iterativen Schleife darstellt, sind beide Konstrukte, die Rekursion und auch die Schleifen, dann problematisch, wenn die Rekursion bzw. die iterative Schleife keine korrekte Abbruchsbedingung besitzen und somit theoretisch unendlich lange laufen.

Dieses Phänomen kann bei bestimmten Eingabewerten durchaus auftauchen und als spezieller Test sogar erwünscht sein. Damit muß nur das Testwerkzeug auch in der Lage

---

[21] s. DUDEN der Informatik, 1993, S. 601

sein, eine Endlosschleife zu erkennen. Bei der Rekursion ist der Endlosigkeit meistens eine Grenze durch den Stapel gegeben. Sehr schnell wird ein Stapelüberlauf auftreten, der zu einem Abbruch führt. Bei einer iterativen Schleife dagegen kann allerhöchstens eine Laufvariable, falls vorhanden, einen großen Wert annehmen und schließlich bei einem Überlauf der Wertgrenze einen Abbruch erzeugen, wenn die verwendete Programmiersprache nicht schlicht mit einem sog. „wrap around“ reagiert und die Schleife damit von vorne beginnt.

Lösen läßt sich dieses Problem leider nur durch den Einsatz eines Timers, der nach einem mehrere Minuten andauernden Halt des Programms den gerade durchgeführten Test abbricht.

Im Falle des Multithreadings ergeben sich hingegen wesentlich größere Probleme. Nehmen wir an, es soll ein Webserver implementiert werden, der auf bestimmte Anfragen horcht und entsprechend der Anfrage etwas an den Fragenden zurück liefern soll. Dabei können natürlich mehrere Anfragen gleichzeitig auftreten, so daß ein Webserver grundsätzlich als ein auf Multithreading basierendes System implementiert wird.

Soll ein solches System getestet werden, kann dabei die Multithreading-Umgebung nur schlecht simuliert werden. Ein entsprechendes Tool müßte in der Lage sein, sich wie mehrere Anfrager zu verhalten, die ermittelten Ergebnisse sammeln und vergleichen. Wird jedoch ein solches Vorgehen angestrebt, ist dies genauso schlecht, als würde man einen funktionalen Test direkt an der GUI einer Software durchführen. Die Zurechenbarkeit eines Fehlers leidet extrem darunter.

Daraus folgt die Konsequenz, daß einzelne Bereiche des zu implementierenden Webservers getrennt getestet werden müssen. Dabei ist es auch vorteilhaft, so lange wie möglich auf das Multithreading zu verzichten. So kann beispielsweise die Methode zur Auswertung der Anfrage auch separat und ohne die tatsächliche Existenz eines oder mehrerer Browser überprüft werden.

Voraussetzung dafür ist selbstverständlich, daß eine solche Funktionalität so weit wie möglich von den speziellen Gegebenheiten eines Webservers getrennt wird. Wird für jedes zu testende Element dieses Webservers darauf geachtet, daß es in einer eigenen Methode existiert, der es egal sein kann, woher seine Daten stammen, dann ist auch ein funktionaler Test mit einem automatisierten Testwerkzeug möglich.

Beispielsweise könnte man zwecks der visuellen Verdeutlichung dafür sorgen, daß die eigentliche Businesslogik bzw. der Kern des Webservers in einer eigenen Klasse vorhanden ist, der als Schnittstelle oder als Eingabegerät den Webserver selbst erhält. Damit könnte diese zusätzliche Schicht auch separat getestet werden.

Mit diesem Verfahren wäre es allerdings schlecht möglich, zu überprüfen, ob der Webserver auch wirklich „Threadsave“ implementiert wurde, also die einzelnen Methoden auch reentrant sind. Zu diesem Zweck müßte der Entwickler entweder großzügig manuell instrumentieren, also Ausgaben in seinem Programm einstreuen, oder aber auf ein anderes Tool zurückgreifen.

## *2.5 Automation von Softwaretests am Beispiel des ggT-Algorithmus*

Die nun folgenden Beispiele sollen verdeutlichen, wie das Testen einer Software bzw. einer einzelnen Methode oder Funktionalität grundsätzlich abläuft. Dabei werden zwei verschiedene Ansätze gezeigt. Im ersten Schritt wird die zu testende Methode nach prozeduralen Gesichtspunkten erstellt und getestet, im zweiten Beispiel werden die Unterschiede zu den objektorientierten Sprachen aufgezeigt.

Um Tests aus der Kategorie der Black-Box- oder Grey-Box-Verfahren erfolgreich zu automatisieren ist es notwendig, den Ablauf der Tests in irgendeiner Form festzulegen. In der Regel wird dies durch ein Skript erfolgen. In diesen ersten Beispielen wird jedoch noch kein Skript eingesetzt werden, sondern die Testfälle werden direkt in Java programmiert.

Durch die Implementierung der Tests mit Java selbst entsteht eine statische Version eines Testtools. Trotzdem vermag es auch diese Technik, alle entscheidenden Tests automatisch durchzuführen und ein Ergebnis auszugeben.

### 2.5.1 Prozeduraler Ansatz des euklidischen ggT-Algorithmus

An dem folgenden fehlerhaften Beispiel zur Berechnung des größten gemeinsamen Teilers zweier natürlicher Zahlen soll gezeigt werden, wie ein automatisierter Test der Methode aussehen könnte.

Wie bereits erläutert, ist für die Durchführung eines Tests zuerst eine Testumgebung zu definieren. Da es sich bei diesem Beispiel um eine Methode handelt, die in keiner

Abhängigkeit steht, ist nur ein Treibermodul zu implementieren, welches die Aufrufe regelt. Im ersten Schritt soll dieses Treibermodul manuell erstellt werden und dann die entsprechenden Testfälle durchführen. Später wird dann das Testtool entstehen, welches prinzipiell als großes Treibermodul angesehen werden kann, dessen Verhalten durch ein Skript festgelegt wird.

Die Methode erwartet zwei natürliche Zahlen `a` und `b`, die vom Typ `long` sind. Da es in Java keinen Typ gibt, der einen größeren Wertbereich aufweist, kann vermutet werden, daß ein Überlauf auszuschließen ist. Dieser würde nur bei einer Benutzereingabe und anschließender Konvertierung in einen Zahlenwert auftreten können.

**Abbildung 6: Fehlerhafter Code zur Berechnung des ggT**

```
public long getGGT(long a, long b)
{
    while (true)
    {
        long r = a % b;
        if (r!=0) return b;

        a=b; b=r;
    }
}
```

(Quelle: eigene Darstellung)

Im Folgenden werden einige sinnvolle Tests vorgestellt. Die Notation der Testwerte sei in der Form {Parameter a, Parameter b, erwartetes Ergebnis}. Es gibt noch weitere Testwerte, welche sinnvoll wären, wie z.B. zwei negative Primzahlen, aber zur Verdeutlichung des Prinzips reichen diese Fälle aus.

{0,0,0}, {5,0,5}, {0,5,5}, {5,7,1}, {8,4,4}, {8,2,2}, {8,1,1}, {-8,-2,-2}, {-8,2,2}

Das zu schreibende Testprogramm muß erst eine Instanz der Klasse `GGT` anlegen, in welcher die Methode `getGGT()` enthalten ist. Danach müssen sukzessive alle zu testenden Werte einmal an die Methode übergeben und das Ergebnis mit dem richtigen verglichen werden. Als Fehler sind sowohl falsche Rückgabewerte sowie das Auftreten einer beliebigen Exception zu werten.

Die folgende Abbildung zeigt den Quellcode des Programms. Aus Gründen der Vereinfachung wird auf die komplette Klassendefinition verzichtet und nur die `Main(..)`-Methode aufgeführt.

**Abbildung 7: Programm zum Testen der getGGT()-Methode**

```
public static void main(String[] args)
{
    long [][] TestWerte = {{0,0,0}, {5,0,0}, {0,5,0}, {5,7,1}, {8,4,4},
                           {8,2,2}, {8,1,1}, {-8,-2,-2}, {-8,2,2}};
    GGT theClass = new GGT();
    long ergebnis;

    for (int index=0; index<TestWerte.length; index++)
    {
        try
        {
            ergebnis = theClass.getGGT(TestWerte[index][0], TestWerte[index][1]);
        }
        catch (Throwable ex)
        {
            System.out.println("Fehler: Die Werte "+TestWerte[index][0]+", "+TestWerte[index][1]+
                               " führten zu der Exception \""+ex.toString()+
                               "\" statt zu "+TestWerte[index][2]);
            continue;
        }

        if (TestWerte[index][2]!=ergebnis)
            System.out.println("Fehler: Die Werte "+TestWerte[index][0]+", "+TestWerte[index][1]+
                               " führten zu dem Ergebnis "+ergebnis+
                               " statt zu "+TestWerte[index][2]);
        else
            System.out.println("OK    : Die Werte "+TestWerte[index][0]+", "+TestWerte[index][1]+
                               " führten zu dem Ergebnis "+ergebnis+
                               ", Vorgabe:"+TestWerte[index][2]);
    }
}
```

(Quelle: eigene Darstellung)

Der Testlauf ergibt dann die folgende Ausgabe. Auffällig sind die häufig aufgetretenen `java.lang.ArithmeticException`, die auf eine Division durch Null zurückzuführen sind.

```
Fehler: Die Werte 0, 0 führten zu der Exception "java.lang.ArithmeticException" statt zu 0
Fehler: Die Werte 5, 0 führten zu der Exception "java.lang.ArithmeticException" statt zu 5
Fehler: Die Werte 0, 5 führten zu der Exception "java.lang.ArithmeticException" statt zu 5
Fehler: Die Werte 5, 7 führten zu dem Ergebnis 7 statt zu 1
Fehler: Die Werte 8, 4 führten zu der Exception "java.lang.ArithmeticException" statt zu 4
Fehler: Die Werte 8, 2 führten zu der Exception "java.lang.ArithmeticException" statt zu 2
Fehler: Die Werte 8, 1 führten zu der Exception "java.lang.ArithmeticException" statt zu 1
Fehler: Die Werte -8, -2 führten zu der Exception "java.lang.ArithmeticException" statt zu -2
Fehler: Die Werte -8, 2 führten zu der Exception "java.lang.ArithmeticException" statt zu 2
```

Da der Code an der Stelle `if (r!=0) return b;` fehlerhaft ist, haben die Testfälle nicht funktionieren. Vor allem die ersten Tests, bei denen `a` oder `b` gleich 0 ist, mußten in Ermangelung einer entsprechenden Abfrage fehlschlagen. Nach der Anpassung der `getGGT()`-Methode, wie in der folgenden Abbildung dargestellt, sieht die Ausgabe besser aus.

**Abbildung 8: Korrigierte getGGT-Methode**

```
public long getGGT(long a, long b)
{
    if (b==0) return a;
    while (true)
    {
        long r = a % b;
        if (r==0) return b;

        a=b; b=r;
    }
}
```

(Quelle: eigene Darstellung)

Ausgabe des Testlaufes:

```
OK    : Die Werte 0, 0 führten zu dem Ergebnis 0, Vorgabe:0
OK    : Die Werte 5, 0 führten zu dem Ergebnis 5, Vorgabe:5
OK    : Die Werte 0, 5 führten zu dem Ergebnis 5, Vorgabe:5
OK    : Die Werte 5, 7 führten zu dem Ergebnis 1, Vorgabe:1
OK    : Die Werte 8, 4 führten zu dem Ergebnis 4, Vorgabe:4
OK    : Die Werte 8, 2 führten zu dem Ergebnis 2, Vorgabe:2
OK    : Die Werte 8, 1 führten zu dem Ergebnis 1, Vorgabe:1
OK    : Die Werte -8, -2 führten zu dem Ergebnis -2, Vorgabe:-2
OK    : Die Werte -8, 2 führten zu dem Ergebnis 2, Vorgabe:2
```

Bei diesem, nun erfolgreichen Testlauf war sehr gut der Vorteil eines programmierten Tests zu beobachtet. Nach der Korrektur des Programmes konnten alle Tests einfach wiederholt werden. Dies ist in der Praxis vor allem bei einem Redesign äußerst vorteilhaft. Schließlich kann so sehr einfach gezeigt werden, daß keine der Änderungen das Ergebnis des Tests beeinflußt haben. Die Qualität der Aussage, daß das Programm wei

terhin fehlerfrei arbeite, ist jedoch selbstverständlich nur so gut, wie die verwendeten Testfälle.

### 2.5.2 Objektorientierter Ansatz des euklidischen ggT-Algorithmus

Das obige Beispiel zur Berechnung des ggT ist rein prozedural verfaßt. Bei einer vergleichbaren objektorientierte Implementierung würde die Methode `getGGT()` in einer Klasse zur Darstellung natürlicher Zahlen untergebracht werden.

**Abbildung 9: Klasse „Number“ mit ggT-Berechnung**

```
package ggt_test;

public class Number
{
    private long theValue;

    public Number()
    {
        super();
    }

    public Number(long aLongValue)
    {
        this();
        theValue = aLongValue;
    }

    public long getGGT(Number aNumber)
    {
        long a = this.theValue;
        long b = aNumber.getTheValue();

        if (b==0) return a;
        while (true)
        {
            long r = a % b;
            if (r==0) return b;

            a=b; b=r;
        }
    }

    public long getTheValue()
    {
        return theValue;
    }

    public void setTheValue(long newTheValue)
    {
        theValue = newTheValue;
    }
}
```

(Quelle: eigene Darstellung)

Diese Klasse mit dem beispielhaften Namen `Number` stellt den Parameter `a` dar. Somit benötigt die Methode `getGGT()` nur noch den Parameter `b`, der wiederum vom Typ

der Klasse `Number` ist. Die grundsätzliche Konstruktion der Methode ist vollkommen äquivalent zu der ursprünglich prozeduralen `getGGT()`-Methode. Der Rückgabewert bleibt vorerst vom Typ `long`, obwohl auch er vom Typ `Number` sein könnte.

Das bereits vorgestellte Programm zur Durchführung der Tests ist nun entsprechend anzupassen. Statt einer einmaligen Instanziierung der Klasse `GGT` ist nun für jeden Testfall eine neue Instanz der Klasse `Number` notwendig, da sie für den ersten Testparameter `a` steht.

**Abbildung 10: Neue Version des Testprogrammes (Ausschnitt)**

```
private void testObjectVersion()
{
    long [][] TestWerte = {{0,0,0}, {5,0,5}, {0,5,5}, {5,7,1}, {8,4,4},
                           {8,2,2}, {8,1,1}, {-8,-2,-2}, {-8,2,2}};
    long ergebnis;

    for (int index=0; index<TestWerte.length; index++)
    {
        try
        {
            Number aNumber = new Number(TestWerte[index][0]);
            Number bNumber = new Number(TestWerte[index][1]);
            ergebnis = aNumber.getGGT(bNumber);
        }
        .
        .
        .
```

(Quelle: eigene Darstellung)

Wie zu erkennen, ist das Vorgehen bei der Durchführung des Tests etwas anders. Nun müssen für jeden Test zwei neue Klassen instanziiert werden und das Ergebnis ist das Resultat des Methodenaufrufs „`getGGT()`“ der Klasse „`Number`“.

Diese Methode kommt in beiden instanziierten Objekten „`aNumber`“ und „`bNumber`“ vor, doch genügt es, nur eine davon zu testen. Dies wäre unter prozeduralen Gesichtspunkten nicht so, da dort keine Gleichheit der beiden Methoden garantiert werden kann.

### 2.5.3 Unterschiede der beiden Techniken

Gerade unter Berücksichtigung des noch zu entwickelnden Skriptes ist es notwendig, trotz der unterschiedlichen Anforderungen eine gemeinsame Basis zu finden, um darauf eine Notation für ein Skript aufzubauen.

Auffällig ist, daß nun nicht mehr einfache atomare Datentypen verwendet werden, sondern mindestens ein Objekt. Dieses Objekt muß nach seiner Instanziierung seinen Wert erhalten. Dies kann bereits über den Konstruktor erfolgen, durch eine direkte Zuweisung oder aber durch den Aufruf einer oder mehrerer Setter-Methoden[22]. Da auch die Setter-Methoden wieder Objekte als Parameter erwarten können, muß diese Instanziierung rekursiv abgehandelt werden können. Demnach muß auch das zu entwickelnde Skript diese rekursive Struktur unterstützen.

Ebenso unterschiedlich zu der vorherigen prozeduralen Lösung ist, daß nun die zu testende Methode in der Klasse des ersten Parameters steht und nicht, wie vorher, in einer eigenen Instanz. Ferner wird auch der Ergebnistyp entweder ein atomarer Typ oder aber ein Objekt sein. Somit muß auch die Vergleichsfunktion entscheiden können, ob sie es mit einem Objekt oder aber einem atomaren Datentyp zu tun hat.

Ebenfalls zu berücksichtigen ist, ob es sich bei der Übergabe der Parameter an die zu testende Methode um die Form *call by value*[23] oder *call by reference* handelt. Schließlich kann bei *call by reference* auch ein zu überprüfender Rückgabewert in einem Parameter übermittelt werden. Die Technik des *call by reference* ist in Java nur mit Objekten möglich[24], dann allerdings müssen bei zu erwartenden Veränderungen des als Parameter übergebenen Objektes diese ebenfalls überprüft werden, um unerwünschte Seiteneffekte ausschließen zu können.

Die Übergabeformen *call by name* und *call by value-result* werden in den heutigen Sprachen nur noch selten verwendet und sind in Java gar nicht mehr enthalten.

---

[22] als Setter-Methode bezeichnet man Methoden, deren einzige Aufgabe darin besteht, privaten Instanzvariablen einen der Methode übergebenen Wert zuzuweisen. Das Pendant zu den Setter-Methoden sind die Getter-Methoden.

[23] s. DUDEN Informatik, 1993, S. 121 ff.

[24] s. Krüger, G., 2001, S. 136 f.

# 3 Implementierung des Tools

Im folgenden Abschnitt soll die Implementierung des Testwerkzeuges besprochen werden. Dabei wird nach der Vorstellung des Pflichtenheftes erst erläutert, welcher Form das Testskript sein soll. Danach wird gezeigt, wie mit atomaren Datentypen, Datenstrukturen und Aufzählungstypen umgegangen werden sollte, um dann auf den erhaltenen Erkenntnissen den Vergleich von Objekten als das essentielle Element des Testwerkzeuges vorzustellen.

## *3.1 Anforderungsdefinition*

Es soll ein Programm entwickelt werden, daß automatisierte funktionale Tests an einer Software durchführen kann. Es soll möglich sein, einzelne Methoden einer Klasse im Black-Box- oder im Grey-Box-Verfahren zu testen. Ein Testdurchlauf soll prinzipiell der Form sein, daß eine Methode mit bestimmten Parametern aufgerufen und das Ergebnis auf Korrektheit überprüft wird. Die Testfälle selbst sollen in einem Skript untergebracht werden, dessen Form hier nicht festgelegt wird. Die Implementierung soll dabei in mehreren Stufen erfolgen.

In der ersten Stufe sollen das Skript und der Interpreter entwickelt werden. Folgende Funktionalitäten sind dabei bereits zu implementieren:

- Instanziieren von Klassen via Konstruktor oder Setter-Methoden
- Aufruf von statischen Methoden oder Instanzmethoden
- Abfragen von Klassen- oder Instanzvariablen
- Direktes Setzen bzw. verändern von Klassen- und Instanzvariablen
- Definition von Arrays zu Testzwecken
- Definition von Parameterlisten für Methodenaufrufe und Konstruktoren
- Vergleichen von Resultaten
- Typkonvertierung
- Abbildung von primitiven Typen, Klassen und Objekten

Für den Zugriff auf Methoden oder Variablen einer Klasse soll es egal sein, ob diese als `private`, `protected` oder als `public` deklariert wurden. Andernfalls wäre ein komplettes oder umfassendes Testen nicht möglich.

Die Tests sollen jederzeit wiederholt werden können und nachvollziehbar sein. Außerdem soll die Software für neue zusätzliche Aufgaben erweitert werden können, die sich möglicherweise erst aus ihrem späteren Einsatz ergeben.

Die Software soll Auswertungen, Ergebnisse von Vergleichen, Fehler und ein Ausführungsprotokoll ausgeben. Ferner müssen Ausführungs- und Ausnahmefehler, sog. Exceptions, abgefangen werden. Die Ausführung der Software darf jedoch nicht aufgrund von Ausführungs- oder Ausnahmefehlern abbrechen. Die Software muß zudem in der Lage sein, zwischen eigenen internen Fehlern, die sich möglicherweise aus einem falschen Testskript ergeben, und Fehler der getesteten Methode unterscheiden können. Es soll zudem möglich sein, daß eine von der zu testenden Methode ausgelöste Ausnahme (Exception) als gewünschtes Ergebnis getestet werden kann.

In der zweiten Stufe soll ein Werkzeug implementiert werden, daß die Ausgaben der Software sammelt und eine übersichtliche Auswertung ermöglicht. Zusätzlich soll der Vergleich mit früheren Tests möglich sein. Ferner soll ein Werkzeug zur Erstellung des Testskriptes implementiert werden, sofern dies möglich ist.

## *3.2 Grundstruktur des Testskripts*

Um die automatisch ablaufenden Tests zu beschreiben, ist die Erstellung eines Skriptes unumgänglich. Dieses muß mindestens die für die erste Stufe der Implementierung festgelegten Aufgaben abbilden können, damit das Testen einer auf objektorientierten Ansätzen beruhenden Anwendung möglich wird.

Diese Liste ist bisher nicht vollständig und wird je nach Anforderung noch zu erweitern sein, sollen z.B. auch graphische Oberflächen getestet werden. Ferner muß dieses Skript sehr viel Flexibilität bieten, um wirklich alle notwendigen Black-Box-Tests durchführen zu können.

Die Abbildung dieser Aufgaben kann auf verschiedene Arten realisiert werden. Möglich ist die Implementierung der Testfälle direkt in Java, wie es in dem Einfüh

rungsbeispiel geschehen ist, oder die Verwendung von Pseudocode oder die Verwendung eines Skriptes auf der Basis von XML[25].

### 3.2.1 Realisation des Skriptes direkt in Java

Es ist denkbar, daß gar kein Skript zum Einsatz kommt, sondern die Testfälle, dem obigen ggT-Beispiel folgend, selbst in Java geschrieben werden. Ein automatisch ablaufendes Tool würde dann nur noch die einzelnen Testmethoden aufrufen und die Ergebnisse protokollieren. Dieses Vorgehen wird beispielsweise mit dem Tool „JUnit“[26] verfolgt.

Vorteilhaft wäre, daß der Entwickler schnell und ohne seine gewohnte Umgebung verlassen zu müssen, Testfälle aufbauen könnte, die dann in die Ausführung des Tools mit eingegliedert würden. Ferner kann der Entwickler durch den Einsatz der gleichen Sprache auch alle Möglichkeiten, die ihm hier geboten werden, voll ausschöpfen. Tests auf dieser Basis sind in jeder Programmiersprache realisierbar.

Als problematisch anzusehen ist jedoch die Tatsache, daß auch in diesem Testcode wieder sehr viele Fehler auftreten können. Ferner wäre nur ein Entwickler in der Lage, Testfälle zu programmieren. Gerade das jedoch soll nicht mehr der Fall sein, auch wenn der Entwickler weiterhin maßgeblich an der Erstellung der Testfälle beteiligt sein wird. Das Hauptaugenmerk soll schließlich auf der Entwicklung geeigneter Testfälle liegen, und nicht auf deren Umsetzung in der Zielsprache.

Ebenfalls als problematisch anzusehen sind die zu erwartenden Probleme bei der Abfrage des Zustandes eines Objektes. Sollen hierzu alle Member-Variablen ausgelesen werden, so wird es nicht möglich sein, als privat gekennzeichnete Fields direkt auszulesen.

In einem solchen Fall müssen daher immer entsprechende Zugriffsmethoden existieren. Diese können jedoch Funktionalitäten enthalten, welche den Inhalt der abgefragten Klassen- oder Instanzvariablen verfälschen. Beispielsweise könnte die getter-Methode einer Variablen bei einem für die Logik der Klasse ungültigen Wert statt dem Wert selbst, einen anderen Wert, z.B. null, zurückgeben oder eine Exception auslösen.

---

[25] s. http://www.w3schools.com/xml/default.asp

[26] s. http://www.junit.org

Das gleiche Problem stellt sich bei dem Vergleich zweier Objekt auf Identität. Hier genügt die in der Java-Klasse `java.lang.Object` vorhandenen `equals()`-Methode i.d.R. nicht mehr. Sie funktioniert nur bei identischer Referenz oder bei den Wrapper-Klassen. In allen anderen Fällen ist man gezwungen, die Inhalte aller Klassen- und Instanzvariablen eines Objektes mit dem Inhalt der korrespondierenden Variablen in dem Vergleichsobjekt auf Identität zu untersuchen.

Da es sich bei jeder dieser Variablen erneut um ein Objekt handeln könnte, wäre das gleiche Vorgehen nötig. Der sich ergebende Vergleichsalgorithmus ist somit rekursiv zu implementieren und damit auch sehr komplex. Aus diesem Grund ist es zweckmäßig, eine fertige Vergleichsmethode für die Überprüfung zweier beliebiger Objekte auf Identität als Serviceroutine anzubieten.

### 3.2.2 Realisation des Skriptes mit Pseudocode

Eine weitere Möglichkeit wäre der Einsatz von Pseudocode. Dieser könnte auf das wesentliche reduziert sein, um die Fehleranfälligkeit und den Lernaufwand zu begrenzen. Dadurch könnten auch solche Entwickler Testfälle erstellen, die mit der eigentlichen Sprache nicht sonderlich vertraut sind.

Beispielhaft könnten die Befehle des Pseudocodes zur Durchführung einer der vorherigen Tests zur `getGGT()`-Methode der folgenden Form sein:

```
create class(„ggt_test.Number“) name(„myNumber1“) parameter(„8“).
create class(„ggt_test.Number“) name(„myNumber2“) parameter(„4“).
result := call method(„getGGT“) inInstance(„myNumber1“)
          parameter(instance(„myNumber2“)).
compare result with (primitiv type(„long“) value(„4“)).
```

Durch den Einsatz von Pseudocode wird ein Interpreter oder Compiler notwendig. Damit wäre das Problem der privaten Klassen- und Instanzvariablen in der Form lösbar, daß nun das Tool selbst die entsprechenden Routinen zur Verfügung stellt und diese nicht erst programmiert werden müßten. Zusätzlich wird durch die Verlagerung des Vergleichsalgorithmus von dem Programm des Entwicklers in das Tool sichergestellt, daß der Programmierer keine Möglichkeiten hat, etwas zu beschönigen.

Hinzu kommt, daß der für diesen Pseudocode nötige Interpreter oder Compiler die Ausführung überwacht, was einen weiteren Vorteil gegenüber den manuell erstellten Testprogrammen darstellt. Sollte z.B. eine Instanziierung fehlschlagen, so kann ein entsprechend programmiertes Tool dies melden, ohne die weitere Bearbeitung der Testfälle abzubrechen.

Die Einführung eines solchen Skriptes hat jedoch den entscheidenden Nachteil, daß es einerseits immer noch sehr komplex ist und auf der anderen Seite die Implementierung eines Parsers nötig macht. Solche Parser können zwar durch sog. „Compiler-Compiler“, wie beispielsweise dem JavaCC für die Zielsprache Java, aus einer Grammatik generiert werden, die Erstellung der zugehörigen Grammatik jedoch wäre äußerst komplex.

### 3.2.3 Einsatz von XML

Soll ein Skript eingesetzt werden, ergeben sich zwei Möglichkeiten, wie dieses umgesetzt werden kann. Bietet die Zielsprache die Möglichkeit, eine Klasse zu instanziieren und dieser Instanz Nachrichten zu schicken, ohne dafür Code zu verwenden, wie es unter Java mit dem Reflection-API möglich ist, so ist die Verarbeitung des Skriptes kein großes Problem. Es muß nur interpretiert werden.

Bei anderen Sprachen dagegen müßte aus dem Skript durch einen Compilervorgang erst Code generiert werden, der dann für den eigentlichen Test ausgeführt wird. Dieses Vorgehen entspricht grundsätzlich der direkten Erstellung von Code in der Zielsprache, was den zusätzlichen Einsatz eines zu kompilierenden Skriptes als überflüssig erscheinen läßt.

Für die Verwendung eines Skriptes spricht jedoch, neben der Möglichkeit, die Beschreibung von Testfällen auf das Wesentliche zu reduzieren, auch die Idee, eine Grammatik für ein Skript zu entwickeln, die in der Lage wäre, unabhängig von der Zielsprache Testfälle zu beschreiben. Zur Durchführung der Testfälle wäre nur noch der entsprechende Interpreter oder Compiler nötig. Dieser Vorteil gilt sowohl für Pseudocode, als auch für XML.

Doch gerade unter dem Gesichtspunkt der Portabilität ist XML zu einem Standard geworden. XML ist eine Teilmenge der SGML und somit eine deklarative Auszeich

nungssprache. Sie bietet die Möglichkeit, eigene Tags zu definieren und mit der Hilfe einer DTD[27] deren Syntax bzw. Grammatik festzulegen.

In der folgenden Abbildung ist dargestellt, wie der obige in Pseudocode formulierte Testfall in XML aussehen würde.

**Abbildung 11: XML-Beispiel für ein Test-Skript**

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE testscript SYSTEM "d:/XML/TestSkript.DTD">

<testscript>
   <class instancename="myNumber1" classname="ggt_test.Number">
      <parameters>
         <primitiv type="long" value="8" />
      </parameters>
   </class>

   <class instancename="myNumber2" classname="ggt_test.Number">
      <parameters>
         <primitiv name="Value(4)" type="long" value="4" />
      </parameters>
   </class>

   <result name="erster_vergleich">
      <call instancename="myNumber1" name="getGGT">
         <parameters>
            <instance name="myNumber2" />
         </parameters>
      </call>
   </result>

   <compare>
      <primitiv name="Value(4)" />
      <result name="erster_vergleich" />
   </compare>
</testscript>
```

(Quelle: eigene Darstellung)

Die DTD zu einem XML-Dokument kann entweder mit in dem XML-Dokument untergebracht werden, oder mit einer externen Referenz angegeben werden, wie in der

[27] s. http://www.w3schools.com/dtd/dtd_intro.asp für eine komplette Beschreibung der dortigen Syntax

Abbildung dargestellt. Die komplette DTD für dieses Beispiel befindet sich im ersten Anhang.

Da XML-Dokumente auf ASCII basieren und ihre eigene Grammatik mitbringen, können XML-Parser sehr allgemein formuliert werden. Ferner wird durch die mitgelieferte Grammatik eine Syntaxüberprüfung ermöglicht, die gleich von dem XML-Parser vorgenommen werden kann. Somit muß nur die Grammatik entwickelt werden, nicht aber der gesamte Parser. Ein weiterer Vorteil der XML gegenüber anderen propritären Formaten ist, daß auch Menschen den Inhalt solcher Dokumente lesen können.

Zusätzlich zu den bisher genannten Vorteilen gelten für XML auch die Vorteile des Pseudocodes, also die Einschränkung des „Befehlssatzes“ auf die wichtigsten Aufgaben, die geringere Komplexität und die überwachte Ausführung.

## *3.3 Die Grammatik des XML-Skriptes*

Die Grammatik eines XML-Dokumentes wird, wie bereits erläutert, in einer sog. DTD abgelegt. Im folgenden soll diese DTD sukzessive anhand der unter Punkt 3.2 gestellten Anforderungen erzeugt werden.

Da die Tags z.T. rekursiv verschachtelt sind, wird die DTD von „innen nach außen“ beschrieben. Es wird jeweils das Tag erläutert, seine Parameter vorgestellt und sein entsprechender Ausschnitt aus der DTD gezeigt.

### 3.3.1 Das Tag <primitiv>

Die erste Aufgabe, die ein Testskript leisten können muß, ist die Abbildung von primitiven Datentypen[28]. Dabei handelt es sich um die unter Java nicht als Objekte abgebildeten atomaren Typen boolean, byte, char, short, int, long, float, double und void. Letzterer ist dabei nur von geringerer Bedeutung und wird von dem Tag `<primitv>` auch nicht unterstützt.

Zusätzlich zu den atomaren Typen kann dieses Tag auch die entsprechenden Wrapper-Klassen und den Typen „`java.lang.String`“ abbilden, sowie auch Objekte solcher Klassen, die sich in ein Objekt des Typs „`BasicDataType`“ unterbringen lassen. Dies

[28] s. Erläuterungen zu der Klasse „BasicDataType“ und im Glossar

sind Objekte von Klassen, die nur eine Variable abbilden und damit auch Wrapper darstellen, und deren Konstruktor nur einen String erwartet.

Dieses Tag hat die drei Parameter `name`, `type` und `value`, wobei `type` den Typ und `value` den Wert des primitiven Typen aufnimmt. Mit dem Attribut `name` kann der Datentyp wieder angesprochen werden. In der DTD sieht die Beschreibung dieses Tags folgendermaßen aus:

```
<!ELEMENT primitiv   EMPTY>
<!ATTLIST primitiv name        CDATA #IMPLIED>
<!ATTLIST primitiv type        CDATA #IMPLIED>
<!ATTLIST primitiv value       CDATA #IMPLIED>
```

### 3.3.2 Das Tag <member>

Um Instanz- bzw. Klassenvariablen aus der Instanz einer Klasse abfragen zu können, wird das Tag <member> eingeführt. Es benötigt als Parameter nur den Namen der Instanz und den Namen der Variablen.

Um auch statische Variablen abfragen zu können, kann statt `instancename` das Attribut `classname` verwendet werden, wo statt einer Instanz der Klassenname angegeben wird. Es dürfen jedoch nicht beide angegeben werden.
Ebenfalls kann mit diesem Tag auch der Inhalt der angegebenen Klassen- oder Instanzvariablen verändert werden. Dazu muß das Tag einen Ergebnistyp umschließen, der aus den Tags `<result>`, `<call>`, `<instance>`, `<class>`, `<primitiv>`, `<member>` oder `<array>` abgeleitet wird.

Damit sieht die Beschreibung in der DTD folgendermaßen aus:

```
<!ELEMENT member     (result | call | instance | class | primitiv |
                      member | array)?>
<!ATTLIST member name          CDATA #REQUIRED>
<!ATTLIST member instancename  CDATA #IMPLIED>
<!ATTLIST member classname     CDATA #IMPLIED>
```

### 3.3.3 Das Tag <array>

Neben den primitiven Datentypen wird es auch nötig sein, ein Array anlegen zu können. Diese Definition kann mit dem Tag `<array>` vorgenommen werden. Es hat nur das Attribut `name`, um das erzeugte Array verwenden zu können. Dieses Attribut ist jedoch nicht zwingend anzugeben. Die von diesem Tag eingeschlossenen Elemente werden dem benannten Array automatisch hinzugefügt. Sie dürfen nur von einem Typ sein. Um mehrdimensionale Arrays zu erzeugen, ist die Verschachtelung mehrerer Arrays notwendig, die entweder vorher definiert wurden, oder ihre Elemente an Ort und Stelle erfahren. Genauere Erläuterungen zu Arrays sind in dem Kapitel „Abbildung von Arrays“ zu finden. Die DTD für <array> hat somit folgendes Aussehen:

```
<!ELEMENT array (result | call | instance | class | primitiv |
                 member | array)*>
<!ATTLIST array name           CDATA #IMPLIED>
```

### 3.3.4 Das Tag <instance>

Ferner muß es möglich sein, einmal angelegte Instanzen von Klassen wiederverwenden zu können, beispielsweise als Parameter für einen Methodenaufruf. Zu diesem Zweck wurde das Tag `<instance>` eingeführt, welches nur den Namen der Instanz benötigt. Zum Anlegen einer Instanz wird das `<class>`-Tag verwendet, welches unter Punkt 3.3.7 erläutert wird.

Somit hat die Beschreibung in der DTD die Form:

```
<!ELEMENT instance   EMPTY>
<!ATTLIST instance name       CDATA #REQUIRED>
```

### 3.3.5 Das Tag <parameters>

Mit dem Tag `<paramters>` wird ein Parameterblock eingeschachtelt, der dann für einen Konstruktor oder einen Methodenaufruf verwendet wird. Dieses Tag kann somit nicht allein existieren, sondern befindet sich immer optional unter den Tags `<call>`, `<settercall>` und `<class>`. Dieses Tag benötigt keine Parameter. Dafür kann es die

Tags <result>, <call>, <instance>, <class>, <primitiv>, <member> und <array> umschließen.

In der DTD sieht die Beschreibung somit folgendermaßen aus:

```
<!ELEMENT parameters (result | call | instance | class | primitiv |
                      member | array)+>
```

### 3.3.6 Das Tag <call> und das Tag <settercall>

Diese beiden Tags dienen dem Aufruf von Methoden, wobei <settercall> eine spezielle Variante von <call> darstellt. Für einen Methodenaufruf wird der Name der Instanz, der Name der Methode und ein Parameterblock benötigt. Da es durchaus parameterlose Methoden gibt, muß der Parameterblock optional sein. Um auch statische Methoden aufrufen zu können, kann statt des Attributs instancename das Attribut classname verwendet werden. Es dürfen jedoch nicht beide Attribute angegeben werden, da sie sich gegenseitig ausschließen.

Das Tag <settercall> wird bei der Instanziierung von Klassen verwendet, um notwendige Setter-Methoden direkt nach der Instanziierung aufrufen zu können. Da dieses Tag somit nur innerhalb einer Instanziierung vorkommen darf, kann der Instanzname der Klasse wegfallen, was den einzigen Unterschied zu dem normalen <call>-Tag darstellt.

In der DTD ergibt sich somit die folgende Beschreibung für <settercall>:

```
<!ELEMENT settercall (parameters?)>
<!ATTLIST settercall name      CDATA #REQUIRED>
```

und für <call>:

```
<!ELEMENT call       (parameters?)>
<!ATTLIST call name            CDATA #REQUIRED>
<!ATTLIST call instancename    CDATA #IMPLIED>
<!ATTLIST call classname       CDATA #IMPLIED>
```

### 3.3.7 Das Tag <class>

Zur Instanziierung von Klassen dient das Tag `<class>`, das zwei Parameter benötigt. Mit dem ersten Parameter `instancename` wird der Instanzname angegeben, unter dem die Klasse mit den Tags `<call>`, `<member>` und `<instance>` wieder angesprochen werden kann, mit dem zweiten Parameter `classname` wird angegeben, welche Klasse instanziiert werden soll.

Das Tag `<class>` kann optional für den Aufruf des Konstruktors noch einen Parameterblock und mehrere `<settercalls>` enthalten.

Damit ergibt sich die folgende Definition für die DTD:

```
<!ELEMENT class      (parameters?, settercall*)>
<!ATTLIST class instancename  CDATA #REQUIRED>
<!ATTLIST class classname     CDATA #REQUIRED>
```

### 3.3.8 Das Tag <result>

Das Skript muß ferner die Möglichkeit bieten, das Ergebnis eines Aufrufs, den Inhalt einer Membervariablen, eine gesamte Klasse oder einen primitiven Typ mit einem Namen zu versehen und eventuell einen Typcast, also das Verändern des bisherigen Datentyps auf einen anderen, durchzuführen. Zu diesem Zweck gibt es die zwei Parameter `name` und `type`, wogegen letzter kein Pflichtfeld darstellt, da nicht immer ein Typecast nötig ist. Zudem schachtelt das Tag <result> beinahe alle bisher angesprochenen Tags, außer `<settercall>`, `<parameters>`, `<compare>` und natürlich sich selbst.

Beispielsweise kann der Aufruf einer Methode überall durchgeführt werden, das Ergebnis bzw. der Rückgabewert wird jedoch erst durch das umfassende Tag `<result>` gespeichert und bleibt nur so weiterhin, z.B. für einen Vergleich, ansprechbar.

Die Syntax der DTD für dieses Tag sieht folgendermaßen aus:

```
<!ELEMENT result ((call | instance | class | primitiv | member |
                   array)?)>
<!ATTLIST result name         CDATA #REQUIRED>
<!ATTLIST result type         CDATA #IMPLIED>
```

### 3.3.9 Das Tag <compare>

Dies ist das wohl wichtigste Tag, das für ein Testskript zu entwickeln ist. Dieses Tag soll den Vergleich zweier beliebiger Objekte aus beliebiger Herkunft durchführen und eine dem Ergebnis des Vergleichs entsprechende Meldung ausgeben.

Es werden aus diesem Grund alle Tags unterstützt, die für ein Objekt oder einen primitiven Datentyp stehen. Diese sind die gleichen Tags, die auch unter `<parameters>` stehen können: `<result>`, `<call>`, `<instance>`, `<class>`, `<primitiv>` und `<member>`.

Das Ergebnis des Vergleichs wird intern gespeichert, um nach der kompletten Durchführung des Skriptes noch eine statistische Auswertung erstellen zu können.

Daraus ergibt sich für die DTD die folgende Definition des Tags:

```
<!ELEMENT compare ((result | call | instance | class | primitiv |
                    member | array), (result | call | instance |
                    class | primitiv | member | array))>
```

### 3.3.10 Das Tag <testscript>

Bei dem letzten noch zu erläuternden Tag <testscript> handelt es sich um das globale, das gesamte Skript umschließende Tag. Ein solches alles umfassende Tag ist für eine XML-Datei vorgeschrieben und muß daher auch in der DTD aufgenommen werden. Es besitzt keine Parameter, doch legt es fest, welche Tags direkt unter diesem in dem Skript stehen dürfen. Dies sind die Tags `<class>`, `<call>`, `<result>`, `<compare>` und `<array>`, alle anderen Tags stehen immer unter einem dieser Tags.

Die Definition in der DTD hat somit folgendes Aussehen:

```
<!ELEMENT testscript (class | call | result | compare | array | member)+>
```

Die bisher aufgeführten Tags stellen eine erste funktionierende Basis dar, mit dessen Hilfe ein Testskript zur Durchführung von Black-Box- oder Grey-Box-Tests erstellt werden kann. Beispielsweise kann mit diesem Skript der komplette ggT-Test durchgeführt werden.

### 3.3.11 Das Tag <include>

Mit diesem Tag kann ein komplettes Skript eingebunden werden. Es wurde eingeführt, um die Übersichtlichkeit des Testskriptes zu erhöhen.. Das angegebene Skript muß eigenständig lauffähig, also ein komplettes Skript sein. Alle erzeugten Konstrukte stehen nach dem Aufruf des Skriptes dem „Vaterskript“ zur weiteren Verwendung zur Verfügung. Ebenso kann das eingebundene Skript auf bereits angelegte Objekte des Vaterskriptes zugreifen. Es findet keine Kapselung der Objekte statt.

**Abbildung 12: Beispiel eines Skripts zur Erzeugung von Testfällen**

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE testscript SYSTEM "d:/XML/TestSkript.DTD">

<testscript>
   <testsuite name="TestCase 1">
      <testcase>
         <class instancename="myNumber1" classname="ggt_test.Number">
            <parameters>
               <primitiv type="long" value="8" />
            </parameters>
         </class>

         <class instancename="myNumber2" classname="ggt_test.Number">
            <parameters>
               <primitiv type="long" value="4" />
            </parameters>
         </class>

         <primitiv name="result" type="long" value="4" />
      </testcase>
      <testcase>
         <class instancename="myNumber2" classname="ggt_test.Number">
            <parameters>
               <primitiv type="long" value="2" />
            </parameters>
         </class>

         <primitiv name="result" type="long" value="2" />
      </testcase>
      <testcase>
         <class instancename="myNumber2" classname="ggt_test.Number">
            <parameters>
               <primitiv type="long" value="0" />
            </parameters>
         </class>

         <primitiv name="result" type="long" value="8" />
      </testcase>
   </testsuite>
</testscript>
```

(Quelle: eigene Darstellung)

Die obige Abbildung zeigt ein solches über `<include>` einbindbares Skript zur Festlegung von Testfällen. Die Tags werden im nachfolgenden Abschnitt erläutert.

### 3.3.12 Tags zum Festlegen einer „Testsuite“

Unter einer Testsuite versteht man die Zusammenfassung mehrerer Testfälle unter einem Namen. Die folgenden Tags wurden eingeführt, um die Erstellung von Testfällen von der eigentlichen Durchführung der Tests zu trennen.

- Das Tag `<testsuite>` mit dem Attribut `name` legt eine Sammlung von Testfällen unter einem bestimmten Namen fest.
- Darunter werden mit dem Tag `<testcase>` die auszuführenden Schritte zur Definition eines Testfalles eingefaßt. Sie werden zum Zwecke des späteren Aufrufs intern durchnumeriert. Die unter `<testcase>` eingefaßten Tags werden an dieser Stelle nur gespeichert, die Interpretation erfolgt erst durch einen expliziten Aufruf.

  Da dieses Tag somit alle Aufgaben erledigen können muß, die nötig sind, um Objekte festzulegen, sind hier alle Tags gestattet, die auch unter `<result>`, `<compare>`, `<array>` oder unter `<member>` gestattet sind.
- Mit dem Tag `<dotestcase>` wird aus der mit dem Namen `testsuite` bezeichnete Suite ein Testfall ausgeführt. Es umschließt einen primitiven Datentypen vom Typ `long` oder `int`, welcher den Index des Testfalles angibt. Gestattet ist entweder das Tag `<primitiv>` oder das Tag `<iterationindex>`. Letzteres wird im folgenden Kapitel näher erläutert.
- Das Tag `<testsuitelength>` stellt einen primitiven Datentypen vom Typ `int` dar und gibt die Anzahl der Testfälle an, die definiert wurden. Dem Beispiel in der obigen Abbildung folgend, wäre `<testsuitelength>` gleichbedeutend mit der Angabe `<primitiv type="int" value="3">`. Dieses Tag ist an all den Stellen gestattet, an denen auch das Tag `<primitiv>` gestattet ist. Es kann somit auch als Parameter für einen Methodenaufruf fungieren. Mit dem Parameter `of` wird der Name der Testsuite angegeben, auf den sich das Tag beziehen soll.

In der obigen Abbildung werden beispielhaft Testfälle entsprechend dem ggT-Beispiel definiert. Diese Definition der Testfälle kann leider nicht in der Art erfolgen, daß nur die zu verwendenden Zahlenpaare angegeben werden. Es ist vielmehr notwen

dig, wie bereits bei der objektorientierten Variante des ggT-Beispieles angedeutet, daß die entsprechenden Klassen erzeugen werden.

Diese Instanziierung wird durch die zwischen dem Tag `<testcase>` eingefaßten Tags auch festgelegt, wobei immer der gleiche Name für die Instanz der Objekte verwendet wird. Dadurch kann der Aufruf der `getGGT()`-Methode immer an die selbe Instanz gerichtet werden, und auch das Ergebnis wird grundsätzlich mit dem Primitive „result“ verglichen, welcher ebenfalls pro Testfall entsprechend angelegt bzw. überschrieben wird.

In der DTD sind die angesprochenen Tags somit folgendermaßen definiert:

```
<!ELEMENT testsuite (testcase+)>
   <!ATTLIST testsuite name CDATA #REQUIRED>

<!ELEMENT dotestcase (primitiv | iterationindex)>
   <!ATTLIST dotestcase suitename CDATA #REQUIRED>

<!ELEMENT testsuitelength EMPTY>
   <!ATTLIST testsuitelength of CDATA #REQUIRED>

<!ELEMENT testcase (result | call | instance | class | primitiv |
                    testsuitelength | iterationindex |
                    member | array)+>
```

### 3.3.13 Tags zur Definition einer Iteration

Um die definierten Testfälle einer Testsuite iterierend ablaufen zu können, müssen Tags eingeführt werden, welche die Iteration einleiten, einen Start- und Endpunkt im Testskript festlegen und eine Abbruchbedingung definieren.

Für die Realisation im Skript, wurden die Tags `<iteration>`, `<iterationindex>`, `<length>` und `<iterate>` eingeführt., wobei durch das Tag `<iterationindex>` die Laufvariable abgebildet wird, die Abbruchbedingung sich durch eine Längenangabe mit dem Tag `<length>` definiert und der Iterationsblock durch das Tag `<iterate>` festgelegt wird.

In der folgenden Abbildung ist ein Beispiel für eine Iteration angeführt, welches alle Testfälle der Testsuite „TestCase 1“ abarbeitet.

**Abbildung 13: Beispiel einer Iteration**

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE testscript SYSTEM "d:/XML/TestSkript.DTD">

<testscript>
   <iteration name="iteration">
      <length>
         <testsuitelength of="TestCase 1" />
      </length>

      <iterate>
         <dotestcase suitename="TestCase 1">
            <iterationindex of="iteration" />
         </dotestcase>

         <result name="vergleich">
            <call instancename="myNumber1" name="getGGT">
               <parameters>
                  <instance name="myNumber2" />
               </parameters>
            </call>
         </result>

         <compare>
            <primitiv name="result" />
            <result name="vergleich" />
         </compare>
      </iterate>
   </iteration>
</testscript>
```

(Quelle: eigene Darstellung)

Das Pendant der in der Abbildung dargestellten Schleife wäre in Java beispielsweise der folgenden Form:

```
for (int x=0; x<testsuitelength(„TestSuite 1“); x++)
{
    doTestcase(x);
    ...
}
```

Eine durch das Tag `<iteration>` eingeleitete Schleife beginnt grundsätzlich bei Null. Da sie grundsätzlich nur zum Zwecke der Iteration von Testsuites implementiert wurde, ist das prinzipiell kein Nachteil. Sollte sich während des Einsatzes des Skriptes zeigen, daß auch andere Schleifentypen benötigt werden, die beispielsweise das Resultat eines Vergleiches als Abbruchsbedingung nutzen oder ähnlich flexibel gestaltet werden

können, wie die for-Schleife in Java, so können diese leicht und schnell nachträglich implementiert werden.

Im einzelnen ist die Funktion die für eine Iteration eingeführten Tags wie folgt festgelegt:

- Das Tag `<iteration>` mit dem Attribut `name` legt die Iteration fest. Durch die Namensvergabe ist es möglich, Schleifen zu verschachteln.
- Mit dem Tag `<length>` wird die Anzahl der Iterationsschritte festgelegt. Es umschließt dazu ein Tag, das repräsentativ für einen primitiven Datentyp des Typs `long` oder `int` steht. Zugelassen sind dazu nur die beiden Tags `<primitiv>` oder `<testsuitelength>`.
- Die mit dem Tag `<iterate>` umschlossenen Tags werden so oft wiederholt, wie mit dem Tag `<length>` angegeben.
- Das Tag `<iterationindex>` stellt einen primitiven Datentypen vom Typ `int` dar und gibt die aktuelle Iteration an. Dieses Tag ist an all den Stellen gestattet, an denen auch das Tag `<primitiv>` oder das Tag `<testsuitelength>` gestattet ist. Es kann somit auch als Parameter für einen Methodenaufruf fungieren. Mit dem Parameter `of` wird der Name der Iteration angegeben, auf die sich das Tag beziehen soll.

In der DTD ergibt sich somit die folgende Beschreibung der Tags:

```
<!ELEMENT iteration   (length, iterate)>
    <!ATTLIST iteration name CDATA #REQUIRED>

<!ELEMENT length (primitiv | testsuitelength)>

<!ELEMENT iterate (class | call | result | compare | array | member |
                   include | iteration | dotestcase)+>

<!ELEMENT iterationindex EMPTY>
   <!ATTLIST iterationindex of CDATA #REQUIRED>
```

## *3.4 Abbildung von primitiven Datentypen*

Um die Abbildung der Datentypen des Tags `<primitiv>` für das zu entwickelnde Tool zu vereinfachen, soll eine Klasse „`BasicDataType`“ entwickelt werden. Die Aufgabe dieser Klasse besteht darin, primitive Datentypen abzubilden. Dabei soll die Definition von „primitiven Datentypen“ über das hinausgehen, was in Java allgemein als „primitive Datentypen“ bezeichnet wird.

**Abbildung 14: primitive Datentypen in Java**

| Typname | Länge | Wertebereich | Standardwert |
|---|---|---|---|
| boolean | 1 | `true, false` | `false` |
| char | 2 | Alle Unicode-Zeichen | \u0000 |
| byte | 1 | $-2^7 ... 2^7-1$ | 0 |
| short | 2 | $-2^{15} ... 2^{15}-1$ | 0 |
| int | 4 | $-2^{31} ... 2^{31}-1$ | 0 |
| long | 8 | $-2^{63} ... 2^{63}-1$ | 0 |
| float | 4 | $+/-3.40282347*10^{38}$ | 0.0 |
| double | 8 | $+/-1.79769313486231570*10^{308}$ | 0.0 |

(Quelle: Krüger, G, 2001, S. 77)

Die Klasse „`BasicDataType`“ soll Datentypen abbilden können, die entweder die atomaren Typen von Java sind oder aber Objekte darstellen, die nur einen Wert abbilden, dessen eigener Datentyp eine Repräsentation des Objektes selbst ist, sog. Wrapper-Klassen. Beispielsweise wird ein Objekt des Typs „`java.lang.Integer`“ mit einem Wert initialisiert, der selbst vom Typ „`int`“ ist. Zusätzlich sind die Konstruktoren dieser Klassen grundsätzlich in der Lage, einen String als Parameter auszuwerten.

Die Klasse „`BasicDataType`“ soll ferner in der Lage sein, den durch sie abgebildeten Datentyp inklusive Wert als Objekt wieder auszugeben. Dies ist nötig, um mit einem `BasicDataType` via Reflection[29] entsprechende Methoden aufzurufen, die mit diesem Wert getestet werden sollen.

[29] s. Krüger, G., 2001, Kapitel 43, S. 979ff.

Ferner werden die Rückgabewerte von getesteten Methoden durch diese Klasse dargestellt, sofern es sich um einen passenden Typ handelt. Andernfalls wird die Klasse des Rückgabetyps selbst verwendet.

## *3.5 Abbildung von Aufzählungstypen und rekursiven Datenstrukturen*

Die bisher erläuterten Tags und Techniken gehen von recht einfachen Objekten oder primitiven Datentypen aus. Dies sind jedoch nicht die einzigen Datentypen, die in Programmen Verwendung finden. Neben diesen einfachen Objekten, die z.T. nur Wrapper für atomare Datentypen darstellen, gibt es auch noch Aufzählungstypen. Diese sind Arrays und Objekte der Klassen „`java.util.Vector`“, „`java.util.Collection`“, „`java.util.Hashtable`“, und viele weitere. Da Arrays in Java eine Sonderstellung einnehmen, ist ihnen ein eigener Unterpunkt gewidmet.

Zusätzlich zu den Aufzählungstypen können auch Klassen aus Kombinationen dieser Typen auftreten. Beispielsweise könnte es eine Klasse geben, die einen `Vector` von `Vectoren` von Strings und einen zweiten `Vector` von Strings umschließt, wie es für eine Klasse „Tabelle“ üblich wäre, welche die Spaltenüberschriften und die einzelnen Zeilen einer Tabelle speichern soll.

Grundsätzlich wäre ein Objekt der Klasse „Tabelle“ trotz seiner Komplexität weiterhin auch eben *nur* ein Objekt und würde damit in das bisherige Konzept sehr gut hineinpassen. Probleme sind nur an zwei Stellen zu vermuten.

- Der Vergleich zweier Objekte des Typs „Tabelle“ dürfte sich als recht kompliziert herausstellen, worauf jedoch erst in einem späteren Punkt näher eingegangen werden soll.
- Die Erzeugung eines solchen Objektes zu Vergleichszwecken wird sehr aufwendig werden.

Auf den zweiten Punkt soll hier näher eingegangen werden. Es gibt zwei grundsätzliche Möglichkeiten, ein Objekt der Klasse „Tabelle“, bzw. Objekte von Aufzählungstypen oder von rekursiven Datenstrukturen, zu erzeugen. Zum einen ist es mög

lich, solche Objekte in dem Skript selbst zu generieren. Da es sich nur um Objekte handelt, sind alle dazu notwendigen Tags bereits vorhanden.

Es ist jedoch zu vermuten, daß die im Skript abgebildete Instanziierung sehr unübersichtlich und komplex werden würde. Um dies zu verdeutlichen, zeigt die folgende Abbildung, welcher Java-Code allein notwendig wäre, um ein Objekt der fiktiven Klasse „Tabelle“ mit der bereits beschriebenen Struktur zu erzeugen. Es sollen zwei Zeilen mit jeweils drei Spalten und die Spaltenüberschriften erzeugt werden, wobei jede Zeile nicht nur allein ein `Vector` ist, sondern ein Objekt der Klasse „`Row`“.

**Abbildung 15: Java-Code zur Generierung eines Objektes der Klasse "Tabelle"**

```
java.util.Vector aRow1 = new java.util.Vector();
aRow1.add("Feld 1.1");
aRow1.add("Feld 1.2");
aRow1.add("Feld 1.3");

tabellen.Row Row1 = new tabellen.Row();
Row1.aRow = aRow1;

java.util.Vector aRow2 = new java.util.Vector();
aRow2.add("Feld 2.1");
aRow2.add("Feld 2.2");
aRow2.add("Feld 2.3");

tabellen.Row Row2 = new tabellen.Row();
Row2.aRow = aRow2;

java.util.Vector allRows = new java.util.Vector();
allRows.add(Row1);
allRows.add(Row2);

java.util.Vector aHeadline = new java.util.Vector();
aHeadline.add("Spalte 1");
aHeadline.add("Spalte 2");
aHeadline.add("Spalte 3");

tabellen.Tabelle aTable = new tabellen.Tabelle();
aTable.theHeadline = aHeadline;
aTable.theRows = allRows;
```

(Quelle: eigene Darstellung)

Würde man versuchen, diesen Java-Code mit der Hilfe des Skripts abzubilden, würde es nicht nur sehr groß, sondern auch unübersichtlich werden. Für die Erzeugung dieses

Objektes der Klasse „Tabelle“ sind allein vier Instanzen der Klasse „java.util.Vector“ nötig.

In Analogie dürfte die Generierung eines konkreten Objektes aus einer Klasse zur Abbildung einer rekursiven Datenstruktur, wie z.B. eines binären Baumes, die Komplexität dieses Beispiels noch um Längen schlagen. Berücksichtigt man nun noch die Tatsache, daß zu Vergleichszwecken erzeugte Objekte i.a. wesentlich mehr Daten enthalten dürften, als die hier präsentierten zwei Zeilen, da sie meist das Resultat einer Datenbankabfrage sein dürften, ist verständlich, daß die Erzeugung dieser komplexen konkreten Objekte möglichst nicht im Skript dargestellt werden sollte.

Es hat sich als wesentlich praktikabler herausgestellt, solche Instanziierungen in statischen Methoden in einem beispielsweise als Testklasse bezeichneten Treibermoduls unterzubringen. Der Aufruf der Methoden kann sehr leicht mit den vorhandenen Techniken des Skriptes durchgeführt werden, und die so generierten Objekte würden dann in einem Result-Tag eingeschlossen für spätere Vergleiche zur Verfügung stehen.

Diese Technik hat selbstverständlich den entscheidenden Nachteil, daß erneut Code geschrieben werden muß, der somit wiederum Fehler enthalten kann. Die mit dem Code erzeugten Objekte könnten in sich fehlerhafte Daten enthalten, was sie für einen Vergleich untauglich macht, oder aber im schlimmsten Fall auf den selben Techniken beruhen, die gerade getestet werden sollen.

Zum Verständnis dieser Problematik sei das folgende Szenario angeführt. Es soll eine Methode getestet werden, die Datensätze aus einer Datenbank ausliest und in einem Tabellenobjekt zurück liefert. Es sei nun unterstellt, daß diese Methode dazu eine bereits existierende Servicemethode in der Datenbankschicht der Anwendung verwendet. Um das Ergebnis der Datenbankabfrage zu überprüfen, soll ein eigenes Tabellenobjekt als Referenz dienen, welches durch eine Methode in einem Treibermodul erzeugt wird. Verwendet nun der Entwickler der Einfachheit halber ebenfalls die Methode aus der Datenbankschicht oder programmiert das Verhalten der zu testenden Methode unbewußt nach, wird das erzeugte Objekt immer gleich sein. Die Qualität dieses Tests ist damit stark anzuzweifeln.

## *3.6 Abbildung von Arrays*

Neben den primitiven Datentypen nehmen Arrays in Java ebenfalls eine Sonderstellung ein. Im Gegensatz zu den primitiven Datentypen sind Arrays zwar grundsätzlich Objekte und fügen sich damit in das bisherige Konzept ein, leider sind sie keine „schönen“ Objekte, wie z.B. Objekte der Klassen „java.util.Vector“[30].

Die Konkretisierung eines Arrays kann nicht, wie bei anderen Objekten üblich, über den Konstruktor, eine Setter-Methode oder den mehrfachen Aufruf einer Methode zum Hinzufügen von Objekten realisiert werden. Vielmehr ist entweder eine Schleife mit direkten Zuweisungen zu programmieren oder aber eine Definition mittels der Aufzählung von Literalen nötig, wie in der folgenden Abbildung dargestellt.

**Abbildung 16: Beispiele der Array-Definition**

```
int [] a = new int [5];
for (int i=0; i<5; i++) a[i]=i;

int [] b = {5,4,3,2,1};
```

(Quelle: eigene Darstellung)

Damit gibt es in dem bisher vorgestellten Skript keine Möglichkeit, ein statisches Array zu Vergleichszwecken zu initialisieren. Grundsätzlich ist das aber nicht notwendig und gerade bei großen Arrays auch nicht ratsam. Schließlich kann zu Definitionszwecken der gleiche Weg beschritten werden, wie bei den vorher diskutierten komplexeren Datenstrukturen auch, nämlich die Definition des Arrays von einer in einem Treibermodul formulierten Methode erledigen zu lassen. Es gilt dabei jedoch immer zu bedenken, daß diese Praxis die bereits beschriebenen möglichen Fehlerquellen hat.

Will man die Initialisierung von Arrays durch ein Skripttag ermöglichen, so muß man sich im Vorfeld etwas mit den Eigenarten von Arrays und ihrer Abbildung in Java auseinandersetzen. Arrays sind in Java, wie in anderen Programmiersprachen auch, eine Reihung von Elementen eines festen Grundtyps, also z.B. ein Array von Integer-Objekten. Da Arrays Objekte sind, werden sie auch mit dem `new`-Operator instanziiert.

---

[30] s. Krüger, G., 2001, S. 82 ff.

Ferner sind Arrays semidynamisch, d.h. ihre Größe wird zwar meistens erst zur Laufzeit bestimmt, kann dann aber nicht mehr geändert werden. Die Abbildung mehrdimensionaler Arrays wird als Array von Arrays realisiert.

Dennoch sind Arrays keine echten Objekte, sondern werden lediglich von der Java Virtuellen Maschine (JVM) als solche dargestellt. Arrays haben daher keine Methoden und nur ein von der JVM generiertes Feld mit Namen `length`, das zur späteren Längenbestimmung verwendet werden kann. Aus diesen Gründen ist es nicht so ohne weiteres möglich, ein Array mit einem beliebigen Grundtyp dynamisch zu erzeugen.

Die folgende Abbildung stellt dar, wie die Beschreibung eines Arrays im Skript aussehen könnte:

**Abbildung 17: Definition mehrerer Arrays im XML-Skript**

```
<array name="array_4Elemente">
   <primitiv type="int" value="1" />
   <primitiv type="int" value="2" />
   <primitiv type="int" value="3" />
   <primitiv type="int" value="4" />
</array>

<array name="array_3Elemente">
   <primitiv type="int" value="5" />
   <primitiv type="int" value="6" />
   <primitiv type="int" value="7" />
</array>

<array name="array_2ArrayElemente">
   <array name="array_4Elemente" />
   <array name="array_3Elemente" />
</array>
```

(Quelle: eigene Darstellung)

Um nun durch den Skript-Interpreter ein leeres Array zu erzeugen, ist die Klasse „`java.lang.reflect.Array`“ und daraus die statische Methode „`newInstance`“ zu bemühen. Genaueres hierzu findet sich im Exkurs über das Reflecion-API von Java.

Bevor das Array jedoch erzeugt werden kann, ist es notwendig, seine Struktur genau zu kennen. Der Grundtyp des Arrays soll aus den verwendeten Elementen resultieren.

Wird hier gemischt, muß das zu einem Fehler führen. Ebenso muß sich die Länge des Arrays erst aus dem Kontext ergeben.

Bei der Deklaration eines mehrdimensionalen Arrays ist es jedoch nicht notwendig, die Längen der einzelnen Dimensionen zu kennen. Arrays sind grundsätzlich eindimensional und haben somit nur eine Länge. Die Mehrdimensionalität eines Arrays ergibt sich dadurch, daß die einzelnen Elemente wiederum eindimensionale Arrays sein dürfen. Daraus folgt in Konsequenz, daß mehrdimensionale Arrays nicht zwingend Rechteckig sein müssen, sondern die einzelnen Zeilen eines beispielsweise zweidimensionalen Arrays sehr wohl von ihrer Länge variieren können. Der obige Auszug aus dem XML-Skript deutet diese Möglichkeit an.

## *3.7 Technik des Vergleichs zweier Objekte*

Der Vergleich zweier Objekte stellt die Kernaufgabe eines Werkzeugs für funktionale Tests dar, schließlich wird es die Hauptaufgabe eines Tests sein, in eine Black-Box etwas hinein zu stecken und das Ergebnis auf Korrektheit zu überprüfen. Diese Prüfung kann von einem automatisierten Werkzeug nur durch den Vergleich des erhaltenen Objekts mit dem erwarteten Objekt realisiert werden. Nun stellt sich der Vergleich zweier beliebiger Objekte nicht als trivial dar, so daß an dieser Stelle näher darauf eingegangen werden soll. Bevor allerdings ein Verfahren zum Vergleich zweier beliebiger Objekte vorgestellt werden kann, sollte definiert sein, wann zwei Objekte als gleich anzusehen sind.

Bei den atomaren Datentypen in Java ist die Definition der Gleichheit relativ leicht, stellen sie doch allesamt, bis auf den Typ `boolean`, Zahlen dar. Damit kann ihre Gleichheit ganz im Sinne der Mathematik definiert werden. Zwei primitive Datentypen sind somit gleich, wenn ihr Wert gleich ist, oder anders ausgedrückt, wenn die Differenz der beiden Werte gleich Null ist. Daraus folgt, daß sie ungleich sind, wenn ihr Wert ungleich ist, bzw. wenn die Differenz der beiden Wert verschieden von Null ist.

Bei Objekten ist die Definition von Gleichheit dagegen schwieriger. Da Objekte im einfachsten Fall eine Sammlung von atomaren Datentypen darstellen, ist die Gleichheit zweier Objekte als die Gleichheit aller Klassen- und Instanzvariablen zu definieren. Im Gegensatz zu den atomaren Datentypen können Objekte jedoch nicht nur gleich, son

dern auch identisch sein, da sie im Gegensatz zu den atomaren Datentypen nicht nur einen inhaltlichen Zustand haben, der zu vergleichen ist, sondern eine Referenz darstellen, die ebenfalls gleich sein kann. Ist die Referenz zweier Objekte gleich, also zeigen sie auf den selben Speicherbereich, so sind sie nicht nur gleich, sondern identisch. Im Gegenzug gilt, daß identische Objekte auch automatisch gleich sind. Die Identität zweier Objekte ist ein stärkeres Kriterium, als die Gleichheit.

Für ein Testwerkzeug ist jedoch die Identität zweier Objekte als schlecht zu werten. Schließlich soll es die Aufgabe des Werkzeuges sein, einen echten Soll/Ist-Vergleich durchzuführen. Sollte jedoch das erwartete Ergebnis als Soll-Vorgabe mit dem erhaltenen Objekt identisch sein, also auf den gleichen Speicherbereich zeigen, so ist zu unterstellen, daß die Generierung des Vergleichsobjektes mit einer schlechten und unbrauchbaren Technik vorgenommen wurde. Beispielsweise könnte es Aufgabe des Tests gewesen sein, eine Datenbankabfrage zu untersuchen. Wird nun sowohl zur Generierung des Vergleichsobjektes, wie auch durch die zu untersuchende Methode, ein einheitliches Framework benutzt, welches über Caching-Strategien die Datenbankzugriffe beschleunigt, so kann es passieren, daß bei dem eigentlichen Methodenaufruf erneut die Referenz auf das gecachte Objekt zurückgeliefert wird. Das Ergebnis des Vergleichs der beiden Objekte ist damit nicht nur vorhersehbar, sondern auch nicht hilfreich.

Ferner gilt, daß nur zwei Objekte des gleichen Typs gleich sein können. Sind die zwei zu vergleichenden Objekte bereits von unterschiedlichen Typen, sind sie als ungleich anzusehen. Das Ableitungssystem einer objektorientierten Sprache könnte jedoch eine Ausnahme dieser Regel ergeben. Handelt es sich bei den beiden zu vergleichenden Typen um Klassen in der gleichen Ableitungshierarchie, so könnte der Vergleich der beiden Objekte auf Basis des kleineren von beiden Typen erfolgen. Diese Definition ist aus zwei Gründen nicht unproblematisch.

Zum einen würde durch den Typecast auf eine der Superklassen eines Objektes die Anzahl der Instanz- und Klassenvariablen reduziert, so daß das gecastete Objekt nur noch eine Teilmenge des ursprünglichen Objektes darstellt. Somit kann nicht mehr von Gleichheit, sondern nur noch von Ähnlichkeit gesprochen werden. Zum anderen erweitern in Java alle Klassen die Klasse `java.lang.Object`, so daß sich jedes zu vergleichende Objekt darauf reduzieren läßt. Damit wären sich alle Objekte ähnlich, was sogar eine korrekte Aussage ist, jedoch bei einem Test keine Hilfe darstellt. Schließlich

könnte so nur noch auf gleich oder ähnlich geprüft werden, nicht jedoch auf ungleich. Es bleibt daher als Fazit festzuhalten, daß nur der Vergleich zweier vom Typ identischer Objekte sinnvoll ist und es auch in der Objektorientierung keine Ausnahme von dieser Regel gibt.

Somit muß eine in das Testwerkzeug zu implementierende Vergleichsmethode mit primitiven Datentypen, mit Arrays und mit beliebigen Objekten umgehen können und zudem in der Lage sein, die Typen zweier Objekte zu vergleichen. Bei einer ersten Betrachtung fällt auf, daß es sich grundsätzlich um zwei beliebige Objekte handeln wird, die verglichen werden müssen, da das Testwerkzeug intern stets dafür Sorge tragen wird, daß die primitiven Datentypen von Java, die ja keine Objekte darstellen, intern nur in den Wrapper-Klassen gehalten werden, um möglichst wenige Spezialfälle berücksichtigen zu müssen. Für die Arrays gilt dies ebenfalls, da sie von sich aus bereits Objekte darstellen.

Ein Blick in die bereits implementierten Methoden der Klasse `java.lang.Object` zeigt, daß es dort eine `equals()`-Methode gibt, die zwei Objekte vergleichen kann. Da jede weitere Klasse in Java gezwungen ist, die Klasse `java.lang.Object` zu erweitern, steht diese Methode somit in jeder beliebigen anderen Klasse zur Verfügung. Es wird vorkommen, daß die zu vergleichenden Instanzen ihre eigene `equals()`-Methode mitbringen. Diese sollte jedoch vor dem zu Testzwecken durchgeführten Vergleich von Objekten dieses Typs getestet worden sein, da sie von dem Algorithmus der Vergleichsmethode des Testwerkzeuges verwendet wird. Im schlimmsten Fall liefert sie immer `true` zurück, wodurch jegliche Vergleiche mit dieser Methode immer sehr positiv wirken würden.

Untersucht man nun die Fähigkeiten der „`equals()`“-Methode in `java.lang.Object`, so stellt man fest, daß sie jedoch nur bei den einfachen Datentypen einen Vergleich durchführen kann. Komplexere Objekte kann diese Methode nicht mehr vergleichen. Dort beschränkt sich der Vergleich auf den Test auf Identität, der immer an erster Stelle steht. Wenn die `equals()`-Methode nun `true` liefert, liegt entweder Identität oder Gleichheit vor und der Vergleich ist erfolgreich. Wird jedoch `false` zurückgeliefert, kann nur mit Gewißheit die Identität ausgeschlossen werden. Ob jedoch die Objekte möglicherweise doch gleich sind, läßt sich damit nicht feststellen.

Der nächste Schritt muß somit eine Fallunterscheidung sein. Es ist nun zu prüfen, ob es sich möglicherweise um atomare Datentypen oder um deren Wrapper-Klassen handelt. In dem Fall ist der Vergleich negativ ausgefallen und bereits abgeschlossen.

Handelt es sich um Arrays, so müssen nun die einzelnen Elemente der beiden Arrays ermittelt und nach dem gleichen Prinzip verglichen werden. Dabei ist es sinnvoll, vor einem Elementtest erst die Grenzen zu vergleichen. Sind die Längen der beiden Arrays unterschiedlich, ist ebenfalls bereits Ungleichheit gezeigt. Auch ist festzuhalten, daß der Basistyp der beiden Arrays gleich sein muß, da es bei Arrays nicht genügen kann, als Typgleichheit bereits das Vorliegen zweier Arrays zu definieren. Ferner ist festzulegen, daß zwei Arrays nur dann gleich sind, wenn die gespeicherten Objekte auch in der gleichen Reihenfolge auftreten. Es genügt hier für den Gleichheitsgedanken nicht, daß alle Objekte mindestens vorkommen und gleich sein müssen.

In jedem anderen Fall liegt ein komplexes Objekt vor. In dem Fall müssen nun die gesamten Klassen- und Instanzvariablen der beiden Objekte nach dem hier vorgestellten Prinzip verglichen werden.

Sowohl im Fall der Arrays, wie auch bei den komplexeren Objekten wird deutlich, daß diese gesamte Prozedur rekursiv ablaufen muß. Um jedoch wirklich jedes beliebige Objekt nach dem hier vorgestellten Verfahren vergleichen zu können, sollte dabei berücksichtigt werden, daß es Objekte gibt, die ihrerseits wieder auf bereits getestete Objektpaare verweisen. Beispielsweise könnte es nötig sein, eine doppelt verkettete Liste zu vergleichen, indem man die beiden Root-Elemente angibt. Diese enthalten eine Instanzvariable für das Folgeobjekt und eine für das Vorgängerobjekt. Mit der Technik, jede Instanzvariable erneut in den Vergleichsalgorithmus zu stecken, würde in diesem Fall eine endlose Rekursion oder Schleife auftreten. Um dies zu vermeiden, ist es daher sinnvoll, sich die bereits überprüften Kombinationen zu merken, um sie nicht erneut zu vergleichen..

Unberührt von dem Vergleich sind die in den Objekten enthaltenen Methoden, da diese für den Zustand eines Objektes völlig unerheblich sind und zudem durch die vorgeschriebene Klassengleichheit auch gleich sein werden.

Ein weiteres Problem dagegen wird die Verwendung von Klassen- und Instanzvariablen sein, die als `private` oder `protected`, nicht jedoch als `public` gekennzeichnet sind. Auf dieses Problem wurde bereits im Rahmen der Erläuterungen zu den Unter

schieden zwischen den prozeduralen und den objektorientierten Sprachen eingegangen, doch in diesem Zusammenhang soll noch mal darauf hingewiesen werden, da das Problem nun wesentlich plastischer wird. Da der Entwickler nicht gezwungen werden kann, seine Variablen alle `public` zu deklarieren, muß eine Möglichkeit geschaffen werden, dennoch die Variableninhalte auslesen zu können.

Eine Möglichkeit hierfür wäre es, den Entwickler zu zwingen, Standardmethoden für den Vergleichsfall zu implementieren. Es könnte zu diesem Zweck ein Interface existieren, welches die Klassen, die durch ein automatisiertes Testtools vergleichbar sein sollen, implementieren müßten, wodurch festgelegte Methoden für den Zugriff auf die Variablen zu definieren wären.

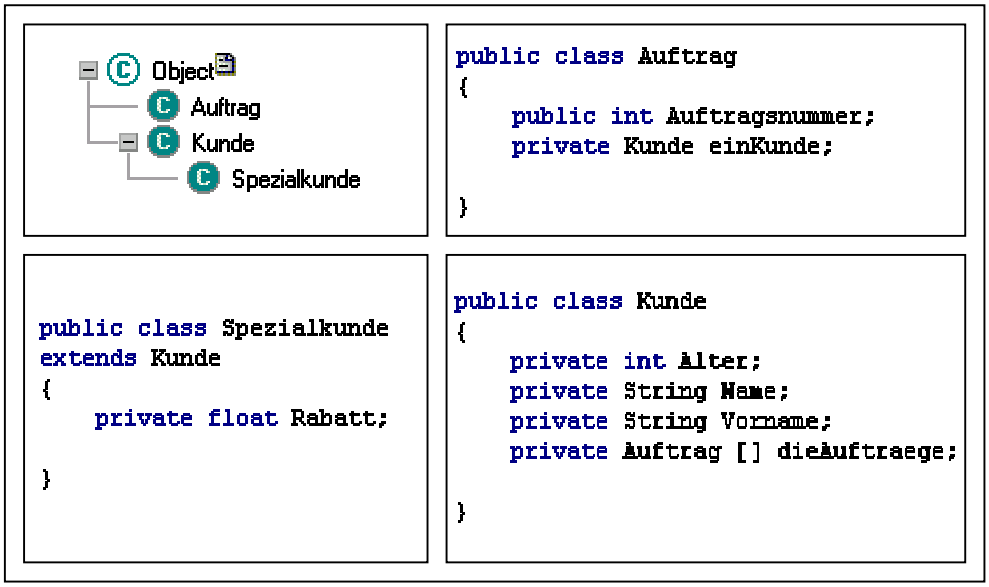
Dieses Vorgehen ist jedoch kaum praktikabel und mit Java auch nicht notwendig. Es ist in Java vielmehr möglich, mit der Hilfe des Reflection-API alle Variablen einer Klasse auszulesen. Dabei kann die komplette Struktur einer Klasse ermittelt werden, also sowohl welche Methoden implementiert sind, welche Klassen- und Instanzvariablen existieren und wie die jeweiligen Deklarationen vorgenommen wurden. Ferner ist es mit der Hilfe des Reflection-API möglich, die Inhalte von jeglichen Variablen auszulesen oder Methoden aufzurufen. Es ist dabei egal, ob die Methoden oder Variablen `private`, `public` oder `protected` deklariert wurden.

Um dem Leser die hier vorgestellte Theorie und vor allem die gezeigten Probleme etwas näher zu bringen, sei an dieser Stelle ein komplexeres Beispiel angeführt. Dazu soll eine Klasse „Kunde“ deklariert werden, die den Namen, den Vornamen und das Alter des Kunden speichern soll. Davon abgeleitet sei die Klasse „Spezialkunde“, die zusätzlich noch ein Feld „Rabatt“ enthält. Ferner soll die Kundenklasse eine Referenz auf Elemente der Klasse „Auftrag“ enthalten. Die Klasse Auftrag soll nur aus einer Auftragsnummer und einer Referenz zurück zu dem zugehörigen Kundenobjekt bestehen. In dem Beispiel sind fast alle Instanzvariablen `private` und nur die Auftragsnummer als `public` deklariert worden. Es gibt für diese Form der Deklaration keinen besonderen Grund. Es soll nur verdeutlicht werden, daß das Problem der privaten Klassen- oder Instanzvariablen existiert und daher gelöst werden muß.

Das Beispiel zeigt, daß ein Kunde und ein Spezialkunde, der obigen Definition folgend, nicht gleich sein können, da sich die Typen unterscheiden und ein Spezialkunde

ein zusätzliches Feld besitzt, das beim Vergleich ausgelassen werden müßte. Wohl aber kann ein Kunde einem Spezialkunden ähnlich sein.

**Abbildung 18: Ableitungsbaum und Klassen des Beispiels „Kunde“**




(Quelle: eigene Darstellung)

Sollen nun zwei Spezialkunden verglichen werden, so sind diese als gleich anzusehen, wenn sie den gleichen Namen, den gleichen Vornamen, das gleiche Alter und den gleichen Rabatt besitzen und auch die gleichen Aufträge erteilt haben. Der dazu nötige Vergleich der Felder `Name`, `Vorname`, `Alter` und `Rabatt` ist als unproblematisch anzusehen. Jedoch stellt das Array von Auftragsobjekten den ersten Spezialfall dar.

Wie bereits erläutert, sind Arrays zwar auch Objekte, müssen aber gesondert behandelt werden, da sie nicht durch das Auslesen von Klassen- oder Instanzvariablen verglichen werden können. Wie dies technisch in Java realisiert wird, soll an dieser Stelle nicht näher gezeigt werden, da im Exkurs über das Reflection-API darauf eingegangen wird.

Sind die Grenzen der beiden Arrays und auch ihr Basistyp gleich, so ist nur noch der Vergleich der einzelnen Auftragsobjekte durchzuführen. Die Vergleichslogik ist dabei völlig analog zu der bei dem Spezialkunden angewandten Strategie. Das einzige Problem stellt die rekursive Verknüpfung zurück zu dem Spezialkunden in dem Auftrags

objekt dar. Diese muß abgefangen werden, da ja der Kunde bereits verglichen wird und sich dadurch die Vergleichslogik ständig im Kreis drehen würde.

Es ist unbestreitbar, daß bei komplexen und vor allem großen Objekten der Vergleich mitunter sehr viel Zeit in Anspruch nehmen wird, da das Laufzeitverhalten in direkter Abhängigkeit zu der Komplexität der zu vergleichenden Objekte steht. Leider gibt es keine Möglichkeit, das hier vorgestellte Verfahren zu beschleunigen oder weniger komplex zu gestalten, will man sich auf das Ergebnis des Vergleichs verlassen können. Auf der anderen Seite wird die Performance des Vergleichs eine eher untergeordnete Rolle spielen, wenn man berücksichtigt, daß das Testskript mit großer Wahrscheinlichkeit meistens Nachts in einem Batch ablaufen wird.

Der Vergleichsalgorithmus ist in einer rekursiven Java-Methode implementiert worden. Ein Ausdruck der Methode befindet sich im Anhang 2. Da der Code komplett auf den hier genannten Strategien beruht, ist jedoch seine Kommentierung auf das Wesentliche reduziert worden.

## *3.8 Andere Vergleichstechniken*

Neben dem Vergleich von Objekten gibt es auch andere Strukturen, die sich einem Vergleich mit dem erwarteten Ergebnis stellen müssen. Schließlich hat nicht jede Methode Funktionscharakter und liefert ihr Ergebnis als Objekt zurück, sondern schreibt dieses z.B. in eine Datei, als Text auf den Bildschirm oder in eine Datenbank. Es ist auch denkbar, daß das Ergebnisobjekt sofort über einen beliebigen Kanal versendet wird und damit der Vergleichsmethode erst gar nicht zur Verfügung steht.

Wenden wir uns noch einmal der Beispielklasse „Tabelle“ aus dem Kapitel 3.5 zu und unterstellen wir, daß diese Klasse eine Methode implementiert, die den Inhalt der Klasse in eine ASCII-Datei schreiben kann, um daraus einen Import in das Programm „Excel“ von Microsoft zu realisieren.

Die manuelle Inspektion der Datei soll durch die Verwendung des Testwerkzeuges vermieden werden. Da das Testwerkzeug die Korrektheit der Datei nur durch den direkten Vergleich nachweisen kann, muß eine solche Funktionalität implementiert werden. Diese sollte jedoch statt eines ASCII-Vergleichs den binären Vergleich vorziehen, um unabhängig von dem verwendeten Dateiformat zu sein. Dies setzt allerdings voraus,

daß die beiden Dateien exakt gleich sein müssen. Der Aufwand zur Erstellung dieser Datei ist abhängig von dem für den Tests verwendeten Umfang der in die Tabelle einzustellenden Datensätze. Um die Vergleichsdatei kleiner und den Aufwand ihrer Erstellung gering zu halten, ist es möglich, relativ wenige Datensätze in der Tabelle einzustellen. Werden allerdings zu wenig Datensätze verwendet, ist die Qualität des Tests nicht sehr hoch.

Zusammenfassend kann gesagt werden, daß die Generierung von Vergleichsobjekten grundsätzlich nicht trivial ist. Dennoch haben die bisherigen Beispiele gezeigt, daß normalerweise eine Lösung gefunden werden kann. Auch die Überprüfung von Ausgaben auf Konsole oder Bildschirm ist dann möglich, wenn das Testwerkzeug ein Verfahren anbietet, diese Ausgaben abzufangen und für einen Vergleich bereitzustellen. Selbst wenn die zu vergleichenden Objekte an dem Testwerkzeug vorbei über das Internet versendet werden, wie es bei der Implementierung eines Webservers wahrscheinlich ist, bieten sich Möglichkeiten an, mit einem entsprechenden Treibermodul oder STUB das Objekt abzufangen, um es vergleichen zu können. Es gibt für diese Module jedoch keine Patentimplementierung, so daß sie je nach Bedarf entsprechend zu entwickeln sind.

Wesentlich umfangreicher werden die bisher verwendeten Soll/Ist-Vergleiche jedoch, wenn eine Datenbank überprüft werden muß. Dem bisherigen Prinzip folgend müßte für jeden Test eine Startdatenbank existieren, um eine wohldefinierte Testumgebung zu erhalten, und eine erwartete Ergebnisdatenbank, um das Resultat überprüfen zu können. Diese Technik macht grundsätzlich auch Sinn, wenn man bedenkt, daß beispielsweise das Löschen eines Datensatzes unangenehme Seiteneffekte haben kann, die nur auf diesem Weg gefunden werden können.

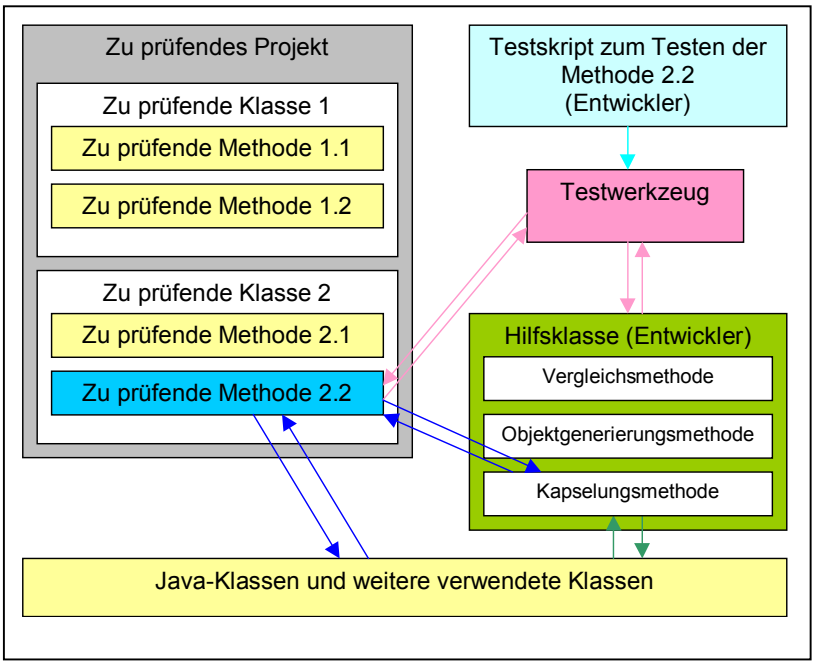
Dieses Vorgehen ist aber keineswegs praktikabel. Schließlich ist davon auszugehen, daß eine gute Testdatenbank sehr umfangreich sein wird. Diese komplett zu vergleichen wäre kaum in vernünftiger Zeit durchzuführen. Ebenso dürfte es sehr aufwendig sein, die Start- und die Vergleichsdatenbank zu erzeugen. Für solche Fälle muß es daher möglich sein, einen anderen Weg beschreiten zu können.

Es ist durchaus praktikabler, den Erfolg einer Datenbankoperation nur an den Feldern und Relationen zu überprüfen, die von dieser Operation tangiert werden. Dabei ist sehr viel Sorgfalt aufzubringen. Gerade das Löschen von Datensätzen kann Inkonsistenzen erzeugen, wenn die Datenbank keine eigene Überprüfung von referentieller Inte

grität vornimmt. Wird z.B. ein Kunde gelöscht, so muß auch überprüft werden, ob seine erteilten Aufträge ebenfalls gelöscht wurden, bzw. ob die Existenz von unfertigen Aufträgen das Löschen des ganzen Kunden verhindert hat.

Soll eine Überprüfung dieser Form von einem Testwerkzeug vorgenommen werden, gibt es mehrere Möglichkeiten, dies zu realisieren. Die einfachste Variante ist es, den Entwickler selbst die Implementierung einer Prüfroutine vornehmen zu lassen, deren Ergebnis dann mit dem `<compare>`-Tag überprüft werden kann. Eine solche Methode würde als statisch deklarierte Methode in einer Klasse Platz finden, welche auch die Methoden zur Erzeugung von komplexeren Datenstrukturen oder die evtl. notwendigen Treibermodule für die Kapselung von Methodenaufrufen aufnehmen kann.

**Abbildung 19: Kommunikationsmodell des Testwerkzeuges**




(Quelle: eigene Darstellung)

Die in der obigen Abbildung dargestellten Kommunikations- und Datenaustauschwege machen deutlich, wie komplex ein Testlauf werden kann. Vor allem der schon oft angedeutete notwendige Einsatz einer Hilfsklasse zur Durchführung von Aufgaben, die nicht oder noch nicht durch das Skript abgebildet werden können, steigert die Komplexität erheblich. Schließlich ist hier der Entwickler gefordert, zusätzlichen Code

zu implementieren, was, wie bereits angedeutet, erneut zu Fehlern führen kann. Gerade durch die Kapselungsmethoden, die helfen sollen, die Testumgebung zu definieren, ist es sogar notwendig, die Hilfsklasse bei der Auslieferung der Software mit in das Produkt zu integrieren, da sie nicht mehr davon zu trennen ist. Sollte diese Hilfsklasse aus irgendeinem unerfindlichen Grund in den Testmodus umschalten, würde das ausgelieferte Programm plötzlich nicht mehr funktionieren, es würde praktisch ein Programm ausgeliefert, daß eine mögliche Zeitbombe mit sich führt. Die durch die umgeleiteten Methoden verursachte Verschlechterung des Zeitverhaltens wird dagegen so gering sein, daß sie nicht weiter berücksichtigt werden muß.

Die bisherigen Ausführungen haben deutlich gemacht, daß der Vergleich als Strategie zur Ergebnisermittlung eines Tests oft sehr problematisch sein kann. Um das Dilemma des Vergleichs in Verbindung mit der Forderung nach einer wohldefinierten Testumgebung noch weiter zu verdeutlichen, sei noch ein letzte Beispiel angeführt.

Aus einem beliebigen Grund sei eine Methode implementiert worden, die bei jedem Aufruf eine Zufallszahl liefert, wobei die Wahrscheinlichkeit ihres Auftretens einer ganz bestimmten Verteilungsfunktion genügt. Beispielsweise sollen die Zufallszahlen in dem Bereich $0 \leq x \leq 100$ liegen und die Anzahl ihres Auftretens sei gleichverteilt, wodurch gilt, daß $p(x) = \frac{1}{100}$.

Es ist nun die Aufgabe des Testers, zu überprüfen, ob das Vorkommen der ermittelten Werte tatsächlich zu dieser Wahrscheinlichkeitsfunktion paßt. Diese Aufgabe kann mit dem bisherigen Skript nicht erfüllt werden. Schließlich macht es bei dieser Aufgabe keinen Sinn, einzelne Werte zu überprüfen. Erst nach dem Sammeln von beispielsweise 10.000 Werten kann möglicherweise eine Aussage über die gute oder schlechte Funktionsweise der Methode getroffen werden.

Das Sammeln der Werte kann durchaus durch das Skript realisiert werden. Da es aber in diesem Fall nicht möglich ist, die Werte vorauszusehen und dieser Umstand auch nicht durch die bisher besprochenen Techniken zur Definition der Testumgebung verbessert werden kann, fällt damit weiterhin dem Tester die Aufgabe zu, selbst eine Prüfmethode zu entwickeln.

In diesem Fall kann aber auch versucht werden, den verwendeten Algorithmus zur Erzeugung gleichverteilter Zufallszahlen auf mathematischem Wege zu verifizieren, also mit der Hilfe eines White-Box-Verfahrens.

## *3.9 Ausgaben und Meldungen des Testwerkzeuges*

Selbstverständlich wird das zu entwickelnde Testwerkzeug zur Durchführung funktionaler Black-Box-Tests auch Ausgaben tätigen müssen, um dem Entwickler oder Tester die Ergebnisse der Tests mitzuteilen. Zu den wichtigen Ausgaben zählen einerseits die Ausgabe der Ergebnisse von Vergleichen und andererseits Ausgaben aufgrund des Auftretens von Ausnahme- oder Ausführungsfehlern.

Im Falle der Ausnahme- und Ausführungsfehler ist jedoch zu unterscheiden, ob diese durch das Testwerkzeug selbst oder durch die zu testende Methode ausgelöst wurden. Ist der Fehler dem Testwerkzeug zuzurechnen, kann dies entweder auf einen internen Fehler hindeuten, oder das verwendete Skript ist fehlerhaft.

In jedem Fall darf bei dem Auftreten von Fehlern die weitere Ausführung des Testskriptes nicht abbrechen. Es kann jedoch sein, daß unter bestimmten Umständen sehr viele Folgefehler auftreten werden und auch einige Tests nicht durchführbar sind. Beispielsweise kann es vorkommen, daß im Testskript ein Tippfehler in der Klassenbezeichnung deren Instanziierung unmöglich macht. Damit steht diese Klasse aber nicht zur Verfügung und jegliche Verwendung der nicht existenten Referenz wird einen Fehler erzeugen.

Ferner kann das Testwerkzeug die durchgeführten Schritte und deren Erfolg protokollieren. Es dürfte auch im Interesse des Entwicklers oder Testers sein, im Nachhinein die Bearbeitung seines Skriptes verfolgen zu können. Eine sinnvolle Ausgabe aus diesem Bereich könnte z.B. sein: „Die Instanz mit dem Namen ‚xxx' der Klasse ‚yyy' über Konstruktor ‚zzz' mit Parameterliste ‚p' wurde erfolgreich angelegt.".

Die Informationsflut in einem solchen Satz ist zwar regelrecht erschlagend, aber es ist durchaus notwendig, dem Anwender des Testwerkzeuges diese Informationen zu liefern.

Werden alle diese Informationen auf einer Konsole bzw. einem Bildschirm untereinander ausgegeben, so wird der Anwender schnell den Überblick verlieren, vor allem, wenn man berücksichtigt, daß auch das zu testende Programm Ausgaben auf dem Bildschirm tätigen könnte, die sich dann mit den Ausgaben des Testwerkzeuges mischen würden. Eine vernünftige Auswertung ist dann nicht mehr möglich.

Eine bessere Möglichkeit zur Protokollierung der Ergebnisse ist das Erzeugen einer XML-Datei, die dann durch ein Programm gelesen und ausgewertet werden kann. Dieses könnte durch farbliche Unterscheidung oder andere Techniken die Struktur der Ausgaben verbessern und lesbarer gestalten oder durch das Ausblenden von Fehlertypen die Informationsflut eingrenzen. Die Implementierung dieses Tools steht jedoch noch aus. Daher erfolgen die Ausgaben in der ersten Stufe der Implementierung noch auf dem Bildschirm.

## *3.10 Technische Umsetzung des Tools*

Nachdem in den vorherigen Abschnitten die Grundlagen besprochen wurden, kann die Implementierung des Testwerkzeuges beginnen. Die hauptsächliche Arbeit des Werkzeuges stellt die Interpretation des XML-Skriptes dar. Nebenbei muß auch der Ablauf der zu testenden Klassen oder Methoden überwacht und auf Fehler reagiert werden. Ferner muß der Status aller abgearbeiteten Tags unter einem eindeutigen Namen gespeichert werden, um im späteren Verlauf des Skriptes wieder darauf zugreifen zu können

Bevor das Skript abgearbeitet werden kann, muß es selbstverständlich erst eingelesen werden. Zu diesem Zweck bedient man sich am Besten eines bereits fertig implementierten und validierenden Parsers, wie z.B. dem Produkt „XML4J“ von IBM. Dieser liest die XML-Datei unter Berücksichtigung der verwendeten DTD ein, weist auf syntaktische Fehler des Skripts hin und stellt es bei erfolgreichem Einlesen in einer Baumstruktur zur Verfügung[31].

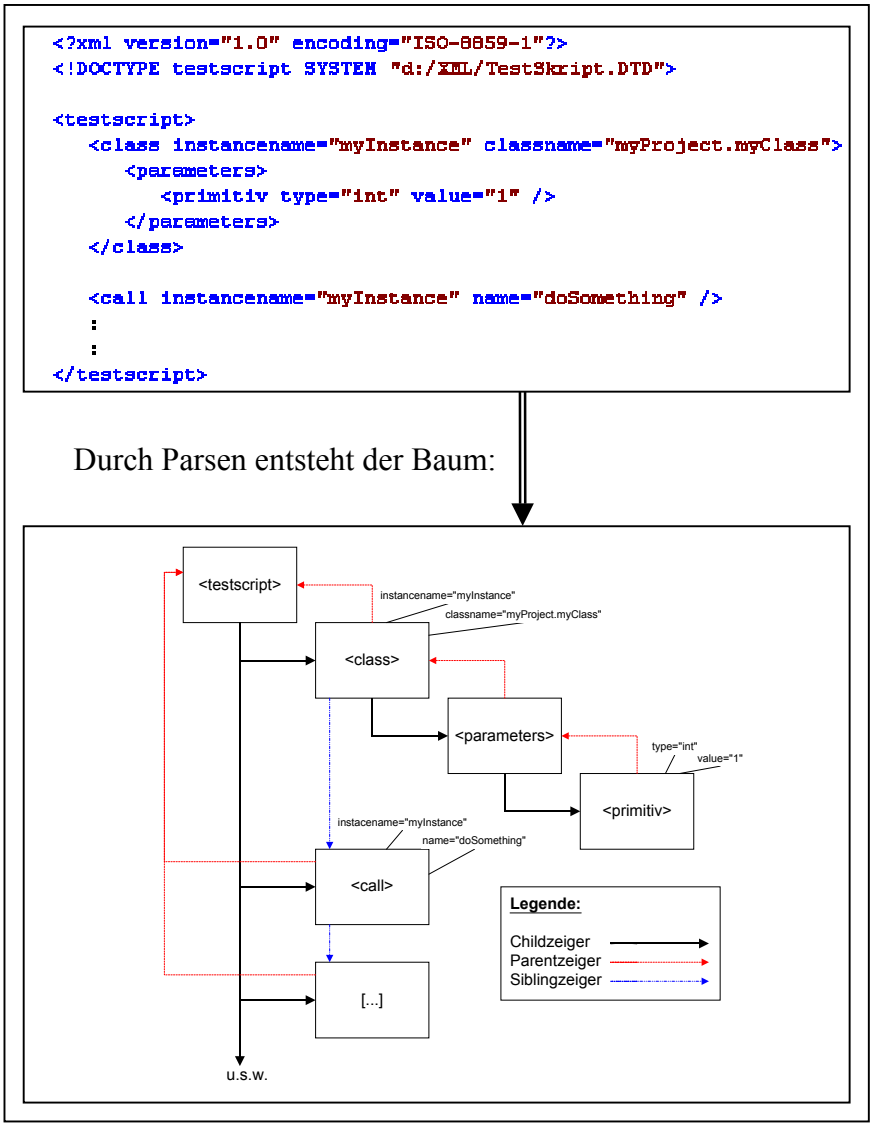
In der folgenden Abbildung ist ein Ausschnitt eines solchen Baumes dargestellt, wie er in etwa von dem XML4J erzeugt wird. Die ermittelten Attribute sind an die entsprechenden Tags angehängt worden. Jedes Kästchen stellt dabei ein Objekt des Typs `org.w3c.dom.Node` dar, welches neben dem Namen des Knoten (Node) und seinen Attributen auch einen Zeiger auf den Vaterknoten (parent), auf evtl. Kinderknoten (childs) und auf in gleicher Ebene hängender Knoten (siblings) enthält. Damit ist es von jedem Knoten aus möglich, den gesamten Baum zu erfassen.

---

[31] Parser mit dieser Technik werden als DOM-Parser bezeichnet. Daneben existieren auch die sog. SAX-Parser. Für nähere Informationen s. Glossar.

Um diesen Baum zu traversieren, ist die Implementierung einer extra Klasse sinnvoll, deren einzige Aufgabe darin besteht, das im aktuellen Knoten gespeicherte Tag abzuarbeiten.

**Abbildung 20: Modell eines aus dem XML-Skript erzeugten Baumes**

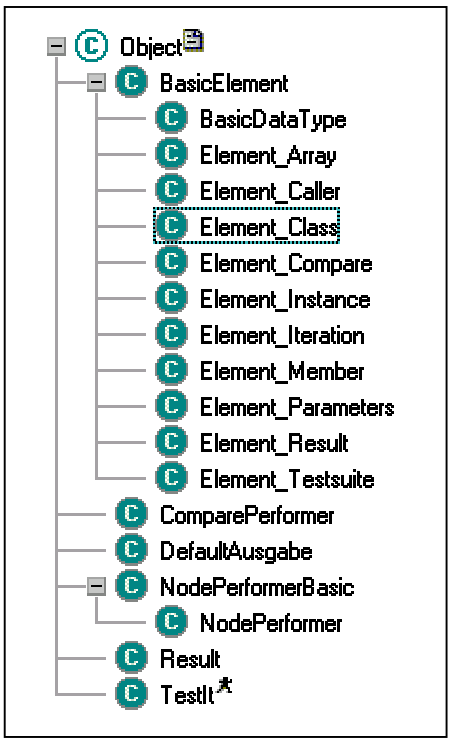



(Quelle: eigene Darstellung)

Zu diesem Zweck wird eine Klasse mit dem Namen `NodePerformer` implementiert, welche prinzipiell eine Erweiterung des Interfaces `org.w3c.dom.Node` darstellt. Um ohne die Ableitung auszukommen, wird der Einfachheit halber der aktuelle Knoten als Instanzvariable dem Konstruktor von `NodePerformer` mitgeliefert. Diese Klasse enthält

für jedes in der DTD definierte Tag eine spezialisierte Methode, welche über eine Mapping-Methode aufgerufen wird. Um Child-Tags zu verarbeiten, wird immer eine neue Instanz von `NodePerformer` angelegt, welche diese verarbeitet. Dabei kennt jede Instanz von `NodePerformer` seinen aufrufenden `NodePerformer`.

**Abbildung 21: Klassenhierarchie des gesamten Testwerkzeuges**



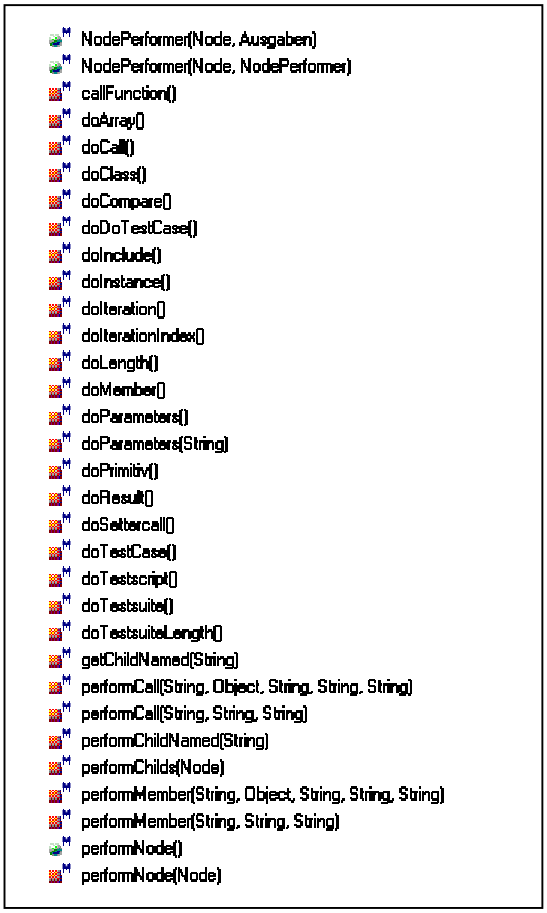

(Quelle: eigene Darstellung)

Für das Tag `<class>` beispielsweise existiert eine Methode `doClass()`. Diese liest die typischen Attribute des Tags `<class>` aus und erzeugt für die zu erwartenden Childs `<parameters>` und `<settercall>`, falls vorhanden, jeweils eine neue `NodePerformer`-Instanz für ihre Verarbeitung. Der verwendete Algorithmus des `NodePerformers` ist somit als pseudorekursiv zu bezeichnen, da zwar immer wieder die gleichen Methoden aufgerufen werden, jedoch pro zu verarbeitenden Knoten eine neue Instanz erzeugt wird.

Nach erfolgreicher Instanziierung des durch das Tag angeforderten Objektes wird ein Objekt des Typs `Element_Class` erzeugt, welches sämtliche in dem Tag `<class>` enthaltenen Informationen und das instanziierte Objekt aufnimmt.

Das Objekt des Typs `Element_Class` erhält als Schlüssel schließlich den Namen der Instanz und wird in einer globalen `java.util.Hashtable` abgelegt. Der Name, unter dem das Objekt in der Hashtable abgelegt wurde, wird von `doClass()` zurückgeliefert, damit evtl. vorhandene Vater-Tags über dem `<class>`-Tag wieder auf das erzeugte Objekt zugreifen können.

**Abbildung 22: Alle Methoden der Klasse NodePerformer**




(Quelle: eigene Darstellung)

Die globale Hashtable wird von allen Objekten verwendet, die in irgendeiner Form Daten ablegen müssen. Aus diesem Grund existiert auch für jedes dieser Tags eine spezialisierte Datenklasse, welche dann in der Hashtable abgelegt wird[32]. Ebenso gibt jede

[32] Dies sind alle Klassen, die von BasicElement abgeleitet wurden.

Methode den Namen, unter dem das Objekt in der Hashtable abgelegt wurde, an die aufrufende Methode zurück.

Wurde beispielsweise bei dem Tag `<class>` auch ein Parameterblock durch das Tag `<parameters>` angegeben, damit der entsprechende Konstruktor aufgerufen wird, so kann die Methode `doClass()` den Inhalt des Parameterblockes nach seiner Verarbeitung wieder aus der Hashtable entnehmen, da die Methode `doParameters()` den Ablagenamen an `doClass()` zurückgeliefert hatte. Das aus der Hashtable zu ermittelnde Objekt ist dann vom Typ `Element_Parameters`.

Für die Tags, die keinen expliziten Namen vom Entwickler erhalten haben, wird ein temporärer Name der Form „@_X_@“ erzeugt, wobei X eine Zahl ist, die pro erzeugtem Namen um eins erhöht wird. Beispielsweise ist es bei dem Tag `<primitiv>` dem Entwickler der Testfälle überlassen, ob er dem erzeugten primitiven Datentypen einen Namen gibt, oder nicht. Diese Entscheidung ist davon abhängig, ob das erzeugte Objekt noch einmal benötigt wird. Ferner werden die in der Hashtable abgelegten, jedoch nicht mehr benötigten Objekte entfernt, um Speicherplatz zu sparen, der bei einem sehr umfangreichen Skript durchaus knapp werden könnte.

Um Objekte zu erzeugen, Methoden aufzurufen oder Variableninhalte abzufragen bzw. zu verändern, bedient sich der Interpreter des Java-Reflection-APIs, welches im nächsten Kapitel näher erläutert wird. Mit dieser Technik ist es, wie bereits angesprochen, ohne weiteres möglich, als `private` deklarierte Methoden oder Variablen anzusprechen. Die für diese Aufgaben zuständigen Methoden sind die `performCall()`- und die `performMember()`-Methoden, die in ihrer Grundstruktur sehr ähnlich sind. Zusätzlich geben diese beiden Methoden umfangreiche Meldungen über den Erfolg oder Mißerfolg des Aufrufs einer Methode, bzw. des Ansprechens einer Klassen- oder Instanzvariablen aus.

# 4 Exkurs „Reflection-API“

Die Realisierung des Werkzeuges in Java als Interpreter ist nur durch einen konsequenten Einsatz des Reflection-APIs möglich. Diese Schnittstelle bietet eine direkte Verbindung zu der JVM an, so daß umfangreiche Manipulationen an Klassen und Methoden ebenso möglich sind, wie die Ermittlung von Informationen über das getestete Programm.

Während der Entwicklung des JDK 1.1 stellten die Entwickler fest, daß einige der zu implementierenden Funktionen mit dem derzeitigen Sprachumfang von Java nicht zu realisieren waren. Bisher war es nur möglich, auf Methoden und Klassen zuzugreifen, die schon während der Compilerzeit bekannt waren. Für viele normale Anwendungen ist das auch ausreichend, doch es wurde mehr Flexibilität benötigt, um z.B. das dynamische Nachladen eines beliebigen JDBC-Treibers[33] zu realisieren.

Normalerweise muß zur Instanziierung von Klassen oder für den Aufruf von Methoden Code geschrieben werden, der an den Interpreter oder Compiler der Zielsprache weitergegeben wird. Mit der Hilfe des Reflection-APIs ist es jedoch in Java ohne weiteres möglich, z.B. die Instanz einer Klasse zu erzeugen, obwohl der Name der Klasse nur als String vorliegt.

Diese Technik entspricht in etwa der Vorstellung, daß man einem String-Objekt, welches einen Ausdruck in der entsprechenden Zielsprache enthält, nur die Nachricht `evaluate()` schickt, damit der String wie ein Codefragment kurz compiliert, dann ausgeführt und das Ergebnis zurückgeliefert wird. Beispielsweise könnte das im Code der folgenden Form sein: „`int x = "7*5+3".evaluate();`“. Das Reflection-API verfolgt zwar eine anderes Prinzip und funktioniert daher auch anders, die Idee ist jedoch ähnlich.

Um überhaupt die Arbeit mit dem Reflection-API zu ermöglichen, gibt es in der Klasse `java.lang.Object` die Methode `getClass()`. Da in Java die Klasse `Object` sozusagen die Mutter aller Klassen ist, kann man daher auf diesem Weg alle instanziierten Objekte nach ihrem Klassentyp fragen. Das zurückgelieferte `Class`-Objekt kann

[33] das JDBC (Java Database Connection)-Interface benötigt einen vom Hersteller der Datenbank bereitgestellten Treiber, um Standardfunktionalität auf beliebige Datenbanken abbilden zu können.

dann verwendet werden, um Informationen über die Klasse abzufragen. Alternativ gibt es in Java auch das Schlüsselwort `class`, das einer Klassenvariablen gleichkommt. Damit kann von eine konkreten Klasse ein `Class`-Objekt erzeugt werden.

## *4.1 Erzeugen von Objekten*

Die Klasse Class bieten neben der Möglichkeit, Informationen auszulesen auch die statische Methode `forName()`. Diese ist der Schlüssel, um aus einem in einem String abgelegten Klassennamen ein `Class`-Objekt zu instanziieren, dem dann die Nachricht `newInstance()` geschickt werden kann. Das folgende Codefragment soll diese Technik verdeutlichen.

**Abbildung 23: Beispiel einer Klasseninstanziierung via Reflection-API**

```
// Classobjekt der Klasse String erzeugen:
Class c = Class.forName("java.lang.String");
// Instanz erzeugen:
Object o = c.newInstance();
```

(Quelle: eigene Darstellung)

Die Variable `o` enthält nun ein leeres String-Objekt, mit dem genauso gearbeitet werden kann, als wäre mit der Hilfe des `new`-Operators entstanden. Auf ähnliche weise kann ein Objekt auch durch den Aufruf eines Konstruktors erzeugt werden. Dazu muß man wissen, daß ein Konstruktor, so wie Methoden auch, durch seine Parameter unterschieden wird. Um also den richtigen Konstruktor zu erreichen, muß bekannt sein, welche Parametertypen in welcher Reihenfolge deklariert wurden.

Dazu erzeugt man ein Array von `Class`-Objekten der entsprechenden Parameter. Jedes dieser `Class`-Objekte steht repräsentativ für den entsprechenden Datentyp. Damit kann man dann das `Class`-Objekt der gewünschten Klasse nach dem entsprechenden Konstruktor fragen und erhält, falls ein solcher Konstruktor existiert, ein Objekt des Typs `java.lang.reflect.Constructor` zurück. Andernfalls wird eine Exception ausgelöst.

Dem `Constructor`-Objekt sendet man schließlich die Nachricht `newInstance()` mit den entsprechenden Parametern, was den selben Effekt hat, als würde man diese Nachricht einem `Class`-Objekt senden. Es wird dann der Konstruktor ausgeführt und die Erzeugte Instanz des neuen Objektes zurückgeliefert.

Die folgende Abbildung soll dies verdeutlichen. Es soll ein Objekt der Klasse `Number` mit dem Wert 1 erzeugt werden, wie sie für das ggT-Beispiel zum Einsatz gekommen war. Der Konstruktor der Klasse erwartet einen `long`.

**Abbildung 24: Instanziierung via Konstruktoraufruf mit Reflection-API**

```
// Class-Objekt erzeugen
Class theClass = Class.forName("ggt_test.Number");

// Parametertypen festlegen
Class [] ParameterTypes = new Class [1];
ParameterTypes[0] = long.class;

// Parameterwert festlegen (Wrapper erforderlich!)
Object [] Parameters = new Object [1];
Parameters[0] = new Long(1);

// Objekt durch Konstrutkoraufruf erzeugen
java.lang.reflect.Constructor theConstructor = theClass.getConstructor(ParameterTypes);
Object o = theConstructor.newInstance(Parameters);

// die beiden Zeilen entsprechen:
Object o = new ggt_test.Number(1);
```

(Quelle: eigene Darstellung)

Nach der Ausführung des Codes stellt das Objekt `o` eine Instanz der Klasse `ggt_test.Number` dar, deren Wert 1 ist, was völlig analog zu dem folgenden Codefragment ist: `Objekt o = new ggt_test.Number(1);`

## *4.2 Aufrufen von Methoden*

Der Aufruf von Methoden funktioniert nach dem gleichen Prinzip, wie der Aufruf des Konstruktors. Die Klasse `Class` wird in diesem Fall nach der entsprechenden Methode mit Namen X und Parametertypenliste Y gefragt. Dazu bedient man sich am besten der Methode `getDeclaredMethod()`, um auch als `private` deklarierte Methoden zu erhalten. Dabei ist darauf zu achten, sollte die Methode in einer Oberklassen deklariert sein, diese nicht gefunden wird. Es kann daher erforderlich sein, auch diese nach der Methode abzusuchen, sollte sie in der aktuellen Klasse nicht vorhanden sein,. Nach

erfolgreicher Suche erhält man eine Instanz der Klasse `java.lang.reflect.Method` zurück.

Dieser kann die Nachricht `invoke()` geschickt werden, wodurch die Methode ausgeführt wird. Dazu wird das instanziierte Objekt mitgegeben, dem die Nachricht geschickt werden soll, und die Parameterliste. Handelt es sich bei der Methode um eine statische Methode, so wird die zu übergebende Instanz ignoriert und kann `null` sein. Um auch private Methoden aufrufen zu können, muß dem `Method`-Objekt nur die Nachricht `setAccessible(true)` gesendet werden.

Das Ergebnis der Methode wird von `invoke()` in der Form eines Objektes zurückgeliefert. Sollte der Ergebnistyp der Methode ein atomarer Datentyp sein, so werden automatisch die entsprechenden Wrapper-Klassen bemüht. Um den echten Datentyp zu ermitteln, kann man sich an das `Method`-Objekt wenden, um diesen mit der Methode `getReturnType()` zu erfragen.

## *4.3 Auslesen und manipulieren von Klassen- und Instanzvariablen*

So wie Methoden ermittelt und ausgeführt werden, wird auch mit den Klassen- und Instanzvariablen verfahren. Da das Vorgehen komplett analog verläuft, wird auf eine detaillierte Erläuterung dazu verzichtet[34]. Es sei dennoch erwähnt, daß es durch das Reflection-API auch möglich ist, Variablen neu zu setzen. Durch den Aufruf der Methode `setAccessible(true)` in dem entsprechenden `java.lang.reflect.Field`-Objekt der Klassen- oder Instanzvariablen können private Variablen nicht nur ausgelesen, sondern auch verändert werden.

Bei der Verwendung des Reflection-APIs sollte nicht vergessen werden, daß es möglich ist, das gesamte Kapselungsprinzip objektorientierter Sprachen komplett zu unterwandern, was durchaus als eine negative Eigenschaft gewertet werden kann. Dennoch ist es ein mächtiges Werkzeug, um die gesamte Ausführung eines Programmes zu überwachen und zu manipulieren. Ohne dieses Interface wäre es nicht möglich gewesen, die gestellte Aufgabe zu lösen.

---

[34] vgl. Krüger, G., 2001, S.979 ff.

# 5 Testen von graphischen Oberflächen

Zu den funktionalen Tests einer Software gehören neben dem Black-Box- oder Grey-Box-Test von Methoden einer Klasse auch der Test einer graphischen Oberfläche. Diese Form der Tests sind noch nicht besprochen worden und können mit der bisherigen Implementierung des Testwerkzeuges auch nicht realisiert werden. Um auch diese Form der Tests zu ermöglichen, sind einerseits Anpassungen und Erweiterungen am Testskript und damit auch an dem Werkzeug selbst notwendig, andererseits sind auch noch bestimmte Voraussetzungen zu erfüllen, die im Folgenden erläutert werden sollen.

Das Prinzip des Oberflächentests ist jedoch ebenso komplex zu betrachten, wie die funktionalen Tests von Methoden, da es eine wahre Flut an verschiedenen Darstellungsformen gibt, die graphische Oberflächen abdecken. Daraus ergibt sich auch, daß es sehr viele Möglichkeiten für Testfälle und Testarten gibt, die soweit wie möglich automatisiert werden sollen. Da die Oberflächen durch einen automatisierten Tests ferngesteuert werden sollen, muß überprüft werden, welche Möglichkeiten zur Realisierung dieser Aufgabe existieren und welche davon umsetzbar sind.

Daher erhebt dieses Kapitel keinen Anspruch auf Vollständigkeit. Es soll kurz angesprochen werden, warum graphische Tests sinnvoll sind, wo ihre Grenzen liegen und was grundsätzlich zu beachten ist, um das Testwerkzeug für die neue Anforderung zu erweitern. Dabei sollen einige Probleme oberflächlich aufgezeigt und besprochen werden, hingegen soll gänzlich darauf verzichtet werden, näher auf Swing oder AWT[35] einzugehen, was für eine umfassende Besprechung dieser Technik unumgänglich wäre.

## *5.1 Aufgaben und Grenzen von Oberflächentests*

Eine graphische Benutzeroberfläche stellt eine Schnittstelle zwischen Anwender und Software dar, welche die programmierte Funktionalität der Software abbildet. Sie sendet die Eingaben des Benutzers an entsprechende Funktionen in der Software weiter und nimmt deren Ausgaben entgegen, um sie dem Benutzer zu präsentieren.

---

[35] AWT steht für „Abstract Windowing Toolkit“, ist der Vorgänger und die Basis von Swing

Werden die Aufgaben einer graphischen Benutzeroberfläche so definiert, ist bei der Entwicklung von Software grundsätzlich eine strikte Trennung zwischen der Businesslogik und der Benutzerinteraktion vorauszusetzen. Wird diese Trennung auch vorgenommen, so beschränkt sich der funktionale Test einer Oberfläche nur noch darauf, die korrekte Kommunikation zwischen Präsentations- und Logikschicht zu verifizieren. Bereits zur Hochzeit der prozeduralen Programmiersprachen wurde vorgeschlagen, die Benutzerschnittstelle, ob nun graphisch oder textbasiert, von dem restlichen Programm zu trennen. Es handelt sich also keineswegs um eine Technik, die erst durch die Objektorientierung ermöglicht oder gar erst mit den neuen Modellierungswerkzeugen, wie z.B. UML, eingeführt wurde.

In der Realität sieht es oft anders aus. Einerseits findet keine saubere Trennung zwischen der Dialogschicht und der Businesslogik statt, was das Auffinden funktionaler Fehler in der Logik unnötig erschwert, auf der anderen Seite wird der komplette Test einer Software nicht selten nur an der Dialogschicht durchgeführt. Begründet wird dieses Vorgehen damit, daß mit dieser Technik die spätere Benutzung des Kunden simuliert wird, wodurch alle potentiellen Fehler gefunden werden, die während der Anwendung des Programms auftreten können.

Doch gerade unter der Prämisse der Wiederverwendbarkeit von Klassen oder gar ganzen Java-Projekten sollte der Anspruch an die Tests wesentlich höher sein. Schließlich können Tests dieser Art kaum die möglicherweise in den Tiefen der Software verborgenen kritischen Konstellationen von Variableninhalten aufdecken, welche Fehler produzieren würden, jedoch nur durch die Eingabe ganz bestimmter Wertkombinationen in die Benutzeroberfläche auftreten würden.

Es ist nicht verwunderlich, daß dieses Problem existiert, handelt es sich bei den Oberflächentests um reine Black-Box-Tests. Gerade diese stellen, wie bereits im entsprechenden Kapitel erwähnt, einen hohen Anspruch an die Qualität der gewählten Testfälle. Bei der alternativen Verwendung von Grey-Box-Verfahren kann zwar die Güte der Testfälle gesteigert werden, da in diesem Fall dem Tester die verwendeten Algorithmen und deren Schwächen bekannt sind, jedoch wird auch dann immer noch davon auszugehen sein, daß einige Fehler unentdeckt bleiben werden.

Wird die Benutzerschnittstelle als eigene Schicht implementiert, so stellt sie ein eigenes Modul dar, welches auch eigenständig getestet werden sollte. In dem Fall gelten

auch hier die Voraussetzungen für Modultests. Es ist somit grundsätzlich ein STUB und ein Treibermodul zu programmieren. Wurde die Funktionalität der Software bereits komplett implementiert, also das Bottomup-Verfahren verwendet, so fällt die Programmierung des STUBs weg. Dieses Vorgehen hat auch den Vorteil, daß davon ausgegangen werden kann, daß die Funktionalität des bereits implementierten Codes getestet wurde.

Durch dieses Vorgehen schränkt sich die Zahl der notwendigen Testfälle stark ein, da nicht die gesamte Funktionalität der Software getestet werden muß. Es bleibt nur noch zu überprüfen, wie der Dialog mit Eingabe- oder Programmfehlern umgeht und ob der Transport der Ein- und Ausgabedaten funktioniert.

Die Verifikation von Meldungen der Software kann dabei nur in Grenzen automatisiert werden. Werden diese z.B. als Reaktion auf einen Fehler in einer Statusleiste des Dialogfensters ausgegeben, so kann diese auch von einem Testwerkzeug ausgelesen und auf den korrekten Inhalt hin überprüft werden. Wird dagegen ein neues Fenster, eine sog. Messagebox, geöffnet und darin die Meldung angezeigt, so muß erst die Existenz des Fensters nachgewiesen, dann dessen Inhalt überprüft und schließlich das Fenster selbst wieder geschlossen werden.

Welches Vorgehen der Dialog verwendet und wie die Meldungen formuliert werden müssen, um verständlich zu sein, sollte im Vorfeld von einem Styleguide festgelegt werden. Schließlich hat ein Testwerkzeug keinen Geschmack, um die Lesbarkeit und Eindeutigkeit der Meldung zu überprüfen. Daraus folgt, daß ein automatisiertes Testwerkzeug nur in der Lage ist, zu überprüfen, ob der Fehler erkannt und gemeldet wurde, nicht jedoch, ob die Meldung auch verständlich ist oder den Anwender eher verwirrt.

Ein ähnliches Problem wird der Test der Oberfläche nach ergonomischen Gesichtspunkten sein. Beispielsweise kann nur der Anwender erkennen, ob die Schriftgröße zu klein oder die Aufteilung des Dialogs unlogisch und konfus ist. Dagegen kann ein Testwerkzeug anhand einer Liste durchaus überprüfen, ob alle Komponenten des Fensters komplett und in der richtigen Reihenfolge durch die Tabulatortaste erreichbar sind. Jedenfalls ist es in Swing kein Problem, z.B. ein Textfeld zu fragen, ob es gerade den Fokus hat und somit die Benutzereingaben entgegennehmen würde. Ebenso wäre ein automatisiertes Verfahren durchaus in der Lage zu überprüfen, ob alle Elemente eines Dialogs gerade ausgerichtet sind und exakt unter- oder nebeneinander stehen. Werden

zusätzlich dem Testwerkzeug auch Positionsangaben und mögliche Toleranzen eines jeden Elementes mitgegeben, kann es weiterhin überprüfen, ob sich alle an dem vom Designer vorgeschriebenen Platz befinden.

Da nicht alle an der Benutzerschnittstelle notwendigen Tests grundsätzlich automatisiert werden können, muß für alle anstehenden Tests einer Oberfläche abgewägt werden, welche dieser Tests durch ein Werkzeug realisiert werden können und welcher Aufwand dann dazu nötig ist, und welche Tests besser von einem humanoiden Tester durchgeführt werden sollten.

## *5.2 Technik der Simulation von Benutzerinteraktion*

Wie im obigen Abschnitt beschrieben gibt es Tests, welche Fehler aufgrund von Eingaben erzeugen sollen, um die Reaktion der Software darauf zu überprüfen. Sollen solche Tests automatisiert werden, ist es unumgänglich, die Interaktion des Anwenders mit dem Programm zu simulieren. Um dies zu erreichen, bieten sich zwei grundsätzlich unterschiedliche Verfahren an, deren Ziel es ist, sämtliche dem Anwender zur Verfügung stehenden Eingabegeräte zu simulieren und die Reaktion des Programms entgegenzunehmen. Die wesentlichen Eingabegeräte sind Tastatur und Maus, doch auch Trackballs oder Graphiktabletts, wie sie bei 3D-Anwendungen zum Einsatz kommen, sind denkbar.

Das eine Verfahren versucht nun, die Simulation der Benutzerinteraktion mit dem Programm auf Betriebsystemebene zu realisieren. Dazu wird auf dem Weg zwischen Eingabegerät und Betriebsystem eine Stelle gesucht, an der erfolgreich das Drücken einer Taste oder das Bewegen der Maus dem System vorgegaukelt werden kann. Diese Form der Betriebsystem-Manipulation wird bereits erfolgreich von Softwareprodukten zur Fernwartung von Client- oder Server-PCs in einem Intra- oder Internet sowie zur Erreichung eines Thin-Client-Konzeptes innerhalb eines Firmenintranets durchgeführt. Beispiele solcher Programme sind VNC[36], WTS[37] oder Citrix Metaframe[38].

---

[36] VNC steht für „Virtual Network Computing“, s.a. http://www.uk.research.att.com/vnc

[37] WTS steht für „Windows Terminal Server“ und wurde von Microsoft entwickelt

[38] s.a. http://www.citrix.com

Auch für die Durchführung automatisierter Softwaretests bietet sich diese Technik an. So können die zu erzeugenden Testfälle von einem Rekorderwerkzeug aufgezeichnet und für die Testdurchführung wieder abgespielt werden. Dabei werden alle Eingaben exakt wiederholt.

Nachteilig an dieser Technik ist allerdings, daß der zu testende Dialog nur etwas verrutscht sein muß, damit beispielsweise die Maus nicht mehr die Schaltflächen trifft. Ebenso werden Änderungen am Design des Dialogs dazu führen, daß die Testfälle komplett neu aufgezeichnet werden müssen.

Ferner sind die für diese Technik notwendigen Eingriffe in das Betriebsystem keineswegs trivial und können nur von einem Systemprogrammierer entwickelt werden. Auch wird es sehr schwierig, von dem Dialog dargestellte Ergebnisse auszulesen und zu verifizieren. Zudem ist die Umsetzung eines solchen Werkzeuges nicht in jeder Sprache uneingeschränkt möglich. So ist z.B. Java für eine solche Aufgabe aufgrund der Plattformunabhängigkeit denkbar ungeeignet, da sie keine Möglichkeit zur Manipulation des Betriebsystems vorsieht. Die Routinen zum Aufnehmen und Abspielen der Eingabesequenzen werden normalerweise in C/C++ oder Assembler zu schreiben sein.

Der Vorteil dieser Technik dagegen ist, daß es keine Rolle spielt, in welcher Sprache der zu testende Dialog programmiert wurde, da die Simulation der Anwenderinteraktion für den Dialog nicht von der Originalanwendung unterschieden werden kann. Ferner sind keine Eingriffe in dem Programmcode des Dialoges notwendig, um ihn testen zu können.

Das zweite mögliche Verfahren verfolgt einen komplett anderen Ansatz. Wird in einem Dialog eine Schaltfläche gedrückt, so löst diese Aktion ein Ereignis (Event) aus, welches an den sog. Event-Listener weitergereicht wird. Dieser wertet das Ereignis aus und ruft dann eine mit dem Ereignis verbundene Methode auf. Es sollte daher möglich sein, ein solches Ereignis manuell zu erzeugen und an den Event-Listener zu senden. Dabei ist zu berücksichtigen, daß die verschiedenen Ereignisse durch unterschiedliche Typen von Event-Listener repräsentiert werden.

In Java ist bei der Verwendung eines Swing-Dialogs die Realisierung der Technik zur „Fernzündung“ von Events sehr einfach. Ist die Instanz des Dialogs und der Name des Elementes bekannt, so kann über alle Elemente des Dialogs mit der Hilfe einer rekursiven Methode traversiert werden, um die Instanz des benannten Elementes zu fin

den. Handelt es sich dabei beispielsweise um eine Schaltfläche, so kann dieser die Nachricht `doClick()` gesendet werden, was einem manuellen Anklicken des Buttons mit der Maus gleichkommt.

Dieses Vorgehen kann äquivalent für alle möglichen Elemente eines Dialoges angewandt werden. Dabei ist es nicht zwingend notwendig, immer auch ein Ereignis auszulösen. Für Textfelder z.B. existiert eine Methode zum direkten setzen des Inhaltes. Da es kein einheitliches Vorgehen für die Manipulation aller möglichen Elemente eines Swing-Dialogs gibt, wird es notwendig sein, für jedes Element eine eigene Routine zu implementieren, welche die notwendigen und möglichen Manipulationen an dem Element vornehmen kann.

Es gibt bereits zahlreiche Programme auf dem Markt, die eine der beiden Techniken verwenden, um Tests an graphischen Benutzeroberflächen durchzuführen. Aus diesem Grund wird bis auf weiteres darauf verzichtet, eine Funktionalität zur Durchführung solcher Tests in das Skript mit aufzunehmen.

# 6 Ausblick und Kritik

Im nächsten Schritt wird versucht werden, das Testwerkzeug mit dem von Mark Hasemann entwickelten System zur Überdeckungsmessung zu koppeln, um damit eine Aussage über die Testgüte zu erhalten. Die Aussagekraft dieser Technik wurde bereits diskutiert und ist sicherlich mit Vorsicht zu interpretieren. Grundsätzlich kann jedoch auf diesem Weg sehr gut überprüft werden, ob die erzeugten Testfälle das gesamte zu testende Programm abdecken.

Ferner wird noch darüber nachzudenken sein, ob die Erstellung des vorgestellten XML-Skriptes in seiner derzeitigen Form vereinfacht werden kann. Der bisherige Testeinsatz der Software hat einen Vorgeschmack darauf geliefert, wie kompliziert und umständlich die Erzeugung des Skriptes ist. Der Entwickler muß zur Beschreibung der Testfälle in der Skriptsprache seine reine Programmiertätigkeiten unterbrechen und umfangreichen Text schreiben, um die Testfälle anzulegen. Erschwert wird diese Tätigkeit zusätzlich dadurch, daß das Skript leicht unübersichtlich werden kann, obwohl es kommentiert werden kann. Der Entwickler wird daher diesen Aufwand als eher hinderlich statt produktiv bewerten. Vor allem durch die in der Arbeit angesprochenen Sonderfälle, für welche entweder Treibermodule bzw. Treiberklassen oder STUBs zu implementieren sind, wird das Erstellen von Testfällen weiter erschwert.

Das Abfangen von Methodenaufrufen, die zur Testzeit vordefinierte Ergebnisse liefern müssen, ist nicht leicht zu realisieren. Einerseits sind diese neuen Methoden, welche im Testfall die vordefinierten und sonst die echten Ergebnisse liefern sollen, ebenfalls zu schreiben, was u.a. Zeit kostet. Andererseits muß während der Entwicklung der Software darauf geachtet werden, die entsprechenden Aufrufe auch umzuleiten.

Das Problem des in der Software verbleibenden Aufrufs dieser Methoden, der möglicherweise bei dem Einsatz der Software zu Fehlern führen könnte, ist noch nicht gelöst. Es ist denkbar, die Umleitung durch einen direkten Eingriff in die JVM mit der Hilfe von Techniken zu realisieren, die normalerweise für das Debugging zur Verfügung stehen und eine manuelle, direkt programmierte Umleitung überflüssig machen. Eine Orientierung in diese Richtung und eine Überprüfung der Möglichkeiten stehen jedoch noch aus.

So lange diese Frage noch nicht geklärt ist, bleibt dem Entwickler nur die Möglichkeit, ein vernünftiges Objektmodell für die STUBs der kritischen Methodenaufrufe zu verwenden, welches deren Einsatz so angenehm wie möglich macht. Es ist beispielsweise vorstellbar, eine globale STUB-Klasse zu schreiben, dessen einziger Inhalt aus einer als `public static boolean` deklarierten Variablen `isTest` besteht. Von dieser Klasse können alle notwendigen Klassen abgeleitet werden, welche die Ersatzmethoden für die als kritisch bewerteten Methoden enthalten. Der Vorteil ist, daß nur einmal die statische Variable `isTest` auf `true` oder `false` gesetzt werden muß und dieser Wert somit für alle abgeleiteten Klassen gleich ist. Das Umstellen der Variablen kann leicht von dem Testskript übernommen werden. Es sind jedoch auch andere Objektmodelle möglich. Die endgültige Entscheidung, wie die STUBs implementiert werden sollen, bleibt damit dem Entwickler überlassen.

Es wird sich zeigen müssen, ob die Anwender das automatisierte Testverfahren trotz des Aufwandes zu schätzen wissen. Schließlich wird bei nachfolgenden Arbeiten für Verbesserungen und Erweiterungen an der Software das Testen der durchgeführten Änderungen durch das bereits existierende Skript stark vereinfacht, was auch die eigentliche Stärke der Automation darstellt.

Um die Skripterstellung zu simplifizieren, ist ein Werkzeug geplant, das die Testfälle übersichtlich darstellt, das Hinzufügen, Löschen oder Ändern von Testfällen komfortabel ermöglicht, dem Prozeß der Programmierung angepaßt ist und möglicherweise auch eine Archivierung und Verwaltung von Testfällen vornimmt. Ein solches Erstellungswerkzeug würde vor allem auch verhindern können, daß Testfälle nicht funktionieren, weil sich der Entwickler bei der Bezeichnung des Klassentyps vertippt hatte.

Ferner ist auch die Implementierung eines Werkzeugs zur Darstellung und Archivierung der Testergebnisse geplant. Ein solches Programm könnte die Auswertungsarbeiten stark vereinfachen. Der Entwickler muß auf einen Blick erkennen können, welche Tests funktioniert haben und welche nicht. Bei den Letzteren ist ferner zu unterscheiden, ob das positive Testergebnis evtl. auf Folgefehlern beruht, wie beispielsweise einer fehlgeschlagenen Instanziierung einer Klasse.

Dazu ist es notwendig, daß die Meldungen des Testtools in eine Datei geschrieben werden. Vorzugsweise kann hier erneut XML zum Einsatz kommen. Bei der Implementierung des Testwerkzeuges wurde dieser Erweiterung bereits Rechnung getragen,

denn es existiert ein Interface, welches die notwendigen Methoden zur Ausgabe aller Meldungen beschreibt. Ferner ist mit der Klasse `DefaultAusgabe`, die in Abbildung 21 auch zu sehen ist, eine Standardimplementierung dieses Interfaces vorgenommen worden, welche alle Ausgaben auf den Bildschirm bzw. die Konsole schreibt und als Basis für eigene Ausgabeklassen dienen kann.

Ein weiteres ungelöstes Problem ist der Abbruch eines Methodenaufrufs, der in einer Endlosschleife festhängt. In dem Fall würde das gesamte verbleibende Skript nicht mehr abgearbeitet werden. Daher muß es ermöglicht werden, daß nach dem Erreichen eines vorher vom Tester festgelegten Timeouts die Ausführung der Methode als Ergebnislos abgebrochen wird. Selbstverständlich muß der Wert für den Timeout sinnvoll gewählt werden.

Die einzige Möglichkeit, diesem Problem zu begegnen, ist die Verwendung eines eigenen Threads für den Methodenaufruf, der nach dem Ablauf des Timeouts abgebrochen werden kann. Leider ist es in Java nicht so ohne weiteres möglich, einen Thread zu terminieren, da möglicherweise gerade eine Aktion durchgeführt wird, die nicht unterbrochen werden darf, wie z.B. das Schließen einer Datei. Auf weitere Details der Threadsteuerung unter Java soll an dieser Stelle nicht eingegangen werden. Es bleibt festzuhalten, daß dieses Problem gelöst werden muß, um die komplette Verarbeitung des Skriptes trotz des fehlerhaften Verhaltens einer getesteten Methode zu garantieren.

Das in dieser Arbeit vorgestellte Programm zur Durchführung automatisierter funktionaler Black-Box-Tests ist fertig gestellt und kann eingesetzt werden. Es wird sich jedoch während des Einsatzes noch zeigen, ob die durch das XML-Skript ermöglichten Tests ausreichend sind, oder ob es noch erweitert werden muß. Vor allem in Bezug auf die Vereinfachung der Definition von Testfällen ist sicher noch Potential vorhanden, das ausgeschöpft werden kann.

# Quellenverzeichnis

## a) Monographien


Binder, Robert V. (2000), Testing object-oriented systems, Models, Patterns, and Tools, Addison Wesley, Massachusetts, 2000

Dustin, E., et al. (2000), Software automatisch testen, Verfahren, Handhabung und Leistung, Rashka, J., Paul, J., Berlin, Springer Heidelberg; New York; Barcelona; Hongkong; London; Mailand; Paris; Singapur; Tokio, 2000

Hansen, H. R. (1998), Wirtschaftsinformatik I, Grundlagen betrieblicher Informationsverarbeitung, Bea, F.X., Dichtel, E., Schweitzer, M. (Hrsg.), 7., völl. neubearb. u. stark erw. Aufl. (1996), durchgeseh. Nachdruck (1998), Lucius & Lucius Stuttgart, 1998

Hasemann, Mark (2001), Konzeption und Realisierung eines automatisierten Whitebox-Testsystems für Java basierende Software mittels der Überdeckungsmessung, Diplomarbeit, Bielefeld, 2001

Krüger, G. (2001), GoTo Java 2, Handbuch der Java-Programmierung, 2. Auflage, Addison-Wesley München, 2001

Myers, G.J. (1979), Methodisches Testen von Programmen, 7. Auflage, Oldenbourg München; Wien, 2001

Oestereich, B. (Hrsg.) (2001), Erfolgreich mit Objektorientierung, Vorgehensmodelle und Managementpraktiken für die objektorientierte Softwareentwicklung, Hruschka, P., Josuttis, N., Kocher, H., Krasemann, H., Reinhold, M., 2., aktualisierte und erg. Aufl., Oldenbourg München; Wien, 2001


## b) Nachschlagewerke


Claus, V. / Schwill, A. (1993), DUDEN Informatik, Ein Sachlexikon für Studium und Beruf, Engesser, H. (Hrsg.), 2., vollst. überarb. u. erw. Aufl., DUDENVERLAG Mannheim; Leibzig; Wien; Zürich, 1993

## Quellenverzeichnis (Fortsetzung)

### c) Internetmedien


| | |
|---|---|
| http://www.frankwestphal.de, | Extreme Programming über die Schulter geschaut..., Letztes Update: 15.07.2001 |
| http://www.w3schools.com, | W3Schools Online Web Tutorials, Refsnes Data , 2001 |
| http://www.junit.org, | JUnit, Testing Resources for Extreme Programming |

# Anhangsverzeichnis

## Anhang 1: Die DTD des verwendeten Skriptes

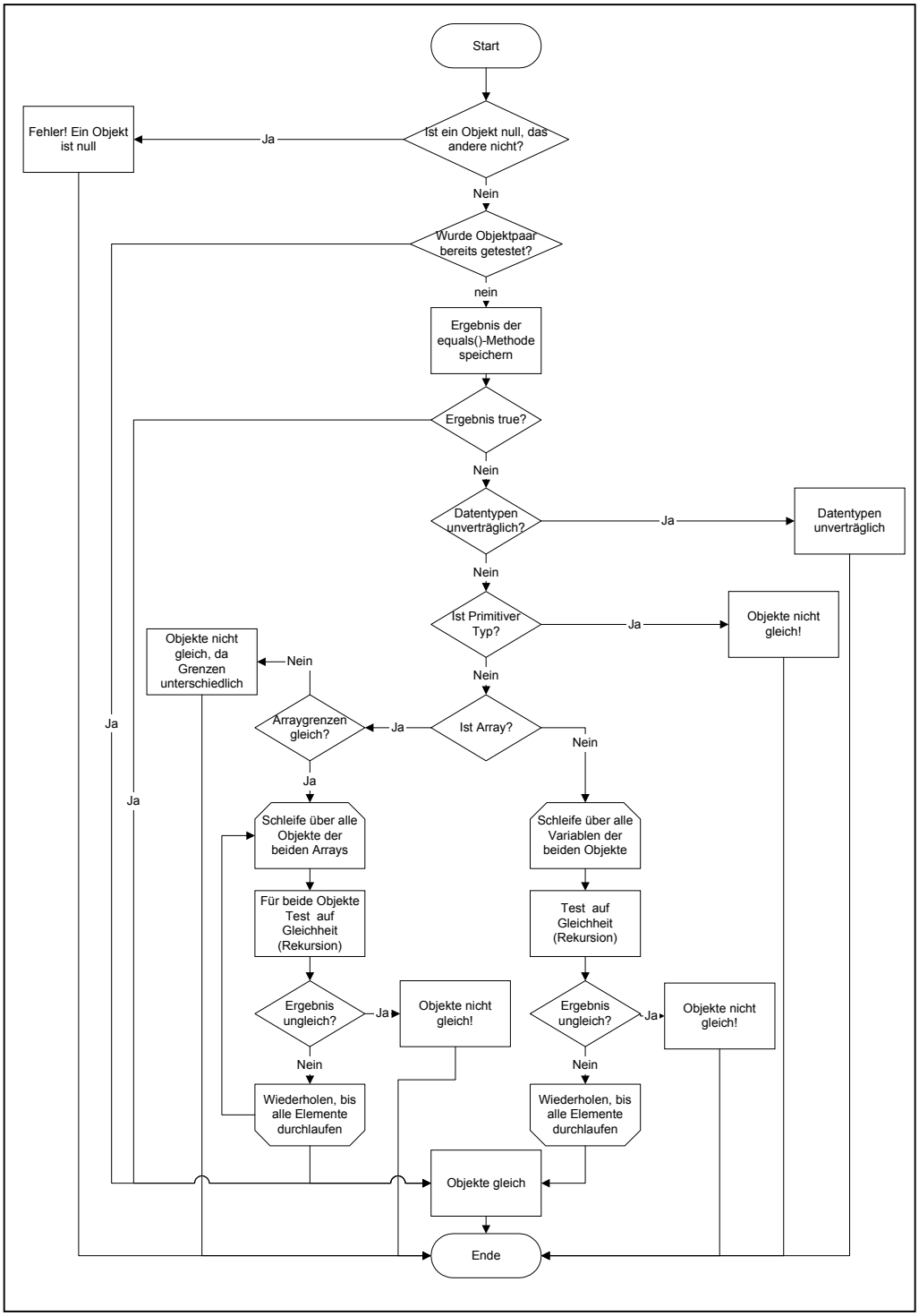
```
<!ELEMENT testscript (class | call | result | compare | array | member | include |
                      iteration | testsuite | dotestcase)+>

<!ELEMENT primitiv   EMPTY>
    <!ATTLIST primitiv name  CDATA #IMPLIED>
    <!ATTLIST primitiv type  CDATA #IMPLIED>
    <!ATTLIST primitiv value CDATA #IMPLIED>

<!ELEMENT member     (result | call | instance | class | primitiv | testsuitelength |
                      iterationindex | member | array)?>
    <!ATTLIST member name         CDATA #REQUIRED>
    <!ATTLIST member instancename CDATA #IMPLIED>
    <!ATTLIST member classname    CDATA #IMPLIED>

<!ELEMENT array      (result | call | instance | class | primitiv | testsuitelength |
                      iterationindex | member | array)*>
    <!ATTLIST array name CDATA #IMPLIED>

<!ELEMENT instance   EMPTY>
    <!ATTLIST instance name CDATA #REQUIRED>

<!ELEMENT parameters (result | call | instance | class | primitiv | testsuitelength |
                      iterationindex | member | array)+>

<!ELEMENT call       (parameters?)>
    <!ATTLIST call name         CDATA #REQUIRED>
    <!ATTLIST call instancename CDATA #IMPLIED>
    <!ATTLIST call classname    CDATA #IMPLIED>

<!ELEMENT settercall (parameters?)>
    <!ATTLIST settercall name CDATA #REQUIRED>

<!ELEMENT class      (parameters?, settercall*)>
    <!ATTLIST class instancename CDATA #REQUIRED>
    <!ATTLIST class classname    CDATA #REQUIRED>

<!ELEMENT result     (call | instance | class | primitiv | testsuitelength |
                      iterationindex | member | array)?>
    <!ATTLIST result name CDATA #REQUIRED>
    <!ATTLIST result type CDATA #IMPLIED>

<!ELEMENT compare    ((result | call | instance | class | primitiv | testsuitelength |
                       iterationindex | member | array),
                      (result | call | instance | class | primitiv | testsuitelength |
                       iterationindex | member | array))>
```

```
<!ELEMENT include     EMPTY>
    <!ATTLIST include name CDATA #REQUIRED>

<!ELEMENT iteration   (length, iterate)>
    <!ATTLIST iteration name CDATA #REQUIRED>

<!ELEMENT length (primitiv | testsuitelength)>

<!ELEMENT iterate (class | call | result | compare | array | member | include |
                   iteration | dotestcase)+>

<!ELEMENT iterationindex EMPTY>
   <!ATTLIST iterationindex of CDATA #REQUIRED>

<!ELEMENT testsuite (testcase+)>
   <!ATTLIST testsuite name CDATA #REQUIRED>

<!ELEMENT testsuitelength EMPTY>
   <!ATTLIST testsuitelength of CDATA #REQUIRED>

<!ELEMENT testcase (result | call | instance | class | primitiv | testsuitelength |
                    iterationindex | member | array)+>

<!ELEMENT dotestcase (primitiv | iterationindex)>
   <!ATTLIST dotestcase suitename CDATA #REQUIRED>
```

## Anhang 2: Codeauszug der Vergleichsmethode

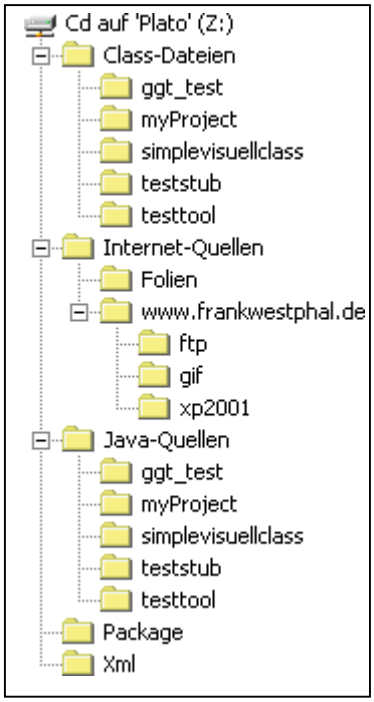
```
public static int performCompare(Object theObject1, Class theType1, Object theObject2, Class theType2, java.util.Hashtable testedObjects)
{
    if (theObject1==null || theObject2==null)
    {
        // ist ein Object null, das andere nicht, dann NOT_OK... Sind beide null, ist das OK!
        if (theObject1!=null || theObject2!=null) return NOT_OK; else return OK;
    }

    if (testedObjects==null) testedObjects = new java.util.Hashtable();

    if (theObject2.equals(testedObjects.get(theObject1))) return OK; // Vergleich wurde schon durchgeführt!
    testedObjects.put(theObject1, theObject2);

    boolean equality = theObject1.equals(theObject2);
    if (equality) return OK; // Bei gleichen Referenzen ist Gleichheit garantiert!

    if (theType1 != theType2) return NO_EQUAL_TYPES;

    if (!testCasting(theObject1, theType1)) return NO_CONVERTING_1;
    if (!testCasting(theObject2, theType2)) return NO_CONVERTING_2;

    if (theType1.isPrimitive()) // Sind zwei primitive Typen nicht gleich, dann raus...
        return NOT_OK;
    else
    if (theType1.isArray()) // Spezialfall Array
    {
        int length1 = java.lang.reflect.Array.getLength(theObject1);
        int length2 = java.lang.reflect.Array.getLength(theObject2);
        if (length1 != length2) return NO_EQUAL_ARRAY;

        Class arrayType1 = theType1.getComponentType();
        Class arrayType2 = theType2.getComponentType();

        if (arrayType1 != arrayType2) return NO_EQUAL_TYPES;

        for (int erg,i=0; i<length1; i++)
        {
            Object arrayObject1 = java.lang.reflect.Array.get(theObject1, i);
            Object arrayObject2 = java.lang.reflect.Array.get(theObject2, i);
            erg = performCompare(arrayObject1, arrayType1, arrayObject2, arrayType2, testedObjects);
            if (erg!=OK) return erg;
        }
    }
    else // Beliebige Objekte...
    {
        // Nun für alle Klassen und unterklassen versuchen, die Parameter zu vergleichen!
        Class [] Types = getClasses(theType1);
        for (int typesCount=0; typesCount<Types.length; typesCount++)
        {
            java.lang.reflect.Field [] fields = Types[typesCount].getDeclaredFields();

            for (int erg,i=0; i<fields.length; i++)
            {
                try
                {
                    fields[i].setAccessible(true);
                    Object o1 = fields[i].get(theObject1);
                    Object o2 = fields[i].get(theObject2);

                    Class t1 = fields[i].getType();

                    erg = performCompare(o1, t1, o2, t1, testedObjects);
                    if (erg!=OK) return erg;
                }
                catch (IllegalAccessException e)
                {
                    return NOT_OK;
                }
            }
        }
    }

    return OK;
}
```

## Anhang 3: Aktivitätsdiagramm der Vergleichsmethode




(Quelle: eigene Darstellung)

## Anhang 4: Inhaltsverzeichnis und Bedienung der CD-ROM




(Quelle: eigene Darstellung)

- Class-Dateien: In diesem Verzeichnis befinden sich alle kompilierten JAVA-Dateien, von denen einige ausgeführt werden können.
  - ggt_test: Die Datei „ggt_test.TestGGT“ kann ausgeführt werden. Zu diesem Zweck existiert die Stapeldatei „StartggTTest.bat“ im Root-Verzeichnis der CD. Der Test wird erst die prozedurale dann die objektorientierte Version testen.
  - myProject: In diesem Verzeichnis finden sich die class-Dateien für den demonstrativen Testlauf des Testtools. Sie können nicht ausgeführt, sondern existieren nur zu Demonstrationszwecken.
  - simplevisuellclass: Die Datei „simplevisuellclass.TestKlasse“ kann ausgeführt werden und demonstriert an einem simplen Dialog, wie dieser ferngesteuert werden könnte. Dazu wird das Textfeld mit einem Beispieltext gefüllt und der Button „OK“ betätigt. Dies führt dazu, daß der Text auf der Konsole ausgegeben wird. Zum Starten des Tests kann die Stapeldatei „StartVisuellTest.bat“ im Root-Verzeichnis der CD verwendet werden. Soll nur der Dialog gestartet werden, kann dazu die Stapeldatei „StartSimpleDialog.bat“ verwendet werden.

- teststub: In diesem Verzeichnis befinden sich drei Klassen, die untereinander in Abhängigkeit stehen. Sie demonstrieren ein STUB-Modul für den Methodenaufruf der Methode `GregorianCalendar.getTime()`.
- testtool: Diese Klassen repräsentieren das gesamte Testwerkzeug. Ein demonstrativer Durchlauf des Testtools kann mit der Stapeldatei „StartTest.bat“ gestartet werden. Die im Verzeichnis \XML abgelegten Skripte werden abgearbeitet. Die Ausgaben sind sehr umfangreich. Daher empfiehlt es sich, die Ausgabe beispielsweise mit „StartTest >%TEMP%\ausgabe.txt“ in eine Datei umzuleiten.

- Java-Quellen: Die Verzeichnisstruktur entspricht dem Verzeichnis „Class-Dateien“. Sämtliche Quellen können hier besichtigt.

- Internet-Quellen: Über die Datei „index.html“ im Root-Verzeichnis der CD können die hier abgelegten Dateien aufgerufen werden.

- Package: In diesem Verzeichnis befindet sich der XML-Parser „XML4J“ von IBM, der für das Testwerkzeug zwingend benötigt wird.

- XML: Hier befinden sich die XML-Skripte und die DTD, welche die Tests beschreiben.

## Ehrenwörtliche Erklärung

Ich versichere, daß ich die beiliegende Diplomarbeit ohne Hilfe Dritter und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt und die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Lemgo, 03.Oktober 2001

Ort, Datum

Unterschrift