Testverfahren für objektorientierte prototypbasierte Software-Entwicklung

Eine Fallstudie aus dem Investmentbanking

Sami Beydeda

Diplomarbeit
am Fachbereich Informatik
der Universität Dortmund

Prof. Dr. Volker Gruhn
Lehrstuhl Software-Technologie

Dr. Hermann Haaf
Dresdner Bank AG, Frankfurt

Dortmund, 18. Dezember 1998

Ich bedanke mich bei Prof. Dr. Volker Gruhn und Dr. Hermann Haaf für die geduldige Betreuung der Diplomarbeit. Mein Dank gilt auch Dr. Hans-Jürgen Tillemans und Plamen Neykov von der Dresdner Bank AG für die freundliche Unterstützung. Ohne ihre Hilfe wäre diese Arbeit nicht in dieser Qualität zustande gekommen. Last not Least möchte ich auch Konrad Novak, ebenfalls von der Dresdner Bank AG, für die anregenden Diskussionen danken.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Ein wichtiger Geschäftsbereich moderner Banken ist das *Investmentbanking*. Das Investmentbanking umfaßt neben Dienstleistungen zur Finanzierung von Unternehmen, wie beispielsweise der Emission von Aktien oder Anleihen, auch Handelsaktivitäten auf verschiedenen Märkten mit unterschiedlichen Finanzinstrumenten. Diese Finanzinstrumente werden mit unterschiedlichen Zielen gehandelt, welche in der Risikominimierung, der Spekulation und der Ausnutzung von Arbitragemöglichkeiten zu sehen sind [Hul96]. Das Gewicht der Handelsaktivitäten wird dabei allein schon aus den erzielten Umsätzen deutlich. Im Monat März dieses Jahres wurde erstmals die Umsatzgrenze von einer Billion DM an den deutschen Wertpapierbörsen überschritten, allein am 20.03.1998 lag der Umsatz nur im Aktienhandel an den deutschen Wertpapierbörsen bei einem Rekordwert von 140 Mrd DM [DB98]. Neben dem Handel an den Wertpapierbörsen wird aber auch auf den *Over-The-Counter Märkten* [Hul96] gehandelt, d.h. Anbieter und Nachfrager eines Finanzinstruments schließen ihre Geschäfte direkt ab, ohne den Umweg über eine Börse. Damit liegen die erzielten Umsätze noch über den oben genannten Zahlen.

Der Handel der Finanzinstrumente erfolgt dabei teilweise rein elektronisch, wie beim XETRA-System in Deutschland. Käufer und Verkäufer geben ihre Aufträge einem lokalen Terminal ein, der Computer bestimmt dann unter Berücksichtigung der Preis- und Mengenvorstellungen der Käufer und Verkäufer den Preis, zu dem der maximale Umsatz getätigt werden kann. Bei einer Präsenzbörse übernimmt diese Aufgabe der Kursmakler [Sch98]. Bei den oben aufgelisteten Zahlen ist die Bedeutung von fehlerfreier Software im Investmentbanking offensichtlich.

Wie abhängig die Wertpapierbörsen schon von der Rechnerunterstützung sind, wird insbesondere bei einem Rechnerausfall deutlich. Durch Rechnerausfälle am 21. und am 23.07.1997 wurde an der Frankfurter Wertpapierbörse der Handel stundenlang unterbrochen [Welt97]. Die Verluste waren an diesen Tagen immens, da der Handel an anderen Börsen und im Ausland weiterhin stattfand und in Frankfurt nicht darauf reagiert werden konnte. Nach [Mah98] bedeutet ein einstündiger Rechnerausfall für eine Bank einen Verlust von rund 12 Mio DM allein im Aktienhandel.

Der Wunsch nach fehlerfreier Software ist also in Anbetracht der erzielten Umsätze und leider auch der Probleme in jüngster Zeit begründet. Bisher wurden die Anforderungen an den Software-Test im Investmentbanking, insbesondere im Handel von

Finanzinstrumenten, kaum untersucht. Es gibt zwar schon Veröffentlichungen, die Entwurfsmuster und Rahmenwerke für die Entwicklung von spezieller Software in diesem Umfeld beschreiben [ZS96a, ZS96b, ZS96c, EG92, BE93], aber es existieren nach bestem Wissen des Autors keine Veröffentlichungen, die die Aspekte des Testens untersuchen.

Aber auch fehlerfreie Software kann ein wirtschaftliches Risiko beinhalten. An der Wall Street in New York wird nach [Vir98] jedes fünfte bis sechste Geschäft in Aktien von einem Computer entschieden. Wenn allerdings zu viele Handelsparteien durch dieselbe Software bzw. ähnliche Software dieselbe Strategie verfolgen, kann in bestimmten Situationen die Nachfrage oder das Angebot in einer Aktie fehlen, so daß der Preis der entsprechenden Aktie sich extrem nach oben oder unten bewegen kann. Am *Schwarzen Montag*, dem 19.10.1987 trugen solche Handelsprogramme zu einem Crash der Börse in New York bei. An diesem Tag verlor der Dow Jones Index mehr als 20 %. Kritisiert wird auch die Qualität der automatisch bestimmten Preise im XETRA-System. Besonders zu Beginn und am Ende eines Handelstages treten häufig große Preisunterschiede zwischen der Präsenz- und der Computer-Börse auf [Mah98]. Software muß also nicht nur fehlerfrei entwickelt, sondern auch richtig eingesetzt werden.

Das Investmentbanking besitzt einige Besonderheiten, die sich auch auf die Software-Unterstützung auswirken. Dazu gehört zum einen die Tatsache, daß die Märkte, auf denen agiert wird, relativ „eng" sind und sich auch schnell verändern. Aus diesem Grund müssen Markterfordernisse schnell erkannt und auch ausgenutzt werden. Dazu kann beispielsweise die Entwicklung neuer Finanzinstrumente gehören. Außerdem müssen auch zur Erhaltung der Wettbewerbsfähigkeit regelmäßig neue Dienstleistungen bzw. neue Finanzinstrumente in die Märkte eingeführt werden. Es muß vor allem möglich sein, auf Marktlücken schnell reagieren zu können. Die Wettbewerbsfähigkeit ist nur dann sichergestellt, wenn auch die Entwicklung der entsprechenden Software-Unterstützung angepaßt ist. Der Software-Prozeß muß dazu relativ flexibel sein. Die Entwicklung der Software muß in möglichst kurzer Zeit geschehen. Diese Aspekte und auch die enge Kommunikation zwischen Anwendern und Entwicklern, die durch die Entwicklung der Software für den eigenen Bedarf gegeben ist, legen das *Prototyping* in der Software-Entwicklung nahe.

In dieser Diplomarbeit steht ein spezielles Problem aus dem Investmentbanking im Mittelpunkt. Es handelt sich dabei um die Bewertung von Zinsderivaten. Die Bewertung von Zinsderivaten erfolgt auf der Grundlage von theoretischen Modellen, die durch Annahmen über bestimmte Marktfaktoren Preise für Zinsderivate implizieren. In günstigen Fällen können aus den Annahmen geschlossene Preisgleichungen hergeleitet werden. Häufig müssen jedoch die Preise numerisch durch *rekombinierende Bäume* ermittelt werden.

Die software-technische Umsetzung der rekombinierenden Bäume für die Bewertung von Zinsderivaten ist jedoch sehr fehleranfällig, da die Modelle eine hohe Komplexität besitzen und die Implementierung einen nicht zu unterschätzenden Aufwand bereitet. Hinzu kommt, daß Fehler, die nur zu kleinen Abweichungen in den berechneten Preisen führen, nicht offensichtlich sind. Diese Fehler können aber durch die hohen Nominalbeträge, die den Zinsderivaten häufig zugrunde liegen, hohe wirtschaftliche Verluste verursachen. Der Bedarf nach zuverlässigen Testverfahren ist also deutlich.

Das Ziel der Diplomarbeit ist damit die Entwicklung eines möglichst zuverlässigen Testverfahrens, das den objektorientierten, prototypbasierten Charakter der Software-Entwicklung im Investmentbanking berücksichtigt.

Das objektorientierte Paradigma hat sich nicht nur im Investmentbanking bei der Entwicklung von Software durchgesetzt. Das Testverfahren sollte die Objektorientierung explizit unterstützen, da Testverfahren für strukturierte Software nicht unbedingt bei objektorientierter Software eingesetzt werden können.

Jedes Zwischenergebnis der Software-Entwicklung sollte so früh wie möglich auf Korrektheit geprüft werden, um die Software-Entwicklung nicht auf einer fehlerhaften Basis fortzuführen. Für die prototypbasierte Software-Entwicklung bedeutet das, daß insbesondere jeder Prototyp getestet werden sollte. Entscheidend ist dabei die Tatsache, daß häufig in einer einzelnen Prototyping-Iteration nur ein bestimmter Abschnitt in der Implementierung des Prototyps verändert wird. In günstigen Fällen ist nur der Test dieser veränderten Abschnitte notwendig, so daß der Testaufwand wesentlich geringer ausfällt.

Die prototypbasierte Software-Entwicklung kann auch aus einer anderen Perspektive betrachtet werden. Wird die erste Prototyping-Iteration einer gewöhnlichen Software-Entwicklung, die beispielsweise nach dem klassischen Wasserfallmodell durchgeführt wird, gleichgesetzt, so können alle nachfolgenden Prototyping-Iterationen als Korrekturen in der Wartungsphase betrachtet werden. Der Test von korrigierter Software wird in der Literatur als *Regressionstest* bezeichnet. Damit liegt auch der Einsatz von Regressionstestverfahren für den Test von Prototypen nahe.

Zu diesem Ziel faßt Kapitel 2 der Diplomarbeit die wichtigsten Begriffe aus dem Prototyping zusammen. Anschließend wird in Kapitel 3 das Testen von Software erläutert. Kapitel 4 enthält die Ausgangsbasis des in dieser Diplomarbeit entwickelten Testverfahrens, wobei das Testverfahren schließlich in Kapitel 5 beschrieben wird. Kapitel 6 stellt die für das Verständnis der Arbeit notwendigen finanzwirtschaftlichen Kenntnisse zur Verfügung, während Kapitel 7 die Anwendung des Testverfahrens an einem konkreten Problem demonstriert. Das letzte Kapitel der Diplomarbeit faßt die Ergebnisse abschließend zusammen und bietet einen Ausblick über die Diplomarbeit hinaus.

2 Prototyping

Dieses Kapitel erläutert die wichtigsten Begriffe im Rahmen des Prototypings. Dazu werden im Abschnitt 2.1 das Prototyping in den Prozeß der Software-Entwicklung eingeordnet und die Begriffe des Prototyps bzw. des Prototypings definiert. Es werden die einzelnen Phasen im Prototyping und die verschiedenen Arten des Prototypings, die nach den verfolgten Zielen unterschieden werden, erläutert. Abschnitt 2.2 faßt schließlich die Besonderheiten im objektorientiertem Prototyping kurz zusammen.

2.1 Klassisches Prototyping

Stark fallende Hardware-Preise und immer leistungsfähigere Rechner haben in Verbindung mit den steigenden Erwartungen der Auftraggeber an Software dazu geführt, daß Software-Systeme einen sehr hohen Komplexitätsgrad erreicht haben. Da jedoch die Entwicklung der software-technischen Methoden nicht mit der Entwicklung im Hardware-Sektor mithalten konnte, ergeben sich bei der Realisierung solcher komplexer Software-Systeme massive Probleme, die sich in der schlechten Qualität der erstellten Systeme oder in größeren Managementproblemen des Software-Prozesses[1] zeigen. Diese Diskrepanz zeigt sich besonders bei verteilten Systemen, wo die Entwicklung der software-technischen Methoden zur Realisierung von verteilten Anwendungen gewaltig der Hardware-Entwicklung hinterherhinkt.

Es wurde schon relativ früh ein Prozeßmodell vorgeschlagen, das die methodische Vorgehensweise aus den Ingenieurwissenschaften in der Entwicklung von Software einsetzen sollte, um Software hoher Qualität kostengünstig und termingerecht produzieren zu können. Das sogenannte *Wasserfallmodell* bzw. *klassisches Phasenmodell* [Roy70] unterteilt den Software-Entwicklungsprozeß linear in verschiedene Phasen, wobei eine Phase nur dann begonnen werden kann, wenn die vorherige Phase vollständig abgeschlossen wurde. Allerdings hat sich diese rein sequentielle Vorgehensweise des Wasserfallmodells für die erfolgreiche Erstellung von komplexen Systemen aus verschiedenen Gründen als ungeeignet erwiesen. Häufig erweisen sich Entscheidungen in frühen Phasen des Wasserfallmodells später als fehlerhaft, so daß Rücksprünge zu diesen

[1] Eine Definition des Begriffs des Software-Prozesses findet sich in [Blz98].

Phasen notwendig werden, die aber im Wasserfallmodell nicht vorgesehen sind. Außerdem ist auch die Annahme der vollständigen Erfassung der Anforderungen vor dem Entwurf der Systemarchitektur problematisch. Aus diesen Gründen wurden verschiedene Varianten zum Wasserfallmodell und auch andere Software-Entwicklungsmodelle vorgeschlagen, die diese und andere Probleme lösen sollten [GJM91, PS94, Som95].

Das *Prototyping* [Flo84] hat sich insbesondere zur vollständigen und korrekten Erfassung der Anforderungen der Nutzer an das Software-Produkt bewährt. Die Idee des Prototyping besteht darin, den linearen Ablauf des Phasemodells an einer bestimmten Stelle zu unterbrechen und eine Iteration aus *Implementierung eines Systemmerkmals*, *Test durch den Auftraggeber* und *Feedback* einzufügen. Diese Iteration kann je nach Art des Prototypings und der verfolgten Ziele unterschiedliche Ausmaße annehmen. Während sich beim explorativen Prototyping diese Iteration nur in der Phase der Anforderungsdefinition vollzieht, wird beim evolutionären Prototyping der Software-Entwicklungsprozeß als ganzes iterativ durchgeführt, d.h. die Software „wächst" zum späteren Endprodukt. Bevor jedoch auf diese Begriffe weiter eingegangen wird, wird zuerst der Begriff des Prototyps bzw. der des Prototypings definiert.

Leider existiert in der Literatur keine eindeutige Definition des Prototypbegriffs, obwohl Prototypen schon seit mehr als einem Jahrzehnt in der Software-Erstellung genutzt werden. In der vorliegenden Diplomarbeit werden Software-Prototypen in Anlehnung an [Boa83], [CS89] und [Smi91] folgendermaßen definiert[2]:

> Unter einem (*Software-*) *Prototyp* soll eine möglichst einfach zu verändernde vorläufige Version oder ein lauffähiges Modell des Software-Systems verstanden werden, das die wesentlichen Eigenschaften des späteren Systems besitzt. Entsprechend soll unter *Prototyping* der Prozeß der Prototypenentwicklung verstanden werden.

Der Prozeß des Prototypings besteht aus den folgenden vier Schritten:

1. Funktionsauswahl
   Im ersten Schritt werden die in dem Prototypen zu realisierenden Funktionen bestimmt. Zu unterscheiden ist dabei das vertikale und das horizontale Prototyping. Beim *vertikalem Prototyping* wird eine bestimmte Untermenge der Systemfunktionen in ihrer endgültigen Form realisiert. Das *horizontale Prototyping* sieht hingegen die Implementierung aller Systemfunktionen in eingeschränkter Form vor.

2. Konstruktion
   Die Konstruktion des Prototyps kann entweder mit speziellen Prototyping-

[2]Dieser Prototypbegriff ist zu unterscheiden vom Prototypbegriff aus prototyporientierten Programmiersprachen, wie zum Beispiel OMEGA oder SELF, wo Prototypen als ein alternativer Ansatz zu Klassen in objektorientierten Programmiersprachen betrachtet werden [Bla94].

Werkzeugen oder mit denselben Werkzeugen erfolgen wie das endgültige Software-Produkt. Diese Entscheidung beeinflußt in starkem Maße den Aufwand für die Prototypenproduktion und für die Weiterverwendung.

3. Auswertung
   In diesem Schritt wird eine Bewertung des Prototyps anhand von vorher festgelegten Maßstäben durchgeführt.

4. Weiterverwendung
   Wie oben schon angedeutet, ist die Weiterverwendung des Prototyps nur dann möglich, wenn der Prototyp in derselben Umgebung realisiert wird wie das Endprodukt. Hinzu kommt, daß der Prototyp auch dieselben Qualitätsanforderungen erfüllen muß. Dieser Schritt kann bei Verwendung von Wegwerf-Prototypen entfallen.

Es werden je nach den verfolgten Zielen verschiedene Arten des Prototypings unterschieden [Flo84]:

Exploratives Prototyping

Das explorative Prototyping wird primär zur eindeutigen Identifikation der zu realisierenden Eigenschaften des Software-Systems und zur Diskussion von alternativen Lösungen genutzt. Hierbei steht vor allem die Beseitigung von Kommunikationproblemen zwischen den beteiligten Personen im Software-Prozeß im Vordergrund. Häufig besitzen die Entwickler geringe Kenntnisse im Anwendungsgebiet des späteren Software-Produkts, so daß sie durch das Prototyping sich das erforderliche Wissen aneignen können. Es kann aber auch vorkommen, daß die Anwender keine genauen Vorstellungen über den gewünschten Leistungsumfang der zu entwickelnden Software haben. In diesen Fällen kann durch die Implementierung von verschiedenen Systemfunktionen als Prototypen eine Auswahl getroffen und die tatsächlich benötigten Funktionen bestimmt werden.

Bei dieser Art des Prototypings werden meistens sogenannte *Wegwerf-Prototypen* verwendet, da das Ziel hier primär in einer verbesserten Kommunikation liegt und der Prototyp nicht unbedingt in das spätere Software-Produkt integriert werden soll. Daraus resultiert auch, daß in diesem Fall Prototypen nicht auf derselben Rechnerplattform implementiert werden müssen und daß der Prototyp nicht denselben Qualitätsanforderungen genügen muß wie das Endprodukt.

Experimentelles Prototyping

Das experimentelle Prototyping wird meistens zur Bewertung von Lösungen zu einer bestimmten Fragestellung innerhalb des Software-Prozesses eingesetzt. Dazu werden die einzelnen Lösungen als Prototypen implementiert und die Bewertung dieser

Lösungen durch Tests an diesen Prototypen durchgeführt. Diese Art des Prototypings kommt also dem Prototyping-Begriff aus den Ingenieurwissenschaften am nächsten.

Prototypen, die im Rahmen des experimentellen Prototypings entstehen, können in den späteren Phasen des Wasserfallmodells als Ergänzung zur Spezifikation dienen. Diese Prototypen können dabei, je nach der Phase in der sich das System befindet, eine komplementäre Form zur Spezifikation, eine verfeinernde Form zur Spezifikation oder aber auch eine Zwischenstufe zwischen Spezifikation und Implementierung bilden.

Experimentelle Prototypen können entweder als Wegwerf-Prototypen realisiert oder aber auch in das Software-Produkt integriert werden. Diese Entscheidung hängt vor allem von der Verfügbarkeit von Prototyping-Tools ab, die die Implementierung und die anschließende Integration bzw. Transformation des Prototyps in das Software-System vereinfachen.

Evolutionäres Prototyping

Das evolutionäre Prototyping weist unter den verschiedenen Formen des Prototypings die meisten Unterschiede zu dem traditionellen Prototyping-Begriff aus den Ingenieurwissenschaften auf. Häufig wird in der Literatur das evolutionäre Prototyping als evolutionäre bzw. inkrementelle Software-Entwicklung bezeichnet und somit nicht in das Prototyping eingeordnet. Das evolutionäre Prototyping basiert auf der Erfahrung, daß die Umgebung, in der das Software-System eingebettet ist, sich mit der Zeit verändert und Veränderungen des Systems notwendig werden, bzw. daß neue Anforderungen durch den Einsatz des Software-Systems entstehen. Unter diesen Voraussetzungen liefern die anderen Formen des Prototypings oder aber auch das Wasserfallmodell mit einem starren Anforderungskatalog nur unzureichende Ergebnisse, so daß eine andere Strategie zur Software-Entwicklung verfolgt werden muß. Aus der Sicht des evolutionären Prototypings besteht das Software-System nicht aus einem einzigen Endprodukt, sondern aus einer Serie von verschiedenen Versionen, wobei jede Vorgängerversion als Prototyp für die Nachfolgeversion dient. Im Rahmen des evolutionären Prototypings, wird zwischen der inkrementellen und der evolutionären Systementwicklung unterschieden:

- Das Prinzip der *inkrementellen Systementwicklung* besteht in der schrittweisen Realisierung von komplexen Problemen, d.h. das zu lösende Problem wird in mehrere kleinere Probleme unterteilt und diese dann nacheinander realisiert. Entsprechend können auch die Anwender schrittweise zur Bedienung des Systems geschult werden, was wiederum zu einer verbesserten Kommunikation beiträgt. Es soll betont werden, daß die inkrementelle Systementwicklung dem Wasserfallmodell nicht widerspricht, da nur die Implementierungsphase durch das inkrementelle Vorgehen beeinflußt wird.

- Bei der *evolutionären Systementwicklung* vollzieht sich der Entwicklungsprozeß als Ganzes innerhalb von verschieden Zyklen, die jeweils aus einer Entwurf-, einer Implementations- und einer Bewertungsphase bestehen. Im Gegensatz zur inkrementellen Systementwicklung wird hierbei nicht von einem starren Anforderungskatalog ausgegangen. Vielmehr wird die Einsatzumgebung des Systems als veränderlich angenommen, was dazu führt, daß auch die Anforderungen sich verändern. Zu betonen ist, daß in dieser Art der Systementwicklung kein der Wartungsphase des Wasserfallmodells entsprechender Prozeßzustand existiert. Vielmehr werden zur Wartung der Software weitere Evolutionszyklen durchlaufen.

2.2 Objektorientiertes Prototyping

Im letzten Abschnitt wurde schon auf die Bedeutung geeigneter Werkzeuge im Rahmen des Prototypings hingewiesen. Für die Erstellung von Wegwerfprototypen stehen dem Entwickler unterschiedliche Werkzeuge, wie zum Beispiel HyperCard[3] oder PowerBuilder[4] zur Verfügung, die mit ihrer eigenen Skriptsprache die Programmierung des Prototypen wesentlich vereinfachen. Diese Werkzeuge unterstützen den Entwickler vor allem auch in der Realisierung der Benutzeroberfläche. Dialogelemente und ihre Eigenschaften werden nicht programmiert, sondern mit der Maus durch Drag und Drop an die entsprechende Stelle auf dem Bildschirm plaziert.

Häufig soll aber der Prototyp, wie in dem Anwendungsbeispiel in dieser Diplomarbeit, in ein bestehendes Software-System integriert oder auch evolutionär zum endgültigen Software-System weiterentwickelt werden. In diesen Fällen können die oben genannten Prototyping-Werkzeuge nur begrenzt eingesetzt werden, da oft eine Entwicklungsumgebung bzw. eine Programmiersprache, in der das endgültige Software-System realisiert werden soll, vorgegeben ist. Wurden noch vor zehn Jahren der Großteil der Software-Projekte mit strukturierten Methoden und Programiersprachen, wie zum Beispiel C oder Pascal, entwickelt, so sind dies heutzutage objektorientierte Methoden und Programmiersprachen, wie C++ oder auch Java. Es liegt also nahe, auch das Prototyping mit objektorientierten Methoden durchzuführen [CS95]. Die Objektorientierung erweist sich aber auch aus anderen Gründen für das Prototyping als vorteilhaft. Die objektorientierten Techniken der Vererbung und des Polymorphismus unterstützen den Aufbau einer Klassenbibliothek, die in mehreren Projekten genutzt werden kann. In dieser Bibliothek werden häufig genutzte Klassen gespeichert, auf die dann bei der Programmierung von neuen Prototypen zurückgegriffen werden kann. Die Vererbung ermöglicht dabei das Anpassen der Klassen in dieser Bibliothek an eine neue Problemstellung. Die objektorientierte Programmierung ermöglicht also das Zusammenfügen von vorhandenen Software-Bausteinen zu größeren Software-Systemen,

[3] URL `http://www.apple.com/hypercard`.

[4] URL `http://www.sybase.com/products/powerbuilder`.

dadurch erübrigt sich die wiederholte Implementierung von häufig benötigten Funktionen. Eine besondere Art von Klassenbibliotheken stellen Application Frameworks dar. *Application Frameworks* bestehen aus einer Menge von Klassen, die untereinander eine festgelegte Beziehung haben, so daß sie zusammen eine übergeordnete Aufgabe erfüllen können, wie zum Beispiel die Microsoft Foundation Class Library.

Das Prototyping und die Objektorientierung können aber auch auf eine andere Art miteinander kombiniert werden. Das Prototyping kann mit den im ersten Abschnitt genannten Zielen in der objektorientierten Software-Entwicklung eingesetzt werden, um beispielsweise während der objektorientierten Spezifikation eine vollständige und fehlerfreie Erfassung der Anforderungen zu ermöglichen oder um einen möglichst optimalen objektorientierten Entwurf zu gewährleisten. Allerdings existieren in der Literatur nur wenige objektorientierte Vorgehensmodelle, die das Prototyping als einen festen Bestandteil beinhalten [Hes97]. Eine andere Idee wird in [MPD+96] verfolgt. In diesem Papier wird das Prototyping vor bestehende objektorientierte Entwicklungsmodelle vorgeschaltet, so daß die Ergebnisse des Prototypings in die objektorientierte Analyse eingehen können.

3 Testen von Software

Dieses Kapitel gibt einen Überblick über das Testen von Software. Das Testen wird hierzu in Abschnitt 3.1 in das Qualitätsmanagement eingeordnet, indem Software-Qualität definiert und das Testen als eine Maßnahme für die Qualitätssicherung identifiziert wird. Anschließend wird das Testen von Software in Abschnitt 3.2 definiert und in Abschnitt 3.3 die Vorgehensweise beim Testen von Software erläutert. Abschnitt 3.4 enthält Ausführungen zum Testen in der strukturierten Software-Entwicklung. Hier werden insbesondere in Abschnitt 3.4.1 die verschiedenen Testgegenstände, die bei der strukturierten Software-Entwicklung entstehen, genannt, und in Abschnitt 3.4.2 Testmethoden für den Test dieser Testgegenstände unterschieden. Eine ausführliche Beschreibung zweier Testmethoden für die strukturierte Software-Entwicklung befindet sich in Abschnitt 3.4.3. Das Testen in der objektorientierten Software-Entwicklung wird dagegen in Abschnitt 3.5 behandelt. In Abschnitt 3.5.1 sind Besonderheiten der objektorientierten Software-Entwicklung, die auch das Testen beeinflussen, aufgeführt. Abschnitt 3.5.2 listet in analogerweise zu Abschnitt 3.4.1 die Testgegenstände bei der Software-Entwicklung auf, wobei hier allerdings die objektorientierte Software-Erstellung im Mittelpunkt steht. Anschließend werden in Abschnitt 3.5.3 verschiedene Testmethoden für objektorientierte Software unterschieden und in Abschnitt 3.5.4 der Klassen- und der Integrationstest erläutert.

3.1 Software-Qualität

Die möglichen Gefahren durch ein Fehlverhalten von Software werden besonders beim Einsatz in sicherheitskritischen Bereichen, wie zum Beispiel bei medizinischen Geräten, deutlich. Hier kann ein Fehler in der Software fatale Folgen haben und unter Umständen zum Tod von Menschen führen[1]. Ein wichtiges Ziel der Software-Entwicklung muß also die Produktion von möglichst fehlerfreier Software sein.

Die Fehlerfreiheit von Software ist allerdings nur ein Merkmal, der die Software-Qualität bestimmt. Doch bevor auf diese Merkmale weiter eingegangen wird, soll

[1] Die THERAC-25 Vorfälle [Lev95] haben gezeigt, daß dieses leider kein theoretischer Worst Case Fall ist.

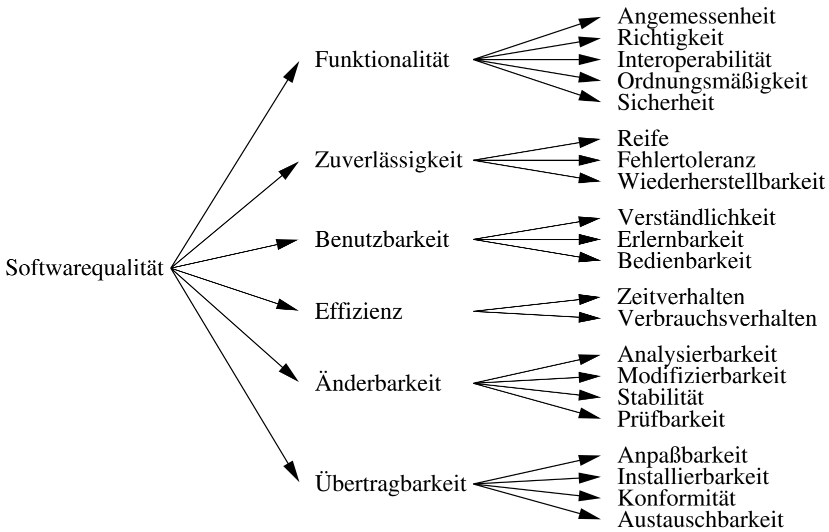


Abbildung 3.1: Qualitätsmodell nach [DIN66272]

der Begriff der Software-Qualität definiert werden. In dem vorliegendem Text soll Software-Qualität gemäß [DIN66272] folgendermaßen definiert werden:

> Unter *Software-Qualität* wird die Gesamtheit der Merkmale eines Software-Produkts verstanden, die sich auf dessen Eignung beziehen, festgelegte oder vorausgesetzte Erfordernisse zu erfüllen.

Software-Qualität ist also aus verschiedenen (Qualitäts-) Merkmalen zusammengesetzt, die es ermöglichen, die Qualität eines Software-Produkts zu beurteilen. Diese Qualitätsmerkmale sollten dabei für eine objektive Bewertbarkeit der Software-Qualität möglichst quantitativ bestimmbar sein. Dazu sind die Merkmale gegebenfalls selber in Teilmerkmale aufzuschlüsseln, die dann durch *Qualitätsindikatoren* [DIN66272], wie zum Beispiel Pfadlänge oder modularer Aufbau, bewertbar sind. Diese Unterteilung der Software-Qualität in verschiedene Merkmale und Teilmerkmale wird dabei als *Qualitätsmodell* bezeichnet. In [DIN66272] wird dazu ein Vorschlag für ein Qualitätsmodell in Form der Abbildung 3.1 gemacht.

Vor dem Beginn der Entwicklung eines Software-Produkts sollten genau definierte Qualitätsziele vorgegeben werden, die das endgültige Software-Produkt zu erfüllen hat. Diese Qualitätsziele sollten dann im Pflichtenheft der Software festgehalten werden, da sie sich sowohl auf die Kosten wie auch auf die Termine auswirken. Aus den Qualitätszielen ergeben sich Qualitätsanforderungen, die für jedes Qualitätsmerkmal die zu erreichende Güte angeben. Zur Erfüllung dieser Qualitätsanforderungen müssen dann geeignete Maßnahmen gewählt werden, die es den Entwicklern ermöglichen, die einzelnen Qualitätsanforderungen und schließlich auch das Qualitätsziel zu erreichen.

Die Maßnahmen des Qualitätsmanagement, mit denen diese Merkmale sichergestellt bzw. überprüft werden müssen, können in konstruktive und analytische Qualitätsmanagement-Maßnahmen unterschieden werden:

- Die *konstruktiven Qualitätsmanagement-Maßnahmen* stellen das Vorhandensein der oben genannten Merkmale während der Entwicklung sicher. Als ein Beispiel hierzu kann die objektorientierte Software-Entwicklung genannt werden, die die Änderbarkeit und die Übertragbarkeit der Software unterstützt.

- Bei den *analytischen Qualitätsmanagement-Maßnahmen* steht hingegen die nachträgliche Überprüfung von Qualitätsmerkmalen im Vordergrund. Es können dabei grob statische und dynamische Verfahren unterschieden werden. Den statischen Maßnahmen werden beispielsweise Inspektionen, Walk-Throughs und die formale Verifikation zugeordnet. Das wichtigste Verfahren unter den dynamischen Maßnahmen bildet das Testen von Software, das im nächsten Unterabschnitt ausführlicher behandelt wird.

  Klassifikationen der analytischen Qualitätsmanagement-Maßnahmen finden sich in [Lig92, Rie97].

3.2 Definition und Ziele des Testens

Das Testen von Software stellt das mit Sicherheit am häufigsten eingesetzte Prüfverfahren dar. In der vorliegenden Arbeit wird der Begriff des Testens von Software folgendermaßen definiert (vgl. [Gri95, Rie97]):

> Unter *Testen von Software* soll die systematische Ausführung eines Testgegenstandes zum Auffinden von Fehlern unter realen Einsatzbedingungen verstanden werden.

Der Testgegenstand, der in den Abschnitten 3.4.1 und 3.5.2 ausführlicher behandelt wird, kann dabei sowohl ein ausführbarer Teil des Software-Systems, als auch das gesamte Software-System selbst sein. Die charakteristische Eigenschaft des Testens ist also die Ausführung des zu testenden Programms in der realen Umgebung, wobei Besonderheiten des Zielrechners, des Compilers, des Betriebssystems und auch der Rechengenauigkeit das Ergebnis des Tests beeinflussen können. Diese Eigenschaft unterscheidet das Testen von allen anderen analytischen Qualitätsmanagement-Maßnahmen. Bei der Ausführung werden stichprobenartig Testeingaben gemacht, die ein Fehlverhalten des Programms auslösen sollen. Die Korrektheit eines Programms kann durch Tests allerdings nicht bewiesen werden, da dies die Eingabe von allen möglichen Eingabekombinationen erfordern würde. Im allgemeinen ist die Zahl der Testeingaben, die dazu erforderlich wären, sehr groß, so daß ein Korrektheitsbeweis praktisch nicht durchführbar ist. Das Ziel von Testverfahren muß also die Bestimmung einer Menge von Testeingaben sein, die repräsentativ, fehlersensitiv, redundanzarm und ökonomisch ist [Lig92].

3.3 Testablauf

Das Testen von Software besteht aus verschiedenen Aktivitäten mit unterschiedlichen Zielsetzungen. In der strukturierten Software-Entwicklung können grob folgende Aktivitäten unterschieden werden [Rie97] (vgl. [Gri95, PS94])[2]:

1. Testplanung
   Zur ersten Phase des Testens gehört insbesondere die Auswahl der Testmethoden, die zur Bestimmung der Testdaten eingesetzt werden sollen. Bei der Auswahl sind auf die Vor- und Nachteile der einzelnen Methoden zu achten und eine Auswahl so zu treffen, daß die Nachteile einer ausgewählten Methode durch die Vorteile einer anderen Methode kompensiert werden. Daraus resultiert, daß für einen Test in ausreichendem Maße sowohl funktionsorientierte als auch strukturorientierte Testmethoden, auf die in den nächsten Unterabschnitten eingegangen wird, genutzt werden müssen. Außerdem können zur Auswahl auch *Maßzahlen* [Rie97] herangezogen werden, die aus der Spezifikation oder der Programmstruktur abgeleitet werden können. Es können beispielsweise Maßzahlen bestimmt werden, die die Komplexität des Prüflings in Bezug auf den Kontrollfluß angeben. Da komplexe Programmstrukturen besonders fehleranfällig sind, sollten sie auch intensiver getestet werden.

   Zur Auswahl der Testmethoden gehört auch die Festlegung eines Abbruchkriteriums für den Test. Als Abbruchkriterium dient häufig eine bestimmte Testgüte [Rie97] (vgl. [PS94]), die aus benutzerorientierter, fehlerorientierter und struktureller Sicht bestimmt werden. Bei der *benutzerorientierten* Testgüte wird die Fehleraufdeckungsrate, also die Zahl der gefundenen Fehler pro Zeiteinheit, bestimmt und der Test beendet, wenn die Fehleraufdeckungsrate minimal geworden ist. Bei der *fehlerorientierten* Beurteilung der Testgüte hingegen werden entweder die Mutationsanalyse oder die Fehlereinpflanzung eingesetzt. Beide Verfahren führen eine gezielte Veränderung des Prüflings durch [Rie97]. Die *strukturelle* Bestimmung der Testgüte sieht genau dann ein Ende des Tests vor, wenn bei einem bestimmten Überdeckungsgrad bezüglich eines Testkriteriums keine Fehler gefunden werden können.

2. Testdatenerzeugung
   In dieser Phase wird die Erzeugung der Testdaten durchgeführt. Die Solldaten, das heißt die Ergebnisse bei korrekter Ausführung des Prüflings, werden

[2] Der hier vorgestellte Testablauf gilt mit einigen Veränderungen auch für die objektorientierte Software-Entwicklung. Siehe dazu [Sne96].

dabei aus einer Testreferenz, wie zum Beispiel der Spezifikation, gewonnen. Zu beachten ist hierbei allerdings, daß die Ausgabe des zu testenden Programms nicht nur von Eingabedaten abhängen kann, sondern auch vom inneren Zustand des Programms und unter Umständen von dem Inhalt einer Datenbank. Außerdem können auch Rechenungenauigkeiten das Ergebnis verfälschen, so daß anstatt eines einzelnen Sollergebnisses eine Menge angegeben werden muß, die das Sollergebnis darstellt.

3. Testrahmenerstellung
   Aufgabe dieser Phase ist die Erzeugung der für den Modultest notwendigen *Treiber* (*driver*) und *Stellvertreter* (*stubs*). Treiber ersetzen dabei übergeordnete Module, die die zu testenden Funktionen und Prozeduren aufrufen, während Stellvertreter die untergeordneten Module ersetzen. Der Testrahmen kann dabei so realisiert werden, daß die erforderlichen Daten direkt von der Testperson während der Testausführung interaktiv eingegeben werden können.

4. Testausführung
   Nachdem die notwendigen Aktivitäten zur Testvorbereitung abgeschlossen wurden, wird in dieser Phase der eigentliche Test durchgeführt. Dazu wird der Testgegenstand mit den vorher bestimmten Eingabedaten ausgeführt und die Ergebnisse in einem Testprotokoll festgehalten.

5. Testauswertung
   Die abschließende Testauswertung besteht zum einen aus der Bestimmung der Testgüte, die als Abbruchkriterium das Ende des Tests festsetzt, und zum anderen aus dem Vergleich der protokollierten Testergebnisse mit den Solldaten, um ein Fehlverhalten des Prüflings auf der Testdatenmenge festzustellen.

Nach der Durchführung des Tests folgt die Lokalisierung der Fehler, die ein Fehlverhalten des Prüflings verursacht haben. Anschließend werden die Fehler behoben und ein Regressionstest[3] durchgeführt. Die Lokalisierung und die Behebung der Fehler gehört allerdings nicht mehr zum Umfang des Testens. Falls das gewünschte Testziel, beispielsweise in Form einer bestimmten Testgüte, noch nicht erreicht wurde, wird der Test mit einer neuen Testdatenmenge wiederholt.

[3] Siehe Abschnitt 4.3.

3.4 Testen in der strukturierten Software-Entwicklung

3.4.1 Testgegenstände

Während der systematischen Software-Entwicklung entstehen verschiedene Zwischenergebnisse, die den Stand der Software-Entwicklung zu dem jeweiligem Zeitpunkt angeben und die in die nächste Phase der Software-Entwicklung als Ausgangsbasis eingehen. Es ist also klar, daß diese Zwischenergebnisse so früh wie möglich getestet werden sollten, um die Software-Entwicklung nicht auf einer fehlerhaften Basis fortzuführen, und um Fehlerfolgekosten minimal zu halten. Für den Test, wie er in dieser Diplomarbeit definiert wurde, können folgende Testgegenstände bei einer strukturierten Software-Entwicklung[4] identifiziert werden [Gri95]:

- Funktion
  Funktionen stellen während der Erstellung der Software die kleinsten, gesondert zu prüfenden Testgegenstände dar. Die Testreferenz bzw. die Testgrundlage sind dabei die Dokumente des Software-Feinentwurfs. Je nach der Kopplung der Funktionen untereinander und der Testreihenfolge ist ein geeigneter Testrahmen zu realisieren, der die aufrufenden und die aufgerufenen Funktionen simuliert.

- Modul
  Als die nächstgrößeren Testgegenstände ergeben sich die Module. Ein Modul stellt dabei eine Zusammenfassung von semantisch zusammengehörigen Funktionen mit einer definierten Schnittstelle nach außen dar. Module sollten zur Unterstützung der Wiederverwendbarkeit in anderen Projekten möglichst kontextunabhängig sein, das heißt ein Modul sollte möglichst unabhängig von anderen Modulen entwickelbar und testbar sein. Bei der Integration der einzelnen Funktionen zu einem Modul ist insbesondere auch das Zusammenwirken der Funktionen zu testen. Bei dem sogenannten *Integrationstest* sind vor allem die Schnittstellen der einzelnen Funktionen und die Verwendung von globalen Daten zu überprüfen. Ähnlich wie beim Funktionstest kann sich auch beim Test eines Moduls, abhängig davon, ob Funktionen aus anderen Modulen importiert werden und von der Reihenfolge, in der die Module getestet werden, die Notwendigkeit zu einer Testumgebung ergeben.

- Subsystem
  Subsysteme entstehen durch Zusammenfassen von mehreren Modulen zu einer Gruppe, wobei auch hier nach der Zusammenfassung ein Integrationstest durchgeführt werden muß. Subsysteme werden dabei gegen den System-Grobentwurf getestet. Auch beim Subsystemtest kann ein Testrahmen für das Testen notwendig sein.

[4]Eine Einführung in die strukturierte Software-Entwicklung findet sich in [PS94, Blz96].

- System
  Beim Systemtest wird das gesamte Software-System, das aus allen vorher getesteten Modulen besteht, einem Test unterzogen. Das System wird dabei gegen den Systementwurf und die Anforderungsspezifikation getestet. Damit hängt auch zusammen, daß der Test in der realen Umgebung durchgeführt werden sollte. Nach der Durchführung des Systemtests wird das Software-System dann dem Auftraggeber übergeben.

- Installiertes System
  Im letzten Schritt wird das System in der Einsatzumgebung installiert und vom Auftraggeber im Rahmen eines Abnahmetests gegen den zu Beginn der Software-Entwicklung ausgehandelten Anforderungskatalog getestet. Der Abnahmetest entspricht also einem Systemtest, der vom Auftraggeber anstelle des Entwicklers durchgeführt wird. Nach dem Abnahmetest beginnt üblicherweise die Garantiezeit und auch die Zahlungsverpflichtung des Auftraggebers.

3.4.2 Testmethoden

Im analytischen Qualitätsmanagement können verschiedene Klassen von Testmethoden unterschieden werden. Im folgenden wird eine Klassifikation von Testmethoden nach [Lig92] vorgestellt (vgl. [PS94, Rie97]). Die Beschreibung verschiedener Testmethoden findet sich in [PS94, Rie97].

White-, Black- und Grey-Box Methoden

Die Einteilung in White- und Black-Box Methoden bildet sich durch die Unterscheidung der Testmethoden nach der benötigten Information bezüglich der Implementierung des Prüflings. Bei White-Box Methoden basiert der Test auf der Implementierung, während Black-Box Methoden unabhängig von der Implementierung des Prüflings die Bestimmung der Testdaten durchführen. Allerdings existieren Testmethoden, die nicht eindeutig einer dieser beiden Klassen zugeordnet werden können. Diese Methoden werden dann als Grey-Box Methoden bezeichnet.

Beispiele:

White-Box Methoden	:	Kontrollflußorientierter Test, Pertubationentest
Grey-Box Methoden	:	Grenzwertanalyse, Back-to-Back-Test
Black-Box Methoden	:	funktionale Äquivalenzklassenbildung, Zufallstest.

Struktur-, funktionsorientierte und diversifizierende Methoden

Werden Testmethoden nach der Testreferenz, gegen den der Prüfling getestet wird, gruppiert, so können struktur-, funktionsorientierte und diversifizierende Methoden unterschieden werden. Die strukturorientierten Methoden testen den Prüfling gegen die Programmstruktur und sind somit auch White-Box Methoden. Dagegen führen die funktionsorientierten Methoden einen Test des Prüflings gegen die Spezifikation durch und sind nach der obigen Unterscheidung den Black-Box Methoden gleichzusetzen. Es soll allerdings betont werden, daß Black-Box Methoden nicht umgekehrt den funktionsorientierten Methoden entsprechen müssen. Als ein Beispiel dazu kann der Zufallstest dienen. Der Zufallstest ist zwar eine Black-Box Methode aber nicht funktionsorientiert, da er nicht auf der Spezifikation basiert. Bei diversifizierenden Methoden werden verschiedene Versionen, die unter Umständen von unterschiedlichen Gruppen realisiert werden, gegeneinander getestet.

Beispiele:

Strukturorientierte Methoden	:	Kontrollflußorientierter Test, Pertubationentest
Funktionsorientierte Methoden	:	Äquivalenzklassenbildung, Partitionsanalyse
Diversifizierende Methoden	:	Pertubationentest, Back-to-Back-Test.

Fehlerorientierte Methoden

Bei den fehlerorientierten Methoden basiert die Auswahl der Testdaten auf bestimmten Annahmen bezüglich der Fehler. Die Testdaten werden dabei so bestimmt, daß sie diejenigen Bereiche des Testgegenstands ausführen, die bei ähnlichen Testgegenständen besonders fehleranfällig waren.

Beispiele:

Pertubationentest, Grenzwertanalyse.

Äquivalenzklassenbildner

Äquivalenzklassenbildende Methoden unterteilen den Eingabebereich des Prüflings in verschiedene Äquivalenzklassen. Diese Methoden können zusätzlich danach unterschieden werden, ob sie die Herleitung der Äquivalenzklassen basierend auf der Spezifikation oder basierend auf der Struktur des Prüflings durchführen.

Beispiele:

Äquivalenzklassenbildung anhand

Implementierung	:	Pfadbereichstest
Spezifikation	:	funktionale Äquivalenzklassenbildung
Implementierung und Spezifikation	:	Partitionsanalyse.

Basis- und höhere Testmethoden

Bei den Basismethoden handelt es sich um Testmethoden, die nicht weiter in andere Basismethoden zerlegt werden können, während höhere Testmethoden aus mehreren

Basismethoden zusammengesetzt sind.

Beispiele:

Basismethoden	:	Back-to-Back-Test
Höhere Testmethoden	:	Pfadbereichstest.

3.4.3 Beispiele für Testmethoden

An dieser Stelle werden zwei Testmethoden beschrieben, um zum einen die Arbeitsweise von Testmethoden zu demonstrieren, und zum anderen die Unterschiede von funktions- und strukturorientierten Testmethoden zu zeigen.

Funktionale Äquivalenzklassenbildung

Als ein einfaches Beispiel für das funktionsorientierte Testen wird die Methode der funktionalen Äquivalenzklassenbildung [Lig92, PS94, Rie97] angegeben. Bei der Äquivalenzklassenbildung werden ausgehend von einer informellen Spezifikation Bedingungen für die einzelnen Variablen der Eingabe aufgestellt, welche die Eingabewerte entsprechend ihren Datenbereichen als gültig bzw. als ungültig klassifizieren. Eine Eingabebedingung induziert also eine Äquivalenzrelation mit zwei Äquivalenzklassen. Während eine der Äquivalenzklassen alle Eingabewerte enthält, die als Normalfall behandelt werden, enthält die andere Klasse Werte, die falsche Eingaben darstellen und somit eine Fehlerbehandlung erfordern. Nach der Bestimmung der Äquivalenzklassen werden anschließend die Testdaten ausgewählt. In einem ersten Schritt werden die Testdaten so ausgewählt, daß sie möglichst viele Äquivalenzklassen überdecken, die gültige Eingabewerte beinhalten. Dabei überdeckt eine Eingabe genau dann eine Äquivalenzklasse, wenn eine Komponente der Eingabe Element der Klasse ist. Hierbei werden solange Eingaben ausgewählt, bis alle gültigen Äquivalenzklassen berücksichtigt wurden. Anschließend werden im nächsten Schritt Eingaben bestimmt, die jeweils genau eine ungültige Klasse überdecken, da die Verarbeitung aller ungültigen Eingabewerte durch die Möglichkeit eines Abbruch, der wiederum von einem ungültigen Eingabewert verursacht werden kann, nicht sichergestellt ist.

Kontrollflußorientierter Test

Als ein Beispiel zum strukturorientierten Software-Test wird an dieser Stelle das kontrollflußbezogene Testen [Rie97] beschrieben. Beim kontrollflußbezogenen Testen werden die Testmuster so erzeugt, daß ein bestimmtes Überdeckungskriterium erfüllt wird. Ausgehend von dem Kontrollflußgraphen legt ein Überdeckungskriterium diejenigen Knoten und Kanten des Kontrollflußgraphen fest, die die zu erzeugenden Testdaten ausführen müssen. Das Ideal in diesem Zusammenhang wäre die sogenannte *Pfadüberdeckung*. Dabei werden diejenigen Testdaten erzeugt, die alle Pfade ausführen, die vom Anfangsknoten zum Endknoten des Kontrollflußgraphen führen. Auf diese Weise könnten alle Fehler, die sich auf den Kontrollfluß auswirken bzw. die zu einer falschen Berechnung führen, gefunden werden. Dies würde aber bedeuten, daß die Zahl der Testdaten sehr groß werden könnte, da die Zahl der Pfade exponentiell mit der Zahl der Verzweigungen im Kontrollflußgraphen zunimmt. Ein zusätzliches Problem stellen Schleifen ohne obere Grenze dar. Bei diesen Schleifen kann die Zahl der Wege vom Anfangsknoten zum Endknoten unendlich sein. Es wird deutlich, daß schwächere Überdeckungskriterien verwendet werden müssen. Werden bestimmte Annahmen über die Art der Fehler in dem zu testenden Programm gemacht, kann beim kontrollflußbezogenen Testen beispielsweise auch das Kriterium der *Anweisungsüberdeckung* oder der *Zweigüberdeckung* [Rie97] verwendet werden.

3.5 Testen in der objektorientierten Software-Entwicklung

3.5.1 Besonderheiten beim Test objektorientierter Systeme

Obwohl die ersten objektorientierten Programmiersprachen schon in den sechziger Jahren entwickelt wurden, blieb der Test von objektorientierter Software von der Forschung bis noch vor wenigen Jahren unberücksichtigt. Es herrschte die Meinung, daß der Test von objektorientierter Software mit herkömmlichen Testmethoden durchgeführt werden kann und daß die Objektorientierung zu einer wesentlichen Vereinfachung des Testens führt [RBP+93]. In den letzten Jahren wurde jedoch erkannt, daß diese Meinung nicht zutreffend ist und daß neue Teststrategien und -methoden für den objektorientierten Test entwickelt werden müssen [Rüp97].

Objektorientierte Systeme besitzen einige Besonderheiten, die den Test besonders erschweren. Dazu gehören vor allem die objektorientierten Grundprinzipien[5] der Datenkapselung, Vererbung und des Polymorphismus. Es existieren aber auch noch andere Besonderheiten der Objektorientierung, die sich auf den Test auswirken. Ein Beispiel dafür ist die Wiederverwendbarkeit, die durch die Objektorientierung explizit unterstützt wird. Klassen, die in ähnlichen Projekten wiederverwendet werden sollen, müssen intensiver getestet werden, um das fehlerfreie Funktionieren auch in einem anderen als dem in der Entwicklung vorgesehenem Kontext zu garantieren.

In den nachfolgenden Ausführungen werden die Auswirkungen der objektorientierten Grundprinzipien auf das Testen erläutert [BS94, Win97] (vgl. [Sne95, Rüp97]):

[5]Zur Definition der Begriffe aus der Objektorientierung siehe [Blz96, FS97].

Datenkapselung

Eine Klasse kapselt Variablen und Funktionen ein, die im objektorientiertem Paradigma als *Attribute* und *Methoden* der Klasse bezeichnet werden. Die Werte der Attribute stellen dabei zu einem bestimmten Zeitpunkt den Zustand des Objekts dar. Der Objektzustand ist durch die Datenkapselung von außen nicht direkt sichtbar, d.h. die Datenkapselung erschwert das Testen, da der Objektzustand nicht beobachtbar ist und somit auch nicht auf Konsistenz überprüft werden kann. Dieses Problem kann auf verschiedene Arten gelöst werden:

- Eine offensichtliche Lösung besteht in dem Erweitern der Klasse um einzelne Befehle oder auch um Methoden, die den momentanen Objektzustand nach außen sichtbar machen. Nachteil dieser Lösung ist allerdings, daß dieses Erweitern der Klasse einen zusätzlichen Aufwand verursacht.
- Der Zustand der zu testenden Klasse kann aber auch indirekt über eine Unterklasse geprüft werden, indem Befehle bzw. Methoden, die den Zustand sichtbar machen, durch Vererbung dem Nachfolger hinzugefügt werden. Allerdings hat auch dieser Ansatz seine Nachteile. Es ist nämlich möglich, daß die Unterklasse keinen vollen Zugriff auf die Attribute der Oberklasse besitzt, wie zum Beispiel `private` Attribute in der C++ Programmiersprache.
- Es sind aber auch andere, von der Programmiersprache abhängige Lösungen möglich. Als ein Beispiel soll dabei das `friend` Konstrukt in C++ genannt werden, der einer Klasse den vollen Zugriff auf die Attribute einer anderen Klasse, die auch als `private` deklariert sein können, ermöglicht.

In diesem Zusammenhang soll auch ein Problem beim Testen von nicht instanziierbaren Klassen erläutert werden. Vor der Überprüfung der verschiedenen Zustände, die ein Objekt annehmen kann, müssen die zu testenden Objekte erzeugt werden. Es sind aber Fälle möglich, wo diese Erzeugung, also die sogenannte *Instanziierung*, nicht möglich ist, wie zum Beispiel bei abstrakten, generischen und Mix-In Klassen. Der Test dieser Klassen erfordert einen Testrahmen, der alle möglichen Ausprägungen, welche die zugehörigen Objekte besitzen können, berücksichtigt.

Vererbung

Durch die Vererbung können Eigenschaften, d.h. Attribute und Methoden, einer oder mehrerer Klassen an eine Klasse weitergegeben werden. Auf dem ersten Blick scheint der Test von Methoden, die von einer erfolgreich getesteten Klasse geerbt wurden, nicht notwendig zu sein. Dabei wird allerdings die Tatsache übersehen, daß die geerbten Methoden in Zuständen aufgerufen werden können, die in der Oberklasse nicht eintreten konnten und deshalb auch beim Entwurf der Methoden nicht berücksichtigt wurden. Es ist also möglich, daß die geerbten Methoden sich bei diesen neuen

Zuständen undefiniert verhalten. Aber auch bei Berücksichtigung dieser Zustände beim Entwurf ist ein Test notwendig, da diese Zustände eine Ausführung von Abschnitten verursachen können, die im Kontext der Oberklasse nicht erreichbar waren. Da aber Erreichbarkeit ein notwendiges Kriterium für den Test ist, muß ein erneuter Test dieser Methode durchgeführt werden, um die Korrektheit der geerbten Methoden auch im Kontext der Unterklasse sicherzustellen.

Bei den nicht-strikten Formen der Vererbung ist die Notwendigkeit eines Tests von geerbten Methoden offensichtlicher als bei dem obigem Fall der strikten Vererbung. Es sollte klar sein, daß Methoden, die bei der Vererbung eine Veränderung erfahren haben, einem Test unterzogen werden müssen. Es müssen danach aber auch alle Methoden getestet werden, die die veränderte Methode zur Erfüllung der eigenen Aufgabe benutzen. Diese Methoden können sowohl geerbte als auch neu in den unteren Schichten der Vererbungshierachie definierte Methoden sein.

Bei der mehrfachen Vererbung treten noch spezielle Probleme zusätzlich zu den oben erwähnten auf, die zu testen sind. Liegt der Fall vor, daß eine Klasse von mehreren anderen Klassen Methoden erbt, die denselben Namen und dieselbe Signatur besitzen, so besteht eine Mehrdeutigkeit, die entsprechend aufgelöst werden muß. Diese Mehrdeutigkeit kann zum einen durch den Entwickler gelöst werden, indem er explizit die zu vererbende Methode angibt oder zum anderen durch die Programmiersprache, die beispielsweise durch die Deklarationsreihenfolge die zu vererbende Methode auswählt. In beiden Fällen ist zu testen, ob die richtige Methode vererbt wurde, die auch vorgesehen war.

Polymorphismus

Durch Polymorphismus können verschiedene Objekte auf dieselbe Nachricht verschieden reagieren. Dieser Mechanismus wird insbesondere in Verbindung mit der Vererbung eingesetzt. Polymorphismus erschwert durch die dynamische Bindung den Testprozeß, da dadurch nur zur Laufzeit festgestellt werden kann, welches Objekt für einen polymorphen Klassenbezeichner eingesetzt wird und welche Implementierung für eine Methode tatsächlich ausgeführt wird.

Als eine Lösung zu diesem Problem können für den funktionalen Test Bedingungen angegeben werden, die die verschiedenen Implementierungen für eine polymorphe Methode erfüllen müssen. Anhand dieser Bedingungen können dann aus der Spezifikation Testdaten erzeugt werden, mit denen alle Versionen der polymorphen Methode getestet werden können.

Ähnliche Probleme ergeben sich beim Test von Methoden mit polymorphen Parametern. Die Testdaten müssen hierbei so bestimmt werden, daß möglichst viele verschiedene Ausprägungen, die die polymorphen Parameter annehmen können, berücksichtigt werden. Die Schwierigkeit dabei besteht in der Bestimmung einer möglichst repräsentativen Menge von Objekten, die für die polymorphen Parameter eingesetzt werden sollen. Die Menge der möglichen Objekte ist dabei im Prinzip unendlich groß. Die Vererbungshierachie kann beliebig erweitert werden, wobei alle Objekte in der Vererbungshierarchie als mögliche Eingabewerte dienen können.

3.5.2 Testgegenstände

In diesem Abschnitt werden die verschiedenen Testgegenstände in der objektorientierten Software-Entwicklung in analoger Weise zu Abschnitt 3.4.1 aufgelistet [BS94, Rüp97, Win97] (vgl. [GDT93]). Diese Auflistung soll die Unterschiede zwischen der strukturierten und der objektorientierten Software-Entwicklung in Bezug auf das Testen verdeutlichen.

- Methode
  Den Funktionen in der strukturierten Software-Entwicklung stehen Methoden in der objektorientierten Software-Entwicklung gegenüber. Allerdings wird die Methode im Gegensatz zur Funktion nicht als die kleinste unabhängig testbare Einheit im Software-Prozeß betrachtet, da die Methoden einer Klasse durch gegenseitige Aufrufe und durch gemeinsam verwendete Instanzvariable, d.h. durch den Objektzustand, stark miteinander gekoppelt sind. Der Test einer Methode erfordert die Instanziierung der entsprechenden Klasse oder zumindest die Simulation der Klasseneigenschaften, die sich auf das Verhalten der Methode auswirken. Der Aufwand für den Test von Methoden ist durch den Testrahmen im allgemeinen höher als der Aufwand für den Test von Funktionen, obwohl Methoden üblicherweise eine einfachere Kontrollstruktur besitzen als Funktionen.

- Klasse
  Da der Test einer Methode mit einem relativ hohen Aufwand verbunden ist, wird häufig die Klasse als die kleinste unabhängig testbare Einheit im Software-Prozeß betrachtet. Klassen entsprechen den Modulen in der strukturierten Software-Entwicklung (vgl. [Lig95]), wobei in der objektorientierten Software-Entwicklung zusätzlich zur Kapselung die Mechanismen der Vererbung und des Polymorphismus zur Verfügung stehen. Durch diese Mechanismen ist allerdings der Zusammenhang zwischen Kontrollfluß und der Struktur des Quelltextes nicht mehr so offensichtlich wie bei strukturierter Software. Es stellt sich also die Frage, inwieweit kontrollflußorientierte Testmethoden aus der klassischen Software-Erstellung überhaupt noch anwendbar sind. Hinzu kommt, daß durch eine Klasse keine Reihenfolge vorgegeben wird, in der die Methoden der Klasse aufgerufen werden können. Diese Tatsache erschwert zusätzlich die Anwendbarkeit kontrollflußorientierter Verfahren. Zum Klassentest existieren verschiedene Verfahren, auf die im Abschnitt 3.5.4 eingegangen wird.

- Subsystem
  Im objektorientiertem Software-Prozeß entstehen Subsysteme durch das Zusammenfassen von Klassen zu sogenannten *Clustern*, wobei Cluster so gebildet werden sollten, daß sie möglichst einfach wiederverwendet werden können. Ein Cluster von Objekten kann nur dann eine übergeordnete Aufgabe erfüllen, wenn die Objekte, aus denen der Cluster zusammengesetzt ist, auch korrekt interagieren. Nach einem Test der einzelnen Klassen muß also ein Test durchgeführt werden, der die Integration der Klassen überprüft, aus denen der Cluster zusammengesetzt ist. Verfahren zum Integrationstest werden im Abschnitt 3.5.4 erläutert.

- System und installiertes System
  Beim Systemtest und beim Abnahmetest, d.h. beim Test des installierten Systems durch den Auftraggeber, bleiben die Verfahren aus der strukturierten Software-Entwicklung im wesentlichen gültig, da die objektorientierte Implementierung des Systems für den Tester nicht sichtbar ist. Allerdings müssen für den Test die Dokumente der objektorientierten Analyse als Testreferenz herangezogen werden, wie zum Beispiel die Dokumente zur Analyse durch *Nutzungsfälle* (*Use Cases*) [FS97].

3.5.3 Testmethoden

Die herkömmlichen Testmethoden, die im Rahmen der Qualitätssicherung von strukturierter Software eingesetzt werden können, haben in der objektorientierten Software-Entwickung nur eine begrenzte Anwendbarkeit. Zur ausreichenden Qualitätssicherung objektorientierter Software müssen neue Verfahren entwickelt werden, die die Besonderheiten der Objektorientierung entsprechend berücksichtigen. Diese Tatsache wurde aber erst in den letzten Jahren erkannt, so daß die Mehrzahl der Veröffentlichungen zum Thema objektorientierte Testmethoden auch in den letzten fünf Jahren erschienen ist.

Objektorientierte Testmethoden können nach [Sne96] in die drei Gruppen regelbasierter, nachrichtenbasierter und nutzungsbasierter Test unterteilt werden, wobei diese Unterteilung analog zur Unterteilung der herkömmlichen Testmethoden in die Gruppen White-, Black- und Grey-Box Methoden durchgeführt wird.

Regelbasierter Test

Der regelbasierte Test von objektorientierter Software entspricht einem White-Box Test, d.h. zur Generierung der Testdaten wird die Implementierung des Testgegenstandes herangezogen. Bei objektorientierter Software wird also entsprechend die

Entscheidungslogik der Methoden untersucht und Testdaten so ausgewählt, daß bestimmte Überdeckungskriterien erfüllt werden. Regelbasierte Testmethoden eignen sich also insbesondere für den Test von Klassen.

Nachrichtenbasierter Test

Der nachrichtenbasierte Test entspricht hingegen einem Black-Box Test. Dabei werden representative Eingangsnachrichten erzeugt und anschließend die Ausgangsnachrichten des zu testenden Systems validiert. Bei nachrichtenbasierten Testmethoden steht das Verhalten eines objektorientierten Systems, also insbesondere auch eines einzelnen Objekts, nach außen im Mittelpunkt. Diese Testmethoden können beim Integrationstest eingesetzt werden.

Nutzungsbasierter Test

Nutzungsbasierte Testmethoden zielen auf die Überprüfung der spezifizierten Nutzungsfälle ab, d.h. der nutzungsbasierte Test entspricht einem funktionalem Test gegen die Spezifikation. Beim nutzungsbasiertem Testen wird also nicht die Korrektheit der internen Abläufe im System sichergestellt, sondern die Einhaltung der Spezifikation überprüft. Dazu zählt neben den Ausgaben auf dem Bildschirm auch der Zustand von Datenbanken und persistenten Objekten. Nutzungsbasierte Testmethoden eignen sich also für den System- bzw. Abnahmetest.

3.5.4 Klassen- und Integrationstest

Klassentest

Der Klassentest besteht zum einen aus dem Test von Methoden, wobei der Test aller Methoden eher die Ausnahme sein wird, da Methoden im allgemeinen eine relativ einfache Kontrollstruktur besitzen. Außerdem sollten auch, insbesondere bei der Nutzung einer Klassenbibliothek, nur diejenigen Methoden getestet werden, die auch in der zu entwickelnden Software benötigt werden, um den Testaufwand so gering wie möglich zu halten. Diese Strategie stellt allerdings wiederum die Anwendbarkeit kontrollflußorientierter Überdeckungskriterien in Frage [Sne95, Sne96].

Zum anderen muß beim Klassentest aber auch das Zusammenspiel der Methoden validiert werden. Der Schwerpunkt beim Klassentest sollte auf diesem Methoden-Integrationstest liegen, da die Komplexität objektorientierter Systeme durch die Interaktion einfacher Methoden entsteht [Sne95].

Der Klassentest umfaßt aber auch noch zusätzlich zu den oben genannten zwei Punkten noch den Integrationstest von Objekten, die derselben Klasse zugehören und den Test der Klasse in Verbindung mit der Vererbung [Ove93].

Der Test einer Klasse erfordert im ersten Schritt die Erzeugung eines Objekts, d.h. die Instanziierung der Klasse, da Klassen nur die Struktur für Objekte angeben und deshalb selbst nicht ausführbar sind[6]. In diesem Zusammenhang existieren zwei Fragestellungen [Rüp97]:

1. *Was ist ein Testfall?*
   Ein Testfall besteht in der Regel aus einer Nachrichtensequenz, d.h. zum Testen eines Objekts wird der Ist-Zustand des Objekts vor und nach der Nachrichtensequenz mit dem Soll-Zustand verglichen. Ein Problem dabei kann die *Fehlermaskierung* darstellen [Rüp97]. Ein Fehlverhalten einer Methode wird durch die Ausführung nachfolgender Methoden verdeckt. Das Problem der Fehlermaskierung wird dabei durch die Datenkapselung verschärft, da diese, wie schon oben beschrieben, die Beobachtbarkeit des Zustands erschwert bzw. verhindert.

2. *Wieviele Testobjekte müssen erzeugt werden?*
   In [MK94] wird die Erzeugung eines Objekts für jeden Testfall vorgeschlagen, da dadurch Fehler in einem Testfall nicht zu Folgefehlern führen können und Testfälle einfacher aus der Menge der Testfälle gelöscht bzw. auch hinzugefügt werden können. Die Testfälle sind also unabhängig voneinander. In [Pel96] wird darauf hingewiesen, daß beim Testen in Verbindung mit `static` Variablen in C++ mehrere Objekte derselben Klasse notwendig sein können.

Beim Klassentest haben sich unterschiedliche Ansätze als erfolgversprechend erwiesen. Während einige Autoren endliche Automaten beim Klassentest einsetzen, schlagen andere Autoren Verfahren vor, die auf formalen Spezifikationen basieren:

- Endliche Automaten
  Endliche Automaten werden in der objektorientierten Software-Entwicklung häufig zur Modellierung von Objekten genutzt, wobei die Zustände eines Automaten die verschiedenen Objektzustände und die Transformationen die Reaktion des Objekts auf entsprechende Nachrichten darstellen. Zur Erstellung von Automaten zu komplexen Objekten kann dabei [SC95] herangezogen werden, wo ein Verfahren zur Konstruktion von komplexen Automaten aus einfacheren Automaten beschrieben wird. Zur Generierung von Testfällen aus endlichen Automaten sei auf [TR93a, TR93b, TR93c, KSG+94, BS95, HKC95, MCM+98] verwiesen.

- Formale Spezifikationen
  Formale Spezifikationen können ebenfalls zum Test von Klassen verwendet werden, wobei jedoch der Aufwand zur Generierung der formalen Spezifikation nicht unberücksichtigt bleiben sollte. Zur Anwendung formaler Spezifikationen beim Klassentest sei auf [DF94, TX96, BBP96, CTC+98] verwiesen.

[6] So gesehen ist die Bezeichung „Klassentest" paradox, da die Ausführbarkeit ein notwendiges Kriterium für den Test ist.

Integrationstest

Nach dem erfolgreichen Test der einzelnen Klassen muß anschließend das korrekte Zusammenspiel der Klassen untereinander getestet werden. Der objektorientierte Integrationstest bezieht sich dabei nur auf Klassen, die nicht in einer Vererbungsbeziehung stehen. Dieses Zusammenspiel muß dabei nicht unbedingt hierarchisch sein, da sich auch zwei Objekte gegenseitig Nachricht schicken können. Es müssen also neue Integrationsstrategien entwickelt werden, die auch die Vererbung und die dynamische Bindung berücksichtigen [Rüp97]. Zum Integrationstest wurden verschiedene Verfahren vorgeschlagen [TR93a, TR93b, Ove93, Ove96].

4 Ausgangsbasis des Testverfahrens

In diesem Kapitel werden die Grundlagen des entwickelten Testverfahrens für die prototypbasierte Software-Entwicklung erläutert. Dazu werden nach einer kurzen Auflistung der Anforderungen an Testverfahren, die sich durch das spezielle Umfeld des Investmentbankings ergeben, in in Abschnitt 4.1, Strategien für den Prototypentest beschrieben. Aus den Schwächen zweier einfacher Teststrategien wird in Abschnitt 4.2 auf eine optimale Strategie geschlossen, die auf der Basis eines Regressionstestverfahrens umgesetzt werden kann. Anschließend werden die Grundlagen des Regressionstests und eine Klassifikation der Verfahren in Abschnitt 4.3.1 erläutert. Das Verfahren von ROTHERMEL/HARROLD wird in Abschnitt 4.3.2 ausführlicher beschrieben, da es die Basis für den Prototypentest darstellt. Da dieses Verfahren jedoch rein strukturell ist, werden in Abschnitt 4.4 die Verfahren von TSE/XU und HONG/KWON/CHA für den funktionalen Klassentest beschrieben. Diese Verfahren werden im nächsten Abschnitt der Diplomarbeit in das Verfahren von ROTHERMEL/HARROLD integriert.

4.1 Anforderungen an Testverfahren im Investmentbanking

Das Investmentbanking stellt, wie schon in Kapitel 1 angedeutet, spezielle Anforderungen an die Software-Entwicklung, die durch entsprechende Maßnahmen im Software-Prozeß berücksichtigt werden müssen. Diese Maßnahmen umfassen auch geeignete Testverfahren, die folgende Merkmale der Software-Entwicklung unterstützen müssen:

1. Prototyping
   Die wichtigste Eigenschaft des Prototypings ist ohne Zweifel die inkrementelle Erstellung des Prototyps, d.h. der Prototyp wird ausgehend von einer ersten Version iterativ zum Endprodukt weiterentwickelt. In jeder dieser Iterationen sollte der Prototyp einem Test unterzogen werden, wobei beachtet werden sollte, daß im allgemeinen in den einzelnen Iterationen bestimmte Abschnitte der Spezifikation oder der Implementierung unverändert verbleiben und somit unter Umständen nicht erneut getestet werden müssen.

2. Objektorientierung
   Die Software-Entwicklung in dem speziellen Umfeld des Investmentbankings erfolgt nach dem objektorientierten Paradigma. Testverfahren müssen also die Objektorientierung explizit unterstützen, wobei die schon oben erwähnten Besonderheiten der Objektorientierung beachtet werden müssen.

3. Zuverlässigkeit
   Testverfahren sollten zuverlässig sein, d.h. sie sollten möglichst viele Fehler aufdecken. Allerdings resultiert diese Anforderung häufig in einer relativ großen Testdatenmenge, d.h. der Testaufwand steigt stark an. Da aber Fehler in der Software-Unterstützung für das Investmentbanking hohe wirtschaftliche Verluste verursachen können, sollte der Testaufwand als Entscheidungskriterium nicht allzustark gewichtet sein.

4.2 Teststrategien in der prototypbasierten Software-Entwicklung

4.2.1 Einfache Strategien

Die besonderen Anforderungen der prototypbasierten Software-Entwicklung an den Testprozeß wurden bislang kaum untersucht. Es existieren nach bestem Wissen des Autors keine Quellen in der Literatur oder im Internet, in denen konkrete Testverfahren für Prototypen bzw. für die evolutionäre Software-Entwicklung vorgeschlagen werden.

Für den Prototypentest sind zwei einfache Strategien naheliegend, die jedoch beide Schwächen besitzen, die deren Eignung für den Test in Frage stellen.

1. In der ersten Strategie werden die einzelnen Prototypen, die während des Prototypings entstehen, als isolierte Software-Produkte betrachtet und auch dementsprechend ohne eine Berücksichtigung der Tests von vorherigen Prototypen isoliert getestet. Diese Vorgehensweise führt allerdings zu einer *Ressourcenverschwendung*, da, wie schon oben angedeutet, die verschiedenen Prototypen gemeinsame Eigenschaften haben können, die nicht bei jedem Prototyp erneut validiert werden müssen.

2. Die zweite Strategie sieht hingegen nur einen Test der veränderten Abschnitte in der Implementierung bzw. in der Spezifikation vor. Allerdings ist auch diese komplementäre Strategie problematisch. Diese Vorgehensweise ist zumindest beim strukturellen Test nicht ausreichend, da durch *Kontroll-* bzw. *Datenflußabhängigkeiten* Fehler in den veränderten Abschnitten zu Fehlern in den nicht veränderten Abschnitten führen können. In bestimmten Fällen fällt also auch der Test von nicht veränderten Programmabschnitten an.

4.2.2 Optimale Strategie

Eine optimale Strategie muß also neben dem Test der Veränderungen, die sowohl die Implementierung als auch die Spezifikation betreffen können, auch den Test derjenigen Teile der Implementierung vorsehen, die durch die Veränderungen nur indirekt über Kontroll- bzw. Datenflußabhängigkeiten betroffen sind. Beim funktionalen Test sollte eine Überprüfung der Veränderungen in der Spezifikation ausreichend sein, wobei vorausgesetzt werden muß, daß die Spezifikation durch Veränderungen nicht inkonsistent geworden ist. Die Korrektheit der Spezifikation wird in den folgenden Ausführungen als gegeben betrachtet.

Das Verfahren, das in dieser Arbeit vorgeschlagen wird, basiert auf einem Algorithmus zum Regressionstest von Klassen in objektorientierten Programmen. Bei dem hier vorgeschlagenen Verfahren wird die fehlende Funktionalität des Prototyps als ein Fehler betrachtet, der schrittweise in den Prototyping-Iterationen behoben wird. Nach jeder „Korrektur“ des Prototyps wird dann ein Regressionstest durchgeführt. Diese Interpretation wird damit auch den Zielen des Regressionstests gerecht.

Es existieren jedoch einige Punkte, die bei der Verwendung von Regressionstesttechniken beim Testen von Prototypen beachtet werden müssen. Dazu gehört, daß der Nutzen der Selective-Retest Strategie[1] sich insbesondere bei relativ stabilen Software-Systemen zeigt, d.h. wenn die Software-Entwicklung sich in einem fortgeschrittenen Stadium befindet und die Veränderungen an der Software, die zu testen sind, nur gering sind. Bei einem Prototyp können die zu testenden Veränderungen jedoch größer sein, so daß sich die hier vorgeschlagene Strategie sehr der ersten Strategie nähert. Die Veränderungen an dem Prototypen sollten in den einzelnen Prototyping-Iterationen also möglichst klein gehalten werden. Ein weiterer Punkt ist aber auch die Tatsache, daß der Regressionstest einen zusätzlichen Aufwand verursacht. Es muß hierbei eine Testdatenmenge mit zusätzlichen Informationen bezüglich der überdeckten Anweisungen des Programms durch die einzelnen Testeingaben[2] bereitgestellt werden. Dieser zusätzlicher Aufwand könnte aber möglicherweise den Zeitgewinn beim Rapid Prototyping wieder zunichte machen. Es ist aber auch möglich, daß diese zusätzlichen Informationen durch die Verwendung von Wegwerf-Prototypen nicht verfügbar ist.

Das in dieser Diplomarbeit vorgeschlagene Testverfahren für Prototypen basiert auf einem Verfahren von ROTHERMEL/HARROLD [RH94b, RH94c], da es ein effizient zu implementierendes, zuverlässiges Verfahren für den Regressionstest von objektorientierten Programmen darstellt. Dieses Verfahren kann zwar werkzeugunterstützt durchgeführt werden, berücksichtigt aber keine Veränderungen der Spezifikation, da es nur auf der Implementierung basiert. Ein rein struktureller Test ist allerdings bei der Software-Entwicklung nicht ausreichend, so daß zusätzlich ein funktionaler Test durchgeführt werden muß. Der funktionale Test basiert hierbei auf den Methoden von TSE/XU [TX96, TX95] und HONG/KWON/CHA [HKC95].

[1] Siehe Abschnitt 4.3.1.
[2] Vergleiche *Test-History* aus Abschnitt 4.3.1.

4.3 Regressionstest

4.3.1 Grundlagen des Regressionstests

Nach der Korrektur eines Programms müssen *Regressionstests* durchgeführt werden, um sicherzustellen, daß die Korrekturen richtig erfolgt sind und daß sie keinen Einfluß auf das richtige Funktionieren der unveränderten Teile des getesteten Programms haben. Dabei entfällt häufig, solange nur Fehler in der Implementierung entfernt werden, die Phase der Testdatengenerierung, da Testdaten schon von vorhergehenden Tests zur Verfügung stehen.

Üblicherweise besteht der Regressionstest aus den folgenden fünf Phasen [RH94b, RH97, GHK^{+}98]:

1. Zu Beginn des Regressionstests müssen die Veränderungen erkannt werden, die an dem Programm zur Korrektur durchgeführt wurden.

2. Im nächsten Schritt wird die Untermenge der Testdatenmenge bestimmt, die zum erneuten Test des Programms verwendet werden soll.

3. Anschließend erfolgt die Testausführung, d.h. das Programm wird mit der oben bestimmten Untermenge der Testdaten getestet.

4. Falls erforderlich müssen in dieser Phase neue Testdaten generiert werden, um beispielsweise ein bestimmtes Überdeckungskriterium zu erfüllen.

5. Anschließend muß der Test mit den neu generierten Testdaten erneut durchgeführt werden.

6. Die Testdaten werden in der letzten Phase für spätere Regressionstests aufbereitet, wie zum Beispiel in der Form einer *Test-History*, die angibt, welches Testdatum welchen Pfad im Programm ausgeführt hat.

Die Auswahl der Testdaten in der zweiten Phase des Regressionstests kann nach zwei Strategien erfolgen: Die erste Strategie sieht einen Test mit allen vorliegenden Testdaten vor (*Retest-All Strategie*), während in der zweiten Strategie der Test mit einer Untermenge der vorliegenden Testdaten durchgeführt wird (*Selective-Retest Strategie*) [RH94a, Rot96]. In [RW97] werden verschiedene Kennzahlen vorgestellt, die bei der Entscheidung zwischen diesen beiden Strategien herangezogen werden können.

In [RH94a, Rot96] findet sich ein Vergleich der verschiedenen Regressionstestverfahren nach der Selective-Retest Strategie. Da die verschiedenen Regressionstestverfahren nach sehr unterschiedlichen Prinzipien funktionieren, sind auch die Ergebnisse der einzelnen Verfahren sehr unterschiedlich. In diesem Papier wird eine Reihe von Kriterien angegeben, die troztdem den Vergleich und die Beurteilung der unterschiedlichen Verfahren ermöglicht:

- Vollständigkeit
  Die Vollständigkeit eines Verfahrens gibt die Rate an, mit der Fehler durch die ausgewählte Testdatenmenge gefunden werden können, d.h. je höher die Vollständigkeit eines Verfahrens, desto mehr Testdaten werden ausgewählt, die Fehler ausführen und sichtbar machen können.

- Präzision
  Die Präzision eines Verfahrens bestimmt die Rate, mit der Testdaten ausgewählt werden, die keine Fehler ausführen und somit auch keine Fehler sichtbar machen können.

- Effizienz
  Die Effizienz eines Verfahrens gibt zum einen dessen Komplexität in Bezug auf Rechenzeit und Platzbedarf, aber auch dessen Automatisierbarkeit an. Die Effizienz entscheidet also über die praktische Anwendbarkeit des Verfahrens.

- Allgemeinheit
  Die Allgemeinheit eines Verfahrens ermöglicht Aussagen über die Fähigkeit des Verfahrens realistische Sprachkonstrukte, beliebig komplexe Programmveränderungen und realistische Testgegenstände verarbeiten zu können.

In [RH94a, Rot96] werden die verschiedenen Verfahren, die bis zu diesem Zeitpunkt vorgeschlagen wurden, in die drei Gruppen Überdeckungsmethoden, Minimierungsmethoden und zuverlässige Methoden eingeordnet (vgl. [RW97, GHK+98]). Für einen empirischen Vergleich verschiedener Verfahren aus diesen Gruppen siehe [GHK+98].

Überdeckungsmethoden

Bei Überdeckungsmethoden werden zum Regressionstest alle Testdaten aus der gegebenen Testdatenmenge ausgewählt, die zur Sicherstellung eines bestimmten Überdeckungskriterium auf den veränderten Abschnitten des Testgegenstandes benötigt werden. Die Generierung von neuen Testdaten fällt dabei nur dann an, wenn die vorhandenen Testdaten zur Erfüllung des Überdeckungskriteriums nicht ausreichen, wie zum Beispiel in den Fällen, in denen das zu testende Programm um neue Abschnitte erweitert wurde. In [RH94a, Rot96] werden Überdeckungsmethoden weiter in Methoden unterteilt, die auf die symbolische Ausführung, die Pfadanalyse, den Datenfluß oder den Program Dependence Graph des Testgegenstandes basieren. Auf diese weitere Unterteilung soll in dieser Diplomarbeit allerdings nicht eingegangen werden. Der interessierte Leser sei auf [RH94a, Rot96] verwiesen.

Im folgenden wird beispielhaft eine Überdeckungsmethode nach [YK87] basierend auf der symbolischen Ausführung des Testgegenstandes beschrieben:

Dieses Verfahren zum Regressionstest sieht in einem ersten Schritt die Partitionierung des Eingaberaums des korrigierten Programms vor. Die Partitionierung wird dabei auf der Grundlage sowohl der Spezifikation als auch der Implementierung durchgeführt, d.h. dieses Regressiontestverfahren berücksichtigt also auch funktionale Veränderungen des Testgegenstandes. Danach werden alle Testdaten aus der gegebenen Testdatenmenge, die gültige Eingaben darstellen, den einzelnen Partitionen zugeordnet. Da ein Überdeckungskriterium unterstellt wird, das genau ein Testdatum pro Partition fordert, werden unter Umständen anschließend neue Testdaten generiert, wenn die Überdeckung aller Partitionen nicht gegeben war. Nach dem Aktualisieren der Testdatenmenge wird das zu testende Programm durch symbolische Ausführung der Testdaten validiert. Da jedoch der Test die tatsächliche Ausführung des Programms erfordert, wird das Programm anschließend mit den Testdaten durchlaufen.

Minimierungsmethoden

Minimierungsmethoden basieren ebenfalls auf Überdeckungskriterien, wobei hier allerdings die Überdeckung der veränderten Programmabschnitte durch minimale Testdatenmengen erreicht wird. Die bisher in der Literatur vorgeschlagenen Minimierungsverfahren lassen im Gegensatz zu Überdeckungsmethoden keine Veränderungen der Kontrollstruktur zu, d.h. es können keine Abschnitte gelöscht oder der Testgegenstand um neue Abschnitte erweitert werden.

Minimierungsmethoden werden in [RH94a, Rot96] weiter in Methoden untergliedert, die auf lineare Gleichungssysteme oder auf die Art der Programmveränderungen basieren. Im folgenden wird nur auf Minimierungsverfahren eingegangen, die auf linearen Gleichungssystemen basierend die Optimierung durchführen. Für Minimierungsverfahren basierend auf Programmveränderungen sei auf [RH94a, Rot96] verwiesen.

Als ein Beispiel für Minimierungsmethoden wird hier die Methode aus [Fis77] beschrieben:

In dieser Minimierungsmethode werden zuerst aus der Implementierung des veränderten Programms Matrizen erzeugt, die die Erreichbarkeit der einzelnen Programmsegmente durch die Testdaten und die der einzelnen Programmsegmente untereinander angeben. Anschließend wird mit Hilfe dieser beiden Matrizen für jedes Programmsegment eine Ungleichung als Nebenbedingung für die spätere Minimierung aufgestellt. Die linke Seite jeder Ungleichung gibt dabei die einzelnen Testdaten an, die das entsprechende Programmsegment überdecken, während die rechte Seite der Ungleichung abgibt, ob das Programmsegment entsprechend dem Überdeckungskriterium durch die entgültige Testdatenmenge überdeckt werden muß oder nicht. Dieses Verfahren unterstellt dabei ein Überdeckungskriterium, das die Ausführung von allen Programmsegmenten erfordert, die von veränderten Programmsegmenten erreichbar sind. Anschließend wird eine Minimierung über die Zahl der Testdaten durchgeführt, wobei die oben aufgestellten Ungleichungen als Nebenbedingungen in die Minimierung eingehen.

Zuverlässige Methoden

Bei zuverlässigen Methoden stehen weniger Überdeckungskriterien im Vordergrund, sondern mehr die Selektion von allen Testdaten aus der Testdatenmenge, die ein Fehlverhalten des veränderten Programms verursachen können. Somit ist also die Retest-All Strategie eine zuverlässige Methode. Da als eine Anforderung an Testverfahren für Software im Investmentbanking auch die Zuverlässigkeit angegeben wurde, kommen zum Test von Prototypen nur Verfahren dieser Kategorie in Frage. Die zuverlässigen Methoden werden in [RH94a, Rot96] ebenfalls weiter untergliedert. Es wird dabei zwischen der Firewall Methode, der Cluster Identifikations Methode, der Slice-basierten Methode und der Program Dependence Graph Methode unterschieden. Im nächsten Unterkapitel wird die Program Dependence Graph Methode nach Rothermel/Harrold ausführlich beschrieben, da dieses Verfahren die Grundlage des in dieser Diplomarbeit entwickelten Verfahrens zum Testen von Prototypen darstellt. Beispiele anderer zuverlässiger Methoden finden sich in [RH94a, Rot96].

4.3.2 Regressionstest nach Rothermel/Harrold

Program Dependence Graph (PDG)

Die Program Dependence Graph Methode [RH94b] basiert auf einer Darstellung des zu testenden Programms, wie schon aus der Bezeichnung ersichtlich, in Form eines *Program Dependence Graphs* (*PDG*) [FOW87]. Aus diesem Grund soll, bevor das eigentliche Verfahren beschrieben wird, das nötige Wissen im Zusammenhang mit PDGs zusammengefaßt werden.

Ein PDG stellt das zu analysierende Programm in Form eines gerichteten Graphen dar. Die einzelnen Anweisungen in dem Programm werden dabei durch Knoten in diesem Graphen repräsentiert, während die Kanten die Abhängigkeiten zwischen den Anweisungen verdeutlichen. In einem Programm können zwei verschiedene Abhängigkeiten zwischen den Anweisungen existieren. Eine *Datenabhängigkeit* kann zwischen zwei Anweisungen genau dann bestehen, wenn in der einen Anweisungen der Wert einer Variablen verändert und in einer anderen Anweisungen auf den Wert der gemeinsamen Variablen zugegriffen wird. Die Reihenfolge dieser beiden Anweisungen darf also nicht verändert werden. Eine *Kontrollabhängigkeit* besteht dagegen genau dann, wenn die Ausführung einer Anweisung von einer anderen Anweisung abhängt. Eine Kontrollabhängigkeit existiert zum Beispiel zwischen einer Fallunterscheidung in Form eines `if-then`-Konstrukts und den Anweisungen, die als Alternativen abhängig von Prädikat der Fallunterscheidung ausgeführt werden können. Bestehen also Kontrollabhängigkeiten zwischen zwei Knoten, dann gehen von einem Knoten (mindestens) zwei Kontrollflußpfade aus, wobei nur einer dieser Pfade den anderen Knoten enthält. Es soll nochmal darauf hingewiesen werden, daß PDGs nicht die Reihenfolge

```
E26 int List::search(int& loc, int item) {
S27   loc = 0;
P28   while (loc < numentries) {
P29     if (list[loc] == item)
S30       return 0;
        else
S31       ++loc;
      }
S32   return -1;
    }
```

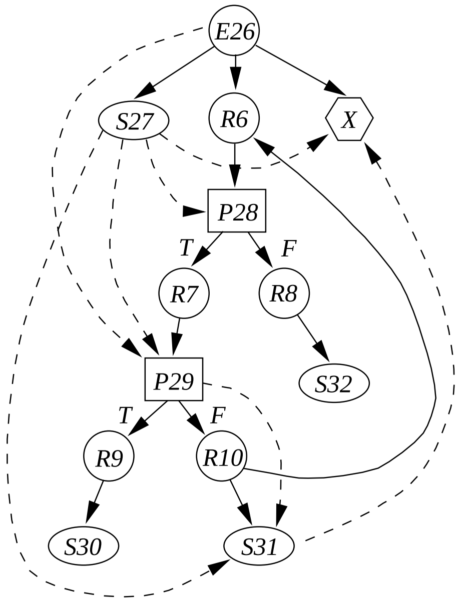


Abbildung 4.1: `search`-Prozedur mit PDG [RH94b]

der Anweisungen, wie das in Kontrollflußgraphen durch Kontrollflußpfade geschieht, wiedergeben, sondern Abhängigkeiten zwischen den Anweisungen.

Eine Anwendung von PDGs besteht in der Bestimmung der Befehle in einem Programm, die parallel ausgeführt werden können. Andere Anwendungsbeispiele für PDGs finden sich in [FOW87].

Abbildung 4.1 zeigt beispielhaft einen PDG, in dem Kontrollabhängigkeiten mit durchgezogenen Linien und Datenabhängigkeiten mit gestrichelten Linien gezeichnet sind. In dieser Abbildung ist auch ersichtlich, daß in einem PDG vier verschiedene Knotentypen unterschieden werden können.

- Statement-Knoten
  Statement-Knoten, die als Ellipsen dargestellt werden, repräsentieren einfache Befehle in dem Sourcecode des Programms, wie zum Beispiel Zuweisungen, Ein- bzw. Ausgabebefehle, Variablendeklarationen oder Funktionsaufrufe.

- Region-Knoten
  Region-Knoten werden durch Kreise abgebildet. Sie fassen Anweisungen, die nur durch bestimmte Bedingungen erreichbar sind, zu einem Block zusammen.

- Predicate-Knoten
  Predicate-Knoten stellen Fallunterscheidungen dar. Sie werden durch Rechtecke symbolisiert, von denen entsprechend den möglichen Fällen mehrere Kontrollpfade ausgehen.

- Exit-Knoten
  Der Exit-Knoten wird im PDG als Sechseck gezeichnet und stellt den Endknoten des PDGs dar.

Algorithmen zur Generierung von PDGs finden sich in [RH93a, CC95].

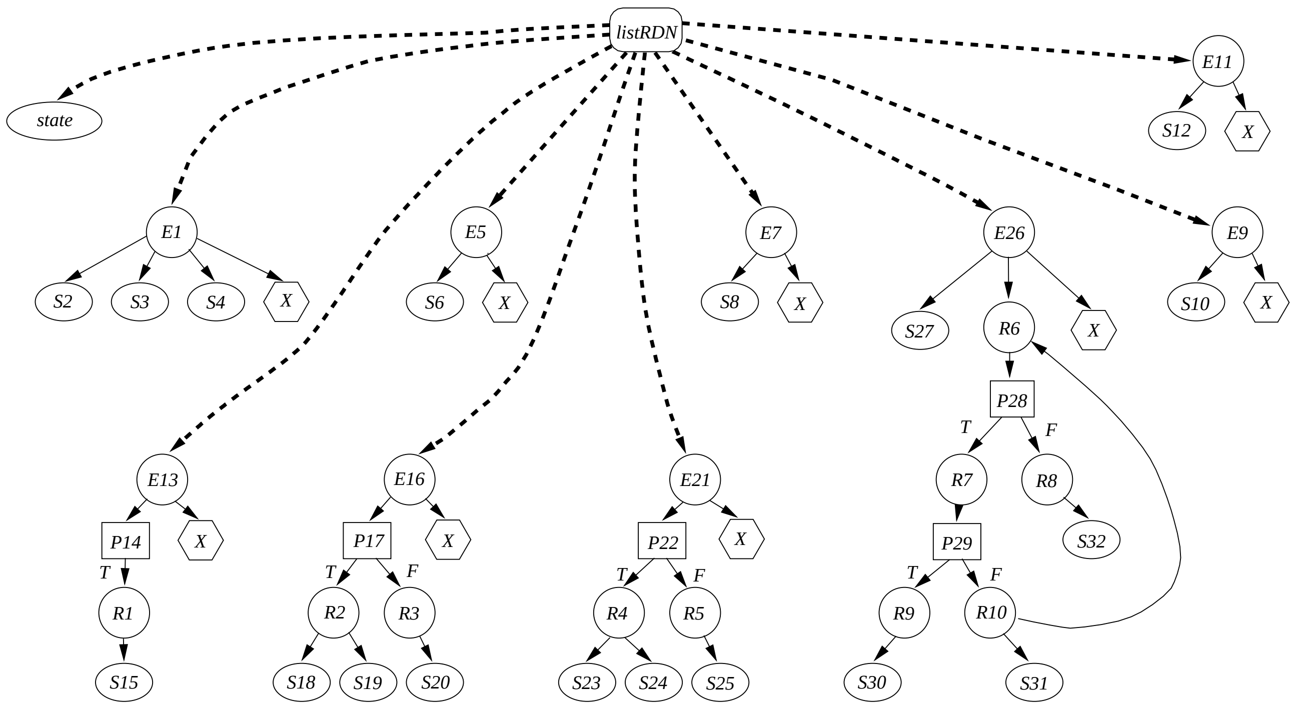
Class Dependence Graph (ClDG)

PDGs stellen Kontroll- und Datenabhängigkeiten in einzelnen Funktionen bzw. in einzelnen Methoden dar. Eine Klasse umfaßt üblicherweise mehrere Methoden, so daß es also naheliegt, die einzelnen PDGs der Methoden einer Klasse zu einem Graphen zusammenzufassen. Dieser Graph wird als *Class Dependence Graph* (*ClDG*) bezeichnet. In Abbildung 4.2 ist ein Beispiel für ein ClDG, der den PDG aus Abbildung 4.1 als einen Untergraphen für die Methode `search` enthält, angegeben. ClDG besitzen jedoch einige Besonderheiten, die noch einer Erklärung bedürfen. Zum Testen einer Klasse sind Treiber notwendig, die eine Instanz der Klasse erzeugen, die Klasse in einen Anfangszustand bringen und die Methoden in einer bestimmten Sequenz aufrufen. In ClDGs werden Treiber, um nicht alle möglichen Treiber einzeln berücksichtigen und die einzelnen PDGs der Methoden mehrfach durchlaufen zu müssen, in einem einzigen Knoten zusammengefaßt. Dieser Knoten wird *Representative Driver Node* genannt. In der Abbildung 4.2 ist dieser Knoten als Wurzelknoten dargestellt und mit der Bezeichnung *listRDN* markiert. In der Abbildung ist auch ersichtlich, daß dieser Knoten zu jeder von außen sichtbaren Methode[3] einen gestrichelten Pfad besitzt, um Kontrollabhängigkeiten zwischen dem Testrahmen und der Klasse anzuzeigen. Die einzelnen Attribute der Klasse, die insgesamt den Zustand eines Objektes der Klasse ausmachen, werden durch einen Knoten mit der Bezeichnung *state* abgebildet. Es soll darauf hingewiesen werden, daß aus Platzgründen in dieser Abbildung Kanten zur Veranschaulichung von Datenabhängigkeiten in den einzelnen Methoden nicht dargestellt sind. Außerdem existieren auch keine Kontrollpfade zwischen den PDGs der einzelnen Methoden, da sich die Methoden in diesem Beispiel nicht gegenseitig aufrufen.

Im objektorientierten Prototyping werden die Klassen des Prototyps nicht durch direktes Verändern den Nutzeranforderungen angepaßt, sondern fehlende oder falsche Funktionalität durch Vererbung hinzugefügt bzw. überschrieben. Vererbung wird in einem ClDG entsprechend der Abbildung 4.3 sichtbar gemacht. In dieser Abbildung ist eine `Stack`-Klasse dargestellt, die Methoden von der obigen `List`-Klasse erbt, wobei wiederum die Kanten zur Darstellung der Datenflußabhängigkeiten fehlen. Die fett gezeichneten Kanten machen in dieser Abbildung die Kontrollabhängigkeiten zwischen den geerbten und den neu definierten Methoden sichtbar, während die gestrichelten Pfeile wie in Abbildung 4.2 Kontrollabhängigkeiten zwischen dem Testrahmen und der Klasse anzeigen.

[3] In C++ sind das genau die `public` deklarierten Methoden.

```
    const int MAXLIST = 10;
    class List {
      int *list;
      int numentries;
      int maxentries;
    public:
E1    List(int n = MAXLIST){
S2      list = new int[n];
S3      maxentries = n;
S4      numentries = 0;
      }
E5    ~List() {
S6      delete list;
      }
E7    int getnumentries()  {
S8      return numentries;
      }
E9    int getmaxentries()  {
S10     return maxentries;
      }
      int search();
      int putitem(int, int);
      int getitem(int&,int);
E11   void setnum(int n) {
S12     numentries = n;
      }
E13   void incnum() {
P14     if(numentries <
         maxentries)
S15       ++numentries;
      }
    }
```




```
E16 int List::putitem(int item,int loc) {
P17    if (0 <= loc && loc < maxentries) {
S18      list[loc] = item;
S19      return 0;
       }
       else
S20      return -1;
    }
```

```
E21 int List::getitem(int& item, int loc) {
P22   if (0 <= loc < maxentries) {
S23     item = list[loc];
S24     return 0;
      }
      else
S25     return -1;
    }
```

```
E26 int List::search(int& loc, int item) {
S27   loc = 0;
P28   while (loc < numentries) {
P29     if (list[loc] == item)
S30       return 0;
        else
S31       ++loc;
      }
S32   return -1;
    }
```

Abbildung 4.2: `List`-Klasse als ClDG [RH94b]

```
    class Stack : public List{
      int top;
    public:
E40   Stack(int n):
S41     List(n);
S42     { top = 0; }
E43   ~Stack():
S44     ~List();
S45     { top = 0; }
      int push(int item);
      int pop(int& item);
      void print();
    };

E46 int Stack::pop(int& item){
P47   if (top > 0) {
S48     getitem(item, --top);
S49     return 0;
      }
      else
S50     return -1;
    }

E51 int Stack::push(int item){
S52   int max=getmaxentries();
P53   if (top < max) {
S54     putitem(item, top++);
S55     return 0;
      }
S56   return -1;
    }
```

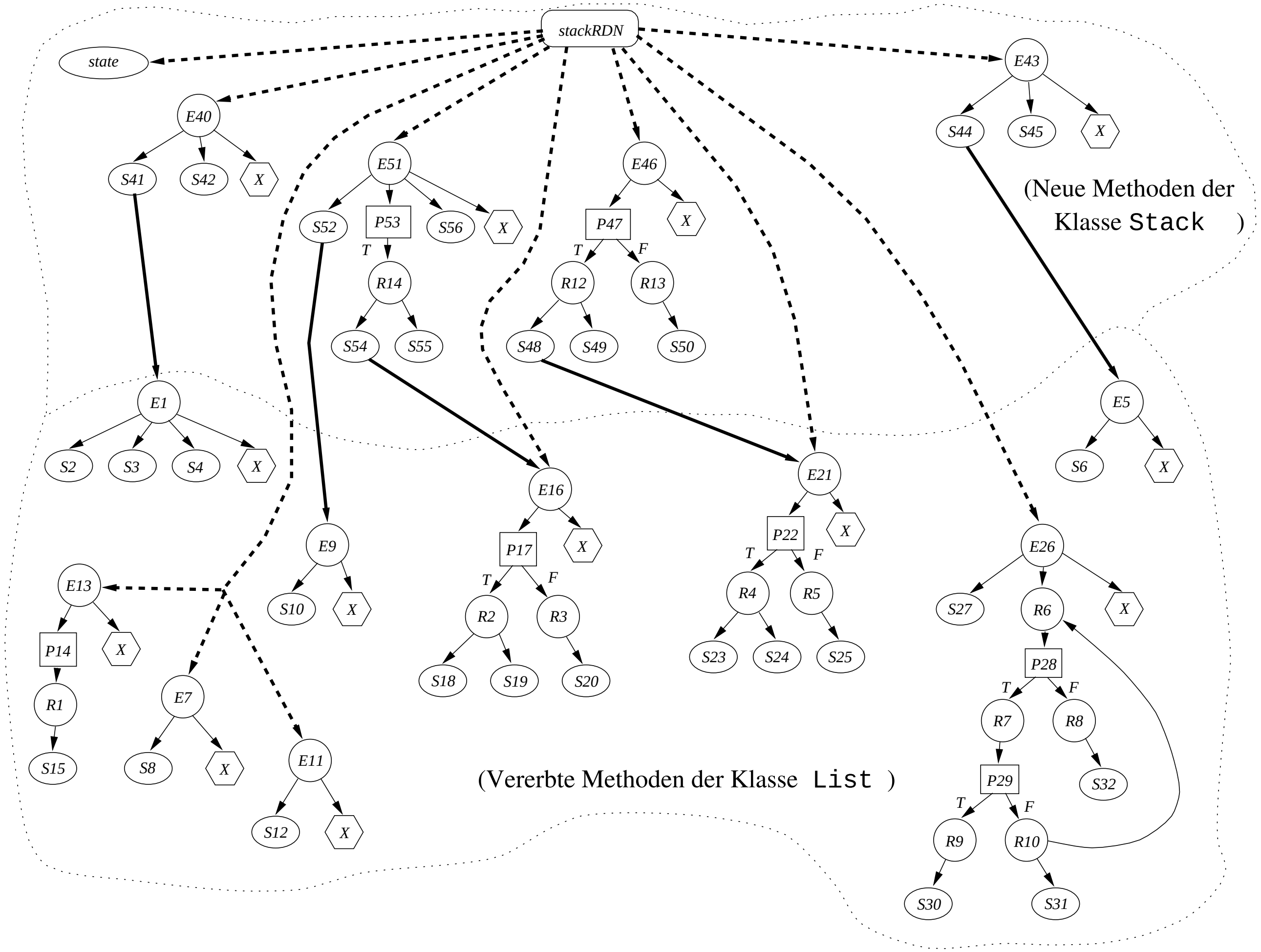


Abbildung 4.3: Vererbung in ClDGs [RH94b]

Algorithmus nach [RH93b]

Fehler können nur dann durch Fehlverhalten sichtbar gemacht werden, wenn die Software mit Eingaben getestet wird, die die fehlerhaften Abschnitte der Software ausführen. Allerdings ist es möglich, daß nicht alle Eingaben, die diese fehlerhaften Abschnitte ausführen, auch den Fehler sichtbar machen, d.h. daß sich ein Fehlverhalten nur bei bestimmten Eingaben einstellt. Da aber schon aus theoretischen Gründen[4] vor dem Test nicht entschieden werden kann, welche Eingaben, die veränderte und unter Umständen fehlerhafte Programmabschnitte erreichen, ein Fehlverhalten verursachen und welche nicht, müssen im Rahmen des zuverlässigen Regressionstests alle Testdaten ausgewählt werden, die veränderte Abschnitte ausführen.

Angenommen die Zeile `S31 ++loc` der Methode `List::search` in Abbildung 4.2 sei zur „Korrektur" durch die Zeile `S31' loc += 2` ersetzt worden. Der ClDG der Klasse `List` sei G und der der veränderten Klasse G', wobei Knoten des veränderten ClDG G' im folgenden durch ein Apostroph kenntlich gemacht werden. Dieser Fehler kann nur mit Testeingaben erkannt werden, die die Zeile $S31'$ erreichen und damit also alle Bedingungen auf dem Kontrollflußpfad zu dieser Zeile erfüllen. Die Eingaben, die alle Pfadbedingungen erfüllen sind genau diejenigen Eingaben, die im veränderten ClDG den Knoten $R10'$ erreichen. Da nur die Zeile $S31$ eine Veränderung erfahren hat, gleichen sich die Knoten $R10$ und $R10'$, so daß diese beiden Knoten genau von denselben Testdaten überdeckt werden. Der Fehler kann also durch den Test mit den Testdaten, die den Knoten $R10$ überdecken, erkannt werden. Der Algorithmus setzt zu diesem Zweck eine *Test-History* voraus, die für jeden Predicate- und Region-Knoten diejenigen Eingaben angibt, die diesen Knoten überdecken. Die Menge dieser Testdaten für einen Predicate- oder Region-Knoten R wird dabei *Region-History* bezeichnet und mit $R.history$ abgekürzt. Der Test ist also in diesem Fall mit den Eingaben in $R10.history$ durchzuführen.

In dem oben erläuterten Beispiel wurde von nur einer Veränderung zur Korrektur ausgegangen. Wird aber in dem Beispiel noch zusätzlich die Zeile `P29 if (list[loc] == item)` in `P29' if (list[loc] <= item)` verändert, so wird die Zeile $S31'$ nicht mehr notwendigerweise von allen Eingaben in $R10.history$ erreicht. Es ist zwar immer noch gültig, daß $S31'$ durch die Testdaten ausgeführt wird, die auch $R10'$ ausführen, aber die Äquivalenz zwischen $R10$ und $R10'$ ist durch Kontrollflußveränderungen nicht mehr gegeben. Die Auswahl der Eingaben in $R10.history$ reicht also für einen zuverlässigen Test von $S31'$ nicht mehr aus. Die Eingaben in $R10.history$, die durch den veränderten Kontrollfluß nicht mehr bei $S31'$ ankommen und somit für einen zuverlässigen Test fehlen, überdecken nun andere Bereiche der Methode, die ebenfalls Veränderungen erfahren haben. Für den zuverlässigen Test von $S31'$ müssen also demnach auch die Testdaten in $R7.history$ ausgewählt werden. In diesem Fall wird also bewußt eine Obermenge der Testdaten ausgewählt, die den Knoten $S31'$ ausführen, um die Zuverlässigkeit des Tests zu garantieren. Offensichtlicherweise müssen für den

[4]Theoretische Überlegungen zum Regressionstest finden sich in [RH94a, Rot96].

Test aller Veränderungen alle Testdaten selektiert werden, die einen Region-Knoten mit veränderten Nachfolgern erreichen. In dem Beispiel wären das also die Testdaten in $R7.history$ und $R10.history$[5]. Die Äquivalenz zwischen den Knoten der beiden ClDGs G und G' wird im Algorithmus durch gleichzeitiges Traversieren der beiden Graphen in Präfixordnung ermittelt. Wird dabei ein Region-Knoten R' in G', der veränderte kontrollabhängige Nachfolger besitzt, gefunden, so werden die Eingaben in der Region-History des äquivalenten Region-Knotens R in G für den Regressionstest ausgewählt. Das weitere Traversieren der Nachfolger von R' ist dann nicht mehr nötig, da schon alle Eingaben ausgewählt worden sind, die die Nachfolger überdecken.

Algorithmus nach [RH94b, RH94c]

Der Algorithmus ist in dieser Form [RH93b] noch relativ unpräzise. Eine neue bzw. veränderte Variablendefinition als erste Anweisung der Methode würde beispielsweise zur Selektion aller Testdaten aus der Testdatenmenge führen, auch wenn die Variable in der restlichen Methode nicht mehr erscheint und somit das Ergebnis der Methode nicht beeinflussen kann. In der vorliegenden Diplomarbeit wird deshalb eine präzisere Version des Algorithmus nach [RH94b, RH94c] benutzt, der auch Datenabhängigkeiten berücksichtigt. Dieser Algorithmus schließt Eingaben, die die Ausführung von veränderten Variablendefinitionen, denen jedoch keine Abfrage des Variablenwerts während der Verarbeitung der Eingabe folgt, von der Selektion aus. Dadurch verringert sich die Zahl der ausgewählten Testeingaben, d.h. die Testmenge wird präziser.

In den Abbildungen 4.4 und 4.5 ist der Algorithmus, der in [RH94b, RH94c] vorgeschlagen wird, in einer etwas veränderter Form[6] angegeben. In diesen Abbildungen werden kontrollabhängige Nachfolger als *cd-Nachfolger* und entsprechend datenabhängige Nachfolger als *dd-Nachfolger* abgekürzt. Der Algorithmus benötigt als Eingabe zur Erzeugung der ClDGs die beiden Versionen C und C' der Klasse mit jeweils einer Liste $PubM$ und $PubM'$ der öffentlichen Methoden. Zusätzlich zu diesen Eingaben wird noch die Menge der Testdaten benötigt, die für den Test der Klasse C eingesetzt wurde und aus der die Testdaten für den Regressionstest ausgewählt werden sollen.

Der Algorithmus sieht als ersten Schritt die Erzeugung[7] der ClDGs G und G' der Klassen C und C' für die darauffolgende Analyse vor (Abb. 4.4, Zeile 7f). Anschließend wird der $visited$-Flag jedes Knotens zurückgesetzt (Zeile 9f), damit jeder Knoten höchstens nur einmal traversiert wird. Unterschiede in den $state$-Knoten der ClDGs entstehen durch Veränderungen an den Attributen der Klasse C. Diese Veränderungen können durch Datenabhängigkeiten zu Fehlern in der Klasse C'

[5] In diesem Beispiel ist allerdings $R10.history$ in $R7.history$ enthalten.

[6] Der Algorithmus ist in beiden Veröffentlichungen leider fehlerhaft.

[7] Die Erzeugung von ClDGs wird in [RH94c] nicht beschrieben. Stattdessen wird auf eine Veröffentlichung verwiesen, die, trotz einer Anfrage bei den Autoren, nicht gefunden werden konnte.

1: **procedure** `SelectClassTests`$(C, C', PubM, PubM', T) : T'$
2: **input** C, C' : Ursprüngliche Klasse C und veränderte Version C'
3: $PubM, PubM'$: Liste öffentlicher Methoden in C und C'
4: T : Vorgegebene Testdatenmenge
5: **output** T' : Für den Regressionstest ausgewählte Menge der Testdaten
6: **begin**
7: $G =$ `ConstructClDG`$(C, PubM)$
8: $G' =$ `ConstructClDG`$(C', PubM')$
9: **for all** Knoten n in G und G' **do**
10: Markiere n *not_visited*
11: **for all** neue/veränderte Variable m in $state'$, gelöschte Variable m in $state$ **do**
12: $m.affected = T$
13: $T' = \emptyset$
14: **for all** Nachfolger N von G_{RDN} **do**
15: **if** $\exists$ zu N äquivalenter Nachfolger N' von G'_{RDN} **then**
16: $T' = T' \cup$ `Compare`(N, N')
17: **else**
18: $T' = T' \cup N.history$
19: **end**

Abbildung 4.4: `SelectClassTests`-Prozedur [RH94b, RH94c]

führen, d.h. die Veränderungen müssen also einem Test unterzogen werden. Für den Test von Datenabhängigkeiten ist jedem Knoten eine Menge mit der Bezeichnung $affected$ zugeordnet, die alle Testdaten enthält, die jeweils diesen und einen Knoten mit einer veränderten Definition ausführen. Ein Knoten wird auch kurz als $affected$-markiert bezeichnet, und zwar genau dann, wenn Testdaten in die $affected$-Menge des entsprechenden Knotens eingefügt wurden. Veränderungen an den Attributen werden dadurch getestet, daß der $affected$-Menge der Knoten, die eine Referenz eines veränderten Attributs enthalten, alle Testdaten in T eingefügt werden (Zeile 11f). Nach einer Initialisierung der ausgewählten Testdatenmenge, wird die `Compare`-Funktion für jeden Nachfolger N des Representative Driver Nodes G_{RDN} von G, der einen äquivalenten Knoten N' in G' besitzt, aufgerufen. Läßt sich allerdings kein äquivalenter Nachfolger des Representative Driver Nodes G'_{RDN} in G' findet, weil beispielsweise dieser Knoten gelöscht wurde, so werden alle Eingaben in $N.history$ für den zuverlässigen Test ausgewählt (Zeilen 14–18).

Aufgabe der `Compare`-Funktion ist die Bestimmung derjenigen Eingaben in $N.history$, die für einen zuverlässigen Test des Knoten N' benötigt werden. Die beiden Knoten N und N' werden zu Beginn der Funktion als *visited* markiert (Abb. 4.5, Zeile 5). Anschließend wird dann `GetCorresp` als Prädikat der darauffolgenden Fallunterscheidung auf diesen beiden Knoten ausgewertet.

Die `GetCorresp`-Funktion vergleicht die Nachfolger zweier Region- bzw. Predicate-Knoten N und N' in den beiden ClDGs und entscheidet darüber, ob für den Test des Knotens N' alle Testdaten in $N.history$ erforderlich sind, oder ob eine Untermenge von $N.history$, die dann im weiteren Ablauf der `Compare`-Funktion bestimmt

1: **procedure** `Compare`$(N, N') : T''$
2: **input** N, N' : Region- oder Predicate-Knoten in G und G'
3: **output** T'' : Für den Test von N' ausgewählte Testdaten
4: **begin**
5: Markiere N und N' *visited*
6: **if** `GetCorresp`(N, N') **then**
7: $T'' = \emptyset$
8: **for all** neuer/veränderter cd-Nachfolger A von N', gelöschter cd-Nachfolger A von N **do**
9: **for all** dd-Nachfolger B von A **do**
10: **if** B ist *visited* **then**
11: $T'' = T'' \cup N.history \cap D.history$, B cd-Nachfolger von D
12: **else**
13: $B.affected = B.affected \cup N.history$
14: **for all** cd-Nachfolger A von N oder N', $A.affected \neq \emptyset$ **do**
15: $T'' = T'' \cup N.history \cap A.affected$
16: **for all** cd-Nachfolger A von N und N', A Endknoten **do**
17: Markiere A *visited*
18: **for all** cd-Nachfolger A von N, der nicht *visited* ist **do**
19: $T'' = T'' \cup$ `Compare`(A, A'), A' zu A äquivalenter Nachfolger von N'
20: **else**
21: $T'' = N.history$
22: **return** T''
23: **end**
24:
25: **procedure** `GetCorresp`(N, N') : **boolean**
26: **input** N, N' : Region- oder Predicate-Knoten in G und G'
27: **output** *true* falls Traversierung unterhalb von N und N' fortgesetzt werden kann, *false* sonst
28: **begin**
29: Kontrollabhängige Nachfolger von N und N' zueinander zuordnen
30: **if** keine Zuordnung gefunden **then**
31: **return** *false*
32: **else if** Knoten, der keine einfache Zuweisung enthält, ist neu/verändert/gelöscht **then**
33: **return** *false*
34: **else**
35: **return** *true*
36: **end**

Abbildung 4.5: `Compare`- und `GetCorresp`-Prozedur [RH94b, RH94c]

wird, ausreicht. Dazu wird als erster Schritt für jeden Nachfolger des Knotens N ein äquivalenter Nachfolger des Knotens N' in dem anderen ClDG gesucht (Abb. 4.5, Zeile 29). Häufig kann jedoch durch weitreichende Korrekturen keine Zuordnung der Knoten in den beiden ClDGs aufgestellt werden, so daß zum Test des Knoten N' alle Eingaben in *N.history* notwendig werden (Zeile 30f + 20f). Aber auch wenn eine Zuordnung aufgestellt werden konnte, ist in einigen Fällen die Selektion aller Testdaten in *N.history* unumgänglich. Bei Veränderungen eines Knotens, der keine einfache Zuweisung repräsentiert, müssen zum zuverlässigen Test alle Testdaten, also alle Testdaten in *N.history*, ausgewählt werden, die diese Veränderungen ausführen (Zeile 32f + 20f). Eine *einfache Zuweisung* kann dabei als Definition einer Variablen ohne Seiteneffekte verstanden werden. Bei Veränderungen hingegen, die eine einfache Zuweisung betreffen, erfolgt der Test nur mit denjenigen Eingaben in $N.history$, die sowohl die veränderte Zuweisung als auch eine Referenz der Variablen ausführen. Diese Eingaben werden in der `Compare`-Funktion ermittelt, d.h. in diesen Fällen, wie auch in den Fällen, wo keine Veränderungen an den Nachfolgern der beiden Knoten N und N' festgestellt werden konnten, liefert die `GetCorresp`-Funktion *true* zurück (Zeile 35). Diese Unterscheidung der Veränderungen wird, wie schon oben angedeutet, zur präziseren Testauswahl durchgeführt.

Die Untermenge T'' von T, die für den Test des Knotens N' genutzt werden soll, wird in einem ersten Schritt auf die leere Menge initialisiert (Zeile 7). Anschließend werden die veränderten kontrollabhängigen Nachfolger A des Knoten N bzw. des Knotens N' betrachtet (Zeile 8–13). Für jeden dieser veränderten Nachfolger werden ihre datenabhängigen Nachfolger B daraufhin untersucht, ob sie schon besucht, also *visited* markiert, worden sind (Zeile 10f). Für *visited* markierte Nachfolger B wird $N.history \cap D.history$, wobei B kontrollabhängig von D ist, für den Test des Knotens B ausgewählt (Zeile 11). Dagegen werden nicht *visited* markierte Knoten als $affected$ markiert, da diese Knoten datenabhängig von anderen veränderten Knoten sind, indem $N.history$ zu $B.affected$ hinzugefügt wird (Zeile 13). In den Zeilen 8 bis 13 des Algorithmus werden also diejenigen Eingaben in $N.history$ ausgewählt, die eine veränderte Variablendefinition und eine Referenz dieser Variablen ausführen. Es soll nochmals darauf hingewiesen werden, daß eine Datenabhängigkeit nur zwischen einem Knoten existiert, der den Wert einer Variablen ändert bzw. setzt, und einem anderen Knoten, der auf den Wert dieser Variablen zugreift. Danach werden diejenigen kontrollabhängigen Nachfolger der beiden Knoten N und N' betrachtet, die vorher bei der Traversierung $affected$ markiert worden sind. Für jeden dieser Nachfolger wird für den Test die Schnittmenge von $N.history$ und der Testdaten, die an den betreffenden Nachfolger angehängt wurden, ausgewählt (Zeile 14f). Das sind also wiederum genau diejenigen Eingaben, die sowohl die veränderte Variablendefinition als auch den Zugriff auf die Variable überdecken. Anschließend werden alle Nachfolger der beiden Knoten N und N', die selber wiederum keine Nachfolger besitzen und somit Blattknoten des entsprechenden Graphen darstellen, *visited* markiert (Zeile 16f). Die `Compare`-Funktion wird danach für jeden kontrollabhängigen Nachfolger A, der in der Traversierung noch nicht besucht worden ist, des Knotens N mit dem äquivalenten Knoten in G' aufgerufen.

Die Rechenzeit dieses Verfahrens ist quadratisch in der Zahl der Knoten der ClDGs beschränkt. Allerdings soll hier auf die Laufzeitanalyse nicht eingegangen werden. Der interessierte Leser sei auf [RH94b] verwiesen.

Dieses Verfahren besitzt einige Eigenschaften, die dessen Einsatz in der Entwicklung von Software mit hohen Qualitätsanforderungen ermöglichen und auch nahelegen. Die wichtigste Eigenschaft ist offensichtlicherweise die Zuverlässigkeit dieses Verfahrens. Wie schon erläutert, wählt dieses Verfahren alle Testdaten aus der gegebenen Testdatemenge aus, die veränderte und unter Umständen fehlerhafte Abschnitte ausführen und damit Fehler sichtbar machen können. Diese Eigenschaft trägt damit zur Fehlerfreiheit der Software bei. Die erhöhten Qualitätsanforderungen müssen allerdings nicht unbedingt mit einem höheren Aufwand erkauft werden. Das Verfahren von Rothermel/Harrold ist, verglichen mit anderen Regressionstestverfahren, im allgemeinen präziser, d.h. die ausgewählte Testdatenmenge ist kleiner als Mengen, die mit Hilfe anderer Verfahren bestimmt werden. Dadurch nimmt die Testausführung weniger Zeit in Anspruch. Ein anderes wichtiges Merkmal dieses Verfahrens ist auch, daß das Verfahren in der Lage ist, Abschnitte in der Implementierung zu erkennen, die durch die vorhandenen Testdaten nicht überdeckt werden und die Erzeugung neuer Testdaten erfordern. Algorithmen dazu finden sich in [RH94b, Rot96]. Außerdem kann dieses Verfahren auch werkzeugunterstützt durchgeführt werden, was wiederum zur Effizienz des Software-Prozesses beiträgt.

Rothermel/Harrold haben in [Rot96, RH97] andere Verfahren vorgeschlagen, die zwar dieselbe Strategie wie das hier vorgestellte Verfahren verfolgen, die aber anstelle auf PDGs bzw. ClDGs auf den Kontrollflußgraphen des Programms basieren. Allerdings haben sie dieses Verfahren nicht auf objektorientierte Programme erweitert[8]. Ball schlägt in [Bal98] aufbauend auf den Veröffentlichungen von Rothermel/Harrold mehrere verbesserte Verfahren vor. Allerdings fehlt auch bei Ball die Anwendung seiner Verfahren auf objektorientierte Programme.

4.4 Funktionaler Test von Klassen

Das oben beschriebene Verfahren für den Regressiontest von Klassen ist rein strukturell und basiert damit nur auf der Implementierung der Klasse. An keiner Stelle des Algorithmus wird die Spezifikation der Klasse benötigt. Gerade aber beim Prototyping, wo die Spezifikation Schritt für Schritt um weitere Funktionalität erweitert wird, ist der funktionale Test besonders wichtig. Es muß nicht nur sichergestellt werden, daß der Prototyp richtig implementiert wird, sondern, daß sich auch die Funktionalität des Prototyps in den Prototyping-Iterationen an die Anforderungen der Nutzer nähert.

Der funktionale Test des Prototyps basiert in dieser Diplomarbeit auf den Verfahren von Tse/Xu und Hong/Kwon/Cha, wobei aus dem ersteren nur die Generierung eines Zustandsautomaten aus einer formalen Spezifikation übernommen wurde, da das zweite Verfahren, das die eigentliche Grundlage des funktionalen Tests darstellt, von einer Spezifikation in Form eines Automaten ausgeht. Der funktionale Test wird also, genau wie der strukturelle Test, nur für einzelne Klassen durchgeführt, d.h. daß im wesentlichen nur die Interaktion der Methoden einer einzelnen Klasse funktional getestet wird.

[8] Auf einer Anfrage per Email teilte Rothermel mit, daß sie zur Zeit daran arbeiten.

4.4.1 Funktionaler Klassentest nach Tse/Xu

Formale Spezifikation in Larch

Zur Generierung des Zustandsautomaten wird in [TX96] eine Spezifikation der Klasse in LARCH [GHG+93, GGH93, Lea98] vorausgesetzt. Eine LARCH-Spezifikation besteht aus zwei Schichten, die in verschiedenen Sprachen formuliert werden. Die obere Schicht der Spezifikation beschreibt die Schnittstelle und das Verhalten der zu implementierenden Prozedur bzw. des abstrakten Datentyps, während die untere Schicht dafür das mathematische Vokabular zur Verfügung stellt. In der oberen Schicht wird eine von der späteren Programmiersprache abhängige *Behavioral Interface Specification Language* (*BISL*)[9] benutzt. Die Sprache in der unteren Schicht ist dagegen unabhängig von der späteren Programmiersprache und wird *Larch Shared Language* (*LSL*) bezeichnet.

Zur Spezifikation der Schnittstelle werden in der Regel Konstrukte der Programmiersprache benutzt, um eine möglichst kurze und zugleich genaue Spezifikation zu erhalten. Dadurch können auch Besonderheiten der Programmiersprache auf eine relativ einfache Art unterstützt und auch ausgenutzt werden. Eine universelle Sprache zur Schnittstellenspezifikation aller möglichen Programmiersprachen kann für eine einzelne Programmiersprache nicht in dem Maße effizient sein, wie eine speziell für diese Programmiersprache entworfene Sprache.

Das Verhalten einer Prozedur oder eines abstrakten Datentyps wird hingegen mit speziellen Konstrukten der entsprechenden BISL spezifiziert. Eine Prozedur wird durch den Zustand, den sie zur Ausführung voraussetzt, durch die Variablen, die sie verändern kann bzw. darf, und durch den Zusammenhang der Zustände vor und nach der Ausführung der Prozedur spezifiziert. Der Zustand vor der Ausführung wird *Pre-Zustand* und entsprechend der Zustand nach der Ausführung *Post-Zustand* bezeichnet. Die Bedingung, die vor der Ausführung, also im Pre-Zustand, erfüllt sein muß, wird *Pre-Zusicherung* genannt. In analoger Weise wird die Bedingung, die nach der Ausführung der Methode, also im Post-Zustand, eintritt, *Post-Zusicherung* bezeichnet. Das Verhalten eines abstrakten Datentyps wird in erster Linie durch eine Menge von abstrakten Werten für die Objekte des abstrakten Datentyps und durch die Spezifikation der Operationen als Prozeduren beschrieben.

Die LSL dient zur mathematischen Modellierung des zu implementierenden Problems. In LSL werden anstelle von Algorithmen mathematische Operationen und entsprechende Theorien angegeben, die im Problembereich gültig sind. Diese mathematische Basis wird in der BISL bei der exakten Formulierung der Pre- und Post-Zusicherungen benötigt. Daneben kann die LSL aber auch Redundanzen, die beispielsweise bei Korrektheitsbeweisen benutzt werden können, beinhalten. Redundanzen können in Form von Implikationen oder Beispielen in die Spezifikation eingefügt werden.

[9] Häufig werden diese Sprachen auch *Larch Interface Languages* genannt.

```
StackBasics (E, C): trait
  introduces
    empty: → C
    push: E, C → C
    top: C → E
    pop: C → C
  asserts
    C generated by empty, push
    ∀ e: E, stk: C
      top(push(e, stk)) = e;
      pop(push(e, stk)) = stk;
  implies converts top, pop
    exempting top(empty), pop(empty)
```

Abbildung 4.6: Grundlegende Operationen auf Stacks [GHG+93]

Für eine vollständige Einführung in LARCH sei auf die Literatur verwiesen. In den folgenden Ausführungen wird nur das für das Verständnis notwendige Wissen erläutert.

Beispiel einer LSL-Spezifikation: `StackBasics`

In [GHG+93] befindet sich als Anhang eine Bibliothek aus verschiedenen Spezifikationen in LSL, die auch eine Spezifikation der grundlegenden Stack-Operationen beinhaltet. Diese Spezifikation ist mit geringen Veränderungen in Abbildung 4.6 angegeben. Die Grundlagen von LSL werden am Beispiel dieser Abbildung im folgenden kurz zusammengefaßt.

Eine LSL-Spezifikation [GHG+93] definiert zwei verschiedene Arten von Symbolen: Operationen und Sorten. Eine *Operation* bildet einen Tupel aus verschiedenen Werten auf einen zugeordneten Wert ab und entspricht somit einer Prozedur bzw. einer Funktion. Eine *Sorte* stellt ähnlich wie ein Typ in einer Programmiersprache eine Untermenge aller möglichen Werte dar. Obwohl die Begriffe der Operation und der Sorte sehr verwandt mit den Begriffen der Prozedur und des Typs sind, dürfen sie nicht verwechselt werden, da sie durch unterschiedliche Konzepte geprägt sind. Operationen werden in LSL, wie in dem Beispiel in Abbildung 4.6, zu einem *Trait* zusammengefaßt, in dem auch spezielle Eigenschaften dieser Operationen aufgelistet werden. Die Deklaration der Operationen eines Traits folgt nach dem `introduces`-Konstrukt. Das Trait `StackBasics` in Abbildung 4.6 definiert die Operationen `empty`, `push`, `top` und `pop`. Zu jeder dieser Operationen, die häufig auch *Trait-Funktionen* genannt werden, wird die Signatur angegeben. Die *Signatur* einer Operation ist ein Tupel aus den Sorten der Argumente und des Ergebnisses. Die Operation `push` besitzt beispielsweise die Signatur $(E, C) \rightarrow C$, d.h. sie bildet ein Paar bestehend aus der Sorte E und der Sorte C auf einen Wert der Sorte C ab. Die Operation `empty` besitzt hingegen keine Argumente und stellt damit einen abstrakten Wert des abstrakten Datentyps

StackBasics dar. Die Definition der Operationen befindet sich nach dem asserts-Konstrukt, wo Eigenschaften der Sorte und Beziehungen zwischen Operationen und Variablen in Form von Gleichungen und Implikationen aufgelistet werden. Die achte Zeile in der Abbildung drückt mit Hilfe des generated-Konstrukts die Eigenschaft der Sorte StackBasics aus, daß jeder Wert dieser Sorte durch eine endliche Kette von empty und push Operationen erzeugt werden kann. In den Zeilen 9 bis 11 werden dann die Beziehungen zwischen den Grundoperationen eines Stacks definiert. Abschließend werden in den Zeilen 12 und 13 Aussagen über die Vollständig des Traits gemacht. Der converts-Konstrukt gibt die Operationen an, die durch die Axiome des Traits vollständig spezifiziert sind, wobei der exempting-Konstrukt die Fälle angibt, für die diese Operationen nicht definiert sind. In dem obigen Beispiel sind die Operationen top und pop vollständig durch die Axiome spezifiziert, allerdings muß beachtet werden, daß beide Operationen für den Wert *empty* nicht definiert sind, da auf leeren Stacks keine top- und pop-Befehle ausgeführt werden können[10].

Es soll darauf hingewiesen werden, daß an keiner Stelle des Traits StackBasics Aussagen über die Implementierung der Operationen gemacht werden, d.h. es werden keine Datenstrukturen oder Algorithmen beschrieben. Es werden aber auch keine Aussagen über das Verhalten der Operationen für Fälle gemacht, die nicht definiert sind.

Beispiel einer Larch/C++-Spezifikation: Stack

In der vorliegenden Arbeit wird die BISL Larch/C++ [Lea96, Lea97a, Lea97b] der Programmiersprache C++ benutzt. Die Schnittstellenspezifikation der Klasse Stack ist in Abbildung 4.7 angegeben. Eine Schnittstellenspezifikation muß diejenige Information beinhalten, die notwendig für die Kommunikation über die Schnittstelle und aber auch für die Implementierung des der Schnittstelle zugrundeliegenden Programms ist.

Die Verbindung der beiden Stufen bei der Spezifikation in Larch wird durch eine Abbildung der Typen in der jeweiligen BISL auf Sorten in LSL hergestellt. Die Typen in einer Schnittstellenspezifikation basieren jeweils auf Sorten, die die Eigenschaften der Werte dieser Typen durch geeignete Operationen beschreiben. Beispielsweise ist dem Typ int in der Zeile 7 der Abbildung in Larch/C++ ein gleichnamiger Trait zugeordnet, der genau die Eigenschaften der ganzen Zahlen und der Operationen auf ganzen Zahlen spezifiziert. Jeder Schnittstellenspezifikation ist automatisch eine Spezifikation in LSL zugeordnet, die die Basistypen der jeweiligen Programmiersprache beschreibt. Bei komplizierten Typen kann die LSL Spezifikation allerdings nicht automatisch erstellt werden, so daß diese Stufe der Spezifikation noch zusätzlich vom Anwender entwickelt und mit dem uses-Konstrukt in die Larch/C++ Spezifikation eingebunden werden muß.

[10] Die Stack-Operationen, die in dem Trait StackBasics definiert werden, haben eine andere Funktionalität als die gleichnamigen Methoden der Klasse Stack aus dem letzten Unterkapitel.

```
//@ uses StackBasics(int for E, IntStack for C);
//@ spec class IntStack;

class BoundedIntStack {
public:
  //@ spec IntStack stk;
  //@ spec int top_, maxentries;

  //@ invariant maxentries\any >= 0 /\ top_\any <= maxentries\any;

  BoundedIntStack(int limit) throw();
  //@ behavior {
  //@   requires limit >= 0;
  //@   modifies top_, maxentries, stk;
  //@   ensures top_ = 0 /\ maxentries' = limit /\ stk' = empty;
  //@ }

  ~BoundedIntStack() throw();
  //@ behavior {
  //@   ensures true;
  //@ }

  int push(int item) throw();
  //@ behavior {
  //@   requires top_^ < maxentries^;
  //@   modifies top_, stk;
  //@   ensures top_' = top_^ + 1 /\ stk' = push(item, stk^) /\ result = 1;
  //@ also
  //@   requires top_^ = maxentries^;
  //@   ensures result = -1;
  //@ }

  int pop(int& item) throw();
  //@ behavior {
  //@   requires top_^ > 0;
  //@   modifies top_, stk;
  //@   ensures top_' = top_^ - 1 /\ item = top(stk^) /\ stk' = pop(stk^)
  //@           /\ result = 1;
  //@ also
  //@   requires top_^ = 0;
  //@   ensures result = -1;
  //@ }
}
```

Abbildung 4.7: LARCH/C++-Spezifikation der Klasse `Stack`

LARCH/C++ genügt, wie es auch in der Abbildung 4.7 deutlich wird, der C++-Syntax, so daß die LARCH/C++ Spezifikation als Header-Datei verwendet werden kann (und sollte). Der C++-Kompiler interpretiert die LARCH/C++ eigenen Konstrukte als Kommentare, da sie sich nach der Zeichenfolge `//@` befinden. In der ersten Zeile der Abbildung wird das Trait `StackBasics` mit Hilfe des `uses`-Konstrukt als Basis der Schnittstellenspezifikation definiert, wobei in dem Trait E durch `int` und C durch `IntStack` ersetzt wird. Anschließend wird in der zweiten Zeile die Klasse `IntStack` deklariert, wobei der `spec`-Konstrukt deutlich macht, daß diese Klassendeklaration nur für die Spezifikation benötigt wird und nicht implementiert werden muß. Entsprechend wird in der Zeile 6 der Spezifikation ein Objekt dieser Klasse ebenfalls mit dem `spec`-Konstrukt deklariert. In der siebten Zeile werden zwei Variablen vom Typ `int` deklariert, die zum einen die Zahl der in dem Stack enthaltenen Elemente und die Größe des Stacks angeben. Auch diese Variablen werden nur zur Spezifikation benötigt. In jedem Zustand muß die Größe des Stacks größer null und die Anzahl der enthaltenen Elemente kleiner oder gleich der Größe des Stacks sein, d.h. diese Bedingungen stellen die Invariante dar, die immer erfüllt sein muß (Zeile 9). Der Wert einer Variablen hängt jeweils von dem Zustand bzw. dem Zeitpunkt ab, in dem sich das Programm befindet. Deshalb muß zu jeder Variablen der Zustand angegeben werden, in dem der Wert abgefragt werden soll. Während `\any` einen beliebigen Zustand angibt, bezeichnen `^` den Zustand vor (Pre-Zustand) und `'` den Zustand nach (Post-Zustand) der Ausführung einer Prozedur.

Im folgenden wird nur die Spezifikation der Methode `push` erläutert, da alle anderen Methoden analog spezifiziert werden. Entsprechend der C++-Syntax besitzt die `push`-Methode die Signatur `int` $\rightarrow$ `int`. Außerdem erzeugt diese Methode keine Exceptions, d.h. der Argument des `throw`-Konstrukts ist leer (Zeile 23). Nach dem Schlüsselwort `behavior` folgt die Beschreibung des Methodenverhaltens, wobei beim Aufruf dieser Methode entweder die Pre-Zusicherung in der Zeile 25 oder die Pre-Zusicherung in der Zeile 29 erfüllt sein muß. Entweder muß die Zahl der in dem Stack enthaltenen Elemente kleiner als die Größe des Stacks sein oder gleich der Größe. Im ersten Fall gibt der `modifies`-Konstrukt in der Zeile 26 an, daß die Variablen *top_* und *stk* verändert werden können. Da in dem zweiten Fall kein `modifies`-Konstrukt erscheint, kann im zweiten Fall auch keine Variable verändert werden. Durch die `ensures`-Konstrukte in den Zeilen 27 und 30 werden die Werte der veränderten Variablen im Post-Zustand angegeben, wobei *result* das Ergebnis der Methode darstellt. Im ersten Fall wird der Wert der Variablen *top_* um eins erhöht, d.h. der Wert der Variablen *top_* ist im Post-Zustand um eins höher als im Pre-Zustand. Anschließend wird dann die Operation `push` der Sorte `StackBasics` aufgerufen und eins als Ergebnis der Methode zurückgegeben. Im zweiten Fall wird minus eins als Ergebnis der Methode festgelegt, um anzuzeigen, daß die `push`-Operation der Sorte `StackBasics` nicht ausgeführt werden konnte.

Generierung eines Zustandsautomaten aus der formalen Spezifikation

Das Verhalten der Methoden einer Klasse ist im allgemeinen abhängig von den Werten der Attribute. Da außerdem die Attribute einer Klasse durch die Kapselung von außen nicht sichtbar sind, werden häufig die Attributwerte in einem Zeitpunkt auch als *Zustand* [RBP+93] der Klasse bezeichnet. Nach einem Methodenaufruf wechselt häufig dieser Zustand, da die Methoden üblicherweise Operationen auf den Attributen ausführen und diese verändern. Der Wechsel des Objektzustandes wird als *Transition* [RBP+93] bezeichnet. Eine Transition besteht neben dem aktuellen und dem Folgezustand aus Ereignis, Wächter und aus Aktion. Ein *Ereignis* ist die Ursache für den Zustandswechsel und entspricht üblicherweise einem Methodenaufruf. *Wächter* sind hingegen bestimmte Bedingungen, die die Attribute erfüllen müssen, damit eine Transition erfolgen kann. Eine *Aktion* ist in diesem Zusammenhang dann die Operation, die bei der Transition ausgeführt werden soll. Diese Operation umfaßt häufig die Veränderung der Attribute und damit auch indirekt des Zustands. An dieser Stelle soll keine formale Definition von Zustandsautomaten gegeben werden, dazu sei auf den Abschnitt 4.4.2 verwiesen.

In [DF93, Hie97, MCM+98] werden Verfahren zur Generierung von Zustandsautomaten aus formalen Spezifikationen beschrieben. Die folgenden Ausführungen basieren auf [TX96, TX95].

Zur Generierung eines Zustandsautomaten muß in einem ersten Schritt der Zustandsraum der Klasse in abstrakte Zustände partitioniert werden. Der *Zustandsraum* besteht aus allen möglichen Zuständen, die ein Objekt der entsprechenden Klasse annehmen kann. Häufig verhält sich ein Objekt für bestimmte Zustände jedoch sehr ähnlich, so daß diese Zustände zu einem *abstrakten Zustand* zusammengefaßt werden. Der *konkrete Zustand* eines Objektes ist in diesem Zusammenhang dann der tatsächliche Zustand, in dem sich das Objekt befindet.

Zur Partitionierung des Zustandsraums auf der Basis der Spezifikation der Klasse wird folgende Heuristik verwendet:

- Bei der Partitionierung des Zustandsraums sollte nur eine Berücksichtigung der wichtigen Funktionen der Klasse ausreichend sein.
- Der Wertebereich jedes Attributs sollte anhand von speziellen Werten partitioniert werden. Spezielle Werte können sein:
  - Werte, die typisch für das Attribut sind.
  - Werte, die in Fallunterscheidungen bzw. in Bedingungen vorkommen.
  - Werte, die an den Grenzen des Wertebereichs liegen.

In dem einfachen Beispiel der `Stack`-Klasse ist die Partitionierung des Zustandsraums offensichtlich. Die abstrakten Zustände eines `Stack`-Objektes ergeben sich durch den minimalen und den maximalen Wert, den das Attribut $top\_$ annehmen kann. Aus der LARCH/C++-Spezifikation in Abbildung 4.7 der `Stack`-Klasse ist ersichtlich, daß $0 \leq top\_ \leq maxentries$ gilt. Daraus folgt, daß ein `Stack`-Objekt die Zustände *leer*, *belegt* und *voll* annehmen kann. Der Zustand *leer* tritt ein, wenn gilt $top\_ = 0$. Der

Zustand *belegt* tritt dagegen ein, wenn der Stack Elemente enthält, jedoch nicht *voll* ist, d.h. wenn gilt $0 < top\_ < maxentries$. Der Zustand *voll* tritt offensichtlicherweise genau dann ein, wenn $top\_ = maxentries$ gilt.

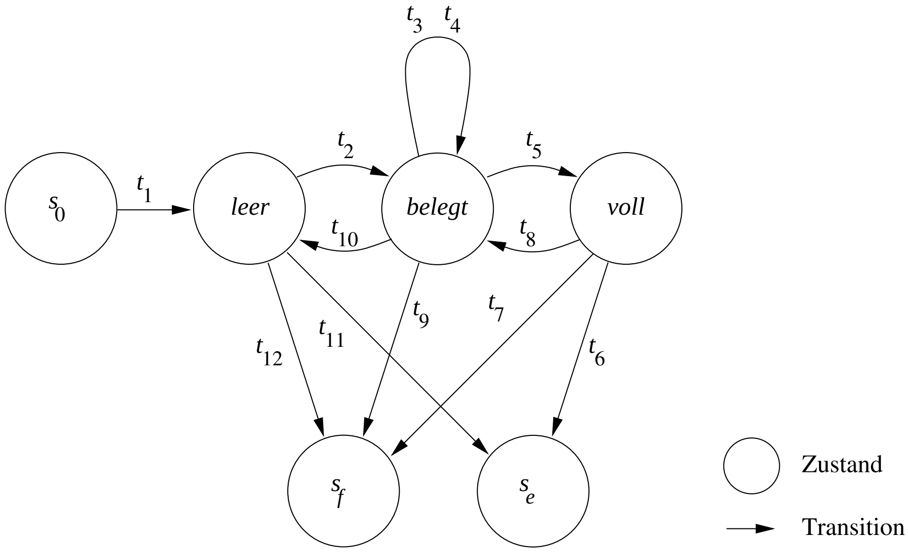
Jeder dieser abstrakten Zustände des `Stack`-Klasse entspricht einem Zustand des Automaten der Klasse. Die Transitionen zwischen den Zuständen des Automaten werden aus der LARCH/C++-Spezifikation der Klasse gewonnen. Das Ereignis jeder Transition entspricht dem Aufruf einer Methode, also einer entsprechenden Nachricht. Der Wächter jeder Transition folgt aus der Pre-Zusicherung, wobei die Aktion der Transition sich aus der Post-Zusicherung der Methode ergibt. Eine Methode kann jedoch mehrere Pre- und Post-Zusicherungen besitzen, so daß in dem Zustandsautomaten der Klasse mehrere Transition zur Darstellung der Methode notwendig werden können. Diese Zusammenhänge sollen am Beispiel der `push`-Methode erläutert werden. Die `push`-Methode der `Stack`-Klasse enthält, wie es aus der Abbildung 4.7 ersichtlich ist, zwei `requires`-Konstrukte, die zwei verschiedene Pre-Zusicherungen angeben, von denen einer im Zeitpunkt des Aufrufs der Methode erfüllt sein muß. Entweder muß $top\_ < maxentries$ oder $top\_ = maxentries$ gelten. Diese beiden Bedingungen entsprechen also nach den obigen Ausführungen den Wächtern zweier Transitionen. Die Aktionen der Transitionen ergeben sich aus den Post-Zusicherungen, also den Termen nach den `ensures`-Konstrukten. Im ersten Fall wird in der Aktion der Wert des Attributs *top_* um eins erhöht, das übergebene Element abgespeichert und eins zurückgegeben. Die Aktion im zweiten Fall besteht nur aus der Rückgabe des Wertes minus eins als Ergebnis der Methode.

In analoger Weise können alle Zustände und Transitionen des Automaten bestimmt werden. Die vollständige Generierung des Automaten soll an dieser Stelle jedoch nicht vorgeführt werden, da die oben erläuterten Ideen ausreichen sollten, um die Vorgehensweise zu verdeutlichen. In der Abbildung 4.8 ist der vollständige Automat der `Stack`-Klasse angegeben, der sich durch diese Vorgehensweise ergibt.

4.4.2 Funktionaler Klassentest nach Hong/Kwon/Cha

Das obige Verfahren nach [TX96, TX95] ging von einer formalen Spezifikation der Klasse aus, von der dann ein endlicher Automat konstruiert wurde. Das Verfahren nach [HKC95] geht dagegen direkt von einer Spezifikation der Klasse als endlicher Automat aus. Aus diesem Automaten, dem *Class State Machine* (*CSM*), wird mit Hilfe eines Algorithmus, der im folgenden beschrieben wird, ein spezieller Datenflußgraph generiert. Dieser Datenflußgraph, der als *Class Flow Graph* (*CFG*) bezeichnet wird, dient zur Generierung der Testdaten, wobei diese Testdaten im Gegensatz zu dem Regressionstestverfahren aus dem letzten Unterkapitel nicht aus einzelnen Eingaben[11], sondern aus Methodensequenzen bestehen. Der Grund hierfür ist, daß das Regressionstestverfahren nach ROTHERMEL/HARROLD die Korrektheit der einzelnen

[11] Abgesehen von Eingaben für den Konstruktor der Klasse.



$V = \{\textbf{int}\ \mathit{top}\}$
$F = \{\texttt{Stack(}\textbf{int}\texttt{)},\ \texttt{\~{}Stack()},\ \texttt{push(}\textbf{int}\texttt{)},\ \texttt{pop(}\textbf{int\&}\texttt{)}\}$
$S = \{s_0,\ \mathit{leer},\ \mathit{belegt},\ \mathit{voll},\ s_f,\ s_e\}$
$T = \{t_1 = (s_0,\ \mathit{leer}, \texttt{Stack(}n\texttt{)},\ \mathit{wahr},\ \{\texttt{List(}n\texttt{)},\ \mathit{top} = 0\}),$
$t_2 = (\mathit{leer},\ \mathit{belegt}, \texttt{push(}\mathit{item}\texttt{)},\ \mathit{wahr},\ \{\texttt{putitem(}\mathit{item}\texttt{,}\mathit{top}\texttt{++)}\}),$
$t_3 = (\mathit{belegt},\ \mathit{belegt}, \texttt{push(}\mathit{item}\texttt{)},\ \mathit{top} < \mathit{maxentries},\ \{\texttt{putitem(}\mathit{item}\texttt{,}\mathit{top}\texttt{++)}\}),$
$t_4 = (\mathit{belegt},\ \mathit{belegt}, \texttt{pop(}\mathit{item}\texttt{)},\ \mathit{top} > 0,\ \{\texttt{getitem(}\mathit{item}\texttt{,--}\mathit{top}\texttt{)}\}),$
$t_5 = (\mathit{belegt},\ \mathit{voll}, \texttt{push(}\mathit{item}\texttt{)},\ \mathit{top} = \mathit{maxentries} - 1,\ \{\texttt{putitem(}\mathit{item}\texttt{,}\mathit{top}\texttt{++)}\}),$
$t_6 = (\mathit{voll},\ s_e, \texttt{push(}\mathit{item}\texttt{)},\ \mathit{wahr},\ \emptyset),$
$t_7 = (\mathit{voll},\ s_f,\ \texttt{\~{}Stack()},\ \mathit{wahr},\ \emptyset),$
$t_8 = (\mathit{voll},\ \mathit{belegt}, \texttt{pop(}\mathit{item}\texttt{)},\ \mathit{wahr},\ \{\texttt{getitem(}\mathit{item}\texttt{,--}\mathit{top}\texttt{)}\}),$
$t_9 = (\mathit{belegt},\ s_f,\ \texttt{\~{}Stack()},\ \mathit{wahr},\ \emptyset),$
$t_{10} = (\mathit{belegt},\ \mathit{leer}, \texttt{pop(}\mathit{item}\texttt{)},\ \mathit{top} = 1,\ \{\texttt{getitem(}\mathit{item}\texttt{,--}\mathit{top}\texttt{)}\}),$
$t_{11} = (\mathit{leer},\ s_e, \texttt{pop(}\mathit{item}\texttt{)},\ \mathit{wahr},\ \emptyset),$
$t_{12} = (\mathit{leer},\ s_f,\ \texttt{\~{}Stack()},\ \mathit{wahr},\ \emptyset)\}$

Abbildung 4.8: CSM der `Stack`-Klasse

Methoden testet, während bei dem Verfahren nach Hong/Kwon/Cha die korrekte Interaktion der einzelnen Methoden über die Attribute im Mittelpunkt steht. Aus dem CFG werden dann Datenflüsse zwischen den Methoden identifiziert und auf der Basis von Datenflußkriterien Testdaten erzeugt[12].

Class State Machine (CSM)

Eine *Class State Machine* (*CSM*) besteht nach [HKC95] neben endlichen Mengen V und F für jeweils die Attribute und die Methoden der Klasse C noch aus endlichen Mengen S und T für die Zustände und die Transitionen. Zum Verständnis der CSM sei auf die Abbildung 4.8 verwiesen.

[12] Ein anderes Verfahren für den datenflußbezogenen Klassentest findet sich in [HR94].

Neben den Zuständen, die durch bestimmte Attributwerte zustande kommen, besitzt eine CSM noch zusätzlich die Zustände s_0, s_f und s_e. Während s_0 den Zustand vor der Objekterzeugung, also vor dem Aufruf des Konstruktors darstellt, gibt s_f den Zustand nach dem Löschen des Objektes, also nach dem Destruktoraufruf, wieder. Der Zustand s_e repräsentiert den Fehlerzustand, d.h. denjenigen Zustand, der nach einem ungültigem Ereignis erreicht wird.

Eine Transition t ist formal als ein Tupel $(source, target, fn, guard, action)$ definiert. Hierbei stellen *source* und *target* den Ausgangs- und den Folgezustand dar, während fn die Methode angibt, die die Transition auslöst. *Guard* ist diesem Zusammenhang die Bedingung, welche die Attribute erfüllen müssen, damit die Transition stattfinden kann. *Action* repräsentiert schließlich die Operation, die während der Transition auf den Attributen ausgeführt werden soll.

Class Flow Graph (CFG)

Der CFG einer Klasse C ist ein gerichteter Graph bestehend aus einer Knotenmenge N und einer Kantenmenge E. Zu jedem Zustand s, Wächter g und jeder Transition t existiert ein entsprechender Knoten in dem Graphen, d.h. die Knotenmenge N ist eine Vereinigung der Mengen N_s, N_g und N_t. Diese Knoten werden durch Kanten aus der Kantenmenge E untereinander verbunden. Die Kanten werden nach dem Typ der Knoten, die sie verbinden, in die Mengen E_{st}, E_{sg}, E_{gt} und E_{ts} eingeordnet, wobei der Index einer Menge die Knotentypen angibt, die die Kanten aus dieser Menge verbinden.

Abbildung 4.9 stellt den CFG der Klasse `Stack` dar. In dieser Abbildung sind Knoten, die die Zustände des Objektes bzw. des Automaten repräsentieren, als Kreise gezeichnet. Die Rechtecke in dieser Abbildung stehen hingegen für die Transitionen des Automaten, wobei die Wächter der jeweiligen Transitionen als Rhomben, die direkt den entsprechenden Rechtecken vorgeschaltet sind, abgebildet werden.

Durch den Vergleich der Abbildungen 4.3 und 4.9 werden die Unterschiede zwischen ClDGs und CFGs deutlich. Die Knoten in einem ClDG entsprechen jeweils genau einer Anweisung in einer Methode der Klasse, während die Knoten in einem CFG dagegen in der Regel mehrere Anweisungen bzw. eine vollständige Methode beinhalten. Auch die Bedeutung der Kanten ist unterschiedlich. Die Kanten in einem ClDG stellen jeweils Abhängigkeiten bezüglich dem Kontroll- und dem Datenfluß dar. Kanten in einem CFG repräsentieren den Kontrollfluß in einer Klasse, d.h. CFGs beinhalten in Bezug auf den Kontrollfluß mehr Informationen als ClDGs. ClDGs beinhalten keinerlei Informationen über die Reihenfolge von Anweisungen, solange die Anweisungen weder Kontroll- noch Datenabhängig sind, wogegen CFGs die Reihenfolge der Anweisungen bzw. der Anweisungsblöcken explizit als Kanten darstellen.

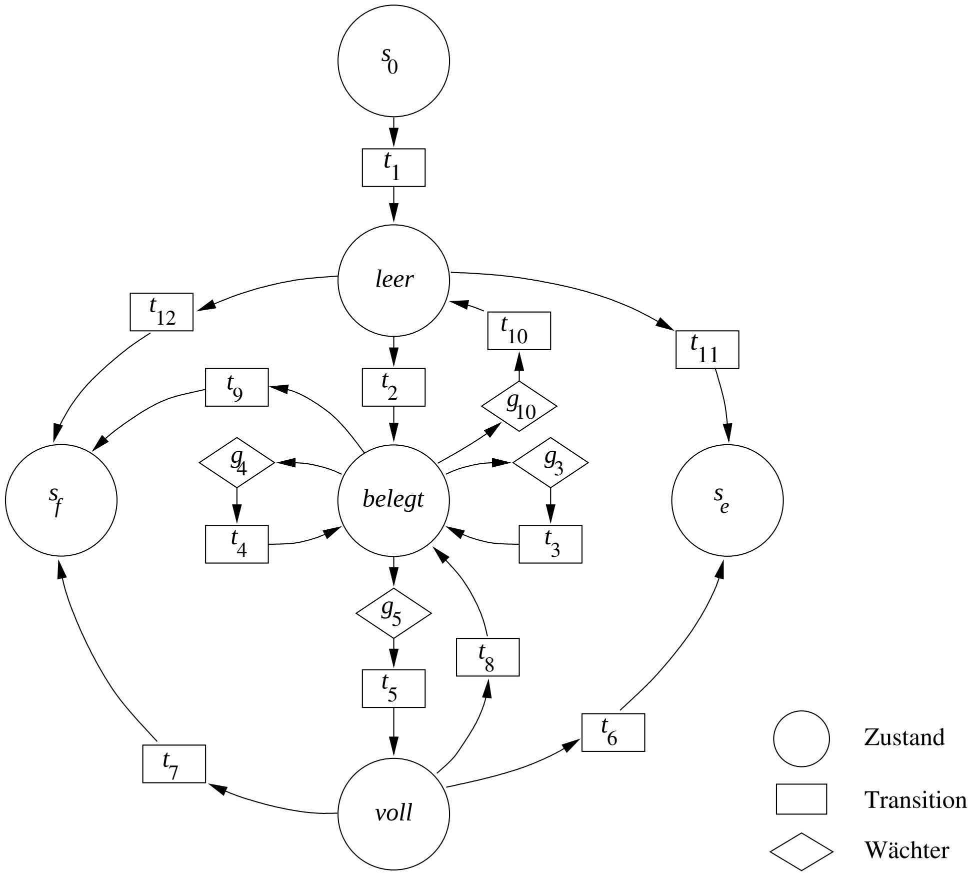


Abbildung 4.9: CFG der `Stack`-Klasse

Transformation von CSMs in CFGs

Die Transformation eines CSMs zu einem CFG wird in [HKC95] mit Hilfe des Algorithmus in Abbildung 4.10 durchgeführt. Im ersten Schritt des Algorithmus werden die einzelnen Untermengen N_s, N_g und N_t der Knotenmenge zu der leeren Menge initialisiert (Zeile 5). Anschließend wird für jeden Zustand s des Automaten die Menge N_s um einen entsprechenden Knoten erweitert, so daß jeder Zustand durch einen Knoten abgebildet wird. Danach wird für jede Transition ein Knoten in N_t eingefügt, wobei allerdings nur dann ein Knoten für den Wächter der Transition in N_g erscheint, wenn der Wächter nicht den konstanten Wert $wahr$ besitzt (Zeile 8–11). An dieser Stelle des Algorithmus ist schon die Erzeugung aller notwendigen Knoten des Graphen abgeschlossen. Im weiteren Verlauf werden dann die entsprechenden Kanten generiert. Dazu werden wiederum die einzelnen Kantenmengen E_{sg}, E_{st}, E_{gt} und E_{ts} auf die leere Menge initialisiert (Zeile 12). Abhängig vom Wächter jeder Transition t wird entweder eine Kante, die den Ausgangszustand $t.source$ und die Transition t repräsentierenden Knoten verbindet, in die Menge E_{st} eingefügt oder zwei Kanten, die zum einen den Anfangszustand $t.source$ und den Wächter $t.guard$ und zum an-

1: **procedure** `CSM2CFG`$(M) : G$
2: **input** M: CSM der Klasse C mit $M = (V, F, S, T)$
3: **output** G: CFG der Klasse C mit $G = (N, E)$
4: **begin**
5: $\quad N_s = N_g = N_t = \emptyset$
6: $\quad$ **for all** $s \in S$ **do**
7: $\qquad N_s = N_s \cup \{s\}$
8: $\quad$ **for all** $t \in T$ **do**
9: $\qquad$ **if** $t.guard \neq wahr$ **then**
10: $\qquad\quad N_g = N_g \cup \{t.guard\}$
11: $\qquad N_t = N_t \cup \{t\}$
12: $\quad E_{sg} = E_{st} = E_{gt} = E_{ts} = \emptyset$
13: $\quad$ **for all** $t \in T$ **do**
14: $\qquad$ **if** $t.guard = wahr$ **then**
15: $\qquad\quad E_{st} = E_{st} \cup \{(t.source, t)\}$
16: $\qquad$ **else**
17: $\qquad\quad E_{sg} = E_{sg} \cup \{(t.source, t.guard)\}$
18: $\qquad\quad E_{gt} = E_{gt} \cup \{(t.guard, t)\}$
19: $\qquad E_{ts} = E_{ts} \cup \{(t, t.target)\}$
20: **end**

Abbildung 4.10: Algorithmus zur Transformation eines CSMs in ein CFG

deren den Wächter $t.guard$ und die Transition t verbinden, in die Mengen E_{sg} bzw. E_{gt} eingefügt (Zeile 13–19). Der erste Fall wird hierbei dann ausgeführt, wenn der Wächter der Transition immer erfüllt ist, der zweite Fall entsprechend in den übrigen Fällen.

Die Laufzeit und der Platzbedarf des Algorithmus ist offensichtlicherweise linear in der Zahl der Zustände und der Transitionen des Automaten, also durch $O(|S| + |T|)$ beschränkt.

Testdatengenerierung

Zur Generierung der Testdaten, also den Methodensequenzen, werden zuerst in den Knoten des CFGs der Kontext der einzelnen Attribute ermittelt. Eine *Definition* eines Attributs ist genau dann in einem Knoten t, $t \in N_t$, gegeben, wenn in der Aktion der entsprechenden Transition ein Wert zu diesem Attribut zugewiesen wird. Die Referenz eines Attributs wird hingegen in zwei Fälle unterschieden. Ist die Referenz eines Attributs innerhalb der Aktion einer Transition, so wird diese Referenz als *Berechnungsreferenz* in dem entsprechenden Knoten t, $t \in N_t$, bezeichnet. Befindet sich jedoch die Referenz im Rahmen eines Zustandsprädikats oder eines Wächters einer Transition, so wird diese Referenz als *Entscheidungsreferenz* im Knoten s, $s \in N_s$, bzw. g, $g \in N_g$, definiert.

Diese Begriffe werden am Beispiel der `Stack`-Klasse erläutert. Die `Stack`-Klasse umfaßt nur das Attribut *top*. Eine Definition dieses Attributs existiert beispielsweise innerhalb der Transition t_3 und damit in dem entsprechenden Knoten der Abbildung 4.9, da in der Aktion dieser Transition das Attribut inkrementiert wird. Für das Inkrementieren des Wertes muß vorher auf das Attribut zugegriffen werden, d.h. es existiert in diesem Knoten auch eine Berechnungsreferenz. Eine Entscheidungsreferenz ist hingegen in jedem Knoten s, $s \in N_s$, des CFGs gegeben, da der Zustand von *top* abhängt und somit zur Bestimmung des Zustandes ein Zugriff auf das Attribut erfolgt. Im Knoten g_4 befindet sich ebenfalls eine Entscheidungsreferenz, da die Transition t_4 nur erfolgt, wenn $top > 0$ gilt. Insgesamt existieren in den Knoten $\{t_1, t_2, t_3, t_4, t_5, t_8, t_{10}\}$ Definitionen, in den Knoten $\{t_2, t_3, t_4, t_5, t_8, t_{10}\}$ Berechnungsreferenzen und schließlich in den Knoten $\{leer, belegt, voll, g_3, g_4, g_5, g_{10}\}$ der Abbildung 4.9 Entscheidungsreferenzen des Attributs.

Anschließend werden Testdaten auf Grundlage eines Datenflußkriteriums auf dem CFG erzeugt. In diesem Beispiel wird das Kriterium *alle-Definitionen*[13] für die Testdatengenerierung zugrundegelegt. Dieses Kriterium wird genau dann von den Testdaten erfüllt, wenn jede Definition des Attributs durch mindestens eine Referenz getestet wird. Zur Bewertung dieses Kriteriums sei auf [Rie97] verwiesen. In dem obigen Beispiel erfüllen die Methodensequenzen <`Stack(3),push(`*i*`),push(`*i*`),pop(`*i*`),~Stack()`> und <`Stack(2),push(`*i*`),push(`*i*`),pop(`*i*`),pop(`*i*`),~Stack()`> als Testdaten dieses Kriterium, da der Test jeder Definition durch mindestens eine Referenz gegeben ist. Die erste Sequenz fügt zum `Stack`-Objekt zwei Elemente hinzu und entfernt anschließend wieder das oberste Element, d.h. es werden in der CSM der Klasse die Zustände s_0, *leer*, *belegt*, *belegt*, *belegt* und s_f durchlaufen. Damit werden im CFG die Definitionen in den Knoten t_1, t_3, t_4, t_5 durch die entsprechenden Entscheidungsreferenzen in den Knoten *leer* und *belegt* des CFGs getestet. Die zweite Sequenz fügt zu dem `Stack`-Objekt zwei Elemente und entfernt wieder diese, dabei wird durch die Größe des `Stack`-Objekts von zwei Elementen der Zustand *voll* erreicht. Diese Sequenz stellt damit den Test der Definitionen in den Knoten t_{11} und t_{13} durch Entscheidungsreferenzen in den Knoten *belegt* und *leer* sicher. Diese spezielle Testdatenmenge überdeckt die Definition/Referenz Paare $\{(t_1, leer)$, $(t_2, belegt)$, $(t_3, belegt)$, $(t_4, belegt)$, $(t_5, voll)$, $(t_8, belegt)$, $(t_{10}, leer)\}$ und genügt also dem Kriterium *alle-Definitionen*.

[13] Andere Kriterien finden sich in [Rie97].

5 Ein Testverfahren für die prototypbasierte Software-Entwicklung

In diesem Kapitel wird das Testverfahren, das im Rahmen der Diplomarbeit entstanden ist, beschrieben. Im letzten Kapitel wurden verschiedene Verfahren für den strukturellen und den funktionalen Klassentest beschrieben, ohne jedoch den Zusammenhang dieser Verfahren zur prototypbasierten Software-Entwicklung in ausreichendem Maße deutlich zu machen. Abschnitt 5.1 beschreibt die Anwendbarkeit des Verfahrens von Rothermel/Harrold für den strukturellen Test von Prototypen, während in Abschnitt 5.2 die Anwendbarkeit der Verfahren von Tse/Xu und Hong/Kwon/Cha für den funktionalen Test von Prototypen beschrieben wird. Da jedoch der isolierte Test der strukturellen und der funktionalen Merkmale eine relativ unbefriedigende Lösung darstellt, wurde eine Integration dieser Verfahren erarbeitet, die schließlich in Abschnitt 5.3 vorgestellt wird. Durch die Integration dieser Verfahren entsteht ein Testverfahren, das den oben aufgelisteten Anforderungen entspricht und somit das Ziel der Diplomarbeit erfüllt.

5.1 Struktureller Test

Der strukturelle Test im Prototyping kann auf effiziente Weise mit Hilfe des Regressionstestverfahrens von Rothermel/Harrold durchgeführt werden. Dazu werden die einzelnen Klassen des ersten Prototyps einem herkömmlichen strukturellen Test unterzogen und die dazu erforderlichen Testdaten in einer Test-History abgespeichert. Alle anderen Prototypen, die im weiteren Verlauf des Prototypings entstehen, werden dann mit Hilfe des Verfahrens von Rothermel/Harrold getestet. Es soll hier nochmals betont werden, daß hier nur der Test der einzelnen Klassen betrachtet wird. Der Integrationstest der Klassen ist zwar nicht unwichtigt, konnte jedoch aus Zeitgründen nicht berücksichtigt werden und bleibt somit ein offenes Problem. Dieses Verfahren ist auch in der Lage Abschnitte in der Implementierung eines Prototyps zu erkennen, die in dem vorherigen Prototyp nicht vorhanden waren und somit durch die Testdaten nicht überdeckt werden können. Dadurch wird die Generierung neuer Testdaten, die in der Regel für jeden Prototyp anfällt, da jeder Prototyp neue Abschnitte enthält, unterstützt. Einziger Nachteil dieses Verfahrens ist jedoch die Tatsache, daß es den funktionalen Test nicht berücksichtigt.

5.2 Funktionaler Test

Der funktionale Prototypentest kann, allerdings aufwendiger als der strukturelle Test, mit Hilfe der beiden Verfahren von Tse/Xu und Hong/Kwon/Cha durchgeführt werden. Abhängig davon in welcher Form die Spezifikation der einzelnen Klassen vorliegt, muß unter Umständen zuerst ein Automat der Klasse basierend auf dem Verfahren von Tse/Xu erzeugt werden. Dieser Schritt entfällt natürlich in den Fällen, wo die Spezifikation der Klassen schon mit Hilfe von Automaten durchgeführt wurde. Aus dem Automaten der Klasse wird anschließend der CFG der Klasse nach dem Algorithmus in der Abbildung 4.10 entwickelt. Das Verfahren nach Hong/Kwon/Cha sieht nach der Generierung des CFGs die Generierung der Testdaten, die ein Datenflußkriterium auf diesem CFG erfüllen, vor. Der funktionale Test, strikt nach den Verfahren von Tse/Xu und Hong/Kwon/Cha durchgeführt, betrachtet die einzelnen Prototypen als isolierte Software-Produkte und führt häufig zur Generierung derselben Testdaten. Der funktionale Test entspricht in dieser Form also der ersten Strategie in Abschnitt 4.2. Diese Strategie verursacht jedoch, wie schon erläutert, eine Ressourcenverschwendung.

5.3 Gleichzeitiger Test funktionaler und struktureller Merkmale

Das Verfahren, das im Rahmen der Diplomarbeit entstanden ist, wird am Beispiel der `Stack`-Klasse beschrieben. Der strukturelle Test von Prototypen kann, wie oben festgestellt wurde, in effizienter Weise auf der Basis des Regressionstestverfahrens von Rothermel/Harrold durchgeführt werden. Aus diesem Grund wurde versucht, Verfahren für den funktionalen Test in das Verfahren von Rothermel/Harrold zu integrieren, um dadurch die Vorteile dieses Regressionstestverfahrens zu erhalten. Der Algorithmus in den Abbildungen 4.4 und 4.5 mußte dazu etwas angepaßt werden. Bevor jedoch auf die Veränderungen des Algorithmus eingegangen wird, soll die dem Algorithmus zugrundeliegende Datenstruktur erläutert werden.

5.3.1 Erweiterter Class Dependence Graph (xClDG)

Beim Klassentest nach Hong/Kwon/Cha werden die Testdaten auf der Grundlage eines Datenflußkriteriums, das auf den Attributen der Klasse erfüllt sein muß, erzeugt.

Dabei wird eine Definition eines Attributs nur mit einer Referenz, die sich in einer anderen Methode befindet, zu einem Definition/Referenz-Paar zusammengefaßt, da die Interaktion der Methoden über die Attribute getestet werden soll. ClDGs stellen zwar in einer übersichtlichen Form Daten- und Kontrollabhängigkeiten innerhalb der Methoden einer Klasse dar, zeigen jedoch keine Datenabhängigkeiten an, die durch die Attribute der Klasse zwischen den Methoden entstehen. Aus diesem Grund wurden ClDGs um Kanten ergänzt, die diese Abhängigkeiten verdeutlichen. Diese neuen Kanten sollen im folgenden *inter method data dependence* (*mdd*) *Kanten* genannt werden. Eine mdd-Kante führt also von der Definition eines Attributs in einer Methode zu einer Referenz dieses Attributs in einer anderen Methode. Ein erweiterter ClDG soll *Extended Class Dependence Graph* (*xClDG*) bezeichnet werden.

Methoden können aber im Prinzip in jeder Reihenfolge aufgerufen werden, so daß sich hier die Frage stellt, welche Referenz datenabhängig zu welcher Definition ist. Sind beispielsweise in zwei Methoden A und B je eine Definition und eine Referenz desselben Attributs vorhanden, so kann entweder die Definition in Methode A mit der Referenz in Methode B oder die Definition in Methode B mit der Referenz in Methode A durch eine mdd-Kante verbunden werden. Die Entscheidung zwischen den beiden Fällen hängt nur von der Reihenfolge ab, in der die Methoden aufgerufen werden. Wird die erste Methode vor der zweiten Methode aufgerufen, so wird bei der Referenz in Methode B auf einen Wert des Attributs zugriffen, der durch die Definition in der anderen Methode gesetzt wurde. Die Definition in Methode A ist also mit der Referenz in Methode B durch eine mdd-Kante zu verbinden. In dem anderen Fall entsprechend umgekehrt. Ein xClDG legt also im Gegensatz zu einem gewöhnlichem ClDG eine bestimmte Reihenfolge fest, in der die Methoden ausgeführt werden können.

Der CFG der Klasse, der mit Hilfe des Algorithmus in Abbildung 4.10 generiert wird, zeigt die möglichen Methodensequenzen an. Es kann auf der Basis des CFGs der Klasse bestimmt werden, ob eine Referenz in einer Methode von einer Definition in einer anderen Methode erreicht werden kann und somit eine Datenabhängigkeit zwischen dieser Definition und der Referenz besteht. Ist eine Datenabhängigkeit zwischen der Definition und der Referenz gegeben, so sind sie also durch eine mdd-Kante zu verbinden.

Abbildung 5.1 zeigt den xClDG der Klasse `Stack`. Die mdd-Kanten sind in dieser Abbildung punktiert gezeichnet, wobei der ClDG der `List`-Klasse zur Übersichtlichkeit weggelassen wurde. Die mdd-Kanten sind in dieser Abbildung entsprechend den Definition/Referenz-Paaren, die zur Sicherstellung des Kriteriums *alle-Definitionen* durch Testdaten überdeckt werden müssen[1], gesetzt.

Zum Erweitern eines ClDGs müssen also Definitionen und Referenzen eines Attributs entsprechend einem Datenflußkriterium zueinander zugeordnet und mdd-Kanten zum ClDG hinzugefügt werden, deren Ursprung eine Definition und das Ziel die zugeordne-

[1]Siehe Beispiel in Abschnitt 4.4.2.

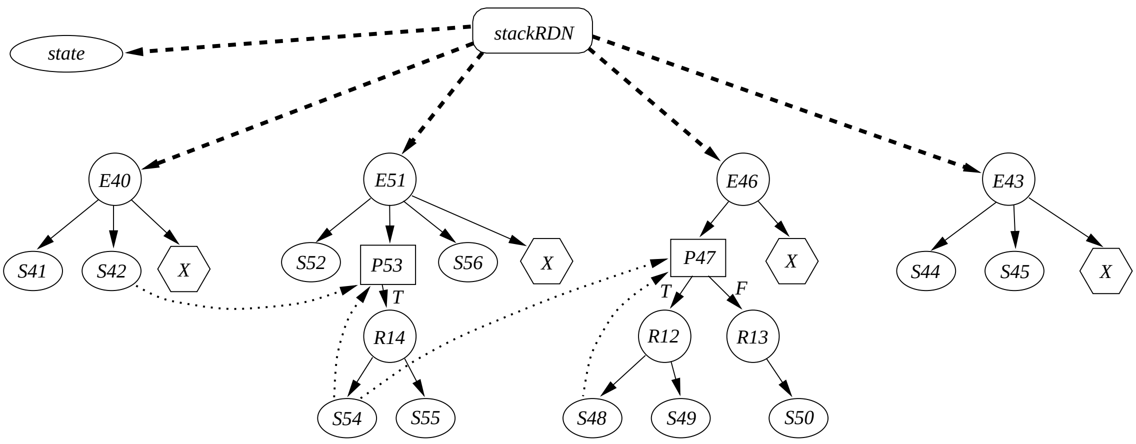


Abbildung 5.1: `Stack`-Klasse als xClDG

te Referenz ist. Dazu müssen die Anweisungen in der Implementierung der Klasse bestimmt werden, die die Definition und die Referenz des Attributs ausführen. Während eine Definition eines Attributs häufig in der Form „`x = ...`“ in der Implementierung erscheint, ist insbesondere eine Entscheidungreferenz jedoch schwieriger in der Implementierung zu identifizieren. Beispielsweise wurde in Abschnitt 4.4.2 zum Erfüllen des Kriteriums *alle-Definitionen* das Paar $(t_1, leer)$ als ein Definition/Referenz-Paar bestimmt, das durch die Testdatenmenge überdeckt werden muß. Während die Definition im Knoten t_1 relativ einfach als die Zeile `S42 {top = 0;}` im Konstruktor identifiziert werden kann, stellt sich die Frage, wo sich die zugeordnete Entscheidungsreferenz befindet. Die Referenz ist in diesem Fall ein Zustandsprädikat, der im leeren Zustand des Stacks erfüllt ist. Der Zustand eines Objektes ist jedoch ein abstrakter Begriff, der nur in der Spezifikation der Klasse erscheint[2]. Aus diesem Grund wird wohl in den wenigsten Fällen in der Implementierung der Klasse ein Ausdruck vorkommen, der genau die Form des Zustandsprädikats besitzt. Da allerdings das Verhalten der Klasse abhängig vom Zustand ist, muß eine Fallunterscheidung in der Methode, die nach der Definition aufgerufen wird, vorhanden sein, die abhängig vom Wert des veränderten Attributs ein bestimmtes zustandabhängiges Verhalten auslöst. Diese Fallunterscheidung repräsentiert also indirekt ein Zustandsprädikat.

Im dem obigen Beispiel wurde angenommen, daß nach Aufruf des Konstruktors `Stack(`n`)` die Methode `push(`i`)` aufgerufen wird. Deshalb wurde der Knoten $S42$ mit dem Knoten $P53$ durch eine mdd-Kante verbunden, da dieser Knoten eine Fallunterscheidung enthält, die abhängig von dem zustandsbestimmenden Attribut *top* verschiedene Alternativen ausführt und somit das Verhalten im Zustand *leer* sichtbar macht.

Bei Entscheidungsreferenzen stellt sich die Situation dagegen ähnlich wie bei Definitionen dar, d.h. häufig kann relativ einfach eine Anweisung in der Implementierung

[2] Zumindestens in der Programmiersprache C++.

```
procedure xCompare(N, N') : (T'', T''')
input N, N' : Region- oder Predicate-Knoten in G und G'
output T'' : Testdaten für den strukturellen Test
  T''' = {(in_1, in_2)} : Testdaten für den funktionalen Test
    in_1 : Eingabe, die eine Definition unterhalb von N' ausführt
    in_2 : Eingabe, die eine mdd-abhängige Referenz in einer anderen Methode ausführt
begin
  Markiere N und N' visited
  if GetCorresp(N, N') then
    T'' = ∅
    T''' = ∅
    for all neuer/veränderter cd-Nachfolger A von N', gelöschter cd-Nachfolger A von N do
      for all dd-Nachfolger B von A do
        if B ist visited then
          T'' = T'' ∪ N.history ∩ D.history, D cd-Vorgänger von B
        else
          B.affected = B.affected ∪ N.history
      for all mdd-Nachfolger B von A do
        if B ist visited then
          T''' = T''' ∪ {(in_1, in_2)}, ∀in_1 ∈ N.history, in_2 ∈ D.history, D cd-Vorgänger von B
        else
          B.mdd_affected = B.mdd_affected ∪ N.history
    for all cd-Nachfolger A von N oder N', A.affected ≠ ∅ do
      T'' = T'' ∪ N.history ∩ A.affected
    for all cd-Nachfolger A von N oder N', A.mdd_affected ≠ ∅ do
      T''' = T''' ∪ {(in_1, in_2)}, ∀in_1 ∈ A.mdd_affected, ∀in_2 ∈ N.history
    for all cd-Nachfolger A von N und N', A Endknoten do
      Markiere A visited
    for all cd-Nachfolger A von N, A nicht visited markiert do
      T'' = T'' ∪ xCompare(A, A'), A' zu A äquivalenter Nachfolger von N'
  else
    T'' = N.history
    T''' = ∅
  return (T'', T''')
end
```

Abbildung 5.2: `xCompare`-Prozedur

bestimmt werden, die die Referenz beinhaltet. Analoge Überlegungen führen zu allen anderen mdd-Kanten in der Abbildung 5.1.

5.3.2 Auswahl der Testdaten

Der Algorithmus in den Abbildungen 4.4 und 4.5 wurde an einigen Stellen erweitert, um auch die mdd-Kanten verarbeiten zu können. Die Veränderungen des Algorithmus beschränken sich aber nur auf die `Compare`-Prozedur. In der Abbildung 5.2 ist die erweiterte Version dieser Prozedur dargestellt.

Die Prozedur `xCompare` bestimmt ausgehend von den Region- bzw. Predicate-Knoten

N und N' als Eingabe diejenigen Testdaten aus einer gegebenen Testdatenmenge, die für den strukturellen und den funktionalen Test des Knotens N' notwendig sind. Die Testdaten für den strukturellen Test des Knotens N' werden genau nach dem Verfahren von ROTHERMEL/HARROLD bestimmt, so daß im folgenden nur die Selektion der Testdaten für den funktionalen Klassentest beschrieben werden soll. Die Testdaten für den funktionalen Test bestehen aus zwei Komponenten. Die erste Komponente ist die Eingabe, die eine veränderte Definition überdeckt, während die zweite Komponente eine mdd-abhängige Referenz in einer anderen Methode ausführt. Da sich Definition und Referenz in verschiedenen Methoden befinden, können sie im Gegensatz zum strukturellen Test, der sich nur auf eine einzige Methode bezieht, nicht mit derselben Eingabe überdeckt werden.

Die Verarbeitung der mdd-Kanten findet in der `xCompare`-Funktion in den Zeilen 18–22 und 25–26 statt. Der Rest der Prozedur ist im wesentlichen identisch mit der `Compare`-Funktion aus Abbildung 4.5 und soll deshalb nicht erläutert werden. In den Zeilen 12 bis 22 werden neue oder veränderte kontrollabhängige Nachfolger A des Knotens N' bzw. gelöschte Nachfolger A des Knotens N betrachtet. Nach der Traversierung der datenabhängigen Nachfolger von A in den Zeilen 13 bis 17 der `xCompare`-Prozedur, werden in den Zeilen 18 bis 22 die mdd-Nachfolger dieses Knotens berücksichtigt. Falls ein mdd-Nachfolger schon besucht wurde und damit *visited* markiert ist, werden zur Testdatenmenge T''' alle Eingabepaare (in_1, in_2) hinzugefügt, die eine Eingabe in_1 zur Überdeckung der veränderten Definition eines Attributs und eine Eingabe in_2 zur Überdeckung einer zur Definition mdd-abhängigen Referenz umfassen. Der zuverlässige Test des Knotens N' erfordert alle möglichen Paare, die mit einer Komponente aus $N.history$ und einer Komponente aus der Region-History des Knotens, zu der die Referenz kontrollabhängig ist, gebildet werden können (Zeile 19f). Falls der mdd-Nachfolger B des veränderten Knotens A jedoch noch nicht besucht worden ist, so werden die Eingaben in $N.history$ zu $B.mdd\_affected$ hinzugefügt. Dadurch wird deutlich gemacht, daß diese Referenz von einer Veränderung in einer Definition, die genau von den Testdaten in $B.mdd\_affected$ überdeckt wird, beeinflußt werden kann.

In den Zeilen 25 bis 26 werden anschließend zu N kontrollabhängige Knoten A betrachtet, die noch nicht besucht worden sind und die zu veränderten Definitionen mdd-abhängig sind. Für diese Knoten gilt: $A.mdd\_affected \neq \emptyset$. Da diese Knoten noch nicht besucht worden sind, wurden zur Testdatenmenge T''' auch keine Eingaben hinzugefügt, die diesen Knoten testen. Aus diesem Grund werden in T''' alle Eingaben eingefügt, die eine Komponente in_1 aus $A.mdd\_affected$ und eine Komponente in_2 aus $N.history$ enthalten.

Für eine Beispielsanwendung dieses Verfahrens sei auf den Kapitel 6 verwiesen.

6 Zinsderivate und deren Bewertung

Dieses Kapitel stellt das benötigte finanzwirtschaftliche Wissen für das Verständnis der Diplomarbeit zur Verfügung. Abschnitt 6.1 gibt dazu einen Überblick über originäre Finanzinstrumente, wie Aktien und Anleihen, während in Abschnitt 6.2 derivative Finanzinstrumente erläutert werden. Zu den derivaten Finanzinstrumenten gehören insbesondere Futures (Abschnitt 6.2.1), Optionen (Abschnitt 6.2.2) und Zinsderivate (Abschnitt 6.2.3). Letztere werden in den darauffolgenden Abschnitten ausführlicher behandelt. Es werden in Abschnitt 6.3 die verschiedenen Modelle, die zur Bewertung von Zinsderivaten eingesetzt werden können, beschrieben und das Zinsstrukturmodell nach Ritchken/Sankarasubramanian, welches auf dem Heath/Jarrow/Morton-Modell basiert, ausführlicher erläutert.

6.1 Originäre Finanzinstrumente

6.1.1 Aktien

Eine der wichtigsten Finanzinstrumente ist die Aktie [Sch98]:

> Eine *Aktie* verbrieft eine Beteiligung an einem Unternehmen in der Rechtsform einer Aktiengesellschaft (AG) oder einer Kommanditgesellschaft auf Aktien (KGaA).

Die beiden Unternehmensformen sind rechtlich im Aktiengesetz [AktG] geregelt. Der Besitzer einer Aktie ist also ein Teilhaber des Unternehmens, das die Aktie emittiert hat. Als Teilhaber besitzt der Aktieninhaber Rechte und auch Pflichten gegenüber der Gesellschaft und der Leitung des Unternehmens. Zu den Rechten gehören das Mitgliedschaftsrecht auf der jährlichen Hauptversammlung und Vermögensrechte, wie zum Beispiel das Recht auf Dividende, also an einer Gewinnbeteiligung. Eine Pflicht besteht insbesondere in einer Einzahlung zum Grundkapital. Das Grundkapital, das bei Aktiengesellschaften auch *gezeichnetes Kapital* genannt wird, entspricht der Summe des Nennwerts aller ausgegebenen Aktien. Der Nennwert ist allerdings von dem Preis einer Aktie zu unterscheiden. Während der Nennwert ein Buchwert ist, ergibt sich der Preis einer Aktie aus Angebot und Nachfrage. Für eine Erläuterung der verschiedenen Aktienarten sei auf [Sch98] verwiesen.

Die Organe einer Aktiengesellschaft sind die Hauptversammlung, der Aufsichtsrat und der Vorstand.

- Hauptversammlung
  Die Hauptversammlung der Aktionäre beschließt beispielsweise über die Änderung der Kapitalgrundlage[1], wie durch Emission neuer[2] Aktien, oder die Auflösung der Gesellschaft. In der Hauptversammlung wird aber auch über die Verwendung des Bilanzgewinns abgestimmt. Das Stimmrecht jedes Aktionärs ist dabei abhängig von der Zahl seiner Aktien, d.h. an seinem Anteil am Grundkapital.

- Aufsichtsrat
  Ein weiteres Organ der Aktiengesellschaft ist der Aufsichtsrat. Der Aufsichtsrat bestellt die Geschäftsführung, also den Vorstand, und überwacht diesen. Gegebenfalls kann der Aufsichtsrat den Vorstand wieder abberufen. Der Aufsichtsrat setzt sich aus höchstens 21 Vertretern der Aktionäre und der Arbeitnehmer zusammen, die in der Hauptversammlung auf maximal vier Jahre gewählt werden. Die Hauptversammlung kann als oberstes Organ der Aktiengesellschaft den Aufsichtsrat auch wieder entlassen.

- Vorstand
  Das dritte Organ einer Aktiengesellschaft ist der Vorstand. Der Vorstand einer Aktiengesellschaft hat die Aufgabe, das Unternehmen in eingener Verantwortung zu führen. Die Mitglieder des Vorstandes werden zu diesem Ziel vom Aufsichtsrat für die Dauer von fünf Jahren bestellt, und können sowohl vom Aufsichtsrat als auch von der Hauptversammlung entlassen werden. Besteht der Vorstand aus mehr als einer Person, so hat der Aufsichtsrat zusätzlich die Aufgabe den Vorsitzenden des Vorstandes zu bestimmen.

Die Aktiengesellschaft ist also eine Unternehmensform, in der Eigentum und Führung getrennt sind, d.h. die Aktionäre sind zwar die Besitzer des Unternehmens, haben aber nicht die Geschäftsführung inne.

6.1.2 Anleihen

In der vorliegenden Diplomarbeit werden Anleihen gemäß [Hav93, Sch98] folgendermaßen definiert (vgl.[BV94]):

[1] Die Änderung der Kapitalgrundlage wird *Kapitalerhöhung* bzw. *-herabsetzung* [FH94] genannt.
[2] In diesem Zusammenhang werden die neu emittierten Aktien auch als *junge* Aktien bezeichnet.

> Eine *Anleihe* ist ein Wertpapier, das eine Schuldaufnahme am Kapitalmarkt gegen Ausgabe einer Schuldverschreibung verbrieft.

Der Käufer bzw. *Gläubiger* einer Anleihe erwirbt mit dem Kauf der Anleihe das Recht auf Tilgung und Verzinsung des Nennwerts der Anleihe. Der Nennwert der Anleihe wird zur Erleichterung der Emission und des Handels in Teilbeträge gestückelt. Die durch die Stückelung entstanden Finanzinstrumente werden *Teilschuldverschreibungen* genannt. Häufig sind auch andere Bezeichnungen für Anleihen, wie Bonds, Renten, Obligationen oder Schuldverschreibungen, anzutreffen.

Die Tilgungs- und Zinszahlungen sind innerhalb der Laufzeit der Anleihe zu zahlen. Kurzfristige Anleihen besitzen eine Laufzeit von bis zu vier Jahren, während die Laufzeit bei mittelfristigen Anleihen zwischen vier bis acht Jahren liegt. Liegt die Laufzeit bei über acht Jahren, so wird von langfristigen Anleihen gesprochen.

Als *Emittent* von Anleihen können sowohl staatliche Institutionen als auch Unternehmen auftreten. Bei Anleihen von staatlichen Institutionen wird beispielsweise zwischen Bundesschatzbriefen, Kommunalobligationen oder Pfandbriefen unterschieden, während Anleihen von emissionsfähigen Unternehmen als Industrieobligationen bezeichnet werden.

Tilgung

Die Tilgung des Nennwerts der Anleihe kann entweder planmäßig oder außerplanmäßig erfolgen:

- Planmäßige Tilgung
  Die planmäßige Tilgung unterscheidet wiederum drei Fälle:
  Bei der *gesamtfälligen Anleihe* wird der Nennwert der Anleihe am Ende der Laufzeit ausgezahlt. Bei einer *Annuitäten-Anleihe* erfolgt die Tilgung derart, daß die Raten als Summe aus Tilgung und Zinszahlung konstant sind. Dagegen wird bei einer *Auslosungsanleihe* die zurückzuzahlenden Anleihen vom Emittenten ausgelost.

- Außerplanmäßige Tilgung
  Die außerplanmäßige Tilgung tritt bei vorzeitiger Kündigung des Emittenten auf. In besonderen Fällen kann auch der Käufer der Anleihe das Recht der vorzeitigen Kündigung besitzen.

Verzinsung

Der Emittent der Anleihe hat dem Käufer Zinszahlungen auf den Nennwert der Anleihe zu leisten. In der Abbildung 6.1 sind die verschiedenen Formen der Verzinsung mit den entsprechenden Anleihen dargestellt.

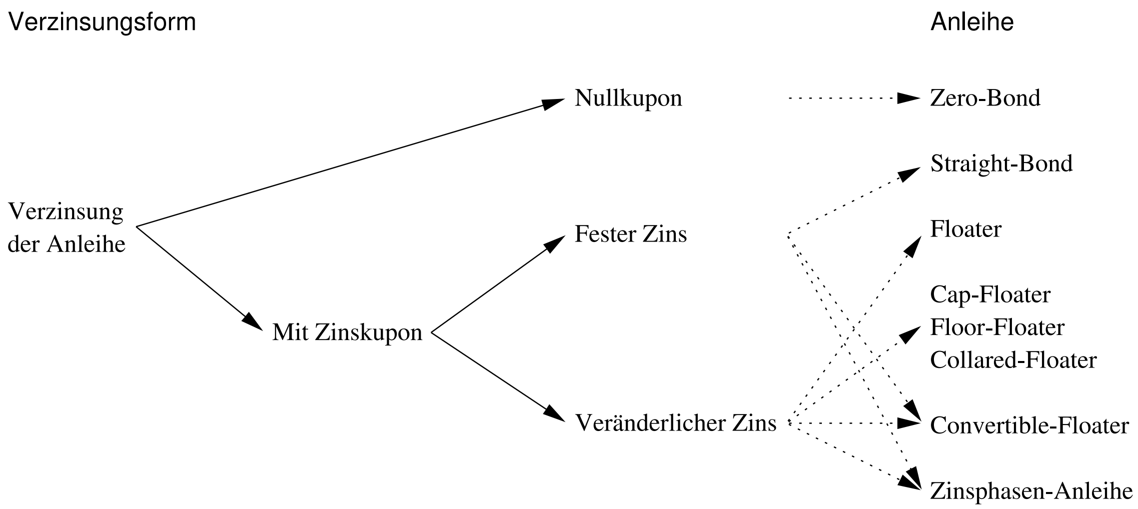


Abbildung 6.1: Formen der Verzinsung bei Anleihen

- Festverzinsliche Anleihe
  Bei einer festverzinslichen Anleihe, die auch als *Straight-Bond* bezeichnet wird, erfolgen die Zinszahlungen in regelmäßigen Zeitpunkten zu einem festen Zinssatz. Die Zinszahlungen sind üblicherweise durch Zinsscheine bzw. Kupons, die der Anleihe beigefügt sind, verbrieft.

- Variabel verzinsliche Anleihe
  Eine Anleihe mit variablem Zinssatz, die auch *Floating Rate Note* genannt wird, verzinst sich dagegen in Abhängigkeit zu einem Referenzzinssatz, wie zum Beispiel dem London Interbank Offered Rate (LIBOR) oder Frankfurt Interbank Offered Rate (FIBOR). Beides sind Zinssätze, die Banken untereinander für kurzfristige Kredite erheben. Die Differenz zwischen dem Referenzzinssatz und dem effektiven Zinssatz der Anleihe wird in diesem Zusammenhang als *Spread* bezeichnet und richtet sich nach der Kreditwürdigkeit, also der Bonität, des Schuldners. Bei einer variabel verzinsten Anleihe können zusätzlich Besonderheiten zur Verzinsung vereinbart werden:

  – Bei der variablen Verzinsung kann die Schwankungsbreite des Zinssatzes eingeschränkt werden. Ein *Cap-Floater* ist beispielsweise eine Anleihe, bei der der Zinssatz einen festgelegten Grenzzinssatz, den *Cap*, nicht überschreiten kann. Dagegen ist ein *Floor-Floater* eine Anleihe, deren Zinssatz eine Untergrenze, den *Floor*, nicht unterschreiten kann. Liegt sowohl eine Ober- als auch eine Untergrenze des Zinssatzes vor, so wird vom einem *Collared-Floater* gesprochen.

  – Eine andere Sonderform der variablen Verzinsung liegt beim *Convertible Floating Rate Note* vor. Diese Anleihe beinhalten, entweder für den Emittent oder für den Gläubiger, das Recht, die variabel verzinste Anleihe in eine festverzinste Anleihe umzutauschen.

- Nullkupon-Anleihe
  Die Zinszahlungen können bei einer Anleihe auch gänzlich fehlen, der Anleihe hängen dann keine Zinskupons an. Entsprechend wird in diesen Fällen auch von Nullkupon-Anleihen oder *Zero-Bonds* gesprochen. Tatsächlich ergeben sich die Zinszahlungen aus der Differenz zwischen dem Emissionskurs und der Rückzahlung der Anleihe am Ende der Laufzeit, d.h. der Gläubiger kauft die Anleihe zu einem Preis, der schon um die Zinszahlungen verringert ist. Ein Zero-Bond bietet also nur einen einmaligen Ertrag am Ende der Laufzeit. Zero-Bonds können auch aus festverzinslichen Anleihen, deren Zinskupons abgetrennt werden können, entstehen. In diesem Fall wird die Anleihe ohne Zinszahlungen, also der Zero-Bond, und die Zinskupons getrennt gehandelt. Eine festverzinsliche Anleihe, die diese Möglichkeit anbietet, wird *Stripped-Bond* bezeichnet.

- Zinsphasen-Anleihe
  Eine Mischform aus festverzinslichen und variabel verzinslichen Anleihen bieten Zinsphasen-Anleihen an. Diese Anleihen zahlen in der Regel zu Beginn der Laufzeit einen festen Zinssatz, danach einen variabeln Zinssatz und am Ende der Laufzeit wieder einen festen Zinssatz. Der variable Zinssatz orientiert sich wiederum an einen Referenzzinssatz.

Bestimmte Sonderformen von Anleihen können neben den oben erläuterten Gläubigerrechten auch Anteilrechte an dem emittierenden Unternehmen beinhalten. Dazu gehört die *Wandelschuldverschreibung*, die innerhalb einer Umtauschfrist in Aktien umgetauscht werden kann. Eine *Options-Anleihe* beinhaltet dagegen noch zusätzlich das Recht, Aktien des Unternehmens zu einem bestimmten Preis zu beziehen. Dieses Recht kann getrennt von der Anleihe gehandelt werden und wird üblicherweise *Optionsschein* bezeichnet. Beide Anleiheformen haben also Eigenschaften von Optionen[3].

6.2 Derivative Finanzinstrumente

6.2.1 Futures-Kontrakte

Futures-Kontrakte gehören zu den einfacheren Derivaten. Ein Futures-Kontrakt wird in der vorliegenden Arbeit folgendermaßen definiert (vgl. [Hul96, Sch98]):

> Ein *Futures-Kontrakt* ist eine Vereinbarung zwischen zwei Parteien, einen Basiswert zu einem bestimmten Zeitpunkt in der Zukunft zu einem festgelegten Preis, dem *Lieferpreis*, zu kaufen bzw. zu verkaufen.

[3] Siehe Abschnitt 6.2.2.

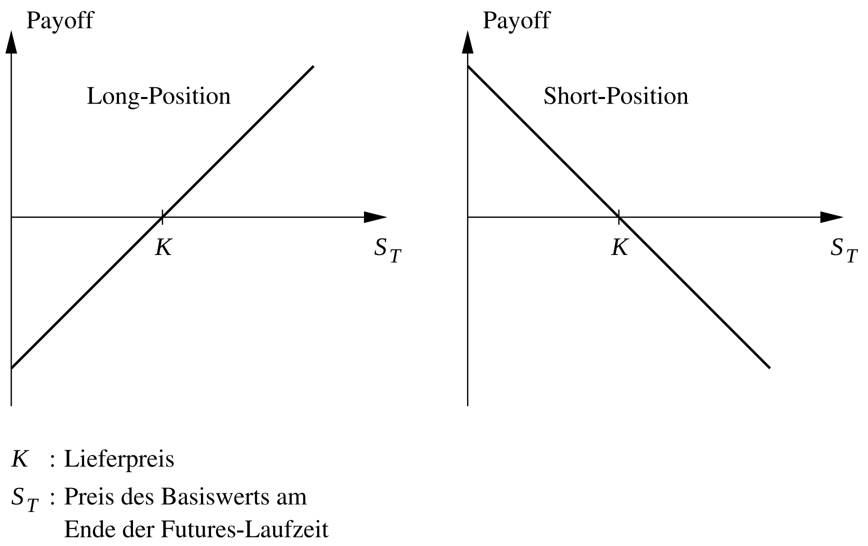


Abbildung 6.2: Payoff-Diagramm bei Futures-Kontrakten

Der Basiswert kann dabei im Prinzip jedes Gut sein, das handelbar ist. Üblich sind Futures auf anderen Finanzinstrumenten, wie Aktien und Anleihen, auf Devisen oder auch auf Rohstoffen, wie Rohöl oder Kaffee. Der spätere Käufer des Basiswerts befindet sich dabei in der sogenannten *Long-Position*, während der Verkäufer des Basiswerts dagegen eine *Short-Position* eingeht.

Der Lieferpreis wird bei der Initiierung des Futures-Kontrakts so bestimmt, daß der Futures-Kontrakt einen Marktwert von Null hat. Damit entstehen für beide Parteien bei Abschluß des Futures-Kontraktes keine Kosten. Der *Futures-Preis* ist in diesem Zusammenhang derjenige Lieferpreis, bei dem der Futures-Kontrakt einen Marktwert von Null hat. Zum Zeitpunkt der Initiierung ist also der Lieferpreis des Futures und der Futures-Preis identisch. Für die Bestimmung des Futures-Preises sei auf [CW92, Hul96] verwiesen.

Futures-Preis und Lieferpreis sind normalerweise nur zum Zeitpunkt der Initiierung des Futures identisch. Durch Preisveränderungen des Basiswerts ändert sich auch der Futures-Preis, also derjenige Lieferpreis des Futures, bei dem der Futures-Kontrakt einen Marktwert von Null hat. Da der ursprüngliche Lieferpreis des Futures natürlich nicht nachträglich verändert werden kann, ist die Position im Futures-Kontrakt nach der Initiierung für eine der beiden Parteien vorteilhaft. Damit ist für diese Partei der Marktwert des Futures größer Null. Dieser Marktwert ist am Ende der Futures-Laufzeit für die Partei in der Long-Position gleich der Differenz aus dem Kassapreis des Basiswerts und dem Lieferpreis des Futures. Der Marktwert des Futures in Short-Position ist entsprechend die Differenz aus dem Lieferpreis und dem Kassapreis des Basiswerts. Diese Differenzen werden als *Payoff* in der entsprechenden Position bezeichnet. In Abbildung 6.2 ist der Payoff eines Futures in den beiden möglichen Positionen aufgezeichnet.

Die bisherigen Ausführungen haben sich zwar auf Futures bezogen, gelten aber in gleicher Form auch für Forward-Kontrakte. Ein *Forward-Kontrakt* [CW92, Hul96] stellt, wie ein Futures-Kontrakt, eine Vereinbarung zwischen zwei Parteien zum Kauf bzw. Verkauf eines Basiswerts dar. Der Unterschied zwischen den beiden Kontrakten ist vor allem, daß Futures standartisiert sind und an Terminbörsen gehandelt werden, während Forward-Kontrakte eigentlich nur auf Over-The-Counter Märkten angeboten werden.

Ein weiterer Unterschied zwischen Futures- und Forward-Kontrakten ist börsentägliche Abrechnung bei Futures. An jedem Börsentag werden die Gewinne und Verluste, die sich an diesem Tag ergeben haben, durch *Marking-To-Market* der Terminbörse direkt auf den *Margin-Konten* der beiden Parteien verrechnet [Hul96].

6.2.2 Optionen

Ein Options-Kontrakt ist im Gegensatz zu einem Futures-Kontrakt nur mit Verpflichtungen für eine der beiden Parteien, die den Options-Kontrakt abschließen, verbunden. Ein Options-Kontrakt, oder kurz Option, ist folgendermaßen definiert [Hul96]:

> Eine *Option* verbrieft dem Käufer der Option das Recht, einen Basiswert zu einem bestimmten Zeitpunkt in der Zukunft zu einem festgelegten Preis vom Verkäufer der Option zu kaufen (Call-Option) bzw. dem Verkäufer der Option zu verkaufen (Put-Option).

Als Basiswert kommen dabei wiederum andere Finanzinstrumente, wie Aktien, Futures oder sogar Optionen, Rohstoffen und Devisen in Frage. Der Käufer hat für sein Recht zum Kauf oder Verkauf dem Verkäufer der Option, der auch *Stillhalter* bezeichnet wird, einen Options-Preis bzw. eine *Options-Prämie* zu zahlen. Optionen, die zum Kauf des Basiswerts berechtigen, werden *Call-Optionen* genannt, während bei Verkaufs-Optionen auch von *Put-Optionen* gesprochen wird. Der bei Abschluß der Option festgelegte Preis für den Basiswert wird auch *Ausübungspreis* oder *Strike* bezeichnet.

Es soll nochmals betont werden, daß eine Option nicht zum Kauf oder Verkauf verpflichtet. Der Käufer der Option hat das Recht dazu, aber nicht die Verpflichtung, d.h. eine Option ist im Gegensatz zu einem Futures-Kontrakt ein asymmetrisches Finanzinstrument. Ein anderer Unterschied zwischen Optionen und Futures ist auch, daß der Käufer der Option dem Verkäufer eine Prämie zu zahlen hat, während bei der Initiierung eines Futures keine Zahlungen zwischen den beiden Vertragspartnern stattfinden.

Je nach den möglichen Zeitpunkten für die Ausübung der Option wird von europäischen, amerikanischen oder von Bermuda-Optionen gesprochen. *Europäische Optionen* berechtigen nur am Ende der Laufzeit zum Kauf bzw. zum Verkauf des Basis-

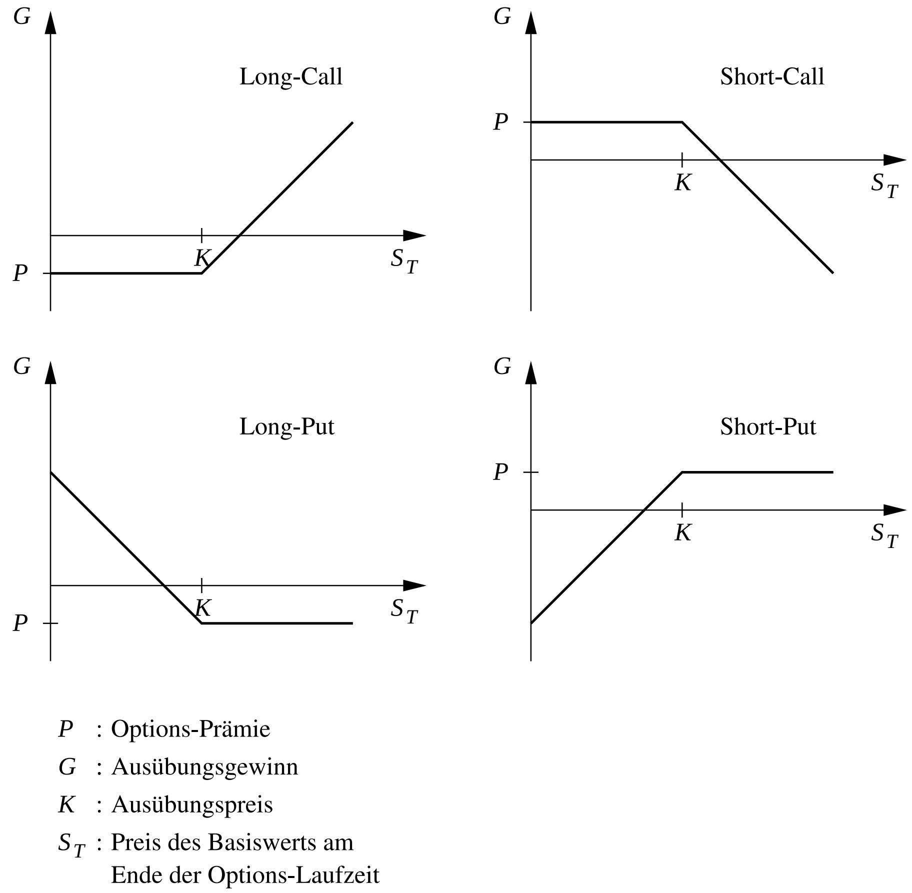


Abbildung 6.3: Ausübungsgewinn bei Optionen

werts, während *amerikanische Optionen* einen beliebigen Zeitpunkt bis zum Laufzeitende zulassen. Bei *Bermuda-Optionen*[4] werden Zeitintervalle innerhalb der Laufzeit festgelegt, in denen die Optionen ausgeübt werden können.

Bei Optionen können vier verschiedene Positionen eingegangen werden. Der Käufer einer Option befindet sich in einer *Long-Position*, während der Verkäufer, in analoger Weise zu Futures-Kontrakten, eine *Short-Position* eingeht. Zusätzlich wird zwischen Call- und Put-Optionen unterschieden, d.h. das beispielsweise der Stillhalter einer Call-Option eine Short-Call-Position hält. In der Abbildung 6.3 sind die möglichen Gewinne und Verluste am Ende der Options-Laufzeit in den einzelnen Positionen in Abhängigkeit vom Preis des Basiswerts dargestellt.

Im folgenden wird beispielhaft die Long-Call-Position, d.h. der Kauf einer Kaufoption, beschrieben. Der Käufer der Call-Option hat beim Eintreten in die Long-Call-Position die Options-Prämie P an den Stillhalter der Option zu zahlen. Obwohl er nun das Recht zum Kauf hat, wird er dieses Recht nur dann zu dem möglichen Zeitpunkt T ausüben, wenn der Preis S_T des Basiswerts *über* oder *gleich* dem Ausübungspreis K

[4] Bermuda liegt zwischen Europa und Amerika.

liegt, also die Option *im* bzw. *am Geld* ist. Sonst könnte er den Basiswert am Kassamarkt billiger kaufen. In diesem Fall liegt sein Gewinn bei $S_T - K - P$ Geldeinheiten (GE). Ist allerdings der Preis des Basiswerts *unter* dem Ausübungspreis, also die Option *aus dem Geld*, dann wird er die Option nicht ausüben und sie ist damit wertlos. Denn wenn er die Option ausüben würde, würde er für den Basiswert den Ausübungspreis bezahlen, der aber über dem Preis des Basiswerts am Kassamarkt ist. In diesem Fall liegt sein Gewinn bei $-P$ GE, da die Options-Prämie unabhängig von der späteren Ausübung zu zahlen ist. Zusammengefaßt ist der Gewinn in der Long-Call-Position gleich $\max(S_T - K, 0) - P$ GE.

Der *Payoff*, d.h. die Zahlung bei der Ausübung, ergibt ohne die Berücksichtigung der Prämienzahlung. Bei der Long-Call-Position ist der Payoff zum Zeitpunkt T also $\max(S_T - K, 0)$ GE.

Options-Preis zum Zeitpunkt der Ausübung

Aus den bisherigen Ausführungen geht hervor, daß die Options-Prämie zum Zeitpunkt der Ausübung bei einer Call-Option $\max(S_T - K, 0)$ und bei einer Put-Option $\max(K - S_T, 0)$ sein muß. Denn wäre beispielsweise die Options-Prämie C bei einer Call-Option kleiner als $\max(S_T - K, 0)$, so könnte ein risikoloser Gewinn erwirtschaftet werden, indem eine Call-Option zu C gekauft und gleichzeitig eine Call-Option mit gleicher Austattung zu $\max(S_T - K, 0)$ verkauft wird. Diese Strategie ermöglicht einen risikolosen Gewinn, da zum einen die Einzahlungen aus dem Verkauf größer sind als die Auszahlungen aus dem Kauf, und zum anderen die Verpflichtungen aus der verkauften Option genau dieselben sind, wie die Rechte aus der gekauften Option. Analoge Überlegungen zeigen auch, daß die Options-Prämie nicht größer als $\max(S_T - K, 0)$ sein kann. Solche risikolosen Gewinnmöglichkeiten, die auch als *Arbitrage* bezeichnet werden, können nicht für längere Zeit bestehen. Denn durch den Kauf der Option zu C GE entsteht zusätzliche Nachfrage, die den Preis C der Option erhöht. Nach einer bestimmten Zeit gilt dann $C = \max(S_T - K, 0)$. Häufig wird aus diesem Grund die Arbitragefreiheit auch als Prämisse vorausgesetzt.

Der Preis der Option ist also zum Zeitpunkt der Ausübung bekannt. Dieser Preis kann aber für andere Zeitpunkte $t < T$ nicht ohne zusätzliche Annahmen bestimmt werden. Ober- und Untergrenzen für den Options-Preis können dagegen durch analoge Arbitrageüberlegungen angegeben werden [Hul96]. Außerdem kann auch eine Beziehung zwischen den Preisen von Put- und Call-Optionen hergeleitet werden. Die *Put-Call-Parität* wird ebenfalls ausführlich in [Hul96] dargestellt.

Binomialmodell

Das Binomialmodell [CR85] ermöglicht die Berechnung von Options-Preisen für beliebige Zeitpunkte innerhalb der Options-Laufzeit. Das ist allerdings nur durch eine

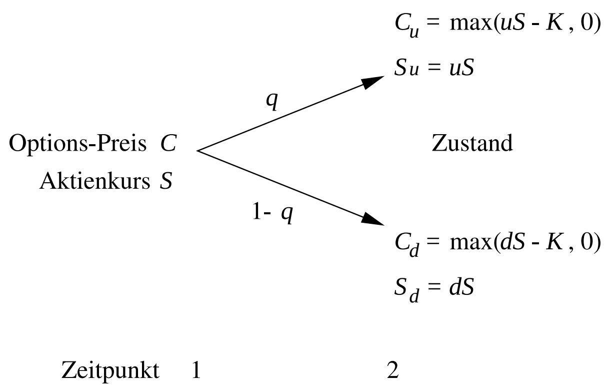


Abbildung 6.4: Binomialmodell

zusätzliche Annahme über die Eintrittswahrscheinlichkeiten für die Kurse des zugrundeliegenden Basiswerts möglich. Dieser Basiswert sei im folgenden eine Aktie. Im Binomialmodell wird angenommen, daß die Aktienkurse binomialverteilt sind. Wenn der Aktienkurs zum Zeitpunkt eins S GE beträgt, dann steigt der Kurs der Aktie zum Zeitpunkt zwei mit der Wahrscheinlichkeit q auf uS GE, oder der Kurs fällt mit der Wahrscheinlichkeit $1-q$ auf dS GE ($d < r < u$). Siehe dazu auch Abbildung 6.4. Daraus folgt, daß der Preis einer Call-Option zum Zeitpunkt zwei entweder mit der Wahrscheinlichkeit q den Wert $C_u = \max(uS - X, 0)$ oder mit der Wahrscheinlichkeit $1-q$ den Wert $C_d = \max(dS - X, 0)$ annimmt, wenn die Option zum Zeitpunkt zwei ausläuft.

Nun wird ein Portefeuille aus Δ Einheiten der Aktie und B GE Kapital betrachtet. Der Preis dieses Portefeuilles beträgt im bei stetiger Verzinsung[5] zum zweiten Zeitpunkt mit der Wahrscheinlichkeit q $uS\Delta + e^r B$ und mit der Wahrscheinlichkeit $1-q$ $dS\Delta + e^r B$ GE. Δ und B werden so bestimmt, daß der Preis des Portefeuilles zum zweiten Zeitpunkt gleich dem Options-Preis ist:

$$\Delta = \frac{C_u - C_d}{(u-d)S}, \quad B = \frac{uC_d - dC_u}{(u-d)e^r}$$

Die Preise des Portefeuilles und der Call-Option müssen aus Arbitragegründen auch zum ersten Zeitpunkt identisch sein:

$$\begin{aligned} C &= S\Delta + B \\ &= \frac{C_u - C_d}{u-d} + \frac{uC_d - dC_u}{(u-d)e^r} \\ &= \frac{\frac{e^r - d}{u-d} C_u + \frac{u - e^r}{u-d} C_d}{e^r} \\ &= \frac{pC_u + (1-p)C_d}{e^r}, \quad p = \frac{e^r - d}{u-d}. \end{aligned} \tag{6.1}$$

[5] Siehe Abschnitt 6.2.3.

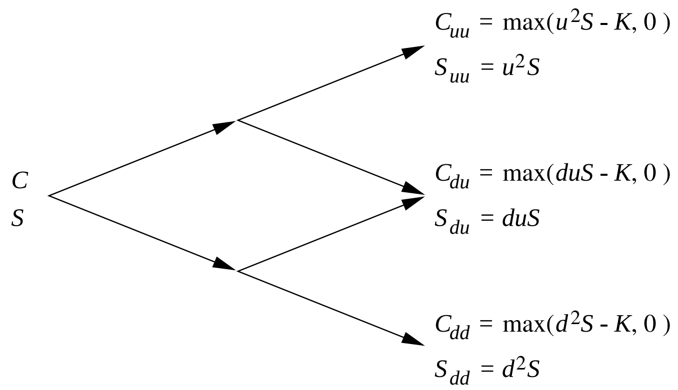


Abbildung 6.5: Binomialmodell mit zwei Perioden

Eine wichtige Eigenschaft der Gleichung (6.1) ist, daß sie nicht von der Wahrscheinlichkeit q und von der Risikoeinstellung des Investors abhängt. Außerdem hat p hier den Charakter einer Wahrscheinlichkeit, es gilt nähmlich $0 \leq p \leq 1$. p ist auch in der Tat der Wert, den die Wahrscheinlichkeit q haben würde, wenn die Anleger *risikoneutral* wären. Bei risikoneutralen Anlegern würde die Aktie eine Rendite in Höhe des risikolosen Zinssatzes r erwirtschaften, d.h. $quS + (1-q)dS = e^r S$ bei $q = \frac{e^r - d}{u-d} = p$.

Der Abbildung 6.4 liegt die Annahme zugrunde, daß der Aktienkurs während der Options-Laufzeit entweder um den Faktor d fällt oder um den Faktor u steigt. Der Aktienkurs kann also am Ende der Options-Laufzeit nur zwei verschiedene Werte annehmen. Diese Annahme ist jedoch für Realität zu grob, so daß es nahe liegt, die Options-Laufzeit in weitere Perioden zu unterteilen.

Abbildung 6.5 zeigt das Binomialmodell mit einer Unterteilung der Options-Laufzeit in zwei Perioden. Der Graph, der zur Bewertung von Optionen nach dem Binomialmodell benutzt wird, wird als *rekombinierender Baum* bezeichnet. Die Rekombinierbarkeit des Baums ist bei der Anwendung eines Modells entscheidend. Läßt das Modell aus theoretischen Gründen keine rekombinierenden Bäume bei der Bewertung zu, so ist damit die praktischen Anwendbarkeit in Frage gestellt, da die Rechenzeit bei einem nicht rekombinierenden Baum exponentiell mit der Tiefe ansteigt.

Eine exponentielle Rechenzeit ist aber in einigen Fällen auch bei rekombinierenden Bäumen nicht vermeidbar. Bei bestimmten Optionen, wie zum Beispiel *Asiatischen Optionen*, ist der Preis pfadabhängig, d.h. der Preis der Option hängt auch von der Reihenfolge ab, in der die Preise des Basiswerts zustande gekommen sind. Da aber die Zahl der Pfade in einem Baum exponentiell mit der Tiefe wächst, ist die Rechenzeit für die Bewertung bei diesen Optionen ebenfalls exponentiell. In diesen Fällen wird auch von *pfadabhängiger Bewertung* gesprochen.

Die Gleichung (6.1) kann dazu rekursiv für beliebig viele Zeitpunkte erweitert werden, indem für C_u und C_d wieder Gleichung (6.1) eingesetzt wird. Es gilt für den Options-

Preis C, bei einer Unterteilung der Options-Laufzeit in n Perioden [CR85]:

$$C = \frac{1}{e^{rn}} \sum_{j=0}^{n} \binom{n}{j} p^j (1-p)^{n-j} \max(0, u^j d^{n-j} S - X)$$

Sei ℓ die kleinste natürliche Zahl, für die $u^\ell d^{n-\ell} - X > 0$ gilt.

$$\begin{aligned} C &= \frac{1}{e^{rn}} \sum_{j=\ell}^{n} \binom{n}{j} p^j (1-p)^{n-j} (u^j d^{n-j} S - X) \\ &= S \sum_{j=\ell}^{n} \binom{n}{j} p^j (1-p)^{n-j} \frac{u^j d^{n-j}}{e^{rn}} - X e^{-rn} \sum_{j=\ell}^{n} \binom{n}{j} p^j (1-p)^{n-j} \\ &= S\Phi(\ell; n, p') - X e^{-rn} \Phi(\ell; n, p) \qquad (6.2) \\ & \text{ mit } p' = \frac{u}{e^r} p \text{ und } \Phi(\ell; n, p') = \sum_{j=\ell}^{n} \binom{n}{j} p^j (1-p)^{n-j}. \end{aligned}$$

Black/Scholes-Gleichung

Das Black/Scholes-Modell [BS73], das 1997 mit dem Nobelpreis für Wirtschaftswissenschaften geehrt wurde, geht im Gegensatz zum Binomialmodell von einem kontinuierlichen Handel aus, d.h. der Handel erfolgt in unendlich kleinen Zeitintervallen. Die Black/Scholes-Gleichung kann dabei durch Unterteilen der Options-Laufzeit in unendlich viele Perioden und anschließender Grenzwertbildung der Gleichung (6.2) für $n \to \infty$ hergeleitet werden [CR85]. Es ergibt sich bei dieser Grenzwertbetrachtung für den Preis einer Call-Option:

$$C = S\mathrm{N}(d) - X e^{-rt} \mathrm{N}(d - \sigma\sqrt{t}), \quad \text{mit } d = \frac{\ln \frac{S}{X} + rt}{\sigma\sqrt{t}} + \frac{1}{2}\sigma\sqrt{t},$$

wobei N die kumulative Standardnormalverteilung bezeichnet.

6.2.3 Zinsderivate

Während Futures und Optionen Derivate darstellen, die sich eigentlich auf beliebig handelbare Basiswerte beziehen können, handelt es sich bei den Derivaten, die im folgenden erläutert werden sollen, um solche, die sich auf die Zinsstrukturkurve beziehen.

Zins

Unter einem *Zins* wird der Preis für die Überlassung von Geld oder Kapital für einen bestimmten Zeitraum verstanden, wobei bei der Berechnung des Zinses verschiedene Methoden und Bezugszeitpunkte unterschieden werden. Bei den folgenden

Ausführungen wird die Anlage als sicher betrachtet, ansonsten müßte bei einem risikoaversem Anleger der Zins um eine Risikoprämie erhöht werden [FH94].

Der Zins kann diskret oder stetig bestimmt werden. Der Unterschied zwischen den beiden Methoden entsteht durch die Anzahl der Zinszahlungen innerhalb eines Jahres. Während bei der diskreten Verzinsung nur endlich viele Zinszahlungen angenommen werden, erfolgen die Zinszahlungen bei der stetigen Verzinsung unendlich oft. Eine Anlage von einer GE steigt bei diskreter m-facher unterjähriger Verzinsung zum Zinssatz i in n Jahren auf

$$(1 + \frac{i}{m})^{mn}$$

GE an. Der stetige Zinssatz ist Grenzwert des unterjährigen Zinssatzes bei unendlich häufiger unterjähriger Verzinsung, also für den Fall $m \to \infty$. Die Höhe der Anlage nach n Jahren wird bei stetiger Verzinsung folgendermaßen berechnet:

$$\lim_{m\to\infty}(1 + \frac{i}{m})^{mn} = e^{in}.$$

Der stetige Zinssatz sollte allerdings nicht mit dem *momentanen Zinssatz* verwechselt werden. Während bei stetigen Verzinsung die Zeiträume, in denen die Zinszahlungen stattfinden als beliebig klein angenommen werden, wird bei einem momentanen Zinssatz von einem einzigen sehr kurzem Zeitintervall ausgegangen.

Weiterhin kann nach dem Bezugszeitpunkt des Zinssatzes unterschieden werden. Ein Zinssatz kann sich zum einen auf den gegenwärtigen, wie auch auf einen in der Zukunft liegenden Zeitpunkt beziehen. Bei einer Anlage, die zum heutigen Zeitpunkt für m Jahre getätigt werden soll, erfolgt die Anlage mit dem m-jährigen *Kassazinssatz*. Dagegen erfolgt die Verzinsung, die erst in k Jahren für m Jahre getätigt werden soll, mit dem *Terminzinssatz*, der in k Jahren für m Jahre gilt. Terminzinssätze können aus Kassazinssätzen unterschiedlicher Laufzeiten bestimmt werden.

Zinsstrukturkurve

Die *Zinsstruktur* (*Term Structure of Interest Rates*) ergibt sich aus allen Zinssätzen zu einem Zeitpunkt für unterschiedliche Laufzeiten. Die graphische Darstellung der Zinsstruktur wird *Zinsstrukturkurve* bzw. *Zinskurve* bezeichnet. Abbildung 6.6 zeigt die Zinsstrukturkurve vom 29.06.1998. Diese Form der Zinsstrukturkurve wird als *normal* bezeichnet, da die Zinssätze für längere Laufzeiten höher sind, als die für kürzere Laufzeiten. Werden jedoch die Zinssätze für längere Laufzeiten kleiner als für kürzere, so wird von einer *inversen* Zinsstrukturkurve gesprochen. Bei einer *flachen* Zinsstrukturkurve sind dagegen alle Zinssätze gleich. Zur Bestimmung der aktuellen Zinskurve aus den Preisen der am Kapitalmarkt gehandelten Finanzinstrumenten siehe [Hul96].

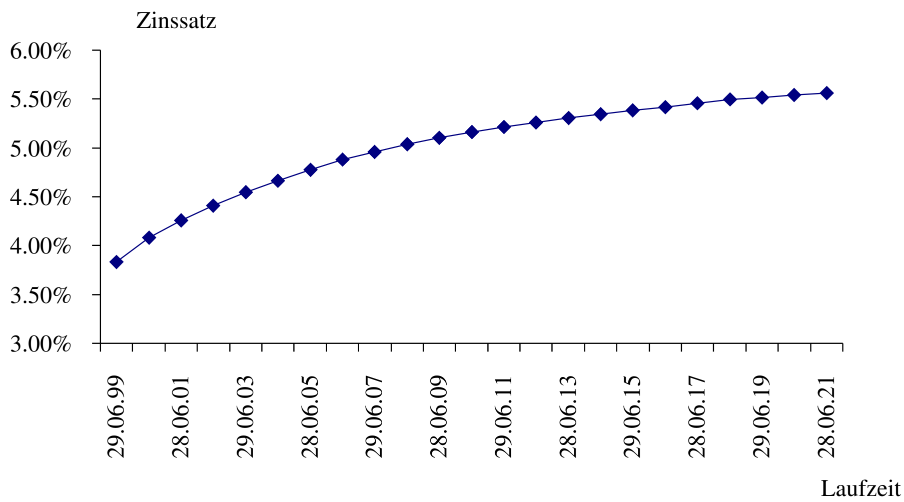


Abbildung 6.6: Zinsstrukturkurve vom 29.06.1998

Cap (Floor)

Ein wichtiges Zinsderivat, das auf dem Over-The-Counter-Markt gehandelt wird, ist der Cap [Hul96]:

> Der Emittent eines *Caps* verpflichtet sich gegenüber dem Käufer zu Zinszahlungen auf das Cap-Nominal, allerdings nur dann, wenn der Referenzzinssatz eine Grenze überschreitet. Der in diesem Fall gezahlte Zinssatz entspricht der Differenz aus dem Referenzzinssatz und der Obergrenze.

Die Obergrenze für den Referenzzinssatz wird bei einem Cap als *Cap-Rate* bezeichnet. Caps bieten sich also insbesondere bei Krediten mit variablen Zinssätzen an, um sicherzustellen, daß der Zinssatz nicht einen bestimmten Wert überschreitet. Aus diesem Grund bieten Kreditinstitute bei Vergabe von Krediten häufig Caps als Sonderleistung in Verbindung mit dem Kredit an. Die Kreditkosten werden dann entsprechend um den Cap-Preis angepaßt.

Ein üblicher Referenzzinssatz ist der 3-Monats-LIBOR. Dieser Zinssatz wird alle drei Monate, bis zum Ende der Cap-Laufzeit, auf Überschreiten der Cap-Rate überprüft. Liegt am Anfang eines Quartals der 3-Monats-LIBOR über der Cap-Rate, dann hat der Emittent des Caps dem Käufer am Ende des Quartals bei einem Cap-Nominal von beispielsweise 10 Mio GE und einer Cap-Rate von 10% einen Zins in Höhe $0.25 \cdot 10 \cdot (\text{LIBOR} - 0.1)$ Mio GE zu zahlen.

Ein Cap kann offensichtlicherweise als ein Portefeuille aus Call-Optionen auf den Referenzzinssatz mit einem Ausübungspreis der Cap-Rate der entsprechenden Zeitperiode aufgefaßt werden. Jede dieser Call-Optionen läuft am Anfang eines Quartals aus, und zahlt dem Käufer den entsprechenden Differenzzinsatz auf das Cap-Nominal aus. Die einzelnen Call-Optionen dieses Portefeuille werden *Caplets* bezeichnet.

Datum	LIBOR Zinssatz	Variable Zahlung	Fixe Zahlung	Summe
01.03.1996	4.20			
01.09.1996	4.80	+2.10	-2.50	-0.40
01.03.1997	5.30	+2.40	-2.50	-0.10
01.09.1997	5.50	+2.65	-2.50	+0.15
01.03.1998	5.60	+2.75	-2.50	+0.25
01.09.1998	5.90	+2.80	-2.50	+0.30
01.03.1999	6.40	+2.95	-2.50	+0.45

Abbildung 6.7: Zahlungen aus einem Swap-Geschäft [Hul96]

Ein Floor ist analog zu einem Cap definiert. Bei einem *Floor* wird zwischen Emittent und Käufer eine Untergrenze vereinbart, bei deren Unterschreitung der Emittent dem Käufer den Differenzzinssatz auf das Floor-Nominal zu zahlen hat. Diese Untergrenze wird auch entsprechend *Floor-Rate* genannt. In diesem Zusammenhang wird bei einer Strategie, die eine Long-Position in dem Cap (Floor) und gleichzeitig eine Short-Position in einem Floor (Cap) vorsieht von einem *Collar* gesprochen.

Swaps

Ein Swap, als ein weiteres wichtiges Zinsderivat, stellt wiederum eine Verpflichtung für beide Parteien dar und ist damit ein symmetrisches Finanzinstrument [Hul96]:

> Ein *Swap* ist eine Vereinbarung zwischen zwei Parteien zum Austausch von fixen und variablen Zinszahlungen an bestimmten Zeitpunkten auf ein gemeinsames Swap-Nominal.

In Abbildung 6.7 ist ein Swap mit einer Laufzeit von drei Jahren zwischen zwei Unternehmen A und B dargestellt. Das Unternehmen A zahlt dem Unternehmen B den 6-Monats-LIBOR, im Gegenzug erhält es einen Zinssatz in Höhe von 5% auf ein Swap-Nominal von 100 Mio GE. Die Zahlungen erfolgen halbjahrlich. Die erste Zahlung erfolgt sechs Monate nach dem Vertragsabschluß, also am 01.09.1996. Da für die erste Zahlung der LIBOR-Zinssatz vom 01.03.1996, also zum Vertragsabschluß, herangezogen wird, existieren für die erste Zahlung keine Ungewißheiten, d.h. die erste Zahlung ist sicher. Unternehmen A zahlt an Unternehmen B einen Zins in Höhe von $0.5 \cdot 0.042 \cdot 100 = 2.1$ Mio GE. Dagegen erhält es am 01.09.1996 wie auch an allen anderen Zahlungterminen von Unternehmen B einen Zinszahlung von $0.5 \cdot 0.05 \cdot 100 = 2.5$ Mio GE. Am 01.03.1997 hat das Unternehmen A dem Unternehmen B eine Zahlung von $0.5 \cdot 0.048 \cdot 100 = 2.4$ Mio GE zu leisten. Diese Zahlung wird wiederum vom LIBOR-Zinssatz bestimmt, der für den Zeitraum vom 01.09.1996 bis 01.03.1997 gültig war. In analogerweise werden alle anderen Zinszahlungen bestimmt.

Ein Unternehmen kann also mit Hilfe eines Swaps variable Zinszahlungen gegen fixe Zinszahlungen austauschen. Swaps können zur Risikominimierung im Rahmen von

Krediten mit variablen Zinssätzen eingesetzt werden. Die variablen Zinszahlungen werden durch fixe ausgetauscht, dadurch existieren keine Unsicherheiten über die Höhe der zukünftigen Zinszahlungen.

Die Zahlungen aus einem Swap können durch ein Portefeuille aus zwei Anleihen dupliziert werden. Im obigen Beispiel können die Zahlungen, die das Unternehmen B aus dem Swap-Geschäft erhält, auch durch den Kauf einer Anleihe mit einem Zinssatz in Höhe des 6-Monats-LIBOR und durch den Verkauf einer Anleihe mit einem festen Zinssatz in Höhe von 5% erreicht werden. Beide Anleihen sollen gesamtfällig getilgt werden und einen Nennwert von 100 Mio GE besitzen. Es gilt für den Preis V des Swaps aus der Sicht des Unternehmens B (vgl. [Hul96]):

$$V = B_{fix} - B_{fl}.$$

B_{fix} ist dabei der Preis der festverzinslichen Anleihe, und B_{fl} der der variabel verzinslichen Anleihe. Der Preis der festverzinslichen Anleihe ergibt sich durch Abzinsen der Zinszahlungen der Anleihe auf den gegenwärtigen Zeitpunkt. Also

$$B_{fix} = \sum_{j=0}^{n-1} Nk\tau \mathrm{DF}_{j+1}.$$

k ist dabei der fixe Zinssatz, der sogenannte *Kupon-Zinssatz*, und N der Nennwert der Anleihe bzw. des Swaps. Außerdem gibt τ die Zeitdauer zwischen den Zinszahlungen an und DF_{j+1} den Diskontfaktor für das Abzinsen vom Zeitpunkt t_{j+1} auf den Zeitpunkt 0. Damit gilt für den Kapitalwert der fixen Zahlungen auch:

$$B_{fix} = \sum_{j=0}^{n-1} Nk\tau e^{-r_{j+1}t_{j+1}}, \tag{6.3}$$

wobei r_{j+1} den Zinssatz für eine risikolose Anlage für den Zeitraum 0 bis t_{j+1} bezeichnet. Der Kapitalwert der fix verzinsten Anleihe kann also ohne zusätzliche Annahmen bezüglich der zukünftigen Entwicklung der Zinsstruktur bestimmt werden. Der Kapitalwert der variabel verzinsten Anleihe bestimmt sich im Gegensatz zu dem der fix verzinsten Anleihe etwas aufwendiger. Es gilt für den Kapitalwert:

$$B_{fl} = \sum_{j=0}^{n-1} Nf_j\tau \mathrm{DF}_{j+1}. \tag{6.4}$$

In dieser Gleichung stellt f_j den Terminzinssatz zum Zeitpunkt t_j für den Zeitraum von t_j bis t_{j+1} dar. Anschließend wird folgende Beziehung zwischen dem Terminzinssatz f_j und dem Diskontfaktor DF_j ausgenutzt:

$$\begin{aligned} \mathrm{DF}_{j+1} &= \frac{\mathrm{DF}_j}{1+\tau f_j} \\ \Leftrightarrow f_j &= \frac{1}{\tau}\left(\frac{\mathrm{DF}_j}{\mathrm{DF}_{j+1}} - 1\right). \end{aligned} \tag{6.5}$$

Durch Einsetzen der Gleichung (6.5) in die Gleichung (6.4) kann folgende Gleichung für den Kapitalwert der variabel verzinsten Anleihe hergeleitet werden:

$$\begin{aligned} B_{fl} &= \sum_{j=0}^{n-1} N \left(\frac{1}{\tau} \left(\frac{\mathrm{DF}_j}{\mathrm{DF}_{j+1}} - 1 \right) \right) \tau \mathrm{DF}_{j+1} \\ &= \sum_{j=0}^{n-1} N(\mathrm{DF}_j - \mathrm{DF}_{j+1}) \\ &= N(\mathrm{DF}_0 - \mathrm{DF}_n) = N(1 - \mathrm{DF}_n) \\ &= N(1 - e^{-r_n t_n}). \end{aligned} \tag{6.6}$$

Somit kann auch der Kapitalwert der variabel verzinsten Anleihe ohne Annahmen für die zukünftige Entwicklung der Zinsstruktur bestimmt werden. Aus den Gleichungen (6.3) und (6.6) folgt für den Preis eines Swaps:

$$\begin{aligned} V &= B_{fix} - B_{fl} \\ &= \sum_{j=0}^{n-1} Nk\tau e^{-r_{j+1}t_{j+1}} - N(1 - e^{-r_n t_n}) \end{aligned}$$

Swaptions

Optionen auf Swaps, oder kurz Swaptions, gehören zu den komplexeren Zinsderivaten [Hul96]:

> Eine *Swaption* gibt dem Käufer das Recht, in einen bestimmten Swap mit einer festgelegten Swap-Rate zu einem Zeitpunkt in der Zukunft eintreten zu können.

Dem Bezugspreis einer Option entspricht bei einer Swaption die Swap-Rate, denn sie bestimmt, ob die Swaption am Ende der Laufzeit ausgeübt wird oder wertlos verfällt. Analog zu Call- und Put-Optionen wird bei Swaptions zwischen Payer- und Receiver-Swaption unterschieden. Bei einer *Payer-Swaption* kann der Käufer der Swaption am Ende der Laufzeit in einen Swap eintreten, in dem er die fixen Zinszahlungen zu leisten hat. Eine *Receiver-Swaption* gibt dem Käufer dagegen das Recht, in einen Swap als Empfänger der fixen Zinszahlungen einzutreten.

Angenommen ein Unternehmen beabsichtigt in sechs Monaten einen variabel verzinsten Kredit aufzunehmen. Die variablen Zinszahlungen sollen dabei aus Gründen der Risikominimierung durch einen Swap gegen fixe Zinszahlungen ausgetauscht werden. Zum gegenwärtigen Zeitpunkt besteht allerdings eine Unsicherheit hinsichtlich der Höhe des Swap-Rates bei der Aufnahme des Kredits. Das Unternehmen kann deshalb heute eine Payer-Swaption abschließen, die es dann am Laufzeitende, also in sechs Monaten, nur ausübt, wenn die Swap-Rate am Markt höher ist, als die Swap-Rate der Swaption. Dadurch sichert sich das Unternehmen eine Swap-Rate, die höchsten gleich der der Swaption ist.

Der Preis einer Swaption kann, im Gegensatz zu einem Swap, nicht ohne zusätzliche Annahmen über die Bewegungen der Zinsstrukturkurve bestimmt werden.

6.3 Bewertung von Zinsderivaten

6.3.1 Probleme bei der Bewertung von Zinsderivaten

Es existiert zu der gegenwärtigen Zeit immer noch kein Modell zur Bewertung von Zinsderivaten, das eine ähnliche Akzeptanz in der Praxis erfahren hat, wie das Black/Scholes-Modell für Aktienoptionen. Der Vorteil des Black/Scholes-Modells ist insbesondere seine einfache Anwendbarkeit: Die Bewertung von europäischen Aktienderivaten kann mit geschlossenen Gleichungen durchgeführt werden. Die Preisbildung bei Zinsderivaten ist jedoch wesentlich schwieriger als die Preisbildung bei Aktienderivaten. Die Gründe sind vielfältig [Hei97] (vgl. [Tsc96]):

1. Die Zinsstruktur ergibt sich durch die Renditen von Anleihen mit unterschiedlichen Laufzeiten. Das Kursverhalten von Anleihen unterscheidet sich jedoch wesentlich von dem der Aktien:

   - Pull-to-Par-Effekt
     Eine Anleihe besitz im Gegensatz zu einer Aktie eine endliche Laufzeit. Der Marktwert der Anleihe am Ende dieser Laufzeit ist bekannt, da die Zins- und Tilgungszahlungen im allgemeinen schon vorher festgelegt sind. Dieser Marktwert ist unabhängig von der zwischenzeitlichen Kursentwicklung der Anleihe.
   - Restlaufzeit und Volatilität
     Die Volatilität, d.h. die Schwankungsbreite, der Anleihekurse ist am Ende der Laufzeit gleich null, da der Marktwert der Anleihe durch den Pull-to-Par-Effekt schon im voraus bekannt ist. Die Volatilität der Anleihenkurse steigt am Anfang der Laufzeit ähnlich wie bei Aktien zunächst an, und sinkt gegen Ende der Laufzeit auf null.
   - Obergrenze des Anleihekurses
     Der Marktwert einer Anleihe ist durch die Zins- und Tilgungszahlungen nach oben begrenzt.

2. Die Anleihen, die die Zinsstruktur bestimmen, unterscheiden sich im Prinzip nur durch die jeweilige Restlaufzeit. Die Kursunterschiede zwischen den Anleihen entstehen somit durch die Laufzeitpräferenzen der Marktteilnehmer, die Anleihen mit bestimmten Laufzeiten vorziehen. Die Kurse von Anleihen mit

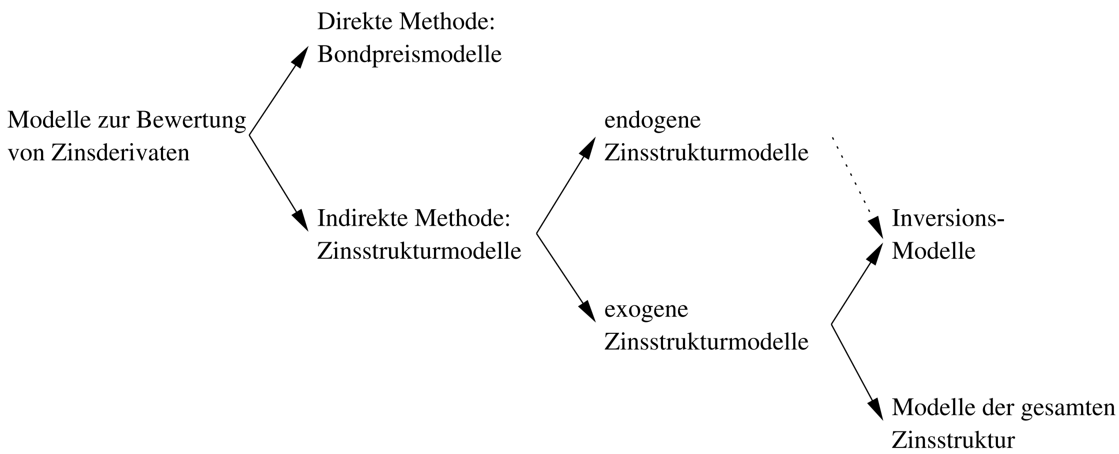


Abbildung 6.8: Klassifikation der Modelle zur Bewertung von Zinsderivaten [Hei97]

ähnlichen Laufzeiten sind also nicht unabhängig von einander, so daß die Modellierung der gesamten Zinsstrukturkurve nicht auf der Basis von unabhängigen Anleihen geschehen kann, sondern Korrelationen berücksichtigt werden müssen.

3. Ein weiteres Problem ist der Umfang der berücksichtigten Informationen. Für die Bewertung einer Option auf einer bestimmten Anleihe ist im Prinzip nur die Modellierung dieser einen Anleihe notwendig. Die Abhängigkeiten unter den Anleihen legt jedoch die Berücksichtigung aller Anleihe nahe. Hinzu kommt, daß zur konsistenten Bewertung verschiedener Zinsderivate die gesamte Zinsstruktur benötigt wird, da die Preise dieser Zinsderivate von unterschiedlichen Zinssätzen abhängen können.

   Ein Zinsstrukturmodell sollte neben der aktuellen Zinsstruktur auch die Schwankungsintensität der einzelnen Zinssätze berücksichtigen. Die Bewegungen der Zinsstrukturkurve könnte mit Hilfe der sogenannten *Volatilitätsstruktur* genauer modelliert werden. Die *Volatilität* ist dabei als die Standardabweichung der relativen Zinsänderungen definiert.

6.3.2 Modelle zur Bewertung von Zinsderivaten

In [Hei97] werden die Ansätze zur Bewertung von Zinsderivaten, wie Caps und Swaptions, gemäß der Abbildung 6.8 zunächst in direkte und indirekte Methoden unterschieden. Bei den *direkten Methoden* wird der Basiswert des Derivats durch einen stochastischen Prozeß beschrieben und aufbauend darauf die Bewertung des Derivats durchgeführt. Da der Basiswert in der Regel eine Anleihe ist, werden die direkten Methoden häufig auch *Bondpreismodelle* bezeichnet. Die *indirekten Methoden* beschreiben dagegen eine oder mehrere Zustandsvariable, wie zum Beispiel bestimmte Zinssätze, durch stochastische Prozesse, aus denen dann die Preise des Basiswerts bestimmt werden können. Die indirekten Methoden werden auch *Zinsstrukturmodelle* genannt, da aus den modellierten Zustandsvariablen häufig auch die gesamte Zinsstrukturkurve ermittelt werden kann.

Zinsstrukturmodelle werden weiterhin in endogene und exogene Zinsstrukturmodelle unterschieden. Die *endogenen Zinsstrukturmodelle* modellieren die Zinsstruktur durch wenige Zustandsvariable. Die endogen ermittelte Zinsstruktur weicht allerdings meistens von der tatsächlichen mehr oder weniger stark ab. Da aber durch kleinste Abweichungen in der Zinsstruktur zu beträchlichen Fehlern bei der Bewertung eines Zinsderivats kommen kann, haben endogene Zinsstrukturmodelle in der Praxis keine Bedeutung. Im Gegensatz dazu können *exogene Zinsstrukturmodelle* beliebige Zinsstrukturkurven abbilden. Dies kann auf zwei verschiedene Arten geschehen. Zum einen können endogene Zinsstrukturmodelle durch weitere Parameter, die die endogene Zinsstruktur beeinflussen, zu *Inversionsmodellen* erweitert werden. Diese Parameter können anschließend im Rahmen einer Kalibrierung so angepaßt werden, daß die endogene Zinsstruktur der tatsächlichen entspricht. Eine andere Möglichkeit besteht im Modellieren aller Zinssätze der Zinsstruktur, so daß die tatsächliche Zinsstruktur als Startwert benutzt werden kann.

6.3.3 Bondpreismodelle

Die wichtigsten Bondpreismodelle stellen ohne Zweifel das Black/Scholes-Modell und seine Erweiterung in Form des Black-Modells [Bla76, Hul96] dar.

Das Black/Scholes-Modell, das eigentlich zur Bewertung von Aktienderivaten entwickelt wurde, kann unter bestimmten Voraussetzungen und mit geeigneter Parametrisierung auch bei Zinsderivaten eingesetzt werden. Dabei müssen allerdings einige theoretische Probleme berücksichtigt werden. Dazu gehört die Annahmen der konstanten Volatilität des Basiswerts. Diese Annahme stellt zwar bei Aktien keine größeren Probleme dar, ist aber, wie schon oben am Pull-to-Par-Effekt erläutert, bei Anleihen definitiv nicht erfüllbar. Der Fehler bei der Bewertung von beispielsweise einer Option auf einer Anleihe hängt insbesondere von den Laufzeiten der Option und der Anleihe ab. Dieser Fehler ist, solange ein größerer Zeitraum zwischen den beiden Fälligkeiten liegt, vernachlässigbar. Haben jedoch die Option und die Anleihe eine vergleichbare Laufzeit, so ist dieser Bewertungsfehler entsprechend zu berücksichtigen.

Im Investmentbanking wird häufig eine Erweiterung des Black/Scholes-Modells eingesetzt, die unterstellt, daß sich das zu bewertende Zinsderivat auf den Terminpreis des Basiswerts und nicht auf den Kassa-, also den aktuellen Preis, bezieht. Der Terminpreis des Basiswerts ist dabei der Preis, den der Basiswert zum Zeitpunkt der Fälligkeit des Derivats besitzt. Bei einer Option mit drei Jahren Laufzeit auf einer Anleihe würde das bedeuten, daß sich die Option auf den Preis der Anleihe bei Lieferung in drei Jahren beziehen würde. Der eigentliche Basiswert ist also ein Forward-Kontrakt auf den Basiswert. Ein wichtiger Unterschied zum Black/

SCHOLES-Modell ist damit der unterschiedliche Bezugszeitpunkt der Volatilität des Basiswerts. Das BLACK-Modell benötigt nur die Volatilität des Basiswerts am Ende der Laufzeit des Derivats.

Andere Bondpreismodelle werden in [Hei97] beschrieben.

Bondpreismodelle besitzen den Nachteil, daß sie die Abhängigkeiten der am Markt gehandelten Anleihen, und damit die Abhängigkeiten innerhalb der Zinsstruktur nicht berücksichtigen. Daraus resultiert, daß für jedes zu bewertende Zinsderivat ein eigenes Modell erforderlich wird.

6.3.4 Zinsstrukturmodelle

Theorien zum Verlauf der Zinsstrukturkurve

Es wurden schon frühzeitig die ersten Theorien zum Verlauf der Zinsstruktur formuliert, die den Zusammenhang zwischen kurz- und langfristigen Zinssätzen erklären sollten. Zu den wichtigsten Theorien gehören die Erwartungs- und die Liquiditätspreferenztheorie.

- Erwartungstheorie
  Nach der Erwartungstheorie [FH94] (vgl. [Hav93, Tsc96]), zu der schon im Jahre 1896 die erste Veröffentlichung erschienen ist, wird die Zinsstruktur, und damit die Form der Zinsstrukturkurve, durch die Erwartungen der Marktteilnehmer über die Entwicklung der kurzfristigen Zinssätze bestimmt. Sei $r_{t,T}$ der Zinssatz für eine risikolose Anlage zum Zeitpunkt t für die Dauer von T Jahren. Es bestehen zwei Möglichkeiten Kapital für die Dauer von T Jahren anzulegen. Zum einen ist diese die einmalige Anlage zum Zinssatz von $r_{0,T}$ und zum anderen die revolvierende Anlage zu den Zinssätzen $r_{0,1}$, $\tilde{r}_{1,1}$, $\ldots$, $\tilde{r}_{T-1,1}$. Bei der revolvierenden Anlage sind alle Zinssätze, bis auf ersten, unsicher, d.h. im Zeitpunkt 0 kann nicht mit Sicherheit vorhergesagt werden, zu welchem Zinssatz $r_{t,1}$ im Zeitpunkt t wiederangelegt werden kann. Bei risikoneutralen Anlegern sind beide Alternativen gleichwertig, da sie sich nur hinsichtlich des Risikos unterscheiden. In einer risikoneutral Welt besitzen zwei Anlagen mit gleicher Laufzeit dieselbe Rendite, da alle Anleger sich nur nach der Rendite richten und in die Anlage mit der größeren Rendite investieren. Es gilt also:

  $$(1 + r_{0,T})^T = (1 + r_{0,1})(1 + \mathrm{E}(\tilde{r}_{1,1}))(1 + \mathrm{E}(\tilde{r}_{2,1}))\ldots(1 + \mathrm{E}(\tilde{r}_{T-1,1})).$$

  Eine steigende Zinsstrukturkurve $r_{0,1} < r_{0,2} < \ldots < r_{0,T}$ ist nach der Erwartungstheorie damit immer dann gegeben, wenn die Marktteilnehmer steigende kurzfristige Zinssätze erwarten. Eine fallende Zinsstrukturkurve entsprechend in dem anderen Fall.

- **Liquiditätspräferenztheorie**
  In der Liquiditätspräferenztheorie [FH94] (vgl. [Hav93, Tsc96]) wird im Gegensatz zur Erwartungstheorie auch das Risiko einer Anlage berücksichtigt. Die Marktteilnehmer schätzen die langfristige Anlage risikoreicher als die kurzfristige Anlage ein, da sie weniger flexibel gegenüber Zinsveränderungen ist. Somit werden sie sich nur bei einer entsprechenden Risikoprämie, die mit der Länge der Laufzeit zunimmt, für die langfristige Anlage entscheiden. Die langfristigen Zinssätze sind damit um die Risikoprämien höher als die kurzfristigen, so daß bei der Liquiditätspräferenztheorie eine steigende Zinsstruktur, und damit eine normale Zinsstrukturkurve, postuliert wird. Diese Theorie erklärt aber keine inversen Zinsstrukturkurven, die im Frühjahr 1992 auch auf dem deutschen Kapitalmarkt gegeben waren.

Kritik zu diesen Theorien und andere Ansätze finden sich in [Hav93, Tsc96].

Endogene Zinsstrukturmodelle

Die beiden oben erläuterten Theorien versuchen den Verlauf der Zinsstrukturkurve zu erklären, sie lassen aber die zeitlichen Bewegungen der Kurve unberücksichtigt. Die Bewegungen der Zinsstrukturkurve sind allerdings bei der Bewertung von Zinsderivaten entscheiden. Bei endogenen Zinsstrukturmodellen werden die Bewegungen der Zinsstrukturkurve durch stochastische Prozesse für den Momentanzins oder für andere Zustandsvariablen modelliert. Der stochastische Prozeß kann dabei auf zwei verschiedene Arten entwickelt werden. Während *Arbitragemodelle* einen Prozeß für den Momentanzins explizit vorgeben, wird bei *allgemeinen Gleichgewichtsmodellen* der Zinsprozeß aus Annahmen über den Kapitalmarkt hergeleitet [Hei97, Hul96]. Ausgehend von dem Prozeß des Momentanzinses oder anderer Zustandsvariablen werden dann Gleichungen für die Preise von Nullkupon-Anleihen abgeleitet. Diese Gleichungen bestimmen die Zinsstrukturkurven, die im Rahmen des entsprechenden Modells abgebildet werden können. Für die genauen Herleitungen der durch das entsprechende Modell implizierten Preisgleichungen von Nullkupon-Anleihen sei auf die Originalliteratur verwiesen.

Die bekanntesten Beispiele für endogene Zinsstrukturmodelle sind die Modelle nach Rendleman/Bartter [RB80], Vasicek [Vas77] und Cox/Ingersoll/Ross [CIR85]. In allen drei Modellen wird der Momentanzins durch einen stochastischen Prozeß in der Form

$$dr = \mu(r,t)dt + \sigma r^{\alpha} dW(t)$$

beschrieben.

Das Rendleman/Bartter-Modell unterstellt, daß der Momentanzins der geometrischen Brownschen Bewegung folgt, also sich verhält wie eine Aktie im Black/Scholes-Modell. Es gilt für die Driftrate $\mu(r,t) = \mu r$ und für die Varianzrate

$\sigma^2 r^{2\alpha} = \sigma^2 r^2$. Die Annahme der Brownschen Bewegung bei Zinssätzen hat jedoch ihre Probleme. Ein wichtiger Unterschied zwischen Aktien und Zinssätzen ist, daß mit großer Wahrscheinlichkeit hohe Zinssätze gegen einen langfristigen Mittelwert fallen und niedrige Zinssätze gegen diesen Wert steigen. Diese Eigenschaft bei Zinssätzen wird *Mean Reversion* bezeichnet. Das RENDLEMAN/BARTTER berücksichtigt jedoch nicht die wichtige Eigenschaft der Mean Reversion bei Zinssätzen.

Das VASICEK-Modell ist dagegen so konstruiert, daß Zinssätze mit einer Rate von a gegen einen langfristigen Mittelwert b streben. Die Driftrate ist zu diesem Zweck mit $\mu(r) = a(b - r)$ spezifiziert. Die Varianzrate ist mit $\sigma^2 r^{2\alpha} = \sigma^2$ festgelegt.

Beide Modelle haben den Nachteil, daß der Momentanzins negativ werden kann. Das COX/INGERSOLL/ROSS-Modell besitzt diesen Nachteil nicht. Die Varianzrate ist dazu so festgelegt, daß sie proportional zum Momentanzins ist. Damit steigt bei steigendem Momentanzins auch die Varianz des Momentanzinses an. Der Prozeß für den Momentanzins hat dabei denselben Drift wie der Prozeß für den Momentanzins im VASICEK-Modell. Im Gegensatz zu den beiden obigen Modellen, ist das COX/INGERSOLL/ROSS-Modell ein allgemeines Gleichgewichtsmodell.

Alle drei beschriebenen Modelle sind sogenannte *Einfaktormodelle*, da sie von der Annahme ausgehen, daß die Zinsstruktur von nur einer zufälligen Variable bestimmt wird. Einfaktormodelle der Zinsstruktur können jedoch nur beschränkte Formen der Zinsstrukturkurve abbilden, so daß bei anderen Modellen weitere zufällige Variablen angenommen wurden. Diese Modelle werden nach der Zahl der zufälligen Variablen *Zwei-* oder *Mehrfaktormodelle* bezeichnet. Endogene Zweifaktormodelle der Zinsstruktur sind beispielsweise die Modelle nach BRENNAN/SCHWARTZ [BS82] und LONGSTAFF/SCHWARTZ [LS92]. Die zwei zufälligen Variablen im BRENNAN/SCHWARTZ-Modell sind zwei Zinssätze, deren stochastische Prozesse sich gegenseitig beinflussen können. Während dieses Modell ein Arbitragemodell ist, ist das Modell nach LONGSTAFF/SCHWARTZ ein allgemeines Gleichgewichtsmodell. Die zufälligen Variablen dieses Modells sind zum einen der Momentanzins und zum anderen die Momentanvarianz.

Das Problem der endogenen Zinsstrukturmodelle ist, daß die tatsächliche Zinsstrukturkurve unter Umständen nicht durch das Modell abbildbar ist, da die tatsächliche Zinsstruktur keine Gleichgewichts-Situation des Modells darstellt. Die Bewertung von Zinsderivaten erfordert jedoch eine möglichst exakte Abbildung der tatsächlichen Zinsstruktur, da sonst die berechneten Preise von denen am Markt mehr oder minder stark abweichen können. Wenn das Modell die Realität nicht in einer bestimmten Qualität abbilden kann, können die Implikationen aus dem Modell auch nicht auf die Realität bezogen werden.

Inversionsmodelle

Inversionsmodelle gleichen in vieler Hinsicht den endogenen Zinsstrukturmodellen. Die Zinsstruktur wird auch bei diesen Modellen ausgehend von einem stochastischen Prozeß für den Momentanzins oder für andere Variablen hergeleitet. Während bei den endogenen Zinsstrukturmodellen die tatsächliche Zinsstruktur keine Berücksichtigung findet, wird bei Inversionsmodellen der stochastische Prozeß an die tatsächliche Zinsstruktur angepaßt, um endogen realistischere Zinsstrukturen zu erhalten. Der stochastische Prozeß wird zu diesem Ziel um zeitabhängige Parameter erweitert. Das Anpassen des Prozesses geschieht durch die Kalibrierung dieser zeitabhängigen Parameter, d.h. es werden diejenigen Werte für die zeitabhängigen Parameter bestimmt, die den geringsten Fehler bei der Bewertung von Zinsderivaten verursachen.

Hull/White [HW90, Hul96] schlagen folgende Erweiterung des Prozesses für den Momentanzins in den Modellen von Vasicek und Cox/Ingersoll/Ross vor:

$$dr = (\theta(t) + \kappa(t)(r_0 - r))dt + \sigma(t)r^{\alpha}dW(t).$$

Dieses allgemeine Modell läßt nicht nur eine Kalibrierung an die aktuelle Zinsstruktur zu, sondern auch an die aktuelle Volatilitätstruktur, da auch die Varianzrate $\sigma(t)$ zeitabhängig ist. Dieser Prozeß stellt für $\alpha = 0$ die Erweiterung des Vasicek-Modells und für $\alpha = 0.5$ die Erweiterung des Cox/Ingersoll/Ross-Modells dar. Andere Inversionsmodelle sind in [Hei97] dargestellt.

Probleme bei der praktischen Realisierung von Inversionsmodellen ergeben sich insbesondere in den Fällen, wo keine geschlossene Preisgleichungen für Anleihen oder Zinsderivate hergeleitet werden können. In solchen Fällen muß die Bewertung von Zinsderivaten numerisch, wie zum Beispiel durch rekombinierende Bäume[6], erfolgen. Die numerische Bewertung von Zinsderivaten durch rekombinierende Bäume erfordert jedoch einen quadratischen bis kubischen Rechenaufwand in der Tiefe des Baums, so daß die Kalibrierung, die durch iterative Bewertung von mehreren Zinsderivaten geschieht, entsprechend lange dauert. Der Aufwand bei der Kalibrierung eines Modells ist also für seine Praxistauglichkeit entscheidend.

Modelle der gesamten Zinsstruktur

Bei den Modellen der gesamten Zinsstruktur stellt sich das Problem der Kalibrierung erst gar nicht, da die aktuelle Zinsstruktur als Startwert in die Zinsstrukturentwicklung eingeht. Damit wird auch die gesamte Information der aktuellen Zinsstruktur berücksichtigt.

Das erste Modell der gesamten Zinsstruktur wurde von Ho/Lee [HL86] veröffentlicht. Das Modell nach Ho/Lee beschreibt die Entwicklung der Zinsstruktur in Form eines

[6]Siehe Abschnitt 6.3.5.

Binomialbaums, in dem die Preise verschiedener Nullkupon-Anleihen modelliert sind. Die Bewertung von Zinsderivaten erfolgt dabei relativ zu den Preisen der modellierten Nullkupon-Anleihen. Dazu werden Beziehungen zwischen dem zu bewertendem Zinsderivat und Nullkupon-Anleihen ausgenutzt, wie zum Beispiel, daß sich ein Cap als ein Portefeille aus Anleihen darstellen läßt. Das Ho/Lee-Modell hat jedoch einige Nachteile, die dessen praktische Anwendbarkeit in Frage stellen. Dazu zählt zum einen, daß die Form der Zinsstrukturkurven, die nach dem Modell in der Zukunft möglich sind, sehr eingeschränkt ist. Die zukünftigen Zinsstrukturkurven sind parallele Verschiebungen der Ausgangskurve. Außerdem ist auch kein Kippen der Zinsstrukturkurve möglich, d.h. es kann keine inverse Zinsstrukturkurve aus einer normalen entstehen. Zum anderen sind im Ho/Lee-Modell negative Zinssätze möglich.

Eine allgemeine Theorie zur Modellierung der gesamten Zinsstruktur wurde von Heath/Jarrow/Morton [HJM92] entwickelt. Sie haben, ausgehend von einem stochastischen Prozeß für den Terminzins, Beziehungen zwischen dem Drift und der Varianz des Prozesses hergeleitet, die die Arbitragefreiheit der durch den Prozeß implizierten Zinsstrukturen sicherstellen. Das Heath/Jarrow/Morton-Modell umfaßt die meisten anderen Zinsstrukturmodelle, die veröffentlich wurden. Die Allgemeinheit dieses Modells stellt aber auch ein Problem bei der Bewertung von Zinsderivaten dar. Die Bewertung von Zinsderivaten muß, da keine geschlossenen Preisgleichungen angegeben werden können, numerisch durch Bäume erfolgen. Diese numerische Approximation kann jedoch nur mit nicht-rekombinierenden Bäumen durchgeführt werden, d.h. die Approximation erfordert eine exponentielle Laufzeit in der Tiefe des Baums. Aus diesem Grund wurden Modelle entwickelt, die zwar den Bedingungen aus Heath/Jarrow/Morton-Modell genügen, die aber durch eingeschränkte stochastische Prozesse für den Terminzins die Rekombinierbarkeit des Baums zur Approximation sicherstellen. Eines dieser Modelle ist das Ritchken/Sankarasubramanian-Modell, das im nächsten Unterkapitel ausführlicher erläutert wird.

6.3.5 Zinsstrukturmodell nach Ritchken/Sankarasubramanian

Theoretische Grundlagen

Die numerische Bewertung von Zinsderivaten erfordert im Heath/Jarrow/Morton-Modell im allgemeinen einen nicht-rekombinierenden Baum [Jar96, HJM+92]. Da die Zahl der Knoten in dem nicht-rekombinierenden Baum exponentiell mit der Tiefe des Baums steigt, ist die Approximation der Preise von Zinsderivaten mit einem exponentiellen Aufwand, sowohl in Bezug auf die Laufzeit als auch auf den Speicherplatzbedarf, verbunden. Der Größe des Baums ist somit eine obere Grenze gesetzt, die zwar für europäische Derivate ausreichend sein mag [HJM+92], die aber bei amerikanischen Derivaten einen nicht zu vernachlässigenden Approximationsfehler verursacht.

RITCHKEN/SANKARASUBRAMANIAN stellen in [RS95a, LRS95a, RS95b] eine Klasse von stochastischen Prozessen vor, die eine pfadunabhängige Bewertung von Zinsderivaten gestatten. Sei folgender Prozeß für den Terminzins $f(t,T)$ gegeben, $f(t,T)$ ist dabei der Wert des momentanen Terminzinssatzes zum Zeitpunkt t für den Zeitpunkt T:

$$df(t,T) = \mu_f(t,T)dt + \sigma_f(t,T)dW(t).$$

Die Zinsstruktur kann ausgehend von dem stochastischen Prozeß für den Terminzinssatz bestimmt werden, da aus dem Terminzinssatz die Preise von Nullkupon-Anleihen berechnet werden können. Es gilt für den Preis einer Nullkupon-Anleihe, die im Zeitpunkt T fällig wird, im Zeitpunkt t [BR96]:

$$P(t,T) = \exp\left(-\int_t^T f(t,s)ds\right). \tag{6.7}$$

Die Volatilität $\sigma_f(t,T)$ ist bei der Modellierung der Zinsstruktur entscheidend, da sie nach HEATH/JARROW/MORTON [HJM92] aus Arbitragegründen die Driftrate $\mu_f(t,T)$ bestimmt:

$$\mu_f(t,T) = \sigma_f(t,T)\int_t^T \sigma_f(t,s)ds.$$

In dieser allgemeinen Form können Zinsderivate nur mit nicht-rekombinierenden Bäumen bewertet werden. RITCHKEN/SANKARASUBRAMANIAN bestimmen in [RS95a] eine Klasse von stochastischen Prozessen für den momentanen Terminzinssatz, die jedoch die effiziente Bewertung mit rekombinierenden Bäumen ermöglicht. Sie leiten folgende Bedingung für die Varianzrate dieser Prozesse her:

$$\sigma_f(t,T) = \sigma_f(t,t)k(t,T), \quad \text{mit } k(t,T) = \exp\left(-\int_t^T \kappa(x)dx\right). \tag{6.8}$$

$\sigma_f(t,t)$ ist dabei die Volatilität des momentanen Kassazinssatzes im Zeitpunkt t. Es soll daran erinnert werden, daß der Terminzinssatz $f(t,t)$ zum Zeitpunkt t für den Zeitpunkt t per Definition dem Kassazinssatz zum Zeitpunkt t entspricht. Die Volatilität des Terminzinssatzes $f(t,T)$ ist also gleich der Volatilität des Kassazinssatzes zum Zeitpunkt t, der seinerseits von der gesamten Zinsstruktur abhängen kann und nicht weiter eingeschränkt ist, multipliziert mit einer exogen vorgegebener Funktion $\kappa(x)$. Diese Funktion dient insbesondere bei der Kalibrierung der Volatilitätsstruktur [LRS95a].

Es kann auf der Basis der Gleichungen (6.7) und (6.8) eine Preisgleichung für Nullkupon-Anleihen hergeleitet werden, die die Zinsstruktur eindeutig festgelegt [RS95a]:

$$P(t,T) = \frac{P(0,T)}{P(0,t)}\exp\left(-\beta(t,T)(r(t)-f(0,t)) - 0.5\beta^2(t,T)\phi(t)\right).$$

Dabei gilt für β und ϕ:

$$\beta(t,T) = \int_t^T k(t,u)du, \quad \phi(t) = \int_0^t \sigma_f^2(u,t)du = \int_0^t \sigma_f^2(u,u)k^2(u,t)du.$$

Der Preis einer Nullkupon-Anleihe, die zum Zeitpunkt T fällig wird, kann also mit Informationen, die zum Zeitpunkt t verfügbar sind, bestimmt werden. Benötigt werden einerseits die Preise von Nullkupon-Anleihen zum Zeitpunkt 0. Diese Preise können aus der aktuellen Zinsstruktur, die als Startwert in die Entwicklung eingeht, entnommen werden. Andererseits werden der momentane Kassazinssatz $r(t)$ im Zeitpunkt t und der Wert einer zweiten Variable $\phi(t)$ benötigt. Diese Variable stellt neben dem Terminzinssatz die zweite zufällige Variable in dem Modell dar.

Die Einschränkung der Volatilität des momentanen Terminzinssatzes impliziert folgende stochastische Prozesse für die Entwicklung der beiden Zustandsvariablen $r(t)$ und $\phi(t)$ [RS95a]:

$$dr = \mu(r, \phi, t)dt + \sigma_f(t,t)dW(t), \quad d\phi(t) = (\sigma_f^2(t,t) - 2\kappa(t)\phi(t))dt,$$

mit

$$\mu(r, \phi, t) = \kappa(t)(f(0,t) - r(t)) + \phi(t) + \frac{d}{dt}f(0,t).$$

Generierung des rekombinierenden Baums

Im folgenden wird die numerische Bewertung von Zinsderivaten im RITCHKEN/SANKARASUBRAMANIAN-Modell beschrieben. Die Ausführungen beziehen sich dabei auf [LRS95a] (vgl. [LRS95b]), wobei eine Spezifikation der Volatilität in der Form

$$\sigma_f(t,t) = \sigma r(t) \tag{6.9}$$

vorausgesetz wird.

Der stochastische Prozeß des momentanen Terminzinssatzes wird, um die Rekombinierbarkeit des Baums sicherzustellen, in eine Form mit konstanter Varianz transformiert. Es ergibt sich dabei eine neue Zustandsvariable Y, die sich bei einer Spezifikation der Volatilität nach Gleichung (6.9) durch

$$Y(t) = \frac{\ln r(t)}{\sigma}. \tag{6.10}$$

ausdrücken läßt. Die Entwicklung dieser Zustandsvariablen wird durch folgenden stochastischen Prozeß beschrieben:

$$dY(t) = m(Y, \phi, t)dt + dW(t).$$

Für den Drift dieses Prozesses gilt:

$$m(Y, \phi, t) = \frac{1}{\sigma}\left(\left(\kappa(f(0,t) - e^{\sigma Y(t)}) + \phi(t) + \frac{d}{dt}f(0,t)\right)e^{-\sigma Y(t)} - \frac{1}{2}\sigma^2\right).$$

Angenommen die Laufzeit T des zu bewertenden Derivats wird in n äquidistante Perioden mit der Länge $\Delta t = \frac{T}{n}$ unterteilt. Sei (Y_0, ϕ_0) der initiale Wert der beiden

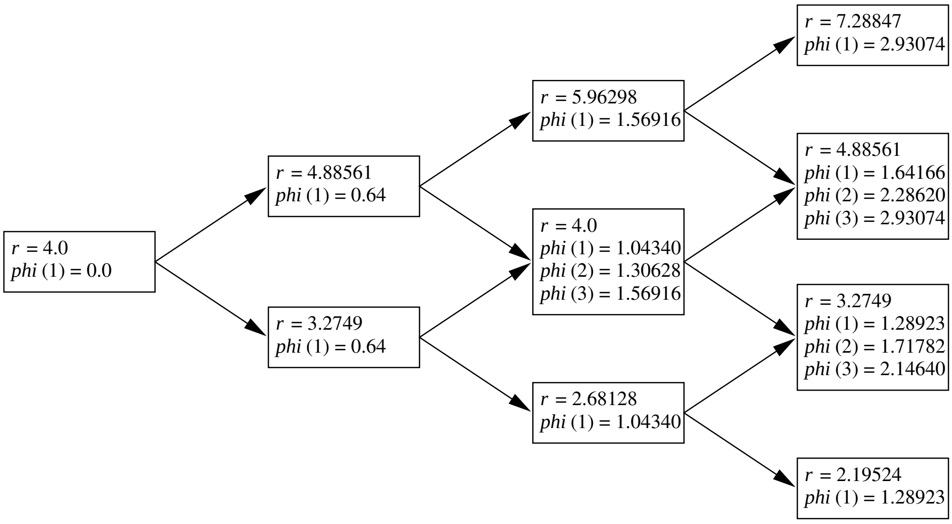


Abbildung 6.9: Rekombinierender Baum nach RITCHKEN/SANKARASUBRAMANIAN [LRS95a]

Zustandsvariablen zum Zeitpunkt 0, d.h. im Wurzelknoten des Baums. Für diese beiden Startwerte gilt zum einen $Y_0 = \frac{\ln r(0)}{\sigma}$ und zum anderen $\phi_0 = 0$. Angenommen Y_i sei der Wert der Zustandsvariable Y nach i Perioden. Der Wert Y_{i+1} kann nach einer Periode zwei verschiedene Werte annehmen:

$$Y_{i+1}^{+} = Y_i + \sqrt{\Delta t}, \quad Y_{i+1}^{-} = Y_i - \sqrt{\Delta t}.$$

Damit gilt in Verbindung mit der Gleichung (6.10) für den momentanen Kassazinssatz:

$$r_{i+1}^{+} = e^{\sigma Y_{i+1}^{+}} = e^{\sigma(Y_i + \sqrt{\Delta t})}, \quad r_{i+1}^{-} = e^{\sigma Y_{i+1}^{-}} = e^{\sigma(Y_i - \sqrt{\Delta t})}.$$

Die zweite Zustandsvariable ϕ ist im Gegensatz zu der Zustandsvariablen Y pfadabhängig, d.h. ihr Wert ist für jeden Pfad in dem Baum verschieden. Genaugenommen müßte in einem Knoten zu jedem Pfad, der zu diesem Knoten führt, ein ϕ-Wert abgespeichert werden. Da aber die Pfadabhängigkeit verhindert werden soll, wird eine maximale Zahl m der ϕ-Werte in einem Knoten vorgegeben. Führen mehrere Pfade zu einem Knoten, so wird das Intervall zwischen dem kleinsten und dem größten ϕ-Wert in $m-1$ Intervalle unterteilt, so daß die Zahl der ϕ-Werte gleich m beträgt. Die maximale Zahl der ϕ-Werte ist in der Abbildung 6.9 aus drei beschränkt.

Sei ϕ_i der Wert der Zustandsvariable ϕ nach i Perioden. Die beiden Nachfolgewerte von ϕ_i bestimmen sich bei einer Spezifikation der Volatilität nach Gleichung (6.9) folgendermaßen:

$$\phi_{i+1}^{+} = \phi_{i+1}^{-} = \phi_i + (\sigma^2 r(i\Delta t)^2 - 2\kappa(i\Delta t)\phi_i)\Delta t. \tag{6.11}$$

Bei der Generierung des Baums wird mit Hilfe der Gleichung (6.11) der Nachfolger eines ϕ-Werts bestimmt. Falls zu einem der beiden nachfolgenden Knoten mehrere

Pfade führen und in diesem schon ϕ-Werte abgespeichert sind, so wird zwischen dem minimalen und den maximalen ϕ-Wert, mit Berücksichtigung des neu berechneten ϕs, erneut interpoliert.

Im folgenden wird zum Ziel der einfacheren Verständlichkeit die Berechnung der Wahrscheinlichkeiten nach [LRS95b] beschrieben, obwohl diese Methode negative Wahrscheinlichkeiten nicht ausschließt. Negative Wahrscheinlichkeiten und, damit verbunden, Wahrscheinlichkeiten größer eins werden bei der Berechnung nach [LRS95a] vermieden.

Die Wahrscheinlichkeit p für eine Aufwärtsbewegung in dem Baum wird so bestimmt, daß der Erwartungswert und die Varianz des Kassazinssatzes dem Drift und der Varianz des approximierten Prozesses bei beliebig kleiner Partitionierung der Laufzeit des Derivats gleicht. Diese Bedingung wird durch folgende Gleichung erfüllt:

$$p = \frac{m(Y, \phi, t)\Delta t + Y_i - Y_{i+1}^-}{Y_{i+1}^+ - Y_{i+1}^-}.$$

Da die Wahrscheinlichkeit p auch von ϕ abhängt, ist die Zahl der Wahrscheinlichkeiten für eine Aufwärtsbewegung in einem Knoten gleich der Zahl der ϕ-Werte. Die Wahrscheinlichkeit für eine Abwärtsbewegung in dem Baum ist entsprechend $1 - p$.

Bewertung eines Zinsderivats

Die zukünftige Entwicklung der Zinsstruktur wird durch einen Zustandsbaum beschrieben, in dem jeder Knoten eine mögliche Form der Zinsstrukturkurve darstellt. Ein Zinsderivat, wie beispielsweise ein Cap, ist nur in bestimmten Zuständen bzw. bei bestimmten Formen der Zinsstrukturkurve mit Zahlungen verbunden. Jedem Knoten entspricht damit auch eine Zahlung, die an den Emittenten oder vom Emittenten des Derivats zu leisten ist. Der aktuelle Wert des Derivats bestimmt sich als abdiskontierter Erwartungswert, da jeder Zustand mit einer bestimmten Wahrscheinlichkeit und nach einer bestimmten Zeit erreicht werden kann. Abbildung 6.10 stellt einen Zustandsbaum, der die Zahlungen einer europäischen Call-Option angibt. Der Basiswert dieser Option ist eine Nullkupon-Anleihe mit einer Laufzeit von fünf Jahren und einem Nominalwert von 100000 GE. Die Option hat eine Laufzeit von drei Jahren und einen Ausübungspreis von 81873.07 GE.

Als erster Schritt werden bei der Bewertung eines Derivats die Zahlungen in den möglichen Zuständen in dem Zustandsbaum eingetragen. Danach werden diese Zahlungen rekursiv auf den Wurzelknoten des Baums, und damit auf den gegewärtigen Zeitpunkt abdiskontiert.

Sei C_i der Wert des zu bewertenden Derivats nach i Perioden, in einem Zustand mit einem ϕ-Wert von ϕ_i und einem Y-Wert von Y_i. Bei der Berechnung von C_i wird zuerst mit Hilfe der Gleichung (6.11) der ϕ-Wert in der Periode ϕ_{i+1} bestimmt. Anschließend werden aus den beiden nachfolgenden Knoten die Werte $C^+_{i+1,\phi_{i+1}}$ und $C^-_{i+1,\phi_{i+1}}$ des

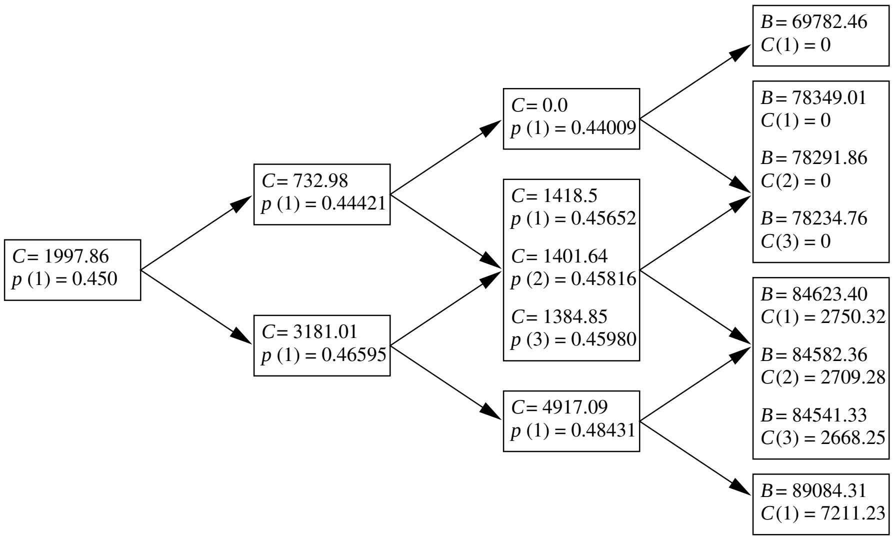


Abbildung 6.10: Bewertung einer europäischen Call-Option [LRS95a]

Derivats, die ϕ_{i+1} entsprechen, entnommen und, mit wiederum der zu ϕ_i entsprechenden Wahrscheinlichkeit, der Erwartungswert gebildet. Dieser Erwartungswert wird danach mit dem Kassazinssatz dieses Zustands abdiskontiert. Also:

$$C_i = (p_{\phi_i} C_{i+1}^+ + (1 - p_{\phi_i}) C_{i+1}^-) e^{-r_i i \Delta t}.$$

Rekursiv kann der Wert des Derivats zum Zeitpunkt 0 und damit im gegenwärtigen Zeitpunkt bestimmt werden.

Es muß allerdings beachtet werden, daß ϕ_{i+1}, durch die beschränkte Zahl der ϕ-Werte in einem Knoten, nicht in den nachfolgenden Knoten vorhanden sein kann. In diesem Fall ist die Zahl der Pfade zu den nachfolgenden Knoten größer als die erlaubte Zahl der ϕ-Werte. Um jedoch trotzdem den zu ϕ_{i+1} entsprechenden Preis des Derivats zu bestimmen, wird in diesen Fällen eine lineare Interpolation zwischen den Preisen des Derivats durchgeführt, die denjenigen ϕ-Werten entsprechen, die ϕ_{i+1} benachbart sind.

7 Ein Anwendungsbeispiel im Investmentbanking

In diesem Kapitel wird beispielhaft ein Prototyp aus einem konkreten Projekt mit Hilfe des entwickelten Testverfahrens getestet. Dazu werden in dem Abschnitt 7.1 das Projekt und in dem Abschnitt 7.2 der in diesem Projekt verfolgte Software-Prozeß beschrieben. Das Testverfahren setzt neben dem zu testendem Prototypen auch den aus der letzten Prototyping-Iteration voraus, um die Veränderungen erkennen zu können. Im folgenden wird der erste Prototyp in dem Abschnitt 7.3 mit der Spezifikation in Larch, der Implementierung, dem ClDG, der CSM, dem CFG und schließlich dem xClDG beschrieben. Der eigentlich zu testende Prototyp wird dann in Abschnitt 7.4 erläutert. Der Test wird anschließend in Abschnitt 7.5 auf der Basis der Algorithmen in den Abbildungen 4.4, 4.5 und 5.2 durchgeführt. Abschließend folgt eine Bewertung des Testverfahrens in Abschnitt 7.6.

7.1 Ein konkretes Projekt aus dem Investmentbanking

Die Anwendbarkeit des in dieser Diplomarbeit entwickelten Testverfahrens wird im folgenden am Beispiel eines konkreten Projekts bei der Dresdner Bank AG in Frankfurt demonstriert. Das Ziel dieses Projekts war die Erweiterung des eingesetzten Software-Systems Stars++ für den Handel mit Zinsderivaten um ein Bewertungsmodell nach dem Heath/Jarrow/Morton-Paradigma, da das schon eingesetzte Black-Modell, wie im letzten Kapitel erläutert, einigen Einschränkungen unterworfen ist. Das Heath/Jarrow/Morton-Modell erlaubt jedoch in seiner ursprünglichen Form keine effiziente Bewertung von Zinsderivaten, so daß ein anderes Zinsstrukturmodell, das zwar dem Heath/Jarrow/Morton-Paradigma folgt, das aber eine pfadunabhängige Bewertung gestattet, implementiert und auf seine Praxistauglichkeit überprüft werden mußte. Die Praxistauglichkeit wird vor allem durch die Qualität, in der das Modell die Realität abbilden kann, bestimmt. Zu diesem Ziel werden die Eingangsparameter des Modells so bestimmt, daß die tatsächlichen Preise für eine Menge von Zinsderivaten, zu denen sie gegenwärtig auf dem Kapitalmarkt gehandelt werden, möglichst genau reproduziert werden. Entscheidend ist dabei auch die Zeit, die zur dieser Kalibrierung benötigt wird. Ausgewählt wurde zu diesem Zweck das im letzten Kapitel ausführlicher erläuterte Zinsstrukturmodell nach RITCHKEN/SANKARASUBRAMANIAN, da die Annahmen, die für die pfadunabhängige Bewertung erfüllt werden müssen, nicht allzu restriktiv sind.

7.2 Vorgehensweise bei der Realisierung

Die Realisierung des Modells folgte einem evolutionären, prototypenorientierten Software-Prozeß, der allerdings bei Projektbeginn nicht vorgegeben war. Die prototypenorientierte Vorgehensweise ergab sich vielmehr aus folgenden Gründen:

1. Die Implementierung des Modells erfordert weitreichende Kenntnisse der Finanzwirtschaft und insbesondere der Stochastik. Es wurden zur Bewältigung der dadurch resultierenden Komplexität des Modells mehrere Teilprobleme identifiziert und diese iterativ implementiert.

2. Die Implementierung des Modells sollte die Bewertung von sehr unterschiedlichen Zinsderivaten erlauben und auch leicht auf zukünftige Derivate erweiterbar sein. Einige Entscheidungen erwiesen sich in Bezug auf die Architektur aus diesem Grund als suboptimal, so daß für den optimalen Systementwurf mehrere Versuche erforderlich waren.

3. Die Umsetzung des Modells ist in zwei Artikeln beschrieben [LRS95a, LRS95b]. Allerdings weisen beide Artikel Unterschiede auf, die zum einen die Berechnung der Übergangswahrscheinlichkeiten und zum anderen die Diskontfaktoren in dem rekombinierenden Baum betreffen. Da eine Anfrage bei den Autoren ergebnislos blieb, wurden beide Versionen als Prototypen implementiert und verglichen.

4. Es hat sich bei dem Test der Praxistauglichkeit des Modells herausgestellt, daß sich beide Artikel zur Implementierung auf einen speziellen Fall beziehen, der zu dem gegenwärtigen Zeitpunkt auf dem deutschen Kapitalmarkt nicht gegeben war[1]. Die Implementierung mußte verallgemeinert und erneut auf die Praxistauglichkeit überprüft werden.

Die Realisierung des Modells konnte also nicht mit einem linearen Prozeßmodell, wie mit dem Wasserfallmodell, durchgeführt werden, da aus den oben genannten Gründen

[1] In beiden Artikel wird von einem konstanten κ ausgegangen. Bei einem konstanten κ kann jedoch kein *Hump* in der Volatilität, also ein Ansteigen für kurze Laufzeiten und wieder ein Abfallen für längere Laufzeiten, abgebildet werden [LRS95a]. Bei den späteren Implementierungen wurde κ durch eine Treppenfunktion dargestellt.

häufig die Ergebnisse aus früheren Phasen der Software-Entwicklung überarbeitet werden mußte. Außerdem war das Software-Produkt nicht das einzige Ziel. Es sollten vielmehr die Eigenschaften des Modells auf der Basis des implemetierten Prototyps untersucht werden.

Das Ziel des Prototypings war damit also nicht die klassische Identifikation der zu realisierenden Anforderungen. Die zu realisierenden Anforderungen waren eindeutig. Vielmehr sollte die software-technische Umsetzung des komplexen Problems überhaupt ermöglicht werden, um aufbauend darauf Wissen über das Modell zu sammeln. Das Prototyping war also, entsprechend der Klassifikation in Kapitel 2, sowohl evolutionär als auch experimentell.

Die Implementierung wurde in der Programmiersprache C++ durchgeführt, da auch das Gesamtsystem in dieser Programmiersprache realisiert ist. Außerdem bietet sich die Objektorientierung bei diesem Projekt auch unabhängig davon an, da alle Zinsderivate gemeinsame Eigenschaften haben, die ihren Preis bestimmen. Dadurch können mit Hilfe des Vererbungsmechanismus verschiedenste Zinsderivate mit demselben Framework bewertet werden.

7.3 Der erste Prototyp

7.3.1 Spezifikation

In den Abbildungen 7.1 und 7.2 ist die Spezifikation der Klasse `RSTree`, die Methoden für die Bewertung von Optionen im Rahmen des Ritchken/Sankarasubramanian-Modells umfaßt, angegeben. Abbildung 7.1 enthält dabei die Spezifikation der Schnittstelle in der BISL Larch/C++ der Programmiersprache C++, während die Abbildung 7.2 die LSL-Spezifikation der der Larch/C++-Spezifikation zugrundeliegenden Funktionen angibt. Der Zustandsbaum wird dabei als gegeben vorausgesetzt, da hier die Anwendung des Testverfahrens und nicht die Realisierung des Modells im Vordergrund steht.

Die Larch/C++-Spezifikation enthält insbesondere die Definition der Schnittstellen der Methoden `PriceOfDerivative` und `PriceInState`. Die `PriceOfDerivative`-Methode gibt dabei den vom Modell implizierten Preis der übergebenen Option wieder, während die `PriceInState`-Methode den Preis der Option in einem bestimmten Zustand als Ergebnis zurückliefert. Jeder Knoten des Baums repräsentiert einen möglichen Umweltzustand, wobei der Wurzelknoten des rekombinierenden Baums den aktuellen Zustand zu dem gegenwärtigen Zeitpunkt entspricht. Aus diesem Grund gleicht ein Aufruf der `PriceOfDerivative`-Methode einem Aufruf der `PriceInState`-Methode mit den Eigenschaften des Wurzelknotens als Übergabeparameter (Zeile 49).

Die `PriceInState`-Methode unterscheidet drei Fälle bei der Berechnung der Zustandspreise. Der erste Fall liegt genau dann vor, wenn der betrachtete Zustand

```
#include <math.h>

//@ uses RSTreeTrait(T);
//@ spec class T;
//@ spec class Deriv;

class RSTree {
private:
  //@ spec T nd;

  double PriceInState(Deriv& dr, double phi, double tm) throw();
  //@ behavior {
  //@ spec double PriceUp, PriceDown, q;
  //@ spec T ndr = nd;
  //@   requires up(nd) = empty;
  //@   ensures result = dr.payoff(nd, phi);
  //@ also
  //@   requires phiExists(nd, phi);
  //@   modifies nd, PriceUp, PriceDown;
  //@   ensures
  //@     nd = up(ndr)
  //@     /\ PriceUp = PriceInState(dr, phiNext(ndr, phi), tm + dt)
  //@     /\ nd = down(ndr)
  //@     /\ PriceDown = PriceInState(dr, phiNext(ndr, phi), tm + dt)
  //@     /\ result = (p(ndr, phi)*PriceUp + (1 - p(ndr, phi))*PriceDown)
  //@          *exp(-r(ndr)*dt);
  //@ also
  //@   requires ~phiExists(nd, phi);
  //@   modifies q;
  //@   ensures q = (phi - phiF(nd, phi))/(phiC(nd, phi) - phiF(nd, phi))
  //@     /\ result = q*PriceInState(dr, nd, phiC(nd, phi), tm)
  //@          + (1 - q)*PriceInState(dr, nd, phiF(nd, phi), tm);
  //@ }

public:
  RSTree() throw();
  //@ behavior {
  //@   constructs self;
  //@   ensures true;
  //@ }

  ~RSTree() throw();
  //@ behavior {
  //@   ensures true;
  //@ }

  double PriceOfDerivative(Deriv& dr) throw();
  //@ behavior {
  //@   ensures result = PriceInState(dr, 0, 0);
  //@ }
}
```

Abbildung 7.1: LARCH/C++-Spezifikation der RSTree-Klasse im Zeitpunkt der Optionsfälligkeit liegt (Zeile 15f). Der Preis der Option ist in diesem Zeitpunkt unabhängig von den nachfolgenden Zuständen eindeutig festgelegt. Er entspricht genau dem Payoff der Option. Der zweite Fall liegt genau dann vor, wenn der übergebene ϕ-Wert, der zusammen mit dem momentanen Kassazinssatz r den Preis der Option in dem entsprechenden Zustand bestimmt, in dem Knoten vorhanden ist (Zeile 18–26). Der übergebene ϕ-Wert ist also nicht bei der Generierung des Baums durch Interpolation weggefallen. Der Optionspreis bestimmt sich in diesem Fall als abgezinster Erwartungswert der Optionspreise in den beiden nachfolgenden Zuständen. Der dritte Fall liegt schließlich genau dann vor, wenn der übergebene ϕ-Wert nicht in dem Knoten des rekombinierenden Baums existiert (Zeile 28–32). Dieser Fall tritt dann ein, wenn die Zahl der Pfade, die zu diesem Knoten führen, größer ist als die zulässige Zahl der ϕ-Werte in einem Knoten. In diesem Fall wird, wie schon im letzten Kapitel erläutert, zwischen dem kleinsten und dem größten ϕ-Wert linear interpoliert. Abgespeichert werden dann der kleinste, der größte und die interpolierten ϕ-Werte. Hier werden die Optionspreise für die beiden ϕ-Werte bestimmt, die den übergebenen ϕ-Wert eingrenzen. Anschließend wird zwischen den Optionspreisen, die sich für diese beiden ϕ-Werte ergeben, linear interpoliert. Das Ergebnis dieser Interpolation ist dann der gesuchte Optionspreis in diesem Zustand.

Die LSL-Spezifikation enthält insgesamt vier Traits, in denen vor allem rekombinierende Bäume (`RecBinTree`) und Bäume im Ritchken/Sankarasubramanian-Modell (`RSTreeTrait`) definiert werden. Die Spezifikation des Traits `RecBinTree` gleicht bis auf die Zeile 12 der Spezifikation eines echten binären Baums, in dem die Zahl der Knoten exponentiell mit der Tiefe des Baums ansteigt. Die Zeile 12 des Traits drückt die charakteristische Eigenschaft rekombinierender Bäume aus. Das Trait `RSTreeTrait` beinhaltet grundlegende Funktionen für rekombinierende Bäume im Ritchken/Sankarasubramanian-Modell. Dazu gehören die Berechnung der Wahrscheinlichkeit für eine Aufwärtsbewegung in dem Baum (Zeile 29f), die Bestimmung des momentanen Kassazinssatzes (Zeile 31) und alle Funktionen im Zusammenhang mit den ϕ-Werten. Das Vorhandensein eines ϕ-Werts in einem Knoten wird `phiExists` geprüft, während der ϕ-Wert in nachfolgenden Zuständen durch `phiNext` berechnet wird. `phiC` (`phiF`) bestimmt schließlich den kleinsten (größten) ϕ-Wert in dem entsprechenden Knoten, der größer (kleiner) ist als der übergebene.

Es soll an dieser Stelle noch auf zwei Punkte im Zusammenhang mit der Spezifikation aufmerksam gemacht werden. Die Spezifikation enthält keinerlei Informationen über die effiziente Implementierung der Bewertungsroutinen, d.h. daß insbesondere die Rekombinierbarkeit des Baums, die erst die effiziente Berechnung der Preise gestattet, nur in einer einzigen Zeile der LSL-Spezifikation berücksichtigt wird. Die Spezifikation der Schnittstelle berücksichtigt diese Eigenschaft des Baums nicht, da das Verhalten der Methoden nach Außen ohne eine Berücksichtigung der Rekombinierbarkeit einfacher zu beschreiben ist. Weiterhin ist in keiner der beiden Spezifikationen eine Aussage über die Zahl der möglichen ϕ-Werte in einem Knoten gemacht worden. Die Zahl der ϕ-Werte ist für die Implementierung der Methoden zur Preisberechnung

```
RecBinTree (E,T): trait
  introduces
    empty: → T
    [__, __, __]: T, E, T → T
    content: T → E
    up, down: T → T
  asserts
    ∀ e: E, t, t1, t2: T
      content([t1, e, t2]) = e;
      up([t1, e, t2]) = t1;
      down([t1, e, t2]) = t2;
      down(up(t)) = up(down(t));  % Bedingung für die Rekombinierbarkeit

doubleArray: trait
  includes Array1(double for E, Int for I, doubleArray for A)

RSNode tuple of shortRate: double, prob, phis: doubleArray

RSTreeTrait (T): trait
  includes
    RecBinTree(RSNode for E, T),
    ModelParam  % Definition von sigma, kappa, dt
  introduces
    phiExists: T, double → Bool
    p, phiNext, phiC, phiF: T, double → double
    r: T → double
  asserts
    ∀ nd: T, phi: double, i: Int
      content(nd).phis[i] = phi ⇒
        p(nd, phi) = content(nd).prob[i];
      r(nd) == content(nd).shortRate;
      phiExits(nd, phi) == content(nd).phis[i] = phi;
      phiNext(nd, phi) == phi + (sigma*sigma*r(nd)*r(nd) - 2*kappa*phi)*dt;
      content(nd).phis[i] < phi ∧ phi < content(nd).phis[i+1] ⇒
        phiC(nd, phi) = content(nd).phis[i+1]
        ∧ phiF(nd, phi) = content(nd).phis[i];
```

Abbildung 7.2: LSL-Spezifikation der RSTree-Klasse

irrelevant, sie wird nur bei der Generierung des Baums benötigt.

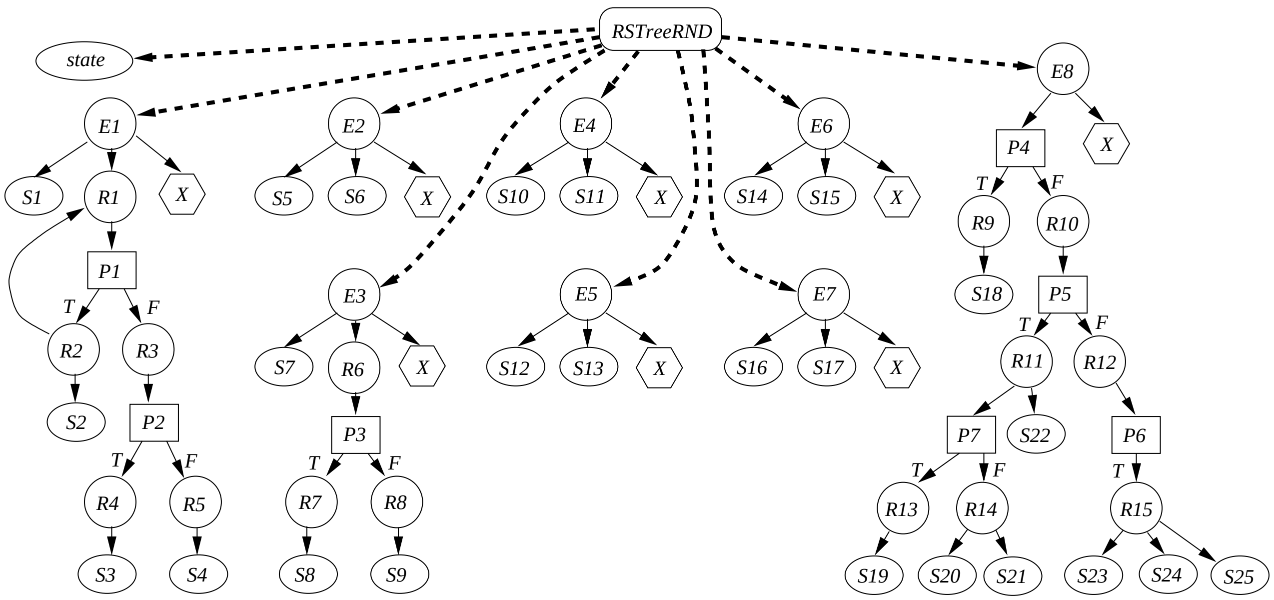
7.3.2 Implementierung und Class Dependence Graph

In der Abbildung 7.3 sind die Implementierung und der ClDG des ersten Prototyps angegeben. Die Implementierung wurde gemäß der Larch/C++-Spezifikation in der Abbildung 7.1 durchgeführt, während der ClDG entsprechend Rothermel/Harrold wiederum aus der Implementierung erzeugt wurde. Dabei wurden Kanten,

```
#include <math.h>
#include "ModelParam.h"
#include "Deriv.h"

class RSTree {
private:
  RSTree *up, *down;
  double r,*prices, *prob, *phis;

  int phiIdx(double phi);
  double p(double phi);
  int phiExists(double phi);
  double phiNext(double phi);
  double phiC(double phi);
  double phiF(double phi);
  double PriceInState(Deriv& dr,
   double phi,double tm) throw();
public:
  RSTree() throw() {}
  ~RSTree() throw() {}
  double PriceOfDerivative(
    Deriv& dr) throw();
};
```




```
E1 int RSTree::phiIdx(double phi){
S1   int i = 0;
P1   while (phi < phis[i])
S2     i++;
P2   if (phi == phis[i])
S3     return i;
S4   else return -1;
   }
E2 double RSTree::p(double phi) {
S5   double i = phiIdx(phi);
S6   return prob[i];
   }
E3 int RSTree::phiExists(double phi){
S7   double i = phiIdx(phi);
P3   if (i != -1)
S8     return 1;
     else return 0;
   }
```

```
E4  double RSTree::phiNext(double phi) {
S10   double result=phi+(pow(sigma*r,2.0)
               - 2.0*kappa*phi)*dt;
S11   return result;
    }
E5  double RSTree::phiC(double phi) {
S12   int i = phiIdx(phi);
S13   return phis[i];
    }
E6  double RSTree::phiF(double phi) {
S14   int i = phiIdx(phi);
S15   return phis[i-1];
    }
E7  double RSTree::PriceOfDerivative(
      Deriv& dr) throw() {
S16   double result=PriceInState(dr,0,0);
S17   return result;
    }
```

```
E8  double RSTree::PriceInState(Deriv& dr, double phi, double tm)
      throw() {
      double result, q;
P4    if (up == NULL)
S18     return dr.payoff(this, phi);
P5    if (phiExists(phi)) {
P7      if (prices[phiIdx(phi)] != 0.0)
S19       result = prices[phiIdx(phi)];
        else {
S20       result = (p(phi)*up->PriceInState(dr,phiNext(phi),tm+dt)
          +(1.0-p(phi))*down->PriceInState(dr,phiNext(phi),tm+dt))
           *exp(-r*dt);
S21       prices[phiIdx(phi)] = result; }
S22     return result;
      }
P6    if (!phiExists(phi)) {
S23     q = (phi - phiF(phi))/(phiC(phi) - phiF(phi));
S24     result = q*PriceInState(dr, phiC(phi), tm)
          + (1.0 - q)*PriceInState(dr, phiF(phi), tm);
S25     return result; }
    }
```

Abbildung 7.3: Implementierung und ClDG des ersten Prototyps die Datenabhängigkeit innerhalb einer Methode anzeigen, zu Gunsten der Übersichtlichkeit nicht dargestellt. An dieser Stelle werden jedoch die Implementierung und der ClDG nicht weiter erläutert, da sie keine größeren Schwierigkeiten für das Verständnis bereiten sollten.

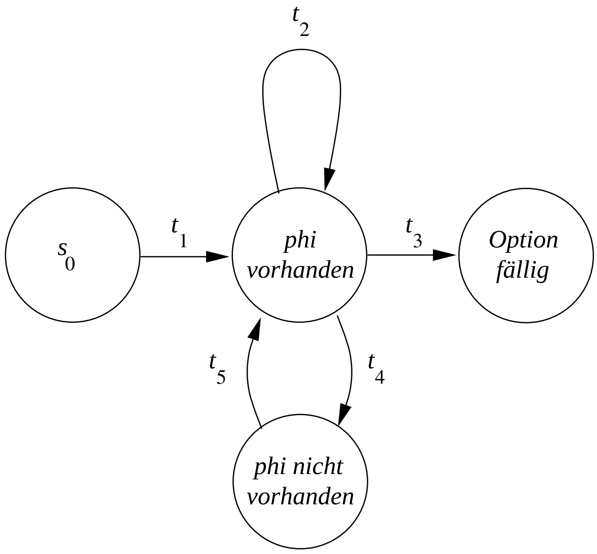
7.3.3 Class State Machine

Die Generierung der CSM ist in diesem Beispiel wesentlich aufwendiger als in dem Beispiel der `Stack`-Klasse. Ein Objekt der Klasse `RSTree` stellt genau genommen nur einen Knoten in dem Zustandsbaum dar. Jeder Knoten enthält, wie auch aus der vierten Zeile der LSL-Spezifikation ersichtlich wird, seine beiden Unterbäume bzw. Verweise auf diese Unterbäume, d.h. der Wurzelknoten enthält die gesamte Information des Baums. Die Aufgabe der CSM in dem `Stack`-Beispiel war offensichtlich, sie sollte das dynamische Verhalten eines einzelnen Objekts der Klasse `Stack` beschreiben. Hier ist allerdings die Aufgabe der CSM nicht mehr so offensichtlich, da hier eine Verkettung von mehreren Objekten derselben Klasse vorliegt. Es ergeben sich mehrere Möglichkeiten. Die CSM könnte so konstruiert werden, daß sie das dynamische Verhalten des gesamten Baums wiedergibt, oder daß sie das Verhalten eines bestimmten Knotens, wie des Wurzelknotens, beschreibt. Im folgenden wird die CSM so entwickelt, daß das dynamische Verhalten des gerade betrachteten Knotens, der durch die Variable nd in der Larch/C++-Spezifikation angegeben wird, beschrieben wird. Somit wird auch indirekt das Verhalten des gesamten Baums erfaßt, da die Variable nd auch ein Attribut des gesamten Baums darstellt.

Die zustandsbestimmenden Attribute der Klasse `RSTree` sind up[2] und $phis$, da das Verhalten der Methode `PriceInState` und damit eines Objekts durch diese Variablen gesteuert wird. Die Zustände eines `RSTree`-Objekts ergeben sich aus den Pre-Zusicherungen dieser Methode. Der erste Zustand ist erfüllt, wenn die Bedingung `up(`nd`)`$=$ *empty* gilt (Zeile 15). Dieser Zustand tritt bei der Preisberechnung dann ein, wenn der betrachtete Knoten in der letzte Ebene des Baums liegt und somit der betrachtete Zustand im Zeitpunkt der Optionsfälligkeit. Es ergeben sich noch zusätzlich zwei weitere Zustände, die durch das Vorhandensein des der Methode `PriceInState` übergebenen ϕ-Werts in dem betrachteten Knoten bestimmt werden. Diese Zustände haben damit die Prädikate **`phiExists(`**nd**`,`** phi**`)`** (Zeile 18) bzw. **`~phiExists(`**nd**`,`** phi**`)`** (Zeile 28).

Eine Besonderheit der hier erzeugten CSM liegt insbesondere in der Form der Zustandswechsel. Zustandswechsel erfolgten in dem Beispiel der `Stack`-Klasse durch Verändern des Attributs *top*. Ein `Stack`-Objekt konnte beispielsweise nur dann vom Zustand *leer* in den Zustand *belegt* wechseln, wenn das Attribut *top* nach einem `push` von null auf eins erhöht wurde. Zustandswechsel erfolgen hier jedoch nicht durch

[2]Hier hätte auch genausogut die Variable *down* ausgewählt werden können, da gilt: $up = NULL \Rightarrow down = NULL$.



$V = \{\texttt{T}\ nd,\ \texttt{doubleArray}\ phis\}$
$F = \{\texttt{PriceInState}(\textbf{Deriv\&},\ \textbf{double},\ \textbf{double})\}$
$S = \{s_0,\ phi\_vorhanden,\ phi\_nicht\_vorhanden,\ Option\_fällig,\ s_f\}$
$T = \{t_1 = (s_0,\ phi\_vorhanden,\ \texttt{PriceInState}(dr, 0, 0),\ wahr,$
$\{nd = \texttt{up}(nd),\ \texttt{PriceInState}(dr, \texttt{phiNext}(nd, 0), dt),$
$nd = \texttt{down}(nd),\ \texttt{PriceInState}(dr, \texttt{phiNext}(nd, 0), dt)\}),$
$t_2 = (phi\_vorhanden,\ phi\_vorhanden,\ \texttt{PriceInState}(dr, phi, tm),\ \texttt{phiExists}(nd, phi),$
$\{nd = \texttt{up}(nd),\ \texttt{PriceInState}(dr, \texttt{phiNext}(nd, phi), tm + dt),$
$nd = \texttt{down}(nd),\ \texttt{PriceInState}(dr, \texttt{phiNext}(nd, phi), tm + dt)\}),$
$t_3 = (phi\_vorhanden,\ Option\_fällig,\ \texttt{PriceInState}(dt, phi, tm), \texttt{up}(nd) = NULL,$
$\{\texttt{dr.payoff}(nd, phi)\}),$
$t_4 = (phi\_vorhanden,\ phi\_nicht\_vorhanden,\ \texttt{PriceInState}(dr, phi, tm),$
$\texttt{\~{}phiExists}(nd, phi),\ \{nd = \texttt{up}(nd),\ \texttt{PriceInState}(dr, \texttt{phiNext}(nd, phi), tm + dt),$
$nd = \texttt{down}(nd),\ \texttt{PriceInState}(dr, \texttt{phiNext}(nd, phi), tm + dt)\}),$
$t_5 = (phi\_nicht\_vorhanden,\ phi\_vorhanden,\ \texttt{PriceInState}(dr, phi, tm), wahr,$
$\{\texttt{PriceInState}(dr, \texttt{phiC}(nd, phi), tm),\ \texttt{PriceInState}(dr, \texttt{phiF}(nd, phi), tm)\})\}$

Abbildung 7.4: CSM der `RSTree`-Klasse

Verändern der Attribute. Die zustandsbestimmenden Attribute sind sogar, wird nur ein einzelnes Objekt betrachtet, konstant. Sie „ändern“ sich dadurch, daß ein anderes Objekt betrachtet wird, das üblicherweise Attribute mit anderen Werten besitzt. Die CSM beschreibt damit genau genommen das Verhalten eines Superknotens, der die Eigenschaften aller Knoten besitzt.

Abbildung 7.4 zeigt die CSM der `RSTree`-Klasse. Der Knoten s_0 hat jedoch eine andere Bedeutung als in dem `Stack`-Beispiel. Der Zustand s_0 stellt hier den Zustand vor dem Aufruf der Methode `PriceInState` dar. In diesem Zustand kann also schon der Konstruktor der Klasse aufgerufen worden sein und ein Objekt existieren. Aus diesem Grund enthält die CSM auch keinen Knoten, der den Zustand nach einem Destruktoraufruf darstellt. Der Knoten s_f, der einen Fehlerzustand repräsentiert, fehlt

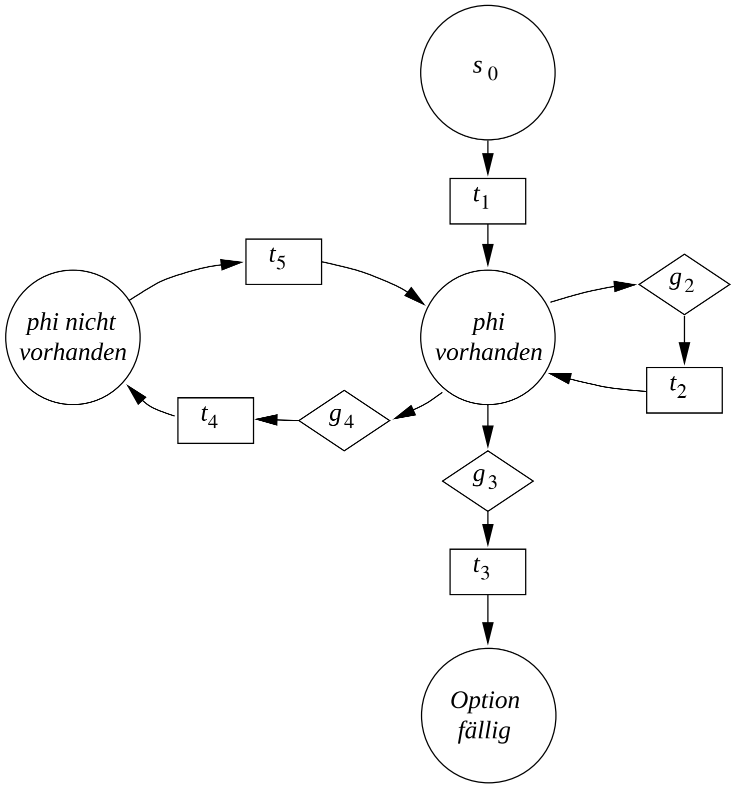


Abbildung 7.5: CFG der `RSTree`-Klasse

ebenfalls, da nur eine einzige Methode betrachtet wird, die für alle Objektzustände definiert ist.

Streng genommen erfüllt die `RSTree`-Klasse nicht die Voraussetzungen, die für die Generierung der CSM erfüllt sein müssen. Tse/Xu [TX96] setzen sogenannte *mutable Objects* voraus, d.h. Objekte, die verschiedene abstrakte Werte annehmen können. Diese Voraussetzung wird hier jedoch durch geeignete Definition der Zustandsprädikate umgangen.

7.3.4 Class Flow Graph

Die Generierung des CFGs erfolgt gemäß [HKC95] und bereitet keine größeren Schwierigkeiten (siehe Abbildung 7.5). Im folgenden wird die Ermittlung der zu überdeckenden Definition/Referenz-Paare beschrieben.

Die Ermittlung dieser Definition/Referenz-Paare gestaltet sich durch die oben genannten Besonderheiten der `RSTree`-Klasse wieder etwas schwieriger. In dem `Stack`-Beispiel konnte jede Anweisung, die den Wert des Attributs veränderte, als eine Definition des Attributs identifiziert werden. In dem Beispiel der `RSTree`-Klasse existieren jedoch keine Anweisungen, die den Wert der Attribute direkt verändern. Die

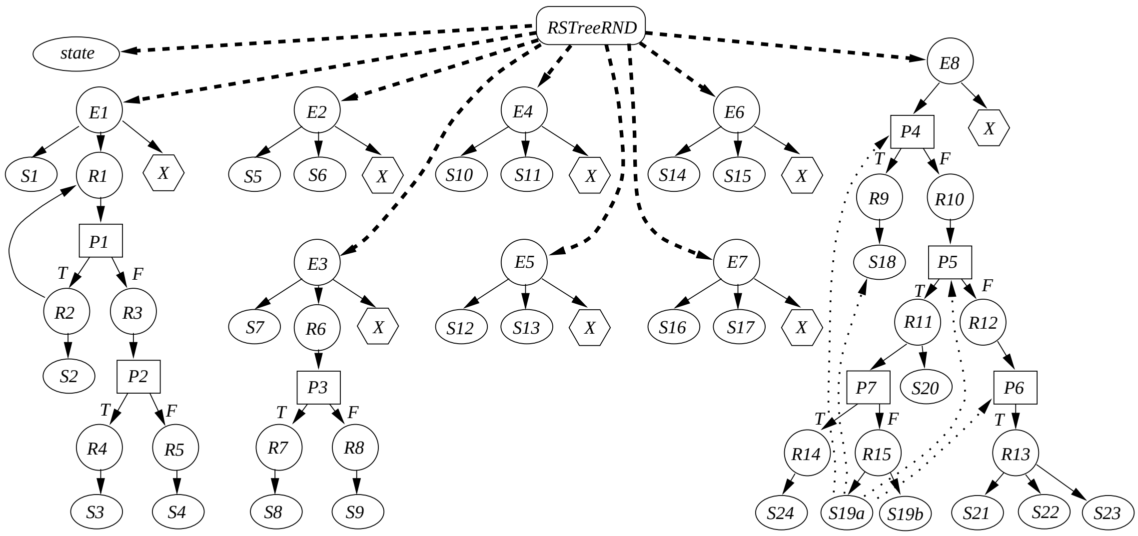


Abbildung 7.6: xClDG des ersten Prototyps

Veränderung der Attribute kommt, wie schon oben erklärt, dadurch zustande, daß ein anderer Knoten, der in der Regel andere Attributwerte besitzt, betrachtet wird. Da der gegenwärtig betrachtete Knoten in der Variablen *nd* abgespeichert wird, können alle Anweisungen, die den Wert der Variablen *nd* verändern, als Definitionen der Attribute *up* und *phis* aufgefaßt werden. Somit befinden sich in den Knoten t_1, t_2 und t_4 Definitionen der Attribute.

In diesen Knoten befinden sich auch logischer Weise Berechnungsreferenzen des *up*-Attributs, da das Attribut *up* den Wert enthält, den die Variable *nd* bei einer Aufwärtsbewegung in dem Baum annimmt. Die ϕ-Werte in einem Knoten werden bei der Berechnung des Payoffs der zu bewertenden Option benötigt, somit enthält der Knoten t_3 also eine Berechnungsreferenz dieses Attributs. Die ϕ-Werte werden zwar auch bei der Berechnung der ϕ-Werte in den nachfolgenden Knoten benötigt, da aber der Baum und damit die ϕ-Werte in den Knoten des Baums als gegeben vorausgesetzt werden, werden diese Berechnungsreferenzen nicht berücksichtigt.

Entscheidungsreferenzen befinden sich in allen Knoten des CFG, die Zustände des CSM darstellen. Die Knoten, die die Zustände *phi_vorhanden* und *phi_nicht_vorhanden* abbilden, beinhalten Entscheidungsreferenzen des Attributs *phis*, während der Knoten für den Zustand *Option_fällig* eine Entscheidungsreferenz des Attributs *up* enthält. Außerdem sind in den Knoten g_2 und g_4 Entscheidungsreferenzen von *phis* enthalten und im Knoten g_3 eine Entscheidungsreferenz von *up*.

7.3.5 Extended Class Dependence Graph

Abbildung 7.6 zeigt den xClDG des ersten Prototyps. Der ClDG aus der Abbildung 7.3 wurde dazu so um mdd-Kanten erweitert, daß das alle-Definitionen-Krite-

```
11:  double PriceInState(Deriv& dr, double phi, double tm) throw();
12:  //@ behavior {
13:  //@ spec double PriceUp, PriceDown, q;
14:  //@ spec T ndr = nd;
15:  //@   requires tm = dr.maturity;
16:  //@   ensures result = dr.payoff(nd, phi);
17:  //@ also
18:  //@   requires phiExists(nd, phi);
19:  //@   modifies nd, PriceUp, PriceDown;
20:  //@   ensures
21:  //@     nd = up(ndr)
22:  //@     /\ PriceUp = PriceInState(dr, phiNext(ndr, phi), tm + dt)
23:  //@     /\ nd = down(ndr)
24:  //@     /\ PriceDown = PriceInState(dr, phiNext(ndr, phi), tm + dt)
25:  //@     /\ result = max((p(ndr, phi)*PriceUp + (1 - p(ndr, phi))*PriceDown)
26:  //@          *exp(-r(ndr)*dt), dr.payoff(ndr, phi));
27:  //@ also
28:  //@   requires ~phiExists(nd, phi);
29:  //@   modifies q;
30:  //@   ensures q = (phi - phiF(nd, phi))/(phiC(nd, phi) - phiF(nd, phi))
31:  //@     /\ result = q*PriceInState(dr, nd, phiC(nd, phi), tm)
32:  //@          + (1 - q)*PriceInState(dr, nd, phiF(nd, phi), tm);
33:  //@ }
```

Abbildung 7.7: LARCH/C++-Spezifikation der veränderten `RSTree`-Klasse

rium erfüllt wird. Es wurden folgende Definition/Referenz-Paare für den Test des Attributs *phis* gebildet: (t_1, g_4), (t_2, t_3), (t_4, g_3). Das Attribut *up* wird dagegen durch folgende Paare getestet: (t_1, g_3), (t_2, g_3), (t_4, g_3).

7.4 Der zweite Prototyp

Die Veränderungen, die an dem ersten Prototypen durchgeführt wurden, sind sowohl funktionaler als auch struktureller Art. Die Funktionalität wird so erweitert, daß auch amerikanische Optionen bewertet werden können. Amerikanische Optionen können im Gegensatz zu europäischen Optionen in jedem Zeitpunkt innerhalb der Optionslaufzeit ausgeübt werden. Dadurch ist der Wert einer amerikanischen Option in jedem Zeitpunkt mindestens so hoch wie der Wert einer europäischen Option, da eine amerikanische Option eine europäische umfaßt. Der Wert kann aber auch in einem bestimmten Zeitpunkt größer sein, und zwar genau dann, wenn die vorzeitige Ausübung in diesem Zeitpunkt günstiger ist als die Ausübung am Laufzeitende. Diese Eigenschaft amerikanischer Optionen wirkt sich in der Spezifikation des Prototyps lediglich auf die Zeilen 25 und 26 der LARCH/C++-Spezifikation aus (siehe Abbildung 7.7). Hierbei wird unterstellt, daß `dr.payoff` bei europäischen Optionen in allen Zeitpunkten, bis auf den Zeitpunkt der Optionsfälligkeit, null als Payoff

```
E8  double RSTree::PriceInState(Deriv& dr, double phi, double tm) throw() {
      double result, q, dis;
P4    if (up == NULL)
S18     return dr.payoff(this, phi);
P5    if (phiExists(phi)) {
P7      if (prices[phiIdx(phi)] != -1.0)
S19       result = prices[phiIdx(phi)];
        else {
S20       result = max((p(phi)*up->PriceInState(dr, phiNext(phi), tm + dt)
            +(1.0 - p(phi))*down->PriceInState(dr, phiNext(phi), tm + dt))
            *exp(-r*dt), dr.payoff(this, dr));
S21       prices[phiIdx(phi)] = result;
        }
S22     return result;
      }
P6    if (!phiExists(phi)) {
S23     q = (phi - phiF(phi))/(phiC(phi) - phiF(phi));
S24     result = q*PriceInState(dr, phiC(phi), tm)
          + (1.0 - q)*PriceInState(dr, phiF(phi), tm);
S25     return result;
      }
    }
```

Abbildung 7.8: Implementierung des zweiten Prototyps

zurückliefert.

Die strukturelle Veränderung wird zum Zwecke der Korrektur durchgeführt. Durch die Rekombinierbarkeit des Baums werden bei der Bewertung von Optionen die meisten Knoten mehr als einmal besucht. Da aber der Preis einer Option jedesmal derselbe ist[3], kann dieser einmal berechnet und in dem Knoten abgespeichert werden, um dadurch Rechenzeit einzusparen. Die Rekombinierbarkeit wird in der Implementierung in Zeile $P7$ ausgenutzt. Hier wird der in dem Knoten gespeicherte Optionspreis für das entsprechende ϕ auf seine Gültigkeit untersucht und gegebenfalls zurückgegeben. Die Gültigkeit wird durch einen einfachen Vergleich des abgespeicherten Werts mit dem Initialwert durchgeführt. Falls schon der Optionspreis berechnet wurde und gültig ist, so unterscheiden sich diese beiden Werte und der abgespeicherte Wert kann zurückgegeben werden. Ansonsten muß der entsprechende Optionspreis neu bestimmt werden. Unglücklicherweise wurde ein Wert als Initialwert gewählt, der auch einen gültigen Optionswert darstellen kann, so daß in einigen Fällen gültige Optionspreise als ungültig betrachtet und neu bestimmt werden. Aus diesem Grund wurde der Initialwert in der Zeile $P7$ auf -1 gesetzt.

Die Generierung des ClDGs, der CSM und des CFGs erfolgen analog zu denjenigen des ersten Prototyps und werden hier nicht nochmals erläutert. Der xClDG wird

[3] Zumindest bei den Optionen, die hier betrachtet werden. Gegenbeispiel: *Asiatische Optionen* [Hul96].

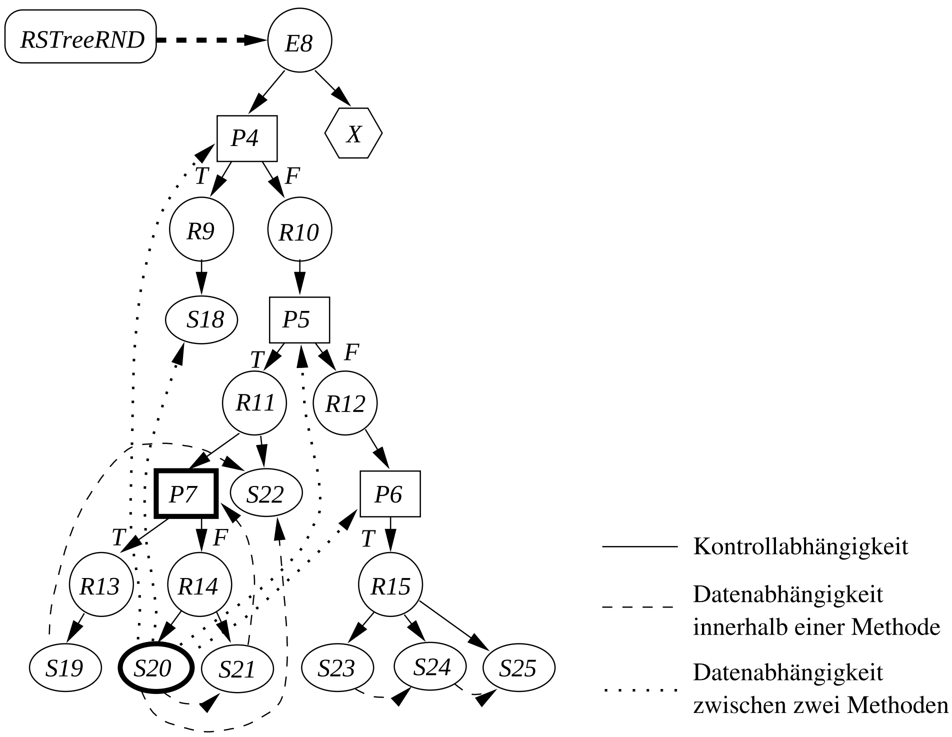


Abbildung 7.9: `PriceInState`-Methode

wiederum durch Erweitern des ClDGs um mdd-Kanten erzeugt. Diese mdd-Kanten werden gemäß den zu überdeckenden Definition/Referenz-Paaren des ersten Prototyps gesetzt, da diese Paare auch beim zweiten Prototyp ihre Gültigkeit haben und keine neuen Definitionen durch die erweiterte Funktionalität des Prototyps entstanden sind. Der gesamte xClDG wird hier jedoch nicht abgebildet, da er sich nicht von dem xClDG des ersten Prototyps unterscheidet.

7.5 Auswahl der Testdaten

In den folgenden Ausführungen wird nur die Methode `PriceInState` betrachtet, da die Veränderungen des Prototyps nur diese Methode betreffen und das Verständnis nicht unnötig erschwert werden soll. Abbildung 7.9 zeigt diese Methode als Teil des xClDGs des zweiten Prototyps.

Die Prozedur `SelectClassTests` sieht nach der Generierung der xClDGs der beiden Klassen die *visited* Markierung jedes Knotens vor (Abbildung 4.4, Zeile 9f). Anschließend werden in die *affected*-Mengen aller Knoten, die durch veränderte Attribute betroffen sind, alle Testdaten eingefügt, um den Test der veränderten Attribute sicherzustellen (Zeile 11f). Da aber in dem vorliegenden Beispiel keine Veränderungen an den Attributen vorgenommen wurden, müssen auch keine Testdaten in die *affected*-

Mengen aufgenommen werden. Danach wird die Menge T' der ausgewählten Testdaten auf die leere Menge gesetzt (Zeile 13) und die `xCompare`-Prozedur mit den Knoten $E8$ und $E8'$[4] als Eingaben aufgerufen (Zeile 14 bis 18).

Die `xCompare`-Prozedur markiert in einem ersten Schritt die übergebenen Knoten als *visited* (Abbildung 5.2, Zeile 8) und versucht mit Hilfe der `GetCorresp`-Prozedur eine Zuordnung zwischen den Nachfolgern dieser Knoten zu finden (Zeile 9). Nach dieser Zuordnung der Nachfolger werden die Mengen T'' und T''' der ausgewählten Testdaten initialisiert und die Nachfolger paarweise miteinander verglichen. Da aber kein kontrollabhängiger Nachfolger der beiden Knoten $E8$ und $E8'$ verändert wurde und nicht auch (*mdd-*)$affected$ ist, wird die `xCompare`-Prozedur rekursiv auf den Nachfolgern aufgerufen.

Die erste Veränderung, die bei der rekursiven Ausführung der Prozedur `xCompare` festgestellt wird, betrifft einen Nachfolger von $R11$. In der `GetCorresp`-Prozedur wird der Knoten $P7$ als ein veränderter kontrollabhängiger Nachfolger von $R11'$ entdeckt und, da der Knoten $P7$ keine einfache Zuweisung enthält, der Wert $false$ zurückgegeben. Durch diesen Rückgabewert werden in der `xCompare`-Prozedur alle Testdaten in $R11.History$ für den strukturellen Test ausgewählt (Zeile 9 + 32). Es werden dagegen keine Testdaten für den funktionalen Test ausgewählt, obwohl Änderungen in der Spezifikation zu strukturellen Änderungen geführt haben. Diese befinden sich jedoch unterhalb des Knotens $P7$.

Angenommen der Knoten $P7$ ist nicht verändert worden, d.h. der Prototyp wurde nur funktional geändert und die einzige Veränderung ist in dem Knoten $S20$. In diesem Fall würde die `GetCorresp`-Prozedur ausgeführt auf den Knoten $R14$ und $R14'$ den Wert *true* zurückliefern, da es sich dabei nur um eine Veränderung in einem Knoten handelt, der eine einfache Zuweisung beinhaltet. Nach der Initialisierung der Mengen T'' und T''' werden die daten- und mdd-abhängigen Nachfolger des veränderten Knotens in den Zeilen 12 bis 22 der `xCompare`-Prozedur betrachtet. Der Knoten $S20$ hat zum einen die datenabhängigen Nachfolger $S21$ und $S22$. Diese Knoten werden, da sie noch nicht besucht worden sind, $affected$ markiert, d.h. die Testdaten in $R14.History$ werden an $S21.affected$ und $S22.affected$ hinzugefügt. Zum anderen besitzt der Knoten $S20$ die mdd-abhängigen Nachfolger $P4$, $S18$, $P5$ und $P6$. Die ersten beiden dieser Knoten sind durch die `xCompare`-Prozedur schon besucht worden, d.h. daß in diesem Fall diejenigen Testdaten ausgewählt werden, die den Knoten $P4$ bzw. $S18$ und den Knoten $S20$ ausführen (Zeile 19f). Die beiden anderen Nachfolger wurden hingegen noch nicht besucht, d.h. diese Knoten werden *mdd-*$affected$ markiert, also die Eingaben in $R14.History$ an $P5.mdd\_affected$ und $P6.mdd\_affected$ hinzugefügt.

[4] Die Knoten in dem xClDG des zweiten Prototyps werden im folgenden durch einen Apostroph kenntlich gemacht.

7.6 Bewertung des Testverfahrens

Das in dieser Diplomarbeit vorgeschlagene Testverfahren für die objektorientierte, prototypbasierte Software-Entwicklung besitzt neben den Vorteilen, die schon in dieser Arbeit genannt wurden, einige Nachteile, die beim obigen Test sichtbar wurden.

- Formale Spezifikation
  Das Testverfahren setzt eine formale Spezifikation voraus, die jedoch einige Probleme bereiten kann. Häufig ist eine formale Spezifikation, wenn sie überhaupt erstellt werden kann, mit einem nicht zu unterschätzendem Aufwand verbunden. Es stellt sich dann die Frage, ob dieser Aufwand überhaupt gerechtfertigt ist. Dagegen kann aber argumentiert werden, daß ein funktionaler Test eine möglichst aussagekräftige Spezifikation voraussetzt, damit überhaupt funktionale Fehler entdeckt werden können. Das Problem der formalen Spezifikation ist also nicht ein Problem des hier vorgestellten Verfahrens, sondern ein Problem an sich.

- Zustandsautomat
  Aus der formalen Spezifikation wird ein Zustandsautomat generiert, der das dynamische Verhalten der Klasse sichtbar machen soll. Die Spezifikation kann also damit auch in Form eines Automats vorliegen. Das Problem, das aber trotzdem bleibt, ist die Tatsache, daß in manchen Fällen kein Zustandsautomat entwickelt werden kann. In dem obigen Beispiel konnte der Zustandsautomat nur über einen Umweg konstruiert werden, da die Attribute der einzelnen Objekte konstant waren.

Diese Nachteile rücken aber bei hohen Qualitätsanforderungen an die zu entwickelnde Software in den Hintergrund. Das Verfahren besitzt eine hohe Zuverlässigkeit, die dazu führt, daß weniger Fehler unentdeckt bleiben. Durch die Effizienz dieses Verfahrens ist es bei der Entwicklung von Software mit hoher Qualität anderen Verfahren sogar vorzuziehen. Das Verfahren sieht nur den Test von Abschnitten der Implementierung vor, die verändert wurden und damit auch Fehler enthalten können. Dadurch werden also beim Testen Ressourcen eingespart, die wiederum in das Testen oder aber auch in andere Phasen der Software-Entwicklung investiert werden können.

8 Zusammenfassung und Ausblick

In der vorliegenden Diplomarbeit wurde ein Testverfahren für die Software-Entwicklung im Investmentbanking entwickelt und die Anwendbarkeit an einem konkreten Problem demonstriert. Dieses Testverfahren wurde zwar speziell für das Investmentbanking entwickelt, ist aber überall dort anwendbar, wo eine objektorientierte, prototypbasierte Software-Entwicklung gegeben ist. Die hohe Zuverlässigkeit legt den Einsatz dieses Testverfahrens insbesondere auch in sicherheitskritischen Bereichen nahe.

Das Testverfahren bestimmt auf der Grundlage der Spezifikationen und der Implementierungen zweier Prototypen diejenigen Testdaten aus einer Testdatenmenge, die für den zuverlässigen Test des letzteren Prototyps benötigt werden. Dazu werden die xClDGs der beiden Prototypen erzeugt und die Unterschiede analysiert. Die ausgewählten Testdaten überdecken alle Abschnitte des zu testenden Prototyps, die entweder zur Korrektur verändert wurden oder aber zu veränderten Abschnitten abhängig sind. Die Veränderungen können dabei sowohl struktureller als auch funktionaler Art sein.

Das Verfahren von Rothermel/Harrold, das eigentlich für den Regressionstest entwickelt wurde, wurde zu diesem Zweck so erweitert, daß auch die Spezifikation des implementierten Problems berücksichtigt wird. Damit liegt auch eine andere Anwendung des vorgestellten Verfahrens nahe. Das Verfahren kann auch im Regressionstest eingesetzt werden. Es besitzt alle Vorteile des Verfahrens von Rothermel/Harrold und zusätzlich den Vorteil, daß auch funktionale Veränderungen überprüft werden. Allerdings konnte der Einsatz des Verfahrens im Regressionstest aus Zeitgründen nicht weiter untersucht werden.

Ein effizienter Test erfordert geeignete Software-Werkzeuge, die möglichst viele Tätigkeiten während des Testens selbstständig durchführen. Die Implementierung eines Software-Werkzeugs war zwar nicht Ziel dieser Arbeit, ist aber für die Anwendbarkeit und vor allem für die Akzeptanz entscheidend. Eine interessante Aufgabe wäre also die Implementierung eines Software-Werkzeugs, denn das Verfahren findet nur dann Einsatz, wenn dem Praktiker die Vorteile des Verfahrens auch offensichtlich sind.

Literaturverzeichnis

[AktG] *Aktiengesetz – GmbH-Gesetz*, Stand 01.12.95, Deutscher Taschenbuch Verlag, München.

[Bal98] Ball, Thomas (1998): On the Limit of Control Flow Analysis for Regression Test Selection, *Proceedings International Symposium on Software Testing and Analysis* (*ISSTA '98*), März, Clearwater Beach, 134–142; `http://www.bell-labs.com:80/~tball/papers/issta98.ps.gz`.

[BBP96] Barbey, Stephane; Buchs, Didier; Peraire, Cecile (1996): A Theory of Specification-Based Testing for Testing for Object-Oriented Software, *Proceedings European Dependable Computing Conference* (*EDCC '96*), Oktober, Taormina, 303–320; `ftp://lglftp.epfl.ch/pub/Papers/barbey-edcc2-96.ps`.

[BE93] Birrer, Andreas; Eggenschwiler, Thomas (1993): Frameworks in the Financial Engineering Domain: An Experience Report, *Proceedings European Conference on Object-Oriented Programming* (*ECOOP '93*), Juli, Kaiserslautern, 21–35; `http://www.ubs.com/e/index/about/ubilab/print_versions/ps/bir93b.ps.gz`.

[Bla76] Black, Fischer (1976): The Pricing of Commodity Contracts, *Journal of Financial Economics* **3** (3), 167–179.

[Bla94] Blaschek, Günther (1994): *Object-Oriented Programming with Prototypes*, Springer Verlag, Berlin.

[Blz96] Balzert, Helmut (1996): *Lehrbuch der Software-Technik: Software-Entwicklung*, Spektrum Akademischer Verlag, Heidelberg.

[Blz98] Balzert, Helmut (1998): *Lehrbuch der Software-Technik: Software-Management, Software-Qualitätssicherung, Unternehmensmodellierung*, Spektrum Akademischer Verlag, Heidelberg.

[Boa83] Boar, Bernard H. (1983): *Application Prototyping: A Requirements Definition Strategy for the 80s*, John Wiley & Sons, New York.

[BR96] Baxter, Martin; Rennie, Andrew (1996): *Financial Calculus – An Introduction to Derivative Pricing*, Cambridge University Press, Cambridge.

[BS73] Black, Fischer; Scholes, Myron (1981): The Pricing of Options and Corporate Liabilities, *Journal of Political Economy* **81** (3), 637–654.

[BS82] Brennan, Michael J.; Schwartz, Eduardo S. (1982): An Equilibrium Model of Bond Pricing and a Test of Market Efficiency, *Journal of Financial and Quantitative Analysis* **17** (3), 301–329.

[BS94] Barbey, Stephane; Strohmeier, Alfred (1994): The Problematics of Testing Object-Oriented Software, *Proceedings Second Conference on Software Quality Management*, Juli, Edinburgh, 411–426; `ftp://lglftp.epfl.ch/pub/Papers/barbey-problematic_of_toos.ps`.

[BS95] Bosman, Oscar; Schmidt, Heinz W. (1995): *Object Test Coverage Using Finite State Machines*, Technical Report TR-CS-95-06, Department of Computer Science, Australian National University; `http://cs.anu.edu.au/techreports/1995/TR-CS-95-06.ps.gz`.

[BV94] Bank-Verlag GmbH (1994): *Basisinformationen über Vermögensanlagen in Wertpapieren*, Bank-Verlag, Köln.

[CC95] Canfora, G.; Cimitile, A. (1995): Algorithms for Program Dependence Graph Production, *Proceedings International Conference on Software Maintenance*, Oktober, Opio (Nice), 157–166.

[CIR85] Cox, John C.; Ingersoll, Jonathan E.; Ross, Stephen A. (1985): A Theory of the Term Structure of Interest Rates, *Econometrica* **53**, 385–407.

[CR85] Cox, John C.; Rubinstein, Mark (1985): *Option Markets*, Prentice Hall, Englewood Cliffs.

[CS89] Connell, John L.; Shafer, Linda I. (1989): *Structured Rapid Prototyping*, Yourdon Press, Englewood Cliffs.

[CS95] Connell, John L.; Shafer, Linda I. (1995): *Object-Oriented Rapid Prototyping*, Yourdon Press, Englewood Cliffs.

[CTC+98] Chen, H.Y.; Tse, T.H.; Chan, F.T.; Chen, T.Y. (1998): In Black and White – An Integrated Approach to Object-Oriented Program Testing, *ACM Transactions on Software Engineering and Methodology* **7** (3), 250–295; `http://www.csis.hku.hk/~tse/Papers/ublackps.zip`.

[CW92] Copeland, Thomas E.; Weston, J. Fred (1992): *Financial Theory and Corporate Policy*, 3. Auflage, Addison-Wesley, Reading.

[DB98] Deutsche Börse (1998): *Rekordumsätze bei Aktien und Derivaten*, Pressemitteilung vom 01.04.1998, `http://www.exchange.de/press_releases/896460631_d.html`.

[DF93] Dick, Jeremy; Faivre, Alain (1993): Automating the Generating and Sequencing of Test Cases from Model-Bases Specification, *Proceedings International Symposium of Formal Methods: Industrial Strengh Formal Methods*, April, Odensee, 268–284.

[DF94] Doong, Roong-Ko; Frankl, Phyllis G. (1994): The ASTOOT Approach to Testing Object-Oriented Programs, *ACM Transactions on Software Engineering and Methodology* **3** (2), 101–130.

[DIN66272] Deutsches Institut für Normung (1994): *Bewerten von Softwareprodukten – Qualitätsmerkmale und Leitfaden zu ihrer Verwendung*, Berlin.

[EG92] Eggenschwiler, Thomas; Gamma, Erich (1992): ET++SwapsManager: Using Object Technology in the Financial Engineering Domain, *Proceedings International Computer Software & Applications Conference* (*COMPSAC '92*), Oktober, Vancouver, 166–177; `http://www.ubs.com/e/index/about/ubilab/print_versions/ps/swaps-oopsla92.ps.gz`.

[FH94] Franke, Günter; Hax, Herbert (1994): *Finanzwirtschaft des Unternehmens und Kapitalmarkt*, Springer-Verlag, Berlin.

[Fis77] Fischer, Kurt F. (1977): A Test Case Selection Method for the Validation of Software Maintenance Modifications, *Proceedings International Computer Software & Applications Conference (COMPSAC '77)*, November, Chicago, 421–426.

[Flo84] Floyd, Christiane (1984): A Systematic Look at Prototyping, in: Budde, R.; Kuhlenkamp, K.; Mathiassen, L.; Züllighoven, H. (Hrsg.), *Approaches to Prototyping*, Springer Verlag, Berlin, 1–18.

[FOW87] Ferrante, Jeanne; Ottenstein, Karl J.; Warren, Joe D. (1987): The Program Dependence Graph and its Use in Optimization, *ACM Transactions on Programming Languages and Systems* **9** (3), 319–349.

[FS97] Fowler, Martin; Scott, Kendall (1997): *UML Distilled – Applying the Standard Object Modelling Language*, Addison-Wesley, Reading.

[GDT93] Graham, John A.; Drakeford, Andrew C. T.; Turner, Chris D. (1993): The Verification, Validation and Testing of Object Oriented Systems, *BT Technology Journal* **11** (3), 79–88.

[GGH93] Garland, Stephen J.; Guttag, John V.; Horning, James J. (1993): An Overview of Larch, in: Lauer, Peter E. (Hrsg.), *Functional Programming, Concurrency, Simulation and Automated Reasoning*, Lecture Notes in Computer Science **693**, Springer-Verlag, 329–348; `http://hypatia.dcs.qmw.ac.uk:80/sites/edu/cs.iastate.edu/papers/TR96-01/TR.ps.gz`.

[GHG+93] Guttag, John V.; Horning, James J.; Garland, Stephen J.; Jones, Kevin D.; Modet, Andres; Wing, Jeannette M. (1993): *Larch: Languages and Tools for Formal Specification*, Springer-Verlag; `http://www.sds.lcs.mit.edu/spd/larch/pub/larchBook.ps`.

[GHK+98] Graves, Todd L.; Harrold, Mary Jean; Kim, Jung-Min; Porter, Adam; Rothermel, Gregg (1998): An Empirical Study of Regression Test Selection Techniques, *Proceedings 20th International Conference on Software Engineering* (*ICSE '98*), April, Kyoto, 188–197; `http://www.cs.umd.edu/~jmkim/papers/icse98.ps`.

[GJM91] Ghezzi, Carlo; Jazayeri, Mehdi; Mandrioli, Dino (1991), *Fundamentals of Software Engineering*, Prentice Hall, Englewood Cliffs.

[Gri95] Grimm, Klaus (1995): *Systematisches Testen von Software – Eine neue Methode und eine effektive Teststrategie*, GMD-Bericht Nr. 251, Oldenbourg Verlag, München.

[Hav93] Haverkamp, Tom (1993): *Ein Zweifaktormodell der Zinsstruktur: Empirische Analyse und Bewertung zinsderivater Finanzinstrumente*, Dissertation, Schweizerisches Institut für Banken und Finanzen an der Hochschule St. Gallen.

[Hei97] Heitmann, Frank (1997): *Arbitragefreie Bewertung von Zinsderivaten*, Deutscher Universitäts Verlag, Wiesbaden.

[Hes97] Hesse, Wolfgang (1997): Wie evolutionär sind die objektorientierten Analysemethoden? Ein kritischer Vergleich, *Informatik-Spektrum* **20** (1), 21–28.

[Hie97] Hierons, Rob (1997): Testing from a Z Specification, *Software Testing, Verification and Reliability* **7** (1), 19–33.

[HJM+92] Heath, David; Jarrow, Robert; Morton, Andrew; Spindel, Mark (1992): Easier Done Than Said, *RISK* **5** (9), 77–80.

[HJM92] Heath, David; Jarrow, Robert; Morton, Andrew (1992): Bond Pricing and the Term Structure of Interest Rates: A New Methodology for Contingent Claim Valuation, *Econometrica* **60**, 77–105.

[HKC95] Hong, Hyoung; Kwon, Yong Rae; Cha, Sung Deok (1995): Testing of Object-Oriented Programs Based on Finite State Machines, *Proceedings Asia-Pacific Software Engineering Conference*, Dezember, Brisbane, 234–241; `http://salmosa.kaist.ac.kr/LAB/MEMBER/CHA/PUB_LIST/ic/apsec95-2.ps`.

[HL86] Ho, Thomas S.Y.; Lee, Sang Bin (1986): The Term Structure Movements and Pricing Interest Rate Contingent Claims, *Journal of Finance* **41**, 1011–1029.

[HR94] Harrold, Mary Jean; Rothermel, Gregg (1994): Performing Data Flow Testing on Classes, *Proceedings Second ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Dezember, New Orleans, 154–163.

[Hul96] Hull, John C. (1996): *Options, Futures, and Other Derivatives*, 3. Auflage, Englewood Cliffs.

[HW90] Hull, John C.; White, Alan (1990): Pricing Interest-Rate Derivative Securities, *Review of Financial Studies* **3** (4), 573–592.

[Jar96] Jarrow, Robert A. (1996): *Modelling Fixed Income Securities and Interest Rate Options*, McGraw-Hill, New York.

[KSG+94] Kung, David C.; Suchak, N.; Gao, Jerry; Hsia, Pei; Toyoshima, Yasufumi; Chen, Chris (1994): On Object State Testing, *Proceedings 18th Annual International Computer Software & Applications Conference (COMPSAC '94)*, 222–227; `ftp://pepe.uta.edu/pub/publications/compsac94.ps`.

[Lea96] Leavens, Gary T. (1996): *An Overview of Larch/C++: Behaviorial Specifications for C++ Modules*, Technical Report TR96-01a, Computer Science Department, Iowa State University; `http://cs-tr.cs.iastate.edu/Dienst/Repository/2.0/Body/ncstrl.iastate%2fTR96-01a/postscript`.

[Lea97a] Leavens, Gary T. (1997): *Larch/C++ Reference Manual*, Department of Computer Science, Iowa State University; `ftp://ftp.cs.iastate.edu/pub/larchc++/lcpp.ps.gz`.

[Lea97b] Leavens, Gary T. (1997): *Larch/C++ – An Interface Specification Language for C++*, `ftp://ftp.cs.iastate.edu/pub/larchc++/poster.ps.gz`.

[Lea98] Leavens, Gary T. (1998): *Larch Frequently Asked Questions*, `http://www.cs.iastate.edu/~leavens/larch-faq.html`.

[Lev95] Leveson, Nancy G. (1995): *Safeware: System Safety and Computers*, Addison-Wesley, Reading; `http://www.cs.washington.edu/research/projects/safety/www/papers/therac.ps`.

[Lig92] Liggesmeyer, Peter (1992): Testen, Analysieren und Verifizieren von Software – Eine klassifizierende Übersicht der Verfahren, in: Liggesmeyer, Peter; Sneed, Harry M.; Spillner, Andreas (Hrsg.), *Testen, Analysieren und Verifizieren von Software*, Springer Verlag, Berlin, 1–25.

[Lig95] Liggesmeyer, Peter (1995): Ein Überblick über objektorientiertes Testen (Kurzfassung), *Softwaretechnik-Trends* **15** (4), 14–15.

[LRS95a] Li, Anlong; Ritchken, Peter; Sankarasubramanian, L. (1995): Lattice Models for Pricing American Interest Rate Claims, *Journal of Finance* **50** (2), 719–737.

[LRS95b] Li, Anlong; Ritchken, Peter; Sankarasubramanian, L. (1995): Lattice Works, *RISK* **8** (11), 65–69.

[LS92] Longstaff, Francis A.; Schwartz, Eduardo S. (1992): Interest Rate Volatility and the Term Structure: A Two Factor General Equilibrium Model, *Journal of Finance* **47** (4), 1259–1289.

[Mah98] Mahler, Gerhard (1998): Börsenprofis beäugen den Computerhandel sehr kritisch, *Computer Zeitung* vom 12.11.1998, S. 47.

[MCM+98] Murray, Leesa; Carrington, David; MacColl, Ian; McDonald, Jason (1998): *Formal Derivation of Finite State Machines for Class Testing*, Technical Report 98-03, School of Information Technology, University of Queensland; `ftp://svrc.it.uq.edu.au/pub/techreports/tr98-03.ps.Z`.

[MK94] McGregor, John D. ; Korson, Timothy D. (1994): Integrated Object-Oriented Testing and Development Process, *Communications of the ACM* **37** (9), 59–77.

[MPD+96] Mitchell, Ian; Parrington, Norman; Dunne, Peter; Moses, John (1996): Practical Prototyping, *Object Currents*, `http://www.sigs.com/publications/docs/oc/9605/oc9605.f.mitchell.html`.

[Ove93] Overbeck, Jan (1993): Testing Object-Oriented Software – State of the Art and Research Directions, *Proceedings First International Conference on Software Testing, Analysis & Review* (*EuroSTAR '93*), Oktober, London, 245–270.

[Ove96] Overbeck, Jan (1996): Objektorientiertes Testen und Wiederverwendbarkeit, in: Müllerburg, Monika; Spillner, Andreas; Liggesmeyer, Peter (Hrsg.), *Test, Analyse und Verifikation von Software*, GMD-Bericht Nr. 260, Oldenbourg Verlag, München, 1–24.

[Pel96] Pelkmann, Ute (1996): Testen von OO-Programmen aus der Sicht der Praxis, in: Müllerburg, Monika; Spillner, Andreas; Liggesmeyer, Peter (Hrsg.), *Test, Analyse und Verifikation von Software*, GMD-Bericht Nr. 260, Oldenbourg Verlag, München, 49–59.

[PS94] Pagel, Bernd-Uwe; Six, Hans-Werner (1994): *Software Engineering: Die Phasen der Softwareentwicklung*, Addison-Wesley, Bonn.

[RB80] Rendleman, Richard; Bartter, B. (1980): The Pricing of Options on Debt Securities, *Journal of Financial and Quantitative Analysis* **15** (3), 11–24.

[RBP+93] Rumbaugh, James; Blaha, Michael; Premerlani, William; Eddy, Frederick; Lorensen, William (1993): *Objektorientiertes Modellieren und Entwerfen*, Hanser, Wien, Prentice Hall, London.

[RH93a] Rothermel, Gregg; Harrold, Mary Jean (1993): Efficient Construction of Program Dependence Graphs, *Proceedings ACM International Symposium on Software Testing and Analysis* (*ISSTA '93*), Juni, Boston, 139–148.

[RH93b] Rothermel, Gregg; Harrold, Mary Jean (1993): A Safe, Efficient Algorithm for Regression Test Selection, *Proceedings Conference on Software Maintenance* (*ICSM '93*), September, Montreal, 358–367.

[RH94a] Rothermel, Gregg; Harrold, Mary Jean (1994): *A Comparison of Regression Test Selection Techniques*, Working Paper 94-111, Department of Computer Science, Clemson University.

[RH94b] Rothermel, Gregg; Harrold, Mary Jean (1994): Selecting Tests and Identifying Test Coverage Requirements for Modified Software, *ACM International Symposium on Software Testing and Analysis* (*ISSTA '94*), August, Seattle, 169–184.

[RH94c] Rothermel, Gregg; Harrold, Mary Jean (1994): Selecting Regression Tests for Object-Oriented Software, *Proceedings International Conference on Software Maintenance* (*ICSM '94*), September, Victoria, 14–25.

[RH97] Rothermel, Gregg; Harrold, Mary Jean (1997): A Safe, Efficient Regression Test Selection Technique, *ACM Transactions on Software Engineering and Methodology* **6** (2), 173–210.

[Rie97] Riedemann, Eike Hagen (1997): *Testmethoden für sequentielle und nebenläufige Software-Systeme*, Teubner, Stuttgart.

[Rot96] Rothermel, Gregg (1996): *Efficient, Effective Regression Testing using Safe Test Selection Techniques*, Dissertation, Clemson University; `http://www.cis.ohio-state.edu/~harrold/webpapers/diss-gregg.ps`.

[Roy70] Royce, Winston W. (1970): *Managing the Development of Large Software Systems: Concepts and Techniques*, Western Electronic Show and Convention Technical Papers, August, Los Angeles, 1–9; Reprinted: Proceedings Ninth International Conference on Software Engineering (ICSE '89), Mai, Pittsburgh, 328–338, 1989.

[RS95a] Ritchken, Peter; Sankarasubramanian, L. (1995): Volatility Structures of Forward Rates and the Dynamics of the Term Structure, *Mathematical Finance* **5** (1), 55–72.

[RS95b] Ritchken, Peter; Sankarasubramanian, L. (1995): Near Nirvana, *RISK* **8** (9), 109–111.

[Rüp97] Rüppel, Peter (1997): *Ein generisches Werkzeug für den objektorientierten Softwaretest*, Dissertation, Fachbereich Informatik, TU Berlin.

[RW97] Rosenblum, David S.; Weyuker, Elaine J. (1997): Using Coverage Information to Predict the Cost-Effectiveness of Regression Testing Strategies, *IEEE Transactions on Software Engineering* **23** (3), 146–156.

[SC95] Sane, Aamod; Campbell, Roy (1995): Object-Oriented State Machines – Subclassing, Composition, Delegation, and Genericity, *Proceedings Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '95)*, Oktober, Austin, 17–32.

[Sch98] Schneck, Ottmar (1998): *Lexikon der Betriebswirtschaft*, 3. Auflage, Deutscher Taschenbuch Verlag, München.

[Smi91] Smith, Michael F. (1991): *Software Prototyping: Adoption, Practice and Management*, McGraw-Hill Book Company, London.

[Sne95] Sneed, Harry M. (1995): Objektorientiertes Testen, *Informatik Spektrum* **18** (1), 6–12.

[Sne96] Sneed, Harry M. (1996): Ein objektorientiertes Testverfahren, in: Müllerburg, Monika; Spillner, Andreas; Liggesmeyer, Peter (Hrsg.), *Test, Analyse und Verifikation von Software*, GMD-Bericht Nr. 260, Oldenbourg Verlag, München, 25–48.

[Som95] Sommerville, Ian (1995): *Software Engineering*, Addison-Wesley, Harlow.

[TR93a] Turner, C.D.; Robson, D.J. (1993): *The Testing of Object-Oriented Programs*, Technical Report TR-13/92, Computer Science Division, University of Durham; `ftp://rievaulx.dur.ac.uk/pub/tech-reports/1992/13-92.ps.gz`.

[TR93b] Turner, C.D.; Robson, D.J. (1993): *State-Based Testing and Inheritance*, Technical Report TR-1/93, Computer Science Division, University of Durham; `ftp://rievaulx.dur.ac.uk/pub/tech-reports/1993/1-93.ps.gz`.

[TR93c] Turner, C.D.; Robson, D.J. (1993): *Guidance for the Testing of Object-Oriented Programs*, Technical Report TR-2/93, Computer Science Division, University of Durham; `ftp://rievaulx.dur.ac.uk/pub/tech-reports/1993/2-93.ps.gz`.

[Tsc96] Tschernig, Markus (1996): *Eine konsistente Bewertung von Zinsswaps durch Berücksichtigung der modernen Zinsstrukturtheorie*, Peter Lang GmbH, Frankfurt.

[TX95] Tse, T.H.; Xu, Zhinong (1995): *Class-Level Object-Oriented State Testing: A Formal Approach*, Technical Report TR-95-05, Department of Computer Science and Information Systems, University of Hong Kong; `http://www.csis.hku.hk/publications/techreps/document/TR-95-05.ps.gz`.

[TX96] Tse, T.H.; Xu, Zhinong (1996): Test Case Generation for Class-Level Object-Oriented Testing, *Proceedings 9th International Software Quality Week (QW '96)*, Mai, San Francisco, 4T4.0–4T4.12; `http://www.csis.hku.hk/~tse/Papers/xqwps.zip`.

[Vas77] Vasicek, Oldrich A. (1977): An Equilibrium Characterization of the Term Structure, *Journal of Financial Economics* **5**, 177–188.

[Vir98] Virtel, Martin (1998): Die mächtigsten Computer, *Die Zeit* 35, `http://www1.zeit.de/bda/int/zeit/print/199835.fuenf_rechner_re.html`.

[Weg93] Wegener, Ingo (1993): *Theoretische Informatik*, Teubner, Stuttgart.

[Welt97] Die Welt vom 26.07.1997, *Vier Computer-Pannen in sieben Monaten*, `http://www.welt.de/archiv/1997/07/26/0726wi12.htm`.

[Win97] Winter, Mario (1997): *Testbarkeit objekt-orientierter Programme*, GI-Fachgruppe Test, Analyse und Verifikation von Software, Arbeitskreis Testen objekt-orientierter Programme; `ftp://ftp.fernuni-hagen.de/pub/fachb/inf/pri3/papers/winter/OOTestbarkeit.ps.gz`.

[YK87] Yau, Stephen S.; Kishimoto, Zenichi (1987): A Method for Revalidating Modified Programs in the Maintenance Phase, *Proceedings International Computer Software & Applications Conference (COMPSAC '87)*, Oktober, Tokyo, 272–277.

[ZS96a] Zhang, John Q.; Sternbach, Efrem J. (1996): Financial Application Design Patterns, *Journal of Object-Oriented Programming* **8** (8), 6–13.

[ZS96b] Zhang, John Q.; Sternbach, Efrem J. (1996): Financial Application Patterns, *Journal of Object-Oriented Programming* **8** (9), 6–12.

[ZS96c] Zhang, John Q.; Sternbach, Efrem J. (1996): Financial Application Design Patterns, *Journal of Object-Oriented Programming* **9** (1), 6–15.