TECHNISCHE UNIVERSITÄT BERLIN

Fakultät IV Elektrotechnik und Informatik
Institut für Softwaretechnik und Theoretische Informatik
Fachgebiet Softwaretechnik

DIPLOMARBEIT

Entwicklung eines Testtreibers für Java-Klassen mit Hilfe von Java-Reflections.

eingereicht im:	April 2003
von:	Nghia Dang-Duc
Matrikelnummer:	187088
1. Gutachter:	Univ.–Prof.Dr. Jähnichen
2. Gutachter:	Dipl.–Inform. Dehla Sokenou

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass die vorliegende Arbeit selbstständig und eigenhändig unter ausschließlicher Zuhilfenahme der angegebenen Literatur erstellt wurde.

Berlin, den 29. April 2003 Nghia Dang Duc

”... it costs a lot more to test oo-software then to test ordinary software - perhaps four or five times as much ... Inheritance, dynamic binding, and polymorphism creates testing problems that might exact a testing cost so high that it obviates the advantages.”

(Beizer, B. Testing Technology - The growing gap. American Programmer, Vol. 7, No. 4, 1994)

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Heutzutage werden an jedes Programm erhebliche Qualitätsansprüche gestellt. Um diesen Anforderungen gerecht zu werden, muss vor allem Testen ein integraler Bestandteil der Softwareentwicklung sein. Gutes Testen birgt eine wesentliche Wertanhebung eines Programms in sich, da hierbei versucht wird, möglichst viele Fehler zu finden, diese zu beheben und somit die Software zuverlässiger, das heißt, robuster, sicherer und korrekter zu machen. Entscheidend beim Testen ist nicht zu zeigen, dass ein Programm funktioniert, sondern die Suche nach Fehlern. Unter Fehler wird dabei jede Abweichung des Verhaltens von dem in der Anforderungsdefinition festgelegten Verhalten verstanden. Ein weiteres Ziel beim Testen ist es, dem Management Informationen zu geben, damit es rationell das Risiko einer Verwendung eines Testobjekts evaluieren kann. Hierzu sollte ein testbares Programm geschaffen werden, das leicht validiert, falsifiziert und gewartet werden kann. Dafür müssen implementierbare, vollständige und konsistente explizite und implizite Anforderungen (Requirements) entwickelt werden [Bei1990].

Explizite Anforderungen werden durch so genannte *clean tests* (konstruktive/positive Tätigkeit) validiert. Ein *clean test* ist ein Test, dessen primärer Zweck Validation ist. Das bedeutet, Tests werden entworfen, um zu zeigen, dass die Software korrekt arbeitet. Implizite Anforderungen werden durch *dirty tests* (subversive/negative Tätigkeit) falsifiziert. Das heißt, ein Test, dessen primärer Zweck Falsifikation ist, wird entworfen, um Fehler in der Software zu erzeugen. Wird etwas nicht getestet, bedeutet das, dass die Fehlerfreiheit garantiert wird, oder dass sicher gestellt ist, dass die Funktionalität unnötig ist. Deshalb muss die nicht getestete Funktionalität entfernt werden. Es ist meistens immer noch üblich, das Testen an das Ende des Entwicklungsprozesses zu legen. Häufig erfolgt dies durch Testpersonen oder es werden per Hand von Test-Designern Testfälle entwickelt. Möglich und besser ist es, das Testen in verschiedenen Formen direkt in den Entwicklungsprozess zu integrieren. Hierbei werden jedoch ebenfalls die oben beschriebenen manuellen Testvorgänge eingesetzt. Dieses Testen hat aber einige entscheidende Nachteile: Da es von Menschen durchgeführt wird, ist es äußerst zeitaufwendig und somit

teuer, vor allem anbetracht dessen, dass Testen heutzutage bis zu 50% des Entwicklungsaufwandes ausmacht; bei der Komplexität heutiger Systeme sogar manchmal auch mehr. Ein weiteres Manko ist, dass der Erfolg von solchen Softwaretests oft aufgrund der komplexen psychologischen und menschlichen Faktoren nicht sichergestellt ist. Des Weiteren existieren selbst für sehr einfache Programme unendlich viele Testfälle, die normalerweise nur zu einem geringen Prozentsatz durchgetestet werden können. Um diese Probleme der heutigen Softwareentwicklung zu beheben, wäre es sinnvoll, die Tests der Software so weit wie möglich zu automatisieren. Ein Ansatz dafür ist das modellbasierte Testen beziehungsweise die modellbasierte Software-Entwicklung.

1.2 Ziele

Das Ziel dieser Arbeit ist es, einen generischen Testtreiber zu entwickeln, der Testfälle an das Testobjekt sendet, um die Testklassen auf ihre Korrektheit und ihr Verhalten im Fehlerfall zu überprüfen. Er soll geschriebenen Code ohne zusätzlichen Programmieraufwand ausführen können. Der Entwickler kann auf Grund der vorliegenden Spezifikationen über das Ergebnis entscheiden oder programmbasiert durch Betrachtung des Zustandes des Objektes befinden. Insbesondere ist die Umsetzung von OCL-Ausdrücken mit Hilfe des Testtreibers zu realisieren. Dabei sollen folgende Teilziele erreicht werden:

- Analyse der Strategien objektorientierter Systeme im Hinblick auf die Testmethodik für Klassen
- Betrachtung einer Reihe bereits existierender Testtreiber (JUnit, BlueJ)
- Untersuchung der Möglichkeiten zur Nutzung von Java-Reflections in Hinblick auf den Test
- Erforschung der Nutzbarkeit der Pre- und Postconditions von OCL für den zu entwickelnden Testtreiber
- Entwicklung eines Prototypen des Testtreibers mittels Java

1.3 Aufbau

Dieser Abschnitt gibt einen kurzen Überblick über den Aufbau der Arbeit. Zunächst werden im zweiten Kapitel die Grundlagen des Testens beschrieben sowie einige für diese Arbeit grundlegende Begriffe definiert und abgegrenzt. Insbesondere wird in dessem Verlauf auf die Besonderheit der Objektorientierung und Testautomatisierung eingegangen. Eine genaue Anforderungsanalyse für den Testtreiber wird im Kapitel 3 erstellt. Die Anwendbarkeit der vorhandenen Techniken wird im vierten Kapitel untersucht. In Kapitel 5 geht es darum, die Designspezifikation und Architektur des Testtreibers zu erläutern. Da im Rahmen dieser Arbeit ein Prototyp entwickelt werden wird, mit dessen Hilfe es möglich ist, aus OCL-Ausdrücken automatisierte Tests für Java-Klassen zu generieren, wird in diesem Kapitel ebenso beschrieben, welche Probleme bei der Implementierung eines solchen Tools auftreten und welche speziellen Anforderungen an die zugrundeliegenden OCL-Ausdrücke gestellt werden. Anschließend folgen ein Resümee und das Literaturverzeichnis. Der Anhang enthält neben einem ausführlichen Glossar die vollständige Dokumentation des Prototyps des Testtreibers.

2 Grundlagen

2.1 Gesetze und Prinzipien für das Testen von Software

Das Ziel jeglichen Testens ist es, möglichst alle im Testobjekt vorhandenen Fehler zu entdecken und zu lokalisieren. Durch Testen können lediglich existierende Fehler entdeckt werden. Die Abwesenheit von Fehlern kann jedoch nicht gefunden werden. Daher zielt das Testen darauf ab, möglichst viele Fehler zu entdecken, um der Fehlerzahl Null möglichst nahezukommen. Dieses Ziel bedeutet, dass ein Test erst erfolgreich ist, wenn Fehler gefunden werden und nicht, wenn keine Fehler gefunden werden. Bei der Entwicklung von Testtreibern müssen zwei Gesetze des Testens beachtet werden:

Das Pestizid-Paradoxon

Jede Methode, die angewendet wird, um Fehler zu vermeiden oder zu finden, hinterlässt einen gewissen Rest von raffinierteren Fehlern, gegen welche die eingesetzte Methode nichts ausrichten kann [Bei1990].

Die Komplexitätsschranke

Die Komplexität der Software (und damit auch die Komplexität der Fehler) wächst über die menschliche Fähigkeit hinaus, diese Komplexität zu meistern [Bei 1990].

Während „Testen" das Finden von Fehlern bedeutet, werden beim Debugging Fehlerursachen diagnostiziert und Fehler behoben. Deshalb stehen Testen und Debugging in enger Beziehung und ergänzen sich gegenseitig. Folgendermaßen können sie unterschieden werden: Beim Testen wird unter bekannten Voraussetzungen begonnen. Es werden vordefinierte Prozeduren verwendet und vorhersagbare Ergebnisse erzielt. Deshalb kann Testen geplant werden, das heißt, Testpläne können entworfen und zeitlich angesetzt werden. Es ist nur nicht vorhersagbar, ob das Programm den Test besteht oder nicht – durch das Testen werden erst die Fehler des Programmierers aufgezeigt. Tests können ohne großes Wissen über das Design erledigt und oft von Außenstehenden durchgeführt werden. Vieles lässt sich automatisieren wie der Entwurf von Testfällen und die Durchführung der Tests. Beim Debuggen hingegen können die Vorgehensweise und die dafür benötigte Zeit nicht von

vornherein festgelegt werden. Ohne detaillierte Design-Kenntnisse ist Debuggen unmöglich und muss daher vom Entwickler selbst durchgeführt werden.

Die Entwicklung eines automatisierten Testtreibers, setzt die Kenntnis der allgemeinen Testprinzipien, die für alle Testarten gelten, voraus. Zunächst müssen die erwarteten Werte oder das Resultat definiert werden, da die Möglichkeit besteht, dass ein plausibles aber fehlerhaftes Ergebnis als korrekt betrachtet wird, wenn keine genaue Untersuchung des ganzen Outputs und ein exakter Vergleich mit den erwarteten Werten erfolgt. Diese Testergebnisse müssen gründlich überprüft werden. Oft werden Fehler übersehen, die bei sorgfältiger Prüfung hätten entdeckt werden können. Ebenso wichtig ist die Überprüfung des Programms auf unerwünschte Nebeneffekte. Es müssen Testfälle nicht nur für gültige und erwartete Eingabedaten erstellt werden sondern auch für ungültige und unerwartete. Häufig wird sich lediglich auf zulässige Eingaben konzentriert, so dass manchmal ungültige Daten erst entdeckt werden, wenn das Programm auf unerwartete Weise eingesetzt wird. Aus diesem Grund dürfen Testfälle nicht unter der Annahme geplant werden, dass keine Fehler gefunden werden. Es ist anzunehmen, dass die Wahrscheinlichkeit für die Existenz weiterer Fehler in einem Abschnitt eines Programms proportional zu der Zahl der bereits entdeckten Fehler in diesem Abschnitt ist. Des Weiteren sollte das Erstellen von Wegwerftestfällen vermieden werden, da Testfälle wiederholbar sein sollten und wiederholt gewinnbringend eingesetzt werden können [Mye 1989].

2.2 Testplanung und Testautomatisierung

Will ein Entwickler Klassen testen, muss er ein Objekt dieser Klasse kreieren, um seine Methoden auszuführen. Er muss ein Testprogramm schreiben, um ein Objekt der zu testenden Klasse zu erzeugen. Dieses Testprogramm testet die Klasse, indem es alle ihre Methoden aufruft. Es muss die Ergebnisse der Methodenaufrufe für den Entwickler darstellen können. Darum stellt das Testprogramm möglicherweise mehr Entwicklungsaufwand für den Entwickler dar als die zu testende Klasse. Ein weiterer Nachteil ist, dass das Testprogramm selbst Fehler enthalten kann. Zudem bleibt der interne Zustand des zu testenden Objektes oft verborgen. Darum muss in der zu testenden Klasse Code eingebracht werden, der in einem Debugmodus Ausgaben über den internen Zustand macht. Dieser zusätzliche Code kann wiederum Fehler haben. Außerdem wird es bei einer langen Liste von Ausgaben

schwierig, diese noch zu interpretieren. Um die oben genannten Nachteile zu übergehen, sollte idealerweise kein Testcode geschrieben werden. Die Aufgabe eines Testprogramms sollte die Entwicklungsumgebung einer Programmiersprache übernehmen. Die Entwicklungsumgebung muss dabei folgende Punkte unterstützen:

- Eine Instanz der zu testenden Klasse erstellen und geeignete Parameter an den Konstruktor liefern
- Methoden der zu testenden Klasse aufrufen und die Ergebnisse anzeigen
- Den internen Zustand der Instanz anzeigen

Eine Automatisierung ist für zuverlässiges Testen unabdingbar, da manuelles Testen zeitaufwendiger ist und über eine höhere Fehlerquote durch Tippfehler oder visuelles Ablesen verfügt. Durch die häufige Wiederholung von Testfällen, insbesondere, da die Weiterentwicklung und Verbesserung bestehender Software einen höheren Stellenwert hat als die Entwicklung neuer Programme, sparen automatisierte Testfälle viel Zeit. Beim manuellen Testen sind Regressionstests nahezu unmöglich.

Es werden zwei Arten von Testtreibern unterschieden: Einwegtestprogramme und allgemeingültige Testtreiber. *Einwegtestprogramme* haben den Nachteil, dass sie mehr Zeitaufwand benötigen und nicht so universell einsetzbar sind wie allgemeingültige Testtreiber. Dafür können mit ihnen gezielt programmspezifische Probleme getestet werden, da sie auf das jeweilige Programm individuell zugeschnitten sind. Meist werden sie direkt in das Programm implementiert, was allerdings den Code schwerer wartbar macht. *Allgemeingültige Testtreiber* führen Tests automatisch aus.

Auf eine detaillierte Testplanung kann bei der Testautomatisierung nicht verzichtet werden. Diese Planung ist nötig, um eine gute Testabdeckung zu erreichen, denn das Ziel ist es, möglichst alle Funktionen, die die Software zur Verfügung stellen soll, durchzuspielen. Ohne ausreichende Planung kann sich leicht in Details verloren werden.

Zunächst wird festgelegt, was getestet werden soll. Üblicherweise wird der gesamte Funktionsumfang der Anwendung getestet, um eine Softwarequalität zu gewährleisten, die sowohl Entwickler als auch Kunden zufrieden stellt. Dazu werden für jeden Test die erwarteten Resultate festgelegt. Ferner werden Vorbedingungen defi-

niert, die für die Durchführung von Interaktionen mit der Software gelten müssen, um ein Feature anzuwenden. Am Ende einer solchen Testplanung steht ein Repertoire an Testfällen zur Verfügung, die manuell oder automatisiert durchgeführt werden können.

In [Mar2000] werden zwei Schritte angegeben, die bei der Automatisierung von Tests besondere Bedeutung haben. Zuerst muss entschieden werden, welche Tests eine Automatisierung lohnen, da diese zunächst aufwendiger zu schreiben sind als manuelle Tests. Es muss genau abgewägt werden, welche manuelle Tests durch den automatisierten Test ersetzt werden können, vor allem im Hinblick darauf, dass durch den zusätzlichen Zeitaufwand für die Automatisierung von Tests weniger Zeit für die Durchführung manueller Tests verbleibt. Dadurch ist es wiederum möglich, dass Fehler, die durch diese unterlassenen Tests gefunden worden wären, übersehen werden. Die Auswirkungen der möglicherweise unentdeckten Fehler müssen kalkulierbar bleiben. Es gibt Tests, die überhaupt nicht mit vertretbarem Aufwand automatisierbar sind. Bei der Entscheidung, ob die Automatisierung eines Tests in einem angemessenen Verhältnis zum gewünschten Ergebnis steht, gibt es einige Anhaltspunkte. Es muss berechnet werden, ob und wie viel Mehrkosten bei einer Automatisierung und anschließendem einmaligem Testdurchlauf entstehen würden im Vergleich zu einem manuellen einmaligen Testen. Wie oft könnte dieser Testtreiber anschließend noch eingesetzt werden und wodurch könnte er hinfällig werden? Welche weiteren Bugs könnte dieser Testtreiber zusätzlich zu denen, die er während des ersten Einsatzes gefunden hat, noch aufdecken? Häufig weiß der Entwickler vor der Entscheidung über die Automatisierung noch nicht genau, wie viel dieser zusätzliche Aufwand kosten wird. Dies gilt insbesondere dann, wenn ein neues Werkzeug zur Erstellung der Tests eingesetzt wird, oder erst eine Infrastruktur für die Erstellung der Tests aufgebaut werden muss. Wie in vielen Bereichen der Softwareentwicklung muss der Entwickler sich auch hier auf sein Gefühl und auf seine Erfahrung verlassen. Es darf nicht vergessen werden, dass bei späteren Änderungen am System Zeit zurückgewonnen wird, da die automatisch ablaufenden Tests Zeit sparen; besonders, wenn davon ausgegangen wird, dass die erstellte Software noch häufig verändert und angepasst werden muss.

Der zweite Schritt, der laut [Mar2000] bei der Automatisierung von Tests besondere Bedeutung hat, ist, einen Ansatz zu finden, der den Test so gut wie möglich von der Anwendung isoliert. Dies stellt sicher, dass der Test möglichst lange funktioniert und nicht bei kleinen Änderungen in der Anwendung bereits unbrauchbar wird. Um das zu erreichen, wird die Anwendung über eine möglichst stabile Schnittstelle gesteuert. Häufig ist es sinnvoll, nur zu diesem Zweck eine Schnittstelle in die Anwendung einzubauen, die eine Steuerung der Software von außen zulässt.

In [Kan1997] werden zwei Ansätze der Automatisierung von Systemtests beschrieben, die sich in Projekten als Erfolg versprechend herausgestellt haben: Zum einen das datengesteuerte Testdesign (data-driven test design) und andererseits das framework-basierte Testdesign (framework-based test design).

Beim *framework-basierten Testdesign* wird für jede Benutzeraktion, die an der Anwendung getätigt werden kann, eine Funktion programmiert, die genau diese Aktion durchführt. Alle Testfälle greifen nun auf diese Funktionen zurück, um die vom Testfall vorgegebenen Aktionen in gewünschter Reihenfolge auszulösen. Bei diesem Ansatz besteht der Vorteil darin, dass, wenn sich das Auslösen einer Aktion in der Anwendung verändert, lediglich die zugehörige Funktion angepasst werden muss, nicht jedoch die darauf basierenden Tests.

Beim *datengesteuerten Testdesign* werden Tabellen erstellt, die in jeder Zeile eine Kombination von Benutzereingaben und daraus resultierenden zu erwartenden Ergebnissen zusammenfassen. Jede einzelne Zeile repräsentiert einen Testfall. Um die Tests automatisch ablaufen lassen zu können, wird ein Testtreiberprogramm benötigt, das die erstellten Tabellen einlesen und interpretieren kann. Der Testtreiber steuert die zu testende Anwendung entsprechend der Eintragungen in den Tabellen. Der Vorteil dieses Ansatzes besteht darin, dass die Tabellen leicht zu erstellen sind und somit mit geringem Zeitaufwand viele Tests entstehen können. Außerdem sind zum Erstellen der Tabellen keine Programmierkenntnisse erforderlich, jedoch muss im Gegenzug ein relativ aufwendiger Testtreiber programmiert werden.

2.2.1 Vorteile

Automatisierte Tests laufen, wenn sie erst einmal erstellt wurden, schneller ab als manuelle Test, sind konsistenter und können ohne großen Mehraufwand beliebig oft wiederholt werden. Automatisierte Tests - das Abspielen eines oder mehrerer Test-

skripts - können unbeaufsichtigt laufen. Dies ermöglicht die Durchführung von Tests nachts oder am Wochenende, was eine effektivere Nutzung der Zeit wochentags erlaubt, um die erkannten Fehler zu analysieren und die Tests zu vervollständigen. Die Ergebnisse der Tests werden automatisch und detailliert dokumentiert und in der Regel in einer Fehlerdatenbank abgelegt. Dies erleichtert die Fehleranalyse und die Fehlerkorrektur

2.2.2 Nachteile

Ein unbeaufsichtigtes Abspielen von Tests setzt voraus, dass unvorhergesehene Ereignisse tolerant verarbeitet werden, was im Idealfall bedeutet, dass Fehler dokumentiert und übergangen werden.

Beim automatisierten Testen treten zunächst zusätzliche Kosten auf. Das sind Anschaffungs-, Entwicklungs- und Instandhaltungskosten des Testtools. Außerdem wird hochqualifiziertes Personal für den Support des Testtools benötigt.

Das Aufzeichnen von Benutzeraktivitäten ist erst zu einem sehr späten Zeitpunkt in der Entwicklung einer Software möglich, wenn die meiste Funktionalität bereits implementiert ist. An diesem Punkt sind Änderungen schwierig und teuer. Um früh mit dem automatisierten Testen beginnen zu können, müssen Skripts von Entwicklern per Hand geschrieben werden. Automatisiertes Testen ist erst sinnvoll, wenn Tests mehrfach wiederholt werden. Die meisten Fehler werden in der Regel nicht beim Abspielen von Testskripts gefunden, sondern beim Entwickeln oder Aufzeichnen von Testskripts. Insofern ist automatisiertes Testen nicht immer notwendig und angebracht und kann manuelles Testen nicht ersetzen.

2.3 Testarten und Testphasen

2.3.1 Testarten

Um auf den Hauptansatz des Testtreibers eingehen zu können, ist es notwendig, an dieser Stelle die bekannten Testarten zu untersuchen und darzulegen, welche Ansätze für den Testtreiber relevant sind. Da ein Testobjekt je nach Betrachtungsweise unterschiedlich untersucht werden kann, wird die Systematik der Testmethoden in folgende Testarten eingeteilt: nach der Art der Testfallermittlung, Testausführung und dem Umfang der Testausführung.

Die Testfallermittlung kann auf zwei unterschiedliche Arten erfolgen: aufgabenorientiert (funktionsorientiert) oder produktorientiert (strukturorientiert). Je nach Anwendungsfall wird zwischen Black-Box-Test und White-Box-Test unterschieden. Die Testausführung kann in statische und dynamische Tests unterteilt werden. Der Umfang der Testausführung kann durch die Art und die Anzahl der verwendeten Testfälle festgelegt werden. Danach wird zwischen dem repräsentativen Test, dem schwachstellenorientierten Test und dem schadensausmaßorientierten Test unterschieden [Bal1998].

Black-Box-Testen

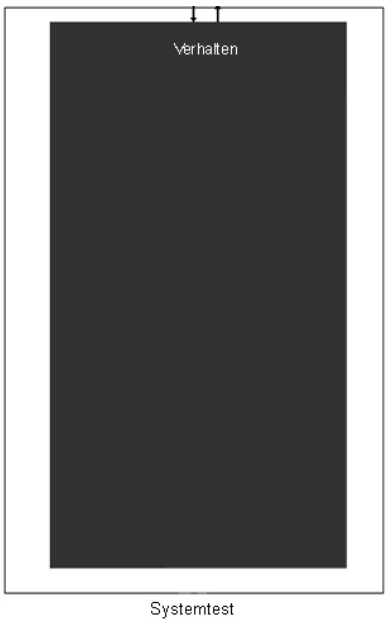


Abb. 2.1: Black-Box-Testen

Das Black-Box-Testen wird auch als funktionaler Test bezeichnet. Hauptmerkmal des Black-Box-Testens ist die Prüfung des Ein- und Ausgabeverhaltens ohne Berücksichtigung der inneren Struktur des Testobjektes (Schnittstellentest). Das bedeutet, dass der Tester nicht an dem internen Verhalten und an der Struktur des Programms interessiert ist, sondern daran, Umstände zu entdecken, bei denen sich das Programm nicht gemäß der Spezifikation verhält. Er betrachtet das Programm somit als *Blackbox* (siehe Abbildung 2.1). Ohne Kenntnisse der internen Struktur des Programms zu verwenden, werden die Testdaten lediglich aus der Spezifikation abgeleitet. Wichtig dabei ist, das Verhalten des Testobjekts bei fehlerhaften Eingabedaten zu überprüfen. Black-Box-Testen wird hauptsächlich für die Ausführung mit Testdaten innerhalb des dynamischen Testens verwendet. Es wird aber auch bei Verfügbarkeit entsprechender Hilfsmittel unter anderem zum Prototyping für Dokumente eingesetzt. Durch das Black-Box-Testen kann das Fehlen von bestimmten Aufgaben und Funktionen beziehungsweise Eigenschaften des Testobjekts erkannt werden. Dagegen können durch das Black-Box-Testen keine Informationen darüber geliefert werden, ob alle Funktionen des Testobjekts auch tatsächlich benötigt werden oder ob Datenobjekte manipuliert werden, die keinen Einfluss auf das Ein- und Ausgabeverhalten des Testobjekts haben. Diese Hinweise auf Design- und Implementierungsfehler liefert nur der White-Box-Test [Bei1995].

White-Box-Testen

Dieses Testverfahren wird auch Strukturtest genannt. Wesentliches Merkmal des White-Box-Testens ist die Prüfung des Ein- und Ausgabeverhaltens unter Betrachtung der inneren Strukturen (Schnittstellen- und Strukturtest). Dies wird häufig als Kontrollflussgraph dargestellt (siehe Abbildung 2.2).

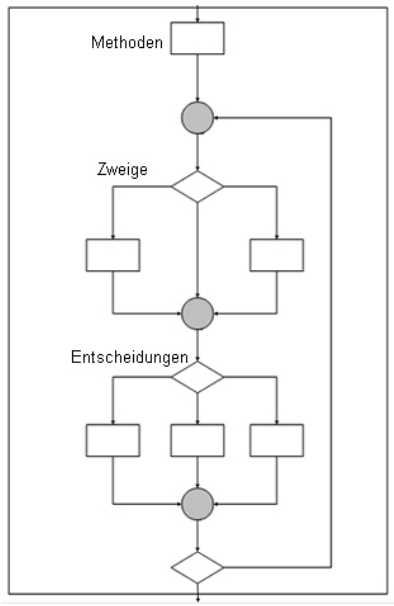


Abb. 2.2: White-Box-Testen

Der Tester definiert somit die Testdaten mit Kenntnis der Programmlogik. Das Ziel ist, für jeden möglichen Pfad durch das Testobjekt das Verhalten des Testobjekts in Abhängigkeit von den Eingabedaten festzustellen. Bei der Wahl der Eingabedaten ist zu berücksichtigen, dass jedes Modul und jede Funktion des Testobjektes mindestens einmal aufgerufen, jeder Zweig mindestens einmal absolviert wird und möglichst viele Pfade durchlaufen werden. Es ist praktisch unmöglich, alle möglichen Anweisungsfolgen für gut strukturierte Programme auszutesten. Auch bei einer automatischen Testdatengenerierung ist es unmöglich, alle Pfade zu testen, da die Anzahl der dazu benötigten Testläufe zu hoch ist. Trotzdem ist der White-Box-Test eine der wichtigsten Testmethoden, weil damit für eine begrenzte Anzahl von Programmpfaden die richtige Manipulation der Datenstrukturen und das Ein/Ausgabeverhalten von Testobjekten überprüft werden kann. White-Box-Testen wird für das statische aber auch für das dynamische Testen eingesetzt. Ein Vorteil dieses Testverfahrens ist, dass das Vorhandensein nicht geforderter oder gewünschter Aufgaben und Funktionen beziehungsweise Eigenschaften des Testobjekts erkannt werden kann. Dafür kann das Fehlen erforderlicher Aufgaben und Funktionen nicht festgestellt werden. Da diese Fehler durch die Bereitstellung der Soll-Ergebnisse und entsprechende Ergebnisprüfung auch gefunden werden können, ist diese Aussage aber nicht generell gültig.

Weder allein durch das Black-Box-Testen noch durch ausschließliches White-Box-Testen ist eine vollständige Abdeckung der Testaufgaben gegeben, erst durch eine Kombination beider Testarten können optimale Ergebnisse erzielt werden [Bal2000].

Statisches Testen

Nach [Ehr1985] ergeben statistische Tests Wahrscheinlichkeitsaussagen über erwartete Restfehlerraten, Zuverlässigkeit und Risiko des Testobjekts. Damit dient das statische Testen zur Analyse von Software-Produkten, wobei ausschließlich das Testobjekt selbst als Datenbestand analysiert und ausgewertet wird. Es erfolgt keine Ausführung des Testobjekts mit Testdaten. Das statische Testen bezieht sich sowohl auf Dokumente als auch auf Programme. So werden vor allem Dokumente zum großen Teil durch statisches Testen getestet, teilweise können aber Dokumente dynamisch getestet werden. Bei Programmen ist das statische Testen der erste Schritt nach der Codierung und kann unterschiedlichen Umfang hinsichtlich der durchzuführenden Analysen haben. Beim statischen Test wird das Testobjekt hauptsächlich syntaktisch, strukturell oder semantisch analysiert. Das Ziel dabei ist, zu einem möglichst frühen Zeitpunkt fehlerverdächtige Stellen des Testobjekts zu lokalisieren. Die wichtigsten Aktivitäten der statischen Analyse sind die Code-Inspektion, die Komplexitätsanalyse, die Strukturanalyse und die Datenflussanalyse.

Bei der *Code-Inspektion* wird durch ein Team von Softwaretechnikern versucht, beim gemeinsamen Lesen des Codes Entwurfs und Implementierungsfehler zu finden, diese aber nicht selbst auszubessern. Normalerweise nehmen mindestens vier Personen an einer Code-Inspektion teil. Einer von ihnen ist der Moderator, der ein kompetenter, aber nicht am Projekt beteiligter Softwaretechniker sein sollte, dessen Pflichten vor allem die Protokollierung aller gefundenen Fehler und die Zeitplanung für die Inspektionssitzungen beinhalten. Weiterhin nehmen an der Inspektion der Designer des Testobjekts, der für die Implementierung zuständige Programmierer und der für den Test zuständige Mitarbeiter teil. Erst wenn die Inspektion vollständig abgeschlossen ist, erfolgt die Beauftragung der Korrektur durch den Moderator.

Das Ziel der *Komplexitätsanalyse* ist die Ermittlung von Messzahlen für die Komplexität eines Programms. Darunter werden Komplexitätsmaße von Modulen, Schachtelungstiefen von Schleifen und Längen von Prozeduren und Methoden verstanden. Diese Ergebnisse gestatten es, in gewissen Grenzen Aussagen über die Qualität eines Softwareproduktes zu machen und fehlerträchtige Stellen im Programmsystem zu lokalisieren.

Bei der *Strukturanalyse* wird versucht, Strukturanomalien eines Testobjektes aufzudecken. Sie soll unter anderem Auskunft darüber geben, ob alle Anweisungen eines Programms vom Programmanfang aus erreicht werden können, ob das Pro-

grammende von allen Anweisungen aus erreichbar ist, ob in einem Programm Schleifen mit Einsprüngen enthalten sind oder ob der Kontrollfluss unerlaubte Strukturen enthält.

Bei der *Datenflussanalyse* werden Datenflussanomalien aufgedeckt. Sie liefert Informationen darüber, ob ein Datenobjekt vor seiner Benutzung einen Wert erhalten hat und ob ein Datenobjekt nach einer Wertzuweisung auch benutzt wird. Die Datenflussanalyse wird sowohl für den Rumpf eines Testobjekts als auch für die Schnittstellen zwischen Testobjekten durchgeführt.

Dynamisches Testen

Das dynamische Testen dient der Analyse von Software-Produkten durch Ausführung mit Testdaten. Ziel hierbei ist es, durch gezieltes Erzeugen und Auswerten einer Datenbasis und deren Vergleich mit Soll-Ergebnissen Ablauffehler zu erkennen [Pom1996]. Dieses Testen bezieht sich nicht nur auf Programme sondern auch auf Dokumente, da das gedankliche Durchrechnen eines Programms oder das Verfolgen eines Ablaufplans mit Testdaten ebenso dazu dient, Ergebnisse zu erzielen und diese einem Soll-/Ist-Vergleich zu unterziehen. Die Tätigkeiten beim dynamischen Test umfassen die Vorbereitung des Testobjekts zur Fehlerlokalisierung, die Bereitstellung einer Testumgebung, die Auswahl geeigneter Testfälle und -daten, und schlussendlich die Testausführung und -auswertung. Auf das dynamische Testen kann im Software-Lebenszyklus nicht verzichtet werden. Jede Prozedur, jedes Modul und jede Klasse, jedes Subsystem und das Gesamtsystem müssen dynamisch getestet werden, auch wenn statische Tests oder Programmverifikationen durchgeführt wurden. Weder allein das statische noch das dynamische Testen sind ausreichend, um alle möglichen Fehler zu erkennen. Beide Testarten sind im unterschiedlichen Maße geeignet, bestimmte Fehler beziehungsweise Fehlerklassen aufzuspüren [Tha2000].

Die zur Anwendung kommende Vorgehensweise beim Testen ist sowohl von der Auswahl der einzubeziehenden Daten als auch von den zu analysierenden Komponenten des Testobjekts abhängig. So wird der Umfang der Testausführung dieser Testarten entweder direkt durch Selektion entsprechender physischer Komponenten oder indirekt durch Auswahl entsprechender Testdaten festgelegt.

Schwachstellenorientiertes Testen

Das schwachstellenorientierte Testen beschreibt die Vorgehensweise, bei der Art und Umfang der Testfälle so ausgewählt werden, dass die Komponenten des Testobjekts entsprechend der auf Grund von Erfahrung und Intuition vermuteten Fehlerhäufigkeit in den Komponenten dynamisch getestet werden, um die Annahmen zu verifizieren oder falsifizieren.

Repräsentatives Testen

Als repräsentativer Test wird ein Test bezeichnet, wenn Art und Umfang der Testfälle so ausgewählt werden, dass die Komponenten des Testobjekts entsprechend der Häufigkeit ihrer Verwendung getestet werden. Das bedeutet, dass für die Testausführung, vor allem beim Testen von Dokumenten, genau festgelegte Teile des Testobjekts einbezogen werden und so bestimmte Aufgaben und Funktionen repräsentiert werden. Wichtig ist hierbei das Vorhandensein detaillierter Erwartungen hinsichtlich der Ergebnisse der Testausführung.

Schadensausmaßorientiertes Testen

Ein Test wird als schadensausmaßorientierter Test bezeichnet, wenn die Komponenten des Testobjekts, bei denen durch das Auftreten von Fehlern ein besonders hoher Schaden verursacht werden kann, intensiver getestet werden als die anderen Komponenten. Vorab muss eine Risikoanalyse zur Einstufung der Testobjekt-Komponenten in Schadensklassen durchgeführt werden.

Repräsentatives, statistisches und schwachstellenorientiertes Testen sind sowohl beim Black-Box- beziehungsweise White-Box-Testen als auch beim statischen und dynamischen Testen sowie für alle Testaufgaben anwendbar. Da bei dem zu entwickelnden Testtreiber lediglich kompilierte Javaklassen und kein Quellcode zur Verfügung stehen, ist bei der Testfallermittlung das Black-Box-Testverfahren die geeignetere Vorgehensweise. Der Testtreiber soll nur durch Testdaten mittels OCL-Ausdrücken automatisiert werden, deshalb muss bei der Entwicklung des Testtreibers die Konzentration auf dem schwachstellenorientierten Testen liegen, welches auf das Dynamische Testen angewendet wird. Auf die Anwendung von OCL

(Object Constraint Language) innerhalb des Testtreibers wird in Kapitel 4.1 näher eingegangen werden.

2.3.2 Testphasen

Eine Definition von Testphasen findet sich in [Sch1982]: ”Testphasen sind sachlich und zeitlich in sich geschlossene Abschnitte des Testens, in denen für bestimmte Software-Produkte definierte Testaufgaben durchgeführt werden.”

In einer Testphase müssen alle sich unterscheidenden Objekte beziehungsweise Objekte der unterschiedlichen Typen einzeln getestet werden. Es gibt aber auch sehr komplexe Objekte, die nicht gänzlich durchschaubar sind. Daher ist es nicht möglich, diese innerhalb eines Tests vollständig zu testen, sondern einzelne Elemente des Objektes müssen in jeweils eigenen Testphasen getestet werden. Außerdem ist es wichtig, Testphasen für Objekte zu unterscheiden, wenn sie verschiedene Verantwortlichkeiten besitzen.

Die Testphasen lassen sich in den Klassentest, den Integrationstest, den Test der Systeme, den Test des installierten Systems, auch Abnahmetest genannt, den Betriebstest und den Wartungstest untergliedern.

Während des *Klassentests* (Komponententest) werden die einzelnen Methoden und Klassen auf ihre Spezifikation hin getestet. Grundlage dafür sind die Dokumente des Software-Feinentwurfs, in welchen die Aufgaben der Methoden und Klasse genau formuliert sind, so dass aus diesen die Testfälle abgeleitet werden können. Ziel dieser Testphase ist die Prüfung der Klasse in Bezug auf die Erfüllung der ihr gestellten Aufgaben im Gesamtsystem. Besonders sollte jedoch beachtet werden, dass einzelne Methoden oder Klassen in Beziehung zu anderen Methoden und Klasse stehen können, die noch nicht fertig implementiert sind oder für diesen einzelnen Test nicht relevant sind. Diese fehlenden Teile müssen durch entsprechende Tools beim Testen simuliert werden. Dafür erfolgt dieser Test unabhängig von den anderen Systemkomponenten. Von Bedeutung für die Entwicklung von Testfällen ist unter anderem die Sichtweise bezüglich des Codes. Entweder wird nur die Spezifikation zur Ableitung der Testfälle herangezogen, das heißt ein Black-Box-Test (auch funktionaler Test) durchgeführt oder die Struktur des Programms wird ebenfalls in die Auswahl der Testfälle mit einbezogen und ein so genannter White-Box-Test (auch struktureller Test) findet statt [Per1999].

Im darauffolgenden *Integrationstest* wird geprüft, ob die Komponenten, wie von der Systemarchitektur gefordert, zusammenspielen. Die Testgrundlage ist hierbei die Design-Spezifikation. Im Hinblick auf das System ist dies ein White-Box-Test, im Hinblick auf einzelne Module jedoch ein Black-Box-Test. Testschwerpunkte des Integrationstests sind die externen und internen Schnittstellen sowie der Leistungs- und der Stresstest. Für Integrationstests gibt es ähnlich dem Klassentest zwei entgegengesetzte Ansätze:

Zum einem existiert der *Bottom-Up-Integration-Ansatz.* Hierbei werden zunächst diejenigen Komponenten getestet, die keine anderen Komponenten aufrufen, das heißt die Komponenten der untersten Ebene der Benutzerhierarchie. Danach erfolgen Testläufe mit Komponenten, die wiederum eine Ebene höher liegen. Dies wird so lange fortgeführt, bis alle Komponenten getestet worden sind. Ein Nachteil dieses Testansatzes ist, dass die Komponenten der oberen Ebenen erst zum Schluss getestet werden. Außerdem können Fehler in Komponenten der oberen Ebenen auf Entwurfsfehler hindeuten, die eigentlich schnell korrigiert werden sollten. Die Aufdeckung von Zeitfehlern erfolgt durch die Bottom-Up-Integration häufig erst spät. Andererseits ist die Bottom-Up-Integration oft für objektorientierte Systeme das Richtige, da nur so geprüft werden kann, ob ein Objekt korrekt auf Nachrichten reagiert.

Der zweite Testansatz ist der *Top-Down-Integration-Test*. Im Prinzip wird bei diesem Test genau umgekehrt vorgegangen wie bei der Bottom-Up-Integration. Da eine zu testende Komponente normalerweise noch nicht verfügbare Komponenten der unteren Ebenen aufruft, müssen als Ersatz sogenannte Teststümpfe (stubs) geschrieben werden, die die fehlenden Komponenten simulieren. Der Vorteil der Top-Down-Integration ist die frühzeitige Überprüfung des gesamten Programms auf grobe Entwurfsfehler. Ein Nachteil kann die hohe Zahl von *stubs* sein, die geschrieben werden müssen. Die *stubs* verfälschen außerdem durch ihre zu starke Vereinfachung die Testergebnisse und können damit einen ausreichenden Test vortäuschen.

Als Kompromisslösung dient die *Sandwich-Integration*, bei der das System in drei Ebenen eingeteilt wird. Über der mittleren Ebene erfolgt die Integration von *top-down*, unter der mittleren Ebene *bottom-up*. [Trau1993]

Im Systemtest werden die Anwenderanforderungen in realer Umgebung getestet. Als Testgrundlagen dienen die Anforderungsspezifikation und die Gebrauchsanlei-

tung. Der Systemtest wird immer als Black-Box-Test durchgeführt. Testschwerpunkte sind Anwendungsszenarien und Überprüfung der zugesicherten Leistungen.

Danach steht der *Abnahmetest*. Hierbei wird das Produkt hinsichtlich der Anwendung des Benutzers getestet. Dies ist jedoch erst bei der Freigabe des Systems aus der Hand der Entwicklungsgruppe möglich. Getestet wird erst nach der Installation und der Einführung des Systems. Normalerweise werden beim Abnahmetest drei Formen von Tests unterschieden. Zunächst der *Benchmarktest*, bei welchen zuerst entsprechend der Problemanalyse Testfälle und Testdaten konstruiert oder aus Beständen an Originaldaten selektiert werden und danach mit den so ermittelten Testdaten in einer weitgehend sicheren Testumgebung oder in der Produktionsumgebung getestet wird. Eine weitere Testart ist der *Pilottest*. Hierbei wird ein potentieller zukünftiger Benutzer ausgesucht, welcher die Software für eine gewisse Zeit im System testet. Dieser Test kann jedoch auch durch den Paralleltest ersetzt werden, beim dem die Software durch einen oder mehreren zukünftigen Anwendern für eine begrenzte Zeit parallel zum alten Verfahren eingesetzt wird.

Nach Abschluss der Abnahmetests erfolgt der *Betriebstest*. Dieser dient zur Erkennung von Fehlern während des Einsatzes im Betrieb. Hierzu wird das Software-Produkt durch sporadische oder permanente Ausführung von Testaktivitäten mit Testfällen aus realen Verarbeitungsdaten geprüft. Dazu wird nach folgendem Ablauf verfahren: Zunächst werden die Ergebnisse der Produktionsläufe überprüft. Während des Ablaufes werden die Zwischenschritte der Programmverarbeitung auf Plausibilität geprüft. Nach dem Ablauf werden verdichtete Daten automatisiert oder nicht automatisiert geprüft, diese Ergebnisse mit alten verglichen und die Unterschiede interpretiert. Außerdem müssen mehrere Produktionsläufe in einer isolierten Umgebung getestet werden. Zuerst müssen Verarbeitungsfälle, welche im normalen Betrieb nicht anfallen, ausgewählt werden. Diese werden in einer Betriebstestdatei gespeichert. Danach wird hier eine sogenannte Testfirma mit diesen Daten simuliert. Hierbei kann auch von der Betriebsdatei abgewichen werden, da die Testfirma so angelegt ist, dass ein ständiges Update durchgeführt werden kann. Durch das Testen unmittelbar vor dem Einsatz kann sichergestellt werden, dass zwischenzeitlich keine Änderungen in die Programme eingebracht worden sind und darüber hinaus auch durch Änderungen in der Hardware/Software-Umgebung kein fehlerhaftes Ver-

halten hervorgerufen wird. Wird ein Fehler erkannt, wird sofort die Wartung des Softwaresystems begonnen.

Zur Analyse hinsichtlich der Auswirkungen von Änderungen oder Erweiterungen des Programmsystems dient der *Wartungstest*, dessen Aufgabe es ist, die geänderten sowie neuen Programmfunktionen zu testen - zuerst getrennt und schließlich in Zusammenhang mit dem restlichen Softwaresystem. Wesentlich hierbei ist, dass auf vorhandene Testfälle und Testdaten zurückgegriffen werden kann und diese somit für neue Funktionen aktualisiert beziehungsweise erweitert werden müssen, um bei anderen Tests optimale Testaktivitäten zu garantieren.

Eine der Anforderungen des Testtreibers ist es, den Entwicklern bei der Überprüfung von Vor-, Nachbedingungen und Invarianten von Methoden einer Klasse zur Hand zu gehen. Das heißt, dass der Testtreiber hauptsächlich beim Entwicklungstest speziell dem Klassentest eingesetzt wird. Der dafür zur Anwendung kommende Ansatz ist der Black-Box-Test, da die Methoden- und Klassenspezifikation zur Ableitung der Testfälle herangezogen wird.

2.4 Besonderheiten der Objektorientierung

Ziel dieser Arbeit ist es, einen Testtreiber für Java-Klassen zu entwickeln. Da Java eine objektorientierte Sprache ist, ist eine Betrachtung der Besonderheiten der Objektorientierung ein Muss, da diese das Testen erheblich erschwert.

2.4.1 Kapselung

Die Kapselung ist kein völlig neues Konzept der Objektorientierung aber stärker ausgeprägt als bei der prozeduralen Programmierung. So erfolgt der Zugriff auf die Daten eines Objektes nach Möglichkeit nur über Methoden des Objekts. Daten, die nur für die interne Implementierung der Klasse von Bedeutung sind, sollten nicht von außen zugänglich sein. Meist bieten objektorientierte Programmiersprachen Konzepte, die Kapselung zu durchbrechen und somit den Testtreibern vollständigen Zugriff auf die Daten eines Objektes zu gewähren wie zum Beispiel die Friend-Klassen in C++ und Reflections in Java. Ein großer Vorteil der Kapselung ist, dass die Objekte nach außen abgeschlossen sind und ein Zugriff nur über fest definierte Schnittstellen möglich ist, was den Test einzelner Objekte unabhängig voneinander erleichtert, und Fehlerursachen leichter lokalisieren lässt. Der Nachteil ist, dass Ob-

jekte eine Vielzahl möglicher Zustände annehmen können, welche alle ausprobiert werden müssen. Hinzu kommt, dass die Schnittstellen zu den Objekten sehr komplex werden können und auch in allen Varianten zu testen sein müssen. Methoden können zum Testen nicht einfach einzeln aufgerufen werden. Da ihre Ausführung vom jeweiligen Objektzustand abhängt und dieser Zustand durch die vorher ausgeführten Methoden geprägt wird, müssen alle Kombinationen der Methodenausführungsfolge erprobt werden. Nur dadurch ist sicher zu stellen, dass die Methoden aufeinander abgestimmt sind.

2.4.2 Vererbung

Die abgeleiteten Klassen verweisen auf Attribute und Methoden der übergeordneten Klassen, daher werden die Probleme in den übergeordneten Klassen auf alle abgeleiteten Klassen übertragen. Demzufolge müssen die abgeleiteten Klassen und ihre übergeordneten Klassen im Voraus sehr gut getestet werden. Die Vererbung gewährt abgeleiteten Klassen direkten Zugriff auf Elemente der Basisklasse und lockert damit das Kapselungsprinzip. Durch das Überschreiben geerbter Methoden wird die Komplexität der Abläufe erhöht, durch das Ändern von Parametertypen werden die Schnittstellen verdeckt, dadurch können leichter unerwünschte Seiteneffekte und Fehler entstehen. Die Methoden der Basisklasse werden außerdem in der Unterklasse in einem anderen, veränderten Kontext ausgeführt, in dem sie eventuell nicht mehr fehlerfrei funktionieren. Das bedeutet, dass abgeleitete Klassen nicht unabhängig von ihren Basisklassen getestet werden können. Vor allem bei tiefen Vererbungshierarchien geht der Überblick über alle geerbten Methoden und Attribute leicht verloren, wodurch es leicht zu unbeabsichtigter Wiederverwendung bereits vorhandener Namen und zum Überschreiben der geerbten Elemente kommen kann. Oft sind geerbte Methoden auch nicht mehr sinnvoll oder müssen angepasst werden, was unter Umständen leicht vergessen werden kann (zum Beispiel Copy(), IsEqual()). Die Möglichkeit der Mehrfachvererbung enthält noch einige weitere Fehlerquellen, ist jedoch nicht in allen objektorientierten Programmiersprachen wie zum Beispiel Java möglich. Der große Nachteil der Vererbung ist, dass – wie bereits oben erwähnt wurde – es nicht möglich ist, die abgeleiteten Klassen unabhängig von den übergeordneten Klassen zu testen. Dies führt dazu, dass in komplexen objektorientierten Systemen mit einer hohen Vererbungstiefe die untergeordneten Klassen im Sinne eines Modultests kaum testbar sind.

2.4.3 Abstrakte Klassen und Schnittstellen

Obwohl abstrakte Klassen oder Schnittstellen eigentlich nicht unbedingt getestet werden müssen, gibt es dennoch prinzipiell zwei Möglichkeiten. Zum einen können die konkreten Ableitungen der abstrakten Klassen getestet werden, wobei eventuell die Testklassen von abstrakten Testklassen einer parallelen Testhierarchie abgeleitet werden. Zum anderen kann eine konkrete Unterklasse extra für den Test erzeugt werden. Die Komplexität einer abstrakten Klasse ist jedoch meist sehr gering, so dass sich der Aufwand dafür nicht rechtfertigt, es sei denn, es existiert gar keine konkrete Ableitung, was vor allem bei der Entwicklung von Frameworks auftritt.

2.4.4 Polymorphie

Das Hauptproblem beim Test, welches die Polymorphie mit sich bringt, besteht in der dynamischen Bindung von Methoden. Da erst zur Laufzeit entschieden werden kann, welche Funktionen in welchem Objekt einen Auftrag erledigen, ist der Programmablauf nicht mehr statisch aus dem Programmtext ableitbar. Das bedeutet, dass bei der Entwicklung der Tests unter Umständen gar nicht genau klar ist, welche Bindungen zur Laufzeit auftreten können.

Zum Beispiel gibt es eine Klasse Vieleck und eine daraus abgeleitete Klasse Rechteck. Die Klasse Vieleck definiert eine Methode getUmfang, welche in der Klasse Rechteck überschrieben wird. Der folgende Codeabschnitt zeigt, wie erst zur Laufzeit entschieden wird, welche Methode benutzt wird:

```
...
Vieleck a;
Rechteck b;
...
a = new Vieleck();
b = new Rechteck();
...
if ( BedingungDieZurLaufzeitErmitteltWurde )
a = b;
System.out.println( a.getUmfang() );
...
```

Daher sollten alle möglichen dynamischen Ablauffolgen getestet werden. Da durch die mehrfache Wiederholung der Polymorphie die Anzahl möglicher Ablaufpfade geradezu explodieren kann, gestaltet sich die vollständige Abdeckung oft sehr schwierig. Der so genannte Jo-Jo-Effekt soll das Problem bei mehrfacher Wiederholung der Polymorphie verdeutlichen. Es erschwert das Verständnis und die Nachvollziehbarkeit des dynamischen Programmverhaltens, erhöht somit die Gefahr von Fehlern und erschwert die Entwicklung von geeigneten Tests und damit die Lokalisierung von Fehlern. Dieser Effekt beschreibt das Gefühl, welches beim Versuch aufkommt, die Aufrufsequenzen von Methoden nachzuvollziehen, wenn Methoden auf unterschiedlichen Hierarchieebenen überschrieben wurden. Es steht fest, dass bei der Objektorientierung einige Probleme geringer werden, zum Beispiel tief verschachtelte Ablaufstrukturen. Dafür treten neue Probleme auf wie die dynamische Bindung mit einem nicht statisch determinierbaren Zielobjekt. Die Komplexität der Programmlogik verlagert sich von der intramodularen zur intermodularen Komplexität. Offenheit und Wiederverwendung fordern neue Testansätze, welche den Besonderheiten der Objektorientierung gerecht werden. Da der Test sich immer auf ein bestimmtes Objekt bezieht, ist der Umfang des Tests von der Größe des Objekts abhängig. Ein Objekt wird aus dem Gesamtzusammenhang herausgeholt und für sich in einer kontrollierten Testumgebung getestet. Es werden Vorbedingungen erfüllt, Ausgangszustände gesetzt, Eingaben zugeführt und Ausgaben abgefangen und untersucht. Bei der Objektorientierung ist das kleinste testbare Objekt eine Methode, weil sie einen eigenen Eingang, einen Ausgang und eigene Parameter hat. Demzufolge ist es ohne weiteres möglich, einzelne Methoden aufzurufen und ihr Verhalten zu prüfen. Das nächst größere Objekt ist eine Klasse, in der alle Methoden, die einen bestimmten Objekttyp verarbeiten, zusammengefasst beziehungsweise gekapselt sind. Anhand der Klasse werden Objekte instanziert, verarbeitet und am Ende entweder zerstört oder aufbewahrt. Getestet wird eine Klasse über ihre Schnittstellen beziehungsweise über ihre Methoden, angefangen mit dem Konstruktor zur Erzeugung eines Objekts bis hin zum Destruktor zu dessen Zerstörung.

In vielen Literaturen herrscht die Meinung, dass die Klasse das kleinste Testobjekt ist. Dies ist aber nicht ganz korrekt, da eine Klasse auch nur als Methodensammlung dienen kann, dabei muss diese Klasse keine eigenen Attribute haben. Alle seine

Methoden funktionieren unabhängig voneinander. Sie sind ebenfalls unabhängig von dem internen Zustand der Klasse.

Im Java sind oft mehrere Klassen in einer Klassenmenge - einem Paket - zugeordnet. Da objektorientierte Systeme sich auf Testbarkeit und Wartbarkeit ausrichten, sollte ein System mehrere Komponenten haben, wobei Komponenten mehr oder weniger abgeschlossene Klassenmengen sind, in denen nur sehr wenige Abhängigkeiten zur Klassen in anderen Komponenten bestehen. Es soll daher nur definierte Schnittstellen beziehungsweise APIs zwischen getrennt gebildeten Komponenten geben. Schließlich ist das System selbst das endgültige Testobjekt. Da der Klassentest der Schwerpunkt dieser Diplomarbeit ist, ist es erforderlich, den Klassentest an dieser Stelle genauer zu untersuchen.

2.5 Klassentest

Es ist möglich, Klassentests in Kettentest und Unit-Test einzuteilen. Der Unit-Test ist ein Test jeder Klasse für sich, wohingegen bei dem Kettentest eine Kette von Objektinstanzen vom Anfang bis zum Ende einer Transaktion getestet wird.

Bei dem *Unit-Test* wird Methode für Methode einer Instanz getestet, wobei jede Methode zunächst für sich und anschließend in jeder möglichen Folge von Methodenabhängigkeiten der gleichen Klasse getestet wird. Hierbei muss darauf geachtet werden, dass sich vor jedem dieser Testfälle die Instanz in einem gültigen Zustand befindet, was entweder durch eine Initialisierung des Objektes oder durch einen vorherigen Testfall erreicht wird. Nach einem Testfall muss der Zustand des Objektes beziehungsweise die Ausgangsnachricht mit den Sollwerten abgeglichen werden.

Bei dem *Integrationstest* geht es darum, die Interaktion zwischen einer Kette von Klassen zu analysieren. Wichtig ist dabei vor allem, dass die Vererbung keine unerwünschten Nebenwirkungen hat. Es werden die geerbten Methoden und Attribute getestet, wobei es notwendig ist, das Zusammenwirken wieder verwendeter Klassen mit neuen Klassen zu validieren. Die Testumgebung für einen Klassentest muss aus folgenden Teilen bestehen:

Zunächst simuliert der Testtreiber die vererbten Werte und Befehle, initialisiert die Ausgangsparameter und stößt die ausgewählte Methode der zu testenden Klasse an. Danach werden die Eingangsparameter durch den Nachrichtengenerator generiert. Nachdem der Objektzustandsinitialisator das Objekt in den gewünschten Vorzustand versetzt hat, vergleicht der Objektzustandsauswerter die Ausgangsnachricht

vom Nachrichtenauswerter mit den spezifizierten Nachbedingungen. Diese Ausgangsnachricht wird durch den Nachrichtenauswerter ausgewertet. Währenddessen registriert und protokolliert der Testmonitor die Ausführungsfolge der Methoden innerhalb einer Klasse und Klassenkette. Damit solch ein Klassentest gemacht werden kann, muss eine ausführliche Klassen- und Methodenspezifikation vorliegen. Test und Spezifikation sind somit in der objektorientierten Entwicklung sehr eng miteinander verbunden.

Methoden können Einschränkungen unterliegen, je nachdem in welchem Objektzustand oder in welcher Reihenfolge sie aufgerufen werden können. Danach richtet sich auch, wie sie getestet werden. Robert V. Binder [Bin2000] unterscheidet anhand dieser Kriterien vier Arten von Klassen: Nonmodal, Unimodale, Quasimodale und Modale. Jede Art hat ihre eigenen Fehlerquellen und dementsprechend müssen sie getestet werden.

2.5.1 Nonmodale Klassen

Nonmodale Klassen unterliegen keinen Einschränkungen bezüglich des Objektzustandes oder der Aufrufreihenfolge. Einfache Klassen, die nur Werte abspeichern und lediglich primitive get- und set-Methoden besitzen, sind beispielsweise nonmodal. Beim Test können alle Methoden einzeln getestet werden.

2.5.2 Unimodale Klassen

Unimodale Klassen sind Klassen, deren Methoden nur in festgelegten Reihenfolgen aufgerufen werden können. Ein Beispiel dafür wäre eine Klasse, die ein Kartenspiel mit einer Methode für jeden Spieler simuliert. Die Spieler kommen immer in einer festgelegten Reihenfolge zum Zug, das heißt, dass die Methoden nur in dieser Reihenfolge aufgerufen werden dürfen. Für den Test bedeutet das, dass alle Methoden in jeder Aufrufsequenz getestet werden müssen.

2.5.3 Quasimodale Klassen

Quasimodale Klassen sind Klassen, deren Methoden nur bei bestimmten Objektzuständen aufgerufen werden können. Das Ergebnis der Methodenausführung wird anders ausfallen, je nachdem in welchem Zustand sich das Zielobjekt befindet. Zum

Beispiel werden, wenn das Konto gesperrt ist, alle Bewegungen unabhängig von der Reihenfolge abgelehnt. Beim Test sollten sämtliche Zustandsübergänge getestet werden.

2.5.4 Modale Klassen

Bei modalen Klassen muss der Zustand der Objekte und die Reihenfolge von Methodenaufrufen berücksichtigt werden. Als Beispiel hierfür dient eine Klasse zur Simulation einer Gangschaltung mit Rückwärtsgang. Hierbei sind die Reihenfolge der verschiedenen Gänge und der Zustand, der durch die Geschwindigkeit bestimmt wird, vorgeschrieben. Derartige Klassen sind am schwierigsten zu testen. Der Test muss nicht nur auf die Reihenfolge der Methodenausführung, sondern auch auf den Zustand der Objekte und die Werte bestimmter Member-Variablen achten. Demzufolge lässt sich dieser Test wieder aus der Abdeckung aller Zustandsübergänge im zugehörigen Zustandsübergangsgraphen ableiten.

2.5.5 Test von Unterklassen

Dass neue und redefinierte Methoden in Unterklassen getestet werden müssen, ist natürlich trivial. Doch auch unveränderte Methoden einer bereits fertig getesteten Basisklasse müssen in Unterklassen erneut getestet werden. Andere überschriebene Methoden können von der unveränderten Methode benutzte Objektattribute verändern oder die unveränderte Methode ruft direkt oder indirekt überschriebene Methoden auf, wodurch sich der Kontext, in dem die Methoden ablaufen, verändert und neue Fehler entstehen oder bereits bestehende Fehler zum Vorschein kommen. Daher wird der Test bei der Wurzel der Klassenhierarchie begonnen und dann werden nacheinander die Unterklassen in der jeweils eins tieferen Hierarchieebene getestet, wobei die dazugehörigen Testfälle bei geerbten und unveränderten Methoden der Basisklasse erneut ausgeführt werden. Für neue und redefinierte Methoden müssen neue Testfälle aufgestellt und ausgeführt werden.

2.5.6 Einschränkungen des Klassentests

Das Hauptziel der Software-Qualitätssicherung ist es, Fehler in der Entwicklung zu entdecken und zu lokalisieren. Der Klassentest ermöglicht dem Entwickler, seine ei-

genen Fehler zu finden. Außerdem wird durch ihn die Lauffähigkeit und Korrektheit seiner Klasse bestätigt. Leider garantiert der Klassentest keine Fehlerfreiheit, da er hauptsächlich Codierfehler und logische Fehler entdecken kann. Die Spezifikations- und Entwurfsfehler können jedoch nicht aufgefunden werden, weil die Klasse nur ein Teil des Systems ist. Die Schnittstellenfehler können nur dann erkannt werden, wenn mehrere Klassen miteinander getestet werden. Trotzdem ist der Klassentest sehr wichtig, da er die Bestätigung gibt, dass der Code den Vorgaben entspricht, alle vorgesehenen Eingangsparameter behandelt und alle erweiterten Ergebnisse erzeugt werden [SneWin2002].

Außer den oben beschriebenen nicht erkennbaren Fehlern besitzen Klassentests weitere Einschränkungen durch die Vorteile objektorientierter Programmierung wie Vererbung, Polymorphie, Wiederverwendung fremder Klassen und das Überladen von Parametern.

Vererbung

Abgeleitete Klassen benutzen stets Teile ihrer Oberklassen, weshalb es nicht möglich ist, diese allein zu testen - die Oberklassen müssen in den Test einbezogen werden. Daraus ergibt sich eine strenge Testreihenfolge beginnend mit den Klassen an der Spitze der Hierarchie bis zur eigentlich zu testenden Klasse. Dies kann mitunter zu einem sehr großen Testgegenstand führen.

Polymorphie

Die Kontrolle des Klassentest wird durch die Eigenheiten der dynamischen Bindung erschwert, da - wenn es aus dem Code nicht erkennbar ist, welche Variante einer Operation ausgeführt wird - es auch nicht determinierbar ist, welches Ergebnis zurückkommen wird. Daher müssen Entscheidungen getroffen werden, welche Validations-Operationen für den Klassentest bedeutsam sind, und deren Ergebnisse simuliert werden. Dadurch wird allerdings die Testüberdeckung erheblich eingeschränkt.

Sollte die polymorphe Methode zur Testklasse gehören, müssen alle möglichen Aufrufe mit den dazugehörigen Parametervariationen im Test durchgeführt werden, was dementsprechend zu einer hohen Anzahl an Testfällen führt.

Überladung von Parametern

Durch die dynamische Veränderung der Parameter wird zwar die Flexibilität der Anwendung erhöht, beim Testen allerdings müssen alle potenziellen Parameterkombinationen, mindestens jedoch die am wahrscheinlichsten vorkommenden, getestet werden. Der Mehraufwand besteht vor allem darin, dass die verschiedenen Variationen manuell im Testtreiber angepasst werden müssen, da diesem meist nur eine Standardparameterliste zur Verfügung steht.

Wiederverwendung

Die Wiederverwendung fremder Klassen stellt den Klassentest vor die Anforderung, diese Klassen ebenfalls mit in den Test einzubeziehen, da deren Fehlerfreiheit nicht garantiert werden kann. Entweder geschieht dies durch simulierte Stubs oder sie müssen direkt mitgetestet werden, wobei den Rückgabewerten besondere Aufmerksamkeit zuteil werden muss, um auszuschließen, dass ungültige Ergebnisse die eigene Klasse nicht beeinflussen [SneWin2002].

2.6 Testansätze

2.6.1 Ablaufbezogenes Testen

Das Programm wird unter Beachtung seiner Struktur gegen sich selbst getestet. Im Grunde wird dadurch die White-Box-Testmethode realisiert. Das eigentliche Testobjekt ist der Ablaufgraph des jeweiligen Programms oder Moduls. Ein Durchlauf durch den Ablaufgraph wird als Pfad bezeichnet. Ziel ist es, alle möglichen Pfade des Programms mittels Tests durch Wahl geeigneter Testdaten zu überdecken.

Diese Testmethode wurde durch Tom McCabe und Edward Miller, der zudem das erste Werkzeug zur Messung der Ablaufdeckung entwickelte, bekannt gemacht [McC1983]. Durch Miller wurden folgende Maßstäbe für den Überdeckungsgrad der Testmethoden eingeführt:

c0 = Ausführung aller Anweisungen
c1 = Ausführung aller Ablaufzweige
c2 = Erfüllung aller Bedingungen
c3 = Wiederholung aller Schleifen n-Mal

c4 = Wiederholung aller unabhängigen Pfade
c5 = Ausführung aller unabhängigen Pfade
c6 = Ausführung aller Vorwärtspfade
c7 = Ausführung sämtlicher Pfade

C2, c3, c4 sind im Allgemeinen zu aufwendig zu realisieren; c7 hingegen ist unerreichbar, vor allem im Hinblick auf beliebig oft wiederholbare Schleifen. C2 lässt sich nur im Objektcode instrumentieren. C3 und c4 sind durch Trace-Werkzeuge zu ermitteln. Geblieben sind die Maßstäbe c0, c1, c5, c6. Beizer [Bei1990] hat die Ablaufüberdeckungsmaßstäbe auf die drei Überdeckungsgrade Anweisungsüberdeckung, Zweigüberdeckung und Vorwärtspfadüberdeckung weiter reduziert.

Anfang der 80er Jahre hat das strukturierte Testen mit seinen mechanistischen Regeln großes Interesse gefunden. Vor allem die graphentheoretischen Grundsätze und die Objektivität der Methoden haben viele Forscher fasziniert. Hinzu kam die relative Leichtigkeit, mit der die Ergebnisse gemessen werden konnten. An der Effektivität dieser Methode gab es jedoch von Anfang an Zweifel; trotz hoher ablaufbezogener Überdeckung wurden weniger als 50% der Fehler angezeigt. Durch diese Form des Testens können zwar Fehler wie unerreichbare Zweige, Irrwege und endlose Schleifen erkannt werden. Inkonsistente und unvollständige Bedingungen und Abbruchfehler werden ebenso entdeckt. Nicht erkennbar sind jedoch vergessene Funktionen, Ein- und Ausgabefehler und inkonsistente Schnittstellen, denn der strukturierte Test prüft nur das, was das Programm tut und wie es dies tut, aber nicht, was es tun sollte und wie es dies tun sollte. Da die Spezifikation des Programms völlig außer Acht gelassen wird, ist der strukturierte Test für sich allein kein zuverlässiger Test. Sein Nutzen liegt vor allem in der intensiven Beschäftigung mit dem Programm. Hierbei stößt der Tester automatisch auf logische Ungereimtheiten und Lücken im Code. [For2000]

2.6.2 Datenbezogenes Testen

Das datenbezogene Testen hat seine Wurzel im Random-Datentest. Hierbei werden wahllose Eingabe-Datenkombinationen generiert und in großer Menge dem zu testenden Programm zugeführt. Die generierten Testdaten sind reine Zufallswerte, welche ebenfalls Random-Ergebnisse erzeugen. Die Auslösung der Funktionen im

Programm geschieht nur zufällig und hängt von der statistischen Streuung der Daten ab.

Dieser Ansatz wurde von G. Myers [Mye1995] etwas verfeinert. Er teilt die Eingaben in Mengen diskreter Werte, zusammenhängende Wertbereiche und Daten mit expliziten Beziehungen zu anderen Daten. Die Mengen diskreter Werte können außerdem in Äquivalenzklassen aufgeteilt werden. Alle Werte einer Äquivalenzklasse erzeugen die gleiche Wirkung im Programm. Von White und Cohen [WC1980] wurden sie als Eingabebereich (*Input Domain)* bezeichnet. Für jede Äquivalenzklasse beziehungsweise jeden Eingabebereich genügt es, einige repräsentative Werte zu testen. Das Testen mit repräsentativen Werten einer Äquivalenzklasse wird auch als *repräsentative Wertanalyse* bezeichnet. Diese Form der Vorauswahl von Testdaten ist kennzeichnend für das statische Testen.

Hierzu gehört unter anderem die *Grenzwertanalyse*. Zusammenhängende Wertebereiche oder *Ranges* werden durch die Erzeugung von Grenzwerten getestet. Nach Myers [Mye1995] genügt es, den unteren Grenzwert sowie den nächstkleineren Wert, den oberen Grenzwert, den nächstgrößeren Wert und einen Mittelwert zu generieren, um einen geschlossenen Wertebereich zu testen.

Im Gegensatz dazu beschäftigt sich die Datenflussanalyse mit der Veränderung der Daten während der Programmausführung. Dabei wird die Programmausführung in Zeitintervalle zerlegt und der Zustand der Daten nach jedem Intervall untersucht beziehungsweise neu gesetzt. Die Datenflussanalyse befasst sich vor allem mit dem temporären Zustand der Daten in Schnittstellen zwischen Modulen und externen Datenträgern. Da manche Daten nur vorübergehend in dieser Form existieren, ist der Zweck der Datenflussanalyse, diesen temporären Zustand festzuhalten und möglicherweise zu verändern. Hierfür sind spezielle Mechanismen zur Ablaufunterbrechung erforderlich, was bei Echtzeit-Programmen besonders schwierig ist. Zuletzt werden die von einem Data Dictionary aus erzeugten Testergebnisse mit den Sollergebnissen verglichen.

Zur Beschreibung der Daten gehören außer dem Typ, der Länge, der Bedeutung usw. auch der Wertebereich des Datenelements beziehungsweise seine Relation zu anderen Datenwerten. Aus diesen Wertbestimmungen heraus können Dateien und Datenbanken generiert und validiert werden. [SneWin2002]

2.6.3 Funktionsbezogenes Testen

Da bei dem ablaufbezogenen Testen oft weniger als 50% der Fehler entdeckt werden, schlug bereits 1982 Howden [How87] das funktionsbezogene Testen als Alternative vor. Statt das Programm ausschließlich gegen sich selbst zu testen, soll stattdessen die Funktion des Programms geprüft werden. Für jede Funktion müssen nicht nur die Eingabe- und Ausgabedaten, sondern auch die Eingabe- und Ausgabewerte spezifiziert werden. Wichtig ist die eindeutige Zuordnung der Funktionen zu den Code-Abschnitten und das gezielte Testen der Abschnitte. Im Vordergrund steht die Spezifkation der Programmfunktionen, entsprechend der die Testfälle ausgewählt werden. Es werden nur jene Pfade getestet, die einer spezifischen Funktion entsprechen, das heißt, nicht spezifizierte Zweige und Pfade werden außer Acht gelassen.

Beim funktionalen Testen wird das Programm gegen die Spezifikation dynamisch geprüft. Der funktionale Ansatz nähert sich den Anforderungen der Programmverifikation, wobei Verifikation im Gegensatz zur Validierung der Beweis ist, dass ein Programm im Sinne der Spezifikation korrekt ist.

2.6.4 Regressionstesten

Regressionstesten wird je nach Quelle unterschiedlich definiert. Ändert sich die Implementierung einer bereits getesteten Komponente, ohne dass ihre Spezifikation geändert wurde, ist ein Regressionstest erforderlich. Sollte zusätzlich die Spezifikation geändert werden, müssen für diese Komponente neue Testfälle erzeugt oder bereits existierende Testfälle entsprechend angepasst und angewendet werden. Dieser Fall wird von McGregor [MgSy2001] nicht mehr dem Regressionstest zugerechnet. Binder [Bin2000] hingegen definiert den Regressionstest als die Wiederverwendung von Testfällen. Darin eingeschlossen sind der Test von Funktionalitäten einer Oberklasse im Kontext von ihr erbender Klassen unter der Wiederverwendung von Testfällen der Oberklasse ebenso wie der erneute Test einer wieder verwendeten Komponente in ihrer neuen Umgebung.

Der Regressionstest kann auf allen Ebenen eines Systems sowie während der verschiedenen Entwicklungsstufen innerhalb eines iterativen Entwicklungsprozesses erfolgen. Eine Testsuite für eine Komponente wird dabei im Laufe der Entwicklung

immer weiter entwickelt, wobei Testfälle, die in einer veränderten Komponente nicht mehr ausgeführt werden können oder die auf Grund einer veränderten Spezifikation keine aussagekräftigen oder sogar falsche Ergebnisse liefern, aus der Testsuite entfernt oder entsprechend angepasst werden. Für neue Funktionalitäten werden Testfälle hinzugefügt. Dafür werden Abgleichmethoden ausgeführt, die die Eigenschaften des neuen Programms mit denen des alten in fünf Bereichen abgleichen. Diese fünf Bereiche sind der Eingabe/Ausgabe-Bereich, die Datenverwendungen, die Geschäftsregeln, Ein/Ausgabe-Pfade und die Datenausgaben.

2.6.5 Theoretische Ansätze zum Klassentest

Es kann zwischen zwei Arten des Klassentests unterschieden werden, dem implementierungsbezogenen und dem spezifikationsbezogenen Klassentest [Bin1996].

Bei dem implementierungsbezogenen Klassentest wird von dem Quellcode der Klasse ausgegangen. Zunächst werden Methoden, Verzweigungen, Parameter und Operationsaufrufe identifiziert und danach der Test so angelegt, dass alle Verzweigungen durchlaufen, alle Methoden mit allen Parametern durchlaufen und alle Operationsaufrufe ausgeführt werden. Dabei wird davon ausgegangen, dass der Entwickler Vor- und Nachbedingungen in den Klassencode eingebaut hat. Aufbauend auf diese Voraussetzung werden die entsprechenden Testfälle generiert, die Klasse unter Test schrittweise interpretiert und ihre Zustände mit den zugesicherten Zuständen abgeglichen [MgSy1992].

Eine andere Herangehensweise ist, alle Klassen zusätzlich zur Implementierungssprache auch noch in einer formalen Spezifikationssprache zu schreiben. Dadurch können diese beiden Formen sowohl statisch als auch dynamisch miteinander abgeglichen werden. Statisch können die Signaturen, Ablauflogikmuster und Bedingungen verglichen werden, dynamisch hingegen die Spezifikationen analysiert werden, um modellhafte Ablaufpfade zu gewinnen, die gegen die tatsächlichen Ablaufpfade geprüft werden. Um dies erreichen zu können, muss zuvor die Relation zwischen Objektvor- und Nachzuständen sowie zwischen Parametereingangswerten

und Rückgabewerten beschrieben werden. Der Treiber versorgt die Vorbedingungen, ruft die Klassenmethoden auf und bestätigt die Nachbedingungen.

Bei der Durchführung des spezifikationsbezogenen Klassentests wird die Klassenimplementation gegen die semantisch äquivalente Klassenspezifikation getestet.

2.6.6 Praktische Ansätze zum Klassentest

Klassentesttreiber

Ein Klassentesttreiber kennt die Operationsschnittstellen der Zielklasse und kann die Operationen der Reihe nach aufrufen. Problematisch ist jedoch das Besorgen der richtigen Parameter. Diese werden entweder über die Benutzeroberfläche oder vor der Testausführung in einem Testskript festgelegt. Die Ergebnisse müssen angezeigt und manuell geprüft oder automatisch mit definierten Sollwerten verglichen werden. Des Weiteren muss der Klassentesttreiber zusätzliche klassenspezifische Anforderungen wie Vererbung, Polymorphie und mehrfache Instanzierung erfüllen.

Test_initialisieren		
Testfall_Anzahl_holen		
Tue für jeden Testfall		
	Funktionsname_holen	
	Testfallnummer_holen	
	Argumente_holen	
	Argumente_setzen	
	Funktion_aufrufen	
	Ergebnis_prüfen	
	Ergebnis: Gültig	Ergebnis: Ungültig
	Melde_Bestätigung	Melde_Fehler
	Testfall_abmelden	Testfall_anmelden
Testprotokoll_ausgeben		
Test_abschließen		

Build-In Tests

Bei der Build-In Testtechnik wird im Vergleich zu den Klassentesttreibern ein völlig anderer Weg gegangen – statt die Klasse von außen zu testen, wurde der Test in die

Klasse eingebaut. Testoperationen werden in die Klasse integriert, von denen aus die anderen Methoden aufgerufen werden. Der Vorteil dabei ist, dass die Testoperationen Zugang zu den Objektzuständen haben und diese manipulieren können. Außerdem werden keine weiteren Quellcode-Dateien benötigt, der Entwickler schreibt sowohl die Implementierungs- als auch die Testoperation gemeinsam fort. Ebenso enthalten sind die Stub-Operationen, welche zur Simulation fremder Operationsaufrufe dienen.

Zusicherungstests

Hierbei werden Operationen in der Schnittstelle durch Angabe ihrer Signatur (Parameter und ihre Typen beziehungsweise Klassen) und ihrer Semantik beschrieben. Präzisiert wird die Semantik durch Vor- und Nachbedingungen sowie durch eine Klasseninvariante. Durch die Invariante können gemeinsame Teile aus allen Vor- und Nachbedingungen gezogen und notiert werden, da sie vor und nach jeder Ausführung jeder Operation der Klasse gilt.

Ziel des Zusicherungstests ist es, die Schnittstelle einer Klasse, das heißt nicht öffentlich sichtbare Operationen und Invariablen, zu testen. So kann die Konformität der Operationen einer Klasse zu ihrer Spezifikation und das Zusammenspiel mehrerer Operationen geprüft werden. Außerdem können die für die Operation möglichen Ausnahmen erzeugt werden. Voraussetzung ist jedoch, dass für jede Operation der zu testenden Klasse Zusicherungen (*constraints* oder *assertions*) vorliegen. Als Ergebnis wird eine Klasse erzeugt, deren öffentlich sichtbaren Operationen gegen die spezifizierten Zusicherungen getestet sind.

Bei der Testfallerzeugung wird zunächst von den Vorbedingungen ausgegangen. Es werden für jede zusammengesetzte Vorbedingung Testfälle ausgewählt, bei denen jeder Term mindestens einmal entscheidend für den Wahrheitswert der gesamten Vorbedingung ist (minimale Mehrfach-Bedingungsüberdeckung). Da Invarianten vor und nach jeder Operationsausführung erfüllt sein müssen, sollte jede Operation für jede mögliche Art erfüllter Invarianten getestet werden. Bei den Nachbedingungen wird versucht, Testfälle für nicht erfüllte Nachbedingungen zu finden. Die Festlegung der Testfälle erfolgt inkrementell. Der Aufruf von Operationen der zu testenden Klasse erfolgt, beginnend mit dem Konstruktor, so lange, bis die geforderte Überdeckung der Terme erreicht ist. Unter Nichtbeachtung bereits vollständig überdeckter Terme, wird die erwartete Auswirkung der Operationsausführung auf die Vorbedin-

gungen weiterer Operationen notiert. Daher können Testfälle für diese Operationen auf bereits ermittelte Testfälle aufbauen. Dieses Verfahren wird so lange fortgesetzt, bis alle Prädikate überdeckt sind.

Der Test gilt als beendet, wenn jede Operation mindestens einmal ausgeführt wurde, jede Zusicherung entsprechend der minimalen Mehrfach-Bedingungsüberdeckung geprüft wurde und jede Ausnahme mindestens einmal geworfen wurde.

Die Zusicherungstechnik ist äußerst wichtig beim Test einzelner Klassen, besonders im Zusammenhang mit Testtreibern und Teststubs. Sie dient vor allem dazu, die Annahmen des Entwicklers zu bestätigen und unangenehme Überraschungen auszuschließen [SneWin2002].

2.7 OCL

Da objektorientierte Software zurzeit hauptsächlich durch UML (Unified Modelling Language) spezifiziert wird und die Spezifikation eines Programms die Spezifikation der Testfälle bestimmt, kann mittels UML die Spezifikation der Testfälle abgeleitet werden. Eine besondere Komponente des UML ist OCL (Object Constraint Language). Mit OCL lassen sich Zusicherungen über Objektzustände sowie über Vor- und Nachbedingungen verfassen, daher ist OCL eine geeignete Sprache für die Testfallspezifikation. Durch die Extrahierung und Allokierung der OCL-Anweisungen lassen sich Testprozeduren bilden, die für den Test der Klassen gegen die Klassenspezifikation genutzt werden. Deshalb wird in dem zu entwickelnden Testtreiber OCL benutzt, um die Zustände der Testobjekte zu sichern, die Eingangsparameter zu spezifizieren und die Ausgangsparameter zu validieren [OMG99].

2.7.1 Eigenschaften

Es ist erforderlich, an dieser Stelle auf die Eigenschaften und das Verhalten von OCL einzugehen, da diese eine entscheidende Rolle bei der Testdatengenerierung des Testtreibers spielen. Die Unified Modelling Language besteht primär aus Diagrammen, die durch natürliche Sprache erweitert wurden, um deren Bedeutung zu verdeutlichen. Es gibt jedoch Feinheiten wie zum Beispiel Einschränkungen und Regeln, die ein Diagramm nicht ausdrücken kann. Aus diesem Grund wird eine

formale Sprache benötigt, die in Kombination mit Diagrammen auch die Feinheiten beschreibt und eine eindeutige Verständlichkeit garantiert.

Diese Sprache sollte für die Formulierung von Einschränkungen (constraints) die formale Strenge der mathematischen Notation nutzen, gleichzeitig aber zur besseren Verständlichkeit eine Syntax ähnlich einer Programmiersprache besitzen.

Hierzu wurde 1995 erstmals die Object Constraint Language (OCL) in einem IBM-Projekt entwickelt. OCL besitzt eine klare Syntax, so dass sie von jedem, der Programmierkenntnisse hat, leicht und schnell erlernt werden kann. Somit eignet sich OCL als Standard zur Definition von Regeln in der objekt-orientierten Welt.

OCL wird immer in Verbindung mit einem UML-Modell verwendet; durch OCL können zusätzliche Information für Modelle ausgedrückt werden, die in UML nicht ausdrückbar sind. OCL ist nicht nur präzise und eindeutig, sondern kann gleichzeitig von Leuten gelesen werden können, die nicht Mathematiker sind. Sie ist eine deklarative Sprache, das heißt, sie besteht nur aus Definitionen; es gibt insbesondere keine Zuweisungen. Ausdrücke in OCL haben keine Seiteneffekte, die Auswertung einer Vorbedingung, einer Nachbedingung oder einer Invariante verändert den Zustand des Systems nicht. Außerdem sind diese Ausdrücke auswertbar, ohne dass das zugehörige UML-Modell ausgeführt werden muss [OMG99].

Ein Constraint ist eine Beschränkung des zulässigen Wertebereichs für einen Teil eines (objekt-orientierten) Modells oder Informatik-Systems. Sie wird durch Invarianten, Vor- und Nachbedingungen definiert, die während des Aufrufs einer Methode gelten müssen. Constraints dienen zur Definition eines so genannten *objektorientierten Vertrags* (Kontrakt). Die Definition eines Kontrakts kann von der Bedeutung eines rechtlichen Vertrages abgeleitet werden.

Ein Vertrag besteht normalerweise aus mehreren Teilen, die Rechte und Verpflichtungen genannt werden. Er wird zwischen Anbieter und Kunden geschlossen. Rechte sind Spezifikationen für die Benutzung von Diensten einer Komponente (Klasse, Subsystem). Der Kunde des Vertrages muss sicherstellen, dass er keines dieser Rechte verletzt, wenn er einen Dienst nutzt. Verpflichtungen hingegen sind Spezifikationen für die korrekte Ausführung eines Dienstes. Der Anbieter des Vertrages muss sicherstellen, dass die Verpflichtungen erfüllt werden, sofern der Benutzer

sich an die Rechte hält. Verträge sind nicht erfüllt, wenn entweder Rechte nicht eingehalten oder Verpflichtungen nicht erfüllt werden. Bei einem Vertragsbruch ist immer klar, wer den Vertrag gebrochen hat; entweder der Kunde, wenn er die Rechte nicht einhält oder der Anbieter, wenn Verpflichtungen nicht erfüllt sind. Ein Vertrag gilt für alle von einer Klasse angebotenen Dienste, für die Vor- und Nachbedingungen spezifiziert werden können. Eine Vorbedingung muss gelten, bevor der Dienst ausgeführt werden kann; eine Nachbedingung hingegen erst nach der Ausführung des Dienstes – sofern alle Vorbedingungen erfüllt sind. Die Klasse, welche die Dienste anbietet, ist der Anbieter (supplier). Die Klasse, die die Dienste eines Anbieters nutzt, wird als Kunde (client oder consumer) bezeichnet. Per Definition ist ein objektorientierter Vertrag (auch OO-Vertrag oder kurz Vertrag genannt) die Spezifikation der Schnittstelle eines Anbieters, bestehend aus Beschreibungen von Invarianten sowie Vor- und Nachbedingungen für jeden angebotenen Dienst. Im Unterschied zu Verträgen im rechtlichen Sinn besteht ein OO-Vertrag unabhängig davon, ob ein Kunde existiert. Diese Definition eines Kontrakts ist auch Grundlage bei der Bildung von Invarianten, Preconditions und Postconditions in OCL.

2.7.2 Invarianten

Eine Invariante kann mit einer Klasse, einem Typ oder einem Interface eines UML-Modells verbunden sein. Ein als Invariante geschriebener Ausdruck muss für alle Instanzen der verbundenen Klasse, des Typs oder des Interfaces zu jeder Zeit *true* sein. Daher können mittels Invarianten Grenzen für Attributswerte definiert werden. Um in einem UML-Modell Invarianten darzustellen, wird der Ausdruck in geschwungenen Klammern in einen Kommentarkasten geschrieben und mit einer gepunkteten Linie mit der verbundenen Klasse, dem Typ oder Interface verbunden. Eine einfachere und übersichtlichere Darstellung kann erreicht werden, wenn die Invarianten in einer eigenen Textdatei ablegt werden [WK1999].

Folgendes Beispiel berechnet mittels der Methode *heronSqrt()* der Klasse *Heron* die Quadratwurzel der positiven reellen Zahl *value*:

```
public class Heron {
   double value;

   public Heron(double value) {
      this.value = value;
   }

   public double getValue() {
      return value;
   }

   public double heronSqrt ( ) {
      double h, x, y;
      x = value; y = 1;
      if (value < 1) {
         h = x;
         x = y;
         y = h;
      }
      while (x > y) {
         x = (x + y)/2;
         y = value / x ;
      }
      return x;
   }
}
```

Die Invariante der Klasse wird wie folgt geschrieben:

```
context Complex
inv: self.value > 0.0
```

2.7.3 Preconditions und Postconditions

Ein OCL-Ausdruck kann auch Teil einer Precondition oder Postcondition sein. Eine Precondition ist ein Ausdruck, der zu Beginn der Ausführung *true* sein muss; eine Postcondition muss am Ende der Ausführung *true* sein. In OCL kann eine Pre- und Postcondition von Operationen und Methoden wie die Invarianten für alle Klassen, Typen und Interfaces eingesetzt werden. Die Notation kann ähnlich der der Invarianten in einem Kommentarkästchen mit dem Zusatz <<precondition>> beziehungsweise <<postcondition>> oder in einer Textdatei erfolgen [WK1999]. Im letzteren Fall würden solche OCL-Ausdrücke wie folgt geschrieben werden:

```
Typename::operationName(parameter1 : Type1, ...): ReturnType
        pre : parameter1 > ...
        post: result = ...
```

Beispiel:

```
public class Complex {
   private float real = 0.0F, imag = 0.0F;
   public Complex(float real, float imag) {
      this.real = real;
      this.imag = imag;
   }
   …
   public void div(Complex z) {
      float d, nenn;
      if (Math.abs(z.real) >= Math.abs(z.imag)){
         if (z.real == 0) {
            real = imag / z.imag;
            imag = - real / z.imag;
         }
         else {
            d = z.imag /z.real;
            nenn = z.real+ d * z.imag;
            real = (real + d * imag) / nenn;
            imag = (imag -  d * real) / nenn;
```

```
        }
      }
      else {
        if (z.imag == 0) {
          real = real / z.real;
          imag = imag / z.real;
        }
        else {
          d = z.real/ z.imag;
          nenn = d* z.real + z.imag;
          real = (d * real + imag) / nenn;
          imag = (d * imag - real) / nenn;
        }
      }
    }
}
```

Die Methode *div(Complex)* der Klasse Complex berechnet den Quotienten zweier komplexer Zahlen. Dabei dürfen der Realteil und der Imaginärteil des Divisors nicht gleichzeitg 0 sein. Außerdem müssen die Ergebnisse lauten:

$$\frac{z_1}{z_2} = \frac{a_1 a_2 + b_1 b_2 + (a_2 b_1 - a_1 b_2) i}{a_2^2 + b_2^2}.$$

Demzufolge lautet der Constraint der Methode *div(Complex)*:

```
context Complex::div(z : Complex)
pre: (z.real <> 0) and (z.imag <> 0)
post: (self.real =  (self.real@pre * z.real + self.imag@pre * z.imag)
                    /( z.real * z.real + z.imag * z.imag)) and
      (self.imag =  (self.imag@pre * z.real -  self.real@pre * z.imag)
                    /( z.real * z.real + z.imag * z.imag))
```

2.7.4 OCL-Konstrukte

Self

Jeder OCL-Ausdruck wird im Kontext einer Instanz eines Typs geschrieben. Manchmal ist es jedoch erforderlich, sich explizit auf das entsprechende Objekt zu beziehen. Dazu gibt es in OCL den Ausdruck *self*. Immer wenn der Ausdruck *self* benutzt wird, bezieht er sich auf den Kontext des OCL-Ausdrucks. In den meisten Fällen kann *self* ausgelassen werden, da der Kontext eindeutig ist [IBM].

Typen und Operationen

In OCL hat jedes Objekt einen Typ, der die auf das Objekt anwendbaren Operationen definiert. Die in OCL verwendeten Typen können in zwei große Klassen eingeteilt werden: vordefinierte Typen und benutzerdefinierte Typen. Die vordefinierten Typen beinhalten die Basistypen *Integer*, *Real*, *String* und *Boolean*, sowie die Collection-Typen *Collection*, *Set*, *Bag* und *Sequence*, die im folgenden genauer beschrieben werden.

Boolean Typ

Der Typ Boolean kann wie in anderen Sprachen nur die Werte *true* oder *false* annehmen. Auch die auf dem Typ Boolean definierten Operationen sind die aus anderen Sprachen bekannten wie *oder*, *und*, *exklusives oder*, *Negation*, *Gleichheit*, *Ungleichheit* und *if then else*.

Operation	Notation	Ergebnistyp
oder	a or b	Boolean
und	a and b	Boolean
exklusives oder	a xor b	Boolean
Negation	not a	Boolean
gleich	a = b	Boolean
ungleich	a <> b	Boolean
implies	a implies b	Boolean
If then else	if a then b else b' endif	Type of b and b'

Standardoperationen für den Typ Boolean

In OCL gibt es darüber hinaus noch eine weitere Operation, die *implies*-Operation. Diese Operation entspricht der mathematischen Implikation.

Integer und Real Typen

In OCL repräsentiert der Integer-Typ die Menge der ganzen Zahlen, bei der es hier keine Obergrenze gibt, da es sich bei OCL um eine Modellierungssprache handelt. Dasselbe gilt auch für den Real-Typ, der die aus anderen Sprachen bekannten Real-Zahlen - Gleitkommazahlen - beinhaltet. Die auf diesen Typen definierten Operationen sind die üblichen wie Addition, Subtraktion, Multiplikation und Division. Dazu gibt es noch eine Absolut-Operation z.B. (*-1).abs*, die den Absolutwert, im Beispiel die 1, zurückgibt. Des Weiteren existieren für den Real-Typ noch zwei Rundungsoperationen. Die *floor*-Operation rundet den Real-Wert auf den nächsten Integer-Wert ab; z.B. *(3.6).floor* wird gerundet auf 3. Mit der zweiten Rundungsoperation, dem *round*, wird eine Real-Zahl auf die nächste Integer-Zahl gerundet; z.B. *(3.6).round* ergibt 4.

Operation	Notation	Ergebnistyp
gleich	a = b	Boolean
ungleich	a <> b	Boolean
kleiner	a < b	Boolean
größer	a > b	Boolean
kleiner gleich	a <= b	Boolean
größer gleich	a >= b	Boolean
plus	a + b	Integer oder Real
minus	a – b	Integer oder Real
Multiplikation	a * b	Integer oder Real
Division	a / b	Real
Modulus	a.mod(b)	Integer
Integer Division	a.div(b)	Integer
Absolutwert	a.abs	Integer oder Real
Minimum von a und b	a.min(b)	Integer oder Real
Runden	a.round	Integer
Abrunden	a.floor	Integer

Standardoperationen für den Typ Integer und Real

String Typ

Strings sind Zeichenfolgen, die in OCL in halben Anführungszeichen geschrieben werden. Die Operationen auf Strings sind *toUpper*, *toLower*, *size*, *substring* und *concat*. Mittels der Operationen toUpper und toLower wird der eingegebene String in Klein- beziehungsweise in Großbuchstaben dargestellt. Die Operation *size* gibt die Länge des eingegebenen Strings aus. Über die Operation substring(int1,int2) wird der Teil des Strings von int1 bis int2 ausgegeben; durch concat(string) wird ein String an den bestehenden String angefügt. Zusätzlich gibt es vergleichende Operationen.

Operation	Notation	Ergebnistyp
Verbindung	string.concat(string)	String
Länge	string.size	Integer
Kleinbuchstaben	string.toLower	String
Großbuchstaben	string.toUpper	String
Substring	string.substring(int.int)	String
gleich	string1 = string2	Boolean
ungleich	string1 <> string2	Boolean

Standardoperationen für den Typ String

Kollektionstypen

In objektorientierten Systemen sind 1-zu-1-Assoziationen selten; die meisten Assoziationen beschreiben Beziehungen zwischen einem Objekt und einer Kollektion anderer Objekte. Um mit solchen Assoziationen arbeiten zu können, sind in OCL vier Kollektionstypen vordefiniert – *Set*, *Bag* und *Sequence* als konkrete Typen zum Gebrauch in Constraints und der abstrakte Typ *Collection* als Supertyp zur Definition gängiger Operationen auf allen Kollektionstypen. Die Elemente werden in geschwungenen Klammern geschrieben; der Typ der Kollektion steht vor der Klammer. Die drei konkreten Kollektionstypen können wie folgt definiert werden:

- Ein Set enthält Instanzen eines gültigen OCL-Typs ohne doppelte Elemente.
  z.B.: Set{1, 3, 5, 7} oder Set{'Apfel', 'Birne', 'Orange'}
- Ein Bag verhält sich wie ein Set, kann jedoch dasselbe Element auch mehrfach enthalten z.B. bei der Kombination von Navigationen.
  z.B.: Bag {1, 3, 5, 7, 1} entspricht Bag {3, 1, 1, 7, 5}

- Eine Sequence ist ein Bag mit geordneten Elementen; das heißt, die Elemente sind nach einer Sequenznummer geordnet; im Falle einer Sequence von Integer- oder Real-Zahlen muss die Reihenfolge nicht durch die Werte der enthaltenen Zahlen bestimmt sein.

  z.B.: Sequence {1, 3, 5, 7, 1} ist nicht identisch mit Sequence {3, 1, 1, 7, 5}

Zu diesen Kollektionstypen gehört eine Vielzahl von vordefinierten Operationen. Alle diese Operationen auf Kollektionen werden in OCL mittels der Pfeil-Notation beschrieben, wodurch einfach zwischen Operationen auf Modelltypen und solchen auf Kollektionstypen unterschieden werden kann. In OCL werden Kollektionen automatisch zusammengefasst, das heißt, eine Kollektion kann niemals andere Kollektionen enthalten, sondern nur Objekte. Wird eine Kollektion in eine andere eingefügt, so wird die neue Kollektion automatisch zusammengefasst, und die Elemente der eingefügten Kollektion werden als direkte Elemente der neuen Kollektion betrachtet. Nach dieser Richtlinie sind die folgenden beiden Kollektionen identisch:

Set { Set {1, 2}, Set {3, 4}, Set {5, 6} } ..... Set {1, 2, 3, 4, 5, 6 }

Operation	Beschreibung
Size	Anzahl der Elemente in der Kollektion
count(object)	Anzahl, wie oft *object* in der Kollektion auftritt
includes(object)	*True*, wenn das Objekt ein Element der Kollektion ist
includesAll(collection)	*True*, wenn alle Elemente der Parameters collection zur Kollektion gehören
isEmpty	*True*, wenn die Kollektion keine Elemente enthält
notEmpty	*True*, wenn die Kollektion ein oder mehrere Elemente enthält
iterate(expression)	Ein Ausdruck wird für jedes Element der Kollektion ausgewertet. Der Ergebnistyp (result type) hängt von dem Ausdruck ab.
sum()	Die Addition aller Elemente der Kollektion. Die Elemente müssen einem Typ zugehörig sein, der Addition ermöglicht (real, integer …)
exists(expression)	*True*, wenn *expression* für mindestens ein Element der Kollektion wahr ist
forAll(expression)	*True*, wenn *expression* für alle Elemente wahr ist.

Standardoperationen für den Kollektion Typ

Neben den hier aufgeführten Standardoperationen gibt es weitere Operationen für alle drei Kollektionstypen, die jedoch jeweils eine etwas andere Bedeutung haben.

- Der *equals*-Operator (’=’) wird im Falle zweier Sets zu *true* ausgewertet, wenn beide Sets dieselben Elemente enthalten. Im Falle von Bags muss ebenfalls die Anzahl des Auftretens des jeweiligen Elements gleich sein; bei Sequences muss außerdem die Reihenfolge dieselbe sein.
  Set {1, 2, 5, 88} = Set {2, 88, 1, 5}
  Bag {’Willi’, ’Erna’, ’Horst’, ’Willi’} = Bag {’Erna’, ’Willi’, ’Horst’, ’Willi’}
  Sequence {1..(6+4)} und Sequence {1..10} sind identisch zu Sequence {1,2,3,4,5,6,7,8,9,10}
- Die *union*-Operation kombiniert zwei Kollektionen zu einer neuen. Es kann auch ein Set mit einem Bag kombiniert werden, jedoch kann eine Sequence nur mit einer anderen Sequence und nicht mit einem Set oder Bag kombiniert werden.
  Set {1, 2, 5, 88}-> union(Bag {3, 7, 27}) = Bag {1, 2, 5, 88, 3, 7, 27}
- Die *including*-Operation fügt ein Element zu einer Kollektion hinzu. Im Falle eines Sets wird das Element nur hinzugefügt, wenn es nicht schon im Set enthalten ist. Bei einer Sequence wird das neue Element am Ende eingefügt.
  Set {1, 2, 5, 88} -> including({27, 5}) = Set {1, 2, 5, 88, 27}
  Sequence {1, 2, 5, 88} -> including({27}) = Sequence {1, 2, 5, 88, 27}
- Die *excluding*-Operation entfernt ein Element aus der Kollektion. Bei einem Set wird nur ein Element entfernt; bei einem Bag oder einer Sequence wird das Objekt jedesmal entfernt, wenn es auftritt.
  Set {1, 2, 5, 88} -> excluding({5}) = Set {1, 2, 88}
  Sequence {1, 2, 5, 2, 88} -> excluding({2}) = Sequence {1, 5, 88}
- Die *intersection*-Operation erzeugt eine neue Kollektion, in der alle Elemente enthalten sind, die in zwei Kollektionen vorkommen. Diese Operation ist für alle Kombinationen von Kollektionstypen möglich, außer für Kombinationen in Verbindung mit einer Sequence.
  Set {1, 2, 5, 88} -> intersection(Bag {5, 5, 1}) = Set {1, 5}
  Bag {1, 2, 5, 1, 88} -> intersection(Bag {1, 27, 5, 1}) = Bag {1, 5, 1}

Es gibt zwei weitere Operationen, die nur für den Typ Set definiert sind. Dies sind die Operationen *minus* ('-') und *symmetricDifference*.

- Die minus-Operation erzeugt ein neues Set, in dem alle Elemente des ersten Sets enthalten sind, nicht jedoch die des zweiten Sets.
  Set{1,4,7,10} – Set{4,7} = Set{1,10}
- Die symmetricDifference-Operation ergibt ein neues Set, in dem alle Elemente enthalten sind, die im ersten oder zweiten Set vorkommen, nicht aber in beiden.
  Set{1,4,7,10}.symmetricDifference(Set{4,5,7}) = Set{1,5,10}

Auch für den Sequence-Typ gibt es zusätzliche Operationen, die mit der Sortierung der Sequence zu tun haben.

- Die Operationen *first* und *last* geben des erste beziehungsweise das letzte Element der Sequence aus.
  Sequence{1,4,7,10}->first = 1
  Sequence{1,4,7,10}->last = 10
- Die Operation *at* gibt das Element an der bezeichneten Stelle aus.
  Sequence{1,4,7,10}->at(3) = 7
- Die Operationen *append* und *prepend* fügen ein Element an letzter beziehungsweise an erster Stelle der Sequence ein.
  Sequence{1,4,7,10}->append(15) = Sequence{1,4,7,10,15}
  Sequence{1,4,7,10}->prepend(15) = Sequence{15,1,4,7,10}

Zusätzlich zu diesen allgemeinen und speziellen Operationen gibt es Operationen, die mit den Elementen einer Kollektion arbeiten; dies sind die Operationen *select*, *reject*, *collect*, *iterate*, *forAll* und *exists*. Die Operationen *select*, *reject* und *collect* ergeben eine neue Kollektion, das Ergebnis von *forAll* und *exists* ist vom Typ Boolean und der Typ der Operation *iterate* hängt von den Argumenten ab. Die *select*-Operation wird benutzt, wenn nur an einem bestimmten Bereich einer Kollektion Interesse besteht. Der Parameter des *select* ist ein boolscher Ausdruck, durch den die auszuwählenden Elemente spezifiziert werden. Das Ergebnis dieser Operation ist immer eine Unterstruktur der ursprünglichen Kollektion, in der die Elemente enthalten sind, für die der boolsche Ausdruck zu *true* ausgewertet wurde. Die Syntax der

select-Operation kann drei unterschiedliche Formen haben, die denselben Inhalt beschreiben:

collection->select(element : Type | <expression>)
collection->select(element | <expression>)
collection->select(<expression>)

Für den Testtreiber wird erstere verwendet, da damit die Typenüberprüfung einfacher ist.

Diese Syntax ist in entsprechender Weise auch für die Operationen *reject*, *collect*, *forAll* und *exists* anzuwenden. (Die folgenden Beispiele beziehen auf das Klassendiagram im Anhang C.)

CustomerCard
self.transactions->select(points > 100)

Dieser Ausdruck fasst alle Transaktionen mit mehr als 100 Punkten einer Kundenkarte in einer neuen Kollektion zusammen. Die *reject*-Operation ist analog dem gerade besprochenen *select* mit dem Unterschied, dass das *reject* alle Elemente einer Kollektion ausgibt, auf die der boolsche Ausdruck nicht zutrifft. Demzufolge haben die folgenden Ausdrücke dieselbe Bedeutung.

Customer
membership.loyaltyAccount->select(points > 0)

Die nächste zu behandelnde Operation für Kollektionen ist das *collect*. Diese Operation arbeitet auf den Elementen einer Kollektion, berechnet einen Wert für jedes Element und legt die Ergebnisse in einer neuen Kollektion ab. Dabei ist die Ergebnis-Kollektion unabhängig von der Ausgangskollektion eine Sequence. So kann in einem OCL-Ausdruck zum Beispiel bestimmt werden, dass mindestens ein Wert des Attributs *points* in *Transaction* 500 sein muss.

LoyaltyAccount
transaction->collect(points)->exist(p : Integer | p = 500)

Für diese Operation gibt es in der OCL auch eine Kurzschreibweise, so dass dieses Beispiel geschrieben werden kann als:

LoyaltyAccount
transaction.points->exist(p : Integer | p = 500)

Es gibt zwei Operationen für Kollektionen, die ein Ergebnis vom Typ Boolean haben und nicht eine neue Kollektion bilden wie die bisherigen drei. Zum einen gibt es die Operation *forAll*, in der eine Bedingung formuliert werden kann, die für alle Elemente einer Kollektion zutreffen muss. Ist die formulierte Bedingung nur für ein Element der Kollektion *false*, so ist das Ergebnis der *forAll*-Operation ebenfalls *false*. Der folgende Ausdruck verlangt, dass das Alter aller Bankkunden mindestens 18 Jahre sein muss:

Kunde
self.customer->forAll(c : Customer | c.age() >= 18)

Die *exists*-Operation wiederum kann benutzt werden, um zu spezifizieren, dass es in einer Kollektion mindestens ein Element geben muss, für das eine bestimmte Bedingung wahr ist. Daraus ergibt sich auch, dass das Ergebnis dieser Operation vom Typ Boolean sein muss. Die Operation *exists* wertet zu *true* aus, wenn die Bedingung für mindestens ein Element der Kollektion *true* ist; ist die Bedingung für alle Elemente *false*, so ist das Gesamtergebnis ebenfalls *false*. Zum Beispiel verlangt der folgende Ausdruck, dass es eine *Transaction* mit mehr als null Punkten geben muss, wenn das Attribut *points* von *LoyaltyAccount* größer als null ist.

LoyaltyAccount
points > 0 implies transaction->exists(points > 0)

Die *iterate*-Operation ist die komplexeste der Operationen für Kollektionen. Alle anderen Operationen - *select*, *reject*, *collect*, *forAll* und *exists* – können als Spezialfälle von *iterate* betrachtet werden. Die Syntax dieser Operation unterscheidet sich von der der bisherigen Operationen und sieht wie folgt aus:

collection->iterate(element : Type1;
result : Type2 = <expression>
| <expression-with-element-and-result>)

Die iterate-Operation kann vereinfacht durch folgenden Pseudocode dargestellt werden:

result = <expression>;
while(collection.notEmpty()) do
    element = collection.nextElement();
    result = <expression-with-element-and-result>
endwhile
return result;

Ein Beispiel der Benutzung von iterate-Operation der OCL :

context b : Branch:
b.employee -> iterate (e : Employee;
names : String = ´´ |
names.concat(e.lastname))

Modelltypen

In OCL werden Constraints immer mit Bezug auf ein bestimmtes UML-Modell definiert. Der Entwickler eines UML-Modells kann somit neue Typen schaffen, die dieselbe Gültigkeit besitzen wie die vordefinierten Typen. Die Eigenschaften solcher Modelltypen sind Attribute, Operationen und Methoden, aus den Assoziationen abgeleitete Navigationen und als Attribute definierte Aufzählungen. Modelltypen werden mit dem Namen bezeichnet, den sie auch im UML-Modell haben. Gültige Modelltypen sind die Typen, Klassen, Interfaces, Assoziationen, Akteure, Use Cases und Datentypen des UML-Modells. Die Attribute einer Klasse im UML-Modell sind ebenfalls gültige Attribute in OCL, wenn die entsprechende Klasse ein gültiger Typ in OCL ist. Wie mit den Attributen verhält es sich auch mit den Operationen und Methoden. Auch sie können in OCL unter Berücksichtigung einer wichtigen

Restriktion benutzt werden. OCL ist eine seiteneffektfreie Sprache. Aus diesem Grund sind in OCL-Ausdrücken nur solche Operationen erlaubt, die den Zustand aller anderen Objekte nicht verändern; solche Operationen heißen *query operations* und liefern nur einen Rückgabewert ohne Veränderungen vorzunehmen. Die dritte Eigenschaft der Modelltypen ist die aus den Assoziationen abgeleitete Navigation. Der Name einer Navigation ist der Rollenname am anderen Ende der Assoziation. Fehlt ein solcher Rollenname, so ist der Name der Navigation der Name des Typen am Ende der Assoziation in Kleinbuchstaben. Das Ergebnis einer solchen Navigation ist entweder ebenfalls wieder ein Modelltyp oder eine Kollektion von Modelltypen, in der Regel Bags. Besteht das Ergebnis nur aus einem Wert, so ist der Ergebnistyp ein Modelltyp; bei einem vielfältigen Ergebnis ist der Ergebnistyp eine Kollektion. In UML ist es möglich, ein Modell in Packages einzuteilen. Diese Einteilung kann auch in OCL fortgeführt werden. Die Assoziationen zwischen Klassen und Interfaces können in der Regel, wie schon zuvor beschrieben, benutzt werden, da in verschiedenen Packages normalerweise nicht Klassen mit gleichen Namen enthalten sind.

Im Gegensatz zu einer Programmiersprache wie z.B. Java ist in UML auch eine Mehrfachvererbung möglich und wird häufig benutzt. Dabei kann es vorkommen, dass Attribute mit identischen Namen aber unterschiedlichen Eigenschaften von verschiedenen Oberklassen geerbt werden. Ein weiterer Typ, der in UML oft als Typ für Attribute verwendet wird, ist der Typ *Enumeration*, der in einem UML-Modell wie folgt aussieht:

enum{ value1, value2, value3 }

Die in einem *Enumeration*-Type aufgeführten Werte können auch in einem OCL-Ausdruck benutzt werden. Dazu muss ihnen das '#'-Symbol vorangestellt werden.

2.7.5 Typkonformität

OCL ist, wie schon zuvor beschrieben, eine getypte Sprache und die Basistypen sind in einer Typenhierarchie organisiert. Daraus ergibt sich Konformität der verschiedenen Typen zueinander; zum Beispiel ist es nicht möglich, einen Integer-Wert mit einem Boolean oder String zu vergleichen. In einem gültigen OCL Ausdruck

müssen alle Typen konform sein. Die Konformität kann durch folgenden Sachverhalt beschrieben werden: ein Typ *t1* ist konform zu einem Typ *t2,* wenn eine Instanz von *t1* an jedem Platz eingesetzt werden kann, wo eine Instanz *t2* erwartet wird. Dazu gibt es folgende Konformitätsregeln für die Typen in Klassendiagrammen:

- Jeder Typ ist konform zu seinem Supertyp.
- Die Konformität ist transitiv, das heißt, ist *t1* konform zu *t2* und *t2* konform zu *t3*, so ist auch *t1* konform zu *t3*.
- Die Typkonformität der value types sieht zum Beispiel wie folgt aus:

Typ	**Konform zu/ ist Untertyp von**
Set	Collection
Sequence	Collection
Bag	Collection
Integer	Real

2.7.6 Vererbungssubtypen

Da eines der Hauptmerkmale objektorientierter Programmiersprachen die Vererbung ist, der UML-Standard jedoch keine Regeln für die Vererbung von Constraints aufweist, muss beim Gebrauch von OCL-Ausdrücken in einem UML-Modell noch das Problem der Vererbung angesprochen werden. Im Falle von Invarianten, die für eine Klasse formuliert wurden, gelten diese auch für die Unterklassen. In der Unterklasse kann die Invariante ebenso neu definiert werden, wobei jedoch immer gilt, dass die Unterklasse die Invariante nur verstärken, nicht jedoch abschwächen kann. Dasselbe Problem ergibt sich bei den Pre- und Postconditions. In diesem Falle erscheint die Regelung vernünftig, dass durch eine Neudefinition der Precondition in einer Unterklasse diese Bedingung nur abgeschwächt, nicht aber verstärkt werden kann. Durch eine Neudefinition der Postcondition in einer Unterklasse hingegen wird diese Bedingung nur verstärkt, nicht aber abgeschwächt.

2.7.7 Fazit

Ein großer Vorteil von OCL ist seine Präzision und Eindeutigkeit, da dadurch die Kommunikation sehr erleichtert wird ohne zu Missverständnissen zu führen. Außerdem zeichnet sich OCL durch seine Erweiterbarkeit aus. Teilweise ist OCL jedoch schwer lesbar und/oder erlernbar. Die häufig unvollständige oder mehrdeutige Spezifikation der Semantik von OCL erschwert zudem das Verständnis.

2.8 Betrachtung bestehender Werkzeuge zum Testen von Java-Klassen

2.8.1 Projekt BlueJ

Sollen Klassen getestet werden, muss ein Objekt einer Klasse erzeugt werden, um seine Methoden auszuführen. Um ein Objekt einer zu testenden Klasse zu erzeugen, wird ein Testprogramm geschrieben, welches die Klasse testet, indem es alle ihre Methoden aufruft. Dieses Testprogramm muss die Ergebnisse der Methodenaufrufe für den Entwickler darstellen können. Darum stellt das Testprogramm möglicherweise mehr Entwicklungsaufwand für den Entwickler dar, als die zu testende Klasse. Ein weiterer Nachteil ist, dass das Testprogramm selbst Fehler enthalten kann. Außerdem bleibt der interne Zustand des zu testenden Objektes oft verborgen. Darum muss in der zu testenden Klasse ein Code eingebracht werden, der in einem Debugmodus Ausgaben über den internen Zustand macht. Dieser zusätzliche Code kann wiederum Fehler haben. Außerdem wird es bei einer langen Liste von Ausgaben schwierig, diese noch zu interpretieren.

Um die oben genannten Nachteile zu übergehen, sollte idealerweise kein Testcode geschrieben werden. Die Aufgabe eines Testprogramms sollte die Entwicklungsumgebung einer Programmiersprache übernehmen. Dabei müssen folgende Punkte unterstützt werden. Die Entwicklungsumgebung muss eine Instanz der zu testenden Klasse erstellen, geeignete Parameter an den Konstruktor liefern und Methoden der zu testenden Klasse aufrufen können. Außerdem müssen die Ergebnisse und der interne Zustand der Instanz angezeigt werden können.

BlueJ ist eine objektorientierte Sprache und eine Entwicklungsumgebung, die hauptsächlich als Lernumgebung für Studenten in den ersten Semestern entwickelt wurde und das Testen von Klassen unterstützt.

Sie bietet die Möglichkeit, eine Instanz von einer durch einen Programmierer geschriebenen Klasse zu erzeugen. Nach der Eingabe der Parameterwerte für den Konstruktor kann der Entwickler jede seiner Methoden aufrufen und wird dabei eventuell wieder nach Parametern gefragt. Die Ergebnisse einer Methode werden dann in einem Dialogfenster dargestellt.

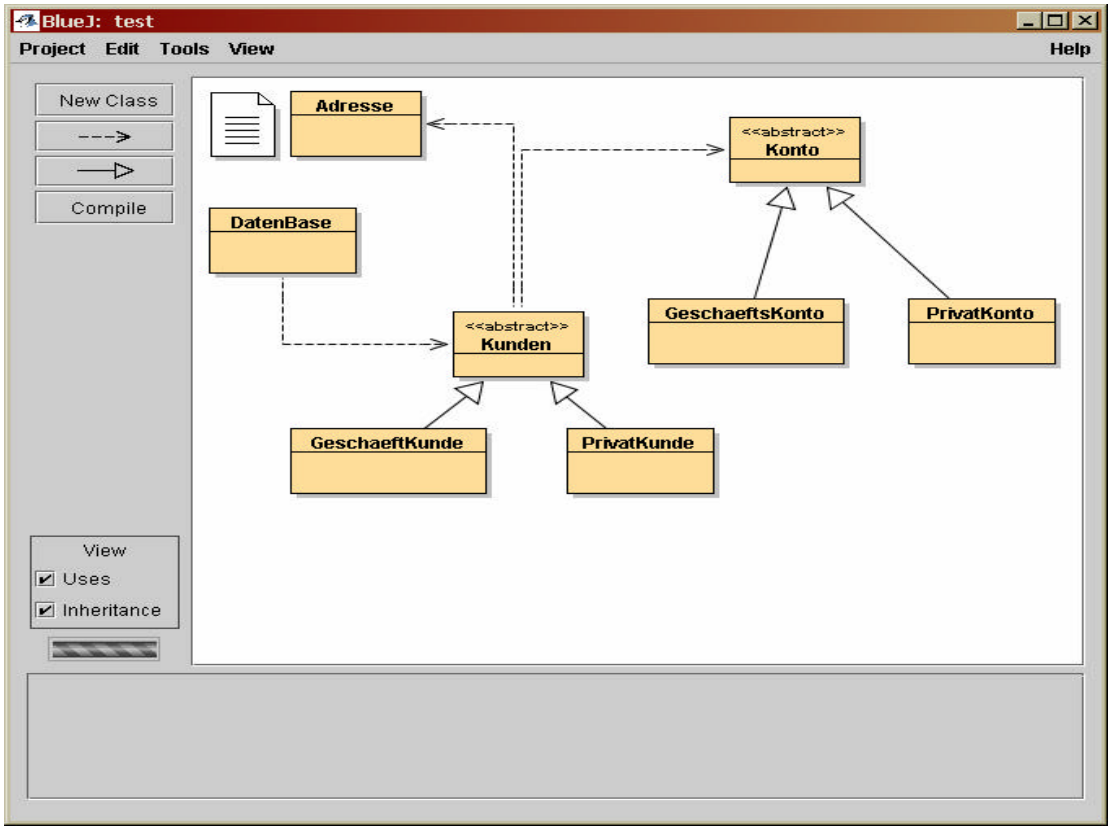


Abb. 2.3: Das BlueJ Hauptfenster

BlueJ stellt eine grafische Umgebung zur Verfügung in deren Hauptfenster (Abbildung 2.3) jede Klasse und ihre Beziehung zu den anderen Klassen dargestellt wird. Es kann der Quellcode der Klasse in einem Editor anzeigt und eine Klasse oder das ganze Projekt kompiliert werden. Das Programm unterstützt überdies die interaktive Erzeugung einer Instanz einer Klasse. Notwendige Angaben wie Parameterwerte oder Bezeichnungen lassen sich bequem eingeben.

Des weiteren kann der interne Zustand eines Objektes betrachtet oder dieses wieder gelöscht werden. Rückgabewerte von Methoden werden in einem eigenen Dialogfeld angezeigt. Objekt-Variablen sind mit ihrem Namen, ihrem Typ und ihrer Ausprägung dargestellt. Ist eine Variable ein Objekt, so wird sie als Referenz angezeigt, welche mittels eines Shortcuts über die *Object bench* separat beobachtet

werden kann, ohne jedes Mal mehrere Referenzobjekte aufrufen zu müssen. Das gilt jedoch nur für Objekte, die nicht vom Entwickler interaktiv erzeugt wurden, sondern über den Programmcode selbst, das heißt intern. [BLUEJ]

BlueJ bietet die Möglichkeit, alle erzeugten Objekte, Methodenaufrufe und Rückgabewerte sowie Textein- und –ausgaben zu dokumentieren, indem alle Interaktionen als Text gespeichert werden können.

Ein großer Vorteil von BlueJ liegt somit darin, dass nicht mehrere Klassen implementiert sein müssen, um diese testen zu können. Ein Projekt kann schrittweise um Klassen erweitert werden, indem bereits vorhandene vorab getestet werden. Erst wenn diese fehlerfrei funktionieren, werden weitere Klassen hinzugefügt.

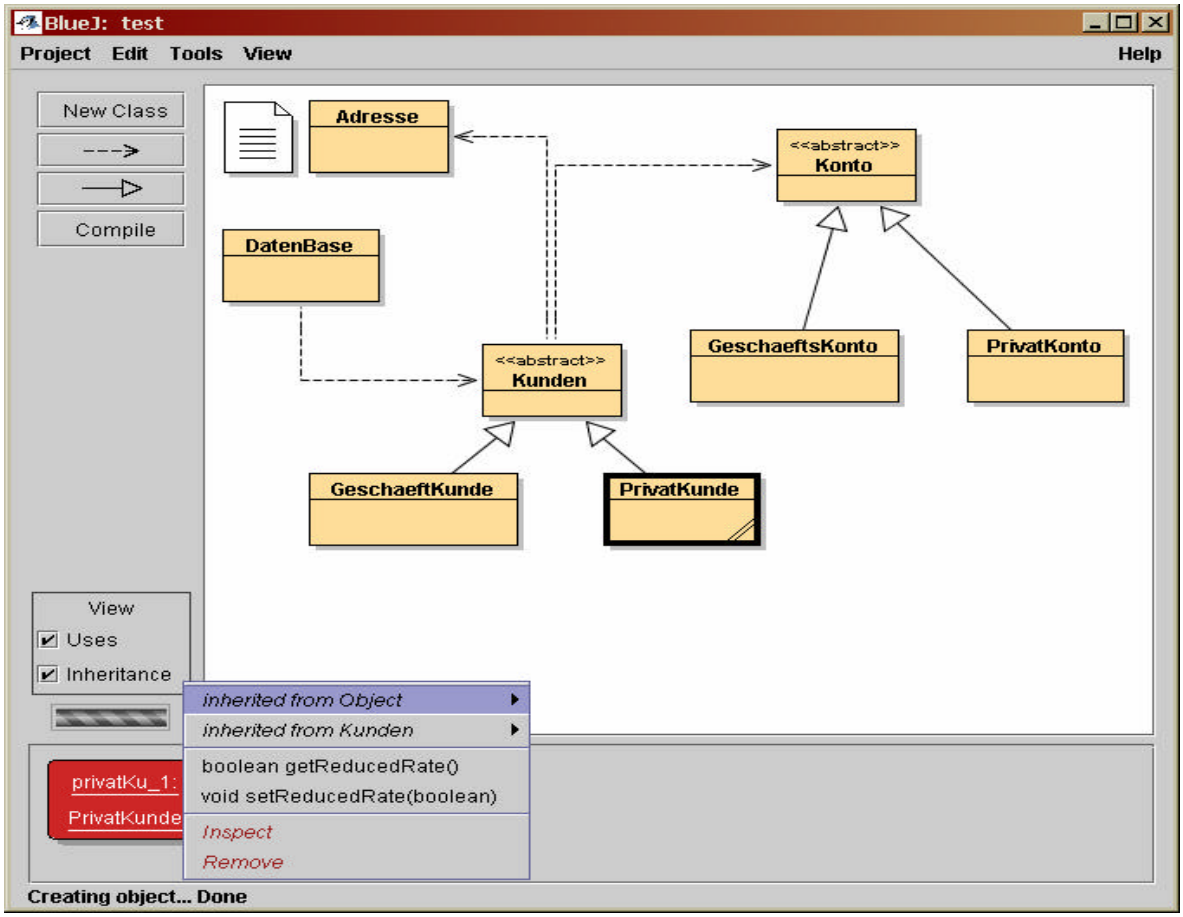


Abb. 2.4: Ein Objekt auf der Object Bench

2.8.2 JUnit

Das JUnit Testframework wurde von Erich Gamma und Kent Beck für verschiedene Sprachen (Java, Smalltalk, C++) entwickelt. Es stellt einen einheitlichen Rahmen zur Spezifikation, Implementation und Organisation, Ausführung und Kontrolle des Tests zur Verfügung und bietet die Möglichkeit, einzelne Testfälle unabhängig von anderen zu testen und Testfälle beliebig zusammenzufassen. Weiterhin werden Testergebnisse übersichtlich und getrennt nach Anwendungscode und Testcode dargestellt.

Das Design des JUnit Testframeworks besteht nur aus sehr wenigen Klassen, da auf Einfachheit sehr großen Wert gelegt wurde.

Das Kernfragment in JUnit ist die abstrakte Klasse TestCase. Diese bildet einen Rahmen, in dem die Testfälle implementiert werden. Hierzu wird eine neue Klasse von Testcase abgeleitet. In dieser Klasse wird pro Testfall eine neue Methode hinzugefügt. Per Konvention beginnt der Name dieser Methode mit *test*, gefolgt von einem kurzen Bezeichner, der angibt, was getestet wird (z.B. testKonto). Am Ende des Testcodes wird das Ist-Ergebnis mit dem Soll-Ergebnis verglichen und das Ergebnis (korrekt/falsch) mittels der geerbten Methode *assert* oder *assertEquals* an das Framework zurückgeliefert.

Die TestCase-Klasse besteht aus folgenden Methoden:

- *run(TestResult)*: Diese Methode ruft die nachfolgenden Methoden in der gegebenen Reihenfolge auf, und wird dazu benutzt, einen Test zu starten.
- *setUp():* In dieser Methode werden alle benötigten Instanzen (z.B. zu testende Klassen) erzeugt.
- *runTest()*: In dieser Methode wird der eigentliche Test durchgeführt. Dazu wird die Methode, die in der Instanzvariable *fName* enthalten ist, mittels des Reflection-API aufgerufen. Dies hat den Vorteil, dass nicht für jede Test-Methode eine eigene Klasse geschrieben werden muss, sondern nur der Namen der Methode in *fName* gespeichert wird.
- *tearDown():* In dieser Methode werden etwaige Ressourcen wieder freigegeben.

Die Ausführung der Testfälle einer Testcase-Klasse wird durch eine SubKlasse, die die Methode runTest() enthält, angestoßen, da diese Klasse abstrakt ist. Diese Sub-Klasse benutzt Javas Reflektionsmechanismus, um alle Methoden, deren Name mit _test beginnt, aufzurufen. Diese Methode kann auch überschrieben werden, um z.B. Testfallmethoden mit anderen Namen aufzurufen.

Die Klasse TestResult enthält das Ergebnis eines Tests. JUnit unterscheidet zwischen Fehler (error), das heißt Java-Ausnahmen, und Fehlschlägen (failures), die auftreten, wenn eine Methode einen falschen Wert berechnet.

Um nicht nur einen Test ablaufen zu lassen, sondern mehrere Tests gleichzeitig durchführen zu können, wurde die Klasse *TestSuite* eingeführt. In dieser Klasse können mehrere Tests gespeichert werden. Da diese Klasse die Schnittstelle *Test* auch implementiert, können in einer *TestSuite* rekursiv weitere Testsuites gespeichert werden. Somit ist es möglich, dass mehrere Entwickler eigene TestSuites schreiben, diese dann in einer Klasse zusammenfassen und alle Tests gemeinsam durchführen.

Die Klasse TestRunner dient zum Start und zur Ergebniskontrolle der Tests. Hierzu wird eine interaktive graphische Benutzeroberfläche (JUnit.awtui.TestRunner, JUnit.swing.TestRunner) sowie eine textbasierte Variante für den Batchmodus (JUnit.textui.TestRunner) zur Verfügung gestellt. [Bec2000]

Um einen tieferen Einblick in die Funktionsweise von JUnit zu ermöglichen, wird hier ein Beispiel gegeben. Die Klasse Adresse speichert die Adresse von Bankkunden:

```
public class Adresse
{
    // instance variables
    private String street, town, phone, fax, email;
    private int postCode ;

    /**
     * Constructor for objects of class Adresse
```

```
 */
public Adresse()
{
   // initialise instance variables
   this( “”, 0, “”, “”, “”, “”);
}

public Adresse (String street, int   ostcode, String town)
{
   this(street,   ostcode, town, “”, “”, “”);
}

public Adresse (String street, int   ostcode, String town, String phone, String fax,
String email)
{
   this.street = street;
   this.postCode = postCode ;
   this.town = town;
   this.phone = phone;
   this.fax = fax;
   this.email = email;
}

//----------------------------------- property methods
…
public String getPhone ()
{
   return (this.phone);
}
public void setPhone ( String phone)
{
   this.phone = phone;
}
…
```

```
    //---------------------------------------- public methods
    public String toString ()
    {
        return street + “\n” +
                ostcode+”” + “\n” +
            town + “\n” +
            phone + « \n » +
            fax + « \n » +
            email + „\n“;
    }
}
```

Um diese Klasse testen zu können, wird das JUnit-Framework benutzt und eine neue Klasse erzeugt, welche die Klasse *TestCase* erweitert:

```
public class AdresseTest extends TestCase
{
//…
  /**
   * Test method: void setPhone(String)
   */
  public void testSetPhone() {
    private Adresse adresse1 = new Adresse (“t”, 111, “t”, null, “”, “”); ..... //(1)
    private Adresse adresse2 = new Adresse (“t”, 111, “t”, “030-5424799”, “”, “”);
    adresse1.setPhone(“030-5424700”) .....
    adresse2.setPhone(“030-5424700”) .....
```

Dieser Testfall besteht aus:

(1) Code zum Erzeugen der benötigten Objekte,
(2) Code zum Beschreiben des zu testenden Verhaltens und
(3) Code zur Verifikation des Testergebnisses.

Um den Test durchführen zu können, muss zuvor noch definiert werden, wie ein individueller Test und eine Testsammlung gestartet werden.

JUnit stellt zur Durchführung einzelner Tests sowohl statische als auch dynamische Tests zur Verfügung.

Bei der statischen Variante wird die Methode runTest() überschrieben, indem eine anonyme Klasse erzeugt wird:

```
TestCase test= new AdresseTest (“simple setPhone”) {
  public void runTest() {
        testSetPhone ();
  }
};
```

Die dynamische Variante benutzt das Reflection API von Java:

```
        TestCase test= new AdresseTest (“testSetPhone”);
```

In diesem Fall wird die Methode, die als Argument übergeben wird, ausgeführt. Ein Überschreiben der *runTest()*-Methode ist nicht mehr notwendig. Nachteil dieser Variante ist, dass, sollte es die Methode nicht geben, erst zur Laufzeit eine *NoSuchMethodException* auftritt.

Um beide Testfälle gemeinsam laufen lassen zu können, wird zuletzt eine TestSuite definiert. Dazu wird in der Klasse *AdresseTest* eine statische Methode geschrieben (Es wird die dynamische Variante verwendet):

```
public static Test suite() {
   TestSuite suite= new TestSuite();
   suite.addTest(new AdresseTest (“simple setPhone “));
   suite.addTest(new AdresseTest (“testSetPhone “));
   return suite;
}
```

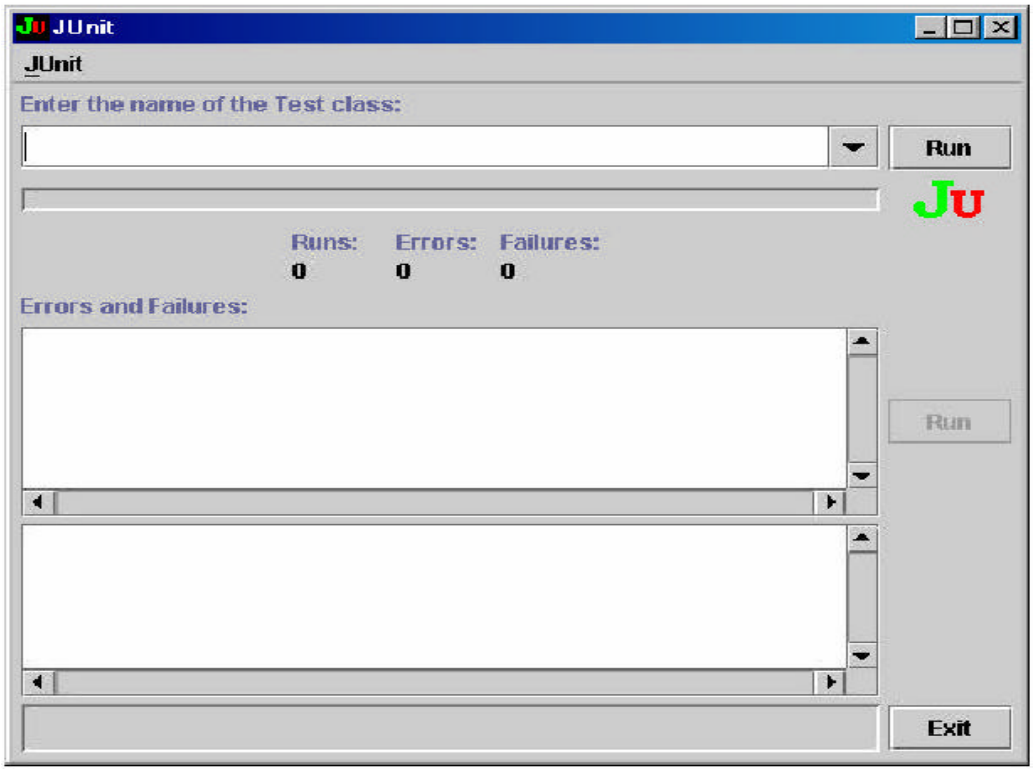


Abb. 2.5: Benutzerschnittstelle JUnit

Um die Tests zu starten, muss in das Eingabefeld der Name der Klasse eingetragen werden. Der Fortschritt des Tests wird in dem *Progress Bar* angezeigt. Verlief der Test nicht erfolgreich, wird dieser Balken rot und der Fehler in der Liste unterhalb angezeigt. [JUNIT]

JUnit ermöglicht damit dem Entwickler, selbstgeschriebene, beziehungsweise fremde Klassen testen zu können. Leider müssen die Testfälle selbst geschrieben werden, jedoch wird die Durchführung in großen Teilen unterstützt. Das Hauptproblem bei JUnit ist, dass der Zugriff auf private Teile der zu testenden Klasse nur eingeschränkt möglich ist (ab JDK 1.2 durch Reflection). Dadurch wird die Initialisierung, Kontrolle und insbesondere die Nutzung von Kontrollabschnitten erschwert.

2.8.3 Fazit

Die beiden vorgestellten Tools haben mehrere Vor- und Nachteile. Der gemeinsame Vorteil von JBlue und JUnit ist, dass ein Projekt schrittweise um Klassen erweitert werden kann. Es besteht damit kein Zwang, erst mehrere Klassen zu implementieren. Außerdem besitzt JBlue noch die Vorzüge, dass es nicht nötig ist, ein Testprogramm zu entwerfen und dass der interne Zustand des zu testenden Objekts nicht verborgen bleibt, so dass der Tester sofort ein Feldback über das Testobjekt erhält. Ein Vorteil von JUnit hingegen ist die schnelle Einsetzbarkeit ohne großen Lernaufwand aufgrund der Einfachheit des JUnit-Frameworks.

Der größte Nachteil von beiden Tools ist, dass die Testfälle selbst bestimmt und die Ausgabedaten einer Methode ebenfalls eigenhändig mit den Solldaten verglichen werden müssen, um so auf die Korrektheit zu schließen. Bei JUnit müssen zusätzlich noch die Testfälle selbst geschrieben werden.

Der Vorteil von JBlue, den internen Zustand des zu testenden Objektes darzustellen, wird in dem zu entwickelnden Testtreiber weiterverfolgt werden.

3 Anforderungsanalyse für den Testtreiber

Der zu entwickelnde Testtreiber soll es ermöglichen, kompilierten Java-Code ohne zusätzlichen Entwicklungsaufwand auszuführen. Dabei werden Instanzen der zu testenden Klasse erstellt und geeignete Parameter an den Konstruktor und die Methoden geliefert. Außerdem können der interne Zustand der Instanz und die Ergebnisse der Methodenaufrufe mittels eines Objektmonitors angezeigt werden.

3.1 Anforderungen

Um die Spezifikation einzubeziehen, soll der zu entwickelnde Testtreiber mittels einer vorliegenden Klassenspezifikation, die durch OCL beschrieben ist, die Invariante der Klasse nach jeder Instanzierung des Testobjekts und nach Ausführung seiner Methoden überprüfen. Ebenso werden die Nachbedingungen (post condition) nach der Ausführung der zugehörigen Methode überprüft, um eine Rückantwort über die Verletzung der Invariante beziehungsweise der Nachbedingungen geben zu können. Zusätzlich bietet der Testtreiber die Möglichkeit des Regressionstests.

Da das wesentliche Merkmal dieses Testtool die Simulation von Interaktionen des Anwenders „von außen“ ist (es bedarf dazu keinerlei Veränderung an der zu testenden Klasse), handelt es sich demnach um ein GUI Capture/Playback Testtool. [Bou1997]

Die Testtreiberarchitektur wird nach dem Model-View-Controller-Konzept gestaltet werden. Ein Vorteil dadurch ist, dass das Erstellen von mehreren Views beziehungsweise das spätere Hinzufügen von Views erheblich erleichtert wird. Durch eine klare und durchgängige Struktur wird die Wartbarkeit des Testtreibers erhöht. Ebenso gesteigert wird die Wiederverwendbarkeit der einzelnen Klassen. Der View beziehungsweise der Controller können dynamisch zur Laufzeit getauscht werden. Außerdem wird durch die Separation gewährleistet, dass bei unverändertem Interface nach einer Änderung in einer der Komponenten nur diese getestet werden muss.

3.2 Fähigkeiten und Einschränkungen des Prototyptesttreibers

3.2.1 Fähigkeiten

Die zu testende Klasse wird vom Testtreiber geladen. Danach analysiert und listet der Testtreiber die deklarierten Konstruktoren, Methoden und Variablen der Testklasse auf, außerdem werden alle implementierten Oberklassen und Interfaces angezeigt. Nachdem die zugehörige OCL-Datei der zu testenden Klasse ebenfalls geladen wurde, überprüft der Testtreiber sofort die Korrektheit der Syntax aller OCL-Ausdrücke und weist die Invarianten, die Vor- und die Nachbedingung an die jeweils zugehörigen Konstruktoren beziehungsweise Methoden zu. Parameter der Konstruktoren und Methoden können mittels Random und anhand der Vorbedingungen der Konstruktoren und Methoden automatisch erzeugt werden. Der Tester kann die Testfälle manuell konfigurieren, indem er die Reihenfolge des Aufrufs der Methoden festlegt und er kann die Parameter zur Erzeugung des Testobjekts eingeben oder auswählen, da es besonders wichtig ist, dass die Parameter keine primitiven Typen sind.

Nach jeder neuen Instanzierung der Testklasse wird die Korrektheit der Invariante der Klasse überprüft, nach jeder Ausführung einer Methode kontrolliert das Testwerkzeug erneut die Invariante und die Nachbedingungen der Methode, wenn sie existieren. Die Information über den internen Zustand des zu erzeugenden Testobjekts, seine Änderung nach jedem Methodenaufruf, die eventuellen Rückgabewerte nach der Ausführung der Methoden sowie die aufgetretenen Exceptions werden angezeigt und in einer Protokolldatei gespeichert. Um den Mindestgrad an Überdeckung der Methoden zu garantieren, wird bei der automatischen Testfallerzeugung jede deklarierte Methode in der Testklasse mindestens einmal ausgeführt.

3.2.1 Einschränkungen

Die sinnvolle Reihenfolge des Methodenaufrufs ist Aufgabe der Benutzer, dafür bietet der Testtreiber die Möglichkeit der manuellen Konfiguration. Methoden der Oberklasse, die nicht geändert wurden, werden nicht erneut getestet. Es wird vorausgesetzt, dass die Oberklasse vorher getestet ist und ihrer Spezifikation entspricht. Es werden keine Stubs verwendet, wenn bei der Testklasse eine andere Klasse instanziert und deren Methode aufgerufen wird. In diesem Fall wird die Klasse erzeugt und ihre Methode aufgerufen. Die Auswertung komplexer OCl-Ausdrücke mit „ if … then … else … “ wird nicht unterstützt; es kann nur einfacher Kontext verarbeitet und ausgewertet werden, eine ausführliche Beschreibung darüber gibt es im Abschnitt Dokumentation.

4 Untersuchung der Verwendbarkeit der vorhandenen Techniken

4.1 Einsatz von OCL zum automatisierten Testen von Java-Klassen

Der Einsatz von OCL-Ausdrücken für den Klassentest und dessen Automatisierung ist sehr wichtig für den entwickelten Testtreiber, da OCL für die Spezifikation von Invarianten für Klassen und Typen in einer Klasse sowie zur Beschreibung von Vor- und Nachbedingungen von Operationen und Methoden verwendet werden kann. Die eindeutige Beschreibung der Definition von Beziehungen, Einschränkungen und Bedingungen durch OCL ermöglicht die Realisierung der Erzeugung von Testfällen und den Nachweis der Korrektheit der Spezifikation der Klasse und Methoden.

Um einen Eindruck über die Einschränkungen an Attributen einer Klasse durch OCL zu erhalten, wird folgendes Beispiel betrachtet:

```
package example;
public class Customer {
    private String name;
    private String dateOfBirth;
    private int age;
    private int cardNumber;

    public Customer(String name, String dateOfBirth, int age, int cardNumber) {
        this.name = name;
        this.dateOfBirth = dateOfBirth;
        this.age = age;
        this.cardNumber = cardNumber;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
```

```
    }
…
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }
    …
}
```

Die Klasse *Customer* hat unter anderem die Attribute *age* und *name* mit der Aussage, *„Kunden müssen mindestens 18 sein“* und „*Der Name des Kunden darf nicht leer sein*“. Laut Syntax der Invariante von OCL:

context *<class name>* ***inv****: <Boolean OCL expression>*

kann die Invariante wie folgt geschrieben werden:

context *Customer*
inv*: self.age >= 18*
inv*: self.name <> null*

Anhand der *Constraints* der Invariante kann der Testtreiber mittels Objektmonitor nach der Erzeugung des Objekts *Customer* sowie der Ausführung der Klassenmethode die Gültigkeit der Invariante überprüfen.

Um die Spezifizierung einer Methode zu bestätigen, bietet OCL die Benutzung von Vor- und Nachbedingungen. Ein einfaches Beispiel für die Benutzung von Vorbedingungen um Eingangsparameter zu erzeugen, ist eine Methode zur Berechnung der Division zweier Zahlen:

```
public class Calculate {
        …
        public double getDivisionResult (double a, double b) {
        return (a/b);
        }
        …
}.
```

Die Vorbedingungen der Methode *getDivisionResult* können wie folgt geschrieben werden:

context *Calculate:: getDivisionResult (a: double, b: double)*
pre*: b <> 0*
post*: result = a/b*

Durch die Zusicherung der Vorbedingung ist es möglich, zwei Testfälle zu erzeugen, davon einen mit einem Eingangsparameter, der die Vorbedingungen verletzt, in diesem Beispiel wäre das b = 0D. Damit kann die Ausnahme-Behandlung betrachtet und aufgezeichnet werden. Im zweiten Testfall hingegen verletzt der Eingangparameter die Vorbedingungen nicht, um das Verhalten der Methode im Normalfall zu prüfen.

Das folgende Beispiel dient zur Veranschaulichung der Benutzung von Nachbedingungen.

```
package example.primitivetypes;

public class Complex {

    private float real = 0.0F, imag = 0.0F;
    public Complex(float re, float im) {
        this.real = re;
        this.imag = im;
    }
    …
```

```
    public void add(Complex z) {
        real = real + z.real;
        imag = imag + z.imag;
    }

    …
}
```

Die entsprechen Nachbedingungen der Methode *add* sind:

context Complex::add(z : Complex)
pre : -- none
post: (self.real = self.real@pre + z.real) and (self.imag = self.imag@pre + z.imag)

Diese add-Methode bekommt als Eingangsparameter eine komplexe Zahl z. Nach der Ausführung der Methode wird der interne Zustand des erzeugten komplexen Objekts geändert, indem der Realteil und der imaginäre Teil der erzeugten komplexen Zahl um den Realteil der Zahl z erhöht werden. Der Testtreiber analysiert und wertet die Nachbedingung aus, um eine Aussage über Verletzung oder Nichtverletzung der Nachbedingung zu geben. Damit wird die Gültigkeit des erwarteten Ergebnisses nach der Ausführung der Methode entweder nachgewiesen oder widerlegt.

4.2 Model-View-Controller-Konzept und Java-Swing

Bei dem Model-View-Controller-Konzept [BAR2002] handelt es sich um das Prinzip, die Datenhaltung (Model), die Datenrepräsentation (View) und die Datenmanipulation (Controller) von einander zu trennen. Der Vorteil dieser Vorgehensweise ist, dass es zu einem späteren Zeitpunkt sehr leicht möglich ist, die vorhandenen Daten neu zu visualisieren, indem einfach nur der *View* ausgetauscht wird. Dabei müssen die datenverarbeitenden beziehungsweise datenmanipulierenden Teile des Programms nicht beachtet werden, da diese entkoppelt sind. Dies gilt natürlich gleichermaßen für *Controller* und *Model*.

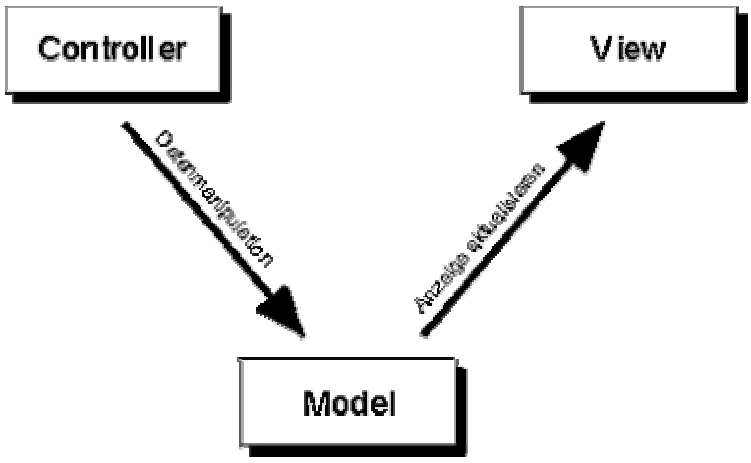


Abb. 4.1: Model-View-Controller

Es ist auch möglich mehrere *Views* zu verwenden, die auf das gleiche *Model* zugreifen. [AKS]

Die einzelnen Komponenten des Konzepts lassen sich folgendermaßen beschreiben:

Model (Datenhaltung, Anwendung)

Das Model ist das Datenmodell, das den aktuellen Zustand und das Verhalten des gesamten Systems repräsentiert.

View (Visualisierung, Darstellung)

View hat die Aufgabe, die Daten des Models auf irgendeine Art darzustellen (im Normalfall zu visualisieren). Wichtig ist hierbei, dass die Darstellung der Daten die einzige Aufgabe ist, die der View erfüllen muss.

Controller (Datenmanipulation, Steuerung)

Der Controller setzt die eingehenden Anforderungen (z.B. Eingaben von der Tastatur oder Messwerte eines Sensors) in Signale um, die das Model dazu veranlassen, die Daten entsprechen zu verändern.

Java-Swing bietet ein sehr komfortables Framework für die Erstellung von graphischen Oberflächen. Bei der Architektur von Swing wurde sehr stark auf das Model-View-Controller Konzept geachtet, zum Beispiel die Entkopplung der Daten von ihrer Darstellung beziehungsweise Manipulation. In einem solchen Fall reagiert das

entsprechende Objekt auf die Aktionen des Benutzers und visualisiert seinen entsprechenden Status

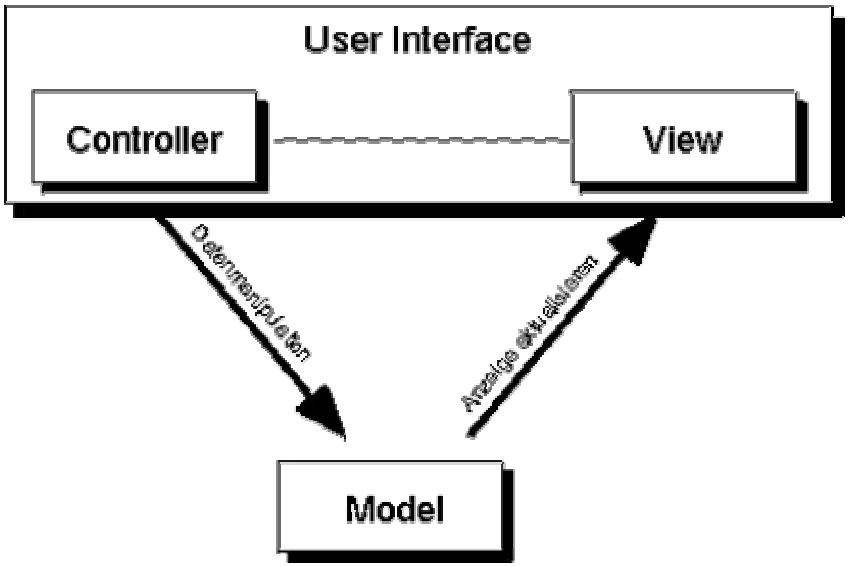


Abb. 4.2: Model-View-Controller mit Interface

4.3 Die Fähigkeiten von Java-Reflections in Hinblick auf den Test

Der Testtreiber, der zu entwickeln ist, muss in der Lage sein, den vorhandenen Bytecode einer Javaklasse zur Laufzeit des Testtreibers zu laden und auf seine Fähigkeiten hin zu untersuchen. Da mit Hilfe von Reflection-API nicht nur ein vollständiger Datentyp-Browser entwickelt, sondern auch eine Applikation geschrieben werden kann, welche andere Klassen analysiert und interpretiert (also Objekte kreiert und Methoden ausführt) ist es möglich, das gestellte Ziel mit diesem Package zu erreichen. Der Vorteil von Reflection ist gleichzeitig sein großer Nachteil: Reflection unterläuft die Typsicherheit, so dass viele Fehler erst zur Laufzeit auftreten und der Compiler keine Chance mehr hat, Fehler frühzeitig abzufangen. Da jedoch der Testtreiber lediglich in der Entwicklungsphase benutzt werden wird, ist dieser Nachteil unerheblich. Viel wichtiger ist, dass Reflection dem Testtreiber gestattet, auf alle Methoden, Konstruktoren und Attribute zuzugreifen. Da beim Testen ein schreibender Zugriff auf Attribute nicht notwendig ist, muss dieser nicht in dem Testtreiber implementiert werden. Wichtig ist aber, das Recht *suppressAccessChecks* (definiert in ReflectPremission) zu haben, da sonst der Zugriff auf Javaklassen durch den SecurityManager des JVM verboten wird.

Um ein besseres Verständnis im Hinblick auf die Implementierung zu bekommen, ist es an dieser Stelle angebracht, einen Überblick über das Package *java.lang.reflect* zu geben.

Mit Ausnahme der Klassen *Object, Class* und *Package* befinden sich alle Klassen für den hauptsächlichen Teil der Reflektions-Mechanismen im Subpackage *java.lang.reflect*

Ausgangspunkt des Reflections-Mechanismus ist *Class*. Diese Klasse stellt Werkzeuge für die Manipulation der Klassen zur Verfügung, hauptsächlich, um neue Objekte dieses Typs zu kreieren oder Klassen zu laden. Eine Referenz zur Class-Instanz wird erlangt durch Abfrage des Objekts mit Hilfe der getClass() Methode, mittels eines Klassenliterals (der Name der Klasse gefolgt von *class*), durch eine vom Reflection-Package zur Verfügung gestellte Methode, als Rückgabewert und anschließender Abfrage aller Klassen in der Hierarchie mittels *Class.getClasses(),* Nachschlagen des Objekts mittels des voll qualifizierten Namens (mit allen Packages) und der Class.forName() Methode.

Um die Konstruktoren einer Klasse zu erhalten, wird folgende Methode genutzt:

```
public Constructor[ ] getConstructors()
        throws SecurityException;
public Constructor getConstructor(Class[ ] pTypes)
        throws NoSuchMethodException,SecurityException;
public Constructor[ ] getDeclaredConstructors()
        throws SecurityException;
public Constructor getDeclaredConstructor(Class[ ] pTypes)
        throws NoSuchMethodException,SecurityException;
```

Die Methode getConstructors() liefert lediglich die *public*-Konstruktoren der betrachteten Klasse. Die Konstruktoren einer Oberklasse sind nicht erkennbar. Die Methode getConstructor() dagegen gibt nur den Konstruktor zurück, dessen Signatur zu den Class-Argumenten passt. Wird die Signatur nicht getroffen, wird eine *NoSuchMethodException* ausgelöst.

Feldinformationen geben folgende Befehle:

```
public Field[ ] getFields()
        throws SecurityException;
public Field getField(String name)
        throws NoSuchFieldException, SecurityException;
public Field[ ] getDeclaredFields()
        throws SecurityException;
public Field getDeclaredField(String name)
        throws NoSuchFieldException, SecurityException;
```

Die Methode getField() wird benutzt, um ein *public*-Feld zu finden, welches lokal deklariert oder geerbt werden kann. Die Methode getDeclaredField() findet die Felder der Klasse, falls diese in der Klassendeklaration enthalten sind, auch wenn sie nicht *public* sind.

Für Methodeninformationen gibt es folgende Zugriffsmethoden:

```
public Method[ ] getMethods()
        throws SecurityException;
public Method   getMethod(String name,Class[ ] parameterTypes)
        throws NoSuchMethodException,SecurityException;
public Method[ ] getDeclaredMethods()
        throws SecurityException;
public Method getDeclaredMethod(String name, Class[ ] parameterTypes)
        throws NoSuchMethodException,SecurityException;
```

Analog zu den Konstruktoren erfolgt die spezielle Auswahl einer Methode ebenfalls über ihren einfachen Namen und ihre Signatur.

```
public Class[ ] getClasses().
```

Die Class-Members werden durch folgende Operation erhalten:

```
public Class[ ] getClasses()
        throws SecurityException;
public Class[ ] getDeclaredClasses()
        throws SecurityException;
```

Die Methode getClasses() liefert alle *public* deklarierten inneren Klassen und Interfaces einer inspizierten Klasse einschließlich der geerbten. Die Declared-Variante liefert dann zusätzlich alle nicht *public* deklarierten, schließt aber die geerbten aus.

Wie bereits oben erwähnt, benötigen alle diese Methoden eine Sicherheitsüberprüfung bevor sie ausgeführt werden dürfen. Daher wird der Security Manager aufgerufen. Sollte kein Security Manager installiert sein, sind alle Methodenaufrufe erlaubt. Ein Security Manager wird natürlich den Zugriff auf *public* Informationen gestatten. Der Zugriff auf nicht *public* Informationen wird dagegen in der Regel durch einen Security Manager abgeblockt oder eine Security Exception geworfen.

Mit Hilfe der newInstance-Methode des Klassenobjekts kann eine neue Instanz (Objekt) der Klasse generiert werden. Diese Methode führt den argumentlosen Konstruktor der Klasse aus und liefert eine Referenz auf ein neues Objekt. Da diese Referenz als Objekt geliefert wird, muss sie noch entsprechend dem erwarteten Datentyp gecastet werden. Falls die Methode newInstance() falsch aufgerufen wird, kann sie unterschiedliche Exceptions werfen. Wenn die Klasse in Wahrheit ein Interface oder eine abstrakte Klasse ist, kann das Objekt nicht kreiert werden, das heißt, es wird eine *InstantiatonException* geworfen. Falls kein Zugriffsrecht auf den argumentlosen Konstruktor besteht, wird eine *IllegalAccessException* geworfen. Sollte die Security Policy das Kreieren neuer Objekte nicht gestatten, wird eine *SecurityException* geworfen. Ein *ExceptionInitializerError* wird geworfen, wenn die Klasse bereits initialisiert sein muss, bevor Reflection-Methoden darauf angewandt werden.

Die newInstance Methode kann lediglich den argumentlosen Konstruktor aufrufen. Sollte ein anderer Konstruktor aufgerufen werden, muss zuerst mit dem entspre-

chenden Class Objekt das relevante Constructor Objekt bestimmt und dann auf diesen Konstruktor die newInstance()-Methode mit den passenden Parametern angewendet werden. Zusammen mit den von der Member-Klasse geerbten Methoden gestattet es die Constructor Klasse, vollständige Informationen über die Deklaration eines Konstruktors zu erhalten und damit auch den Konstruktor zur Konstruktion neuer Objekte einzusetzen.

Zu den Besonderheiten der Methode newInstance() zählt unter anderem, dass, falls das Konstrukt Constructor.newInstance(...) mit Parametern verwendet werden soll, zuvor die Basisdatentypen gewrappt werden müssen. Außerdem kann *Class.newInstance()* nicht eingesetzt werden, um Objekte innerer Klassen zu kreieren.

Die Field Klasse definiert Methoden für die Abfrage des Datentyps eines Feldes und für das Setzen und Abfragen von Werten eines Datenfeldes. Zusammen mit den geerbten Member-Methoden kann damit alles über die Datenfeld-Deklaration herausgefunden oder die Datenfelder einer spezifischen Klasse oder Instanz manipuliert werden. Mit den get... und set... Methoden wird der Wert eines Feldes gesetzt oder bestimmt. Diese Methoden besitzen eine einfache Form, welche ein Objekt als Argument akzeptieren und ein Objekt liefern. Zudem gibt es spezifischere Methoden, welche direkt mit den Basisdatentypen umgehen können. Falls die Methode statisch ist, wird auch die Eingabe *null* als Objekt akzeptiert. Wenn auf ein Feld eines spezifizierten Objekts nicht zugegriffen werden kann und eventuell ein Zugriffsschutz definiert wurde, wird eine *IllegalAccessException* geworfen. Sollte das Objekt, welches als Argument übergeben wird, von einem anderen Typ sein als das zugrundeliegende Feld, wird eine *IllegalArgumentException* ausgegeben. Wenn der Zugriff auf ein statisches Feld erfolgen soll, kann es sein, dass das Feld zuerst initialisiert werden muss. Falls dies nicht geschieht, wird eine *ExceptionInitializerError* geworfen.

Die Method-Klasse zusammen mit der Member-Klasse gestatten es, vollständige Informationen über die Deklaration einer Methode zu gewinnen und diese Methoden auszuführen, sollte ein entsprechendes Objekt zur Verfügung stehen. *Public Class getReturnType()* liefert das Klassenobjekt für den Datentyp dieser Methode. Wenn

diese Methode als *void* deklariert wurde, liefert sie ein Objekt vom *void.class*-Typ. *Public Class[ ] getParameterTypes()* liefert ein Datenfeld mit Klassenobjekten. Diese werden in derselben Reihenfolge wie in der Methodendeklaration ausgegeben – jedoch nur jeweils eines pro Parameter. Falls die Methode keinen Parameter besitzt, wird ein leeres Array zurück geliefert. Die Methode *public Class[ ] getExceptionTypes()* liefert ein Datenfeld mit Klassenobjekten, je eines pro deklarierter Exception in der throws-Erweiterung der Klassendefinition. Falls keine Exceptions angegeben wurden, wird ein leeres Array geliefert. Mit dem Befehl *public Object invoke(Object onThis, Object[ ] args) throws IllegalAccessException, IllegalArgumentException, InvocationTargetException* wird die Methode aufgerufen, welche im Method-Objekt, das heißt im *onThis*-Objekt definiert ist. Die Parameter werden aus dem Array *args* bestimmt. Wenn die Methode zu einer statischen Klasse gehört, kann an Stelle von *onThis* auch *null* angegeben werden. Die Länge des *args*-Datenfeldes muss mit der Anzahl der Parameter der Methode übereinstimmen, da sonst eine *IllegalArgumentException* geworfen wird. Diese Fehlermeldung erfolgt ebenfalls, wenn *onThis* nicht ein Objekt der Klasse ist, zu der die Methode gehört. Sollten keine Zugriffsrechte auf diese Methode bestehen, wird die *IllegalAccessException* geworfen. Eine *NullPointerException* wird geworfen, falls *onThis* null ist. Wenn die Methode statisch ist, muss eine Instanz kreiert werden, da ansonsten ein *ExceptionInitializerError* geworfen wird. Beim Abbruch der Methode erfolgt die Fehlermeldung *InvocationTargetException*. Sollte die invoke-Methode mit Basisdatentypen aufgerufen werden, müssen diese als Wrapper-Objekte mit den entsprechenden Werten vorliegen. Auch die Rückgabewerte des Methodenaufrufes werden in Wrapper-Klassen geliefert. Falls diese Methode *void* liefert, wird „null“ zurückgegeben.

Die Klassen Field, Constructor und Method sind alle Unterklassen der Klasse *AccessibleObject*. Diese gestattet es, die Zugriffsmodifier einer Klasse zu überprüfen, beispielsweise *public* und *private*. In der Regel werden die Zugriffe auf Methoden durch bestimmte Zugriffsrechte und Modifier geregelt. Falls auf ein Datenfeld ohne Reflection nicht zugegriffen werden kann, kann auch kein Zugriff mit Reflection erfolgen.

Die Klasse Array liefert statische Methoden, mit deren Hilfe die Elemente des Arrays gesetzt und gelesen werden können. Auch das Erstellen eines Arrays mit der newInstance() Methode ist möglich. Hierfür werden zwei Methoden eingesetzt:

public Object newInstance(Class componentType, int length)

Diese Methode liefert eine Referenz auf ein neues Array der angegebenen Länge mit / für Komponenten des angegebenen Typs.

public Object newInstance(Class componentType, int[ ] dimensions)

Diese Methode liefert eine Referenz auf ein mehrdimensionales Array, mit Dimensionen gemäß der Angabe in der Methode und Datentypen als Elementen gemäß dem *componentType*. Sollte das dimensions-Array leer oder nicht korrekt definiert sein (beispielsweise eine Dimension aufweisen, die größer als erlaubt ist), wird die Fehlermeldung *IllegalArgumentException* geworfen.

Mit den oben genannten Fähigkeiten von java.lang.reflect ist es möglich, den internen Zustand des Testobjekts zu betrachten und im Objektmonitor darzustellen [ESS2001].

5 Implementierung eines Prototyps des Testtreibers mittels Java

Um einen Klassentest automatisch durchführen zu können, müssen die Testwerkzeuge verschiedenen Anforderungen entsprechen. Sie müssen fähig sein, Testdaten und Testfälle zu generieren, selbstständig Objektinstanzen zu kreieren und anschließend die zu testenden Methoden aufzurufen. Außerdem müssen sie den internen Zustand des Objekts nach jeder Instanzierung, die Änderung des internen Zustands bei jedem Methodenaufruf sowie die Ausgabewerte der Methoden, falls vorhanden, gegen die Klassenspezifikation überprüfen können.

Daher besteht der entwickelte Testtreiber aus drei wesentlichen funktionalen Teilbereichen. Für die Betrachtung des internen Zustands der getesteten Objekte und die Auswirkung der Ausführung der Methode auf den internen Objektzustand dient der Objektmonitor (siehe Abbildung 5.1).

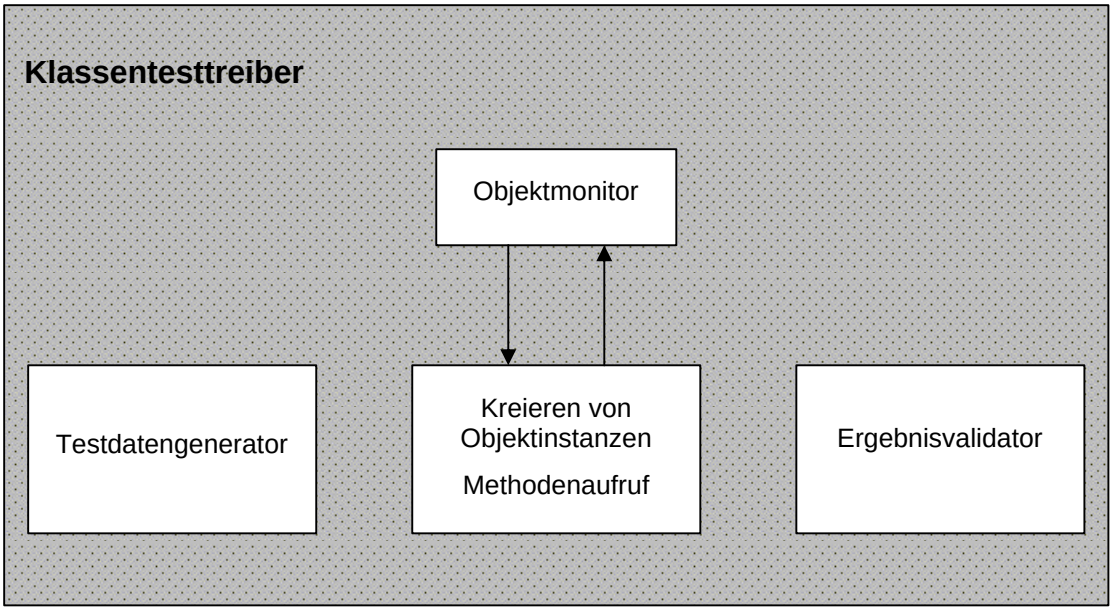


Abb. 5.1: Überblick über Teilbereiche des Klassentesttreibers

Der erste Bereich enthält den Testdatengenerator, der Testdaten und Testfälle anhand der Klassenspezifikation durch OCL generiert und Informationen der zu testenden Klasse durch java.lang.reflect ermittelt.

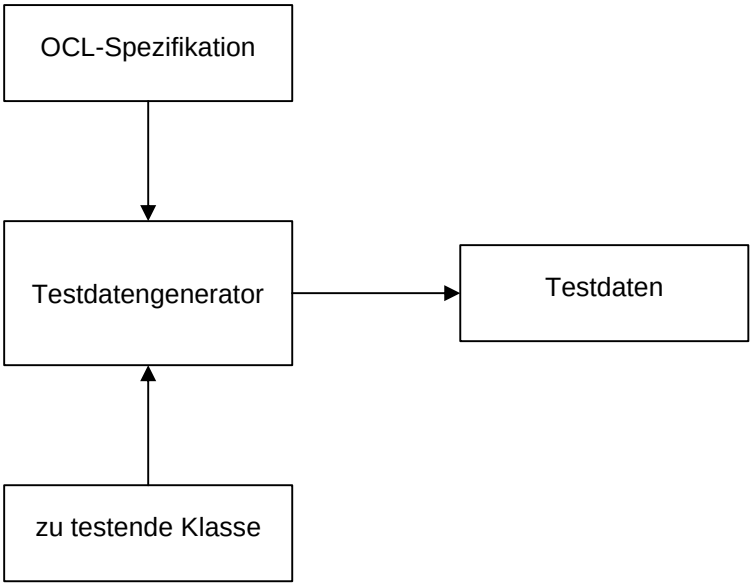


Abb. 5.2: Testdatengenerator

Der zweite Teilbereich ist für die Instanzierung des Testobjekts, sowie den Aufruf der einzelnen Methoden des Testobjekts zuständig. Da das Ziel das Testen von Java-Klassen ist, kann dies nur unter Zuhilfenahme von java.lang.reflect bewältigt werden.

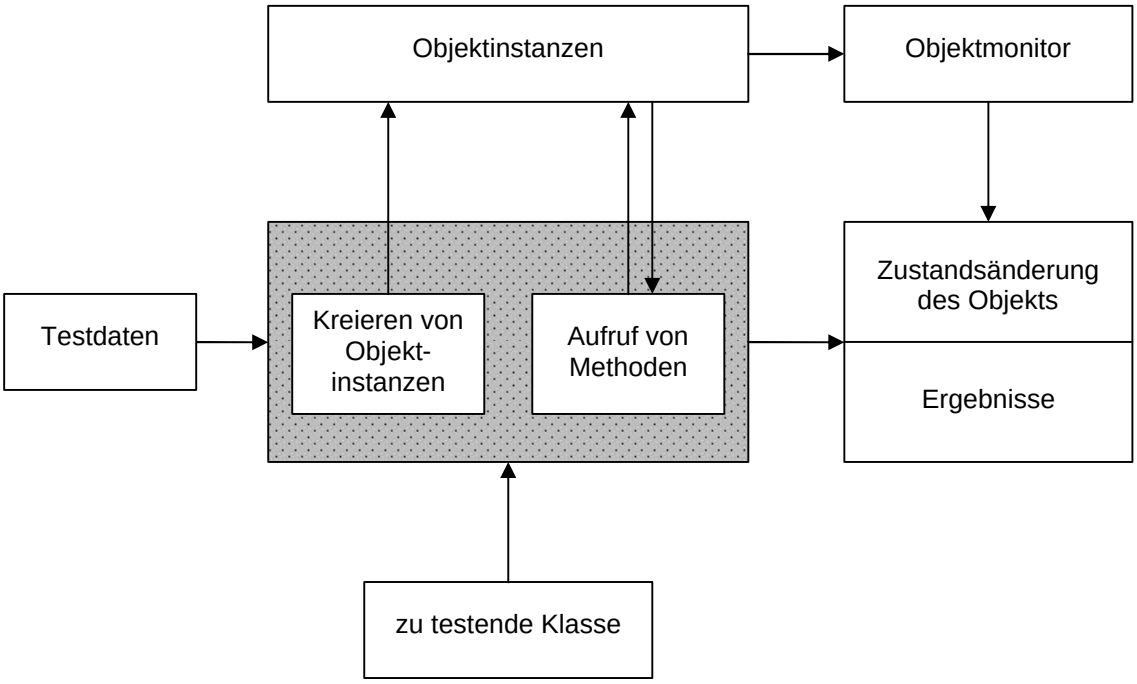


Abb. 5.3: Instanzierung und Ausführung

Die Validierung der internen Zustände des Testobjekts und der Ergebnisse nach dem Testen aller Methoden gegen die Klassenspezifikation ist die Aufgabe des dritten Bereichs. Des Weiteren gehört zu diesem Aufgabenbereich die Ausgabe aller

Testfälle und Testergebnisse in Form eines Testprotokolls beziehungsweise Testberichts.

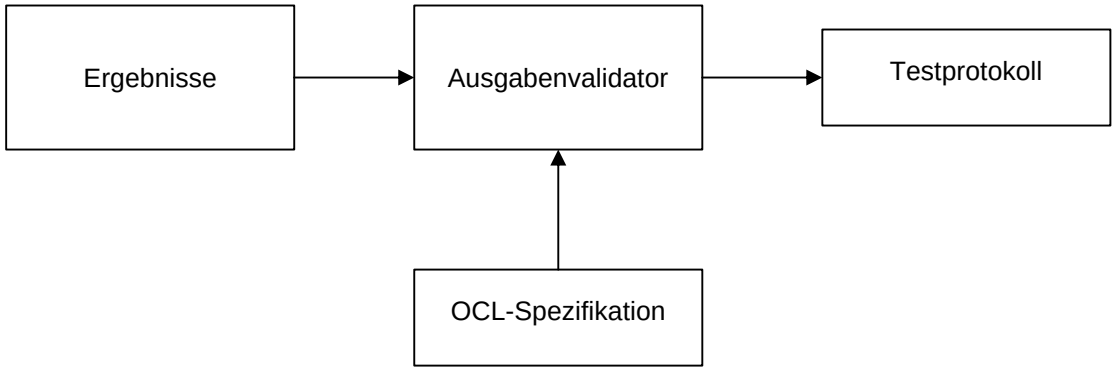


Abb. 5.4: Ausgabenvalidator

Der Aufbau und die Implementierung der einzelnen Teilbereiche wird in diesem Abschnitt noch einzeln erläutert werden, wobei alle wichtigen Pakete, Klassen und Methoden angesprochen werden. Zuvor muss jedoch noch die Art der Einbindung der Zusicherung an den Testtreiber diskutiert werden, da für den Klassentest die OCL-Constraints als Anhaltspunkte genutzt werden.

5.1 Einbeziehung der OCL-Spezifikation in den Testtreiber

Die Einbeziehung der Spezifikation wurde in der Anforderungsanalyse sowie in der Untersuchung der Verwendbarkeit der vorhandenen Techniken bereits angesprochen. Wie die Spezifikation durch OCL eingebunden werden kann, so dass der Testtreiber für den Benutzer komfortabel bleibt und die Komplexität der Implementierung des Testtreibers sich im Rahmen hält, wird in diesem Abschnitt erörtert.

Das Ziel dieser Diplomarbeit ist die Entwicklung eines Testtreibers, durch den die Testläufe durch OCL-Spezifikation weitestgehend automatisiert werden. Über die Korrektheit der einzelnen Testfälle muss der Benutzer nicht mehr selbst befinden. Daher wurde der Testreiber so konstruiert, dass die Invariante, sowie Vor- und Nachbedingungen automatisch überprüft werden und somit selbstständig Fehler festgestellt werden können. Eine Voraussetzung, um die Testläufe automatisieren zu können, ist jedoch die Kenntnis des Benutzers von OCL.

Es existieren mehrere Möglichkeiten, um die Einbeziehung der OCL-Spezifikation zu realisieren, es kann in Form von Textdateien, Quelltextkommentaren oder Java-Bytecode erfolgen. Jede diese Form muss folgende Bedingungen erfüllen: Erstens

sollten sie frei von Seiteneffekten auf den tatsächlichen Code sein und zweitens für den Benutzer keine Schwierigkeiten bei der Benutzung des Testtreibers bereiten, insbesondere wenn die Spezifikation nicht existiert.

Wenn die OCL-Spezifikation in Form von Java-Bytecode vorliegt, ist für die Entwicklung des Testtreibers ein Vorteil, da nur die Klassen-Datei herangezogen werden muss. Eine Implementierungsmöglichkeit ist es, alle Methoden, die für die Spezifikation zuständig sind, in eine innere Klasse der zu testenden Klasse zu schreiben. Damit wird diese innere Klasse nur bei Testphasen und nicht zur Laufzeit außerhalb der Testumgebung hochgeladen. Eine andere Möglichkeit besteht darin, alle diese Methoden als privat zu definieren, weil sie keine Zustandsänderung bewirken sollen. Ein Vorteil ist die einheitliche Sicht des Benutzers auf Programm und Spezifikation. Durch die Integration der Spezifikation in den Quellcode der zu testenden Klasse können die Syntaxfehler in der Spezifikation wie die Syntaxfehler im Quellcode behandelt werden. Dies ermöglicht es, nicht erst bei Laufzeit solche Fehler zu entdecken, sondern bereits beim Kompilieren. Das spart Nacharbeitszeit, da der Benutzer sofort ein Feedback auf seine Syntax erhält und diese zeitnah korrigieren kann.

Der Nachteil ist, dass der Benutzer zu viel Arbeit investieren muss, um die Spezifikation als Methode oder innere Klasse in den Quellcode einzubinden. Außerdem ist nicht sicher, dass die Freiheit von Seiteneffekten garantiert wird, weil die Spezifikationsmethoden möglicherweise eine Zustandsänderung herbeiführen und der Benutzer dies bei dem Kompilieren nicht bemerken kann. Darüber hinaus wirkt sich nachteilig aus, dass der Byte-Code der Testklasse vergrößert wird. Die Wartbarkeit der zu testenden Klasse ist ebenfalls schwieriger, weil der Überblick über die eigentlichen Methoden erschwert wird.

Um die Forderung zu garantieren, dass sich die Spezifikation seiteneffektfrei zum Klassen-Quellcode verhält, ist es vorteilhafter, die Spezifikation außerhalb des eigentlichen Quellcodes zu definieren. Dabei wird der Bytecode der zu testenden Klasse nicht unnötig vergrößert. Vorteilhaft ist, dass der Benutzer nur einen geringen Aufwand investieren muss, um die Spezifikation zu schreiben.

Ein Nachteil dieser Form besteht darin, dass die einheitliche Sicht des Benutzers auf Klasse und Spezifikation verloren geht, da die Klasse und ihre Spezifikation separat gespeichert werden. Manchmal ist eine einheitliche Sicht allerdings unerwünscht, weil die Spezifikation Aufgabe des Softwaredesigners ist, die Klassenimplementierung jedoch separat durch den Entwickler erfolgt. Zudem können die Syntaxfehler und logischen Fehler erst zur Laufzeit des Testtreibers und nicht sofort beim Kompilieren entdeckt werden.

Weil es das Ziel ist, einen Testtreiber so zu entwickeln, dass die Testläufe so weit wie möglich automatisiert stattfinden und der Aufwand für den Benutzer möglichst gering gehalten wird, wird die Spezifikation separat vom Quellcode behandelt. Dies kann in Form einer Textdatei oder in Form von Quelltextkommentaren erfolgen. Obwohl die Einbindung von Kommentaren die geeignetste Variante wäre, da Quelltextkommentare ohnehin existieren und diese somit „nebenbei“ geschrieben werden könnten, anstatt eine zusätzliche Datei zu erstellen, bedeutet dies jedoch einen erhöhten Aufwand für die Entwicklung des Testtreibers, da die Spezifikation zusätzlich extrahiert werden muss. Deshalb wurde für den zu entwickelnden Testtreiber die Form der separaten Textdatei gewählt.

In Abschnitt 4.1 wurde bereits erwähnt, dass nach jeder Instanzierung des Testobjektes die Invariante der Testklasse überprüft werden muss, um eine Aussage über den internen Zustand im Hinblick auf die Spezifikation zu machen. Weiterhin müssen vor jeder Ausführung der Methode die Eingangsparameter mit Hilfe der Vorbedingungen generiert und die Invariante erneut überprüft werden. Nach der Ausführung der Methode müssen nicht nur die Nachbedingungen, sondern auch die Invariante ausgewertet werden. Um diesen Anforderungen gerecht zu werden, müssen im Rahmen dieser Diplomarbeit eine Syntaxüberprüfung und Auswertungen von OCL-Ausdrücken in den Testtreiber implementiert werden. Nachfolgend wird ein Einblick in den Entwurf und die Implementierung gegeben.

Die OCL-Spezifikation wird als Textdatei durch den Testtreiber in den OCL-Parser geladen. Dieser Parser besteht aus zwei Teilen, dem Syntaxparser und dem funktionalen Parser. Basierend auf der OCL-Grammatik analysiert der Syntaxparser die Korrektheit der OCL-Ausdrücke, kann jedoch keine Typenüberprüfung oder

Prüfung der logischen Konsistenz durchführen. Dies wird nach korrekter Beendung des Syntaxparsers durch den funktionalen Parser durchgeführt. Da die Auswertung der logischen Konsistenz ein sehr komplexes Thema ist und um die Diplomarbeit im Rahmen zu halten, kann der funktionale Parser des entwickelten Testtreibers nur einfache logische Ausdrücke auswerten. Die daraus folgenden Einschränkungen des funktionalen Parsers werden im Anhang B erläutert.

Der funktionale Parser zerlegt die OCL-Spezifikation in zwei Teile. Zum einen extrahiert er die Invarianten, zum anderen die Vor- und Nachbedingungen der Methoden. Aus den Vorbedingungen werden Testfälle für die dazugehörige Methode generiert. Diese müssen dem Grundsatz entsprechen, dass die Testdaten einmal den Vorbedingungen entsprechen und mindestens einmal verletzen. Dadurch soll bestätigt werden, dass die Invarianten und Nachbedingungen funktionieren und dass sie zwischen korrekten und nicht korrekten Zuständen unterscheiden können.

Bei jedem Testfall werden die Objektinstanzen der zu testenden Klasse durch ihren Konstruktor erzeugt. Nach der erfolgten Erzeugung, wird die Invariante durch den funktionalen Parser validiert. Nach der Ausführung einer Methode werden die dazugehörigen Invarianten und Nachbedingungen (soweit existent) ausgewertet, um zu prüfen, ob die Methode mit den Spezifikationen übereinstimmt. Die Auswertung der Vorbedingungen mittels OCL erfolgt in dem Teilbereich „Testdatengenerator“, die Auswertung der Invarianten sowie die Auswertung der Nachbedingungen im Bereich des Ergebnisvalidators.

5.2 Testdatengenerator

Die zu testende Klasse ist nur als kompilierte Java-Klasse vorhanden und nicht als Quellcode. Somit ist das interne Verhalten und die Struktur dieser Klasse unbekannt, der Test kann nur durch die Prüfung des Ein- und Ausgabeverhaltens des Testobjektes erfolgen. Daher werden die Testdaten lediglich aus der Spezifikation abgeleitet. Dies sind alles Merkmale des Black-Box-Testens. Demzufolge ist die zur Anwendung kommende Testart bei dem entwickelten Testtreiber auch als solche zu sehen.

Weil eine der Hauptanforderungen ist, dass die Bedienung des Testtreibers ohne Klassenspezifikation möglich sein soll, müssen die Quellen der Testfälle aus der

Java-Klasse ableitbar sein. Das bedeutet, dass die Parametertypen sowie die Parameterreihenfolge des Konstruktors beziehungsweise der Methoden bekannt sein müssen. Java bietet hierfür die Möglichkeit mittels java.lang.reflect die interne Information der Testklasse darzustellen, dabei werden nicht nur die definierten Klassenattribute, Konstruktoren, Methoden und Ausnahmefälle sichtbar, sondern auch alle Oberklassen und Interfaces, die die zu testende Klasse implementiert. Alle diese Informationen stehen in Form von Java-Klassen zur Verfügung: java.lang.reflect.Class, java.lang.reflect.Constructor, java.lang.reflect.Method und java.lang.reflect.Field. Die einzige Voraussetzung für den Zugriff auf alle Konstruktoren, Methoden und Felder der zu testenden Klasse ist, dass der von JRE (Java-Lauftzeitsystem) verwendete SecurityManager diese Zugriffe nicht verbietet. Die folgende Implementierung in dem entwickelten Testtreiber dient als Beispiel zur Benutzung von java.lang.reflect zur Auflistung der Oberklasse.

```
public static Class[ ] getSuperClass(Class clazz) {
        OCLArrayList list = new OCLArrayList();
        Class subclass = clazz;
        Class superclass = subclass.getSuperclass();
        while (superclass != null) {
            list.add(superclass);
            subclass = superclass;
            superclass = subclass.getSuperclass();
        }
        if (null == list) {
            return null;
        } else {
            Class[] toReturn = new Class[list.size()];
            for (int i = 0; i < list.size(); i++) {
                toReturn[i] = (Class) (list.get(i));
            }
            return (toReturn);
        }
    }
```

Der Aufbau der zuständigen Pakete und Klassen im Testtreiber zur Analyse und Darstellung von internen Dateninformationen der zu testenden Klasse ist in Abbildung 5.5 wiedergegeben.

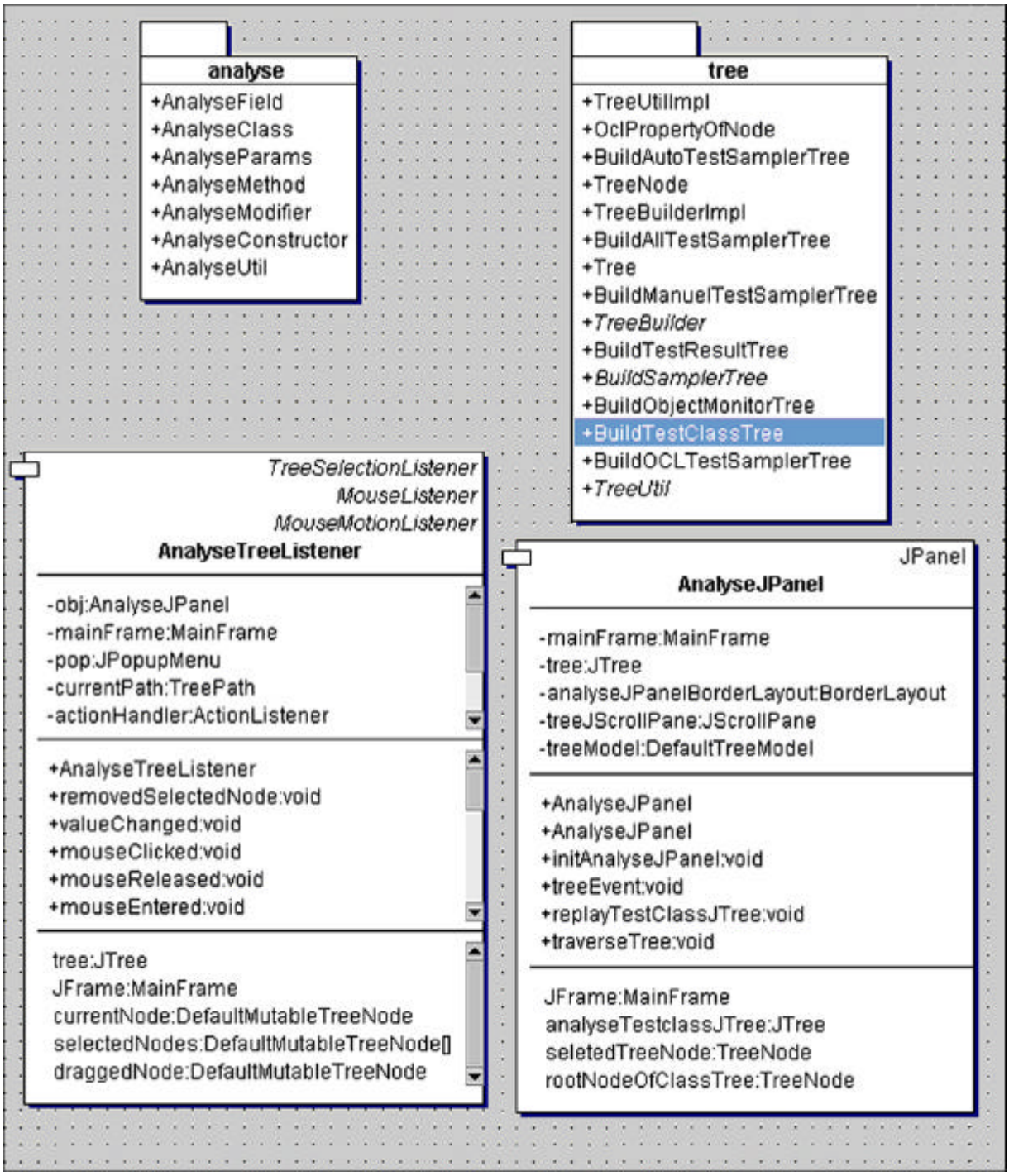


Abb. 5.5: Klassen und Pakete zur Klassenanalyse

Das Paket „Analyse“ beinhaltet Klassen zur Extraktion der Informationen aus der zu testenden Klasse, soweit dies durch Java-Reflection unterstützt wird. Als Speicher dieser extrahierten Informationen dient das Paket „Tree“. Das Besondere des Objekts „TreeNode“ ist, dass Namensgleichheit von Knoten zugelassen wird, sofern sich diese in unterschiedlichen Unterknoten befinden. Diese Besonderheit ist erforderlich, da oft die Parameter mehrerer Methoden einer Klasse zum Teil gleiche Bezeichnungen und gleiche Typen aufweisen.

Das UI-Paket enthält neben anderen Klassen auch die oben dargestellten Klassen „AnalyseTreeListener“ und „AnalyseJPanel“. Während „AnalyseJPanel“ zur Darstellung der erhaltenen Informationen dient, ist „AnalyseTreeListener“ für die Bedienung der Oberfläche durch den Benutzer zuständig.

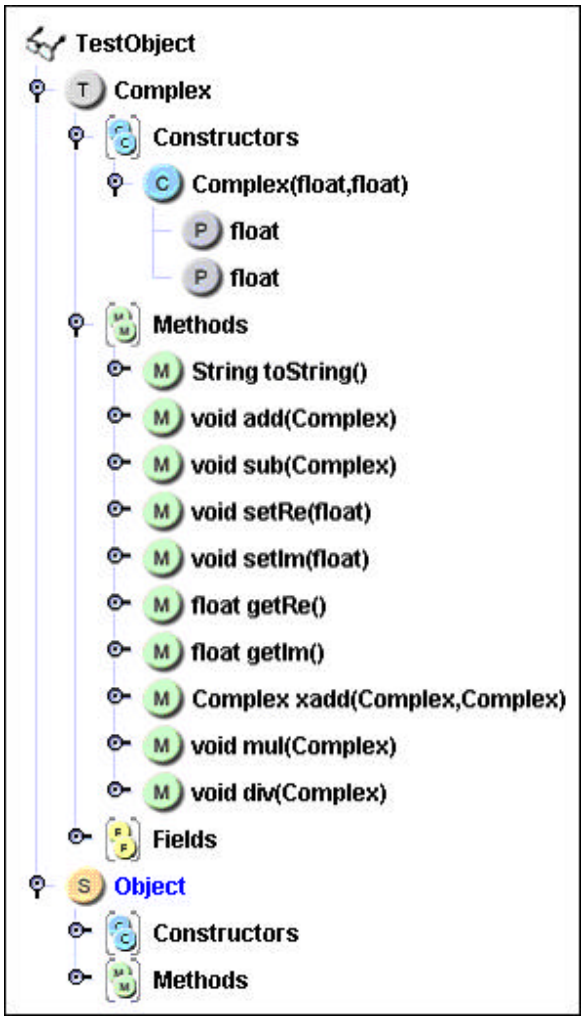


Abb. 5.6: Darstellung von Klassenelementen

Die Darstellung der Information der Klasse kann in Form einer Liste, einer Tabelle oder eines Baums erfolgen. Die Darstellung durch die Baumstruktur (siehe nebenstehende Abbildung) garantiert am besten die Übersichtlichkeit und Abstufung der einzelnen Informationen. Bei vielen Methoden respektive Konstruktoren einer Klasse, die wiederum sehr viele Parameter enthalten können, wird bei einer Liste oder Tabelle schnell die Übersicht verloren.

Der Testtreiber soll die Eingabendaten automatisch generieren und damit Testfälle formulieren (Datenbezogene Testfälle). Daher werden alle relevanten Datenwerte erzeugt, um auf diese Weise festzustellen, ob das Testobjekt alle möglichen Eingaben korrekt behandelt. Für den Klassentest ist die automatische Ermittlung der Eingabedaten und seiner Struktur nicht einfach, da sie möglicherweise Attribute eines geerbten Objekts beziehungsweise Ergebnisse einer fremden Operation sind. Durch die oben genannten Pakete und Klassen wird ermöglicht, die Eingabendaten zu identifizieren und somit die Testfälle zu spezifizieren.

Für die Zuweisung der Datenwerte an die Eingabedaten bietet der Testtreiber drei Möglichkeiten, die im Abschnitt 2.6.2 „Datenbezogenes Testen“ angesprochen wurden. Die erste Möglichkeit ist die Zuweisung durch Zufallswerte. Dies ist die

einfachste Form der Zuweisung, da sie keine Anwendungskenntnisse voraussetzt; die Existenz der Klassenspezifikation ist ebenfalls nicht erforderlich. Weil hierbei allerdings nicht immer eine sinnvolle Generierung von Datenkombinationen möglich ist, wird oft als zweite Möglichkeit die Grenzwertanalyse herangezogen, welche aber nur auf Typen mit Rangordnung anwendbar ist. Diese Form der Zuweisung geht von dem Grundsatz aus, dass, wenn die Algorithmen für die Grenzwerte funktionieren, dies auch für alle dazwischen liegenden Werte gilt. Als dritte Variante steht die Zuweisung durch repräsentative Werte zur Verfügung. Dabei ist die Spezifizierung der Klasse durch OCL unabdingbar. Dieser Ansatz führt zu zwei Haupttestfällen: OCL-Testfällen und automatischen Testfällen ohne OCL, die für die Zuweisung der Parameter die Varianten eins und zwei benutzen (siehe Abbildung 5.7).

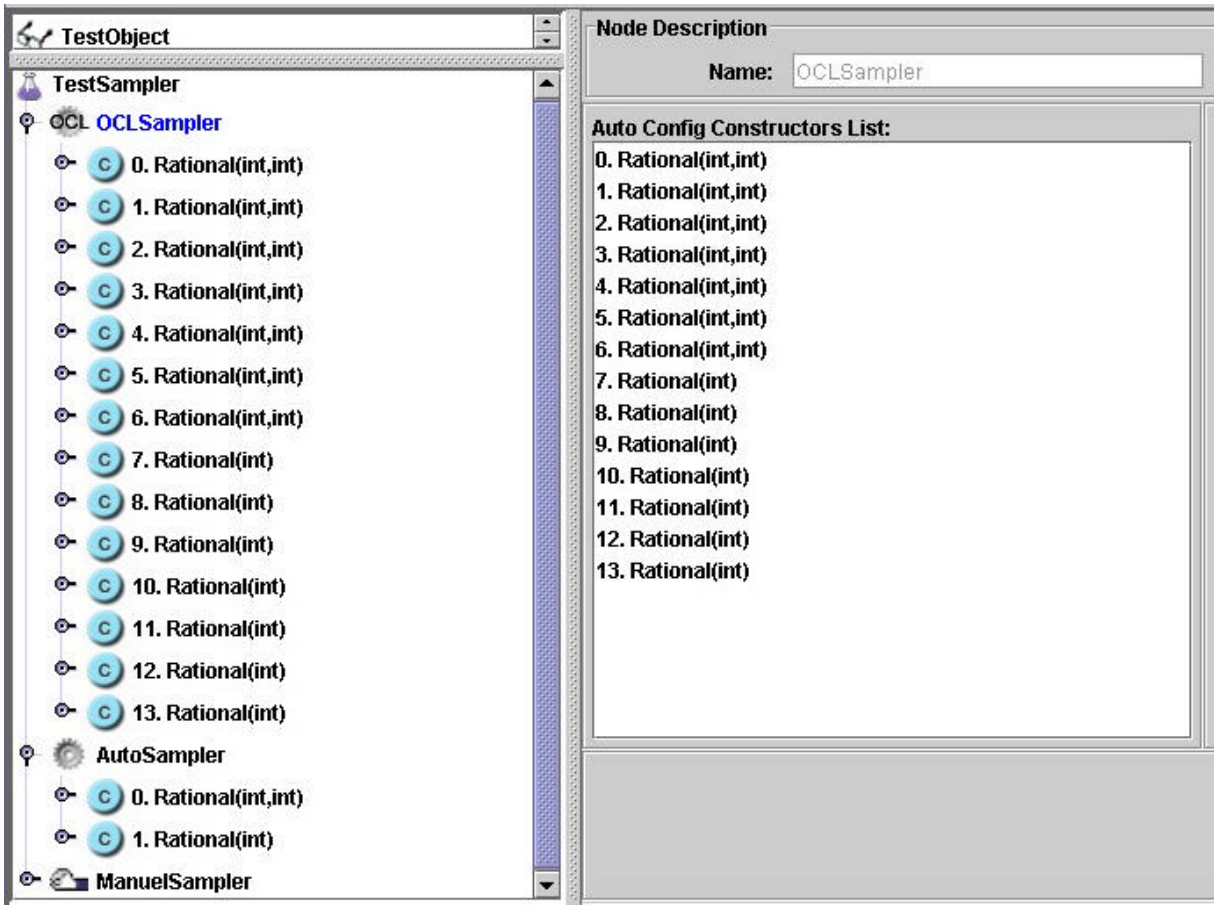


Abb. 5.7: Testfälle

Um die Effizienz des Testtreibers zu steigern, sollte die Zahl der Testfälle zum Erreichen des Testziels auf ein Minimum reduziert werden. Außerdem soll erreicht werden, dass ein Testfall möglichst viel über Anwesenheit und/oder Abwesenheit von Fehlern aussagt. Die Parameter der Testfälle werden mit Hilfe der Bildung von Äqui-

valenzklassen erzeugt. Eine Äquivalenzklasse wird dadurch gekennzeichnet, dass das Testobjekt bei der Verarbeitung eines Vertreters aus dieser Klasse genauso reagiert wie bei allen anderen Werten dieser Klasse. Wenn das Testobjekt mit dem repräsentativen Wert der Klasse fehlerfrei läuft, dann funktioniert es auch mit anderen Werten dieser Klasse (dieser Ansatz wurde bereits im Abschnitt 2.6 „Testansätze“ erwähnt). Demzufolge genügt es, die Parameter in zwei Äquivalenzklassen aufzuteilen, in die den Vorbedingungen entsprechenden und die den Vorbedingungen nicht entsprechenden Parametern.

Die drei Klassen *PrePostUtils, PreConditionAnalyse, EvaluationPreCondition*, welche sich in dem Paket jtestocl*.base.ocl.functionparser* befinden, sind für die Auswertung und die Generierungsparameter von Vorbedingungen zuständig. Die Argument-Klasse *Array* sowie das Parameter-Objekt *Array* werden in den zwei Klassen *BuildConstructorParams* und *BuildMethodParams*, welche sich im Paket *param* befinden, zusammengesetzt, um die Klasse instanzieren zu können beziehungsweise die Methode aufzurufen (siehe Abbildung 5.8). Um der Anforderung, dass der Testtreiber ohne OCL-Spezifikation weiterhin funktioniert, gerecht zu werden, werden die Parameter der OCL-Testfälle im Falle des nicht Vorhandensein der OCL-Spezifikation mit Zufallswerten generiert. Damit kann allerdings nicht sichergestellt werden, dass die Klasse wirklich ihrer Spezifikation entspricht, da die Erzeugung und Auswertung von Zufallsparametern nicht nur einen höheren Aufwand erfordern, sondern auch fehlerträchtiger sind.

Für die automatischen Testfälle werden zwei Ansätze herangezogen. Sie werden benutzt, um die Überdeckung der Testfälle zu garantieren. Zumeinen der Random-Datentest, bei welchem die generierten Testdaten reine Zufallswerte sind. Zum anderen der Datentest mittels Grenzwerterzeugung. Hierbei werden der untere Grenzwert, der obere Grenzwert und der Mittelwert des Wertebereichs der Parameter generiert. Leider ist die Grenzwerterzeugung nur für Datentypen mit Rangordnungen einsetzbar wie zum Beispiel *Double, Float* oder *Integer,* nicht jedoch für *Boolean*. Für String-Datentypen kann die Rangordnung durch alphabetische Ordnung definiert werden, aus diesem Grund wird in dem implementierten Testtreiber die Rangordnung von String-Datentypen wie oben erwähnt definiert und umgesetzt.

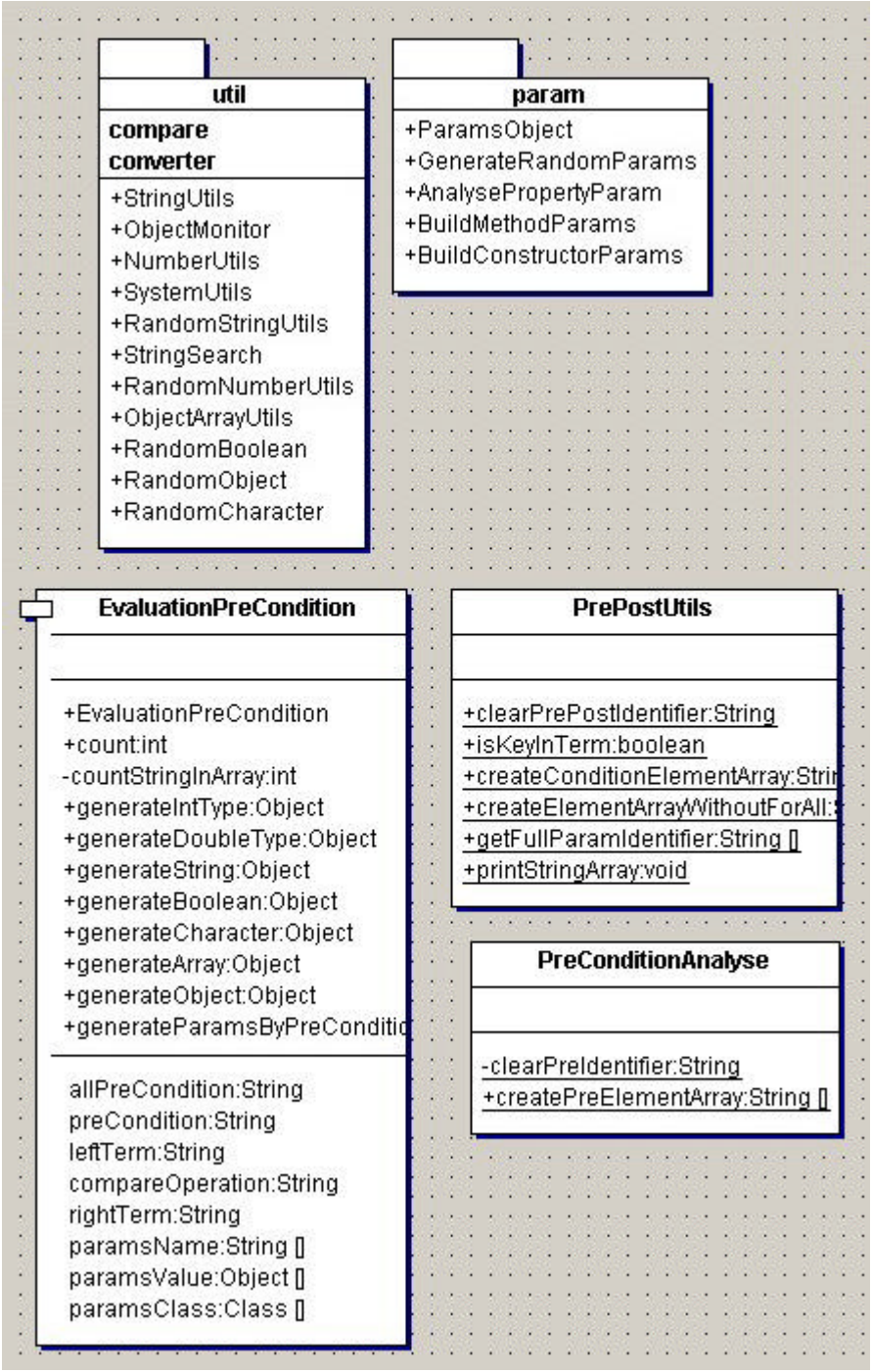


Abb. 5.8: Pakete und Klassen zur Parametergenerierung

Es ist wichtig zu erwähnen, dass die Argumente zur Instanzierung der zu testenden Klasse sowie die Parameter für den Aufruf der Methode, nicht nur primitive Typen sein dürfen, es sollte auch eine Klasse sein. Um solche Parameter zu erzeugen, muss diese Klasse im Klassenpfad enthalten sein, danach werden alle Konstruktoren der Parameterklasse aufgelistet. Der Parameter wird erzeugt, wenn eine neue Instanzierung eines der Konstruktoren erfolgt. Stehen jedoch mehrere Konstruktoren zur Verfügung, muss entschieden werden, welcher Konstruktor zur Instanzierung

verwendet werden soll. Als Defaulteinstellung versucht der Testtreiber immer einen Parameter-Array für den Konstruktor mit der längsten Parameteranzahl zu generieren, um somit die Defaultwerte der Klassenfelder der Parameter so weit wie möglich zu vermeiden.
Die generierten Parameter in diesem Arbeitsschritt werden für die Instanzierung des Testobjekts und dessen Methodenaufruf benutzt, um das Verhalten der zu testenden Klasse zu untersuchen.

5.3 Instanzierung von Testobjekten und Methodenaufrufe

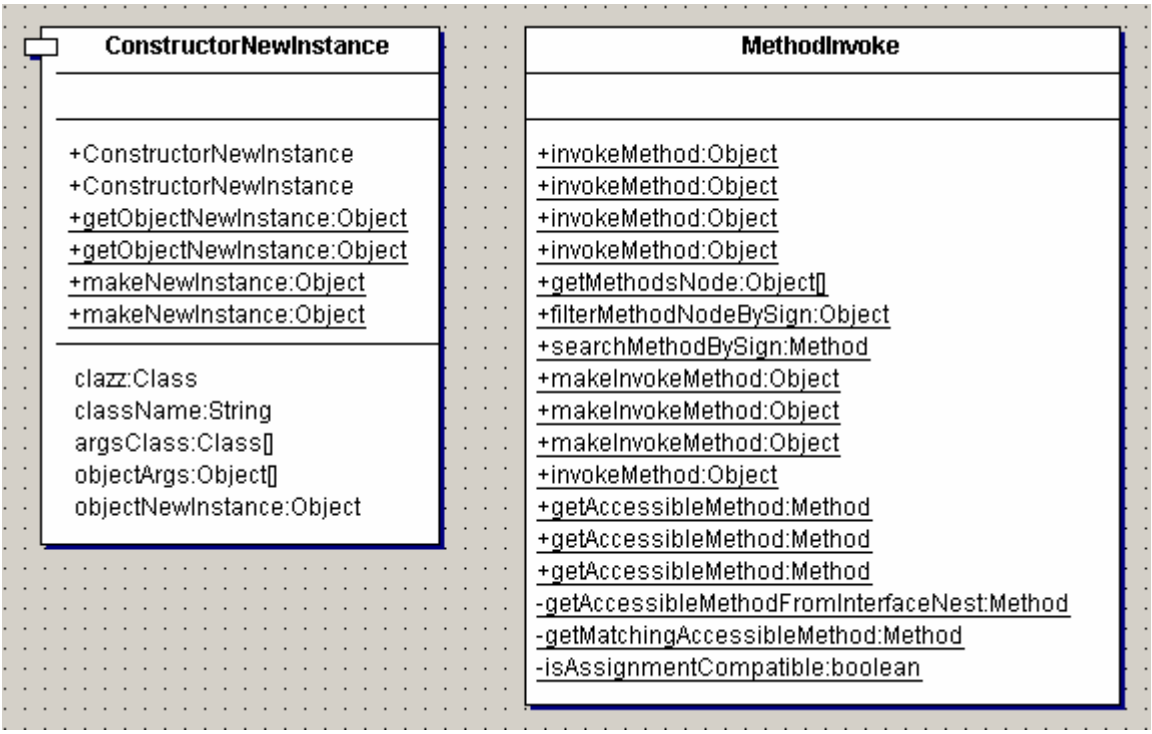


Abb. 5.9: Klassen zur Instanzierung und zum Aufruf von Methoden

Objektinstanzierung

Für jede Testfalldurchführung muss zuerst eine neue Instanz des Testobjekts erzeugt werden. Mit dem Klassenobjekt können nun verschiedene Aufgaben erledigt werden. Insbesondere ist es möglich, Informationen über Membervariablen oder Methoden der Klasse abzufragen und deren öffentliche Methoden aufzurufen. Um Exemplare bestimmter Klassen zu erzeugen, bietet Java-APIs zwei Möglichkeiten.

Zum einen wird durch die Methode *new-Operator* zur Laufzeit ein Exemplar einer Klasse erzeugt. Dazu muss der Compiler den Namen der Klasse wissen, sodass er einen passenden Konstruktor-Aufruf und die Argumente, die dem Konstruktor über-

geben werden, erzeugen kann. Sollte jedoch der Name der gewünschten Klasse für das Testobjekt erst zur Laufzeit bekannt werden, so kann die Methode *new-Operator* nicht benutzt werden. In diesem Fall wird die Methode *newInstance (Object[ ])* der Klasse *java.lang.reflect.Constructor* des Reflection-APIs eingesetzt. Dies wird in der Klasse *ConstructorNewInstance* im Paket jtestocl.action implementiert (siehe Abbildung 5.9).

```
public static Object getObjectNewInstance
                    (String className, Class[ ] argsClass, Object[ ] objectArgs)
{
    Object result = null;

    try {
        Class _clazz = AnalyseClass.getClassForName(className);
        Constructor constructor = clazz.getConstructor(argsClass);
        Object obj = constructor.newInstance(objectArgs);
        result = obj;
    } catch (InstantiationException e) {
        jtestocl.application.Application.exceptionlog.fatal(e.getMessage());
    } catch (IllegalAccessException e) {
        jtestocl.application.Application.exceptionlog.fatal(e.getMessage());
    } catch (IllegalArgumentException e) {
        jtestocl.application.Application.exceptionlog.fatal(e.getMessage());
    } catch (InvocationTargetException e) {
        testocl.application.Application.exceptionlog.fatal(e.getMessage());
    } catch (NoSuchMethodException e) {
        jtestocl.application.Application.exceptionlog.fatal(e.getMessage());
    } finally {
        return result;
    }
}
```

Um Exemplare bestimmter Klassen dynamisch zu erzeugen, wird ein passendes Klassenobjekt benötigt. Dies geschieht durch den Aufruf der Methode

getClassForName (String) der Klasse *AnalyseClass*, wobei der Parameter der Klassenname ist. Danach wird mit *getConstructor()* ein Konstruktorobjekt geholt, das den gewünschten Konstruktor beschreibt. Jedes Konstruktorobjekt kennt eine *newInstance (Object[])*-Methode, die ein neues Exemplar erschafft, indem sie den zugrunde liegenden Konstruktor aufruft. Die Parameter vom *newInstance()* sind ein Feld von Werten, die an den echten Konstruktor übergeben werden. Ein parameterloser Konstruktor wird durch *newInstance (null)* aufgerufen.

Dabei gibt es einige Exceptions zu beachten. Wird die Exception *IllegalAccessException* ausgegeben, bedeutet dies, dass auf den Konstruktor nicht zugegriffen werden kann – bei privaten Konstruktoren zum Beispiel. Sollte die Anzahl der Parameter falsch beziehungsweise eine Konvertierung der Parameterwerte in die benötigten Typen nicht möglich sein, so wird die Ausnahme *IllegalArgumentException* geworfen. Die Exception *InstantiationException* hingegen wird ausgegeben, wenn das Konstruktorobjekt sich auf einen Konstruktor einer abstrakten Klasse bezieht. Falls bei der Ausführung des angegebenen Konstruktors eine Ausnahme auftritt, erscheint die Meldung „*InvocationTargetException*“. Wird die Ausnahme *NoSuchMethodException* angezeigt, bedeutet das, dass der zu instanzierende Konstruktor nicht gefunden werden kann.

Methodenausführung

Nach Aufruf des Konstruktors zum Erzeugen eines Testobjekts, ist der nächste Schritt das Abfragen und Setzen von Variablenwerten der Methode und Aufrufen von Methoden durch die Methode *invoke (Object, Object[])* der Klasse *java.lang.reflect.Method* des Reflection-APIs. Dies wird in der Klasse *MethodInvoke* in Paket *jtestocl.action* implementiert Die wichtigste Methode dabei ist die Methode *invokeMethod (Object, String, Class[], Object[]):*

```
public static Object invokeMethod
                (Object obj, String name, Class[] paramsTypeOfMethod,
                Object[] paramsValueOfMethod)
{
    Object returnValue = null;
    try {
       Class _class = obj.getClass();
       Method method = _class.getMethod(name, paramsTypeOfMethod);
```

```
            returnValue = method.invoke(obj, paramsValueOfMethod);
        } catch (IllegalAccessException e) {
            jtestocl.application.Application.exceptionlog.fatal(e.getMessage());
        } catch (IllegalArgumentException e) {
            jtestocl.application.Application.exceptionlog.fatal(e.getMessage());
        } catch (InvocationTargetException e) {
            jtestocl.application.Application.exceptionlog.fatal(e.getMessage());
        } catch (NoSuchMethodException e) {
            jtestocl.application.Application.exceptionlog.fatal(e.getMessage());
        }
        finally {
            return returnValue;
        }
    }
```

Zunächst wird von einem Klassenobjekt ausgegangen, welches die Klasse des Objekts beschreibt, für das eine Objektmethode aufgerufen werden soll. Anschließend wird ein Methodenobjekt zur Beschreibung der gewünschten Methode benötigt. Wenn während des Kompilierens der Name der Methode nicht feststeht, so lässt sich zur Laufzeit eine in der Klasse definierte Methode nur dann aufrufen, wenn ihr Name als Zeichenkette vorliegt. Für überladene Methoden wird zur Unterscheidung zusätzlich als Klassenobjekte die Parameterliste benötigt. Dies geschieht durch die Methode *getMethod()* aus dem Klasse-Exemplar. Sie verlangt zwei Argumente, zunächst einen String mit dem Namen der Methode und danach ein Array von Klasse-Objekten. Da jedes Element dieses Arrays einem Parametertyp aus der Signatur der Methode entspricht, können so überladene Methoden unterschieden werden. Danach wird die Zielmethode mittels *invoke()* ausgeführt. Die Methode *invoke()* besitzt zwei Argumente: einen Array mit Argumenten, die der aufgerufenen Methode übergeben werden und eine Objektreferenz, welche als *this*-Referenz fungiert und zur Auflösung der dynamischen Bindung dient.

Im Unterschied zu einer parameterlosen Methode muss der parametrisierten Methode ein nicht leeres Objekt-*Array* mit den aktuellen Argumenten übergeben werden. Dabei können in einem Array von *Object[]* keine primitiven Typen abgelegt werden. Um Methoden mit primitiven Parametern aufrufen zu können, werden diese in

die passende Wrapper-Klasse umgewandelt. Beim Aufruf von invoke werden sie dann automatisch zurückgewandelt und dem primitiven Argument zugewiesen. Sollte die Methode ein Resultat liefern, wird es von *invoke()* zurückgeliefert. Bei primitiven Typen erfolgt dies wieder als Wrapper-Objekt.

Bei fehlerhafter Benutzung beziehungsweise Ausführung wird eine der oben angegebenen Ausnahmen analog zum Konstruktor ausgelöst (siehe Tabelle).

IllegalAccessException	Kein Zugriff auf Methode
IllegalArgumentException	Anzahl der Parameter ist falsch beziehungsweise Konvertierung der Parameterwerte nicht möglich
InvocationTargetException	Exception bei Ausführung der Methode
NoSuchMethodException	Methode kann nicht gefunden werden

Nach der Instanzierung der zu testenden Klasse und dem Aufrufen ihrer Methoden ist die Auswertung des internen Zustands der Testklasse und der Resultate der Ausführung der Methoden sowie die Erstellung eines Testberichts der letzte Schritt.

5.4 Ergebnisvalidator

Der interne Zustand des Testobjekts sowie das Objekt als Resultat nach der Ausführung der Methode - sofern es existiert - werden durch die Klasse *ObjectMonitor* im Paket *jtestocl.base.util* sichtbar gemacht. Dabei werden alle Felder des Objekts aufgelistet. Die zwei wichtigsten Methoden sind *getNameAllFields(Object)* und *getValueAllFields(Object)*. Die Methode *getNameAllFields* liefert die gesamten Feldnamen des Objekts als String-Array zurück, während die Methode *getValueAllFields* alle Werte der Felder des Objekt als Objekt-Array zurückliefert. Wenn ein Objekt keine Instanz von einem primitiven Typ beziehungsweise von einem String-Datentyp ist, wird bei der Auswertung die Methode *getValueAllFields* rekursiv solange aufgerufen, bis alle Felder als Instanz von einem primitiven Typ sind.

Die Objektzustandshistorie wurde bei der Auswertung im Testresultat-Baum gespeichert. Somit wird es ermöglicht, zu beliebigen Zeitpunkten den internen Zustand nachzuvollziehen (siehe Abbildung 5.10).

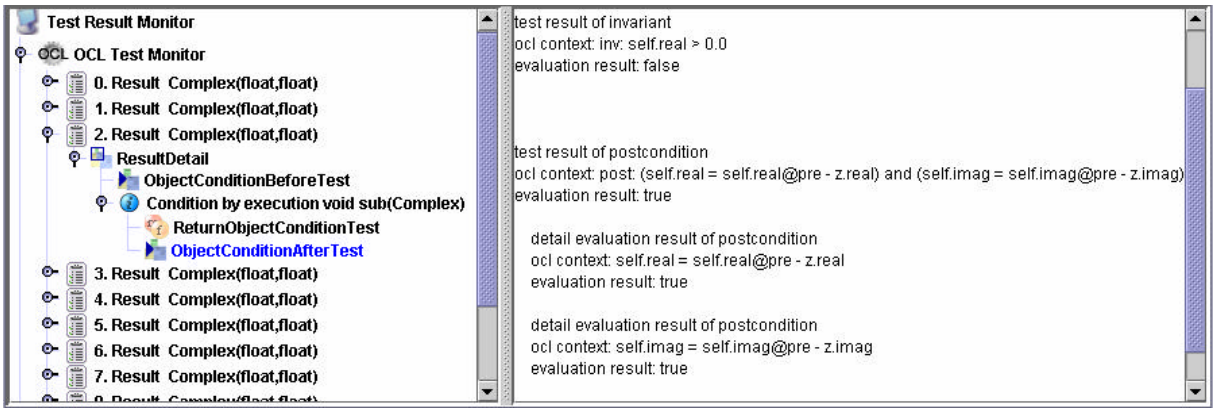


Abb. 5.10: Objektmonitor

Wie im Abschnitt 4.1 „Einsatz von OCL zum automatisierten Testen von Java-Klassen“ bereits erwähnt, muss nach jeder Instanzierung der zu testenden Klasse und deren Methodenaufruf die Invariante erneut überprüft werden. Außerdem muss der Testtreiber zusätzlich nach jedem Methodenaufruf die Nachbedingungen der Methode, so weit sie existieren, überprüfen. Um diese Aufgabe zu bewerkstelligen, muss zuerst der Funktionsparser die Bezeichnung von einzelnen Termen der Invariante respektive der Nachbedingungen durch diejenigen Werte ersetzen, die sich im Testobjekt nach der Instanzierung, vor dem Aufruf der Methoden, nach der Ausführung der Methode oder im Resultobjekt der Methode befindet. Der Testtreiber muss danach in der Lage sein, diese neuen Ausdrücke auszuwerten, zu berechnen und das Ergebnis auszugeben. Dazu muss der Testtreiber mit allen numerischen Typen, boolschen Typen, Character-, String- sowie Objekttypen umgehen können.

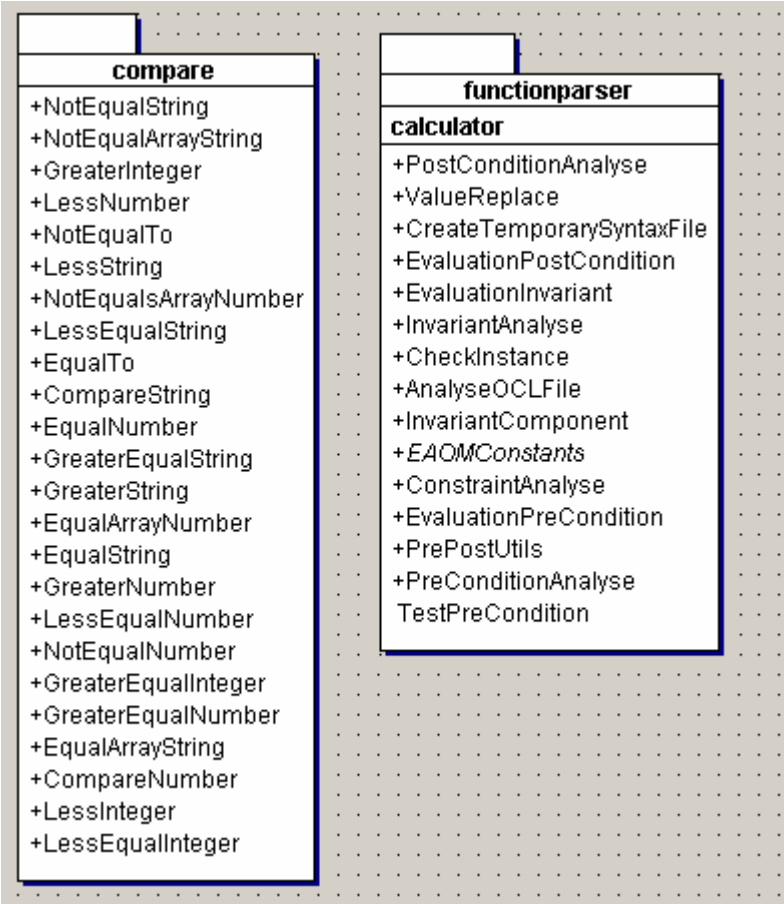


Abb. 5.11: Pakete zur Testauswertung

Die hierfür zuständigen Pakete sind *jtestocl.base.ocl.functionparser* und *jtestocl.base.util.compare* (siehe Abbildung 5.11). Eine sehr wichtige Besonderheit bei der Auswertung sind die Rundungsfehler bei der Berechnung von Double-Datentypen. Eine Lösung wäre die Verringerung der Nachkommastellen in Abhängigkeit der Anforderungen der zu testenden Klasse. Um die Testergebnisse nicht zu verfälschen, rundet der Testtreiber nur bis auf die sechste Stelle nach dem Komma. Dies ist nur ein Default-Wert, der jederzeit durch den Benutzer angepasst werden kann.

6 Resümee

Der entwickelte Testtreiber ermöglicht die Unabhängigkeit von Tester und Programmierer, da Kenntnisse der Implementierung keine Voraussetzung für ein erfolgreiches Testen sind. Existierende Klassen können außerdem bei einer erneuten Freigabe erneut getestet werden, ohne dass der Testtreiber verändert werden muss. Ein wichtiger Grundsatz des Klassentests, dass die Testfälle nicht nur auf gültige Funktionsbereiche, sondern auch auf ungültige Funktionsbereiche beruhen, wurde in dem entwickelten Testtreiber vollständig umgesetzt. Damit kann die Robustheit der zu testenden Klasse bewertet werden. Es ist bekannt, dass ein vollständiger Black-Box-Test nicht möglich ist, weil die Menge der Eingangsparameter, die generiert werden müssen, um den gesamten Funktionsbereich zu überdecken, selbst bei einfachen Methoden der Testklasse unermesslich groß wäre. Das bedeutet, die Auswahl der Testfälle kann nicht auf Vollständigkeit abzielen, sondern auf eine hohe Wahrscheinlichkeit, Fehler zu finden. Diese Anforderung wird in dem entwickelten Testtreiber durch die Kombination verschiedener Verfahren wie Bildung von Äquivalenzklassen und Grenzwertanalyse realisiert. Mit solchen systematischen Testfallbestimmungen ist es möglich, mit geringem Aufwand ein Maximum an unterschiedlichen Reaktionen des Testobjekts zu erzielen. Durch die Automatisierung zur Erzeugung von Testfällen wird nicht nur die Zeitspanne für die Durchführung der Tests verringert, es kann auch auf die langwierigen und ermüdenden Testeingaben, die sonst von einem Tester manuell durchgeführt werden müssten, verzichtet werden. Dadurch werden Ressourcen sinnvoller genutzt. Außerdem ist die Methode der Grenzwertanalyse sehr effektiv im Aufdecken der kritischen Fehler.

Die Benutzung von OCL zur Spezifikation der Testklasse und somit als Grundlage für die Parametergenerierung und Validierung von Invarianten der Klasse sowie der Nachbedingungen von Methoden ist in dem entwickelten Testtreiber implementiert. Die Vorteile dieser Benutzung sind die Aufdeckung der fehlerhaften Implementierung der Testklasse im Vergleich mit ihrer Spezifikation und die einfachere Erreichbarkeit der Testfallüberdeckung durch die Bildung von Äquivalenzklassen der Eingabeparameter. Die gründliche Inspizierung aller Testergebnisse mittels Objektmonitor und der OCL-Spezifikation vermittelt dem Tester einen tiefen Einblick über das Testergebnis.

Der Entwurf und die Implementierung des Testtreibers wurden in mehreren Iterationen durchgeführt. Viele Komplikationen und sinnvolle Beschränkungen wurden durch erste Ergebnisse deutlich, zum Beispiel die Auswertung von Invarianten und Nachbedingungen. Die Fähigkeiten und Grenzen des Reflektion-APIs wurden erst bei der Verwendung voll erfasst. Für die Implementierung der *View*-Ebene (siehe Kapitel 4.2) wurde *Swing* benutzt.

Neben den oben genannten positiven erreichten Zielen sind auch einige Nachteile zu bedenken, die durch die Automatisierung der Parametergenerierung und durch die Klassenspezifizierung entstehen. Der Testreiber kann Qualitätseigenschaft des Testobjekts erkennen; das Vorhandensein unerwünschter Eigenschaften des Testobjekts hingegen kann er nicht mit Sicherheit nachweisen. Ein weiterer Nachteil ist, dass der Testtreiber fehlerhafte beziehungsweise unvollständige Spezifikationen der Klasse nicht erkennen kann. Eine Methode zum Beispiel, die nicht in der Klassenspezifikation enthalten ist, aber unerwünschte Ergebnisse liefert, wird durch den Testtreiber nicht erkannt.

Der Testtreiber befindet sich noch im Entwicklungsstand eines Prototyps, da es im Zeitrahmen der Diplomarbeit nicht möglich war, alle konzeptionell vorgesehenen Fähigkeiten zu implementieren. Eine Weiterentwicklung des Testreibers sowohl auf der funktionalen Ebene als auf der Ebene der Bedienbarkeit ist möglich. Zurzeit kann der Testtreiber nur ein- und zwei-dimensionale Arrays generieren. Um beliebige mehrdimensionale Arrays zu erzeugen, steht die Klasse *ArrayCreator* im Paket *jtestocl.base.ut*il zur Verfügung. Da das Thema zu komplex ist, um es vollständig innerhalb der vorgesehenen Zeit zu bearbeiten, ist es noch nicht möglich gewesen, diese Klasse zu integrieren. Ebenso konnte nicht die Klasse *java.lang.Thread* bei der Implementierung eingebunden werden, welche benutzt werden kann, um während des Testlaufs in jedem Moment manuell einzugreifen, um Parameter zu ändern beziehungsweise den Test abzubrechen. Zukünftig könnte es möglich sein, die Aufrufsequenz der Methoden oder die Testobjektzustände unter Zuhilfenahme von OCL zu beschreiben, um anhand dieser Spezifikation Testfälle zu generieren. Im Rahmen dieser Diplomarbeit konnte die Auswertung von OCL-Ausdrücken nicht

vollständig gelöst werden. Die Einschränkungen, die bei der Auswertung der OCL-Constraints durch den Funktions-Parser bestehen, sind im Anhang B beschrieben.

In Java ist nicht nur die Eingabe von Parametern als primitive Typen oder Strings möglich, sondern auch als Objekte. Die Generierung von Objekten als Parameter in dem entwickelten Testtreiber erfolgt durch rekursive Algorithmen in der Klasse *BuildConstructorParams.java* und in der Klasse *BuildMethodParams.java*, allerdings sind diese Parameterwerte Zufallswerte. Die Grenzwertanalyse und die Bildung von Äquivalenzklassen für Objekte wurden nur innerhalb der ersten Parameterebene realisiert, für die tieferen Parameterebenen erfolgte noch keine Implementierung. Ein Vorschlag wäre, mit rekursiven Algorithmen, die schon in den beiden oben erwähnten Klassen enthalten sind, dies zu realisieren – die dafür benötigten Parameter *String oclCondition* zur Parametergenerierung wären bereits vorhanden.

Die Behandlung von Interface ist in diesem Entwicklungstand nur durch manuelle Eingabe möglich. Eine mögliche Variante wäre, die gesamte Java-Klasse im ClassPath zu durchsuchen und damit diejenigen Klassen, die Interface implementieren, zu lokalisieren. Anhand solcher Klassen und mittels zufälliger Auswahl könnten die Parameter generiert werden. Dies hat jedoch einen großen Nachteil – es würde die Effizienz des Testtreibers erheblich beeinträchtigen, da die automatische Suche sehr viel Zeit in Anspruch nehmen kann. Ein weiterer Vorschlag wäre, die OCL-Spezifikation dieser Klassen aufzulisten. Mit Hilfe dieser Liste kann der Testtreiber die Klassen nacheinander abarbeiten oder zufällig auswählen, um die benötigen Parameter zu generieren.

7 Literatur- und Quellenverzeichnis

[AKS] Aksit, Mehmet: *The Model-view-controller Pattern*, <http:// trese.cs. utwente.nl /courses/university_courses/patterns/material/ >

[Bal1991] Balzert, Helmut: *Lehrbuch der Software-Technik*, Oldenburg, München-Wien,1991

[Bal1998] Balzert H.: *Lehrbuch der Software-Technik. Software-Management, Software-Qualitätssicherung, Unternehmensmodellierung*, Heidelberg (1998)

[Bal2000] Balzert H.: Lehrbuch *der Software – Technik*, Spektrum, (2000)

[BAR2002] Barthelmann, Klaus G. : http://www.informatik.uni-mainz.de/~barthel/OOPJ /Lektionen1/11/Lektion11.html

[Bec2000] Beck, Kent: *Extreme Programming explained: embrace change*, Addison Wesley (2000)

[Bei1990] Beizer, B.: *Software Testing Techniques,* Second Edition, Van Nostrand Reinhold, (1990)

[Bei1995] Beizer, B.: *Black-BoxTesting,* John Wiley & Sons, (1995)

[Bin1996] Binder, Robert V.: *Testing object-oriented systems,* Software Testing, Verification & Reliability, Vol. 6 Nr. 3/4 (1996)

[Bin2000] Binder, Robert V.: *Testing object-oriented systems: models, patterns, and tools,* The Addison-Wesley object technology series, Addison-Wesley (2000)

[Bou1997] Bourne, Kelley C.: *Testing Client/server System*, Mc Graw-Hill (1997)

[BLUEJ] BLUEJ: <http://www.bluej.org/>

[BRO2002] Broy, Manfred, Pretschner, Dr. Dussa-Zieger, Dr. Kriebel: <http://wwwbroy.informatik.tu-muenchen.de/~pretschn/teaching/> (2002)

[Ehr1985] Ehrenberger, W.: *Statistical Testing of Real-Time Software,* Springer (1985)

[ESS2001] Esser, Friedrich: *Java 2 Patterns, Idioms, Java-Zertifizierung,* Galileo Computing (2001)

[For2000] Forbrig, P.: *Testen - Einführung und Überblick* < wwwswt.informatik.uni-rostock.de/deutsch/ Lehre/> (2000)

[How1987] Howden, W.: *Functional Program testing*, McGraw-Hill, New York (1987)

[IBM] IBM: <http://www.software.ibm.com/ad/ocl>

[JUNIT] JUnit: <http://www.JUnit.org>

[McC1983] McCabe, T.: *Structured Testing Tutorial*, New York (1983)

[MgSy2001] McGregor, J., Sykes, D.: *A Practical Guide to Testing Object-Oriented Software*, Addison Wesley (2001)

[MgSy1992] McGregor, J., Sykes, D.: *Object-Oriented Software Development - Engeneering Software for Reuse*, Int. Thomson Computer Press, London (1992)

[Kan2002] Kaner, Cem: *Improving the Maintainability of Automated Test Suites, Paper Presented at Quality Week,* <http://www.kaner.com/lawst1.htm (Dez2002)>

[Mar2000] Marick, Brian: *A Survey and Discussion of Automated Testing,* <http://www.testingcraft.com/automated-testingsurvey.html (Nov 2000)>

[Mye1989] Glenford J Myers: *The Art of Software Testing,* 3. Aufl., Oldenbourg (1989)

[Mye1995] Myers, Glenford J.: *Methodisches Testen von Programmen,* München, Oldenbourg Verlag (1995)

[OMG99] OMG: *UML 1.3 Specification*, http://www.omg.org/technology/documents/formal/unified_modeling_language.htm, (1999)

[Oes1996] Oestereich, Bernd: *Objektorientierte Softwareentwicklung,* 4. Aufl., Oldenbourg Wiss., (1998)

[PCJN1994] Coard, Peter, Nicola, Jill: *OOP. Objektorientierte Programmierung,* Markt und Technik, (1994)

[Per1999] Perry, W.E.*: Effective Methods for Software Testing*, Wiley (1999)

[Pom1996] Pomberger, Gustav, Blaschek, Günther: *Software Engineering, Prototyping und objektorientierte Software-Entwicklung,* 2.Aufl., Carl Hanser, (1996)

[Ric2002] Richter, Mark: *A Precise Approach to validating UML Models and OCL Constraints*, Logos Verlag, Berlin (2002)

[Sch1982] Schmitz, Paul, Bons, Heinz, Megen, Rudolf van: *Testen im Software-Lebenszyklus,* Braunschweig (1982)

[SneWin2002] Sneed, Harry M., Winter, Mario: *Testen objektorientierter Software*, Leipzig (2002)

[Tha2000] Thaller, Georg Erwin: *Software-Test, Verifikation und Validation*, Hannover, Verlag Heinz Heise (2000)

[Trau1993] Trauboth, H.: *Software Qualitätssicherung Konstruktive und analytische Maßnahmen*, Oldenburg (1993)

[WC1980] White, L., Cohen, E.: *A Domain Strategy for Computer Testing*, IEEE Trans. On SE, Vol. 6, Nr. 3, 1980

[WK1999] Warmer, Jos, Kleppe, Anneke: *The Object Constraint Language*, Addison Wesley, Reading, Massachusetts (1999)

Anhang A

Dokumentation des Testtreibers

Es folgt nun die Dokumentation der Klassen, die von dem Prototypen verwendet werden.

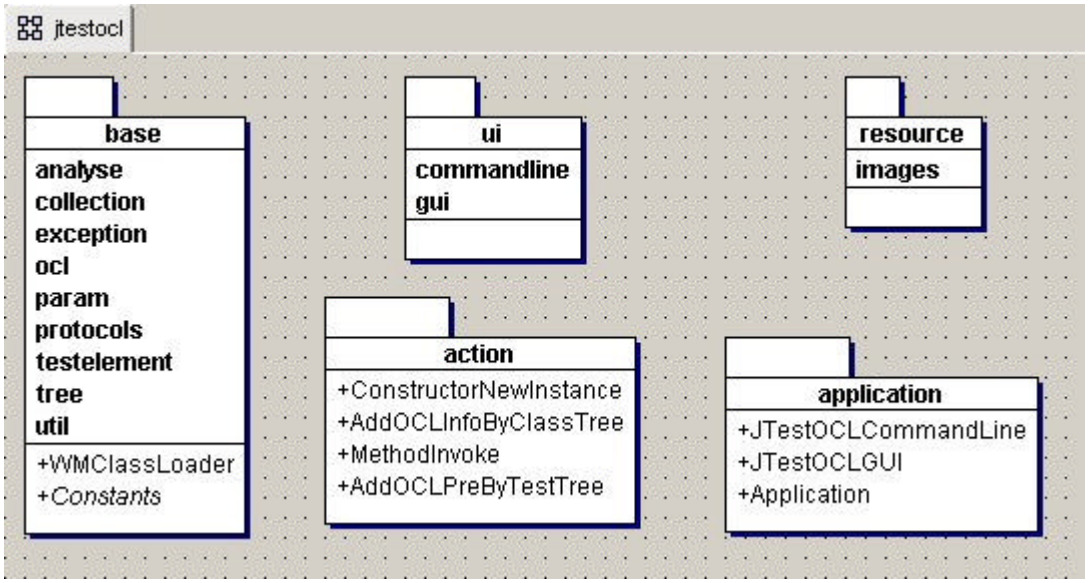


Abb. A.1: Hauptpakete

Der Testtreiber besteht aus den fünf Hauptpaketen *application, base, ui, action* und *resource* (siehe Abbildung A.1). Das Paket *resource* beinhaltet lediglich die Gif-Bilder für die Darstellung der Oberfläche des Testtreibers.

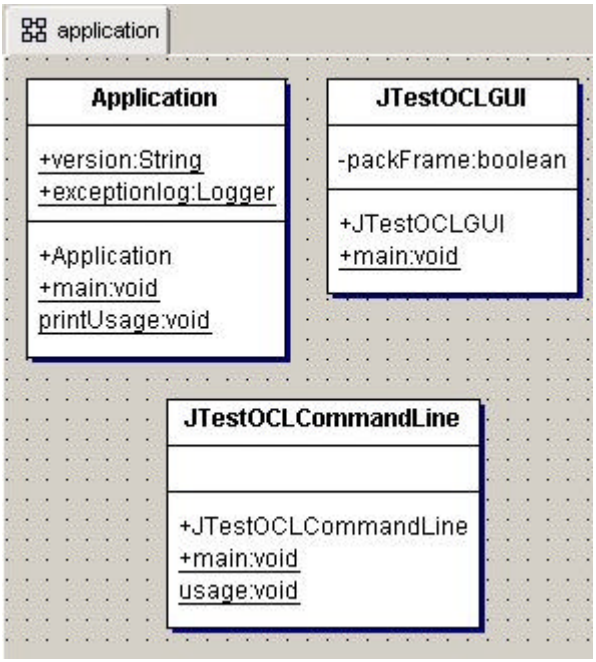


Abb. A.2: Paket application

Die Klasse *Application* ist für den Start des Testtreibers zuständig. Das Testen mittels Grafik-Oberfläche wird durch die Klasse JTestOCLGUI ermöglicht. Für den Test durch eine Kommandozeile dient die Klasse JTestOCLCommandLine (siehe Abbildung A.2).

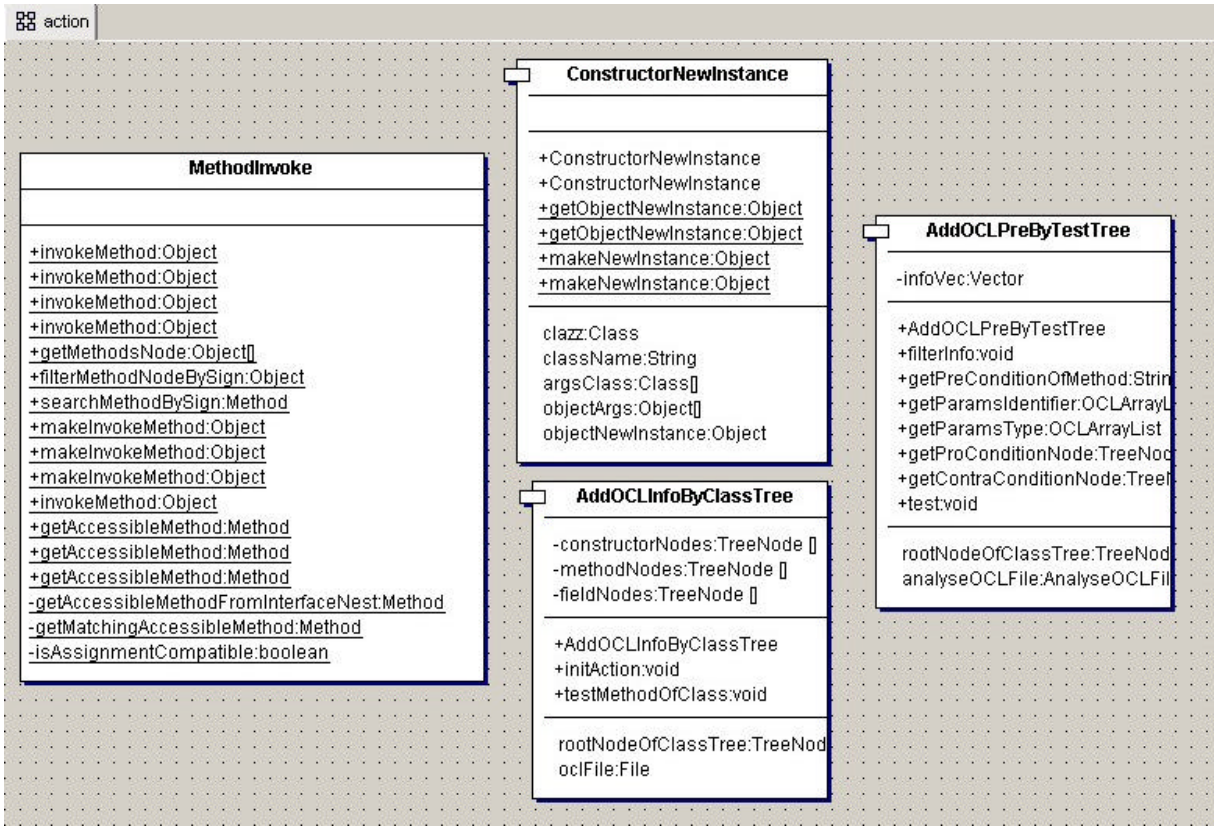


Abb. A.3: Paket action

Um ein Testobjekt zur erzeugen, wird die Klasse *ConstructorNewInstance* benutzt. Dafür werden die zu testende Klasse, die Argument-Klasse und die Objekt-Klasse als Parameter benötig. Ein dynamischer Methodenaufruf erfolgt durch die Klasse *MethodInvoke.* Um dieses Ziel zu realisieren, steht eine Reihe von Methoden zur Verfügung: *invokeMethod* und *makeInvokeMethod*. Falls die OCL-Spezifikation in Form einer Textdatei geladen wird, werden die Klassen *AddOCLInfoByClassTree* und *AddOCLPreByTestTree* benutzt, um die Informationen im *TreeNode* zu erweitern. Diese Informationen können ebenfalls bei der späteren Auswertung einbezogen werden (siehe Abbildung A.3).

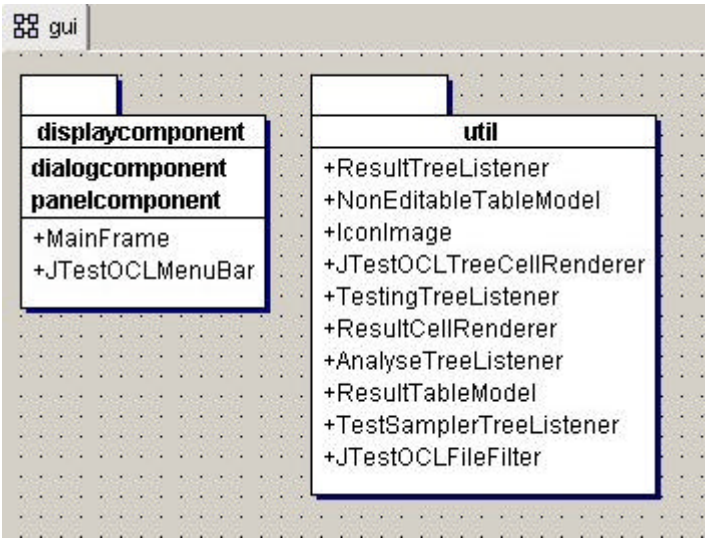


Abb. A.4: Paket gui

Die Paket *ui* besteht aus den zwei Unterkomponenten *gui* (Siehe Abbildung A.4) und *commandline*. *Gui* dient zur Darstellung der Oberfläche des Testtreibers, um die Interaktion zwischen Tester und Testtreiber zu ermöglichen. Das Paket *gui* besteht wiederum aus zwei Komponenten – *displaycomponent* und *util*. *Displaycomponent* beinhaltet ebenfalls zwei Pakete – *dialogcomponent* und *panelcomponent,* welche den Informationsaustausch zwischen dem Benutzer und dem Testtool und die Darstellung der Oberfläche des Testtools ermöglichen. Das Paket *util* beinhaltet eine Sammlung von Werkzeugen zur Aktualisierung der Oberfläche nach einer Interaktion. Die Klasse *JTestOCLMenuBar* ist nicht nur für die Darstellung der Menüs des Testreibers zuständig, sondern wird ebenfalls für die Testaktionen benötigt. Die Klasse *MainFrame* benutzt all diese Pakete zur Darstellung der entgültigen Testoberfläche.

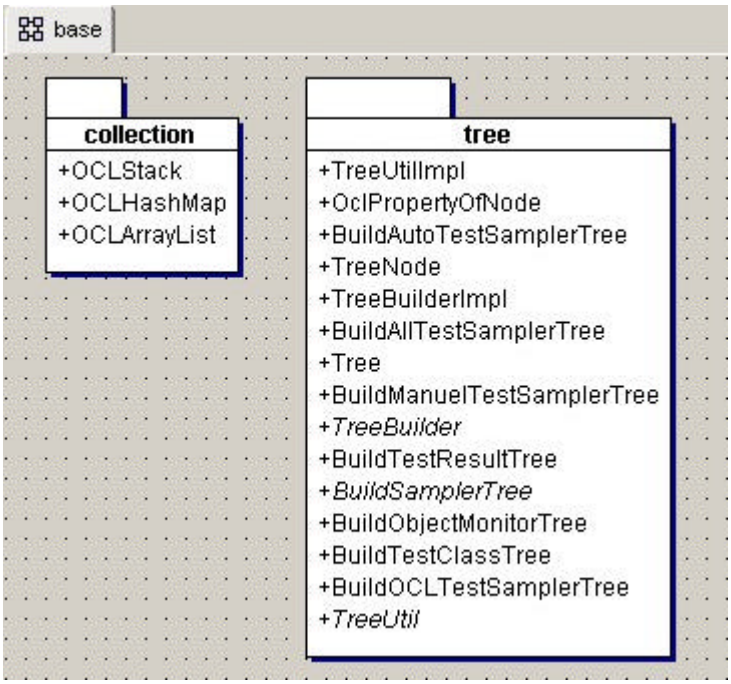


Abb. A.5: Pakete collection und tree

Die zwei Pakete *collection* und *tree* im Paket *base* werden für die Speicherung von Informationen der zu testenden Klasse und der OCL-Spezifikation benötigt (siehe Abbildung A.5).

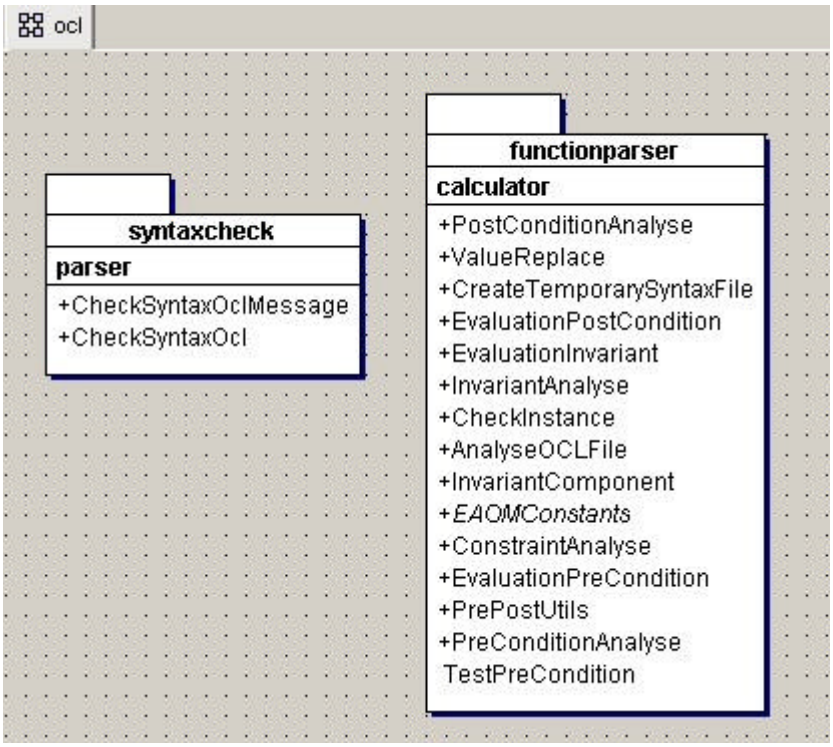


Abb. A.6: Paket ocl

Um die OCL-Syntax zu überprüfen, wird das Paket *syntaxcheck* verwendet, das sich zusammen mit dem Paket *funktionparser* in dem Paket *ocl* befindet. Das Paket *functionparser* dient zur Berechnung und Auswertung von Vor- und Nachbedingungen sowie von Invarianten der OCL-Spezifikation (siehe Abbildung A.6).

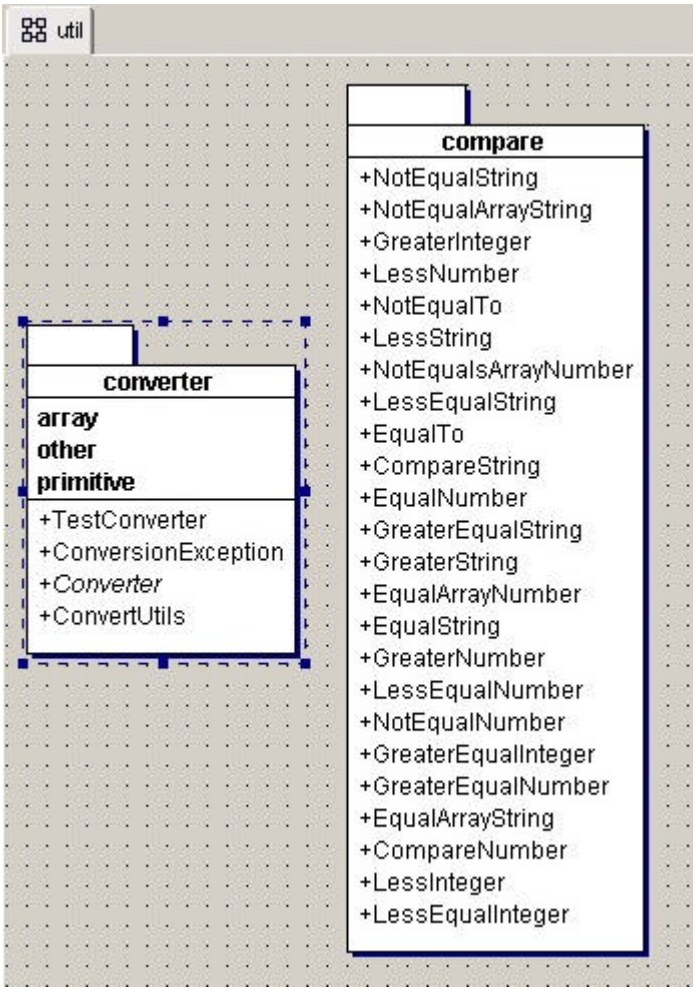


Abb. A.7: Pakete converter und compare

Das Paket *converter* ist für die Konvertierung der Eingabeparameter zuständig. Das Paket *compare* wird dazu benutzt, den Vergleich von Numbertyp, Stringtyp, Objekttyp und Arraytyp zu unterstützen, welcher für die Auswertung der OCL-Spezifikation vonnöten ist (siehe Abbildung A.7).

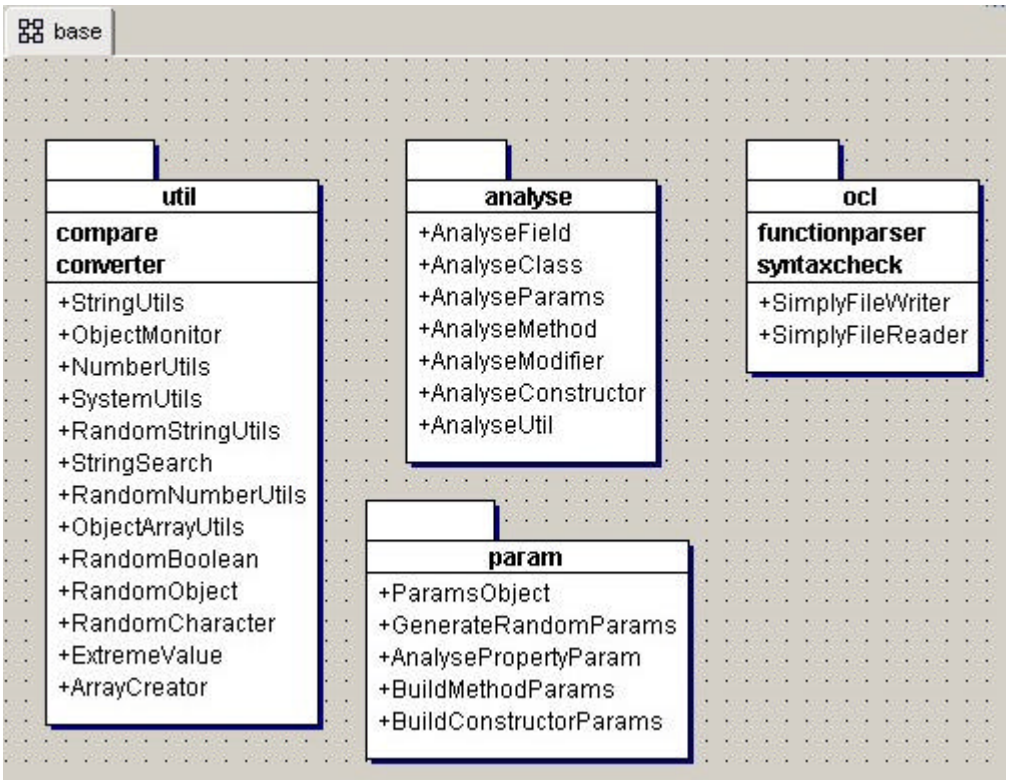


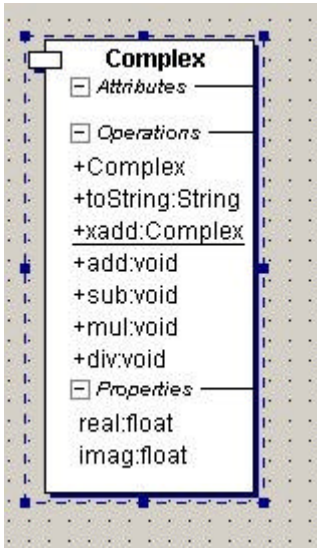
Abb. A.8: Pakete uitl, analyse und param

Das *analyse*-Paket dient zur Analyse der zu testenden Klasse. Dabei werden alle Eigenschaften der Klasse durch *java.lang.reflect* herausgefiltert. Die Klassen in den Paketen *util* und *param* ermöglichen nicht nur die Parametergenerierung für die Testfälle, sondern auch die Betrachtung des internen Objektzustands des Testobjekts durch die Klasse *ObjectMonitor* (siehe Abbildung A.8).

Anhang B

Benutzung von OCL innerhalb des Testtreibers

Anhand der folgenden Java-Klasse wird auf die Benutzung von OCL näher eingegangen.




```
public void mul(Complex z) {
        float t1, t2, t3;
        t1 = (real + imag) * z.real;
        t2 = real * (z.imag- z.real);
        t3 = imag*(z.imag+ z.real);
        real = t1 -t3;
        imag = t1+t2;
    }
```

Die Methode *mul(Comlex)* berechnet die Multiplikation zwischen zwei komplexen Zahlen. Dabei handelt es sich nicht um die Standardalgorithmen zur Multiplikation von komplexen Zahlen sondern um Optimierungsalgorithmen, bei denen die Anzahl der Multiplikationen von vier auf drei notwendige Multiplikationen reduziert wurde.

$$Z_1 = a_1 + b_1 * j$$
$$Z_2 = a_2 + b_2 * j$$
$$Z = Z_1 * Z_2 = (a_1 * a_2 - b_1 * b_2) + (a_1 * a_2 + b_1 * b_2) * j$$

Eine Operationsspezifikation findet in drei Schritten statt. Zunächst erfolgt die Spezifikation der Signatur. Um die Signatur aus der Klasse zu übernehmen, wird die Klasse *Complex* gebraucht:

context Complex::mul(z : Complex)

Danach werden die Vorbedingungen spezifiziert. Sollte eine Operation keine speziellen Vorbedingungen haben, wird dies im Testreiber durch

pre: -- none

ausgedrückt.

Die Spezifikation der Nachbedingungen erfolgt nach folgendem Muster:

*post: (self.real = self.real@pre * z.real - self.imag@pre*z.imag)*
*and (self.imag = self.imag@pre * z.real + self.real@pre * z.imag)*

Bei der Spezifikation von Nachbedingungen spielen auch zeitliche Aspekte eine Rolle, wenn diese sich auf den Zustand des Objekts unmittelbar vor oder nach der Operationsausführung beziehen. @pre repräsentiert, an einen beliebigen Attribut-Bezeichner angefügt, den Wert, den das Attribut vor der Ausführung der Operation hatte. *self.real@pre,* zum Beispiel, bezeichnet die Werte des Realteils der komplexen Zahl vor der Ausführung der Operation.

Nach der oben aufgeführten Formel ist es möglich, die Nachbedingungen in zwei Hauptkomponenten einzuteilen, welche durch den Operator *and* miteinander verbunden sind. Die Auswertung der Nachbedingungen erfolgt durch die Auswertung der einzelnen Komponenten. Erst danach wird die Zusammenlegung beider Teile durch den Verbindungsoperator ausgewertet. Zurzeit unterstützt der Testtreiber nur die Auswertung der Nachbedingungen, welche die Verbindungsoperatoren *and, or, not* und *implies* benutzen. Um die Auswertung der Verbindungsoperatoren *nand* und *xor* zur realisieren, kann der Tester die de Morganschen Gesetze

a nand b = (not a) or (not b)

a xor b = (not a) and (not b)

benutzen, um dadurch die Operatoren zu ihren Basisoperatoren zurückzuführen. Die Auswertung der Invarianten erfolgt analog.

Ein weiteres Beispiel für eine Vor- und Nachbedingung ist

```
public void div(Complex z) {
    float d, nenn;
    if (Math.abs(z.real) >= Math.abs(z.imag)){
       if (z.real == 0) {
          real = imag / z.imag;
          imag = - real / z.imag;
       }
       else {
          d = z.imag /z.real;
          nenn = z.real+ d * z.imag;
          real = (real + d * imag) / nenn;
          imag = (imag -  d * real) / nenn;
       }
    }
    else {
       if (z.imag == 0) {
          real = real / z.real;
          imag = imag / z.real;
       }
       else {
          d = z.real/ z.imag;
          nenn = d* z.real + z.imag;
          real = (d * real + imag) / nenn;
          imag = (d * imag - real) / nenn;
       }
    }
  }
```

Für die Spezifikation der Signatur wird die Klasse *Complex* benutzt:

context Complex::div(z : Complex)

Um die Vorbedingungen spezifizieren zu können, muss sichergestellt sein, dass der Realteil und der imaginäre Teil der komplexen Zahl nicht gleichzeitig 0 ist.

pre: (z.real <> 0) and (z.imag <> 0)

Da die Parametergenerierung der Testfälle auf der OCL-Spezifikation basiert, ist die Auswertung der OCL-Spezifikation ein wichtiger Bestandteil des Testtreibers. Die Auswertung von Vorbedingungen ist ein sehr komplexes Thema, weil die Syntax und Logik der OCL-Ausdrücke sehr komplex sein können. In Rahmen dieser Diplomarbeit ist es nicht möglich, alle Varianten der Vorbedingungen auszuwerten. Es beschränkt auf *and* als Verbindungsoperator zwischen den einzelnen Termen. Jeder Parameter darf höchstens zwei Mal vorkommen. Wenn es sich um eine *or*-Beziehung handelt, kann der Tester diese in zwei OCL-Textdateien aufspalten und für jede Testdatei einzeln Testfälle generieren.

Die Spezifikation der Nachbedingung sieht wie folgt aus:

*post: (self.real =, (self.real@pre * z.real + self.imag@pre * z.imag)/(z.real * z.real + z.imag * z.imag))*
*and (self.imag = (self.imag@pre * z.real - self.real@pre * z.imag)/(z.real * z.real + z.imag * z.imag))*

Anhang C

In diesem Anhang wird das Klassendiagramm, das für die Beispiele im Abschnitt 2.7 benutzt wurde, ausführlich erläutert.

Royal & Loyal Klassendiagramm

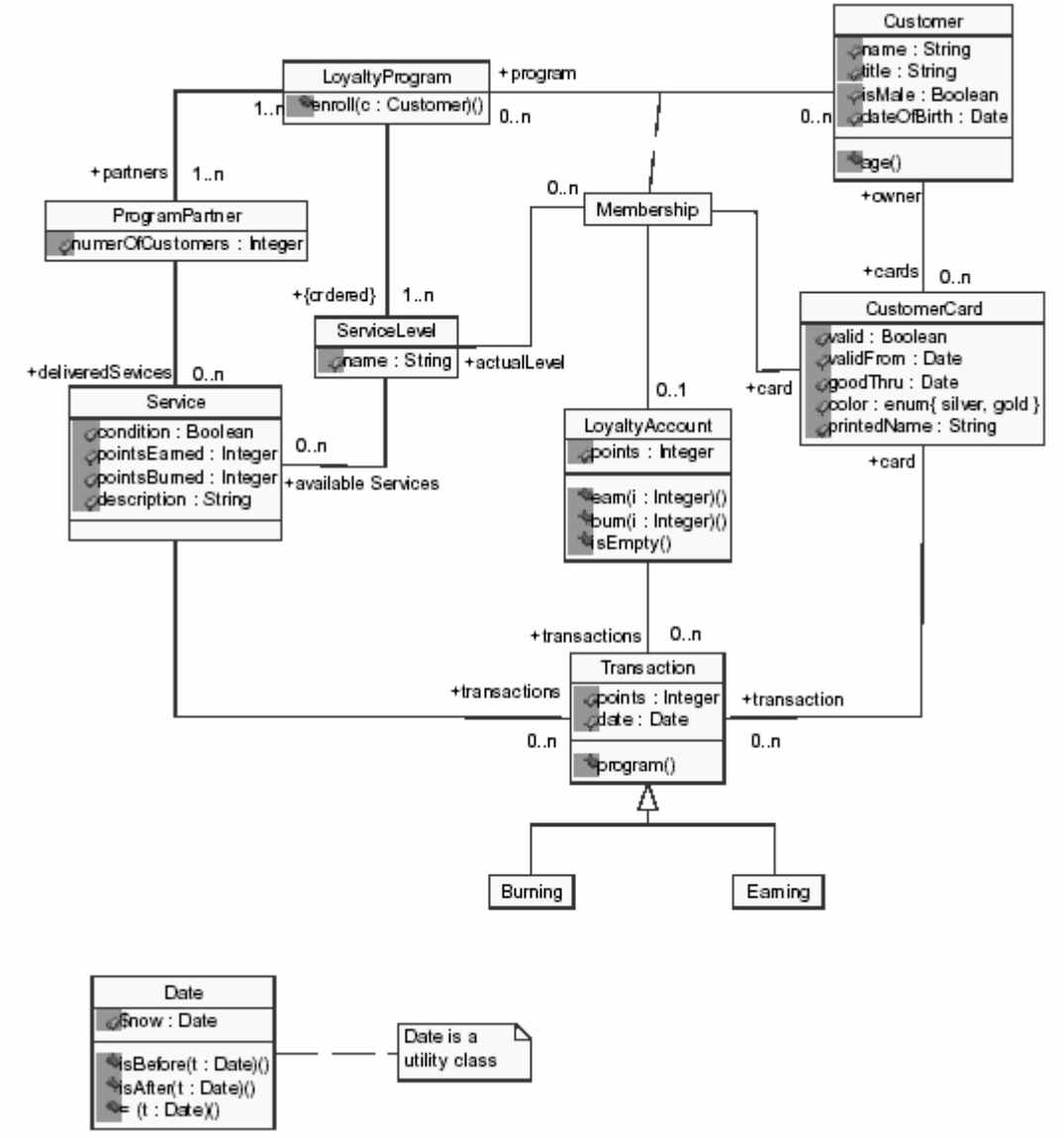


Abb. C.1: Royal & Loyal Beispielklassendiagramm

In diesem Diagramm wurde ein Computersystem für die fiktionale „Royal & Loyal“-Gesellschaft erstellt, welche in diesem Beispiel Bonusprogramme für verschiedene Unternehmen verwaltet. Die Hauptklasse dieses Klassendiagramms ist die Klasse *LoyaltyProgram*, die die verschiedenen Programme enthält. Als *ProgramPartner* wird ein Unternehmen bezeichnet, das den Kunden eine Mitgliedschaft in einem Bonusprogramm anbietet. Mehrere Unternehmen können gleichzeitig in einem

Programm sein, was bedeutet, dass ein Kunde eines LoyaltyProgram von allen angebotenen Diensten (*Service*) der Partner profitieren kann. Jeder Kunde, der in ein Programm eintritt, erhält eine Kundenkarte (*CustomerCard*). Diese Kunden werden durch die Klasse *Customer* repräsentiert. Innerhalb der einzelnen Programme können die Kunden Bonuspunkte erwerben, wobei jeder Partner selbst entscheidet, wie viele Punkte er für einen Service anbietet. Mit diesen Bonuspunkten können spezielle Dienste eines der Programmpartner erworben werden. Um die Punkte eines Kunden sammeln zu können, ist jede Mitgliedschaft mit einem *LoyaltyAccount* verbunden. Transaktionen, bei denen ein Kunde Bonuspunkte erwirbt, werden durch die Unterklasse *Earning* der Klasse *Transaction* abgebildet. Transaktionen, bei denen Bonuspunkte eingesetzt werden, werden durch die Unterklasse *Burned* von *Transaction* repräsentiert.

Anhang D

Installation

Zunächst wird die zip-Datei *JTestOCL.zip* im Arbeitverzeichnis oder auf irgendeinem Laufwerk (zum Beispiel: [z:/]) entpackt. Dabei wird automatisch ein Verzeichnis *JTestOCL* angelegt. Im diesem Verzeichnis gibt es fünf weitere Unterverzeichnisse: [\bin], [\example], [\jre], [\lib], [\src].

Die Applikation wird durch die Datei *JTestOCL.bat* Im Verzeichnis [\bin] gestartet.

Diese Batch-Datei hat folgenden Aufbau:

```
@echo off
:: ------------------------------------------------------------------
:: Before you run JTestOCL specify the location of the
:: directory where JTestOCL is installed
:: In most cases you do not need to change the settings below.
:: ------------------------------------------------------------------
SET JTestOCL_HOME=D:\JTestOCL
:: ------------------------------------------------------------------
:: In most cases you do not need to change the settings below.
:: ------------------------------------------------------------------
SET JAVA_EXE=%JTestOCL_HOME%\jre\bin\java.exe

IF NOT EXIST "%JAVA_EXE%" goto error

SET MAIN_CLASS_NAME=jtestocl.application.Application

SET JTestOCL_POPUP_WEIGHT=heavy

:: ------------------------------------------------------------------
:: You may specify your own JVM arguments in JTestOCL_JVM_ARGS variable.
:: ------------------------------------------------------------------
IF "%JTestOCL_JVM_ARGS%" == "" set JTestOCL_JVM_ARGS=-Xms16m -
Xmx128m -Dsun.java2d.noddraw=true

SET OLD_PATH=%PATH%
SET PATH=%JTestOCL_HOME%\bin;%PATH%

SET
CLASS_PATH=%JTestOCL_HOME%\lib\jtestocl.jar;%JTestOCL_HOME%\lib\log4j-
1.2.7.jar;%JTestOCL_HOME%\lib\junit.jar;%JTestOCL_HOME%\lib\ext;%JTestOCL_
HOME%\example
```

```
:: ---------------------------------------------------------------------
:: You may specify additional class paths in JTestOCL_CLASS_PATH variable.
:: It is a good idea to specify paths to your plugins in this variable.
:: ---------------------------------------------------------------------
IF NOT "%JTestOCL_CLASS_PATH%" == "" SET
CLASS_PATH=%CLASS_PATH%;%JTestOCL_CLASS_PATH%

"%JAVA_EXE%" %JVM_ARGS% -cp "%CLASS_PATH%"
%MAIN_CLASS_NAME% %*

SET PATH=%OLD_PATH%
goto end
:error
echo ---------------------------------------------------------------------
echo ERROR: cannot start JTestOCL.
echo No JRE found in JTestOCL installation directory
echo ---------------------------------------------------------------------
pause
:end
```

Vor dem ersten Start müssen in der Zeile

SET JTestOCL_HOME=D:\JTestOCL

die Laufwerkbezeichnung und das Verzeichnis eingetragen werden.

Die zu testende Klasse muss sich im System ClassPath oder im Verzeichnis [\example] befinden.

Der Testtreiber ist ebenfalls unter Linux/Unix lauffähig. Im Gegensatz zu Windows wird die Applikation hier jedoch mit *jtestocl.sh* gestartet.

Bedienungsanleitung

Zunächst erfolgt die Auswahl der Java-Klasse im Menü File / LoadClass (siehe Abbildung D.1).

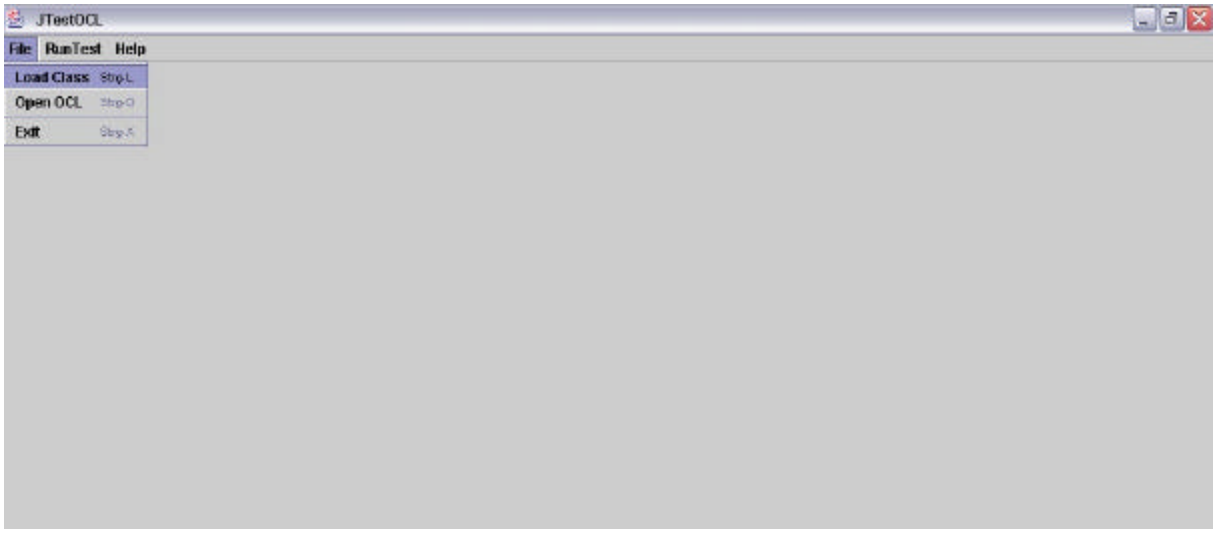


Abb. D.1: Load Class

In dem daraufhin angezeigten Dialogfeld *Load JavaClass* kann die zu testende Klasse ausgewählt und geladen werden (siehe Abbildung D.2).

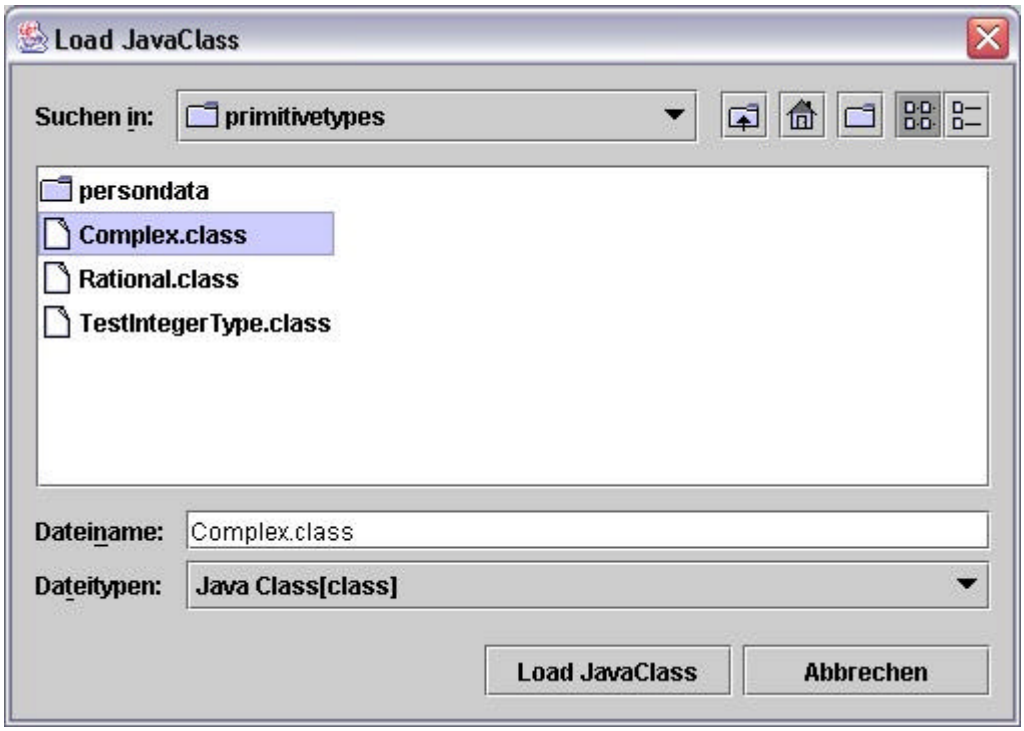


Abb. D.2: Java-Klasse auswählen

Nachdem die zu testende Klasse geladen wurde, kann die dazu gehörige OCL-Spezifikation in Form von *.ocl Datei geöffnet werden, um die Klassenspezifikation

bei der Parametergenerierung und bei der Auswertung der Testergebnisse einzubeziehen (siehe Abbildung D.3).

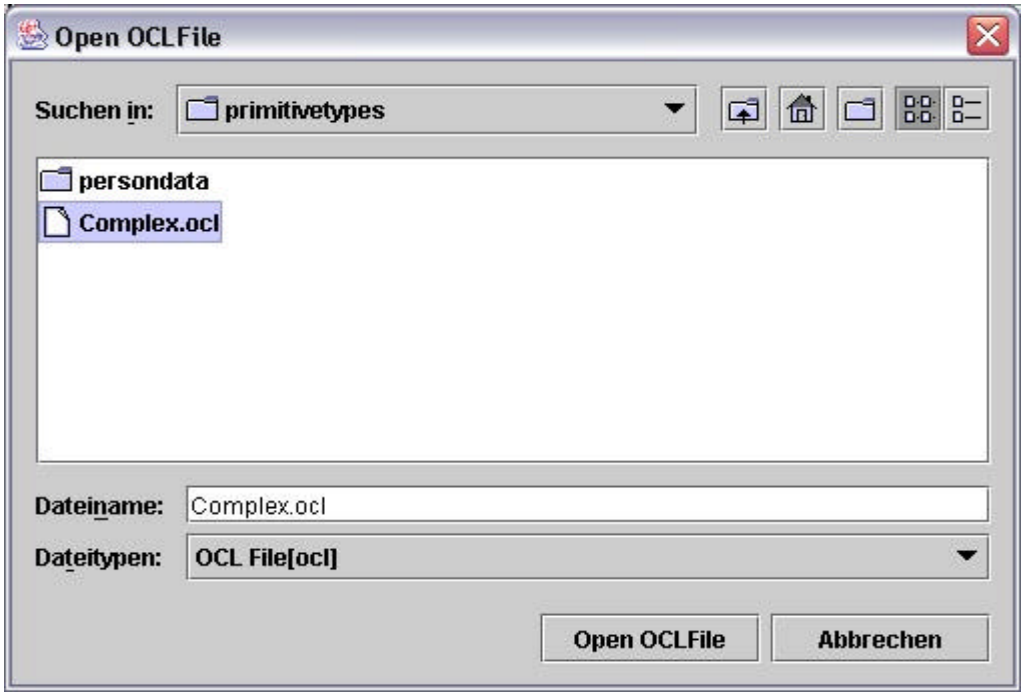


Abb. D.3: OCL-Datei öffnen

Es erscheint das Dialogfeld Info um den Tester zu informieren, ob die OCL-Syntax korrekt oder nicht korrekt ist (siehe Abbildung D.4).

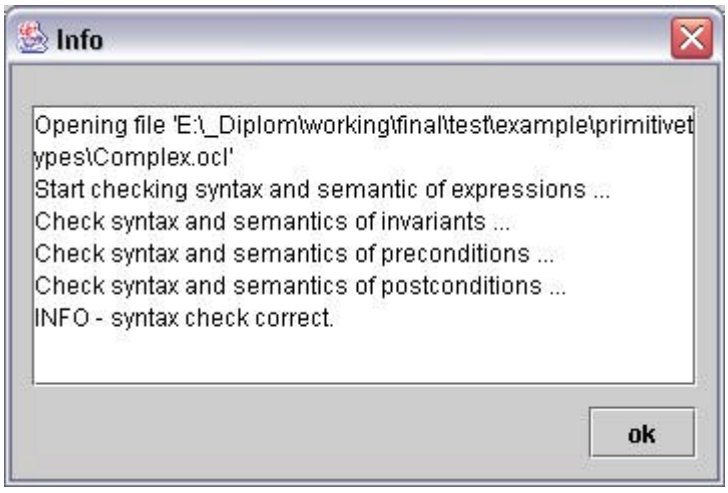


Abb. D.4: Info OCL-Spezifikation

Wenn die OCl-Syntax korrekt ist, wird die Information den dazugehörigen Feldern, Konstruktoren und Methoden zugewiesen (siehe Abbildung D.5).

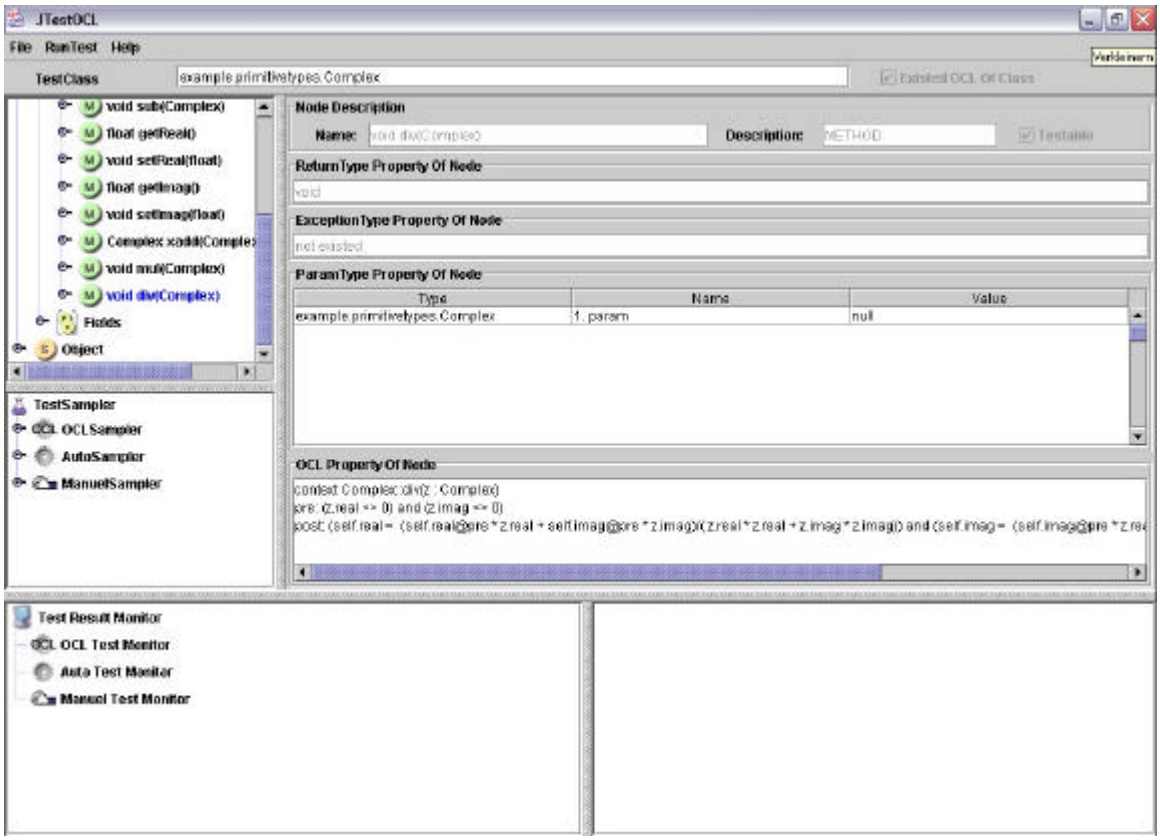


Abb. D.5: OCL-Spezifikation als Eigenschaften von Knoten

Der Start der Testfälle erfolgt durch die Auswahl im Menü RunTest (siehe Abbildung D.6).

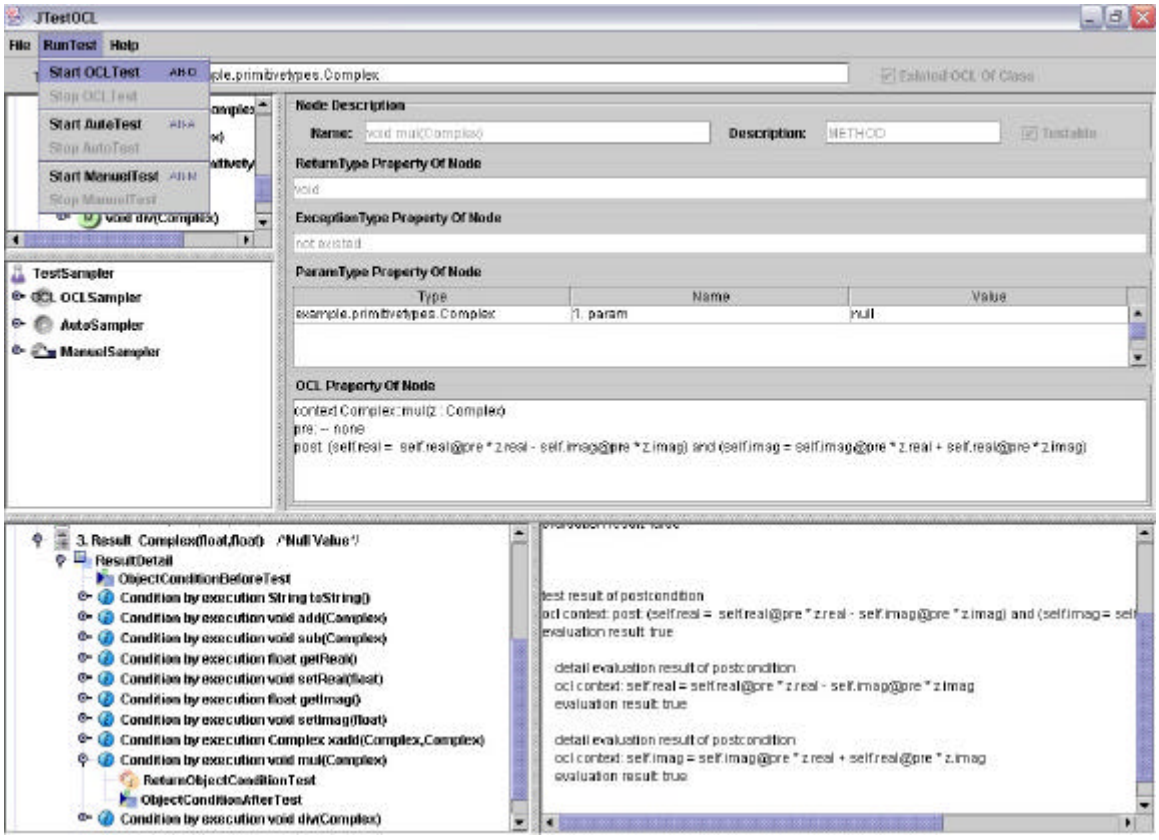


Abb. D.6: Start der Tests

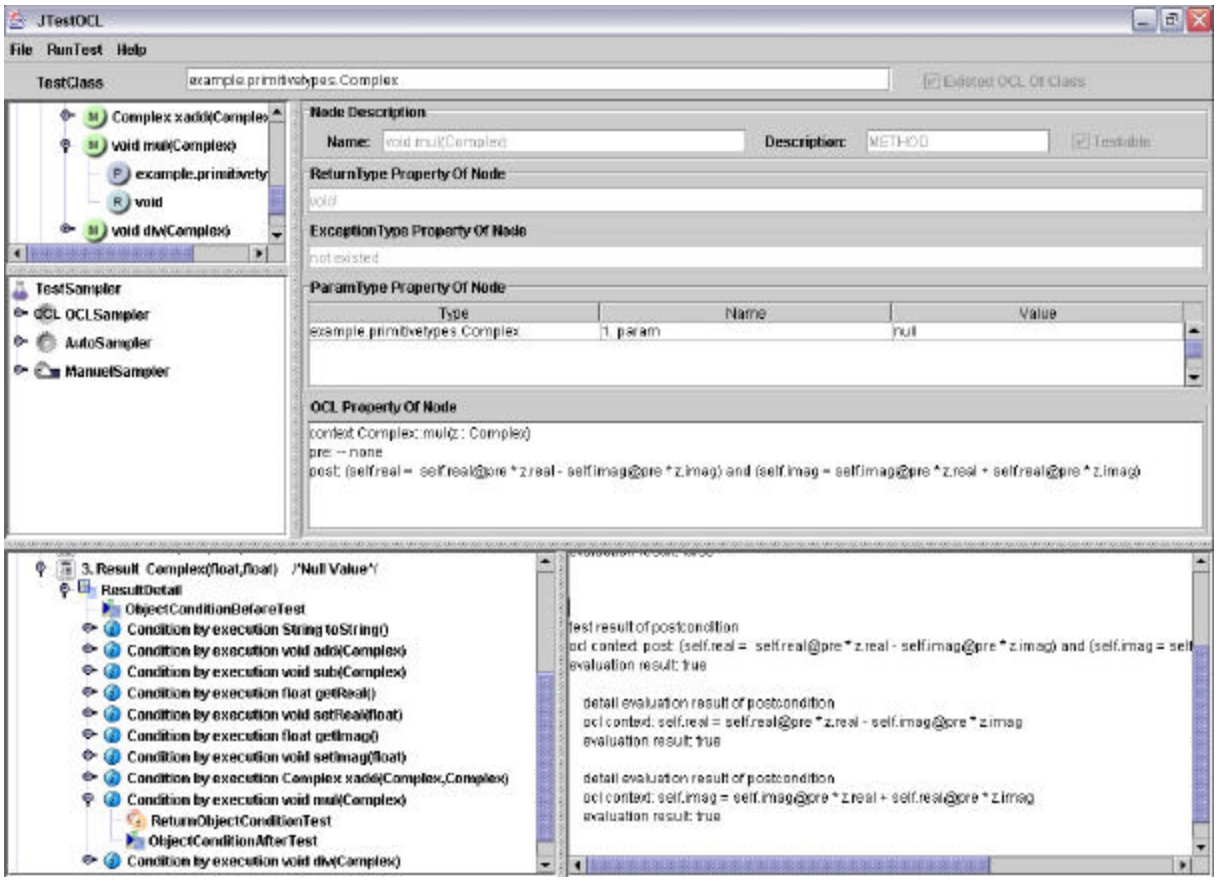


Abb. D.7: Testresultate

Nach dem Durchlauf der Testfälle kann der Tester das Testresultat bei *TreeResult* genau untersuchen. Dabei werden die Testergebnisse mit der Klassenspezifikation verglichen, falls diese existiert.

Anhang E

Glossar

Validation

Validation ist die Bestätigung durch Bereitstellung eines objektiven Nachweises, dass die Anforderungen für einen spezifischen beabsichtigten Gebrauch oder eine spezifische beabsichtigte Anwendung erfüllt worden sind.

Verifizierung

Verifikation ist die Bestätigung aufgrund einer Untersuchung und durch Bereitstellung eines Nachweises, dass festgelegte Forderungen erfüllt worden sind.

Falsifizierung

Falsifikation ist das Gegenteil der Verifizierung. Hierbei wird nach Ereignissen gesucht, die die eigenen Hypothesen widerlegen.

Modell

Ein Modell ist eine auf ein bestimmtes Ziel ausgerichtete, vereinfachte Darstellung der Funktion eines Gegenstands oder des Ablaufs eines Sachverhalts. Eine Untersuchung, Erforschung oder Planung wird durch diese Darstellung entweder erleichtert oder sogar erst möglich gemacht [Bal1991].

Modellbasiertes Testen bzw. die modellbasierte Software-Entwicklung

Modelle werden immer häufiger in der Softwareentwicklung eingesetzt. Besonders durch die weite Verbreitung von objektorientierten Sprachen wie Java, die einem methodischen Vorgehen mittels Modellierung des Systems durch Klassendiagramme entgegenkommen, bietet eine modellbasierte Entwicklung einen sinnvollen Ansatz, um eine Software zu erstellen. Ein besonders wichtiger Punkt ist, dass Software grundsätzlich aus zwei verschiedenen Modellen besteht. Auf der einen Seite dem reinen Strukturmodell eines Systems (vergleiche Klassenmodell unter Java), auf der anderen Seite seinem Verhaltensmodell. Das Strukturmodell bildet die Struktur der Komponenten ab. Im einzelnen werden hier ausschließlich die Klassen, Schnittstellen und Attribute, sowie die Beziehungen untereinander abgebildet, wobei das Ver-

haltensmodell die Art und Weise beschreibt, wie ein System sich zu verschiedenen Zeitpunkten verhält und wie es auf verschiedene Situationen reagiert (Methoden, Zustände, Interaktionen).

Syntax

System von Regeln, das die Form einer (Programmier-)Sprache beschreibt.

Parser

Der Parser organisiert die Reihenfolge von Zeichen (Worte, Nummern, Sonderzeichen, Kommentare) in hierarchisch – syntaktische Strukturen (Expression, Statement, Blöcke). Er formt die syntaktische Struktur einer Sprache.

Objekt

Ein Objekt ist eine Menge von Daten, die nur über wohldefinierte Operationen (Methoden) zugänglich sind.

Objekt-Klasse (Objekt-Typ)

Durch eine Objekt-Klasse werden Operationen und Datenstrukturen für gleichartige Objekt-Typen spezifiziert.

Objekt-Instanz

Eine Objekt-Instanz ist die Ausprägung eines Objektes, welches zu einer (vorher definierten) Klasse gehört.

Frameworks

Ein “Framework" (Rahmenwerk, Anwendungsgerüst) ist eine Menge von zusammengehörigen Klassen, die einen abstrakten Entwurf für eine Problemfamilie darstellen.

Seiteneffekte / seiteneffektfrei

Seiteneffekte sind Nebenwirkungen eines Aufrufs, die sich auf den Nachzustand eines Objektes auswirken.

GUI Capture/Playback Testtool

GUI Capture/Playback Testtools sind Programme, die bei der Entwicklung von Anwendungssoftware eingesetzt werden. Sie bieten dem Entwickler die Möglichkeit, automatische Tests anzufertigen, um ihm die Beurteilung der Qualität der entwickelten Software zu erleichtern. Es bedarf dazu keinerlei Veränderung an der zu testenden Software. Die Testfälle sind beliebig wiederholbar.

Wrapper-Klassen /-Objekt

Die Datenstrukturen, die in Java Verwendung finden, können nur Objekte aufnehmen. Daher stellt sich das Problem, wie primitive Datentypen zu diesen Containern hinzugefügt werden können. Die Klassenbibliothek bietet daher für jeden primitiven Datentyp eine entsprechende Wrapper-Klasse (auch »Ummantelungsklasse« oder »Envelope Class« genannt) an. Exemplare dieser Klassen kapseln je einen Wert des zugehörigen primitiven Typs. Zusätzlich zu dieser Eigenschaft bieten die Wrapper-Klassen Funktionen zum Zugriff auf den Wert und einige Umwandlungsfunktionen. Es existieren Wrapper-Klassen zu allen primitiven Datentypen und zusätzlich eine Klasse für void.