Entwicklung eines Werkzeugs zur Unterstützung zustandsbasierter Testverfahren für JAVA-Klassen

Diplomarbeit

zur Erlangung des Grades einer Diplom-Informatikerin

Dehla Sokenou
Matr.-Nr. 131044

Berlin, den 26.6.1998

Technische Universität Berlin
Fachbereich 13 - Informatik
Institut für Kommunikations- und Softwaretechnik
Fachgebiet Softwaretechnik
Prof. Dr. Stefan Jähnichen
Franklinstr. 28/29
10587 Berlin

Betreuer: Wilfried Koch

Inhaltsverzeichnis

1 Einleitung **5**

2 Zustandsbasierte Testverfahren **7**

2.1 Einführung 7
2.2 Traditionelle Testverfahren 7
2.3 Objektorientierte Testverfahren 9
2.4 Zustandsautomaten 11
2.4.1 Mealy-Automaten 11
2.4.2 Moore-Automaten 13
2.4.3 Harel-Automaten (Statecharts) 14
2.4.4 Implementation von Zustandsautomaten 24
2.5 Zustandsautomaten und Objektorientierung 28
2.5.1 Zustandsautomaten aus objektorientierter Sicht 28
2.5.2 Vererbung und Zustandsautomaten 30
2.6 Nutzung von Zustandsautomaten für den Test 37
2.6.1 Überprüfung der Spezifikation 37
2.6.2 Testdatenbildung 39
2.6.3 Objektüberwachung 39
2.6.4 Überdeckungsmessung 41
2.7 Testbarkeit 41

3 Systementwicklung **43**

3.1 Einführung 43
3.2 Konzeption 43
3.2.1 Funktionelle Anforderungen 43
3.2.1.1 Aufgabenangemessenheit 44
3.2.1.2 Selbstbeschreibungsfähigkeit 44
3.2.1.3 Steuerbarkeit 44
3.2.1.4 Erwartungskonformität 45
3.2.1.5 Fehlerrobustheit 45
3.2.1.6 Individualisierbarkeit 45
3.2.1.7 Erlernbarkeit 45
3.2.2 Benutzerschnittstellenbeschreibung 46

3.2.2.1 Projektverwaltung 46
3.2.2.2 Definition des Automaten 50
3.2.2.3 Generierung des Automaten 58
3.2.2.4 Ausgaben des Automatenobjekts 59
3.2.2.5 Statistik 60
3.3 Entwurf 61
3.3.1 Projektverwaltung 61
3.3.2 Klassenrepräsentation 63
3.3.3 Zustandsautomaten 66
3.3.4 Vererbte Zustandsautomaten 67
3.3.5 Java-Code-Generierung 71
3.3.6 Instrumentierung und generierter JAVA-Code 73
3.3.7 Statistikverwaltung 76
3.3.8 Fensterklassen 77
3.3.9 Internationalisierung und Oberflächenanpassung . . . 78
3.4 Implementierung 80
3.4.1 Wahl der Programmiersprache 81
3.4.2 Probleme bei der Implementierung 81
3.4.3 Differenzen zwischen Entwurf und Implementierung . 82

4 Zusammenfassung und Ausblick 83

Eidesstattliche Erklärung

Hiermit versichere ich an Eides statt, die vorliegende Arbeit selbständig und eigenhändig angefertigt zu haben.

Berlin, den 26.6.1998 Dehla Sokenou

Kapitel 1

Einleitung

Die Entwicklung objektorientierte Software nimmt einen immer größeren Platz in der Softwareentwicklung ein. Wurde zu Beginn der Schwerpunkt der Forschung auf Analyse und Entwurf gelegt, so gewinnt die Entwicklung und Erprobung objektorientierter Testverfahren immer mehr an Bedeutung. Dies ist zum einen darauf zurückzuführen, daß sich traditionelle Testverfahren nicht ohne weiteres auf objektorientierte Software anwenden lassen, zum anderen können die besonderen Eigenschaften objektorientierter Software den Test erschweren, anstatt ihn, wie zunächst angenommen, zu erleichtern.

Da Objekte in der Regel einen Zustand besitzen, nehmen zustandsbasierte Testverfahren beim Test objektorientierter Software eine wichtige Stellung ein. Jedoch ist die Anzahl der Werkzeuge, die zustandsbasierte Testverfahren unterstützen, gering.

Ein besonderer Aspekt objektorientierter Software, die Vererbung, wird von zustandsbasierten Testwerkzeugen bisher nicht unterstützt. Dabei ist besonders die typkonforme Vererbung ein bislang aus der Acht gelassener Aspekt. Die typkonforme Vererbung unterstützt die Sicht von Objekten als Implementierung abstrakter Datentypen. Außerdem kann typkonforme Vererbung von Zustandsautomaten den Aufwand beim zustandsbasierten Testen reduzieren.

Neben typkonformer Vererbung sollte ein Testwerkzeug zur Unterstützung zustandsbasierter Testverfahren auch die automatische Auswertung der anfallenden Daten des Tests wie Überdeckungsmessungen anbieten.

In dieser Arbeit soll ein Werkzeug für die Unterstützung zustandsbasierter Testverfahren vorgestellt werden, das den Ist-Zustand eines Objekts mit seinem Soll-Zustand vergleicht und Differenzen an den Benutzer meldet. Dazu wird aus der Spezifikation in Form eines Zustandsautomaten, der vom Benutzer eingegeben wird, eine Automatenklasse in JAVA generiert, die an die zu testende Klasse angebunden wird. Objekte der zu testenden Klasse

werden von den Objekten der Automatenklasse überwacht, alle Ereignisse und Zustandsänderungen werden protokolliert. Die anfallenden Daten werden automatisch ausgewertet.

Das entwickelte Werkzeug unterstützt die typkonforme Vererbung durch die Möglichkeit, Zustandsautomaten der Oberklasse an die Unterklasse zu vererben. Dabei basiert diese Vererbung von Zustandsautomaten auf der strengen Vererbung nach McGregor und Dyer [MD93].

Diese Arbeit führt im zweiten Kapitel in die theoretischen Grundlagen zustandsbasierter Testverfahren ein, darauf folgt im dritten Kapitel eine Beschreibung des entwickelten Systems.

Kapitel 2 stellt zunächst traditionelle Testverfahren vor, um dann die Schwierigkeiten des Tests objektorientierter Software zu beschreiben. Die verschiedenen Formen von Zustandsautomaten werden in Abschnitt 2.4 behandelt, Zustandsautomaten aus objektorientierter Sicht, insbesondere die Vererbung von Zustandsautomaten, finden sich in Abschnitt 2.5. Anschließend folgt ein Blick auf zustandsbasierte Testverfahren, ein Abschnitt über Testbarkeit schließt das Kapitel.

Kapitel 3 beschreibt die Systementwicklung des Werkzeugs zur Unterstützung zustandsbasierter Testverfahren in den Abschnitten zu Konzeption, Entwurf und Analyse.

Abschließend findet sich im vierten Kapitel eine Zusammenfassung der Ergebnisse und ein Ausblick auf zukünftige Entwicklungen.

Kapitel 2

Zustandsbasierte Testverfahren

2.1 Einführung

Traditionelle Testverfahren sind in der Literatur hinreichend beschrieben, Kategorien zugeordnet und auf ihre Güte untersucht worden. Für objektorientierte Testverfahren gilt dies nicht, auch lassen sich Testverfahren für konventionelle Software nicht ohne weiteres auf objektorientierte Software übertragen (siehe Kapitel 2.3).

Dieses Kapitel gibt zuerst einen kurzen Abriß über traditionelle Testverfahren (Abschnitt 2.2), und anschließend einen Überblick über den Stand der Technik bei Testverfahren für objektorientierte Software (Abschnitt 2.3). Bevor das zustandbasierte Testen beschrieben wird (Abschnitt 2.6), behandelt Abschnitt 2.4 Zustandsautomaten im allgemeinen und Abschnitt 2.5 die Vererbung von Zustandsautomaten im besonderen. Zuletzt erfolgt in Abschnitt 2.7 eine kurze Abhandlung über die Testbarkeit von Software.

2.2 Traditionelle Testverfahren

Berard [Ber93] klassifiziert (traditionelle) Testverfahren einmal nach White-Box-, Black-Box- und Gray-Box-Testverfahren und andererseits nach statischen und dynamischen Testverfahren. Zusätzlich werden formale Testverfahren definiert.

Eine weitergehende, sonst aber übereinstimmende Klassifizierung findet sich bei Liggesmeyer [Lig90].

Dynamische Testverfahren sind nach [Ber93] Verfahren, die zur Durchführung des Tests die Ausführung des Programms erfordern. Im Gegenteil

dazu können statische Testverfahren durchgeführt werden, ohne daß eine Ausführung des Programmcodes nötig ist.

White-Box-Testverfahren sind Testverfahren, die zur Testfallbildung die Struktur des zu testenden Programmcodes heranziehen. Zu den dynamischen White-Box-Verfahren zählen die kontrollfluß- und die datenflußorientierten Testverfahren.

Kontrollflußorientierte Testverfahren basieren auf der Kontrollstruktur der zu testenden Software. Dabei unterscheidet man zwischen verschiedenen Verfahren, die das Ziel verfolgen, eine möglichst große Überdeckung der vorhandenen Kontrollstrukturen zu erreichen. Der Anweisungsüberdeckungstest orientiert sich an den Anweisungen der zu testenden Software und zielt auf die mindestens einmalige Überdeckung aller Anweisungen, also die einmalige Ausführung aller Knoten im Kontrollflußgraphen, wobei der Kontrollflußgraph als gerichteter Graph mit einer endlichen Anzahl von Knoten, einem Start- und einem Endknoten, definiert ist. In der Standardform (für den kontrollflußorientierten Test) beschreiben die Knoten die Anweisungen, die gerichteten Kanten (Zweige) den möglichen Kontrollfluß zwischen den Knoten. In der Datenflußdarstellung werden zusätzliche Knoten für den Import und Export von Informationen über Parameter oder globale Variablen eingeführt. Der Zweigüberdeckungstest basiert auf der Überdeckung aller Zweige des Kontrollflußgraphen des zu testenden Programmcodes. Beim Pfadüberdeckungstest dient die möglichst große Überdeckung der Pfade des Kontrollflußgraphen als Testziel. Dabei ist eine vollständige Pfadüberdeckung meist nicht zu realisieren, da die Anzahl der Pfade in einem Programm aufgrund von Wiederholungsanweisungen so groß werden kann, daß die Anzahl der Testfälle zur Überdeckung aller Pfade zu einer technisch nicht handhabbaren Größe wächst, ohne feste Wiederholungsanzahl kann die Anzahl der Pfade gar unendlich sein.

Alle Überdeckungsmessungen haben gemeinsam, daß sich der Grad der Überdeckung aus dem Quotienten der Anzahl der ausgeführten Vertreter der jeweiligen Kontrollstruktur (Anweisung, Zweig, Pfad) und der Anzahl der insgesamt von dieser Kontrollstruktur vorhandenen Vertreter ermittelt ($\ddot{U}berdeckung\ C = \frac{Anzahl\ der\ ausgef\ddot{u}hrten\ Vertreter}{Anzahl\ aller\ Vertreter}$, nach [Lig90, Kapitel 2.2]). Dabei ergibt sich beim Pfadüberdeckungstest aufgrund der oben genannten Problematik die Schwierigkeit, daß bei einer unendlichen Anzahl der Pfade eine Überdeckungsmessung nicht möglich ist. Die Überdeckung aller Zweige beinhaltet die Überdeckung aller Knoten, die Überdeckung aller Pfade beinhaltet die Überdeckung aller Zweige.

Datenflußorientierte Testverfahren bilden Testfälle dagegen anhand der Zugriffe auf Variablen [Lig90, Kapitel 2.2]. Vertreter dieser Testkategorie werden auch als Defs/Uses-Kriterien bezeichnet. Dabei werden die Zugriffe auf Variablen einer von drei Kategorien zugeordnet: Wertzuweisung, Berechnung von Werten innerhalb eines Ausdrucks und in Prädikaten. Die

Kategorien bilden dabei die Grundlage für die Überdeckungsmessung.

Black-Box-Testverfahren sind Testverfahren, die zur Testfallbildung die Spezifikation des zu testenden Programms heranziehen. Zu den bekanntesten Black-Box-Verfahren zählt die funktionale Äquivalenzklassenbildung.

Hier werden anhand der Spezifikation Äquivalenzklassen der Eingabe- und Ausgabewerte gebildet. Es wird davon ausgegangen, daß Eingabewerte, die einer Äquivalenzklasse zugeordnet sind, ein identisches Verhalten der zu testenden Software verursachen. Für die Ausgabewerte einer Äquivalenzklasse müssen Eingabewerte gefunden werden, die diese Ausgabewerte erzeugen.

Es sollten immer Verfahren aus beiden vorgestellten Kategorien (White-Box-, Black-Box-Verfahren) angewendet werden, da White-Box-Verfahren aufgrund ihrer Orientierung an der Implementierung keine Fehler, die aus einer nicht vollständig oder nicht korrekt implementierten Spezifikation resultieren, entdecken können, Black-Box-Verfahren führen dagegen meist nicht zu einer vollständigen Überdeckung[1] des Programmcodes [Lig90, Kapitel 2.3].

Neben Black-Box- und White-Box-Testverfahren gibt es Testverfahren, die sich nicht eindeutig einer der beiden Kategorien zuordnen lassen, da sie sowohl auf die Spezifikation als auch auf die Implementierung angewendet werden. Diese bezeichnet man als Gray-Box-Verfahren.

Testverfahren für das Testen objektorientierter Software können nicht einfach aus den oben stehenden Verfahren abgeleitet werden, da gerade objektorientierte Software einige Besonderheiten aufweist, die den Einsatz traditioneller Testverfahren erschwert. Eine ausführliche Beschreibung dieser Besonderheiten und ein Überblick über speziell für objektorientierte Software geeignete Testverfahren findet sich im nächsten Abschnitt.

2.3 Objektorientierte Testverfahren

Traditionelle Testverfahren lassen sich nicht ohne Schwierigkeiten auf objektorientierte Software anwenden [Ber93, Sne95, SR92, Rüp97, Lig95, Pro95]. Gerade die Eigenschaften objektorientierter Software, die Analyse und Entwurf erleichtern und Wiederverwendung unterstützen, wie Abstraktion und Klassenbildung, Polymorphie, dynamisches Binden, Datenkapselung[2] und Vererbung, verursachen einen erhöhten Testaufwand [Pro95].

[1] Liggesmeyer spricht von einer durchschnittlichen Überdeckung von ca. 70% des Programmcodes durch Black-Box-Verfahren [Lig90]

[2] Einige Autoren (bzw. Übersetzer) setzen den Begriff Datenkapselung mit *Information Hiding* gleich [RBP+91, RBP+93], andere mit *Encapsulation.* Da die Einkapselung von Daten meist mit der Verbergung von Informationen über diese Daten verbunden ist, wird der Begriff hier wie in [RBP+91, RBP+93] gebraucht.

Objektorientierte Software schränkt gerade durch die Datenkapselung die Beobachtbarkeit von Fehlern ein, was sich auf die Testbarkeit objektorientierter Software negativ auswirkt [Bin94](siehe auch Kapitel 2.7). Als kleinste testbare Einheit wird im allgemeinen die Klasse angesehen, da Methoden durch gegenseitige Aufrufe und gemeinsam genutzte Attribute stark voneinander abhängig sind und nur mit einer Klasse als Testumgebung getestet werden können [Ber93, Rüp97]. Testfälle sind Sequenzen von Methoden, die an ein Objekt oder mehrere Objekte geschickt werden.

Im Gegensatz zur intuitiven Annahme, man müßte geerbte Methoden in Unterklassen nicht erneut testen, muß auch die Unterklasse vollständig getestet werden [PK90], da durch die enge Beziehung der Methoden einer Klasse jede redefinierte oder neu hinzugekommene Methode in der Unterklasse das Verhalten dieser in Vergleich zum Verhalten ihrer Oberklasse vollständig ändern kann.

Es gibt verschiedene Ansätze, die entstehenden Probleme beim Testen objektorientierter Software zu lösen. Mehrere Ansätze sind dabei zu erkennen.

Dem ersten Ansatz liegt die Sicht zugrunde, Klassen als Implementierungen von abstrakten Datentypen (ADTs) zu betrachten (siehe [DF91, HS96, GMH81]). Zur Testdatenbildung werden Sequenzen von Methoden gebildet, die sich aus den Axiomen von ADTs herleiten. Dabei müssen Sequenzen von Methoden, die äquivalent sind, Objekte erzeugen, die der gleichen Äquivalenzklasse angehören. Kirani und Tsai [KT94a, KT94b] benutzen eine von ihnen entwickelte Technik, die Bedingungen für die Bildung von Methoden-Sequenzen beschreibt, als Basis für die Testfallbildung. Andere Ansätze benutzen formale Spezifikationen, z.B. in der Spezifikationssprache Z [HP94], oder Graphen, bei denen jeder Knoten eine Nachricht und jede Kante die Möglichkeit einer Nachrichtenfolge repräsentiert [PBC93], als Grundlage für die Testfallbildung.

Der hier vorgestellte Ansatz betrachtet Objekte als Zustandsautomaten. Die Testdatenbildung erfolgt hierbei anhand der Zustände und Transitionen eines Zustandsdiagramm. In den folgenden Kapiteln wird dieser Ansatz ausführlich beschrieben.

Es fällt auf, daß alle hier vorgestellten Ansätze zu den Black-Box-Verfahren zu zählen sind, da sie die Spezifikation zur Testdatenbildung heranziehen. Das liegt einerseits daran, daß Methoden oft nur aus wenig Programmcode bestehen und andererseits an der Abhängigkeit von Methoden voneinander. Außerdem besitzen Objekte durch die Einkapselung und Verbergung von Methoden und Attributen eine Black-Box-Natur [Ber93], die einen Zugriff auf die Implementation verwehrt.

2.4 Zustandsautomaten

Bei der Definition von Zustandsautomaten handelt es um ein Konzept der Definitionsphase der Softwareentwicklung zur Beschreibung des Verhaltens eines Systems auf Eingaben [Bal96]. Dieses Konzept läßt sich jedoch in allen Phasen der Softwareentwicklung von der Spezifikation über Design und Implementation bis hin zur Testphase einsetzen [Zuc96].

Ein Zustandsautomat (auch endlicher Automat[3]) besteht aus einer endlichen Anzahl von Zuständen (internen Konfigurationen) und einer endlichen Anzahl von Übergängen zwischen diesen Zuständen. Dabei beinhaltet der aktuelle Zustand eines Systems implizit Informationen über zuvor erfolgte Eingaben und bestimmt das Verhalten des Systems auf folgende Eingaben. Ereignisse (Eingaben) können den Übergang in einen neuen Zustand oder eine Aktion auslösen und ein Ergebnis erzeugen. [Bal96].

In den folgenden Abschnitten 2.4.1, 2.4.2 und 2.4.3 werden die verschiedenen Definitionen und Notationen von Zustandsautomaten eingeführt, Abschnitt 2.4.4 stellt Möglichkeiten der Implementation von Zustandsautomaten vor. Betrachtungen zur Verwendung von Zustandsautomaten im Zusammenhang mit objektorientierter Softwareentwicklung, insbesondere im Zusammenhang mit Vererbung, finden sich im nächsten Kapitel (Kapitel 2.5).

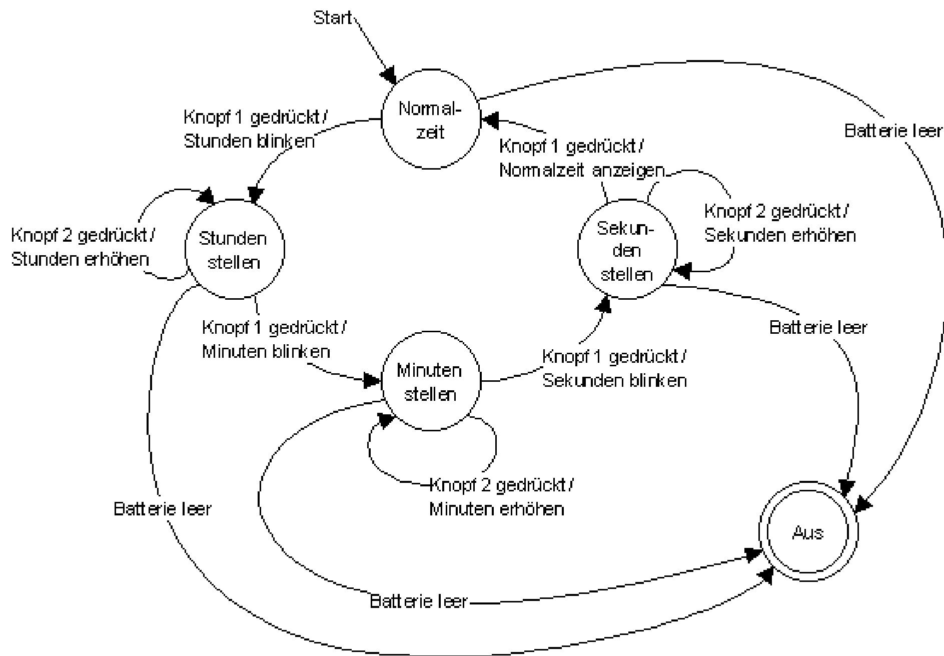
Obwohl Zustandsautomaten auch als Zustandstabellen oder Zustandsmatrizen dargestellt werden können, beschränken sich die folgenden Ausführungen auf die grafische Darstellung als Zustandsdiagramm[4]. In Mealy- und Moore-Automaten (2.4.1, 2.4.2) werden Zustände als Kreise bzw. Rechtecke dargestellt, Zustandsübergänge (Transitionen) als beschriftete Pfeile, der Startzustand durch einem Pfeil mit der Beschriftung "Start" gekennzeichnet und Endzustände durch doppelte Kreise bzw. doppelte Rechtecke. Die Notation für Harel-Automaten (2.4.3) stützt sich auf die von Harel [Har87] verwendete Notation (Rechtecke mit abgerundeten Ecken für die Zustände, Startzustände werden mit einem unbeschrifteten Pfeil markiert, keine ausgewiesenen Endzustände).

2.4.1 Mealy-Automaten

Bei den Mealy-Automaten werden die Zustandsübergänge durch beschriftete Pfeile dargestellt, bei denen die Beschriftung aus zwei Teilen besteht. Der erste Teil gibt die Eingabe (Ereignis), der zweite das Ergebnis der Eingabe (Ausgabe), das während des Übergangs ausgegeben wird, bzw. die Aktion, die während des Übergangs durchgeführt wird, an [Bal96].

[3] oder *FSM, finite state machine* (nach [Bal96])

[4] Das beinhaltet keine Einschränkung, da alle Darstellungsformen ineinander überführt werden können [Bal96].



Abbildung~2.1: Beispiel für einen Mealy-Automaten

Ein Beispiel für einen Mealy-Automaten findet sich in Abbildung 2.1[5]. Diese Abbildung zeigt die Modellierung des Stellens einer Uhr als Zustandsdiagramm. Dabei gibt es fünf Zustände: `Normalzeit`, `Stunden stellen`, `Minuten stellen`, `Sekunden stellen` und `Aus`. Durch das Ereignis `Start`, zum Beispiel das Einlegen einer Batterie in die Uhr, geht das System in den Anfangszustand `Normalzeit`. Das Drücken des Knopfes 1 löst jeweils einen Zustandsübergang außer im Zustand `Aus` aus. Zum Beispiel wird durch das Drücken des Knopfes 1 im Zustand `Normalzeit` der Zustandsübergang in den Zustand `Stunden stellen` ausgelöst. Gleichzeitig wird die Aktion `Stunden blinken` ausgeführt. Das Drücken des Knopfes zwei löst in keinem Zustand einen Zustandsübergang aus (es führt immer in den selben Zustand zurück), bewirkt aber außer im Zustand `Aus` eine Aktion. Zum Beispiel wird durch das Drücken des Knopfes zwei im Zustand `Stunden stellen` die Aktion `Stunden erhöhen` ausgeführt. In jeden Zustand führt das Ereignis `Batterie leer` zu einem Übergang in den Endzustand `Aus`. Dabei wird ersichtlich, daß nicht zu jeden Ereignis auch auch eine Ausgabe oder Aktion gehören muß, wie zum Beispiel beim Ereignis `Batterie leer`.

Die mathematische Definition eines Mealy-Automaten kann als ein Sechstupel beschrieben werden [Bal96]:

[5] Das Beispiel ist angelehnt an das Beispiel in [Bal96] für einen Mealy-Automaten.

M=(Q,Σ,Δ,δ,λ,q_0) mit

Q	endliche Menge von Zust nden
Σ	endliches Eingabealphabet
Δ	Ausgabealphabet
δ	bergangsfunktion, die Q×Σ auf Q abbildet
q_0	Anfangszustand
λ	Abbildung von Q×Σ nach Δ

Die Übergangsfunktion δ bildet einen Zustand aus Q und eine Eingabe aus Σ auf einen neuen Zustand aus Q ab. Die Abbildung λ nimmt einen Zustand aus Q und eine Eingabe aus Σ und erzeugt eine Ausgabe aus Δ.

2.4.2 Moore-Automaten

Im Gegensatz zu den Mealy-Automaten, wo die Aktionen oder Ereignisse an die Zustandsübergänge gebunden sind, sind sie bei den Moore-Automaten an die Zustände gebunden [Bal96].

Bis auf die Abbildung λ ist die mathematische Definition identisch, diese bildet hier eine Zustand aus Q auf eine Ausgabe aus Δ ab:

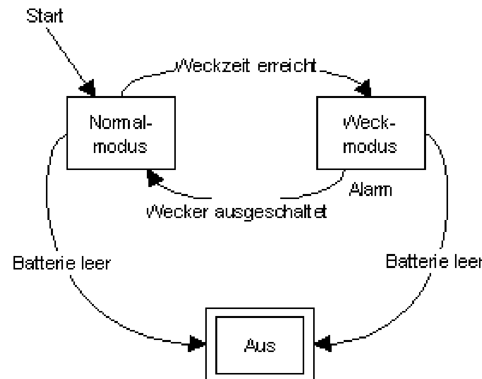
M=(Q,Σ,Δ,δ,λ,q_0) mit

Q	endliche Menge von Zust nden
Σ	endliches Eingabealphabet
Δ	Ausgabealphabet
δ	bergangsfunktion, die Q×Σ auf Q abbildet
q_0	Anfangszustand
λ	Abbildung von Q nach Δ

Mealy- und Moore-Automaten sind äquivalent. Sie können ineinander überführt werden [Bal96].

Allerdings ist die Voraussetzung für die Modellierung eines Moore-Automaten, daß in jeden Zustand nur genau eine Ausgabe bzw. Aktion erfolgt [Bal96]. Das Beispiel in Abbildung 2.1 ist kein geeignetes Beispiel, um es als Moore-Automat darzustellen, da im Zustand `Stunden stellen` sowohl die Aktion `Stunden blinken` als auch die Aktion `Stunden erhöhen` auftreten kann. Man müßte für die Modellierung als Moore-Automat weitere Zustände einführen, um eine äquivalente Darstellung zu erreichen.

Abbildung 2.2 zeigt als Beispiel für einen Moore-Automaten die Modellierung der Weckfunktion einer Uhr. Es gibt die drei Zustände `Normalmodus`, `Weckmodus` und `Aus`. Aus den erstgenannten beiden Zuständen führt wieder analog zum Beispiel eines Mealy-Automaten in Abbildung 2.1 jeweils das Ereignis `Batterie leer` in den Endzustand `Aus`. Der Zustand `Weckmodus` löst gleichzeitig das Ereignis `Alarm` aus.



Abbildung~2.2: Beispiel für einen Moore-Automaten

Zur Modellierung des Verhaltens eines Systems eignet sich also ein Mealy-Automat besser, wenn es sich um ein System handelt, bei dem die Aktionen und Ausgaben während der Zustandsübergänge ausgelöst bzw. erzeugt werden, Moore-Automaten sind besser für Systeme geeignet, wo eine Aktion oder eine Ausgabe ausgelöst bzw. erzeugt wird, während sich das System in einem bestimmten Zustand befindet.

Handelt es sich um Systeme, auf die sowohl das eine als auch das andere zutrifft, bieten weder Mealy- noch Moore-Automaten die ideale Lösung. Eine Erweiterung der vorgestellten klassischen Formen von Zustandsautomaten sind die Harel-Automaten oder Statecharts, die neben anderen Erweiterungen Mealy- und Moore-Automaten kombinieren[6].

2.4.3 Harel-Automaten (Statecharts)

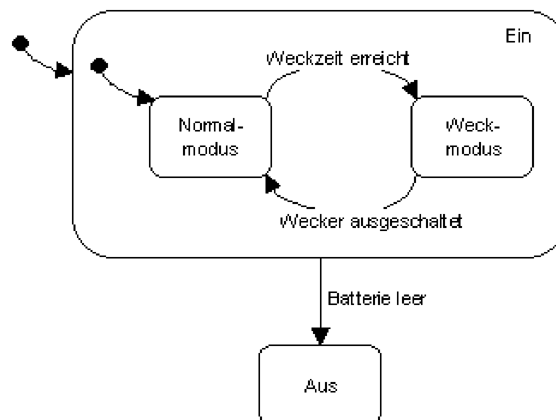
Da klassische Zustandsautomaten wie die in den Abschnitten 2.4.1 und 2.4.2 beschriebenen Mealy- und Moore-Automaten nicht allen Anforderungen an die Modellierung des Verhaltens eines Systems genügen, erweiterte Harel diese in [Har87] zu den sogenannten Statecharts oder Harel-Automaten (siehe auch [Bal96], [Mar95]). Dabei führte er Konzepte ein, die im folgenden erläutert werden[7]:

- Hierarchie (*Clustering*, *Refinement*)
- Zustände mit Gedächtnis (*History*)
- Nebenläufige Zustandsdiagramme (*Independency*, *Concurrency*)
- Zeitliche Aspekte (*Delays*, *Timeouts*)

[6] In [Bal96] werden sie deshalb auch als *hybride Zustandsautomaten* bezeichnet.
[7] Die von Harel [Har87] benutzten Originalbegriffe finden sich in Klammern.

- Bedingungen (*Condition*)
- Aktionen und Aktivitäten (*Actions*, *Activities*)

Einige Beispiele sollen im folgenden die einzelnen Konzepte vorstellen[8]. Harel schlägt in [Har87] noch Erweiterungen wie parametrisierte oder überlappende Zustände vor, auf die hier nicht näher eingegangen werden soll, da sie nicht Bestandteil seiner Statecharts sind.



Abbildung~2.3: Beispiel für Hierarchie

Grundkonzept der Statecharts ist die Hierarchisierung von Zuständen. Dabei werden Zustände zu einem Oberzustand zusammengefaßt (*Clustering*) oder Zustände in Unterzustände zerlegt (*Refinement*). Dabei handelt es sich beim *Clustering* um ein Konzept, das die Entwicklung eines Zustandsautomaten von unten nach oben (bottom-up), beim *Refinement* um eines, das die Entwicklung von oben nach unten (top-down) durchführt [Har87].

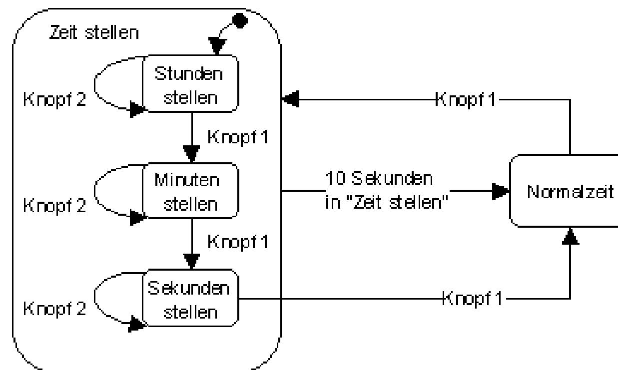
Befindet sich das System in einem Oberzustand, so heißt das, daß es sich in genau einem der Unterzustände befindet. Bei dem Oberzustand handelt es sich also um eine ODER-Beziehung der Unterzustände.

Betrachtet man die Beispiele aus den Abbildungen 2.1 und 2.2, so stellt man fest, daß in beiden das Ereignis `Batterie leer` von allen anderen Zuständen in den Zustand `Aus` führt. Man kann diese Zustände jetzt zu einem Oberzustand, zu Beispiel dem Zustand `Ein` zusammenfassen.

Dies wurde in Abbildung 2.3 für den Moore-Automaten, der in Abbildung 2.2 dargestellt wurde, vorgenommen. Da Harel Endzustände nicht explizit ausweist [Bal96], wird der Zustand `Aus` wie ein normaler Zustand behandelt. Die Zustände `Normalmodus` und `Weckmodus` wurden zum Zustand

[8] Alle Definitionen beziehen sich auf [Har87], soweit nichts anderes angegeben ist.

`Ein` zusammengefaßt. Durch die unbeschrifteten Pfeile wird der Startzustand gekennzeichnet. Wird die Uhr zum Beispiel durch das Einlegen einer Batterie eingeschaltet, so geht sie in den Zustand `Ein` über. Dieser ist jedoch nur ein abstrakter Zustand. Wird dieser Zustand vom System angenommen, so ist der Zustand, der innerhalb des Oberzustands `Ein` angenommen wird, der Zustand `Normalmodus`, wieder gekennzeichnet durch den unbeschrifteten Pfeil.



Abbildung~2.4: Beispiel für Zustandübergänge zwischen verschiedenen Hierarchieebenen

Zustandsübergänge sind nicht auf eine Ebene wie in dem Beispiel in Abbildung 2.3 beschränkt, sondern können auch über verschiedene Hierarchieebenen hinweg gehen. Zustandsübergänge können auf jeder Hierarchieebene ihren Ursprung haben und auf jeder Hierarchieebene enden [Bal96].

Abbildung 2.4 veranschaulicht in einem Beispiel Zustandsübergänge zwischen verschiedenen Hierarchieebenen. Die Ereignisse `Knopf 1` im Zustand `Normalzeit` und `10 Sekunden in Zeit stellen` im Zustand `Zeit stellen` lösen Zustandsübergänge innerhalb einer Hierarchieebene aus. Das Ereignis `Knopf 1` im Zustand `Zeit stellen` dagegen löst einen Zustandübergang zwischen verschiedenen Hierarchieebenen aus, die Transition[9] führt vom Zustand `Sekunden stellen` in den Zustand `Normalzeit`. Da es sich bei dem Zustand `Sekunden stellen` um einen Unterzustand von `Zeit stellen` handelt und sich die Zustände `Zeit stellen` und `Normalzeit` auf einer Hierarchieebene befinden, führt die Transition von einer Hierarchieebene auf die nächsthöhere.

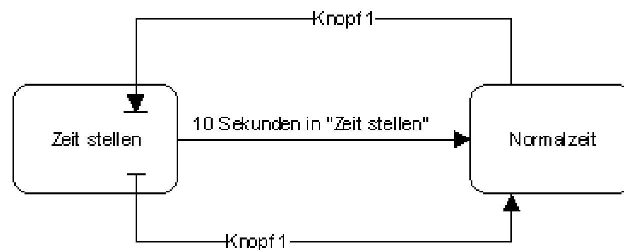
Harel beschreibt in [Har87][10] verschiedene Abstraktionsebenen der Hierarchiesierung.

[9] Synonym für Zustandsübergang

[10] Harel spricht auch vom Herein- und Herauszoomen innerhalb des Zustanddiagramms [Har87].

Abbildung~2.5: Beispiel einer Abstraktion von Zuständen

Abbildung 2.5 stellt das Beispiel aus Abbildung 2.4 nur noch mit der obersten Hierarchieebene dar. Um zu verdeutlichen, daß das Ereignis `Knopf 1` nur in einem Unterzustand von `Zeit stellen` einen Zustandsübergang in den Zustand `Normalzeit` auslöst, beginnt der entsprechende Pfeil in der Abbildung innerhalb des Zustandes `Zeit stellen` und wird an seinem Ursprung durch einen kleinen Strich rechtwinklig zum Pfeil gekennzeichnet.

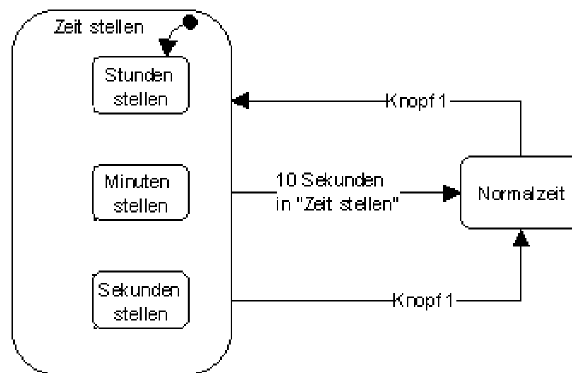


Abbildung~2.6: Beispiel einer Abstraktion von Zuständen

Analoges gilt für einen Übergang von einem Oberzustand in einen Unterzustand. Angenommen, das Beispiel in Abbildung 2.4 wäre so gestaltet, daß der Zustand `Stunden stellen` nicht als expliziter Anfangszustand gekennzeichnet wäre, sondern der Zustandsübergang durch das Ereignis `Knopf 1` von Zustand `Normalzeit` in den Zustand `Stunden stellen` direkt durch einen Pfeil gekennzeichnet, dann würde das Beispiel in Abbildung 2.5 wie in Abbildung 2.6 dargestellt aussehen. Der entsprechende Zustandsübergang ist durch einen Pfeil gekennzeichnet, der innerhalb des Oberzustandes endet und dessen Pfeilspitze durch einen kleinen, senkrecht zum Pfeil stehenden Strich markiert wird.

Abbildung 2.7 zeigt eine größere Abstraktion des Beispiels. Zustandsübergänge zwischen verschiedenen Ebenen werden nicht explizit ausgewiesen.

Abbildung~2.7: Beispiel einer Abstraktion von Zuständen



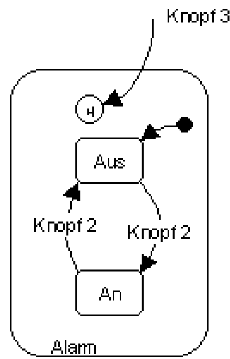
Abbildung~2.8: Beispiel einer Abstraktion von Zuständen

Abbildung 2.8 stellt die Unterzustände des Zustandes `Zeit stellen` dar, ohne die Zustandsübergänge innerhalb dieses Zustands zu wiederzugeben.

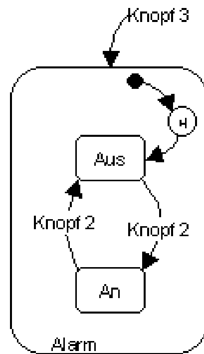
Alle bisher vorgestellten Beispiele hatten zur Grundlage, daß es einen ausgezeichneten Anfangszustand innerhalb des Oberzustandes gibt, der bei jedem erneuten Eintritt in den Oberzustand eingenommen wird.

Um zu beschreiben, daß der jeweils letzte Zustand nach dem Verlassen des Oberzustands beim erneuten Eintritt in den Oberzustand angenommen wird, führte Harel das Konzept der Zustände mit Gedächtnis (Historie, *History*) ein. Zustände mit Gedächtnis werden durch ein großes H in einem Kreis gekennzeichnet.

Abbildung 2.9 verdeutlicht das Konzept anhand eines Beispiels. Der Alarm einer Uhr kann ein- oder ausgeschaltet sein. Im Beispiel stellt der Zustand `Alarm` einen Oberzustand für die Zustände `(Alarm) An` und `(Alarm) Aus` dar. Beim ersten Eintritt in den Zustand `Alarm` befindet sich die Uhr im Zustand `(Alarm) Aus`. Durch Drücken von `Knopf 2` geht die Uhr in den Zustand `(Alarm) An` über.



Abbildung~2.9: Beispiel für Historie

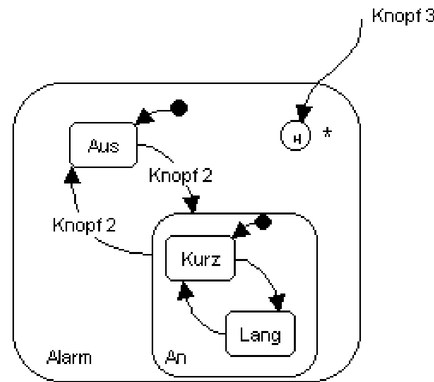


Abbildung~2.10: Analoges Beispiel für Historie

Abbildung 2.10 zeigt eine analoge Darstellung des Beispiels aus Abbildung 2.9. Beide Darstellungsformen sind möglich.

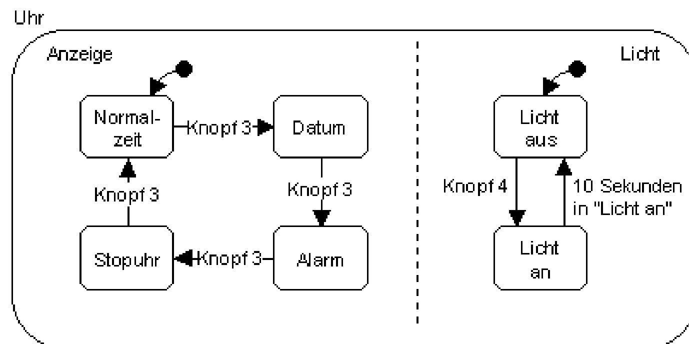
Die Historie gilt immer nur auf der Hierarchieebene, in der sie auftaucht. Möchte man Historie auch auf darunterliegende Hierarchieebenen anwenden, so gibt es zwei Möglichkeiten. Man kann einmal alle Zustände mit Gedächtnis einzeln markieren, also die entsprechenden Unterzustände wieder mit einem großen H in einem Kreis markieren. Die zweite Möglichkeit ist, das große H im Kreis noch mit einem Stern (*) zu versehen, dann gilt die Historie rekursiv für alle Unterzustände.

Abbildung 2.11 zeigt ein Beispiel für rekursive Historie. Dabei wurde der Zustand `An` aus dem Beispiel in Abbildung 2.9 in zwei unterschiedliche zur Auswahl stehende Alarmarten unterteilt (`Kurz` für kurzen und `Lang` für langen Alarm). Auch im Zustand `An` wird wie auf der darüberliegenden Hierarchieebene im Zustand `Alarm` der Unterzustand beim letzten Verlassen des Zustands vermerkt und beim erneuten Eintritt angenommen. Dies gilt



Abbildung~2.11: Beispiel für rekursive Historie

auch für alle weiteren Unterzustände, auch für Unterzustände der Zustände `Kurz` und `Lang`.



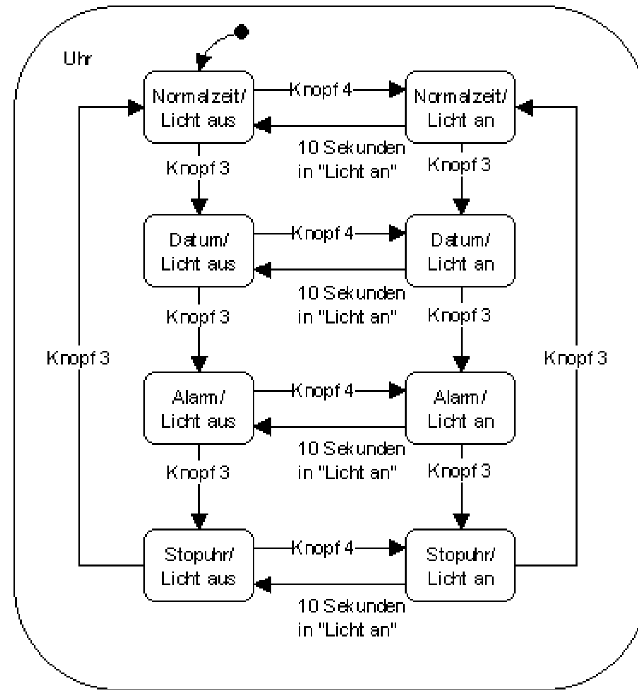
Abbildung~2.12: Beispiel für nebenläufige Zustandsdiagramme

Ein weiteres Konzept der Statecharts von Harel ist die Modellierung nebenläufiger, voneinander unabhängiger Zustandsdiagramme für ein zu beschreibendes System. Gerade dieses Konzept sorgt für Übersichtlichkeit, wie das folgende Beispiel deutlich macht.

Wenn man annimmt, daß eine Uhr neben allen anderen Funktionen noch die Mö glichkeit besitzt, Licht einzuschalten, das sich dann nach 10 Sekunden wieder ausschaltet, so ist diese Funktion und sind die Zustände `Licht an` und `Licht aus` vollkommen unabhängig von den anderen Zuständen der Uhr, da das Licht zu jeder Zeit eingeschaltet werden kann und keinen Einfluß

auf andere Zustände der Uhr hat[11]. DieUhr befindet sich damit gleichzeitig in zwei voneinander unabhängigen Zuständen. Ein solches Verhalten läßt sich wie in dem Beispiel in Abbildung 2.12 beschreiben.

Nebenläufige Zustandsdiagramme werden wie in dem Beispiel illustriert durch eine senkrechte unterbrochenen Linie dargestellt.

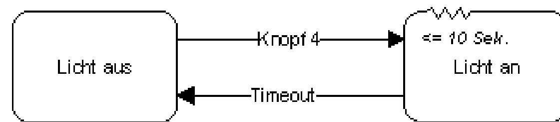


Abbildung~2.13: Beispiel für aufgelöste Nebenläufigkeit

Die nebenläufigen unabhängigen Diagramme lassen sich auch zu einem Diagramm zusammenfassen, wie es in Abbildung 2.13 für das Beispiel aus Abbildung 2.12 vorgenommen wurde. Dabei sieht man, daß viel von der vorherigen Übersichtlichkeit verloren geht. Außerdem kann die Anzahl der Zustände in einem zusammengefaßten Diagramm sehr groß werden, da sie sich aus der Multiplikation der Zustände aller nebenläufigenden Diagramme berechnet. In unserem Beispiel werden aus vier Zuständen in einem Diagramm und zweien im anderen acht Zustände im zusammengefaßten Diagramm.

Bei einigen der betrachteten Beispiele fällt auf, daß ein Zustand verlassen wird, ohne daß ein explizites Ereignis eintritt, sondern das System nach einer gewissen Dauer in einen anderen Zustand übergeht. Ein Beipiel dafür ist das

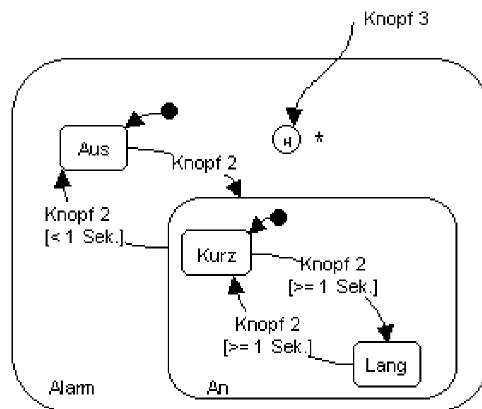
[11]Das Einschalten des Lichts verursacht keine Änderung bestehender Zustände, es ergänzt sie nur.



Abbildung~2.14: Beispiel für Timeout

Licht in Abbildung 2.12. Dort wird der Zustand `Licht an` verlassen, wenn sich die Uhr länger als 10 Sekunden in diesem Zustand befindet, das Licht schaltet sich also nach 10 Sekunden wieder aus. Harel bezeichnet dieses Verhalten eines Systems als *Timeout*. Ein weiteres dazugehörendes Konzept ist das der *Delays*. Hier kann ein Ereignis erst nach einer bestimmten Zeit eine Wirkung, zum Beispiel in Form eines Zustandsübergangs haben.

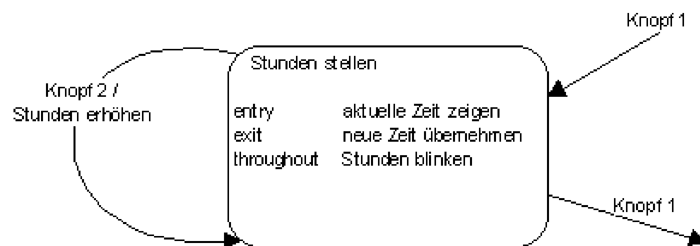
Für beide Konzepte benutzt Harel ein verändertes Rechteck für die Darstellung eines Zustands, wie es in Abbildung 2.14 gezeigt ist. Innerhalb des Zustands `Licht an` ist in diesem Fall eine obere Grenze vermerkt (`<= 10 Sek.`), die besagt, wann der Zustand spätestens verlassen wird, hier nach 10 Sekunden. Es handelt sich also um ein *Timeout*, das nach dem Erreichen der oberen Grenze einen Zustandsübergang, gekennzeichnet durch den mit `Timeout` beschrifteten Pfeil, in den Zustand `Licht an` auslöst. *Delays* werden als untere Grenze angegeben und mit > `untere Grenze` gekennzeichnet.



Abbildung~2.15: Beispiel für bedingte Zustandsübergänge

Oft ist es nötig, zu einem Ereignis eine Bedingung anzugeben. Nur wenn das Ereignis eintritt und diese Bedingung erfüllt ist, kann ein Zustandsübergang erfolgen. Man spricht auch von bedingten Zustandsübergängen. Bedingungen stehen hinter dem Ereignis in eckigen Klammern.

Abbildung 2.15 zeigt ein Beispiel für einen bedingten Zustandsübergang. Hier wurde das Beispiel aus Abbildung 2.11 aufgegriffen und leicht modifiziert. Jetzt löst nicht das einfache Drücken des `Knopf 2` den Zustandsübergang vom Zustand `An` in den Zustand `Aus` aus, sondern nur unter der Bedingung, daß der Knopf kurzzeitig, also nicht länger als eine Sekunde (`< 1 Sek.`) gedrückt wird. Der andere Fall, also ein Drücken des Knopfes länger oder gleich einer Sekunde (`>= 1 Sek.`), führt die inneren Zustände des Zustandes `An` ineinander über.



Abbildung~2.16: Beispiel für Aktionen und Aktivitäten

Bei Harel gibt es die Einschränkungen in Bezug auf Aktionen wie bei den Moore- und Mealy-Automaten nicht. Hier können Aktionen sowohl an den Zustand gebunden sein als auch an die Transitionen.

Dabei unterscheidet Harel bei zustandsgebundenen Aktionen zwischen drei verschiedenen Arten. Aktionen können bei Eintritt in einen Zustand oder bei Verlassen des Zustands durchgeführt werden oder während des gesamten Verbleibens innerhalb eines Zustands. Im letzteren Fall spricht Harel nicht von Aktionen, sondern von Aktivitäten, da sie von dauerhafter Natur sind, während Aktionen quasi keine Zeit kosten[12]. Aktionen werden bei Eintritt in einen Zustand gestartet und bei Austritt wieder gestoppt.

Aktionen, die an Zustandsübergänge gebunden sind, werden wie bei den Mealy-Automaten nach dem Ereignis, durch einen Schrägstrich von diesem getrennt an den Pfeil für die Transition geschrieben. Aktionen und Aktivitäten, die an den Zustand gebunden sind, werden in das Rechteck für den Zustand hineingeschrieben. Dabei werden Aktionen, die beim Eintritt ausgeführt werden, mit dem Wort `entry` eingeleitet, Aktionen beim Austritt mit `exit` und Aktivitäten während der Dauer des Verbleibens in dem Zustand mit `throughout`. Die Modellierung der Aktionen während eines Zustands ist also genauer definierbar als mit Moore-Automaten.

[12] *We shall reserve the word action for split-second happenings, instantaneous occurences that take ideally zero time. [...] An Activity always takes a nonzero amount of time [...].* [Har87]

Als Beipiel (Abbildung 2.16) dient der Zustand `Stunden stellen`. Er wird durch Drücken von `Knopf 1` angenommen und durch das gleiche Ereignis wieder verlassen. Bei Eintritt in diesen Zustand wird als Eintrittsaktion die aktuelle Zeit angezeigt (`entry aktuelle Zeit zeigen`). Während des Verbleibens in diesem Zustand blinken die Stunden die gesamte Zeit (`throughout Stunden blinken`). Beim Verlassen des Zustands wird die neu eingestellte Zeit übernommen (`exit neue Zeit übernehmen`). Durch den Drücken von `Knopf 2` wird der Zustand in sich selbst überführt, gleichzeitig wird die Aktion `Stunden erhöhen` ausgeführt.

Die Erweiterungen von Harel erleichtern das Modellieren von Zustandsautomaten erheblich und ermöglichen eine Definition des Systemverhaltens, die mit Moore- und Mealy-Automaten auf diese Art und Weise nicht möglich ist.

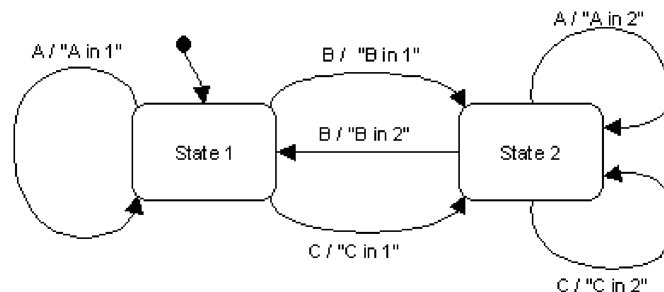
Die Schachtelung von Zuständen und die Einführung von unabhängigen Zustandsdiagrammen dienen in erster Hinsicht der Übersichtlichkeit des Zustandsdiagramms, der Abstraktion und der Erleichterung des Entwurfs durch den Bottom-Up/Top-Down-Ansatz.

Bedingungen ermöglichen die Modellierung eines Systems, in der ein Ereignis in Abhängigkeit von Bedingungen ein unterschiedliches Systemverhalten zur Folge hat.

Die Kombination von Moore- und Mealy-Automaten in Hinblick auf Aktionen (und Aktivitäten) läßt die Modellierung sowohl von Systemen zu, bei denen die Aktionen an den Zustandsübergang gebunden sind, als auch von Systemen, bei denen die Aktionen ausgeführt werden, während sich das System in einem Zustand befindet.

2.4.4 Implementation von Zustandsautomaten

Marick beschreibt in [Mar95] drei typische Varianten zur Implementierung von Zustandsautomaten. Das folgende Beispiel und die Implementationen lehnen sich an Maricks Ausführungen an.



Abbildung~2.17: Beispiel für einen Zustandsautomaten

Abbildung 2.17 stellt einen Mealy-Automaten dar. Es gibt die Zustände `State 1` und `State 2` und die Ereignisse `A`, `B` und `C`. Jedes der Ereignisse erzeugt als Ausgabe eine Nachricht, die angibt, welches Ereignis in welchem Zustand ausgelöst wurde, zum Beispiel die Nachricht `''A in 1''`, wenn das Ereignis `A` im Zustand `State 1` ausgelöst wurde.

Folgende drei Implementationsmöglichkeiten schlägt Marick vor [Mar95][13]:

1. Für jedes Ereignis wird eine Funktion implementiert:

```
void eventB()
{
   switch(State)
   {
      case state1:
         printf("B in 1");
         State = state2;
         break;
      case state2:
         printf("B in 2");
         State = state2;
         break;
      default:
         program_error("eventB");
   }
}
```

   Dabei wird nach Aufruf der Funktion, im Beispiel `eventB` für das Ereignis `B`, der aktuelle Zustand ermittelt, im Beispiel durch die switch-Anweisung. Anschließend wird die Aktion ausgeführt (Ausgabe durch `printf`) und der neue Zustand eingenommen, im Beispiel durch die Zuweisung des neuen Zustands an die Variable `State`.

2. Für jede Aktion oder Ausgabe wird eine Funktion implementiert:

[13] Die Beispiele wurden aus [Mar95] übernommen, alle Beispiele sind in der Programmiersprache *C* notiert.

```
void Cin1(event, state)
   enum eventT event;
   enum stateT state;
{
   printf("C in 1");
}


typedef struct
{
   enum stateT next_state;     /* The next state */
   void (*action)();           /* Action during transition */
} transitionT;


transitionT State_table[NUM_EVENTS][NUM_STATES]=
{
/*                 state1            state2    */
/* event A */     {{state1,Ain1},   {state2,Ain2}}
/* event B */     {{state2,Bin1},   {state1,Bin2}}
/* event C */     {{state2,Cin1},   {state2,Cin2}}
};


void handle_event(event)
   enum eventT event;
{
   static enum stateT state = state1;                      /* Initial state */
   (*(State_table[event][state].action))(state,event);     /* Process the event */
   state = State_table[event][state].next_state;           /* Update the state */
}
```

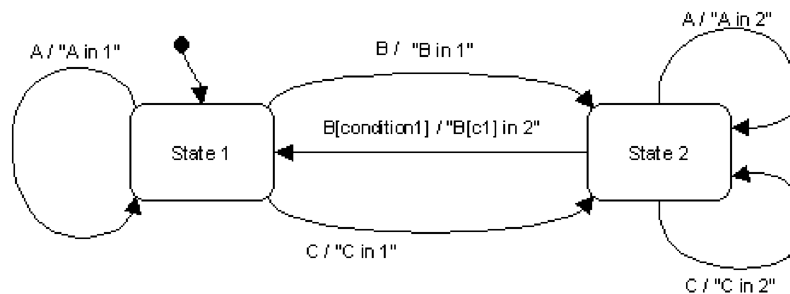
Als Beispiel wurde hier die Aktion mit der Ausgabe `''C in 1''` implementiert durch die Funktion `Cin1`. Diese tut nichts anderes, als die entsprechende Ausgabe zu tätigen[14]. Der Übergang in den neuen Zustand und der Aufruf der Funktion für die Aktion geschieht in der Funktion `handle_event`. Dort wird anhand der `State_table` und der darin gespeicherten Menge von Transitionen des definierten Typen `transitionT` zuerst die Funktion für die Aktion aufgerufen durch `(*(State_table[event][state].action))(state,event);`, anschließend der Variablen für den Zustand, `state`, der neue Zustand zugewiesen (`state = State_table[event][state].next_state;`).

3. Für jeden Zustand wird eine Funktion implementiert:

[14] Marick schreibt dazu in [Mar95], daß das Übergeben der Parameter `event` und `state` zum Beispiel für die Fehlersuche nötig ist:
"Its's conventional to pass the state and event to the transition function. If nothing else, it helps with debugging."

```
void state2()
{
   enum eventT event = get_event();
   switch(event)
   {
      case eventA:
         printf("A in 2");
         state2();
         break;
      case eventB:
         printf("B in 2");
         state1();
         break;
      case eventC:
         printf("C in 2");
         state2();
         break;
      default:
         program_error("state1");
   }
}
```

Hier wird es exemplarisch für den Zustand `State 2` gezeigt. Dabei wird das Ereignis durch eine switch-Anweisung ermittelt, anschließend die Aktion ausgeführt und die Funktion für den nachfolgenden Zustand aufgerufen.



Abbildung~2.18: Beispiel für einen Zustandsautomaten mit Bedingungen

Die Implementierung von Harels Statecharts zeigt Marick [Mar95] nur in Form der ersten vorgestellten Möglichkeit einer Implementierung und beschränkt sich dabei auf die Implementierung der Bedingungen. Das Beispiel aus Abbildung 2.17 wurde zur Vorstellung dieser Implementierungsmöglichkeit um eine Bedingung (`condition1`, dargestellt in Abbildung 2.18) erweitert.

Die Implementierung würde dann wie folgt aussehen:

```
switch(State)
{
   case state1:
      printf("B in 1");
      State = state2;
      break;
   case state2:
      if (condition1)
      {
         printf("B[c1] in 2");
         State = state2;
      }
      break;
   default:
      program_error("eventB");
  }
}
```

Der einzige Unterschied zur ursprünglichen Definition besteht in der Einführung einer if-Anweisung für alle Bedingungen. Diese müssen vor Ausführung der Aktion und des Zustandsübergangs zuerst überprüft werden (im Beispiel `condition1`).

2.5 Zustandsautomaten und Objektorientierung

2.5.1 Zustandsautomaten aus objektorientierter Sicht

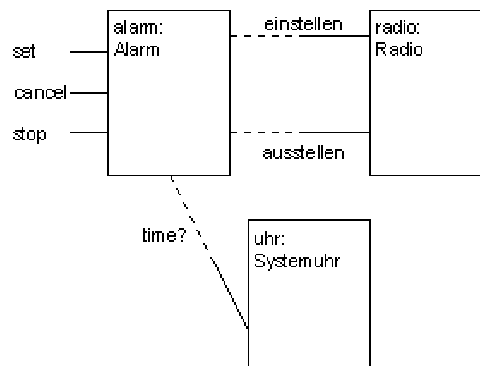
Zustandsautomaten eignen sich gut für die Modellierung von Objektlebenszyklen [Bal96, CHB92]. Dabei werden zur Modellierung meist die in Kapitel 2.4.3 vorgestellten Statecharts oder Harel-Automaten verwendet und in Hinblick auf ihre Verwendung in der objektorientierten Softwareentwicklung modifiziert und erweitert [RBP$^+$91, CHB92, HG95]. Da es in der Literatur weitgehend übereinstimmende Modelle gibt, soll hier auf einzelne Abwandlungen der Harel-Notation nicht in jedem einzelnen Fall eingegangen werden.

Eine der bekanntesten Formen von Zustandsautomaten findet sich in [RBP$^+$91] und wird von Rumbaugh et al. als dynamisches Modell bezeichnet. Dabei erweitert er die Statecharts um Endzustände, gekennzeichnet wie bei den Mealy- und Moore-Automaten durch die doppelte Umrandung der Zustände, Aktivitäten leitet er mit `do:` ein. Eine andere Abwandlung der Statecharts findet sich in [CHB92].

Um Objektlebenszyklen durch Zustandsautomaten zu modellieren, müssen die Zustände und Ereignisse für Objekte interpretiert werden. Im allgemeinen wird davon ausgegangen, daß Objekte interne Zustände haben, die durch Äquivalenzklassen von Verknüpfungen der Attribute des Objekts definiert werden [RBP$^+$91, MD93, Mar95]. Ereignisse sind Nachrichten, die an das Objekt gesandt und durch Methoden implementiert werden. Dabei gehen einige Autoren darüber hinaus und bezeichnen als Ereignisse nur die Aufrufe von nach außen sichtbaren Methoden, die das Objekt veranlassen, einen der Attributwerte zu ändern [MD93], andere unterscheiden

zwischen Methoden[15], die zustandsändernd sind (diese implementieren Ereignisse) und Methoden, die den Wert eines Attributes zurückgeben (Observer) [CHB92][16]. Auch der Einfluß nach außen nicht sichtbarer Attribute auf den Zustand ist umstritten. Einige beziehen diese in die Modellierung des Zustandsdiagramms mit ein [CHB92], andere definieren den Zustand einer Klasse als den nach außen sichtbaren Zustand [MD93].

Turner und Robson [TR95] weisen den Konstruktoren und Destruktoren einer Klasse eine Sonderstellung zu, da es sich um besondere Arten von Methoden handelt. Konstruktoren werden ausgeführt, wenn sich ein Objekt in einem undefinierten Zustand befindet, Destruktoren führen in einen undefinierten Zustand [TR95]. Konstruktoren führen das Objekt also in den Anfangszustand, Destruktoren aus dem Endzustand heraus. Turner und Robson repräsentieren in [TR95] den undefinierten Zustand durch das Symbol Θ.

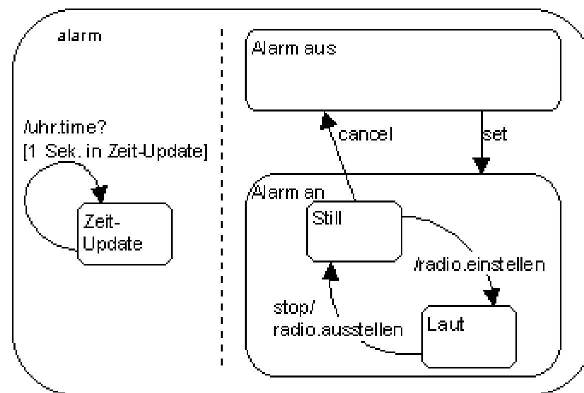


Abbildung~2.19: Beispiel für ein Konfigurationsdiagramm

Coleman, Hayes und Bear beschreiben in [CHB92], wie sich Zustandsautomaten auch auf das Verhalten eines ganzen Systems anwenden lassen (siehe auch [RBP+91]). Dazu erweitern sie die Statecharts von Harel durch die Darstellung von Kommunikation zwischen Objekten. Dabei nutzen sie das Konzept der Aktionen, um Nachrichten an andere Objekte zu definieren. Sie beschränken sich nicht wie andere Autoren (z.B. [Bal96]) darauf, für alle Objekte einer Klasse einen Zustandsautomaten zu definieren, sondern modellieren auch das Verhalten von Instanzen. Zusätzlich zum Zustandsdiagramm nutzen sie ein Konfigurationsdiagramm, um die Nachrichten zwischen Objekten darzustellen. Hier wird die Kommunikation zwischen zwei

[15] Methoden sind vom Objekt bereitgestellte Dienste und werden mit diesem Begriff synonym verwendet.

[16] Dabei ist es bei unsauberer Programmierung möglich, zustandsändernde Methoden zu schreiben, die gleichzeitig den Wert eines Attributes zurückliefern.



Abbildung~2.20: Beispiel für ein Zustandsdiagramm mit Kommunikation

Objekten durch eine Linie zwischen diesen Objekten dargestellt, die auf der Seite des Objekts, das einen Dienst anfordert, gestrichelt ist, auf der Seite des Objekts, das den Dienst bereitstellt, durchgezogen.

Das Beispiel in Abbildung 2.19 zeigt ein Konfigurationsdiagramm für eine Alarmuhr. Dabei werden die Instanzen der Klassen dargestellt. Es gibt die Instanzen `alarm` der Klasse `Alarm`, `radio` der Klasse `Radio` und `uhr` der Klasse `Systemuhr`. Die Klasse `Alarm` stellt die Dienste `set`, `cancel` und `stop` zur Verfügung, die Klasse `Systemuhr` den Dienst `time?` und die Klasse `Radio` die Dienste `einstellen` und `ausstellen`. Die Klasse `Systemuhr` nutzt die Dienste der anderen beiden Klassen.

Die Einbindung der Nutzung der Dienste anderer Objekte in ein Zustandsdiagramm zeigt das Zustandsdiagramm des Objekts `uhr` der Klasse `Systemuhr` in Abbildung 2.20. Der Aufruf der Dienste erfolgt als Aktion an den Transitionen. Um zu kennzeichnen, daß es sich um einen Aufruf eines Dienstes eines anderen Objekts handelt, wird der Objektname dem Aufruf vorangeschrieben.

Das Verhalten einer Klasse wird wie üblich (siehe oben) als Zustandsautomat beschrieben. Das Verhalten des gesamten Systems kann jetzt sehr einfach beschrieben werden durch das Verhalten aller Objekte innerhalb des Systems mit allen Interaktionen zwischen den Objekten.

2.5.2 Vererbung und Zustandsautomaten

Bisher wurde das Basiskonzept Vererbung der objektorientierten Softwareentwicklung bei der Modellierung von Zustandsautomaten außer Acht gelassen.

McGregor und Dyer beschreiben in [MD93] ein Verfahren, Zustandsautomaten zu vererben. Sie beschränken sich jedoch auf die sogenannte strenge Vererbung. Die Theorie von McGregor und Dyer kann dazu verwendet wer-

den, eine Unterklasse auf ihre Eigenschaften als Untertyp zu überprüfen und unterstützt auf diese Weise den Entwurf von Unterklassen, die Untertypen der Oberklasse sind.

Es finden sich auch Ansätze für die Modellierung von Vererbung in [San94] und [CHB92].

In [CHB92] wird die Untertypenbildung (sprich Vererbung) durch die Hinzufügung eines neuen Zustands oder einer neuen Transition oder der Verstärkung der Spezifikation einer Transition.

Eine etwas andere Auffassung vertreten McGregor und Dyer [MD93]. Sie definieren die strenge Vererbung wie folgt:

- Die Vorbedingungen einer Methode der Unterklasse können in Bezug auf die Vorbedingungen der entsprechenden Methode der Oberklasse nur abgeschwächt werden.

- Die Nachbedingungen einer Methode der Unterklasse können in Bezug auf die Nachbedingungen der entsprechenden Methode der Oberklasse nur verstärkt werden.

Dieses folgt aus der Tatsache, daß bei strenger Vererbung die Spezifikation der Unterklasse die Spezifikationen aller Oberklassen einschließen muß (siehe auch [Mey88]).

Daraus kann man implizieren, daß für strenge Vererbung folgende drei Aussagen gelten müssen [MD93]:

1. Eine Unterklasse darf keinen Zustand der Oberklasse löschen.

2. Jeder neue Zustand der Unterklasse muß vollständig in einem Zustand einer der Oberklassen enthalten sein.

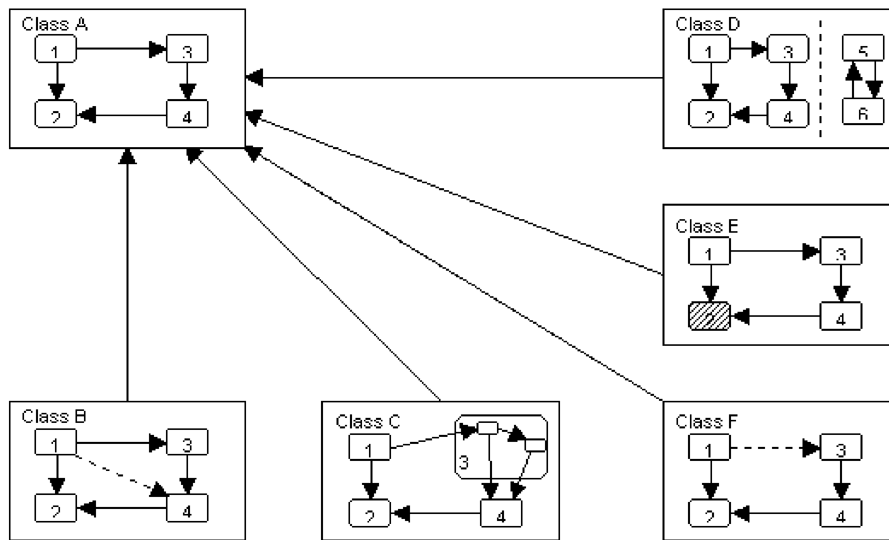
3. Eine Unterklasse darf keine Transition der Oberklasse löschen.

McGregor und Dyer [MD93] sind im Gegensatz zu Coleman, Hayes und Bear [CHB92] der Ansicht, daß keine neuen Zustände in den Zustandsautomaten der Unterklasse eingeführt werden dürfen, sofern die Zustände der Oberklasse nicht vollständig überdeckt werden.

Sane beruft sich in [San94] auf die Definition von Untertypen in [LW94]. Im Gegensatz zu McGregor/Dyer [MD93] geht er jedoch nicht auf die daraus folgende Vererbung von Zustandsautomaten ein.

Abbildung 2.21[17] zeigt die in [MD93] vorgestellten Möglichkeiten, strenge Vererbung von Zustandsautomaten zu realisieren. Klasse A ist die Oberklasse der anderen Klassen B, C, D, E und F. Für jede Klasse ist in der

[17] Die Abbildung wurde aus [MD93] übernommen, aus Platzgründen ein wenig modifiziert.



Abbildung~2.21: Strenge Vererbung von Klassen

Abbildung der dazugehörende Zustandsautomat in einem Kasten mit der Beschriftung `Class` und dem entsprechenden Klassennamen angegeben.

Klasse B fügt dem Zustandsautomaten der Klasse A eine Transition hinzu. Klasse C teilt den Zustand 3 der Klasse A in zwei neue Zustände auf, die den Zustand 3 der Oberklasse vollständig überdecken. In Klasse D wurde dem Zustandsautomaten der Oberklasse ein unabhängiges Zustandsdiagramm hinzugefügt. In Klasse E wurde ein Zustand des Zustandautomaten der Oberklasse A modifiziert, in Klasse F eine Transition, wobei sich das Zustandsdigramm dadurch nicht signifikant ändern darf.

Alle vorgenommenen Änderungen des Zustandsautomaten der Klasse A, um die Zustandsautomaten der Unterklasse B bis F zu erhalten, erfüllen die zuvor beschriebenen Bedingungen für strenge Vererbung.

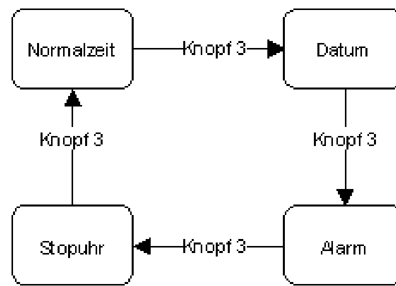
Für Mehrfachvererbung beschreiben McGregor und Dyer ein Verfahren, das die Zustandsautomaten aller Oberklassen durch unabhängige Zustandsdiagramme im Zustandsautomaten der Unterklasse repräsentiert. Dabei ist Voraussetzung, daß die Oberklassen unabhängig voneinander sind [MD93].

Um die Vererbungsformen im einzelnen zu beschreiben, wird jede der oben angegebenen Möglichkeiten in Abbildung 2.21 durch ein Beispiel verdeutlicht.

Als Oberklassen dienen zwei unterschiedliche Klassen.

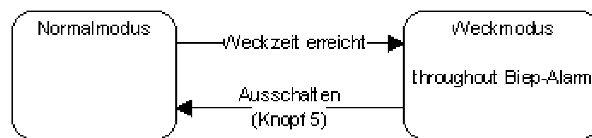
Abbildung 2.22 zeigt die Modellierung einer Uhr als Oberklasse (Klasse A)[18], von der Unterklassen durch Einfügen einer Transition (Klasse B),

[18]Die Klassenbezeichnung in Klammern entspricht an dieser wie an den weiteren Stellen



Abbildung~2.22: Oberklasse 1

Teilen eines Zustandes (Klasse C) und Einführung eines nebenläufigen Diagramms (Klasse D) gebildet werden. Die Uhr besitzt vier Funktionen, das Anzeigen der Normalzeit und des Datums, einen Alarm und eine Stopuhr. Ein Knopf (`Knopf 3`) schaltet zwischen den einzelnen Modi um. Jeder Modus wird durch einen Zustand repräsentiert, das Drücken des Knopfes stellt ein Ereignis dar.



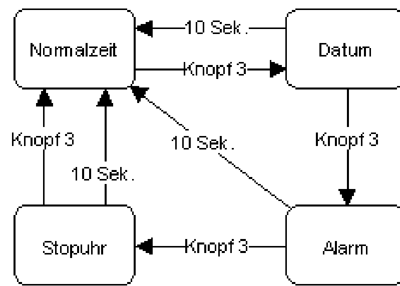
Abbildung~2.23: Oberklasse 2

Abbildung 2.23 zeigt die Modellierung einer Uhr als Oberklasse (Klasse A), von der Unterklassen durch Modifizierung eines Zustandes (Klasse E) oder einer Transition (Klasse F) gebildet werden. In diesem Beispiel werden Normal- und Weckmodus eines Weckers modelliert, der bei Erreichen der Weckzeit einen Alarm auslöst, in diesem Fall durch ein periodisches Alarmsignal (`Biep-Alarm`). Ausgeschaltet wird der Alarm durch einen Knopf (`Knopf 5`).

Zuerst sollen die aus der Oberklasse in Abbildung 2.22 gebildeten Unterklassen betrachtet werden.

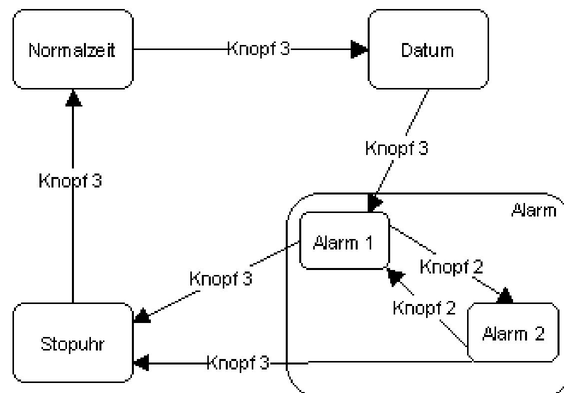
Die erste Möglichkeit ist die Bildung einer Unterklasse durch Einfügung einer neuen Transition (Klasse B). Die Uhr aus Abbildung 2.24 besitzt im Gegensatz zu der Uhr in Abbildung 2.22 die Eigenschaft, von allen anderen Modi nach 10 Sekunden in den Modus für die Anzeige der Normalzeit

der Klassenbezeichnung aus Abbildung 2.21. Später werden auch die Beispiele zur besseren Orientierung entsprechend benannt.



Abbildung~2.24: Unterklasse (neue Transition)

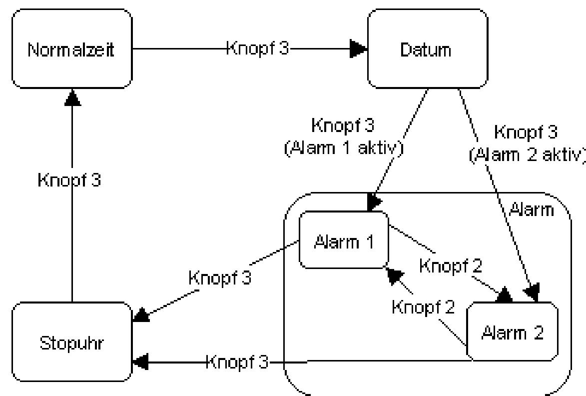
zurückzuspringen. Diese zusätzliche Eigenschaft wird durch ein Zustandsdiagramm modelliert, daß zusätzlich zum Diagramm der Oberklasse neue Transitionen einführt, in Abbildung 2.24 durch die Beschriftung `10 Sek.` zu erkennen.



Abbildung~2.25: Unterklasse (Zustand teilen)

Es ist weiterhin möglich, einen Zustand zu teilen (Klasse C). Abbildung 2.25 zeigt das Zustandsdiagramm einer Unterklasse der modellierten Uhr aus Abbildung 2.22, bei der der Zustand `Alarm` in die Zustände `Alarm 1` und `Alarm 2` unterteilt ist. Für die Uhr bedeutet das, daß sie zwei verschiedene Alarme gibt, zwischen deren Anzeige sich im Alarmmodus mit einem Knopf (`Knopf 2`) hin und her schalten läßt.

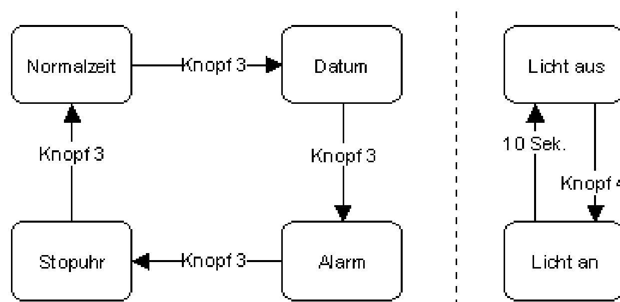
Dabei ist zu beachten, daß die beiden neuen Zustände, die aus dem Zustand `Alarm` hervorgegangen sind, disjunkt sind und diesen vollständig überdecken. Alle in den Zustand führenden Transitionen müssen auf die neuen Zustände aufgeteilt werden; in unserem Beispiel in Abbildung 2.25 führt nur eine Transition hinein, die in den Zustand `Alarm 1` führt. Es wäre



Abbildung~2.26: Unterklasse (Zustand teilen)

auch möglich, hineinführende Transitionen durch Bedingungen auf die neuen Zustände aufzuteilen, wie es in Abbildung 2.26 beispielhaft vorgenommen wurde. Hier bestimmt der jeweils aktive Alarm als Bedingung den Eintritt in einen bestimmten Zustand. Ist `Alarm 1` aktiv, wird der Zustand `Alarm 1` angenommen, anderenfalls `Alarm 2`. Dabei müssen alle Bedingungen disjunkt sein und vollständig die ursprüngliche Bedingung[19] überdecken.

Hinausführende Transitionen müssen aus allen neuen Zuständen heraus in den Zustand führen, in den sie auch in der Oberklasse führen, in den Beispielen aus den Abbildungen 2.25 und 2.26 in den Zustand `Stopuhr`.



Abbildung~2.27: Unterklasse (Nebenläufiges Zustandsdiagramm)

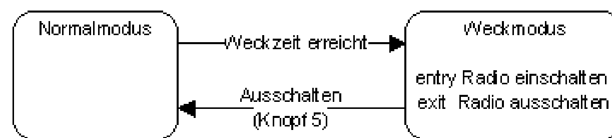
Auch die Einführung eines nebenläufigen Zustandsdiagramms stellt eine gültige Form der strengen Vererbung dar (Klasse D). Die Einführung neuer Zustände ist nicht erlaubt, aber wenn man die Definition nebenläufiger Zu-

[19] Bei keiner angegeben Bedingung, wie in diesem Fall, ist die Bedingung immer wahr, also *true*.

standsdiagramme nach Harel aus Kapitel 2.4.3 betrachtet, so stellt man fest, daß die Zustände des nebenläufigen Zustandsdiagramms keine eigenständigen Zustände sind, sondern in einer UND-Beziehung zu den ursprünglichen Zuständen stehen, es handelt sich also um das Teilen von Zuständen und die Einführung neuer Transitionen zwischen diesen.

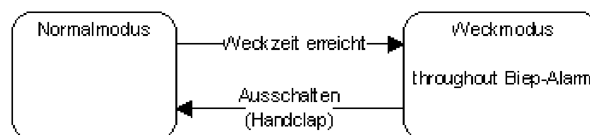
Im Beispiel aus Abbildung 2.27 wird der Zustandsautomat der Oberklasse um ein nebenläufiges Zustandsdiagramm mit den Zuständen `Licht an` und `Licht aus` erweitert. Es handelt sich bei der Oberklasse um eine Uhr ohne Licht, bei der Unterklasse um eine mit Licht. Das Licht wird an und ausgeschaltet, wie es schon im Beispiel in Abbildung 2.12 beschrieben ist.

Die nachfolgenden Zustandsautomaten modellieren das Verhalten von Klassen, die von der zweiten Oberklasse aus Abbildung 2.23, einem Wecker, abgeleitet wurden. Die vorgenommenen Modifizierungen des Zustandes beziehungsweise der Transition dürfen das Zustandsdiagramm nicht signifikant ändern.



Abbildung~2.28: Unterklasse (Modifizierter Zustand)

Die Modifizierung des Zustandes (Klasse E), wie sie in Abbildung 2.28 vorgenommen wurde, betrifft in diesem Fall die Aktion, die an den Weckmodus geknüpft ist. War es in der Oberklasse ein Alarm, der ertönte, so hat der Wecker, der die Unterklasse repräsentiert, die Funktion, ein Radio einzuschalten. Wird der Weckmodus beendet, so wird das Radio wieder abgeschaltet. Da beide Aktionen sich dazu eignen, einen Menschen zu wecken, ändert sich das Diagramm nicht signifikant. Eine weitere nicht signifikante Zustandsänderung wäre zum Beispiel die Änderung des Namens eines Zustands.



Abbildung~2.29: Unterklasse (Modifizierte Transition)

Abbildung 2.29 zeigt einen Wecker, der als Unterklasse des ursprü nglichen Weckers modelliert wurde, in dem er eine Transition nicht signifikant modifiziert. Das geschieht zum Beispiel durch das Überschreiben einer Methode der Oberklasse [MD93]. Hier wurde das Ausschalten redefiniert, indem es nicht durch das Drücken eines Knopfes geschieht, sondern durch ein Klatschen mit den Händen (`Handclap`).

Nur wenn man die Arbeit von McGregor und Dyer [MD93] berücksichtigt, lassen sich Objektlebenszyklen effizient durch Zustandsautomaten modellieren. Strenge Vererbung unterstützt die saubere Definition von Unterklassen. Zustandsdiagramme von Unterklassen müssen nicht neu modelliert, sondern können in Abhängigkeit zur Oberklasse gebildet werden. Damit entspricht dieser Vorgang in der Definitionsphase des objektorientierten Softwareentwicklungsprozesses vielmehr dem Prozeß der Unterklassenbildung in der Implementationsphase. Unterklassen entstehen aus Oberklassen durch Hinzufügen neuer Attribute und der Neu- oder Redefinition von Methoden. Ein ererbter Zustandsautomat unterstützt dieses Vorgehen, in dem er schon in der Definitionsphase die Unterschiede zwischen Ober- und Unterklasse verdeutlicht.

2.6 Nutzung von Zustandsautomaten für den Test

Das Testen von Software anhand von Zustandsautomaten zählt zu den Black-Box-Testverfahren, da es sich bei der Modellierung von Zustandsautomaten um eine Spezifizierung der Kontrollstruktur (Zustände und Transitionen) und der Aktionen und Aktivitäten der Software handelt.

Dieses Kapitel erläutert im Abschnitt 2.6.1 zuerst die Überprüfbarkeit der Spezifikation. Dies ist nicht ein Thema für Testen im engeren Sinne, ist jedoch für einen sinnvollen Test unabdingbar, da eine mangelhafte Spezifikation, zum Beispiel durch unerreichbare Zustände und Kontradiktion, das Testverfahren erschweren bzw. unmöglich machen würde.

Abschnitt 2.6.2 beschreibt die Testdatenbildung anhand von Zustandsautomaten, die Abschnitte 2.6.3 und 2.6.4 die Gewinnung und Auswertung der Testdaten.

2.6.1 Überprüfung der Spezifikation

Marick [Mar95] stellt die verschiedenen beim Modellieren von Zustandsautomaten auftretenden Fehler dar und diskutiert die Möglichkeiten der automatisierten Überprüfung von Zustandsautomaten. Eine korrekte Spezifikation ist eine unverzichtbare Basis für einen sinnvollen Test. Treten beispielsweise Differenzen zwischen Spezifikation und Implementierung auf, so kann die Ursache bei einem von beiden oder gar beiden zu finden sein. Eine über-

prüfte Spezifikation vermindert das Risiko eines Fehler in derselben, so daß man sich bei der Fehlersuche auf die Implementation konzentrieren kann.

Marick [Mar95] diskutiert einzelne Punkte, die hier in gleicher Reihenfolge wiedergegeben und erläutert werden sollen:

- Vollständigkeit

  Zur Vollständigkeit gehört, daß das Modell das Verhalten des Systems lückenlos beschreibt. Insbesondere sollte die Definition von Fehlerzuständen einer Prüfung unterzogen werden [Mar95]. Ohne vorliegende Spezifikation in anderer Form ist eine automatisierte Überprüfung fehlender Spezifikationsteile nicht zu realisieren und fällt somit in den Bereich des Entwicklers des Modells. Jedoch wäre es denkbar, anhand einer bestehenden Spezifikation, z.B. in einer Spezifikationssprache wie Z, die Modellierung des Zustandsautomaten zu inspizieren.

- Zweideutigkeit

  Unkorrekte Abstraktion kann wichtige Details verbergen. So kann zum Beispiel ein definierter Zustand in Wirklichkeit aus zwei Zuständen bestehen oder eine Transition müßte noch durch Bedingungen in zwei Transitionen geteilt werden. Im Modell sollten alle Transitionen anhand von realen Eingaben geprüft werden [Mar95]. Für die Automatisierbarkeit der Prüfung gilt das in diesem Zusammenhang schon für die Vollständigkeit Gesagte.

- Kontradiktion

  Zu Widersprüchen kann es zum Beispiel kommen, wenn ein Ereignis gleichzeitig in zwei Zustände führen soll [Mar95]. Dieses läßt sich durch Kontrolle aller Nachfolgezustände durch Aufruf aller Transitionen von einem Zustand aus automatisieren.

- Unerreichbare Zustände

  Zustände, die nicht von einem der Anfangszustände aus durch eine Folge von Transitionen eingenommen werden können, sind unerreichbar. Marick stellt in [Mar95] einen Algorithmus zur Aufdeckung unerreichbarer Zustände vor. Alle erreichbaren Zustände von den Startzuständen aus werden in einer Liste vermerkt und von diesen aus weitere Nachfolgezustände gesucht. Sind alle Transitionen ausgeführt, die in den erreichbaren Zuständen beginnen, so sind alle Zustände, die nicht in der Liste stehen, nicht erreichbar.

  Hier wie auch bei der Kontradiktion verursachen Bedingungen Probleme bei der Aufdeckung der Fehler im Modell. So ist nicht immer offensichtlich, daß zwei Bedingungen sich wiedersprechen oder gleich sind.

- Tote Zustände

  Unter toten Zuständen versteht man Zustände, die nicht wieder verlassen werden können. Tote Zustände können durch einen abgewandelten Algorithmus für unerreichbare Zustände gefunden werden. Der Unterschied besteht darin, von jedem Zustand aus einen Nachfolgezustand zu finden statt einen Vorgänger, also eine Transition, die hinausgeht statt hinein [Mar95].

  Unerreichbare und tote Zustände haben gemeinsam, daß sie in manchen Fällen sinnvoll sind (tote Zustände könnten Endzustände sein), oft jedoch auf eine fehlerhafte Modellierung hinweisen [Mar95].

2.6.2 Testdatenbildung

Zustandsautomaten können zur Bildung von Testdaten verwendet werden. Dabei kann der Zustandsautomat dazu verwendet werden, Sequenzen von Methoden zu bilden, die eine möglichst große Überdeckung von Zuständen, Transitionen und Diagramm (siehe Abschnitt 2.6.4) erreichen.

Zustandsautomaten in der Objektorientierung werden so verstanden, daß die Zustände durch die Attribute des Objekts bestimmt werden und die Ereignisse aufgerufene Methoden des Objekts sind (siehe Kapitel 2.5.1). Dieser Sicht folgend, werden Testdaten durch Sequenzen von Methoden gebildet, die das Objekt jeweils in einen neuen Zustand führen. Fehler können durch die Überwachung (beschrieben in Abschnitt 2.6.3) des Objekts aufgedeckt werden.

2.6.3 Objektüberwachung

Während der Testläufe muß das Objekt überwacht werden, um Differenzen zwischen dem erwarteten Verhalten und dem tatsächlichen Verhalten festzustellen.

Zur Überwachung des Zustandes von Objekten können die Methoden verwendet werden, die nur den Wert der Attribute zurückliefern, ohne den Zustand des Objekts zu ändern, sogenannte Observer (siehe auch Kapitel 2.5.1).

Tests können einerseits nur Objekte einer Klasse in einer speziellen Testumgebung testen, so daß der Tester eine Sequenz von Methoden der Klasse aufruft und nach jeder Methode den aktuellen Zustand des Objekts erfragt, andererseits innerhalb eines Systems stattfinden.

Wird die Klasse innerhalb eines Systems getestet, so ist nach außen nicht ersichtlich, ob und welche Methode gerade aufgerufen wurde. Deshalb ist es nötig, den Quellcode der Klasse so zu instrumentieren, daß der Aufruf einer Methode und die Zustandsänderungen protokolliert werden können. Bei der Instrumentierung wird eine Kopie der zu testenden Klasse angelegt,

in der manuell oder automatisch Programmcode hinzugefügt wird, der eine Überwachung ermöglicht, zum Beispiel Aufrufe eines Zustandsautomaten. Dies ist nicht ganz unproblematisch, da der neu hinzugefügte Programmcode Fehler enthalten kann.

Geht man davon aus, daß der Zustand eines Objekts der nach außen sichtbare Zustand des Objekts ist, muß der Zustand wieder über nach außen sichtbare Variablen und die Observer erfragt werden.

Chow [Cho78] und Marick [Mar95] beschreiben mögliche Fehler, die durch das Testen anhand von Zustandsautomaten entdeckt werden können.

Marick [Mar95] zählt folgende Fehler auf, die entdeckt werden können:

- Eine falsch ausgewählte Transition, z.B. versursacht durch eine falsch implementierte Bedingung, z.B. eine Transition, die in einem Zustand nicht aufgerufen werden darf
- Eine falsch ausgeführte Transition, die z.B. in den falschen Zustand führt oder eine falsche Aktion auslöst
- Eine Transition, die nicht vorhanden ist, z.B wenn ein Ereignis in einem Zustand im Gegensatz zur Spezifikation keinen Zustandsübergang auslöst

Auch Fehler, die nicht aufgedeckt werden können, beschreibt Marick [Mar95]:

- Eine Aktion, die unabhängig vom Modell fehlerhaft ist, so z.B. eine Aktion, die zwar ausgeführt wird, jedoch ein falsches Ergebnis liefert
- Transitionen, die nicht zusammenarbeiten und so in einen toten Zustand führen können

Chow definiert in [Cho78] die Fehlerarten:

- Operationsfehler, also Transitionen, die falsche Aktionen auslösen
- Transferfehler, daß heißt Transitionen, die in den falschen Zustand führen
- Zusätzliche Zustände, die eigentlich gelöscht werden müßten
- Fehlende Zustände, das Gegenteil der zusätzlichen Zustände, die die Einführung neuer Zustände verlangen

2.6.4 Überdeckungsmessung

Analog zu den Überdeckungsmaßen der traditionellen kontrollflußorientierten Testverfahren, beschrieben in Kapitel 2.2, kann auch ein Zustandsdiagramm als Graph verstanden und eine Überdeckungsmessung durchgeführt werden.

Die Überdeckung aller Knoten würde bezogen auf das Zustandsdiagramm eine Überdeckung der Zustände bedeuten, die Überdeckung der Zweige eine Überdeckung der Transitionen.

Während für Marick [Mar95] eine Überdeckung aller Zweige, die wie in Kapitel 2.2 geschildert eine Überdeckung der Knoten einschließt, ausreichend ist, gehen andere Autoren, z.B. Chow in [Cho78] oder Hoffmann und Strooper in [HS94], darüber hinaus. Da eine vollständige Pfadüberdeckung auch bei Zustandsdiagrammen nur in den seltensten Fällen möglich ist, da sie meist zyklisch sind, definiert Chow [Cho78] neben der Zweigüberdeckung weitere Überdeckungsmaße und die mit den Überdeckungsmaßen aufdeckbaren Fehler; Hoffmann und Strooper [HS94] haben zum Ziel, neben der Zweigüberdeckung eine möglichst große Pfadüberdeckung zu erreichen.

2.7 Testbarkeit

Unter Testbarkeit versteht man, wie einfach oder schwierig der Test von Software sich gestaltet.

Binder unterscheidet in [Bin94] noch zwischen Kontrollierbarkeit und Beobachtbarkeit als zwei Teilbereiche der Testbarkeit. Um Software zu testen, ist es einerseits nötig, Eingaben und interne Zustände kontrollieren zu können, andererseits müssen Ausgaben beobachtbar sein.

Objektorientierte Software birgt das Problem, daß sie sich durch Datenkapselung schwerer beobachten läßt als traditionelle Software. Um dieses Problem zu umgehen, kann für jedes Attribut, das für die Durchführung des Test nötig ist (bei zustandsbasierten Testverfahren die für den Zustand relevanten Attribute), ein Observer implementiert werden. So ist der Wert des Attributs nach außen hin sichtbar, ohne verändert werden zu können. Dazu muß natürlich ein Vertrauen in die Korrektheit der Implementation der Observer bestehen [Ber93].

Die Kontrolle der aufgerufenen Methoden entzieht sich zum Teil dem Tester, da Methoden einer Klasse sich oft untereinander aufrufen und der Tester nur Zugriff auf die nach außen sichtbaren Methoden hat. Um gegenseitige Aufrufe von Methoden beobachten zu können, muß die Klasse in entsprechender Weise instrumentiert werden, was jedoch die Gefahr neuer Fehler birgt.

Objektorientierte Software ist nur so gut testbar, wie der Entwickler der Software dem Tester Zugang zu derselben gewährt. Eine vollständig

eingekapselte Klasse, die ihren Zustand nicht nach außen sichtbar macht, ist nur schwer zu testen.

Kapitel 3

Systementwicklung

3.1 Einführung

Im Rahmen dieser Diplomarbeit wurde das System JavaABT[1] zur Unterstützung zustandsbasierter Testverfahren für JAVA-Klassen in der Programmiersprache JAVA entwickelt. JavaABT soll einen Betrag zur Lösung der Problematik des Testens objektorientierter Software leisten. Es erlaubt dem Benutzer, die einzelnen Klassen anhand von Spezifikationen in Form von Zustandsautomaten (Automaten) zu testen. Dazu muß er die Klassen seines Systems spezifizieren, indem er die Zustände seiner Klassen benennt und die Zustandsübergänge zwischen diesen definiert. JavaABT übersetzt diese Spezifikation in ausführbaren Java-Programmcode, der dann zusammen mit dem Testobjekt ausgeführt wird. So kann JavaABT das Verhalten eines Objekts einer Klasse zur Laufzeit in Bezug auf seine Spezifikation überwachen.

Im folgenden werden Konzeption (Abschnitt 3.2), Entwurf (Abschnitt 3.3), und Besonderheiten der Implementierung (Abschnitt 3.4) von JavaABT vorgestellt.

3.2 Konzeption

Das folgende Kapitel beschreibt die Konzeption von JavaABT. In Abschnitt 3.2.1 werden funktionelle Anforderungen angeführt, Abschnitt 3.2.2 stellt die Benutzerschnittstelle dar.

3.2.1 Funktionelle Anforderungen

Die in den folgenden Abschnitten 3.2.1.1 bis 3.2.1.7 genannten Anforderungen an das entwickelte System, Aufgabenangemessenheit, Selbstbeschreibungsfähigkeit, Steuerbarkeit, Erwartungskonformität, Fehlerrobustheit, In-

[1] Java Automata Based Testing Tool

dividualisierbarkeit und Erlernbarkeit, werden gemäß der ISO-Norm 9241 Teil 10 eingeordnet [(Hr93].

3.2.1.1 Aufgabenangemessenheit

JavaABT soll eine Unterstützung des zustandsbasierten Testens objektorientierter Klassen in JAVA leisten. Da Klassen jedoch meist Teil eines Projekts[2] sind, soll JavaABT auch ein projektorientiertes Arbeiten unterstützen. Alle spezifizierten Zustandsautomaten sollen im Rahmen eines Projekts gemeinsam verwaltet werden können. Es sollte möglich sein, durch Angabe einer Klasse alle von ihr abhängenden Klassen aus einem Testwerkzeug zur Ermittlung der Testreihenfolge (siehe [Bei98]) zu importieren.

Der Benutzer soll die Wahl zwischen verschiedenen Arten, Zustandsautomaten zu definieren, haben (siehe Kapitel 2.4). Auch bei der Vererbung von Zustandsautomaten sollen verschiedene Vererbungsformen zur Auswahl stehen (siehe Kapitel 2.5.2).

Der Dialog in JavaABT soll an die Sprache des Benutzer angepaßt sein, Englisch sollte dabei als Standardsprache gelten, da in der Regel Informatiker das Programm benutzen werden.

3.2.1.2 Selbstbeschreibungsfähigkeit

JavaABT soll die Fachsprache der Informatiker benutzen. Es soll abgestufte Hilfe von einer Kurzbeschreibung bis hin zu langen Erklärungen zur Benutzung des Programms bieten. Allgemeines Wissen über die Definition von Zustandsautomaten wird dabei vorausgesetzt.

Alle Teile des JavaABT-Systems sollen eindeutig benannt werden, so daß der Benutzer den Überblick nicht verlieren kann und immer weiß, wo er sich gerade befindet.

3.2.1.3 Steuerbarkeit

JavaABT soll mehrere Möglichkeiten zur Steuerung anbieten. Alle möglichen Funktionen sollten als Menüpunkte vorhanden sein. Zusätzlich sollten wichtige und oft gebrauchte Funktionen als Buttons neben dem zu bearbeitenden Material plaziert werden. Dialogschritte sollen rückgängig gemacht werden können. Außerdem sollte das Programm jederzeit beendet werden können, immer mit der Möglichkeit, bisherige Eingaben und Ergebnisse zu speichern.

[2] Projekte werden hier verstanden als Verwaltung für Klassen, die Teil eines Systems sind.

3.2.1.4 Erwartungskonformität

Alle Masken von JavaABT sollen gleichartig aufgebaut sein. Da JavaABT als in JAVA geschriebene Anwendung plattformunabhängig ist, sollte beim Start des Programms die aktuelle Plattform ermittelt werden, um bei Vorhandensein des entsprechenden *Look&Feel*[3] das Aussehen der Fenster an die entsprechende Plattform anzupassen.

JavaABT soll die Vererbung von Zustandsautomaten unterstützen. Dabei soll das Hauptgewicht auf konforme Vererbung nach [MD93] gelegt werden (siehe Kapitel 2.5.2).

3.2.1.5 Fehlerrobustheit

JavaABT soll Korrekturen der Eingaben des Benutzer jederzeit zulassen. Fehler, die bei der Übersetzung des definierten Zustandsautomaten in ausfü hrbaren Java-Code entstehen, sollen erkannt und die Stelle der fehlerhaften Eingabe angezeigt werden. Eingabefehler sollen verständlich erklärt werden. Neben einer kurzen Fehlerbeschreibung soll der Benutzer sich auch eine längere Erläuterung anzeigen lassen können.

Um Fehler bei der Definition von Zustandsautomaten möglichst zu vermeiden, sollte JavaABT Eingabehilfen bieten, so zum Beispiel die einfache Einfügung von Methoden und Variablen in die Definition durch Auswahl der zur Verfügung stehenden Methoden und Variablen.

3.2.1.6 Individualisierbarkeit

Das Aussehen der Fenster und die Lage der Menüs und Buttons von JavaABT sollten so flexibel wie möglich an die Bedürfnisse des Benutzer anpaßbar sein. Der Benutzer sollte auch das *Look&Feel* nach seinen Wünschen unabhängig von der aktuellen Plattform ändern können.

Eine Festlegung von Optionen, zum Beispiel zur Voreinstellung von Parametern zur Definition der Zustandsautomaten, sollte dem Benutzer möglich sein.

3.2.1.7 Erlernbarkeit

JavaABT sollte im wesentlichen selbsterklärend sein und den Benutzer mit einem verständlichen Dialog und Hilfestellung auf Wunsch führen. Anhand eines Beispiels sollte der Funktionsumfang erläutert werden, wenn es verlangt wird.

---

[3]Das *Look&Feel* ist eine Erweiterung von JAVA, die es erlaubt, das Aussehen eines in JAVA geschriebenen Programms an die Plattform anzupassen oder ein typisches eigenes Aussehen zu definieren. Das Aussehen bezieht sich auf die Gestaltung der Fensteroberfläche.

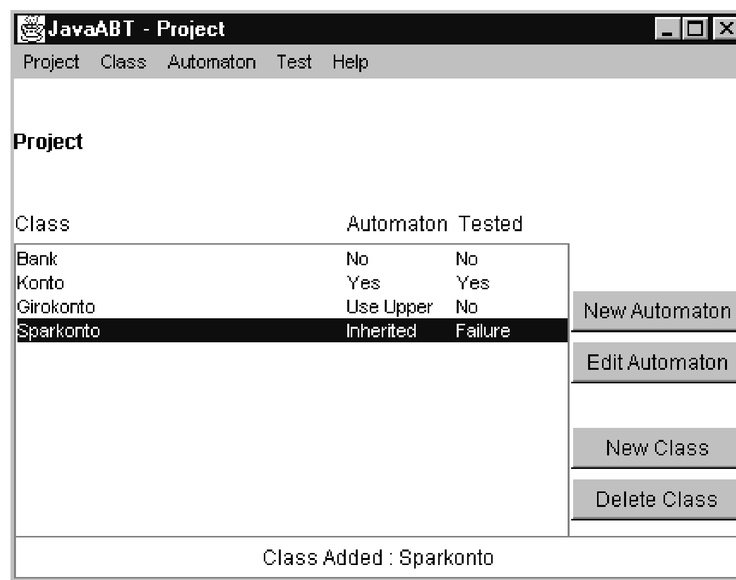
Die Definition der Zustandsautomaten sollte in JAVA-Code erfolgen, so daß es sich für den Anwender, der in der Regel JAVA beherrscht, einfach gestaltet, das System zu bedienen.

3.2.2 Benutzerschnittstellenbeschreibung

Die folgenden Abschnitte beschreiben die Benutzerschnittstelle von JavaABT. Abschnitt 3.2.2.1 beschäftigt sich mit der Projektverwaltung, Abschnitt 3.2.2.2 erklärt die Definition der Zustandsautomaten. In Abschnitt 3.2.2.3 folgt eine Erläuterung der Java-Code-Generierung, schließlich beschreibt Abschnitt 3.2.2.4 die Ausgaben des Automatenobjekts während des Test und Abschnitt 3.2.2.5 die Auswertung der Testdaten.

Zur Illustration werden Screenshots des Programms benutzt, da diese eine Umsetzung der zuvor gemachten Konzeption darstellen.

3.2.2.1 Projektverwaltung



Abbildung~3.1: Projektfenster

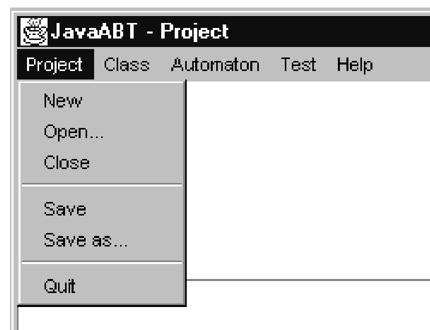
Nach dem Start von JavaABT erscheint ein Fenster (das Projektfenster, Abbildung 3.1), das zu Beginn nur die Möglichkeit bietet, ein Projekt zu laden oder ein neues zu erstellen. Ein geladenes oder erstelltes Projekt wird mit allen Klassen und den Informationen über die Automaten und den Teststatus im Fenster angezeigt. Es stehen die Buttons `New Automaton` und `Edit Automaton` für die Manipulation des Automaten und die Buttons `New Class` und `Delete Class` für das Einfügen und Löschen von Klassen

zur Verfügung. Dieselben Funktionen finden sich auch unter den Menüs `Automaton` und `Class`.

Klassen können im Projektfenster markiert werden, dann wird die jeweilige Funktion auf die markierte Klasse angewendet (im Beispiel in Abbildung 3.1 Klasse `Sparkonto`).

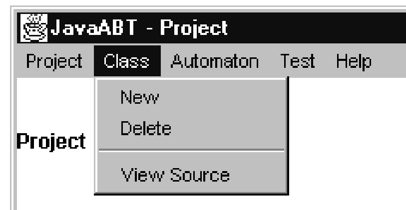
Ist bereits ein Automat definiert, wird dieses mit einem `Yes` in der Spalte `Automaton` angezeigt (z.B. Klasse `Konto`, Abbildung 3.1), Klassen ohne Automaten werden mit in dieser Spalte mit `No` identifiziert (im Beispiel in Abbildung 3.1 Klasse `Bank`). Analoges gilt für getestete und ungetestete Klassen in der Spalte `Tested`. Klassen, die den selben Automaten benutzen wie ihre Oberklasse, werden in der Spalte `Automaton` mit `Use Upper` gekennzeichnet (hier Klasse `Girokonto`, Abbildung 3.1), wird der Oberklassenautomat vererbt, so wird dieses mit `Inherited` gekennzeichnet (im Beispiel Klasse `Sparkonto`, Abbildung 3.1). Klassen, bei deren Test Fehler aufgetreten sind, werden in der Spalte `Tested` mit `Failure` gekennzeichnet (im Beispiel Klasse `Sparkonto`, Abbildung 3.1).

Außer den Buttons, die direkt im Fenster anwählbar sind, enthält das Projektfenster ein Menü, das die gesamte Funktionalität des Projektfensters enthält. Dazu eine kurze Erläuterung der Menüpunkte im einzelnen.

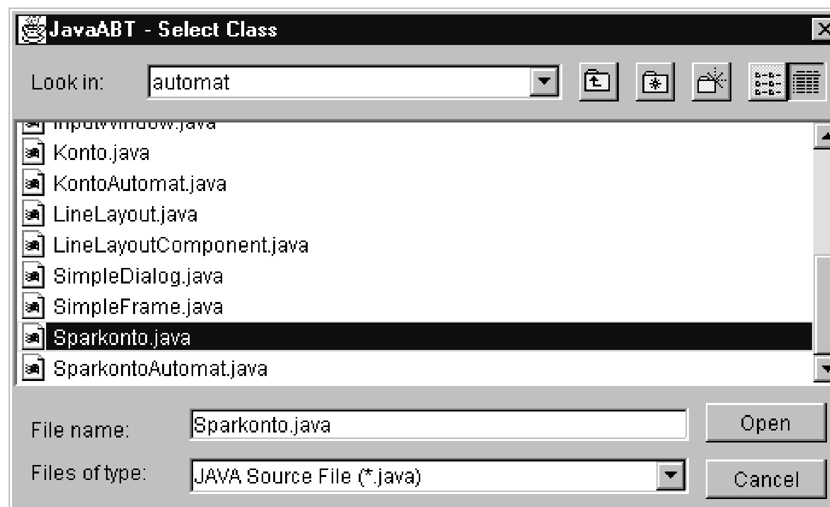


Abbildung~3.2: Projektmenü

Im Menü Projekt (Abbildung 3.2) werden neue Projekte angelegt (Menüpunkt `New`) und vorhandene Projekte geöffnet (Menüpunkt `Open...`). Neben diesen beiden Punkte ist nur `Quit` zum Verlassen des Programms nach dem Starten von JavaABT anwählbar. Mit `Close` läßt sich ein Projekt schließen, mit `Save` und `Save as...` speichern. `New` erzeugt ein neues leeres Projekt. Wird der Menüpunkt `Open` gewählt, so wird der Standarddialog, den JAVA zur Verfügung stellt, aufgerufen. Nur Projekte mit der Dateinamenerweiterung `'.jabt'` können geöffnet werden.



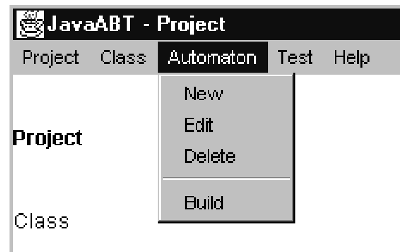
Abbildung~3.3: Menü `Class`



Abbildung~3.4: Standarddialog zum Öffnen und Speichern

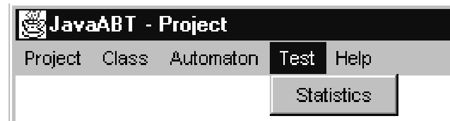
Das Menü Class (Abbildung 3.3) enthält die Menüpunkte `New` zum Hinzufügen einer Klasse, `Delete` zum Löschen einer Klasse aus dem Projekt und `View Source` zum Anzeigen des Sourcecodes der Klasse. Das Hinzufügen einer Klasse geschieht durch den Aufruf des Standarddialogs von JAVA zum Öffnen von Dateien (Abbildung 3.4), wobei nur JAVA-Source-Dateien ausgewählt werden können. Dies ermöglicht die Anzeige des Sourcecodes durch `View Source` und die automatische Instrumentierung der Klasse (siehe Kapitel 3.3.5). Die Menüpunkte `New` und `Delete` entsprechen den Buttons `New Class` und `Delete Class` im Projektfenster (Abbildung 3.1).

Das in Abbildung 3.5 dargestellte Menü `Automaton` ermöglicht die Definition eines Automaten durch die Menüpunkte `New` und `Edit`. Dieselbe Funktionalität besitzen die Buttons `New Automaton` und `Edit Automaton` im Projektfenster (Abbildung 3.1). Der Menüpunkt `New` läßt sich nur anwählen, wenn die selektierte Klasse noch keinen Automaten besitzt, `Edit` läßt sich aufrufen, wenn für die selektierte Klasse bereits ein Automat defi-



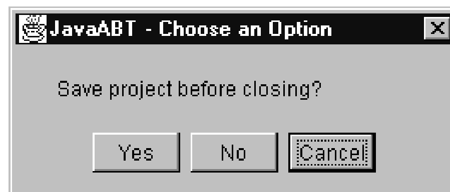
Abbildung~3.5: Menü Automaton

niert wurde. `Delete` löscht den Automaten nach Rückfrage beim Benutzer. `Build` generiert den Automaten (siehe Abschnitt 3.2.2.3).



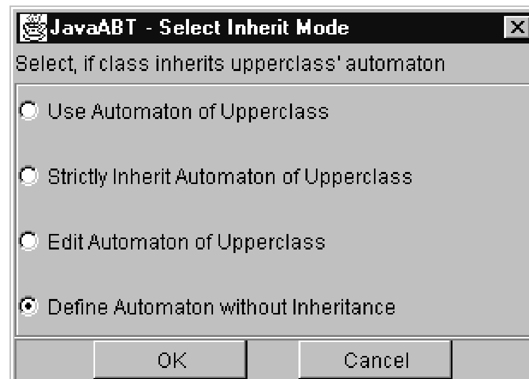
Abbildung~3.6: Testmenü

Das Menü `Test` (Abbildung 3.6) enthält den Menüpunkt `Statistics`. Durch Wahl dieses Menüpunktes wird die Teststatistik der selektierten Klasse, daß heißt die Werte der Überdeckungsmessung angezeigt (siehe Abschnitt 3.2.2.5).



Abbildung~3.7: Sicherungsdialog

Alle Aktionen, die eine nicht wiederherzustellende Veränderung des Projekts oder der Automatendefinition bewirken, wie zum Beispiel das Schließen eines ungesicherten Projekts oder das Löschen eines definierten Zustandes, werden in JavaABT durch Rückfragen abgesichert. Zum Beispiel wird beim Schließen eines ungesicherten Projekts nachgefragt, ob das Projekt gespeichert werden soll (Abbildung 3.7).



Abbildung~3.8: Dialog zur Auswahl der Vererbungsform

Soll ein neuer Automat durch New Automaton definiert werden, so wird vor dem Anzeigen des Definitionsfensters ein Dialog zur Auswahl der Definitionsform und, wenn in JavaABT eine Oberklasse der aktuellen Klasse vermerkt ist, ein Dialog zur Auswahl der Vererbungsform angezeigt. Abbildung 3.8 zeigt den Dialog zur Auswahl der Vererbungsform, der Dialog zur Auswahl der Definitionsform besitzt das gleiche Aussehen.

Als Definitionsformen stehen zwei Arten zur Verfügung. Die erste Art erlaubt die explizite Definition aller Transitionen, die in einem Zustand nicht erlaubt sind, alle nicht definierten Transitionen verursachen keine Zustandsänderung. Die zweite Definitionsart läßt den Benutzer alle Transitionen definieren, die keine Zustandsänderung bewirken, die Ausführung aller nicht definierten Transitionen sind dann verboten. Dies untersützt die Implementierung eines effektiven Überprüfungsalgorithmus, der durch die Meldung aller verbotenen Ereignisse in einem Zustand die Einhaltung der Vorbedingungen von Methoden überwacht.

Als Vererbungsformen stehen vier Arten zur Verfügung (siehe Abbildung 3.8), keine Nutzung des Oberklassenautomaten (`Define Automaton without Inheritance`), Benutzung des Automaten der Oberklasse ohne Editiermöglichkeit (`Use Automaton of Upperclass`), freie Editierung des Automaten der Oberklasse (`Edit Automaton of Upperclass`) und strenge Vererbung nach McGregor [MD93] (`Strictly Inherit Automaton of Upperclass`).

3.2.2.2 Definition des Automaten

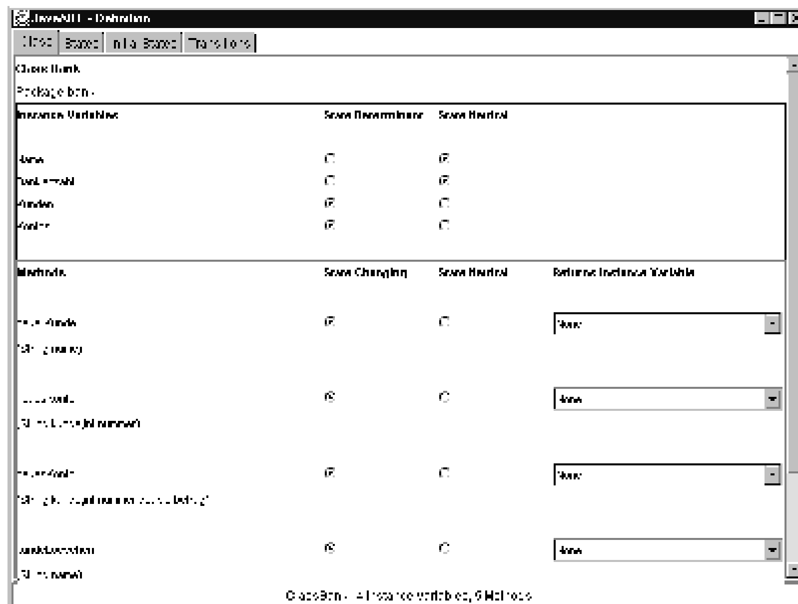
Bevor der Automat genutzt werden kann, muß er definiert werden. Soll eine Klasse getestet werden, so muß zuerst der Automat für diese Klasse spezifiziert sein. Eine Klasse, die von einer anderen erbt, kann den Automaten der Oberklasse verwenden oder erben. Dabei wirken sich Änderungen im

Automaten der Oberklasse auf die Unterklasse aus.

Die Definition eines Automaten umfaßt drei Teilbereiche:

1. die Hervorhebung einzelner Instanzvariablen als zustandsbestimmend und die Hervorhebung einzelner Methoden als zustandsändernd
2. die Definition der Zustände
3. die Definition der Zustandsübergänge (Transitionen)

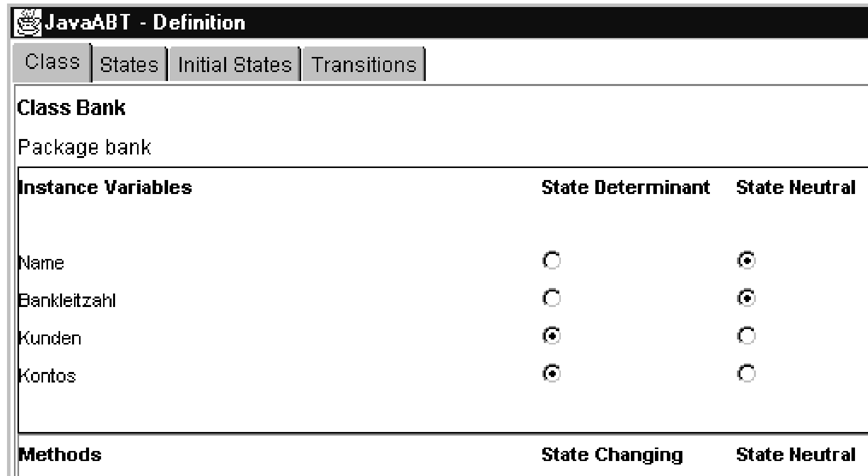
Es sollten schon während der Definitionsphase möglichst Fehler, die zu Übersetzungsfehlern des Compilers führen könnten, vermieden werden. Dazu bietet die Benutzeroberfläche verschiedene Definitionsmechanismen an, die dem Benutzer die Eingabe erleichtern und ihm gleichzeitig helfen, Fehler zu vermeiden.



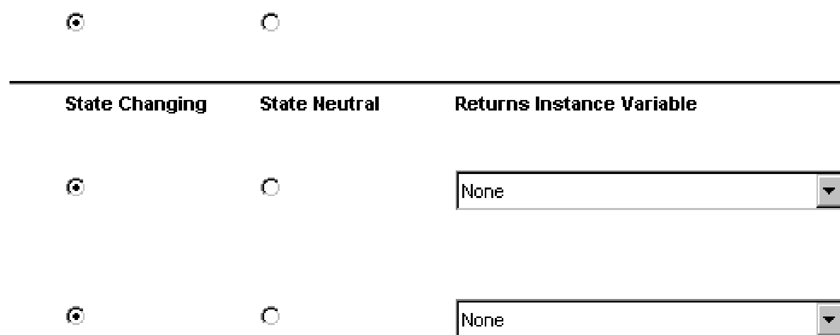
Abbildung~3.9: Definitionfenster

Das Definitionsfenster (Abbildung 3.9[4]) besitzt vier Ansichten, die durch die Tabs `Class`, `States`, `Initial States` und `Transitions` anwählbar sind. Die Statusleiste unten ist in allen Ansichten dieselbe und zeigt die letzte getätigte Aktion an, so daß der Benutzer in allen Ansichten einen Überblick über sein Vorgehen besitzt.

[4]Abbildung 3.9 zeigt das Definitionsfenster in voller Größe. Die weiteren Abbildungen zeigen aus Gründen der besseren Übersicht Teibereiche oder Verkleinerungen dieses Fenster.



Abbildung~3.10: Klassenansicht des Definitionsfensters



Abbildung~3.11: Methoden in der Klassenansicht

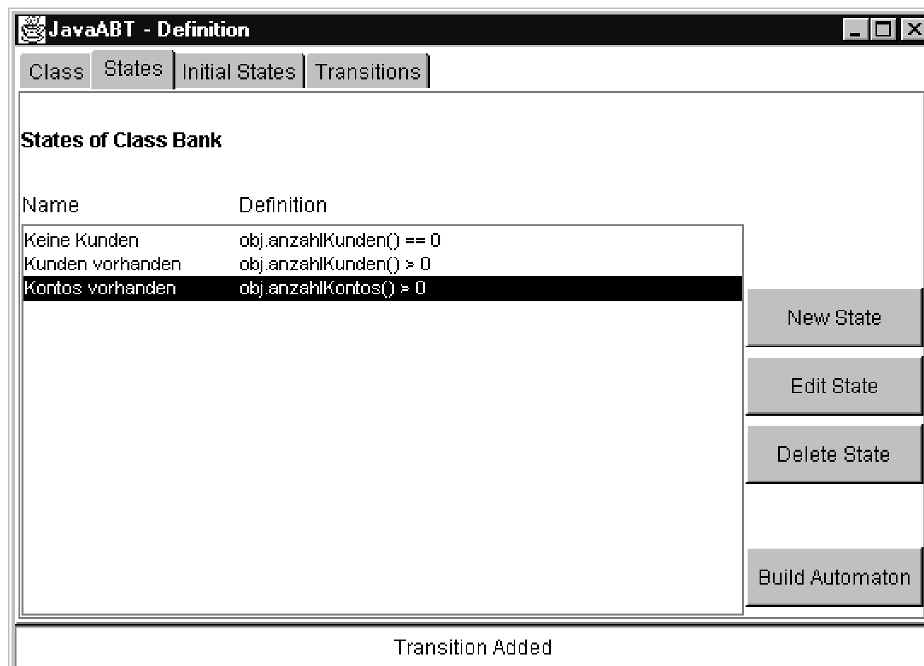
Methods	State Changing	State Neutral
neuerKunde (String name)	◉	○
neuesKonto (String kunde,int nummer)	◉	○
neuesKonto (String kunde,int nummer,double betrag)	◉	○

Abbildung~3.12: Darstellung von Überladung von Methoden im Definitionsfenster

Die Klassenansicht des Definitionsfensters (Abbildung 3.10 zeigt nur einen Ausschnitt) enthält für die Definition des Automaten wichtige Informationen über die Klasse (hier: `Bank`). Dazu zählen Methoden und Instanzvariablen.

Dabei können für die Automatendefinition irrelevante Methoden und Instanzvariablen an dieser Stelle angeben werden und tauchen dann in der Zustand- und Transitionsansicht nicht mehr auf. Im Beispiel sind die Instanzvariablen `Name` und `Bankleitzahl` als zustandsneutral (`State Neutral`) und `Kunden` und `Kontos` als zustandsbestimmend (`State Determinant`) definiert. Analoges gilt für Methoden, die zustandsändernd (`State Changing`) oder zustandsneutral (`State Neutral`) sein können (Abbildung 3.11). Für Methoden läßt sich zusätzlich die zurückgelieferte Instanzvariable angeben (Returns Instance Variable). Sollten Instanzvariablen zustandsbestimmend, aber als versteckt deklariert sein, so gibt es für den Benutzer an dieser Stelle die Möglichkeit, eine Methode zur Definition eines Zustands zu nutzen, die den Wert der Instanzvariable zurückliefert.

Da Methoden überladen werden können, daß heißt, es gibt mehrere Methoden mit dem gleichen Namen, aber anderen Parametern, werden in der Klassenansicht und bei der Definition in den Definitionsdialogen alle Parameter von Methoden angegezeigt. Im Beispiel (Abbildung 3.12) betrifft das die Methoden `neuesKonto(String kunde, int nummer)` und `neues Konto(String kunde, int nummer, double betrag)`. Das Anzeigen aller Methodenparameter ist auch wichtig, da Methodenparameter in der Definition von Bedingungen für Transitionen verwendet werden können.



Abbildung~3.13: Zustandsansicht des Definitionsfensters

In der Zustandsansicht des Definitionsfensters werden die Zustände der Klasse (hier: `Bank`) definiert.

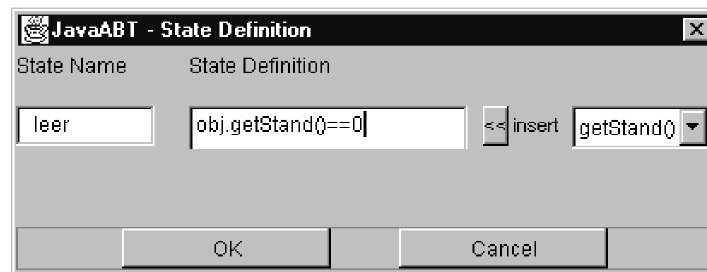
Ein Zustand kann mit dem Button `New State` neu definiert werden, mit `Edit State` kann diese Definition verändert werden, `Delete State` löscht einen Zustand. Die Funktionen zum Löschen und Editieren von Zuständen werden auf den markierten Zustand (im Beipiel `Kontos vorhanden`) angewandt.

Handelt es sich um einen geerbten Zustandsautomaten, so gibt es die weiteren Buttons `Divide State` zur Teilung eines Zustands und `Concurrent Diagram` zur Einführung eines nebenläufigen, konkurrierenden Zustandsdiagramms in der Zustandsansicht. Dabei können die in der Abbildung dargestellten Funktionen nicht auf den geerbten Automaten angewandt werden, die weiteren Funktionen nicht auf den nebenläufigen Zustandsautomaten.

Wird der Zustandsautomat der Oberklasse nur benutzt, gibt es keine Editiermöglichkeit in Form von Buttons. Dieses gilt auch für die im folgenden vorgestellten Ansichten des Definitionsfensters.

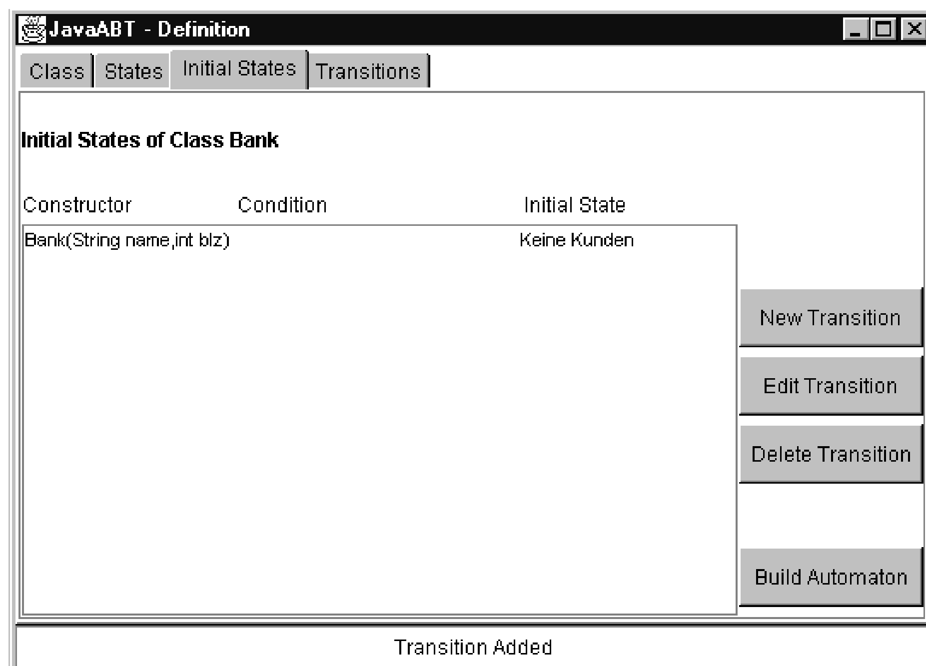
`Build Automaton` entspricht dem Menüpunkt `Build` im Menü `Automaton` des Projektfensters (Abbildung 3.5) und generiert den Programmcode für den Automaten.

Für die Zustandsdefinition (Menüpunkt `New State`, Abbildung 3.13)



Abbildung~3.14: Dialogfenster für die Zustandsdefinition

steht ein Dialogfenster zur Verfügung (Abbildung 3.14), in dem der Benutzer Namen und Definition des Zustands eingeben kann. Sichtbare Instanzvariablen und Methoden, die Instanzvariablen zurückliefern, finden sich dabei im PopUp-Menü (ganz rechts) und können durch `insert` in die Definition eingefügt werden. Dabei ergänzt JavaABT selbständig den für die spätere Code-Generierung wichtigen Zusatz `obj`, wobei `obj` für das zu testende Objekt steht.



Abbildung~3.15: Anfangszustandsdefinition im Definitionsfenster

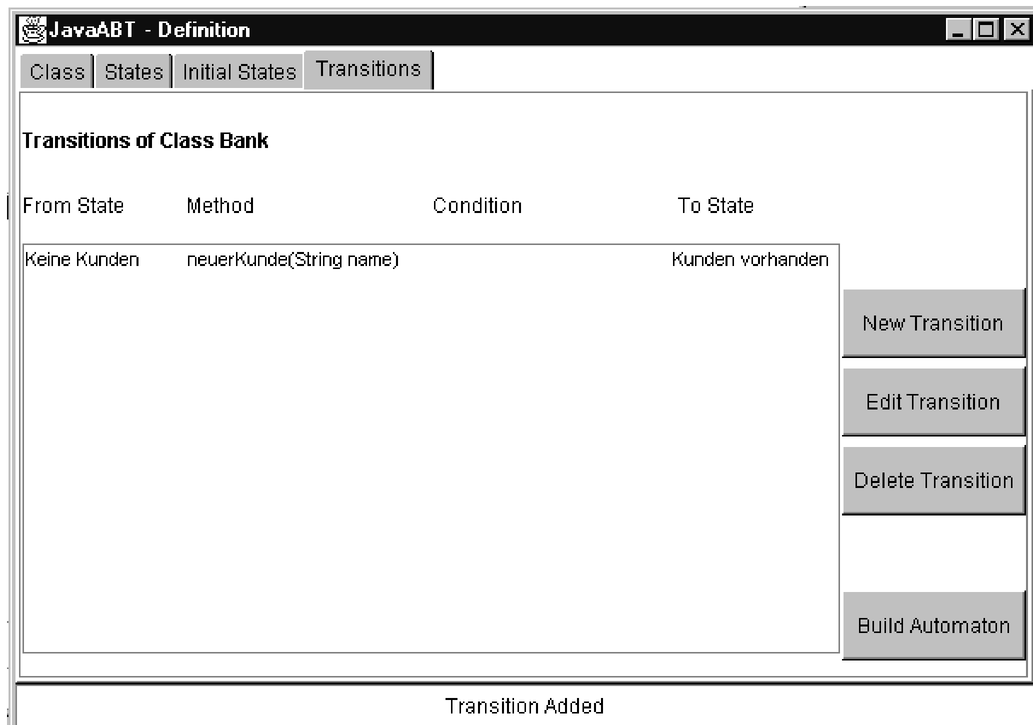
In der in Abbildung 3.15 dargestellten Ansicht des Definitionsfensters lassen sich die Anfangszustände definieren. Anfangszustände sind Zustän-

de, in denen sich Objekte nach Aufruf eines Konstruktors, also nach der Initialisierung befinden.

Mit dem Button `New Transition` wird ein Dialog zur Definiton aufgerufen, mit `Edit Transition` lassen sich Transitionen ändern und mit `Delete Transition` löschen. Build Automaton besitzt dieselbe Funktionalität wie in der Zustandsansicht (Abbildung 3.13).

Im Beispiel führt der Aufruf des Konstruktors mit den Parametern `name` und `blz` in den Zustand `Keine Kunden`.

Wurde der Zustandsautomat nach strenger Vererbung geerbt, so findet sich in diesem ebenfalls wie in der Zustandsansicht (Abbildung 3.13) der Button `Concurrent Diagram`, mit dem ein nebenläufiges, konkurrierendes Zustandsdiagramm definiert werden kann. Da nach McGregor [MD93] im geerbten Zustandsdiagramm auch neue Transitionen definiert werden können, läßt sich die Funktion `New Transition` auch auf das geerbte Zustandsdiagramm anwenden. Editieren lassen sich jedoch nur neu hinzugefügte Transitionen, Transitionen nur zwischen Zuständen eines Zustandsdiagramms definieren (ursprüngliches oder konkurrierendes Zustandsdiagramm).

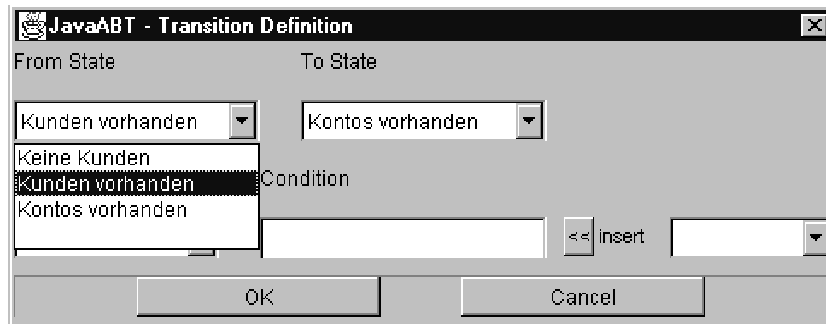


Abbildung~3.16: Transitionsansicht Definitionsfenster

Transitionen, die als Ereignis eine Methode besitzen, werden in der Tran-

sitionsansicht (Abbildung 3.16) definiert. Diese ist aufgebaut wie die Ansicht zur Definition von Anfangszuständen, zeigt jedoch in der Liste auch den Ausgangszustand einer Transition an, der bei der Definition der Anfangszustände nicht angezeigt wird, da er immer undefiniert ist.

Alle Angaben für die Definitionsansicht der Anfangszustände gelten auch für die Transitionsansicht.



Abbildung~3.17: Dialogfenster zur Definition von Transitionen

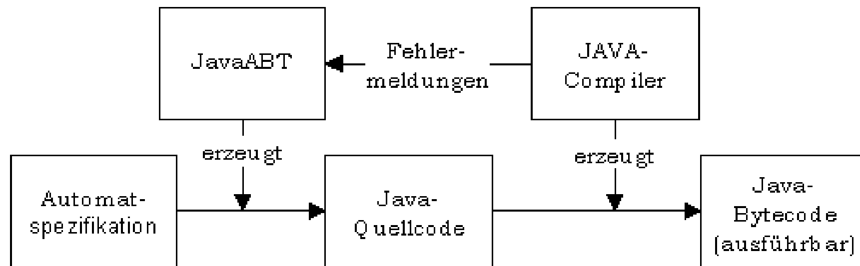
Der Dialog zur Definition von Transitionen (Abbildung 3.17) ist ähnlich aufgebaut wie das Fenster zur Definition der Zustände (Abbildung 3.14).

Alle Zustände finden sich in den PopUp-Menüs `From State` und `To State` und können dort selektiert werden. Nur zwischen bereits definierten Zuständen können Transitionen gebildet werden. Dies dient der Vermeidung von Fehlern. Wird ein Zustand gelöscht, werden auch alle von ihm abhängigen Transitionen gelöscht.

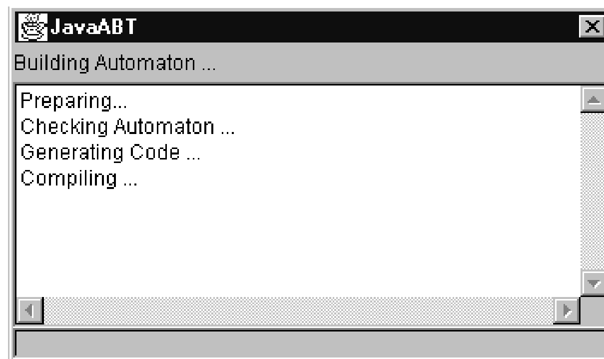
Alle zustandsändernden Methoden finden sich in einem PopUp-Menü (in der Abbildung vom PopUp-Menü `From State` verdeckt). Bedingungen werden wie die Definitionen der Zustände definiert.

Ein ähnliches Dialogfenster steht auch für die Definition der Anfangszustände zur Verfügung, einzige Unterschiede sind, daß statt Methoden Konstruktoren ausgewählt werden können und der Ausgangszustand (`From State`) nicht durch ein PopUp-Menü auswählbar ist, sondern fest auf `Undefined` gesetzt wird.

3.2.2.3 Generierung des Automaten



Abbildung~3.18: Generierung des Automaten



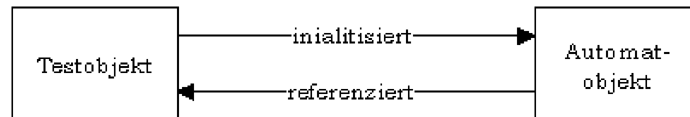
Abbildung~3.19: Ausgaben der JAVA-Code-Generierung

Die Generierung des Automaten geschieht grundsätzlich in zwei Schritten (Abbildung 3.18). Zuerst wird aus den Daten der Eingabe durch den Benutzer der JAVA-Code für die Automatenklasse erzeugt. Anschließend wird dieser Programmcode übersetzt, in dem JavaABT den JAVA-Compiler (javac) anstößt.

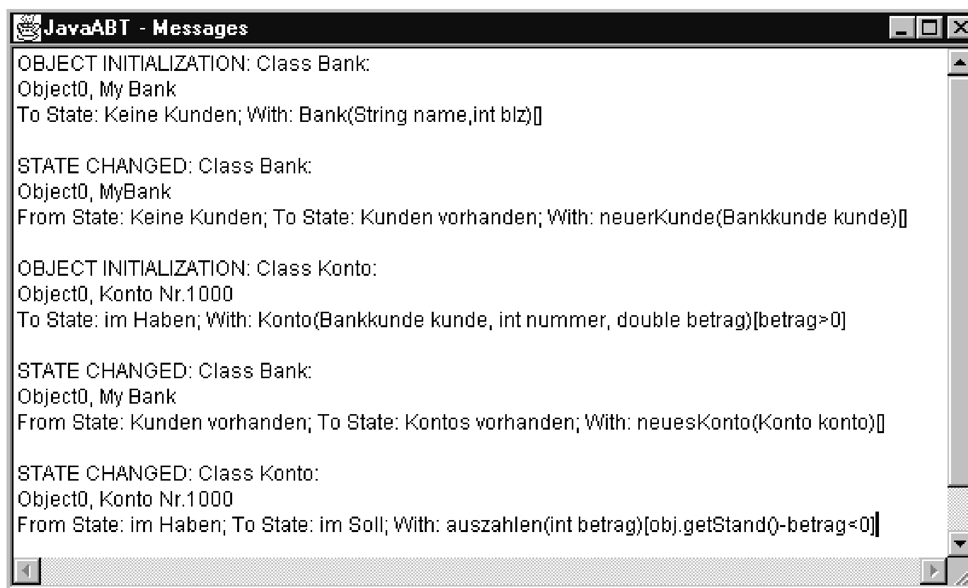
Die Ausgaben erfolgen in einem Fenster (Abbildung 3.19), das den aktuellen Status der Generierung wiedergibt.

Der JAVA-Compiler gibt beim Compilieren auftretende Fehler an JavaABT zurück. JavaABT gibt diese Fehlermeldungen in dem Ausgabefenster an den Benutzer weiter und zwingt ihn damit, seine vorher geleistete Definition noch einmal auf Fehler zu untersuchen. Die Fehler lassen sich lokalisieren über in den generierten JAVA-Code eingebrachte Kommentare, die in jeder Zeile den Zustand oder die Transition identifizieren, die den Fehler verursacht hat.

3.2.2.4 Ausgaben des Automatenobjekts



Abbildung~3.20: Beziehung zwischen zu testendem und Automatenobjekt



Abbildung~3.21: Ausgabefenster der Automatenklassen

Ein Objekt des so generierten Automaten wird erzeugt für jedes Objekt der zu testenden Klasse (in Abbildung 3.20 Testobjekt genannt). Um die Automaten an die zu testende Klasse zu binden, wird diese zuerst mit einem Instrumentierer bearbeitet oder von Hand instrumentiert. Der Instrumentierer schreibt in die Ursprungsklasse eine Referenz auf die Automatenklasse, so daß bei der Erzeugung eines Objekts der zu testenden Klasse ein Objekt der Automatenklasse erzeugt wird. Dieser hält eine Referenz auf das zu testende Objekt und protokolliert alle Aktivitäten desselben.

Während der Laufzeit erhält das Automatenobjekt bei jedem Aufruf einer Methode im zu testenden Objekt eine Nachricht, die es prüfen läßt, ob das zu testende Objekt laut spezifizierten Automaten diese Methode in diesem Zustand überhaupt aufrufen darf und ob der Zustand des Objekts nach Ausführung der Methode dem Zustand entspricht, in dem es sich befinden

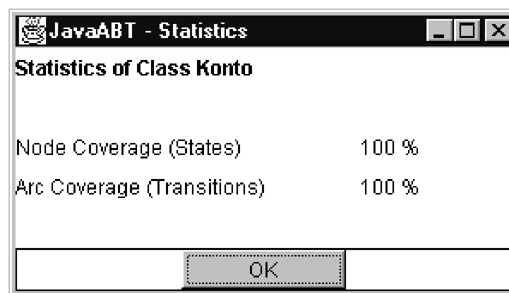
sollte. Dazu hält das Automatenobjekt neben der Referenz auf das von ihm zu testende Objekt den Zustand, in dem es sich befindet (bzw. befinden sollte).

Fehler, die auftreten können und gemeldet werden, sind:

- Eine Methode, die im aktuellen Zustand nicht angewendet werden kann
- Ein Zustand, der nicht dem erwarteten entspricht
- Ein Konstruktor, der das Objekt nicht in einen definierten Zustand überführt
- Eine Methode, die das Objekt in einen undefinierten Zustand überführt
- Eine nicht zustandsändernde Methode, die den Zustand ändert

Die Fehler und andere Meldungen über aufgerufene Methoden und Zustandsänderungen werden im Fenster für die Ausgabe aller aktuell vorhandenen Automatenobjekte aller Klassen angezeigt (Abbildung 3.21).

3.2.2.5 Statistik



Abbildung~3.22: Statistikfenster

Gleichzeitig zu der Fehlerentdeckung werden Teile der anfallenden Daten protokolliert, um sie später zu statistischen Zwecken zu nutzen. Die Statistik umfaßt Zustands- und Transitionsüberdeckung und Anzeige der aufgetretenen Fehler. Die Überdeckungsmaße werden dabei nach der Formel von Liggesmeyer [Lig90], vorgestellt in Kapitel 2.2, berechnet. Dabei entspricht die Zustandsüberdeckung der Überdeckung der Knoten (`Node_Coverage`), die Transitionsüberdeckung der Überdeckung der Kanten (`Arc Coverage`).

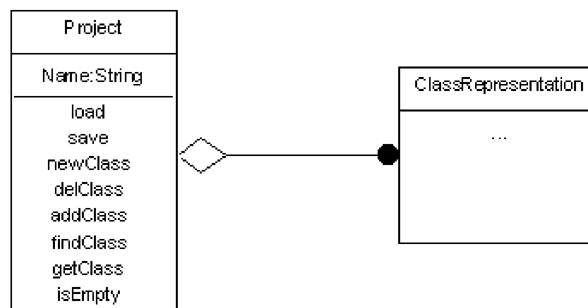
Die Ausgabe der Überdeckungsmaße erfolgt im Statistikfenster (Abbildung 3.22).

3.3 Entwurf

In diesem Kapitel wird der Entwurf des in Kapitel 3.2 konzeptionell beschriebenen Systems vorgestellt. Der Entwurf wurde in der *Object Modeling Technique* (OMT), vorgestellt in [RBP+91], vorgenommen. Es wurden bereits Besonderheiten der Programmiersprache JAVA, zum Beispiel die fehlende Mehrfachvererbung, berücksichtigt.

Zunächst werden stets die modellierten Klassen und ihre Beziehungen untereinander vorgestellt, anschließend, wenn es nötig erscheint, ein dynamisches oder funktionales Modell aufgestellt, um den Sachverhalt zu verdeutlichen. Im Klassendiagramm werden nur die Methoden aufgeführt, die wichtig sind, da die Angabe aller Methoden zu einer unübersichtlichen grafischen Darstellung führt.

3.3.1 Projektverwaltung



Abbildung~3.23: Objektmodell Project

Zur Verwaltung von Projekten existiert in JavaABT die Klasse `Project`. Das entsprechende Objektdiagramm zeigt Abbildung 3.23. Die Klasse `Project` besitzt das Attribut Name und besteht aus einer Anzahl von Klassenrepräsentationen (`ClassRepresentation`), die auch leer sein können. Die Klassenrepräsentation wird ausführlich in Abschnitt 3.3.2 beschrieben.

Die Klasse `Project` stellt verschiedene Methoden zur Klassenverwaltung zur Verfügung. Im einzelnen sind das:

- Methoden für das Laden und Speichern von Projekten

  `load` zum Laden

  `save` zum Speichern

- Methoden zur Verwaltung der Liste von Klassenrepräsentationen

  `newClass` erzeugt aus dem Namen und dem Quellcode-Pfad einer Klasse eine Klassenrepräsentation und fügt sie der Liste hinzu

`addClass` fügt eine schon bestehende Klassenrepräsentation der Liste hinzu

`delClass` löscht eine Klassenrepräsentation aus der Liste

`findClass` sucht eine Klassenrepräsentation und gibt ihren Index zurück

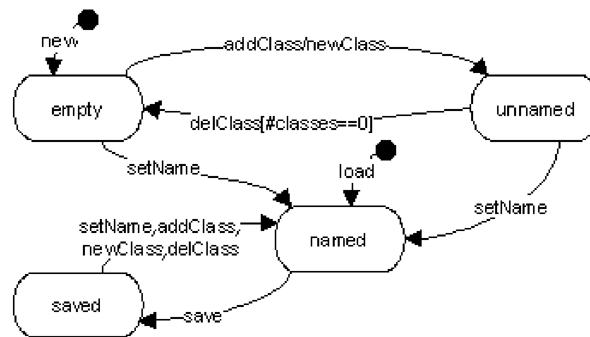
`getClass` sucht eine Klassenrepräsentation und gibt sie zurück

- Methoden zum Projektzustand

  `isEmpty` meldet, ob die Liste der Klassenrepräsentationen und damit das Projekt leer ist

  `getClassesCount` gibt die Anzahl der Klassenrepräsentationen zurück

Zusätzlich gibt es wie bei allen weiteren vorgestellten Klassen die üblichen Methoden für lesenden und schreibenden Zugriff auf die Instanzvariablen.

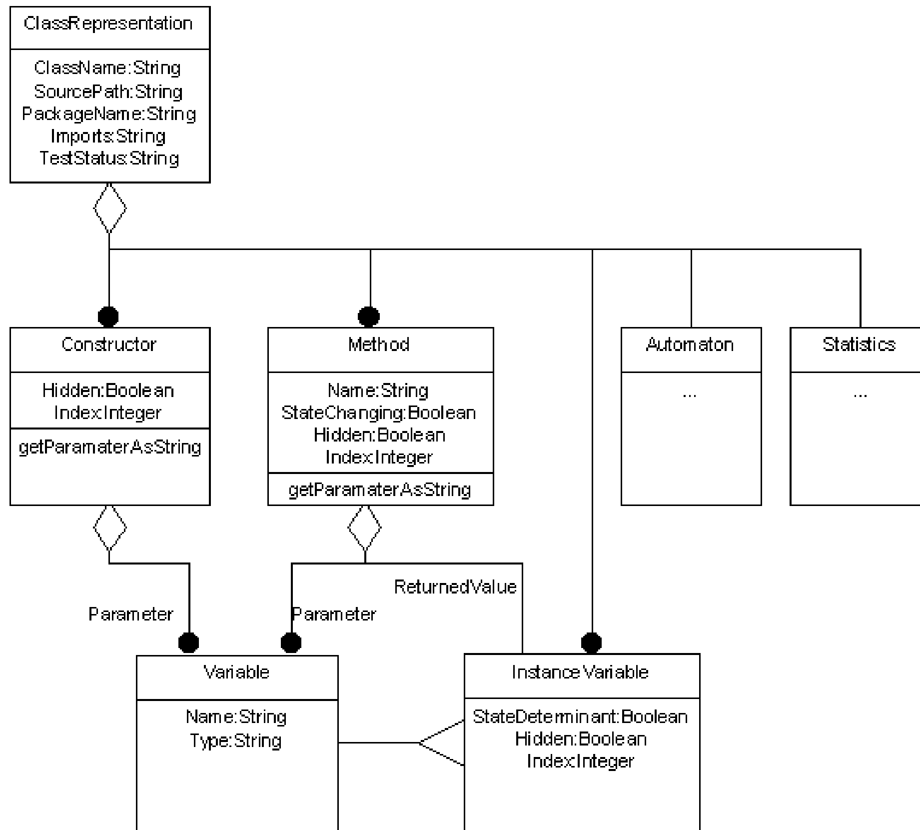


Abbildung~3.24: Dynamisches Modell Project

Anhand eines dynamischen Modells kann das grundlegende Verhalten der Klasse `Project` verdeutlicht werden (Abbildung 3.24). Es gibt die Zustände `empty`, `unnamed`, `named` und `saved`. Der Zustand `empty` repräsentiert ein Projekt ohne Klassenrepräsentationen und ohne Namen. Der Zustand `unnamed` steht für ein Projekt mit mindestens einer Klassenrepräsentation, aber ohne Namen. Der Zustand `named` wird angenommen, nachdem das Projekt benannt wurde, der Zustand `saved` nach dem Speichern.

Das dynamische Modell in Abbildung 3.24 verdeutlicht einige Eigenschaften von Projekten in JavaABT. Ein Projekt kann durch `new` oder `load` erzeugt werden. Dabei führt `new` das Projekt in den Zustand `empty` über, `load` in den Zustand `named`. Ein Projekt kann nicht gespeichert werden, ohne vorher benannt zu werden. Jede Aktion, die durch eine Methode verursacht wird, die die Liste der Klassenrepräsentationen oder den Namen des Projekts ändert, führt aus dem Zustand `saved` heraus.

3.3.2 Klassenrepräsentation



Abbildung~3.25: Objektmodell Klassenrepräsentation

In JavaABT müssen vor der Definition eines Automaten Informationen über die zu testende Klasse erfragt und in einer Objektstruktur gespeichert sowie Automat, Teststatus und Statistik verwaltet werden. Dazu wurde eine Klasse `ClassRepresentation` entworfen, deren Objektdiagramm in Abbildung 3.25 dargestellt ist.

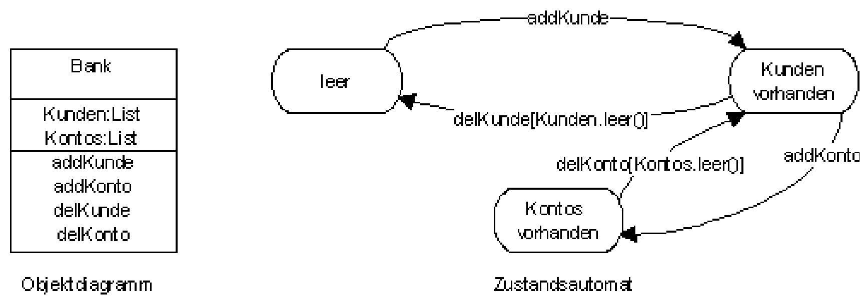
Eine Klassenrepräsentation besitzt als Attribute den Namen der Klasse, die sie repräsentiert (`ClassName`)[5], den Pfad des Quellcodes (`SourcePath`), den Namen des Packages[6] (`PackageName`), die importierten Klassen (`Imports`) und den Teststatus (`TestStatus`).

[5]Die in Klammern stehenden Begriffe enthalten den Namen der Attribute der Klasse, die in der Abbildung 3.25 dargestellt sind.

[6]Klassen können in Java Packages zugeordnet werden, so daß sie gemeinsam importiert werden können und Zugriff auf bestimmte Instanzvariablen und Methoden haben.

Der Klassenname wird zur Anzeige im Projektfenster (siehe Kapitel 3.2.2.1) und zur Generierung des Java-Code des Zustandsautomaten (siehe Kapitel 3.3.5) benötigt.

Der Quellcode-Pfad und der Name des Packages werden gebraucht, da sich die Klasse des generierten Automaten im selben Package wie die zu testende Klasse befinden soll und deshalb auch den gleichen Pfad aufweisen muß.



Abbildung~3.26: Beispiel eines Objekt- und Zustandsdiagramms einer Bank

Alle von der zu testenden Klasse importierten Klassen und Packages müssen auch von der Automatenklasse importiert werden, da nur so sicher gestellt ist, daß die Automatenklasse alle definierten Operationen auch ausführen kann und die Typen aller Instanzvariablen und Methodenparameter kennt.

Dies wird anhand eines kleinen Beispiel, dargestellt in Abbildung 3.26, verdeutlicht. Die Klasse `Bank` besitzt die Attribute `Kunden` und `Kontos`, beide vom Typ `List`. Auf Listen läßt sich die Methode `leer` anwenden, die als Wahrheitswert zurückgibt, ob eine Liste leer ist oder Elemente beinhaltet. Die Methode `addKunde` der Klasse Bank fügt einen Kunden hinzu, die Methode `delKunde` löscht einen Kunden. Analog dazu addieren oder löschen die Methoden `addKonto` und `delKonto` ein Konto. Ein Konto kann nur eingefügt werden, wenn mindesten ein Kunde (der Besitzer des Kontos) vorhanden ist.

Ein einfaches Zustandsdiagramm, ebenfalls in Abbildung 3.26 dargestellt, wurde für die Klasse `Bank` aufgestellt. Alle Zustände werden über die Attribute `Kunden` und `Kontos` definiert. Da der Zustandsautomat sowohl die Attribute `Kunden` und `Kontos` in der Zustandsdefinition nutzt als auch auf die Methode `leer` der Klasse `List` in den Transitionen `delKunde[Kunden.leer()]` und `delKonto[Kontos.leer()]` zugreift, muß die Klasse `List` in der Automatenklasse importiert werden. Bei Übernahme aller Importe der Klasse `Bank` in die Automatenklasse geschieht das automatisch, da die

Klasse `List` von der Klasse `Bank` zur Deklaration der Instanzvariablen[7] importiert wird.

Ein Objekt der Klasse `ClassRepresentation` (Abbildung 3.25) besteht darüber hinaus aus Objekten der Klassen `Constructor` als Repräsentation der Konstruktoren, `Method` als Repräsentation der Methoden und `InstanceVariable` als Repräsentation der Instanzvariablen. Davon können jeweils beliebig viele vorhanden sein. Jeweils genau ein Automat (Klasse `Automaton`) und eine Statistik (Klasse `Statistics`) sind einer Klassenrepräsentation zugeordnet.

Die Repräsentationen von Konstruktoren, Methoden und Instanzvariablen besitzen einen Index, mit dem sie eindeutig identifiziert werden können[8] und der zur Sortierung dient. Allen gemeinsam ist überdies das Attribut `Hidden`, das angibt, ob das jeweilige Element der zu testenden Klasse für die Automatenklasse lesbar ist. Nichtlesbar oder `Hidden` sind für die Automatenklasse Elemente, die als *private* oder *protected*[9] deklariert wurden, lesbar sind Elemente, die als *public* oder im Package[10] lesbar deklariert wurden.

Konstruktoren und Methoden bestehen aus einer Anzahl von Parametern, die durch die Klasse `Variable` repräsentiert werden, die nur die Attribute `Name` und `Type` besitzt und Methoden zum lesenden Zugriff auf diese Attribute zur Verfügung stellt (`getName`, `getType`).

Von dieser Klasse `Variable` erbt die Klasse `InstanceVariable`, da es sich bei Instanzvariablen nur um eine in der zugehörenden Klasse besonders ausgezeichnete Variable handelt.

Instanzvariablen können zustandsbestimmend oder zustandsneutral sein (siehe Kapitel 3.2.2.2), vertreten durch das Attribut `StateDeterminant`, Methoden können zustandsändernd oder zustandsneutral sein, vertreten durch das Attribut `StateChanging`. Der Rückgabewert von Methoden wird, wenn es sich um eine Instanzvariable handelt, durch `ReturnedValue` angegeben. Diese Attribute dienen der vereinfachten Definition von Zustandsautomaten (beschrieben in Kapitel 3.2.2.2).

Die Klasse `Method` besitzt ein Attribut `Name`, daß den Methodennamen vertritt. Dagegen besitzt die Klasse `Constructor` dieses Attribut nicht, da Konstruktoren in JAVA nicht benannt werden.

Besonderheit der Klassen `Method` und `Constructor` ist die beiden gemeinsame Methode `getParameterAsString`, die die Parameter in Java-Code zurückgibt. Als Beispiel: Hätte eine Methode die Parameter `name` vom

---

[7] Instanzvariable und Attribut werden synonym gebraucht.

[8] Objekte sind natürlich eindeutig identifizierbar. Da JAVA jedoch keinen Mechanismus zum Speichern und Laden von Objekten bietet, sondern nur von Basistypen, wird der Index gebraucht, um Beziehungen nach den Laden wieder herzustellen.

[9] Elemente, die in JAVA als *protected* deklariert wurden, können nur von erbenden Klassen gelesen werden.

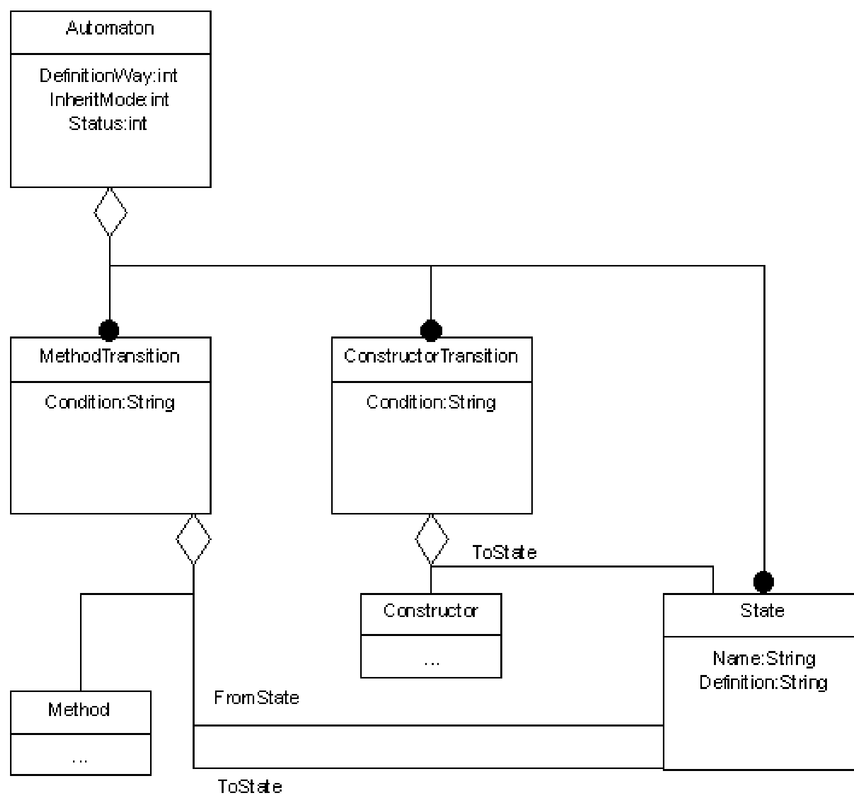
[10] Im Package lesbare Elemente haben keinen spezifischen Modifizierer.

Typ `String` und `alter` von Typ `int`, so würde `getParameterAsString` folgende String zurückliefern: `''(String name, int alter)''`. Dieses dient einerseits der Unterscheidung überladener Methoden im Definitionsfenster (siehe Kapitel 3.2.2.2), andererseits dem Java-Code-Generator (Kapitel 3.2.2.3 und 3.3.5). Die Namen der Parameter von Methoden und Konstruktoren werden von an die JAVA-Reflection-Klassen angelehnte Klassen, vorgestellt in [Pos97], zurückgeliefert, wenn die Klassen zuvor instrumentiert wurden (siehe Kapitel 3.3.6).

Alle anderen Methoden der in diesem Kapitel vorgestellten Klassen sind Methoden für den lesenden und schreibenden Zugriff auf Instanzvariablen.

Die Klasse `Automaton` wird ausführlich in folgenden Kapitel vorgestellt, die Klasse `Statistics` in Kapitel 3.3.7.

3.3.3 Zustandsautomaten



Abbildung~3.27: Objektmodell Automaton

JavaABT läßt die Definition von Zustandsautomaten zu, die einen Teil der Statecharts (Kapitel 2.4.3) implementieren. Unterstützt werden Bedin-

gungen, nicht unterstützt werden Aktionen/Aktivitäten und hierarchische Zustandsautomaten. Aktionen/Aktivitäten sind für das Automatenobjekt nicht wahrnehmbar, sondern müssen auf andere Weise getestet werden, hierarchische Zustandsautomaten dienen mehr der Übersichtlichkeit des Entwurfs eines Systems und haben keinen Einfluß auf die Durchführbarkeit des Tests.

Die vom Benutzer definierten Zustandsautomaten werden in JavaABT durch einige Klassen repräsentiert, die nachfolgend beschrieben und in Zusammenhang gesetzt werden sollen (Abbildung 3.27).

Die Modellierung von Zustandsautomaten setzt die Definition von Zuständen voraus, zwischen denen Zustandsübergänge definiert werden. Diesem Konzept folgt der Entwurf. Als Basisklasse[11] des Automaten gibt es die Klasse `State`, darauf aufbauend die Klassen `MethodTransition` und `ConstructorTransition`. Die Klasse `State` verfügt über die Attribute `Name` und `Definition`, beide vom Typ String, so daß jedem Zustand ein Name und eine Definition zugewiesen werden kann (siehe Kapitel 3.2.2.2).

Die Klasse `ConstructorTransition` repräsentiert den Übergang eines Objekts aus dem undefinierten Zustand in einen definierten Anfangszustand. Dabei bildet der Konstruktor (Klasse `Constructor`) der zu testenden Klasse das Ereignis, das den Zustandsübergang auslöst. Konstruktortransitionen besitzen keinen Zustand, aus dem sie hinausführen, nur einen, in den sie hineinführen (`ToState`). Die Klasse `MethodTransition` repräsentiert den Übergang eines Objekts von einem definierten Zustand in einen anderen definierten Zustand, wobei eine Methode (Klasse `Method`) das auslösende Ereignis darstellt. Die beiden Zustände werden durch `FromState` und `ToState` repräsentiert. Beide Klassen besitzen eine Bedingung (`Condition`), die die Bedingung darstellt, unter der die Transition ausgelöst wird.

Der Automat (siehe Abbildung 3.27) besteht aus Zuständen, Methodentransitionen und Konstruktortransitionen und besitzt die Attribute `DefinitionWay`, `InheritMode` und `Status`.

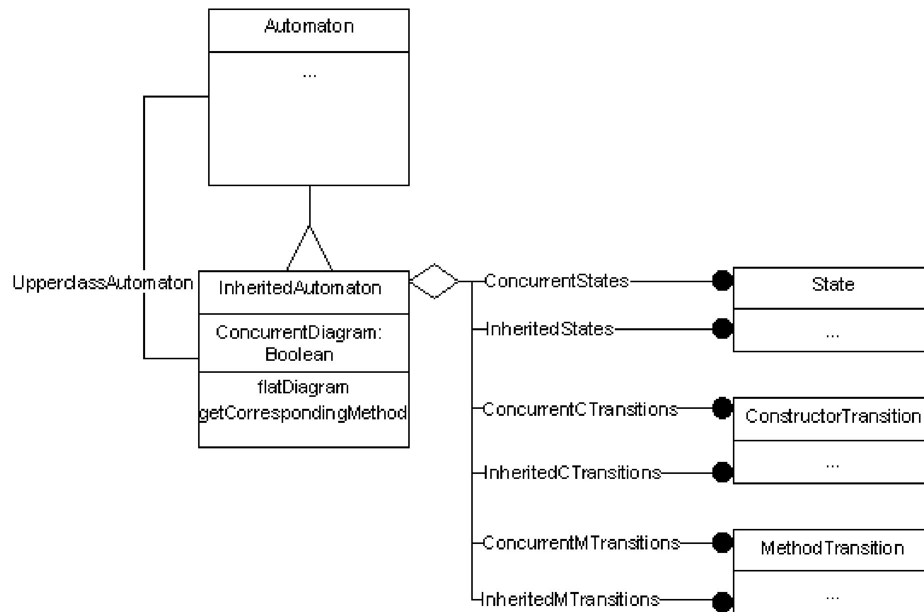
Das Attribut `DefinitionWay` dient dazu, die Art der Definition festzulegen, `InheritMode` speichert die Vererbungsart und Status gibt den Definitionsstatus an, welcher undefiniert, definiert und generiert[12] sein kann (siehe Kapitel 3.2.2.2).

3.3.4 Vererbte Zustandsautomaten

Da JavaABT die strenge Vererbung von Zustandsautomaten nach [MD93] (erläutert in Kapitel 2.5.2) unterstützt, mußte auch ein Vererbungsmecha-

[11] Basisklassen sind hier Klassen, auf die andere Klassen von JavaABT aufbauen und die selbst nur JAVA-eigene Klassen benutzen, sprich die unterste Ebene von JavaABT.

[12] Generiert bedeutet, daß die korrespondierende Automatenklasse in JAVA erzeugt wurde.



Abbildung~3.28: Objektmodell Vererbte Zustandsautomaten

nismus modelliert werden. Für die Repräsentation eines streng vererbten Unterklassenautomaten wurde eine Unterklasse der Klasse Automaton definiert, `InheritedAutomaton` (Abbildung 3.28).

Ein geerbter Automat weist auf den Automaten der Oberklasse (in der Abbildung 3.28 `UpperclassAutomaton`), wodurch alle Zustände und Transitionen[13] in der Unterklasse zur Verfügung stehen. Die Konstruktor- und Methodentransitionen, die mit dem Präfix `Inherited` gekennzeichnet sind, repräsentieren nachträglich in das geerbte Zustandsdiagramm eingefügte oder veränderte Transitionen. `Inherited`States sind die Zustände, die durch die Teilung eines aus den Zuständen der Oberklasse entstanden sind sowie durch Änderung des Namens modifizierte Zustände der Oberklasse. Die zur Oberklasse verschiedenen Zustände und Transitionen werden durch eigene Klassen repräsentiert, die einen Verweis auf den ererbten Zustand oder die ererbte Transition enthalten. Mit `Concurrent` werden die Zustände und Transitionen gekennzeichnet, die einem konkurrierenden, nebenläufigen Zustandsdiagramm angehören. Ob ein konkurriendes Zustandsdiagramm vorliegt, wird im Attribut `ConcurrentDiagramm` vermerkt.

Die geerbten Eigenschaften `States`, `MethodTransitions` und `ConstructorTransitions` ohne Präfix dienen zur Speicherung des durch die Methode `flatDiagramm` der Klasse `InheritedAutomaton` flachgedrückten Zustands-

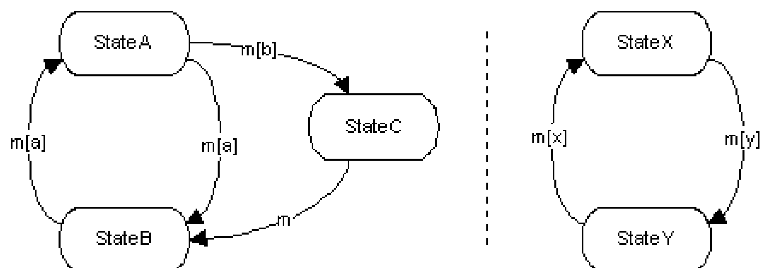
[13]Unter Transitionen werden im folgenden beide Arten von Transitionen verstanden, Methoden- und Konstruktortransitionen.

diagramms. Das Flachdrücken von Zustandsautomaten entspricht der Auflösung der Nebenläufigkeit von Zustandsdiagrammen, wie sie bei der Beschreibung der Harel-Automaten in Kapitel 2.4.3 dargestellt wurde. In diesem Kapitel findet sich auch ein entsprechendes Beispiel in den Abbildungen 2.12 und 2.13. Ist kein nebenläufiges, konkurrierendes Zustandsdiagramm vorhanden, bewirkt `flatDiagramm` nur ein Kopieren der Zustände und Transitionen, die im geerbten Zustandsdiagramm enthalten sind. Auf diese flachgedrückten Zustandsdiagramme kann dann sowohl die Unterklasse des aktuellen Automaten als auch der Java-Code-Generierer einheitlich zugreifen, ohne daß die Klasse des Automaten (`Automaton` oder `InheritedAutomaton`) bekannt sein muß.

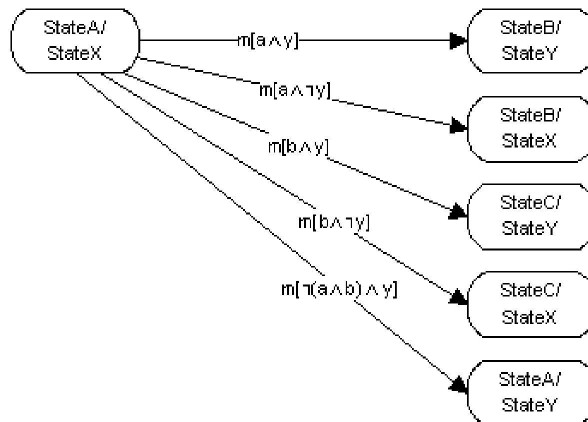
Vererbte Zustandsautomaten weisen durch ihre Struktur einige Besonderheiten auf, die zu einer gesonderten Behandlung dieser Automaten führen. So muß vor der Generierung des Programmcodes der Automatenklasse der definierte Automat, der ja zum Teil aus dem Automaten der Oberklasse besteht, zusammengeführt und nebenläufige Zustandsdiagramme müssen zu einem Diagramm zusammengedrückt werden.

Die Überführung von Oberklassenzustandsautomat und Unterklassenzustandsautomat zu einem Automaten bringt das Problem mit sich, daß aufgrund der gewählten Struktur Methoden der Ober- und Unterklasse getrennt verwaltet werden. Eine Transition der Oberklasse besitzt als Ereignis eine Methode der Oberklasse, eine Transition der Unterklasse eine Methode der Unterklasse. Durch die Methode `getCorrespondingMethod` in der Klasse `InheritedAutomaton`, die den Unterklassenautomaten repräsentiert, kann zu jeder Methode der Oberklasse die entsprechende Methode der Unterklasse ermittelt werden.

Alle im Unterklassenautomaten veränderten Zustände und Transitionen, zum Beispiel durch die Teilung eines Zustandes, ersetzen bei der Zusammenführung die korrespondierenden Zustände und Transitionen des Oberklassenautomaten.



Abbildung~3.29: Beispiel für ein nebenläufiges Zustandsdiagramm

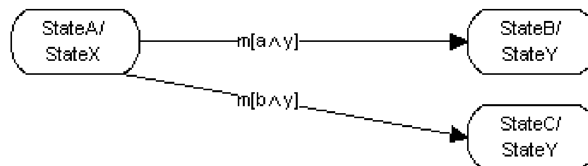


Abbildung~3.30: Beispiel für aufgelöste Nebenläufigkeit

Nebenläufige Zustandsdiagramme werden in JavaABT zusammengeführt, bevor der JAVA-Programmcode generiert wird.

Der Algorithmus zur Zusammenführung der Zustände nimmt alle Zustände der beiden Diagramme und bildet alle Permutationen daraus. `StateA` und `StateX` werden so zu `StateA/StateX` (siehe Abbildungen 3.29 und 3.30).

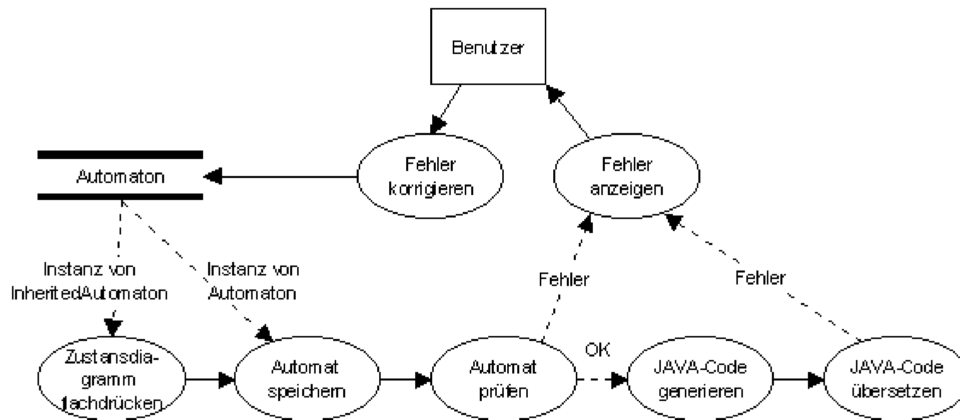
Der Algorithmus zum Zusammenführen der nebenläufigen Zustandsdiagramme muß die möglichen Überschneidungen von Bedingungen in Transitionen berücksichtigen. Das Beispiel in Abbildung 3.29 verdeutlicht dieses. Die nebenläufigen Zustandsdiagramme nutzen dieselbe Methode zur Definition der Transitionen (`m`), jedoch mit unterschiedlichen Bedingungen. Deshalb ist eine Permutation aller Bedingungen der Methoden nötig, die aus einem Zustand herausführen (im Beispiel Zustand `StateA/StateX`, Abbildung 3.30). Dieses gilt nur, wenn die Definition des Automaten auf die Weise erfolgt ist, daß alle nicht definierten Transitionen keine Zustandsänderung bewirken.



Abbildung~3.31: Beispiel für aufgelöste Nebenläufigkeit

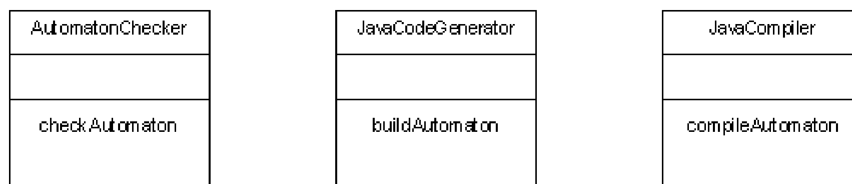
Wurde als Art der Definition die Möglichkeit gewählt, daß alle nicht definierten Transitionen verboten sind, vereinfacht sich das Diagramm in der in Abbildung 3.31 angegebenen Weise.

3.3.5 Java-Code-Generierung



Abbildung~3.32: Funktionales Modell Automatengenerierung

Die Generierung des Java-Codes des Automaten geschieht in mehreren Schritten. Abbildung 3.32 zeigt das funktionale Modell der Generierung des JAVA-Codes des Automaten. Die im Automatenobjekt gespeicherten Daten werden gespeichert. Handelt es sich um einen vererbten Zustandsautomaten, so wird zuvor der Zustandsautomat flachgedrückt. Anschließend werden die Automatendaten auf ihre Zulässigkeit überprüft und der JAVA-Code generiert. Nach dem Generieren erfolgt die Übersetzung durch Aufruf des JAVA-Compilers. Treten Fehler an dieser Stelle oder bei der Automatenüberprüfung auf, so erhält der Benutzer eine Rückmeldung und hat die Möglichkeit, die Automatendaten zu korrigieren.



Abbildung~3.33: Objektmodell der Klassen zur Automatengenerierung

Zur Generierung der Automatenklasse wurden drei Klassen modelliert, die jeweils einen Teil der Aufgabe übernehmen (Abbildung 3.33). Der `AutomatonChecker` überprüft den Automaten durch `checkAutomaton`, der `JavaCodeGenerator` generiert den JAVA-Code mit `buildAutomaton` und der `JavaCompiler` übernimmt die Übersetzung in JAVA-Bytecode durch `compileAutomaton`.

Dieser Ansatz wurde gewählt, da diese drei Klassen in jeweils einer nach außen sichtbaren Methode und einigen versteckten Hilfsmethoden sehr viel Programmcode enthalten. Eine Aufteilung nach Aufgaben hat den Vorteil der Übersichtlichkeit. Aus diesem Grund wurde die genannte Funktionalität, die auch in der Klasse `Automaton` hätte modelliert werden können, in drei eigenständige Klassen ausgelagert.

Zur Vereinfachung des Entwurfs des Java-Code-Generierers trägt der Umstand bei, daß Zustände und Transitionen in sortierter Reihenfolge in JavaABT verwaltet werden. Dabei folgt die Sortierung folgenden Kriterien:

- Zustände werden nach der Reihenfolge ihrer Definition durch den Benutzer sortiert.
- Transitionen werden in der folgenden Reihenfolge ihrer Attribute sortiert (siehe Abschnitt 3.3.3): Index der Methode, Ausgangszustand (`FromState`), Bedingung, Zustand, in den die Transition hineinführt (`ToState`). Bei Konstruktortransitionen ist die Reihenfolge bis auf den fehlenden Ausgangszustand gleich. Diese Reihenfolge erleichtert die Generierung der in Kapitel 2.4.4 vorgestellten Implementierung von Marick [Mar95]. Außerdem läßt sich so leicht überprüfen, ob zwei Transitionen bis auf den Zustand, in den sie hineinführen, identisch sind (Kontradiktion).

Der Java-Code-Generierer arbeitet bei der Generierung der Methoden der Automatenklasse nach folgendem Algorithmus:

1. Für alle Methoden m der zu testenden Klasse
2. Generiere den Methodenkopf
3. Für alle Transitionen t der Liste bis Methode der Transition ungleich der aktuellen Methode m
4. Für alle Zustände z der zu testenden Klasse
5. Generiere if-Anweisung mit Zustand als Bedingung
6. Nimm Bedingung c der aktuellen Transition t solange Ausgangszustand der Transition gleich aktuellem Zustand z
7. Generiere if Anweisung mit c als Bedingung, wenn c existiert
8. Generiere Übergang in Zustand, in den Transition hineinführt.

Die Methode `compileAutomaton` der Klasse `JavaCompiler` (Abbildung 3.33) ruft den Compiler, der mit dem JDK[14] mitgeliefert wird (javac), auf. Da JavaABT ein System ist, das zumeist von JAVA-Entwicklern genutzt wird, kann vom Vorhandensein eines JDK ausgegangen werden.

3.3.6 Instrumentierung und generierter JAVA-Code

```
public class MyClass extends MyUpperclass
{
   ...
   protected MyClassAutomaton Automat;

   public MyClass(type1 parameter1)
   {    super();
        Automat = new MyClassAutomaton(this, parameter1);
   }

   public type3 method1()
   {    Automat.checkState();
        type3 returnedValue = operation1();
        operation2;
        Automat.method1();
        return returnedValue();
   }

   public void method2(type2 parameter2)
   {    Automat.checkState();
        super.method2(parameter2);
        ....
        Automat.method2(parameter2);
   }

   protected void finalize()
   {    Automat.checkState();
        super.finalize();
   }
}
```

Abbildung˜3.34: Beipiel für eine instrumentierte Klasse

Zur Verwendung von JavaABT für den Klassentest ist es nötig, zuvor die zu testenden Klassen zu instrumentieren. Dadurch wird eine Überwachung des Zustands eines Objekts möglich (siehe Kapitel 2.6.3). Von der zu testenden Klasse aus muß bei deren Initialisierung das zugehörige Automatenobjekt erzeugt werden und Ereignisse der zu testenden Klasse an die Automatenklasse weitergeleitet werden. Die Instrumentierung wird durch den in [Pos97] vorgestellten Instrumentierer durchgeführt.

[14] *Java Development Kit*, Programmierumgebung für JAVA

Die Automatenaufrufe werden an den in Abbildung 3.34 im Beispiel gezeigten Einfügungsstellen vorgenommen. Die Aufrufe des Automaten sind mit fetter Schrift hervorgehoben.

Zuerst muß in die zu testende Klasse die Automatenklasse als neue Instanzvariable eingeführt werden (`protected MyClassAutomaton Automat`).

Bei Aufruf eines Konstruktors in der zu testenden Klasse wird zuerst das Automatenobjekt initialisiert. Neben den Parametern des Konstruktors, die an den Automaten weitergereicht werden, muß auch das zu testende Objekt übergeben werden (`this`), um dem Automaten eine Überprüfung des Zustands zu ermöglichen.

Zu Beginn jeder Methode wird der aktuelle Zustand des Automatenobjekts mit dem zu testenden Objekt durch `checkState` abgeglichen. Ausnahme sind Konstruktoren, da diese das zu testende Objekt aus einem undefinierten Zustand herausführen. Beim Ausstieg aus einer Methode, also entweder am Ende bei Methoden ohne Rückgabewert (im Beispiel `method2`) oder vor jedem Aufruf von `return` (im Beispiel `method1`) bei Methoden mit Rückgabewert, muß der Aufruf der entsprechenden Methode der Automatenklasse mit allen Parametern, die auch der Methode der zu testenden Klasse übergeben wurden, erfolgen (Abbildung 3.34).

Dabei kommt es im Beispiel in Abbildung 3.34 zu einem doppelten Aufruf von `method2` in der Automatenklasse, wenn die Oberklasse ebenfalls instrumentiert ist, einmal durch die zu testende Klasse selbst und einmal durch den Aufruf der korrespondierenden Methode der Oberklasse durch `super.method2(parameter2);`. Dieses kann zur Überprüfung von Inkompatibilitäten in der Zusammenarbeit von Methoden der Ober- und Unterklasse genutzt werden.

Instrumentiert werden nur Methoden, die nach außen hin sichtbar sind.

Der durch die vorhergehend vorgestellte Instrumentierung aufgerufene Automat wird im folgenden vorgestellt. Ein Beispiel für eine Automatenklasse findet sich in Abbildung 3.35. In den Methoden werden Ausgangszustand und Bedingung für den Übergang in einen neuen Zustand gemäß der in Kapitel 2.4.4 vorgestellten Implementierung von Marick [Mar95] durch if-Anweisungen realisiert. Die Protokollierung erfolgt durch die Methode `report` (im Beispiel aufgerufen in `method1`, Abbildung 3.35).

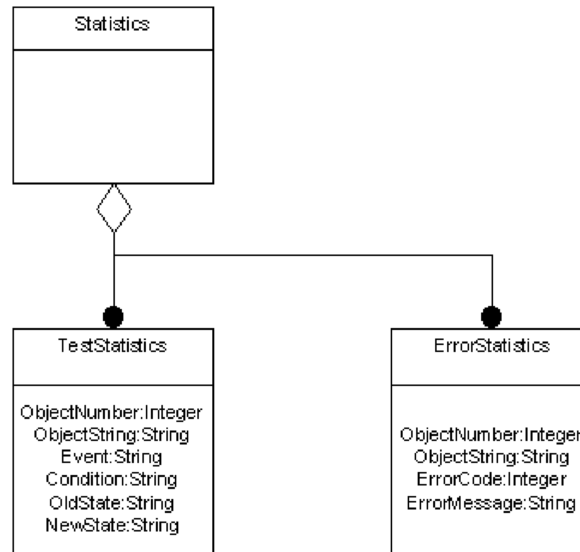
Die Methode `checkState` vergleicht den aktuellen Zustand (zurückgegeben durch `getActuellObjectState`) mit dem in der Instanzvariable State gespeicherten Zustand, der durch den Aufruf der Methoden der Automatenklasse geändert wird. In `getActuellObjectState` wird der Zustand des zu testenden Objekts durch if-Anweisungen, in denen die Zustandsdefinitionen des Benutzers als Bedingung dienen, ermittelt.

Auftretende Fehler werden durch die Methode `error` protokolliert (in Abbildung 3.35 aufgerufen in Methode `checkState`).

```
....

public class MyClassAutomaton
{
   private static int Counter;
   private MyClass obj;
   private int ObjectNumber;
   ....

   public MyClassAutomaton(MyClass myObj, type1 parameter)
   {    obj = myObj;
        State = "StartState";
        ObjectNumber = getCounter();
        ....
   }

   public void method1()
   {    if (State.equals("StartState"))
        {  if ("condition1")
           {    changeState("State2");
                report("StartState", "State2", "method1", "condition1");
           }
           else ...
        }
        else ...
   }

   public void method2(type2 parameter2)
   {    ....
   }

   public void checkState()
   {    if (!State.equals(getActuellObjectState()))
        {  error(AutomatonError.STATE_FAULT);
        }
   }
   ....
}
```

Abbildung~3.35: Beispiel für einen generierten Automaten

3.3.7 Statistikverwaltung



Abbildung~3.36: Objektmodell Statistikverwaltung

Zur Verwaltung der Teststatistik, die während der Ausführung des zu testenden Programms in einer Datei protokolliert wird, wurde die in Abbildung 3.36 dargestellte Objektstruktur eingeführt. Die Statistik eines Automaten unterscheidet zwischen der Ausführung einer Methode (dem Eintreffen eines Ereignisses) und dem Auftreten eines Fehlers. Das Eintreffen eines Ereignisses wird in der Klasse `TestStatistics` gespeichert, das Auftreten eines Fehlers in der Klasse `ErrorStatistics`.

Da alle Objekte einer Automatenklasse zur Laufzeit auf dieselbe Datei zugreifen, ist eine Unterscheidung der Objekte voneinander nötig. Dazu wird bei der Erzeugung eines Objekts der Automatenklasse an jedes Objekt eine Nummer über eine Klassenmethode vergeben, die es auch nach Erlöschen seines Lebenszyklus eindeutig identifizierbar macht. Die Aufzeichnung aller Meldungen von Objekten einer Automatenklasse in einer Datei hat den Vorteil, daß die zeitliche Reihenfolge überwacht werden kann, in der die Automaten ihre Zustände ändern und Fehler aufgetreten sind. Problematisch ist aber die möglicherweise zeitliche Trennung vom Eintreffen eines Ereignisses bis zur Aufdeckung des Fehlers, in der Meldungen anderer Objekte in die Datei eingetragen wurden. Deshalb meldet jedes Objekt auch bei der Anzeige eines Fehlers seine Objektnummer. Diese Trennung bewirkt auch die Trennung der Statistik in zwei Klassen.

Ein Objekt der Klasse `Statistics` besteht aus beliebig vielen Objekten der Klassen `TestStatistics` und `ErrorStatistics`, die in der Reihenfolge

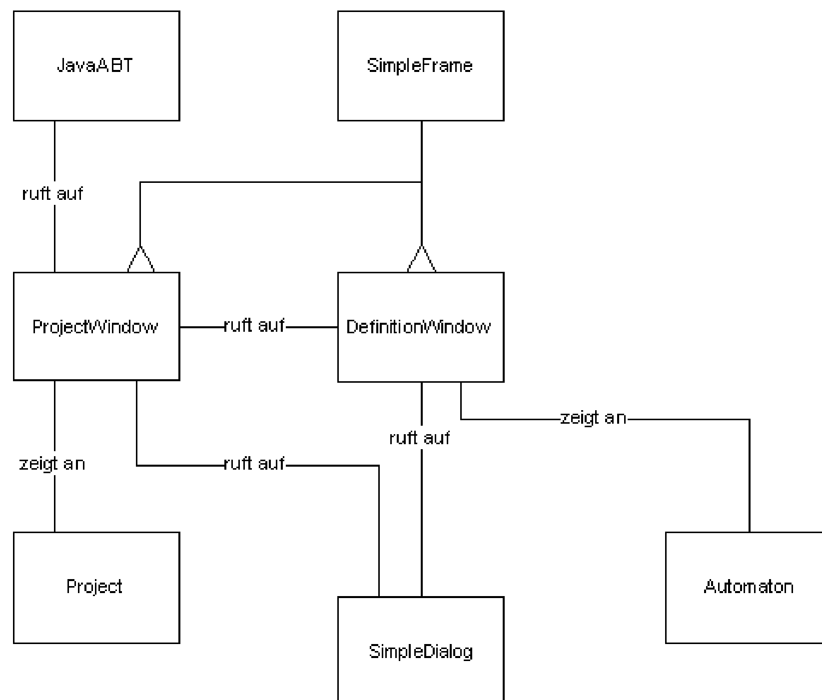
ihres zeitlichen Auftretens verwaltet werden (Abbildung 3.36). Beide besitzen als Attribute die Nummer des Objekts (`ObjectNumber`) und einen `ObjectString`, die String-Repräsentation des getesteten Objekts. Dieser `ObjectString` wird durch die Methode `toString`, die allen JAVA-Klassen gemein ist, erzeugt. Für eine bessere Lesbarkeit der String-Repräsentation sollte der Benutzer von JavaABT die Methode `toString` überschreiben, es ist jedoch nicht Voraussetzung für die Verwendung von JavaABT.

Die Klasse `TestStatistics` verwaltet das eingetroffene Ereignis (`Event`), die dabei erfüllte Bedingung (`Condition`) und alten und neuen Zustand (`OldState`, `NewState`) des getesteten Objekts.

In der Klasse `ErrorStatistics` werden Informationen über den aufgetretenen Fehler durch die Attribute `ErrorCode` und `ErrorMessage` gespeichert.

Die Klasse Statistics besitzt Methoden zur Überdeckungs- und Fehlermessung. So gibt es die Methoden `stateCoverage`, `constructorTransitionCoverage` und `methodTransitionCoverage` zur Angabe der erreichten Überdeckung und `numFailures` zur Angabe der Anzahl aufgetretener Fehler.

3.3.8 Fensterklassen



Abbildung~3.37: Objektdiagramm Fensterklassen

JavaABT besitzt zwei Hauptfenster, das Projekt- und das Definitionsfenster (siehe auch Benutzerschnittstellenbeschreibung in Kapitel 3.2.2). Jedes Fenster wurde durch eine Klasse realisiert, `ProjectWindow` und `DefinitionWindow` (Abbildung 3.37). Aufgerufen wird das Projektfenster durch die Klasse JavaABT, die die Methode `main` zum Start des Programms enthält. Das Projetkfenster selbst ruft bei Bedarf das Definitionsfenster auf. Beide Fenster erben von der Klasse `SimpleFrame`, die Eigenschaften beider Klasse wie das Schließen des Fenster auf bestimmte Ereignisse implementiert. Beide Fenster rufen Dialoge auf, die alle von der Klasse `SimpleDialog` erben. Zu den Dialogfenster zählen die Fenster zur Definition einzelner Zustände und Transitionen, Fenster, die auf Fehler hinweisen und Fenster, die Fragen an den Benutzer stellen.

Das Projektfenster zeigt das Projekt an, das gerade geöffnet ist, realisiert durch die Klasse `Project`. Das Definitionsfenster zeigt die bisher getätigten Definitionen für den Automaten (`Automaton`) an und erlaubt Änderungen der Definition.

3.3.9 Internationalisierung und Oberflächenanpassung

JavaAbt nutzt die Möglichkeiten der Programmiersprache JAVA, Sprache und Oberfläche an die Bedürfnisse des Benutzers anzupassen.

JAVA bietet einen Mechanismus zur unabhängigen Implementierung aller sprachabhängigen Bestandteile eines Programms, wie Fenstertitel, Beschriftungen, Erklärungen und Hilfedateien, genannt Internationalisierung. Dabei wird für jede Sprache ein sogenanntes *RessourceBundle* angelegt, eine Art Datenbank, in der alle sprachabhängigen Bestandteile abgelegt sind. In der Implementierung werden dann nur noch Platzhalter für diese Bestandteile angegeben. Zu Beginn der Ausführung des Programms sollte eine Sprache angegeben werden, die auch durch Optionen festgelegt werden kann.

JavaABT besitzt in der vorliegenden Version nur ein *RessourceBundle* für die englische Sprache, da Englisch die Standardsprache der Informatik darstellt. Durch die Implementierung der Internationalität ist jedoch die Einführung einer neuen Sprache lediglich von der Definition eines neuen *RessourceBundles* und der Implementierung einer entsprechenden Wahlmöglichkeit der Sprache abhängig. In Abbildung 3.38 findet sich ein Auszug aus dem englischen *RessourceBundle* für JavaABT. In `contents` sind die sprachabhängigen Bestandteilt definiert, wobei sich links der Schlüssel befindet und rechts der dazugehörige englische Begriff.

Benutzt werden kann das definierte RessourceBundle durch den Aufruf `getString(Schlüssel)`. ImBeispiel in Abbildung 3.39 wird der Titel des Projektfensters durch den Aufruf von `setTitle(Bundle.getString(''abt project''));` gesetzt. Bei `''abt project''` handelt es sich um den Schlüssel, der durch den Titel `''JavaABT Project''` ersetzt wird (Abbildung

```
public class AutomataBundle_en extends ListResourceBundle
{

   public Object[][] getContents()
   {
      return contents;
   }


   static final Object[][] contents =
   {

      // window titles
      {  "abt project",               "JavaABT - Project"},
      {  "abt definition",            "JavaABT - Definition"},
      {  "abt state definition",      "JavaABT - State Definition"},

       ......

   };

}
```

Abbildung~3.38: Englisches *RessourceBundle* für JavaABT

3.38). Das Bundle, welches aufgerufen werden soll, wird durch die Methode `getRessourceBundle` der Klasse `AutomataWindowDefaults` zurückgeliefert (Abbildung 3.39).

Die Methode getRessourceBundle der Klasse AutomataWindowDefaults findet sich in Abbildung 3.40. Hier wird als Umgebung fest die englische Sprache vorgegeben (`Locale locale = Locale.ENGLISH);`). Möglich wäre auch eine Wahl zwischen verschiedenen Sprachen. Dazu ist nur die Anlage eines neuen AutomataBundle nötig, in dem die englischen Begriffe durch die der anderen Sprache ersetzt werden müssen. Der eigentliche JavaABT-Programmcode bleibt gleich.

```
public ProjectWindow()
{  super();
   setSize(550,450);
   Bundle = AutomataWindowDefaults.getResourceBundle();
   setTitle(Bundle.getString("abt project"));

    ......
 }
```

Abbildung~3.39: Beispiel für die Verwendung von *RessourceBundles*

```
 public static ResourceBundle getResourceBundle()
 {  Locale locale = Locale.ENGLISH;
    ResourceBundle bundle =
     ResourceBundle.getBundle("automaton.AutomataBundle", locale);
    return bundle;
 }
```

Abbildung~3.40: Methode `getRessourceBundle`

```
public class JavaABT
{
   public static void main(String[] args)
   {  String systemLookAndFeel = UIManager.getSystemLookAndFeelClassName();
      try
      {  UIManager.setLookAndFeel(systemLookAndFeel);
      }
      catch(Exception e)
      {  // do nothing, cause UIManager.getSystemLookAndFeelClassName()
         // always returns a valid look and feel
      }

       ......

   }
}
```

Abbildung~3.41: Implementierung der Anpassung des Look&Feel

Die Anpassung des *Look&Feel* der Oberfläche von JavaABT wird direkt nach dem Start des Programms vorgenommen (siehe Abbildung 3.41). In der Methode `main` der Klasse `JavaABT` wird zuerst duch den Aufruf von `UIManager.getSystemLookAndFeelClassName` das aktuelle *Look&Feel* der Plattform ermittelt. Ist für die Plattform kein *Look&Feel* verfügbar, so wird das Standard-*Look&Feel* zurückgeliefert. Anschließend wird durch den Aufruf von `UIManager.setLookAndFeel` das ermittelte *Look&Feel* zugewiesen.

3.4 Implementierung

Die Abschnitte dieses Kapitels beschreiben Besonderheiten der Implementierung von JavaABT. Dabei wird zunächst die Wahl der Programmiersprache begründet (Abschnitt 3.4.1), anschließend bei der Implementierung aufgetretene Probleme behandelt (Abschnitt 3.4.2) und schließlich ein Vergleich von Entwurf und Implementierung vorgenommen (Abschnitt 3.4.3).

3.4.1 Wahl der Programmiersprache

Als Programmiersprache zur Implementierung wurde JAVA gewählt. Dafür gibt es mehrere Gründe.

Der erste, wohl einleuchtendste ist, daß der Entwurf im vorangegangenen Kapitel gezeigt hat, daß eine gründliche Kenntnis der Programmiersprache JAVA nötig ist und der Zugriff auf JAVA-Klassen nicht beschränkt sein darf. Außerdem müssen zu testendes und Automatenobjekt einfachen Zugriff aufeinander haben. Da liegt es nahe, dieselbe Programmiersprache einzusetzen, in der die zu testende Klasse bereits vorliegt. JAVA bietet gute Möglichkeiten, ein entwickeltes Prinzip exemplarisch für alle anderen objektorientierten Sprachen zu implementieren, obwohl sie durch fehlende Mehrfachvererbung gegenüber Programmiersprachen wie Eiffel eingeschränkt ist (was jedoch auch eine Stärke von JAVA ist).

JAVA hat darüber hinaus den Vorteil der Plattformunabhängigkeit, so daß eine Einschränkung der Einsatzfähigkeit weder durch Betriebssystem noch durch das Vorhandensein einer bestimmten Entwicklungsumgebung nicht gegeben ist. Jeder Entwickler von JAVA-Programmen kann das System einsetzen, da er die für die Entwicklung nötige Entwicklungsumgebung bereits besitzt. Leider ist JAVA noch eine sehr junge Sprache, so daß Probleme (siehe Abschnitt 3.4.2) nicht auszuschließen sind.

Ein nicht zu übersehender Punkt, der für die Verwendung von JAVA spricht, ist die durch die Beschäftigung mit der Struktur von JAVA erworbene Kompetenz des Entwicklers des Systems für die Programmiersprache JAVA.

3.4.2 Probleme bei der Implementierung

Wie bereits in Abschnitt 3.4.1 angedeutet, reicht die Geschichte der Programmiersprache nicht allzu weit in die Vergangenheit zurück. Dies birgt die üblichen Probleme eines recht jungen Softwaresystems in sich. Viele Fehler sind noch nicht behoben oder treten neu auf.

Die programmierte Oberfläche wirkt nicht wie ein Ganzes, es lassen sich im Gegenteil alle Elemente der Oberfläche eindeutig identifizieren, da beim Aufbau der Fenster Komponenten sichtbar übereinander gelegt werden.

Da JAVA in der Regel (ohne JIT-Compiler[15]) als Byte-Code ausgeführt wird, ist das gesamte System sehr langsam und könnte in manchen Fällen die Geduld des Anwenders überfordern.

Die verwendeten Swing-Klassen[16] sind noch nicht eingefroren, daß heißt,

[15] Just-in-Time-Compiler, erzeugt aus JAVA-Byte-Code zur Laufzeit ausführbaren Code.

[16] Die Swing-Klassen sind Teil der *JAVA Foundation Classes*, einer Erweiterung der Standard-JAVA-Klassen, die in der nächsten JAVA-Version (1.2) in diese integriert werden sollen.

daß nicht nur Implementierung, sondern auch Schnittstellen sich noch ändern können, es kommt sogar vor, daß ganze Klassen in der nächsten Version der Swing-Klassen nicht mehr vorhanden sind.

Für JavaABT wurden die Swing-Klassen in der Version 1.0 verwendet. Durch Neukompilierung des Systems sowohl mit der vorangegangenen Version als auch mit nachfolgenden Versionen der Swing-Klassen wurden Inkompatibilitäten festgestellt, bei Verwendung anderer Versionen der Swing-Klassen als der Version 1.0 ist eine Anpassung von JavaABT nötig.

Auch enthalten die Swing-Klassen einige Fehler, zum Beispiel, daß Listen in einem Fenster mit Tabs, wie sie im Definitionsfenster (Kapitel 3.2.2.2) benutzt werden, nicht sofort aktualisiert werden. Um ein Arbeiten mit JavaABT zu ermöglichen, wurde das Problem vom Entwickler durch einen Trick übergangen.

Insgesamt wirkt JAVA ein wenig unhandlich, besonders, was die Implementierung einer grafischen Oberfläche betrifft, da GUI-Builder erst langsam auf den Markt gebracht werden.

3.4.3 Differenzen zwischen Entwurf und Implementierung

Dieser Abschnitt beschreibt die Abweichungen vom Entwurf, die bei der Implementierung aufgetreten sind.

Die statistische Auswertung ist bisher nur rudimentär vorhanden. Sie berücksichtigt noch keine Einzelauswertung für ein Objekt, einen Zustand oder den Aufruf einer Methode. Auch werden die Automaten vor der Generierung des Javacodes noch nicht auf fehlerhafte Definitionen untersucht.

Die Instrumentierung durch den in [Pos97] vorgestellten Instrumentierer wurde bisher nicht realisiert. Dazu müßte dieser in entsprechender Weise angepaßt werden, um die nötigen Aufrufe des Automaten in die zu testende Klasse hinein zu instrumentieren. Deshalb ist die Instrumentierung der zu testenden Klasse sowie die Eingabe der Methodenparameter noch manuell vorzunehmen.

Einige Teile des Systems, wie der JavaCodeGenerator und die Implementation der Vererbung, sind noch ein wenig instabil. Sonst wurde der vorliegende Entwurf möglichst vollständig umgesetzt. Gravierende Änderungen des Entwurfs wurden bei der Implementierung nicht vorgenommen.

Kapitel 4

Zusammenfassung und Ausblick

Das im vorangegangenen Kapitel vorgestellte System unterstützt zustandsbasierte Testverfahren durch die Protokollierung von Zuständen, Ereignissen und Zustandsübergängen eines getesteten Objekts. Dabei wird dem Benutzer der Umgang mit dem System durch Automatisierung vieler Aufgaben so weit wie möglich erleichtert. Aus einer bestehenden Spezifikation in Form eines Zustandsautomaten wird durch Generierung einer Klasse, die diesen Zustandsautomaten implementiert, ein Werkzeug, um Spezifikation und Implementation einer Klasse gegeneinander zu testen.

Dabei könnten einige Verbesserungen des Systems zur Förderung der Benutzbarkeit beitragen. So könnte eine Minimierung der Fenster auf ein einziges Fenster für Projektdarstellung und Zustandsautomatendefinition eine bessere Übersichtlichkeit über bereits getestete Klassen und spezifizierte Zustandsautomaten bieten. Eine grafische Darstellung der Zustandsautomaten in Form eines Zustandsdiagramms könnte die Überprüfung der richtigen Eingabe der Daten des Zustandsautomaten für den Benutzer erleichtern.

Die Unterstützung anderer Definitionsformen von Zustandsautomaten und die vollständige Implementierung der Statecharts von Harel [Har87] könnte ein weiteres Ziel einer möglichen Weiterentwicklung sein. Die Teilung von Zuständen und die Definition nebenläufiger Zustandsdiagramme könnte auch auf nicht vererbte Zustandsautomaten anwendbar sein.

Weiterführende Entwicklungen könnten die Integration des Systems in andere Testsysteme wie dem in [Bei98] vorgestellten Werkzeug für den Integrationstest, aus dem sich Klassen als Projekt importieren lassen könnten, oder dem Import von Zustandsautomaten aus anderen Werkzeugen, zum Beispiel Werkzeugen zur Unterstützung der Spezifikation von Softwaresystemen, betreffen. Es besteht die Möglichkeit, die implementierten Zustandsautomaten auch nach der Testphase zu nutzen. Dazu müßte man

ihre Ausgaben an- und ausschalten können. Sie könnten dann Wartung und Pflege eines Systems erleichtern.

Mit dem beschriebenen System wurde ein Ansatz zur programmtechnischen Umsetzung zustandsbasierter Testverfahren für objektorientierte Software vorgestellt. Insbesondere die Unterstützung der Vererbung von Zustandsautomaten berücksichtigt die besonderen Eigenschaften objektorientierter Software. Der dabei auf typkonforme Vererbung gelegte Schwerpunkt unterstreicht die Besonderheit dieser Vererbungsform, da sie die Sicht von Objekten als Implementationen abstrakter Datentypen berücksichtigt und den Testaufwand durch Benutzung und Erweiterung des Zustandsautomaten der Oberklasse reduziert.

Literaturverzeichnis

[Bal96] Helmut Balzert. *Lehrbuch der Software-Technik Band 1*. Spektrum Akademischer Verlag, 1996.

[Bei98] Eduard Beier. Entwicklung eines Werkzeugs zur Unterstützung von Integrations- und Regressionstests von JAVA-Klassen. Diplomarbeit, Technische Universität Berlin, Fachbereich 13 - Informatik, Institut für Kommunikations- und Softwaretechnik, Fachgebiet Softwaretechnik, Berlin, 1998.

[Ber93] Edward V. Berard. Issues in the Testing of Object-Oriented Software Engeneering. In *Essays on Object-Oriented Software Engeneering*, chapter 15, pages 257–270. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1993.

[Bin94] Robert V. Binder. Design for Testability in Object-Oriented Systems. *Communications of the ACM*, 37(9):87–101, Sept. 1994.

[CHB92] Derek Coleman, Fiona Hayes, and Stephen Bear. Introducing Objectcharts or How to Use Statecharts in Object-Oriented Design. *IEEE Transactions on Software Engineering*, 18(1):9–18, Jan. 1992.

[Cho78] Tsun S. Chow. Testing Software Design Modeled by Finite-State Machines. *IEEE Transactions on Software Engineering*, SE-4(3):178–187, Mai 1978.

[DF91] Roong-Ko Doong and Phyllis G. Frankl. Case Studies on Testing Object-Oriented Programs. In *Proceedings of the Symposium on Testing, Analysis and Verification (TAV'4)*, 1991.

[GMH81] J. Gannon, P. McMullin, and R. Hamlet. Data-Abstraction Implementation, Specification and Testing. *ACM Transactions on Programming Languages and Systems*, 3(3):211–233, Juli 1981.

[Har87] David Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, (8):231–274, 1987.

[HG95] David Harel and Eran Gery. Executable Object Modeling with Statecharts. Technical report, The Weizman Institut of Science, Rehovot, Israel, 1995.

[HP94] Hans-Martin Hörcher and Jan Peleska. The Role of Formal Specifications in Software Test. In *Symposium Industrial Benefit of Formal Methods (FME'94)*, Okt. 1994.

[(Hr93] Karl-Heinz Rödiger (Hrsg.). *Software-Ergonomie '93*. German Chapter of the ACM - Berichte 39. B.G. Teubner, Stuttgart, 1993.

[HS94] Daniel Hoffmann and Paul Strooper. Graph-based Class Testing. *The Australian Computer Journal*, 26(4):158–163, Nov. 1994.

[HS96] M. Hughes and D. Stotts. Daistish: Systematic Algebraic Testing for OO Programs in the Presence of Side-effects. In *Proceedings of the 1996 International Symposium on Software Testing and Analysis (ISSTA)*, pages 53–61, 1996.

[KT94a] Shekhar Kirani and W.T. Tsai. Method Sequence Specification and Verification of Classes. *Journal of Object-Oriented Programming*, pages 28–37, Okt. 1994.

[KT94b] Shekhar Kirani and W.T. Tsai. Specification and Verification of Object-Oriented Programs. Master's thesis, University of Minnesota, MN 55455, Dez. 1994.

[Lig90] Peter Liggesmeyer. *Modultest und Modulverifikation - State of the Art*. BI-Wissenschaftsverlag, Mannheim, Wien, Zürich, 1990.

[Lig95] Peter Liggesmeyer. Ein Überblick über objektorientiertes Testen. In *8. Treffen des Arbeitskreises 'Testen, Analysieren und Verifizieren'*, Hamburg, Okt. 1995. MAZ.

[LW94] Barbara Liskov and Jeanette M. Wing. A Behavioral Notion of Subtyping. *ACM Transactions on Programming Languages and Systems*, 1994.

[Mar95] Brian Marick. *The Craft of Software Testing (Subsystem Testing)*. Prentice Hall, 1995.

[MD93] John D. McGregor and Douglas M. Dyer. A Note on Inheritance and State Machines. *Software Engineering Notes*, 18(4):61–69, Okt. 1993.

[Mey88] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1988.

[PBC93] Allen S. Parrish, Richard B. Borie, and David W. Cordes. Automated Flow Graph-Based Testing of Object-Oriented Software Modules. *Journal of Systems and Software*, pages 95–109, 1993.

[PK90] Dewayne E. Perry and Gail E. Kaiser. Adequate Testing and Object-Oriented Programming. *Journal of Object-Oriented Programming*, pages 13–19, Jan./Feb. 1990.

[Pos97] Frank Posadny. Entwicklung eines Systems von Protokollierungsklassen für den objektorientierten Intergrationstest in der Programmiersprache JAVA. Diplomarbeit, Technische Universität Berlin, Fachbereich 13 - Informatik, Institut für Kommunikations- und Softwaretechnik, Fachgebiet Softwaretechnik, Berlin, Nov. 1997.

[Pro95] W. Prokscha. Kategoriepartionierung++ (KP++) - Ein Testverfahren für OO-Software auf Klassenebene. In *8.Treffen des Arbeitskreises 'Testen, Analysieren und Verifizieren'*, Hamburg, Okt. 1995. MAZ.

[RBP+91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[RBP+93] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Objektorientiertes Modellieren und Entwerfen*. Carl Hanser Verlag, München Wien, 1993.

[Rüp97] Peter Rüppel. *Ein generisches Testwerkzeug für den objektorientierten Softwaretest - neue Möglichkeiten zur Testunterstützung*. Dissertation, Technische Universität Berlin, Fachbereich 13 - Informatik, Institut für Kommunikations- und Softwaretechnik, Fachgebiet Softwaretechnik, Berlin, 1997.

[San94] Aamod Sane. Object-Oriented State Machines: Subclassing, Composition, Delegation and Genericity. Technical report, University of Illinois at Urbana-Champaign, Department of Computer Science, Nov. 1994.

[Sne95] H.M. Sneed. Objektorientiertes Testen. *Informatik Spektrum*, (18):6–12, 19995.

[SR92] M. Smith and D.J. Robson. A Framework for Testing Object-Oriented Programs. *Journal of Object-Oriented Programming*, pages 45–53, Juni 1992.

[TR95] Christopher D. Turner and David J. Robson. A State-Based Approach to the Testing of Class-Based Programs. *Software - Concepts and Tools*, (16):106–112, 195.

[Zuc96] Lin Zucconi. Building Testable Software. *Software Engineering Notes*, 21(5):51–55, Sept. 1996.