Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Nico Manske

Eine einfache, schnelle und speicherschonende Technologie zur Implementation des Zustands-Entwurfsmusters.

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Nico Manske

Eine einfache, schnelle und speicherschonende Technologie zur Implementation des Zustands-Entwurfsmusters.

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang Technische Informatik
am Studiendepartment Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer : Prof. Dr. rer. nat. Stephan Pareigis
Zweitgutachter : Prof. Dr. rer. nat. Reinhard Baran

Abgegeben am 17. März 2006

Nico Manske

Thema der Bachelorarbeit

Eine einfache, schnelle und speicherschonende Technologie zur Implementation des Zustands-Entwurfsmusters.

Stichworte

Zustandsautomat, Zustands-Entwurfsmuster, C++, Operator `Placement-new`

Kurzzusammenfassung

Zustandsautomaten bieten eine sehr effiziente Möglichkeit das dynamische Verhalten von Systemen und Komponenten zu beschreiben. Zur Implementierung von Zustandsautomaten wird häufig das von Gamma et al 1994 vorgeschlagene Muster genutzt. Ein Problem dieses Musters ist das Bekanntmachen der Zustände untereinander, um selbstständig in den Folgezustand wechseln zu können. Ein weiteres Problem ist die notwendige Haltung aller vorhandenen Zustände im Speicher. In dieser Arbeit wird ein neuartiges Prinzip zum Wechseln des Zustandes mit Hilfe des Operators `Placement-new` analysiert und angewandt, um eine Alternative zur Verfügung zu stellen. Die Alternative ist einfach, schnell und speicherschonend zugleich.

Nico Manske

Title of the paper

A State Design Pattern Implementation that is simple fast and memory sparing.

Keywords

State Machine, State-Design Pattern, C++, Operator `Placement-new`

Abstract

State machines provide a powerful way to describe dynamic behavior of systems and components. Often the suggested Pattern of Gamma et al from 1994 is used for implementing state machines. One Problem of that Pattern is the publication of the state among each other for changing to the next state by its one. An other problem in that context is the bearing of all existing states in the memory. This thesis analyses and uses a novel principle for changing states with the help of the operator `Placement-new`. It provides an alternative Pattern for the Implementation of state machines. That alternative pattern is simple, fast and memory sparing at the same time.

Inhaltsverzeichnis

Inhaltsverzeichnis ... **I**

Abbildungsverzeichnis ... **III**

Tabellenverzeichnis ... **IV**

Abkürzungsverzeichnis ... **V**

Danksagung ... **VI**

1 Einleitung ... **1**

1.1 Motivation ... 1

1.2 Zielsetzung ... 2

1.3 Inhalt der Arbeit ... 2

2 Grundlagen ... **4**

2.1 Zustandsautomaten ... 4

2.1.1 Komponenten ... 5

2.1.2 Flache endliche Zustandsautomaten (FSM) ... 6

2.1.3 Hierarchische Zustandsautomaten (HSM) ... 6

2.2 Zustands-Entwurfsmuster ... 8

2.2.1 Prozeduraler Ansatz ... 8

2.2.2 Objektorientierter Ansatz ... 9

2.2.3 Diverse Abwandlungen und Erweiterungen ... 9

2.3 C++ Grundlagen ... 10

2.3.1 Operator `Placement-new` ... 10

2.3.2 Templates ... 11

2.4 XML ... 13

2.4.1 Document Object Model (DOM) ... 13

2.4.2 Xerces XML Parser ... 14

3 Analyse existierender Zustands-Entwurfsmuster ... **15**

3.1 Geschachtelte `switch-case` Anweisungen (Nested switch Statement) ... 16

3.2 Zustandstabelle (State Table) ... 18

3.3 State Design Pattern (SDP) von Gamma et al ... 21

3.4 Optimaler endlicher Zustandsautomat (optimal FSM) ... 24

3.5 Erweiterungen und Spezialisierungen ... 27

3.6 Muster für hierarchische Zustandsautomaten (HSM) ... 28

4 Anforderungen an ein State Pattern ... **29**

4.1 Funktionale Anforderungen ... 29

4.2 Strukturelle Anforderungen (Modifikationen) ... 32

4.3 Leistungsanforderungen 32
5 Design – Implementation 34
5.1 Grundprinzip der Technologie 34
5.2 Flacher endlicher Zustandsautomat (FSM) 36
5.2.1 Struktur 38
5.2.2 Reaktion – Zustandsübergang 39
5.2.3 Datenhaltung mit Template-Klassen 40
5.3 Ansatz eines hierarchischen Zustandsautomaten (HSM) 42
5.3.1 Struktur 43
5.3.2 Zustandsübergänge 45
5.3.3 History-Funktionalität (flache History) 47
5.4 HSMXml2Pattern Generator für hierarchische Automaten 50
5.4.1 XML-Datei 50
5.4.2 Struktur 53
5.4.3 Verarbeitung der Daten 54
5.4.4 Codeerzeugung 54
5.4.5 Test 56
5.4.6 Erweiterungen 57
6 Auswertung und Vergleich mit analysierten Techniken 58
6.1 Struktur 58
6.2 Leistung 59
6.2.1 Laufzeit 60
6.2.2 Speicher 61
6.3 Funktionalität 63
6.4 Wiederverwendbarkeit 64
6.4.1 Quellcodegenerator 65
7 Fazit und Ausblick 66
Glossar 67
Literaturverzeichnis 68
Anhang A: 71
Versicherung über Selbstständigkeit 72

Abbildungsverzeichnis

Abbildung 2-1: Beispiel Template-Klasse ... 12
Abbildung 2-2: DOM-Struktur ... 14
Abbildung 3-1: Zustandsautomat zur Erklärung ... 15
Abbildung 3-2: Codeausschnitt Zustände, Ereignisse, Funktionen (nested switch) ... 16
Abbildung 3-3: Codeausschnitt `dispatch()`(nested switch) ... 17
Abbildung 3-4: Quellcodeausschnitt generischer Teil (Zustandstabelle) ... 19
Abbildung 3-5: Quellcodeausschnitt anwendungsspezifischer Teil (Zustandstabelle) .. 20
Abbildung 3-6: Quellcodeausschnitt Initialisierung der Tabelle (Zustandstabelle) ... 21
Abbildung 3-7: Klassendiagramm State Design Pattern ... 22
Abbildung 3-8: Quellcodeausschnitt Kontextklasse (GoF) ... 23
Abbildung 3-9: Quellcodeausschnitt Zustandswechsel (GoF) ... 23
Abbildung 3-10: Klassendiagramm optimal FSM ... 25
Abbildung 3-11: Quellcodeausschnitt optimaler FSM ... 25
Abbildung 3-12: Quellcodeausschnitt Ereignisverarbeitung (optimal FSM) ... 26
Abbildung 5-1: Beispiel Zustandswechsel mit Operator Placement-new ... 35
Abbildung 5-2: virtuelle Funktionstabellen ... 36
Abbildung 5-3: Zustandsautomat für ersten Entwicklungsschritt ... 37
Abbildung 5-4: Klassendiagramm für den konkreten flachen endlichen Automaten ... 38
Abbildung 5-5: Beispiel Einbinden einer TPP-Datei ... 39
Abbildung 5-6: Quellcodeausschnitt Delegation Ereignis Kontext zu Zustand ... 39
Abbildung 5-7: Quellcodeausschnitt Aktion + Zustandswechsel im FSM ... 40
Abbildung 5-8: Quellcodeausschnitt Anwendung Template für Datenzugriff ... 41
Abbildung 5-9: Quellcodeausschnitt Definition Zeiger mit Vorlagenparameter ... 41
Abbildung 5-10: Quellcodeausschnitt Instantiierung mit Vorlagenparameter ... 41
Abbildung 5-11: Hierarchischer Zustandsautomat für zweiten Entwicklungsschritt ... 43
Abbildung 5-12: Quellcodeausschnitt Reaktion im Hierarchischen Automaten ... 44
Abbildung 5-13: Klassendiagramm für konkreten hierarchischen Automaten ... 45
Abbildung 5-14: Quellcodeausschnitt Initialfunktion ... 46
Abbildung 5-15: Quellcodeausschnitt History-Funktion ... 48
Abbildung 5-16: Quellcodeausschnitt Zustand speichern in Kontextklasse ... 48
Abbildung 5-17: Debug-Ausschnitt vtbl und history ... 49
Abbildung 5-18: Gesamtstruktur XML-Dokument ... 51
Abbildung 5-19: Zustandsdefinition in der XML-Datei ... 52
Abbildung 5-20: Ereignisdefinition in der XML-Datei ... 52
Abbildung 5-21: Transitions-Definition in der XML-Datei ... 53
Abbildung 5-22: Klassendiagramm Quellcodegenerator ... 54
Abbildung 6-1: Quellcodeausschnitt Struktur Tran (Zustandstabelle) ... 62
Abbildung 6-2: Diagramm Speicherbedarf ... 63

Tabellenverzeichnis

Tabelle 3-1: Beispiel Zustandstabelle 18

Abkürzungsverzeichnis

API	Application Programming Interface
DLL	Dynamic Link Library
DOM	Document Object Model
DTD	Document Type Definition
FSM	Finite State Machine
GoF	Gang of Four
HSM	Hierarchical State Machine
IDL	Interface Description Language
OMG	Object Management Group
SDP	State Design Pattern
UML	Unified Modelling Language
VTBL	Virtual Function Table
XML	Extensible Markup Language

Danksagung

In erster Linie danke ich meinen Eltern, ohne deren finanzielle Unterstützung das Studium schwieriger zu absolvieren gewesen wäre. Außerdem danke ich ihnen für die aufbauenden und unterstützenden Worte während des gesamten Studiums, insbesondere für die im letzten Studienabschnitt.

Ich danke auch meiner Lebensgefährtin Wiebke, weil sie mich immer wieder dazu brachte weiter zu arbeiten und mir außerdem hilfreiche Tipps für die äußere Form der Arbeit gab.

Besonderen Dank richte ich an meinen betreuenden Professor Stephan Pareigis, der mir dieses interessante Thema zur Verfügung stellte und während unserer Gespräche immer wieder inspirierende Ideen und Vorschläge äußerte.

1 Einleitung

Zustandsautomaten werden oft zur Spezifikation und Implementation von Softwaresystemen genutzt. Außerdem werden sie in reaktiven Systemen genutzt, um das dynamische Verhalten von Objekten darzustellen. Dabei wird das System mit wachsender Größe immer komplexer, was die Realisierung hinsichtlich der Umsetzung in eine Programmiersprache weiter erschwert. Wenn es gilt, einen Zustandsautomaten zu realisieren, sind Entwurfsmuster sehr hilfreich. Durch solche Muster ist es möglich, den gewünschten Automaten nach einem bestimmten Prinzip zu implementieren. Diese Möglichkeit spart Zeit und ist deshalb sehr effizient.

Softwarewerkzeuge zur Generierung alternativer Zustandsautomaten sind oft zur Realisierung spezieller Automaten zu generisch. Aus diesem Grund wird immer wieder auf die bestehenden Zustands-Entwurfsmuster zurückgegriffen. Wenn jedoch ein sehr spezielles Problem gelöst werden soll, bleibt vielen Entwicklern oftmals keine andere Möglichkeit, als ein bestehendes Muster anzupassen oder sogar ein eigenes zu entwickeln. Bei der Implementierung eines Zustandsmusters nach eigenen Vorstellungen besteht die wesentliche Herausforderung darin, effizienten Code zu schreiben, ohne die Wartbarkeit bezüglich der Struktur zu gefährden. Diese Punkte sind oft Gründe, weshalb die existierenden Muster den gewünschten Anforderungen nicht entsprechen.

In dieser Arbeit werden verschiedene Muster zur Implementation von Zustandsautomaten vorgestellt. Ausgehend von dem wohl bekanntesten und am meisten verwendeten Zustands-Entwurfsmuster von Gamma et al aus dem Buch „*Design Patterns: Elements of Object Oriented Software*“ [GoF95], wird ein neues Muster aufgebaut, um es mit den vorhandenen Mustern zu vergleichen. Dabei ist das Grundprinzip ein neuartiger Zustandswechsel unter der Verwendung des Operators `Placement-new`. Des Weiteren wird geprüft, ob die Anwendung der neuartigen Zustandswechsel Nachteile mit sich bringt, und ob das resultierende Muster alle Anforderungen eines Zustandsautomaten erfüllt.

1.1 Motivation

Das oft benutzte Zustandsmuster von Gamma et al, und die verschiedenen Ausprägungen dieses Musters haben einige Nachteile. So wird zum Beispiel die Kapselung der Klassen (Zustände) aufgebrochen. Ein weiterer Nachteil ist die Verwendung des Singelton-Musters für die Instantiierung der einzelnen Zustände. Hierbei werden alle vorhandenen Zustände komplett im Speicher gehalten, auch wenn sie nicht benutzt werden. Es stellt sich die Frage, ob dieses Muster unter Anwendung

neuer Techniken in irgendeiner Weise verbessert werden kann. Das betrifft sowohl die Leistung als auch die Struktur.

Mein Betreuer, Prof. Pareigis, hat eine Möglichkeit gefunden, die Zustandswechsel auf eine andere Art und Weise zu vollziehen als es bisher üblich war. Bei dieser Technik handelt es sich um die Verwendung des Operators `Placement-new` zur Realisierung der Zustandswechsel. Mit Hilfe dieser Technik soll unter anderem der Speicher geschont werden. Des Weiteren sollte eine Vereinfachung der Struktur stattfinden. Zur Erforschung der Verwendbarkeit neuer Techniken ist es notwendig einen kompletten Zustandsautomaten auf dieser Grundlage aufzubauen. Der Automat muss überdies ausgebaut werden, um die üblichen Funktionen von Zustandsautomaten zu integrieren. Bei erfolgreicher Umsetzung entsteht ein neues Muster für den Entwurf von Automaten.

Da ein Zustands-Entwurfsmuster, und somit die Realisierung eines Zustandsautomaten, gewisse Anforderungen erfüllen soll, gilt es zu prüfen, ob sich die neuartige Technik für diese Problematik eignet, oder ob sich Nachteile daraus ergeben können. Außerdem muss sichergestellt werden, dass diese Technik in dieser Form für diesen Zweck noch nicht angewandt wurde.

1.2 Zielsetzung

Das Ziel dieser Arbeit soll es sein, eine Auswahl der existierenden Zustandsmuster zu analysieren und übersichtlich darzustellen, damit der Leser einen Überblick über die vorhandenen Techniken bekommt. Die notwendigen Grundlagen, zum besseren Verständnis der Arbeit, werden dem Leser einführend vermittelt. Des Weiteren soll die neuartige Technologie, die bei den Zustandsübergängen angewendet wird, genutzt werden, um ein neues Zustandsmuster aufzubauen. Daneben ist zu prüfen, ob ein hierarchischer Automat umsetzbar ist. Zur Umsetzung des Musters müssen geeignete Modelle sowohl flacher als auch hierarchischer Automaten gefunden werden. Bei der Umsetzung ist darauf zu achten, dass die üblichen Anforderungen eines Zustandsmusters erfüllt werden. Außerdem soll das Muster so entworfen werden, dass ein Einsatz in einem Echtzeitsystem denkbar ist (möglichst wenig Speicher nutzen). Ein exemplarischer Quellcodegenerator zur Erstellung eines Automaten nach diesem Muster soll zeigen, dass es auch möglich ist, das Muster generisch anzuwenden. Abschließend soll eine vergleichende Diskussion durchgeführt werden, um die neue Idee gegenüber den vorhandenen Techniken zu bewerten.

1.3 Inhalt der Arbeit

Im Anschluss an diese Einleitung werden in Kapitel 2 einige Grundlagen erläutert, die zum besseren Verständnis dieser Arbeit beitragen beziehungsweise dazu notwendig

sind. Es handelt sich dabei unter anderem um Grundlagen aus dem Bereich der Zustandsautomaten. In diesem Abschnitt werden die Komponenten von Zustandsautomaten und die möglichen Architekturen der Automaten erläutert. Ein weiterer Punkt der nötigen Grundlagen ist die Einführung in Zustands-Entwurfsmuster und die Möglichkeiten der Realisierung solcher Muster. Um der Umsetzung der neuartigen Implementation und der Realisierung des Quellcodegenerators folgen zu können, wird auf spezielle Techniken der Programmiersprache C++ und der Markup-Language XML eingegangen.

Auf existierende Muster zur Realisierung von Zustandsautomaten wird in Kapitel 3 eingegangen. Da es viele verschiedene Muster als Lösung für dieses Problem gibt, findet bei der Analyse eine Einschränkung der Muster auf die meist verbreiteten Lösungen statt. Es wird aber auch kurz auf Erweiterungen und Spezialisierungen einiger Muster eingegangen.

Die Anforderungen, die eine Realisierung eines Zustands-Entwurfsmusters erfüllen muss, werden in Kapitel 4 dargestellt. Es erfolgt eine Aufteilung in funktionale, strukturelle und Leistungsanforderungen.

In Kapitel 5 wird die im Zuge dieser Arbeit entwickelte Lösung vorgestellt. Zunächst wird das Grundprinzip dieser Realisierung erklärt. Später erfolgt eine Beschreibung der gesamten Lösung, dabei wird die Lösung in zwei Schritte unterteilt. Der erste Schritt ist die Entwicklung eines flachen endlichen Zustandsautomaten. Im zweiten Schritt wird ein hierarchischer Automat, aufbauend auf dem ersten Schritt, entwickelt. Die Beschreibung des in dieser Arbeit entwickelten Quellcodegenerators schließt das Kapitel der Implementation ab.

Der Vergleich von existierenden Realisierungen mit der neu entwickelten Lösung erfolgt in Kapitel 6. Zum Vergleich werden verschiedene Kriterien verwendet. Darunter befinden sich Punkte wie Speicher und Laufzeit. Auch die Struktur und die Funktionalität spielen dabei eine große Rolle. Schließlich wird auch die Wiederverwendbarkeit diskutiert.

Eine Zusammenfassung der Ergebnisse und ein Ausblick, wie die Weiterentwicklung des neuen Zustands-Entwurfsmusters aussehen kann, erfolgt in Kapitel 7.

2 Grundlagen

In diesem Kapitel werden grundlegende Techniken, die in der Arbeit angewendet werden, erläutert. Dazu gehören unter anderem spezielle Techniken der Programmiersprache C++ und in der Verarbeitung von XML-Dokumenten. Erläutert werden ebenfalls Verhalten, Eigenschaften und Aufbau von Zustandsautomaten. Des Weiteren wird auf das Prinzip und die Ausbaumöglichkeiten von Zustandsmustern eingegangen.

2.1 Zustandsautomaten

David Harel schrieb in [Harel98 S.54], dass Zustandsautomaten eine Technik zur Beschreibung der Dynamik eines Systems darstellen, dabei befindet sich der Automat immer in einem bekannten Zustand. Das System reagiert auf auftretende Ereignisse. Die Art der Reaktion hängt von dem momentan aktiven Zustand und des Ereignisses ab. Mögliche Reaktionen sind das Ausführen von Aktionen, das Verändern von Variablen oder das Wechseln des Systems in einen anderen Zustand. So kann sich eine Lampe beispielsweise im Zustand AN oder AUS befinden, in diesem Fall würde ein Schalter die Ereignisse EINSCHALTEN und AUSSCHALTEN auslösen.

In der Regel werden Zustandsautomaten mit Hilfe von Modellen dargestellt, um die Funktionsweise klar und verständlich zu definieren. Um einen Zustandsautomaten zu implementieren, bedarf es eines solchen Modells. Solch ein Modell ist außerdem nötig, um einen Überblick über die Funktionsweise zu erhalten. Für diesen Zweck gibt es bestimmte Standards:

UML-Statecharts

Bei den UML Statecharts handelt es sich um einen sehr bekannten und oft verwandten Modellstandard. Der Standard wurde von der Object Management Group (OMG) in [OMG04] definiert. Er beruht auf den Statecharts von Harel aus [Harel87], welcher laut B. Douglass auch für den Entwurf von *erweiterten* Zustandsautomaten genutzt werden kann [Dougla99 S.310].

ROOMcharts

Ein anderer bekannter Standard für die Modellierung von Zustandsautomaten sind die ROOMcharts [SeGuWa94 S.231-248]. Auch dieser Standard baut auf den Ideen von Harel auf. Beide Ansätze unterscheiden sich nur minimal in der Notation und der Darstellung der Modelle.

In dieser Arbeit wird für alle folgenden Modelle der Standard der OMG (UML-Statecharts), zur Darstellung der Automatenmodelle in Form von Zustandsdiagrammen, genutzt.

Mit dem Statecharts Ansatz von Harel lassen sich hybride Zustandsautomaten modellieren. Es handelt sich hierbei um ein Automatenmodell, dass die Eigenschaften von Mealy- und Moore-Automaten verbindet. Mealy- und Moore-Automaten sind ähnliche Ansätze, die beide endliche Zustandsautomaten beschreiben. Mealy verbindet eine Aktion immer mit einem folgenden Zustandswechsel während Moore eine Aktion immer vom Zustand abhängig macht (unabhängig von dem Ereignis) [Samek02 S.28].

2.1.1 Komponenten

Ein kompletter Zustandsautomat besteht aus mehreren Komponenten, deren Zusammenspiel das Verhalten des Automaten ausmacht.

Zustände (engl.: States)

Die Zustände gehören zu den wichtigsten Komponenten eines Zustandsautomaten. Die Anzahl der Zustände in einem Automaten ist bekannt. Das Verhalten der verschiedenen Zustände ist unterschiedlich. So können unterschiedliche Zustände auf die gleichen Ereignisse durch die Ausführung verschiedener Aktionen reagieren. Gerade diese Eigenschaft zeichnet einen Zustandsautomaten aus. Zusammenfassend modelliert ein Zustand die Reaktionsbereitschaft des Systems auf die akzeptierten Ereignisse.

Ereignisse (engl.: Events)

Die bereits erwähnten Ereignisse, auch als Signale bezeichnet, können Reaktionen auslösen. Hier wird zwischen internen und externen Ereignissen unterschieden. Die internen Ereignisse werden vom System selbst ausgelöst, wobei die externen Ereignisse von außerhalb auf das System einwirken und somit das Verhalten beeinflussen können. Die Art der Verarbeitung der Ereignisse (Reaktionen) hängt von dem aktiven Zustand ab. Die Ereignisse können zudem Parameter enthalten, die dem System erlauben weitere Information über das Ereignis zu verarbeiten.

Wächter (engl.: Guards)

Die Informationen aus den Ereignissen können für die Wächter verwendet werden (nur in erweiterten Automaten). In einigen Zuständen kann es vorkommen, dass für ein Ereignis verschiedene Reaktionen vorgesehen sind. Die Wächter entscheiden, auf der Grundlage der Informationen der Ereignisse oder der vorhandenen Daten im System, welche Reaktion ausgeführt wird. Die Wächter können somit auch als Bedingungen betrachtet werden.

Aktion (engl.: Action)

Sofern der aktive Zustand auf das aufgetretene Ereignis reagiert, wird als Resultat auf ein Ereignis eine Aktion ausgeführt. Als Aktionen kommen verschiedene Möglichkeiten in Betracht. Zum einen können Daten des Systems verändert werden und zum anderen können Zustandswechsel eine mögliche Aktion repräsentieren. Eine Kombination von

Beidem ist ebenso möglich. In erweiterten Modellen kommen oft Initial-Aktionen zum Einsatz, um den betretenen Zustand zu initialisieren. Außerdem werden in erweiterten Modellen oft Eintritts- und Austrittsaktionen (entry- und exit-Aktionen) verwendet. Diese Aktionen ermöglichen zustandsspezifische Reaktionen beim Eintreten und Austreten.

Zustandsübergänge (engl.: Transitions)

Der Wechsel des Systems von einem aktiven in einen anderen Zustand, wird als Zustandsübergang bezeichnet. Es kann aber auch ein Zustandswechsel in den gleichen Zustand vollzogen werden (engl.: self-transition). Bei der Verwendung von „self-transitions" kann zwischen externen und internen Übergängen unterschieden werden. Bei externen „self-transitions" werden die Eintritts- und Austrittsaktionen ausgeführt, falls vorhanden, während dies bei internen Übergängen nicht der Fall ist.

Das Verhalten des gesamten Automaten wird durch die Modellierung beziehungsweise die Kombination dieser Komponenten dargestellt. Es ist darauf zu achten, alle Komponenten im Model korrekt anzuwenden. Wenn die Modellierung nicht korrekt ist und dem gewünschten Verhalten des Systems nicht entspricht, wird das Verhalten der Implementation auch vom gewünschten Verhalten abweichen. Dieses Modell ist die Grundlage für die Implementierung.

2.1.2 Flache endliche Zustandsautomaten (FSM)

In flachen endlichen Automaten (engl.: Finite State Machines) ist die Anzahl der Zustände bekannt. Es handelt sich nicht um erweiterte, sondern um Standardautomaten. Die beschriebenen Komponenten aus dem vorherigen Abschnitt können bei der Modellierung dieser Automaten eingesetzt werden. Ein hierarchischer Aufbau der Zustände ist bei Finite State Machines (FSM) nicht vorgesehen.

Die meisten Systeme können mit Hilfe von FSM realisiert werden, jedoch ist es ab einer bestimmten Komplexität sinnvoll, auf ein erweitertes Modell zurückzugreifen. Speziell die Nutzung von geschachtelten Zuständen erleichtert oft die Modellierung von komplexen Systemen.

2.1.3 Hierarchische Zustandsautomaten (HSM)

Bei dem Modell eines hierarchischen Automaten (engl.: Hierarchical State Machine) handelt es sich um ein erweitertes Modell des FSM, wie es Harel in *„Statecharts: A Visual Formalism For Complex Systems"* [Harel87] beschreibt. Folglich beinhaltet dieses Modell alle Komponenten aus Abschnitt 2.1.2. Es ist möglich Zustände zu schachteln und Subautomaten beziehungsweise Subzustände (engl.: Substates) zu modellieren. Diese Eigenschaft beschreibt eine der wichtigsten Erweiterungen der

Statecharts zu normalen flachen Automaten. Die wichtigsten Eigenschaften werden nachfolgend erläutert.

In einem HSM gibt es immer einen Top-Zustand, den obersten Zustand im gesamten Modell. Alle weiteren Zustände sind Subzustände von dem Top-Zustand. Wenn der aktive Zustand ein Subzustand ist, befindet sich der Automat auch gleichzeitig in dem dazu gehörigen Superzustand[1]. Subzustände weisen das gleiche Verhalten wie der jeweilige Superzustand auf, sofern kein anderes Verhalten in dem relevanten Zustand definiert ist. Wenn weiteres Verhalten definiert ist, kann das auch als zusätzliches Verhalten zum Superzustand bezeichnet werden. Das Prinzip dieser Verhaltensvererbung kann laut M. Samek [Samek02 S.32] mit der Vererbung von Klassen in der objektorientierten Programmierung verglichen werden. Es handelt sich jedoch nicht um Klassen, sondern um Zustände.

Die Zustandsübergänge bei der Verwendung von hierarchischen Automaten sind etwas komplexer im Vergleich zu Zustandsübergängen in flachen Automaten. Die Schachtelung der Zustände kann bis in beliebige Tiefe erfolgen. Harel lässt Spielraum was die Zustandsübergänge betrifft. So ist es möglich, von einem Subzustand direkt in einen Subzustand eines anderen Zweiges der Hierarchie zu wechseln. Die Anwendung dieser Möglichkeit unterliegt der Verantwortung des Anwenders und ist unter Umständen nicht unbedingt sinnvoll. Die hängt jedoch von der speziellen Anwendung ab. Es muss nach einem speziellen Muster vorgegangen werden, wenn ein Zustandswechsel vollzogen wird. Dabei ist zu unterscheiden, ob sich der Zielzustand[2] im gleichen, oder in einem anderen Zweig der Hierarchie befindet als der Quellzustand[3]. In den meisten Fällen müssen mehrere Zustände durchlaufen werden, um den Zielzustand zu erreichen. Es ist zu beachten, dass auf diesem Weg beim Verlassen der Zustände die Austrittsaktionen und beim Betreten von neuen Zuständen die Eintrittsaktionen ausgeführt werden. Eine weitere Eigenschaft von hierarchischen Automaten, die den Zustandswechsel erschweren, ist die Ermittlung von Reaktionen in Zustandshierarchien. Darunter kann man einen Mechanismus verstehen, der den nächsten Zustand ermittelt, welcher auf das Ereignis reagieren kann. Beim Eintreten eines Ereignisses überprüft der aktive Zustand, ob er für dieses Ereignis eine Reaktion vorsieht. Wenn das nicht der Fall ist, findet ein Zustandswechsel in den direkten Superzustand statt. Dort wird nach dem gleichen Muster verfahren, bis einer der Superzustände reagieren kann oder der oberste Zustand (Top-Zustand) erreicht ist.

Hierarchische Automaten verfügen zudem über die Möglichkeit sich zu „merken", welches der zuletzt aktive Subzustand war, um diesen wieder betreten zu können. Diese Funktion heißt Gedächtnis (engl.: History). Es lassen sich die flache History und die

[1] Superzustand – der Zustand, der in der Hierarchie direkt über dem relevanten Zustand liegt
[2] Zielzustand – aktiver Zustand nach Zustandswechsel
[3] Quellzustand – aktiver Zustand vor Zustandswechsel

tiefe History unterscheiden. Bei der Verwendung der flachen History merkt sich jeder Zustand den letzten aktiven Subzustand in der Ebene direkt darunter. Im Gegensatz dazu merkt sich ein Zustand bei Verwendung der tiefen History seinen letzten aktiven Subzustand für alle Subzustände unter ihm in der Hierarchie.

Einer der wichtigsten Eigenschaften ist die Art der Aktivität der Subzustände. Sie können parallel und damit nebenläufig sein. Das bedeutet, es können verschiedene Subzustände gleichzeitig ablaufen. Die Realisierung eines Zustandsautomaten mit parallelen Subzuständen erfordert die Verwendung von mehreren Threads, um die Nebenläufigkeit zu gewährleisten. Auf der anderen Seite können Subzustände sequentiell ablaufen. Das heißt, es gibt immer nur einen aktiven Zustand im System und somit auch nur einen aktiven Subzustand. Bei dem Einsatz eines solchen Automaten genügt ein Thread, der die verschiedenen Zustände abarbeitet.

2.2 Zustands-Entwurfsmuster

Entwurfsmuster sind wieder verwendbare Prinzipien der Programmierung zur Lösung spezieller Thematiken oder Probleme. Es werden hauptsächlich die relevanten Objekte in Form von Klassen dargestellt. Unter anderem werden die Beziehungen zwischen den benötigten Klassen, ihre Schnittstellen und die Vererbungshierarchien dargestellt. Auf diese Weise kann ein Muster nachvollzogen und auf einen konkreten Problemfall als Lösung angewandt werden. Diese Idee ist die Grundlage der in [GoF95] beschriebenen Entwurfsmuster.

Ein spezielles Entwurfsmuster zur Realisierung eines Zustandsautomaten ist das Zustands-Entwurfsmuster (State Design Pattern - SDP), welches von Erich Gamma et al entwickelt wurde. Es beschreibt einen objektorientierten Ansatz. Auf diesem Muster basieren viele verwendete Zustands-Entwurfsmuster. Details zu diesem Muster folgen in Kapitel 3.3. Darüber hinaus gibt es Muster, die auf einer anderen Technik oder auf anderen Prinzipien basieren. Diese Techniken zur Implementierung von Zustandsautomaten sind meist objektorientiert, prozedurale Ansätze sind auch möglich.

2.2.1 Prozeduraler Ansatz

Prozedurale Lösungen werden in Systemen angewandt, in denen keine objektorientierte Programmiersprache zur Verfügung steht. Es ist möglich, dass eine prozedurale Programmiersprache, wie zum Beispiel ANSI C, aus anderen Gründen gewählt wird. Bei diesen Ansätzen muss auf Techniken wie Vererbung und Laufzeitpolymorphie verzichtet werden. Das bedeutet nicht, dass die Lösung deshalb aufwendiger oder weniger gut an das Problem angepasst ist.

Eine bekannte prozedurale Technik zur Realisierung eines Zustandsautomaten ist die Verwendung von geschachtelten `switch-case` Anweisungen (engl.: nested switch statement). Diese Möglichkeit wird in Kapitel 3.1 erläutert.

In einem speziellen prozeduralen Ansatz von H. Kaltenhäuser werden Datenstrukturen genutzt, um Zustandsdaten zu halten. Hier werden die Zustände in Form dieser Datenstrukturen dargestellt. In diesem Ansatz werden Funktionszeiger auf C-Routinen genutzt, die die Reaktionenrealisieren. Die Zuordnung der Funktionszeiger erfolgt in der Datenstruktur des jeweiligen Zustandes. Die Ereignisverarbeitung wird auch bei diesem Prinzip grundsätzlich durch die Anwendung von `switch-case` Anweisungen durchführt. Dieser Ansatz ermöglicht zudem die Implementierung von hierarchischen Zustandsautomaten. H. Kaltenhäuser entwickelte dieses Prinzip, um es als Laufzeitsystem für seine Prozesslenkung Vorlesungen zu nutzen. Für weitere Details ist der Quellcode des Laufzeitsystems unter [Kalten05] zu finden.

2.2.2 Objektorientierter Ansatz

Die vorgeschlagenen Lösung von Erich Gamma et al, die in Kapitel 3.3 detailliert beschrieben wird, stellt einen objektorientierten Ansatz zur Implementation eines Zustandsautomaten dar. In [GoF95 S.398] heißt es, dass es das Ziel ist, ein spezielles Objekt so aussehen zu lassen, als ob es während der Ausführung seine Klasse und damit auch das Verhalten verändert.

Bei den objektorientierten Ansätzen spielt Laufzeitpolymorphie und Vererbung oft eine große Rolle. Unter Verwendung dieser Techniken kann ein Funktionsaufruf (Ereignis) an die konkrete Klasse delegiert werden. Die konkrete Klasse entspricht dann dem jeweiligen aktiven Zustand.

Andere Ansätze, wie beispielsweise der in Kapitel 3.4 erläuterte Optimal FSM Ansatz, sind ebenfalls objektorientiert. Diese Lösung benutzt zwar die Vererbungstechnik, verzichtet jedoch auf Laufzeitpolymorphie. Es wird eine Kombination von verschiedenen Techniken gebildet. So wird bei dem Beispiel des Optimal FSM Vererbung in Kombination mit `switch-case` Anweisungen genutzt. Des Weiteren wird Delegation zur Ereignisverarbeitung verwendet.

2.2.3 Diverse Abwandlungen und Erweiterungen

Wie bereits im vorherigen Abschnitt erwähnt, werden häufig verschiedene Ansätze kombiniert, welche die Vorteile der verwandten Lösungen zu nutzen. In anderen Fällen wird auf spezielle Probleme eingegangen, indem die Techniken erweitert oder verändert werden. Dabei ist das State Design Pattern der GoF sehr beliebt und wird sehr oft als Basis für individuelle Lösungen benutzt.

In Kapitel 3.5 wird auf einige Abwandlungen und Erweiterungen etwas konkreter eingegangen. Im Rahmen dieser Arbeit ist es nicht sinnvoll noch tiefer als dort auf diese Problematik einzugehen, weil keine relevanten Techniken genutzt werden.

2.3 C++ Grundlagen

Die in dieser Arbeit untersuchte Technik zur Realisierung von Zustandsübergängen ist eine spezielle Anwendung des Operators `Placement-new`. Da die Funktionsweise beziehungsweise der Umfang mit diesem Operator nicht zu den Grundkenntnissen von C++ Programmierern gehört, wird die Anwendung von `Placement-new` in diesem Abschnitt erläutert. Ein anderer wichtiger Bestandteil der neuartigen Implementation ist die Verwendung von Template-Klassen. Die Anwendung von Templates ist ebenfalls ein Thema, das nicht unbedingt zu den programmiertechnischen Grundlagen gehört. Aus diesem Grund wird auch dieses Thema etwas näher betrachtet.

2.3.1 Operator `Placement-new`

Der Operator **`new`** wird in C++ weitgehend genutzt, um Objekte auf dem Heap[4] zu erstellen und abzulegen. Im Gegensatz zum Stack[5], ermöglicht der Heap dem Entwickler den Speicher selbst zu verwalten und somit zu entscheiden, wann der benutzte Speicher freigegeben wird. Der Operator `new` muss überschrieben werden, wenn er für die Realisierung einer eigenen Speicherverwaltung eingesetzt wird.

- **Beispiel:** Erzeugung einer Instanz von `MyClass` auf dem Heap mit dem Operator `new`.

  ```
  MyClass* myClass = new MyClass;
  ```

Bei dem Entwurf von Frameworks ist auf bestimmte Programmierstandards zu achten. D. Schmidt stellt mit [Schmidt98] ein Dokument zur Verfügung, in dem wichtige Regeln für die Erstellung von Frameworks festgehalten sind. Diese Standards sind bei kleineren Programmen sehr sinnvoll. Einer dieser Standards besagt, dass nach der Benutzung des Operators `new`, ein korrespondierender expliziter Aufruf des jeweiligen Destruktors der verwendeten Klasse erfolgen muss. S. Meyers weist in seinem Buch *„Effektiv C++ programmieren“* [Meyers98 S.43] auf die Notwendigkeit des Aufrufes des Destruktors hin, wenn Speicher auf dem Heap belegt wird. Bei nicht Einhaltung dieser Regel kommt es zu einer Speicherlücke. Der belegte Speicher wird nicht freigegeben, wenn das Objekt, welches den Speicher belegt, nicht mehr benötigt wird. Dieser Speicherbereich kann demzufolge nicht mehr vom Programm benutzt werden.

---

[4] Heap – Freispeicher, dynamischer Speicher, vom Entwickler verwaltet
[5] Stack – automatischer Speicher, keine Verwaltung durch Entwickler möglich

Die Möglichkeiten dieses Operators gehen jedoch weit darüber hinaus. Er kann außerdem als Platzierungsoperator (`Placement-new`) verwendet werden, wie in [Alexan03 S.193, Stroustrup97 S.255-257] beschrieben. Auf dieser Art und Weise kann ein Objekt an einer beliebigen Stelle im Speicher angelegt werden. Anders formuliert wird ein neues Objekt an der übergebenen Adresse erzeugt.

- **Beispiel:** Erzeugung einer Instanz von `MyClass` an der Position `myClass1` auf dem Heap. Das Objekt, auf das `myClass1` vor der Operation gezeigt hat ist danach nicht mehr vorhanden. Beide Zeiger zeigen auf das gleiche Objekt. Das neue Objekt ist eine neue Instanz von `MyClass`.

```
MyClass* myClass1 = new MyClass;
. . .
MyClass* myClass2 = new (myClass1) MyClass;
```

Das erste Argument des Operators `new` ist immer die Größe des zu erzeugenden Objektes. In dem dargestellten Beispiel ist es das erste Argument, also die Größe von `MyClass`. An welcher Stelle das Objekt erzeugt werden soll wird von `myClass1` angegeben. Das würde dem folgenden Code sinngemäß entsprechen:

```
operator new(sizeof(MyClass), myClass1);
```

In diesem Fall sollte auf den expliziten Aufruf des Destruktors für `myClass1` verzichtet werden, weil es das Objekt, auf das `myClass2` zeigt, zerstören würde. Es entsteht keine Speicherlücke, weil der allokierte Speicher weiter verwendet wird. Zudem wird bei der Verwendung des Operators `Placement-new` keine weitere Zeit für die Reservierung des Speichers benötigt, weil der Speicher bereits reserviert ist.

2.3.2 Templates

Templates (dt.: Vorlagen oder Schablonen) sind „Programmgerüste", die eine vom Datentyp unabhängige Programmierung ermöglichen. Aufgrund dieser Eigenschaft unterstützen sie die generische Programmierung. Scott Meyers [Meyers98 S.232] schreibt dazu:

> *Ein Template sollte verwendet werden, um eine Reihe Klassen zu generieren, wenn der Typ der Objekte das Verhalten der Elementarfunktionen der Klassen beeinflusst.*

Der Template-Mechanismus erlaubt einem Typ als Parameter bei der Definition einer Klasse oder einer Funktion eingesetzt zu werden. Das Thema Templates in C++ ist sehr groß, da der Template-Mechanismus sehr mächtig ist und damit viele Möglichkeiten der generischen Programmierung bietet. Weitere Details zu diesem Thema werden in [Stroustrup97 Kapitel 13] erläutert. Für diese Arbeit ist nur die Anwendung von

Klassen-Templates von Bedeutung. Aus diesem Grund beschränkt sich die folgende Erläuterung auf die Grundlagen der Klassen-Templates.

Bei der Benutzung von Klassen-Templates muss also die Klasse eines verwendeten Typs vorerst noch nicht bekannt sein. Die Bekanntgabe erfolgt erst bei der Instantiierung des Objektes der Template Klasse. An dieser Stelle muss angegeben werden, von welchem Typ der Vorlagenparameter ist. Der Typ des Vorlagenparameters wird in der gesamten Klasse als Typ verwandt. Im Beispiel aus Abbildung 2-1 ist eine beispielhafte Template-Klasse dargestellt, dort ist `T` der Vorlagenparameter. In der virtuellen Funktion `vf(T* ptr)` wird ein Zeiger vom Typ des Vorlagenparameters als Parameter übergeben, ebenso wie es in der Funktion `f(T* ptr)` der Fall ist. Der Typ von `T` ist noch nicht bekannt. Die notwendige Instantiierung erfolgt in der letzten Zeile des Beispiels. In diesem Fall wird `char` als Typ genutzt. Das hat zur Folge, dass in der Klasse `MyClass` die beiden Funktionen mit `char*` als Parameter arbeiten. Der Vorlagenparameter kann beliebig bestimmt werden. Bei der Verwendung von dem Zeiger `ptr` ist darauf zu achten, dass nur Funktionen implementiert werden, die mit den möglichen Typen der Vorlagenparameter kompatibel sind. In dem Beispiel wurde absichtlich eine virtuelle und eine nicht virtuelle Funktion deklariert. Bei der Nutzung von Klassen-Templates können sowohl virtuelle als auch nicht virtuelle Funktionen implementiert werden. Reine Template-Funktionen könne dagegen nicht virtuell sein [Stroustrup97 S.348]. Template-Funktionen sind nicht mit Funktionen innerhalb von Template-Klassen zu verwechseln.

```
// MyClass.h
#ifndef _MY_CLASS_H_
#define _MY_CLASS_H_

template <class T>
class MyClass{
public:
     virtual void vf(T* ptr);
     void f(T* ptr);
private:
     T* ptr_;
};
// Definition der Funktion mit Template-Parameter
#include „MyClass.tpp"
#endif
. . .
// main
#include „MyClass.h"
MyClass myClass<char>; // Instantiierung
```

Abbildung 2-1: Beispiel Template-Klasse

Im Zuge der Quellcodeorganisation ist es üblich, die Deklarationen von den Definitionen zu trennen. An dieser Stelle tritt bei den Funktionen der Template-Klassen eine Besonderheit auf. Die Funktionsdefinitionen, in denen die Vorlagenparameter als

Übergabeparameter genutzt werden, müssen in eine Datei ausgelagert werden (zum Beispiel mit der Datei-Endung `.tpp`). Diese Datei muss nach der Definition (am Ende der Header-Datei) wieder eingebunden werden (siehe Beispiel). Diese Besonderheit tritt nicht bei allen Compilern auf.

2.4 XML

In diesem Abschnitt wird erläutert, was benötigt wird, um mit der Programmiersprache C++ eine XML-Datei zu parsen und anschließend zu verarbeiten beziehungsweise die darin befindlichen Daten zu speichern. Es wird ein Model zur abstrahierten Darstellung des XML-Dokumentes benötigt, das Document Object Model (DOM). Um schnell und einfach ein DOM-Objekt zu erhalten, kann man einen XML-Parser benutzen. Es gibt viele existierende XML-Parser, die frei verfügbar und gut nutzbar sind. Ein weit verbreiteter und außerdem freier XML-Parser ist der Xerces XML Parser [XercesXML].

2.4.1 Document Object Model (DOM)

Aus dem XML-Dokument wird beim Parsen ein DOM Dokument-Objekt erstellt. Ein DOM ist eine sprachneutrale, baumorientierte API, die es ermöglicht ein XML-Dokument als eine Menge von verschachtelten Objekten, den Nodes[6] darzustellen. Diese Objekte können Eigenschaften besitzen. Der Kern von DOM ist ein Satz von abstrakten Schnittstellen (für Details siehe [HarMea04]). Die verschachtelten Knoten repräsentieren das eingelesene XML-Dokument, inhaltlich und strukturell. Es werden Funktionen zum Zugriff auf die einzelnen Knoten bereitgestellt. Es ist möglich, sämtliche Informationen aus den DOM-Objekten zu erhalten, womit alle Daten aus einem XML-Dokument erreichbar werden. Die Abbildung 2-2 verdeutlicht die Struktur und die Möglichkeiten des DOM. Speziell die Schachtelung einzelner Knoten, mit Hilfe der DOM-Elemente, wird deutlich.

Das DOM ist in Bezug auf das Betriebssystem und die Programmiersprache neutral, das wird durch die eingesetzte Interface Description Language[7] (IDL) ermöglicht. So gibt es speziell für C++ einen XML-Parser, der die IDL nutzt und das DOM in C++ verfügbar macht.

---

[6] Node – Knoten, die die XML Struktur repräsentieren
[7] Interface Description Language – Programmiersprachen unabhängige Schnittstellendefinition

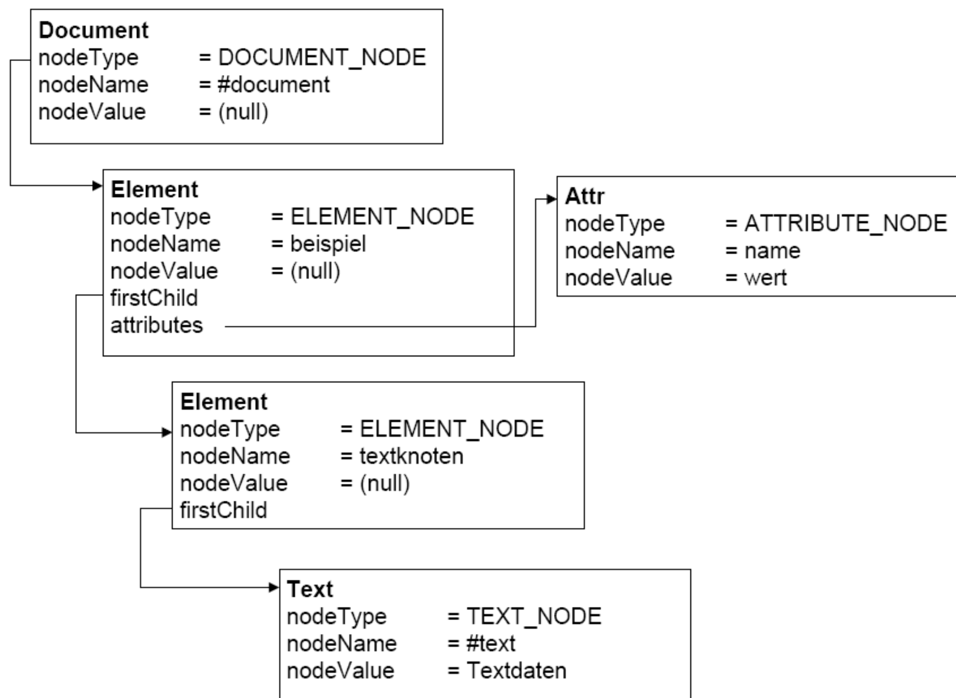

Abbildung 2-2: DOM-Struktur

2.4.2 Xerces XML Parser

Der Xerces XML-Parser, ist ein bereits vorhandener und sofort einsetzbarer XML-Parser, dessen Bedienung sich als sehr einfach erweist. Der Parser für C++ wird als Paket in Kombination mit dem DOM angeboten und ist kostenlos im Internet verfügbar [XercesDL].

Es bedarf einiger relativ aufwendiger Konfigurationen, bevor das Parsen eines Dokuments beginnen kann. Um diesen Schritt zu beschleunigen, bietet sich ein Beispiel zum Parsen eines XML-Dokumentes in ein DOM an. Es heißt „XercesDOMParser", es kann unter [XDOMPar] eingesehen und wie vorhanden übernommen werden. Es muss lediglich der Dateiname des einzulesenden Dokumentes angegeben werden und das DOM-Dokument kann erzeugt werden. Der weiteren Verarbeitung des DOM-Dokumentes steht somit nichts mehr entgegen.

3 Analyse existierender Zustands-Entwurfsmuster

Es gibt viele Zustands-Entwurfsmuster zur Implementation von Zustandsautomaten, dabei ist die Gewichtung der verschiedenen Kriterien weit verteilt. So steht bei einigen Mustern die einfache und schnelle Implementierung im Vordergrund, während andere Muster das Gewicht mehr auf eine möglichst schnelle Ausführung legen. Wiederum andere versuchen ein Muster bereit zu stellen, in dem Änderungen an den Zustandsautomaten möglichst einfach vorzunehmen sind. Im Idealfall sollte ein Zustands-Entwurfsmuster diese Eigenschaften kombinieren, beziehungsweise einen Kompromiss bilden. Hier kommt es darauf an, für welchen Zweck beziehungsweise für welches Problem das Muster als Lösung benötigt wird.

Bereits vor der Implementierung eines Zustandsautomaten sollten einige Fragen bezüglich des Designs und der Funktionalität geklärt werden.

- Wie werden Ereignisse dargestellt?
- Wie werden eventuelle Ereignisparameter gehandhabt?
- Wie werden Zustände dargestellt?
- Wie werden Zustandsübergänge realisiert?
- Wie werden die Ereignisse an den Zustandsautomaten weitergegeben.

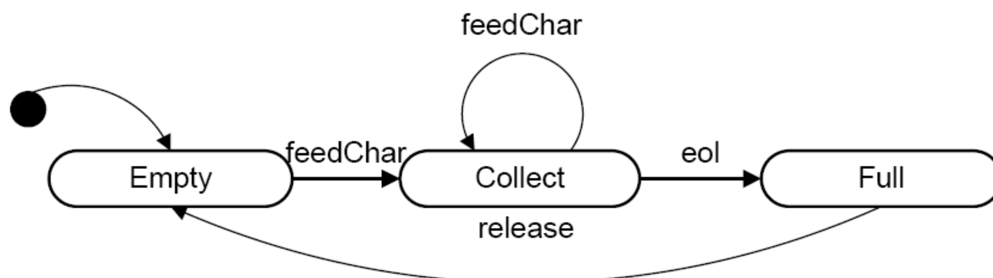

Abbildung 3-1: Zustandsautomat zur Erklärung

Nachfolgend werden die bekanntesten und am weitesten verbreiteten Muster dargestellt und analysiert, um einen Überblick über die bestehenden Muster und deren Vor- und Nachteile zu erhalten. Dabei werden Zustände, Ereignisse und Zustandsübergänge auf verschiedene Arten realisiert. Es werden hauptsächlich die Standardtechniken zur Implementation von *flachen* Zustandsautomaten dargestellt. Bereits M. Samek stellte in

[Samek02 S.55] fest, dass es sehr schwer ist, Standardtechniken zur Implementation von *hierarchischen* Automaten zu entwerfen.

Die in den Abschnitten 3.1 bis 3.4 erläuterten Techniken werden anhand des in Abbildung 3-1 dargestellten Zustandsautomaten verdeutlicht. Detaillierte Erläuterungen dieser Techniken sind in [Samek02 Kapitel 3] zu finden.

3.1 Geschachtelte `switch-case` Anweisungen (Nested switch Statement)

Eine der bekanntesten Techniken zur Implementation von Zustandsautomaten ist die Verwendung von geschachtelten `switch-case` Anweisungen. Diese Technik wird oft bei der Verwendung von modularen beziehungsweise prozeduralen Programmiersprachen wie C angewandt, weil hier die Vorzüge beziehungsweise die Möglichkeiten der *objektorientierten* Programmiersprachen nicht benötigt werden.

```
enum Signal {
   FEEDCHAR_SIG, EOL_SIG, RELEASE_SIG
};
enum State {
   EMPTY, COLLECT, FULL
};
. . .
State myState;

void dispatch(unsigned const sig);
void tran(State target) { myState = target;}
```

Abbildung 3-2: Codeausschnitt Zustände, Ereignisse, Funktionen (nested switch)

Die verwendeten Zustände und die auftretenden Ereignisse (Signale) werden üblicherweise als Aufzählung (engl.: Enumeration) gehalten. Zur Repräsentation beziehungsweise zum Speichern des aktiven Zustandes wird eine Variable benötigt (siehe Abbildung 3-2 `myState`). Grob betrachtet, besteht die Technik aus zwei in sich geschachtelten `switch-case` Anweisungen. Um den Code übersichtlich zu strukturieren, empfiehlt es sich die `switch-case` Anweisungen in eine Funktion `dispatch()` auszulagern (siehe Abbildung 3-3). Der Aufruf erfolgt vom Client aus, somit arbeitet diese Funktion wie ein Event-Handler. In der ersten Stufe der `switch-case` Anweisung wird entschieden, um welchen Zustand es sich handelt. In der zweiten Stufe wird das aufgetretene Ereignis entsprechend dem aktiven Zustand verarbeitet. Der Wechsel des Beispielautomaten vom Zustand `Collect` nach `Full` ist in Abbildung 3-3 implementiert. Der eigentliche Zustandswechsel erfolgt mit Hilfe der Funktion `tran()`.

Vorteile dieser Technologie:

- Es ist ein sehr einfaches Prinzip, mit einer einfachen Struktur, so wird ist leicht verständlich und umsetzbar.
- Es wird wenig Speicher benötigt, nur eine Variable zur Repräsentation des Zustandes wird genutzt.

Nachteile dieser Technologie:

- Zustände und Ereignisse müssen als Aufzählungen gehalten werden.
- Die Wiederverwendung von Quellcode wird nicht unterstützt, für jeden Zustand erfolgt eine eigene Ereignisverarbeitung.
- Die Verarbeitungszeit der Ereignisse ist durch die geschachtelten `switch-case` Anweisungen von der Anzahl der Zustände und Ereignisse abhängig (O(log n), n ist die Anzahl der auftretenden Fälle).
- Sehr fehleranfällig, weil Änderungen an mehreren Stellen vorgenommen werden müssen.
- Verwendung von `entry` / `exit` Aktionen müsste pro Zustand an mehreren Stellen erfolgen.
- Die Technik ist in dieser Form nicht für hierarchische Automaten geeignet.

```
void dispatch(unsigned const sig) {
   switch (myState) { // erste Stufe
   case EMPTY:
 . . .
   }
   case COLLECT:
      switch (sig) { // zweite Stufe
      case FEEDCHAR_SIG:
         count++;
         break;
      case eol
         tran(FULL);
         break;
      }
      break;
 . . .
   }
}
```

Abbildung 3-3: Codeausschnitt `dispatch()`(nested switch)

3.2 Zustandstabelle (State Table)

Ebenfalls eine bekannte Technik zur Realisierung eines Zustandsautomaten, ist die Verwendung einer Zustandstabelle. Bei dieser Technik wird eine Tabelle genutzt, in der die Reaktionen aller Zustände bei den auftretenden Ereignissen festgehalten werden. Eine beispielhafte Zustandstabelle ist in der Tabelle 3-1 dargestellt.

In den Zellen der Tabelle sind die Übergänge definiert und können folgendermaßen betrachtet werden. Der erste Parameter gibt eine Aktion und der zweite Parameter den Zielzustand an. Die Funktionen `a1()-a3()`, die spezielle Aktionen darstellen, werden als Reaktion ausgeführt werden. Bei einem Zustandswechsel ohne angegebene Aktion wird die Funktion `doNothing()` ausgeführt. Freie Zellen in der Tabelle entsprechen der Aktion `doNothing()` und keinem Zustandswechsel. Wenn der Automat zum Beispiel das Ereignis `EOL_SIG` empfängt, während der Zustand `Collect` aktiv ist, wird die Aktion `a2()` ausgeführt. Darauf folgt der Wechsel nach Zustand `Full`.

Allgemein lässt sich die Technik in zwei große Teile aufgliedern. Die Technik lässt sich in einen generischen und anwendungsspezifischen Teil gliedern. Den generischen Teil stellt die Klasse `StateTable` aus Abbildung 3-4 dar. Dieser Teil kann wieder verwendet werden, weil er als Basisklasse eingesetzt wird, er enthält die eigentliche Funktionalität. Den anwendungsspezifischen Teil stellt die beispielhafte Klasse `CParser` aus Abbildung 3-5 dar. In diesem Teil werden die Zustände und Ereignisse als Aufzählungen gehalten. Die benötigten Variablen und Funktionen werden ebenfalls dort definiert. Dieser Teil muss bei der Realisierung angepasst werden, denn in diesem Teil werden die Zustandstabelle und die Daten gehalten, auf die alle Zustände zugreifen. Die zu implementierende Klasse ist von der Klasse `StateTable` aus Abbildung 3-4 abzuleiten.

Tabelle 3-1: Beispiel Zustandstabelle

↓ Zustände / Ereignisse →	**FEEDCHAR_SIG**	**EOL_SIG**	**RELEASE_SIG**
empty	a1(), collect		
collect	a1(), collect	a2(), full	
full			a3(), empty

Im generischen Teil wird der Zustandswechsel implementiert. Für diesen Zweck wird eine `Pointer-to-Member` Funktion der Klasse `StateTable` und eine Struktur `Tran` genutzt. In der Struktur werden die auszuführende Aktion und der Zielzustand

festgehalten. Die Tabelle ist ein Array in dessen Zellen die Übergänge in Form von `Tran` Strukturen gespeichert werden. Die Größe der Tabelle ist abhängig von der Anzahl der Zustände und Ereignisse. Beim Betrachten des Konstruktors ist zu erkennen, dass die Anzahl der Zustände und Ereignisse sowie ein Zeiger auf eine Instanz einer Zustandstabelle übergeben werden. Für diese Werte werden Variablen bereitgestellt. Ebenso wird der aktive Zustand vom Typ `unsigned int`. Die oben erwähnte Funktion `doNothing()` ist ebenfalls dort implementiert, enthält jedoch keinen Code, weil sie keine Aktion ausführen soll.

```
class StateTable {
public:
   typedef void (StateTable::*Action)(); // Pointer-to-member
   struct Tran {
      Action action;
      unsigned nextState;
   };
   StateTable(Tran const *table, unsigned nStates, unsigned
         nSignals)
      : myTable(table), myNsignals(nSignals),
      myNstates(nStates) {}
   virtual ~StateTable() {}   // virtual dtor
   void dispatch(unsigned const sig) {
      register Tran const *t = myTable + myState*myNsignals +
      sig;
      (this->*(t->action))();
      myState = t->nextState;
   }
   void doNothing() {}
protected:
   unsigned myState;
private:
   Tran const *myTable;
   unsigned myNsignals;
   unsigned myNstates;
};
```

Abbildung 3-4: Quellcodeausschnitt generischer Teil (Zustandstabelle)

Für die Ereignisverarbeitung existiert die Funktion `dispatch()`. Wie im Beispiel zu sehen ist, wird in drei Schritten vorgegangen. Zuerst wird festgestellt, welche Zelle der Tabelle die relevante Transition enthält. Anschließend wird die entsprechende Aktion mit Hilfe der zugehörigen `Pointer-To-Member` Funktion ausgeführt. Abschließend erfolgt der Zustandswechsel.

Die spezifische Funktionen (Aktionen) werden im anwendungsspezifischen Teil definiert. Des Weiteren werden die Daten, auf die die Zustände während der Durchführung zugreifen, in diesem Teil gehalten, um sie für alle Zustände zugängig zu machen. Die Zustandstabelle vom Typ `StateTable::Tran` ist in diesem Teil definiert. Die Schlüsselwörter `static` und `const` fallen besonders auf. Diese Schlüsselwörter werden verwendet, weil die Tabelle aus allen Instanzen von `CParser`

erreichbar sein muss (`static`). Das Schlüsselwort `const` sorgt für ausschließlich lesenden Zugriff.

Nicht zu vergessen ist die Initialisierung der Zustandstabelle. Hierzu ist ein Eintrag für jede Kombination aus Zustand und Ereignis notwendig. Das Beispiel in Abbildung 3-6 zeigt die Initialisierung der Tabelle für das verwendete Automatenmodell. Für die speziellen Funktionen aus dem anwendungsspezifischen Teil ist ein „upcast" der `Pointer-to-Member` Funktion notwendig, weil diese Funktionen Mitglieder der abgeleiteten Klasse sind und nicht direkt zur Klasse `StateTable` gehören.

```
enum Event {
   FEEDCHAR_SIG, EOL_SIG, RELEASE_SIG, MAX_SIG
};
enum State {
   EMPTY, COLLECT, FULL, MAX_STATE
};
class CParser : public StateTable { // Zustandsautomat
public:
   CParser2() : StateTable(&myTable[0][0], MAX_STATE, MAX_SIG) {}
   void init() { count = 0; myState = EMPTY; }
   int getCount() const { return count; }
private:
   void a1() { count ++; }
   void a2() { count = 0; }
private:
   static StateTable::Tran const myTable[MAX_STATE][MAX_SIG];
   int count;
};
```

Abbildung 3-5: Quellcodeausschnitt anwendungsspezifischer Teil (Zustandstabelle)

Vorteile dieser Technik:

- Ein Zustandsautomat wird direkt in Form einer Zustandstabelle abgebildet.
- Geschwindigkeiten der Ereignisverarbeitung und des Zustandswechsels sind konstant (O(const)). Jedoch werden die spezifischen Aktionen nicht berücksichtigt.
- Der Quellcode wird teilweise wieder verwendet (generischer Teil).

Nachteile dieser Technik:

- Die Zustände und Ereignisse (Ereignissee) werden als Aufzählungen gehalten.
- Die benötigte Zustandstabelle ist in der Regel sehr groß und speicherintensiv.
- Jede Aktion wird durch eine Funktion von sehr feiner Granularität repräsentiert.
- Die Initialisierung der Zustandstabelle ist sehr kompliziert. Änderungen im Automatenmodell sind sehr aufwendig zu implementieren, dadurch können sich

schnell Fehler einschleichen. (Zum Beispiel das Hinzufügen eines neuen Zustands erfordert das Einfügen einer komplett neuen Zeile in der Tabelle.)

- Die Technik ist nicht für hierarchische Automaten geeignet, eine Erweiterung ist möglich aber sehr aufwendig. Zudem müssten komplette Übergangsketten fest programmiert werden, dadurch würde die Zustandstabelle wesentlich komplizierter werden.

```
StateTable::Tran const CParser2::myTable[MAX_STATE][MAX_SIG] = {
   {{static_cast<StateTable::Action>(&CParser2::a1), COLLECT },
    {&StateTable::doNothing, EMPTY },
    {&StateTable::doNothing, EMPTY}},
   {{static_cast<StateTable::Action>(&CParser2::a1), COLLECT },
    {static_cast<StateTable::Action>(&CParser2::a2), FULL },
    {&StateTable::doNothing, COLLECT }},
   {{&StateTable::doNothing, FULL },
    {&StateTable::doNothing, FULL },
    {static_cast<StateTable::Action>(&CParser2::a3), EMPTY }}
};
```

Abbildung 3-6: Quellcodeausschnitt Initialisierung der Tabelle (Zustandstabelle)

3.3 State Design Pattern (SDP) von Gamma et al

Die von der „Gang of Four“ (GoF) [GoF95 S.398-409] entwickelte Technik entspricht einem objektorientierten Ansatz zur Implementation eines Zustandsautomaten. Jeder Zustand wird durch eine Klasse dargestellt und jedes Ereignis wird durch eine Funktion repräsentiert. Ausgehend von einer zentralen Kontextklasse kann auf die Zustände zugegriffen werden.

Das Prinzip basiert auf der Anwendung von Delegation und Laufzeitpolymorphie. Die Kontextklasse, die eine abstrakte Zustandsklasse aggregiert, bildet die Grundlage für die Delegation eines Ereignisses an den aktiven Zustand. Dies erfordert das Vorhandensein der gleichen Funktionen in den beiden Klassen. Die Funktionen unterscheiden sich jedoch in der Signatur. Die abstrakte Zustandsklasse agiert wie eine Schnittstelle, in der jedes Ereignis einer virtuellen Funktion entspricht. Durch das Ableiten einer konkreten Zustandsklasse von der abstrakten Zustandsklasse, ist es der konkreten Zustandsklasse möglich, die benötigten Funktionen zu überschreiben. So wird bei dem Aufruf einer Funktion durch die Kontextklasse der Aufruf an die Funktion einer konkreten Zustandsklasse (Instanz des aktiven Zustandes) delegiert.

Die Abbildung 3-7 zeigt ein konkretes Klassendiagramm zum SDP unter Verwendung des oben genannten Automatenmodells. Die Klasse `Context` steht für die Kontextklasse, die abstrakte Zustandsklasse wird durch die Klasse `State` dargestellt. Aus der Abbildung ist ersichtlich, dass es genügt, in den konkreten Zuständen die Funktionen zu implementieren, die als Ereignis vorkommen können und sinnvoll sind.

Für diesbezügliche Entscheidungen ist das Automatenmodell (Abbildung 3-1) zu betrachten. So kann beispielsweise in der Klasse `StateCollect` auf die Funktion `release()` verzichtet werden. Falls sie doch aufgerufen wird, wird die Funktion von der abstrakten Basisklasse ausgeführt. Die virtuellen Funktionen der Basisklasse müssen demzufolge definiert sein.

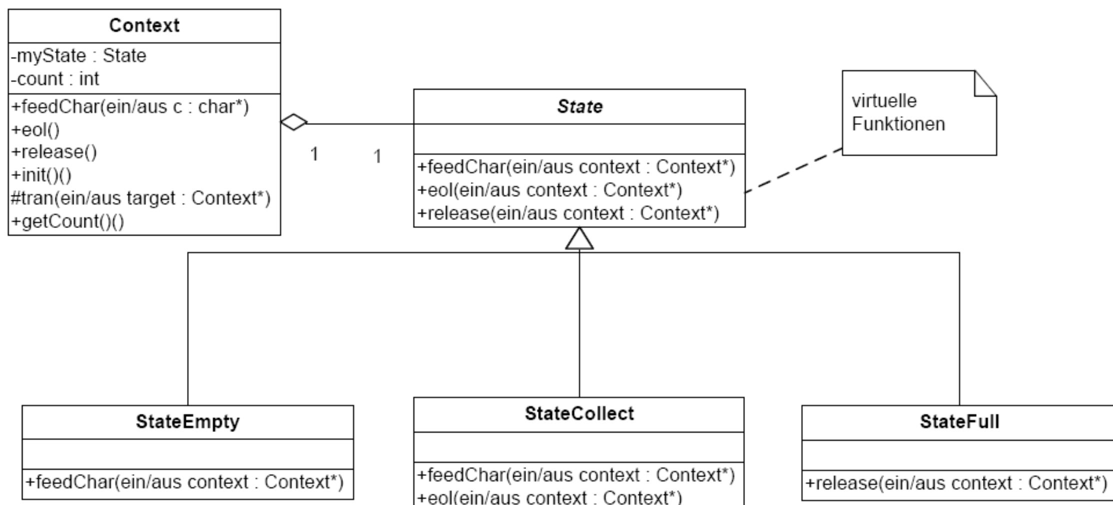

Abbildung 3-7: Klassendiagramm State Design Pattern

In der Kontextklasse muss von jedem Zustand eine Instanz gehalten werden, um von jedem Zustand eine Exemplar zur Verfügung zu stellen. Dieses Vorgehen ist notwendig, da der aktive Zustand in der Kontextklasse als Zeiger auf eine Instanz realisiert ist. Bei der Verwendung solcher Exemplare ist die Anwendung des Singleton-Entwurfsmusters empfehlenswert. In [GoF95 S.157] wird dieses Muster detaillierter beschrieben. Die konkreten Zustände, die repräsentativen Klassen, werden mit dem Schlüsselwort `friend` versehen, um den Zugriff auf die privaten Variablen und Funktionen (`tran`) der Kontextklasse zu ermöglichen. In der Kontextklasse werden weiterhin eine Initialfunktion `init()` und eine Funktion für den Zustandswechsel `tran(State* state)` bereitgestellt. Die Funktion `tran` ist als `private` Funktion deklariert, um sie vor unerlaubten Zugriffen zu schützen. Die Datenhaltung erfolgt ebenfalls in dieser Klasse. Das Auslösen eines Ereignisses ist Aufgabe des Klienten. Dies erfolgt über das aufrufen der entsprechenden Funktion der Kontextklasse und das Weiterleiten an den aktiven Zustand. Die Kontextinstanz übergibt mit dem Aufruf eine Referenz auf sich selbst, so kann jeder konkrete Zustand wieder auf Daten und Funktionen von der Klasse Kontext zugreifen. Zudem ist dieser Mechanismus für den Zugriff auf die Zustands-wechsel_funktion in der Kontextklassse notwendig. In Abbildung 3-8 ist die Klasse `Context` in Bezug auf das Beispielmodell abgebildet.

```
class Context {
   friend class StateEmpty;
   friend class StateCollect;
   friend class StateFull;
   static StateEmpty myStateEmpty;
   static StateCollect myStateCollect;
   static StateFull myStateFull;
   State *myState;
   int count;
public:
   Context(State *initial) : myState(initial){}
   void init() { count = 0; tran(&myStateEmpty);}
   long getCount() const { return count;}
   void feedChar() { myState->feedChar(this);}
   void eol() { myState->eol(this);}
   void release() { myState->release(this);}
protected:
   void tran(State *target) { myState = target; }
};
```

Abbildung 3-8: Quellcodeausschnitt Kontextklasse (GoF)

Die Ereignisverarbeitung findet im konkreten aktiven Zustand statt. An dieser Stelle werden die entsprechenden Aktionen integriert und ein eventueller Zustandswechsel vollzogen. Der Quellcodeausschnitt in Abbildung 3-9 zeigt den Zustandswechsel vom Zustand `Collect` nach `Full`. Es wird keine Aktion, sondern lediglich die Funktion `tran` der Kontextinstanz ausgeführt, der Parameter ist eine Referenz auf den Zustand `Full`. Die Notwendigkeit des Parameters vom Typ `Context` wird in diesem Beispiel verdeutlicht. Hierbei befindet sich die eigentliche Funktion zum Wechseln des Zustandes in der Klasse `Context` (siehe Abbildung 3-8).

```
Context context;
context.init();
 . . .
context.eol();  // myState->eol(this)


void StateCollect::eol(Context *context) {
   context->tran(&Context::myStateFull);
}
```

Abbildung 3-9: Quellcodeausschnitt Zustandswechsel (GoF)

Vorteile dieser Technik:

- Die Zustände werden in Klassen eingeteilt, wodurch das Verhalten klassen-spezifisch wird.
- Die Zustandsübergänge sind sehr effizient, da lediglich ein Zeigerwert über-schrieben wird.

- Durch die Technik der späten Bindung wird eine sehr schnelle Ereignisverarbeitung erreicht (O(const) ohne Rücksicht auf die spezifischen Aktionen).
- Jedes Ereignis ermöglicht eine spezifische Signatur erhalten, so ist es möglich Parameter zu übergeben.
- Diese Technik ist speicherschonend, die konkreten Zustände besitzen keine Attribute sondern nur Funktionen (4 Byte je Zustand).
- Die Zustände und Ereignisse müssen nicht als Aufzählungen definiert werden.

Nachteile dieser Technik:

- Die Kapselung der Kontextklasse wird durch die Freundschaft (Schlüsselwort: `friend`) zu den konkreten Zuständen aufgebrochen.
- Es ist ein indirekter Zugriff (via `Context` Zeiger) auf die Kontextklasse notwendig. Es entsteht eine gegenseitige Abhängigkeit.
- Von jedem Zustand muss eine Instanz gehalten werden. Dadurch wird bei der ansonsten speicherschonenden Technik Speicher verschwendet.
- Um auf neue Ereignisse reagieren zu können, müssen diese zur Schnittstelle `State` (abstrakte Zustandsklasse) und zu der Kontextklasse hinzugefügt werden.
- Hierarchische Automaten sind mit dieser Technik sehr schwer realisierbar.

3.4 Optimaler endlicher Zustandsautomat (optimal FSM)

Das Muster des optimalen FSM wurde von M. Samek entwickelt. Diese Technik ist eine Mischung der drei zuvor erläuterten Techniken, mit dem Ziel, eine optimale Kombination zu erhalten. Das Klassendiagramm aus Abbildung 3-10 spiegelt die Struktur des Musters wieder.

Die Rolle der Kontextklasse der GoF und zugleich die Rolle des generischen Teils aus 3.2 übernimmt die Klasse `FSM` (Abbildung 3-11). Die Zustände werden direkt durch Funktionen repräsentiert, die gleichzeitig die Ereignisverarbeitung übernehmen. Die Funktionen sind Mitglieder von der Klasse `FSM`. In diesem konkreten Fall jedoch von der Klasse `CParser`, da diese von der Klasse `FSM` abgeleitet wird und die Ereignisverarbeitung steuert. Es wird ein direkter Zugriff auf den Zustand (wichtig für Zustandswechsel) ermöglicht. Im Gegensatz zu dem Muster der GoF wird die Kapselung nicht aufgebrochen. Der aktive Zustand wird mit einem selbst definierten Typ realisiert. Dieser Typ ist eine `Pointer-to-Member` Funktion, die je nach Zustand auf eine andere Funktion abgebildet wird. Auftretende Ereignisse werden durch Aufzählungen dargestellt.

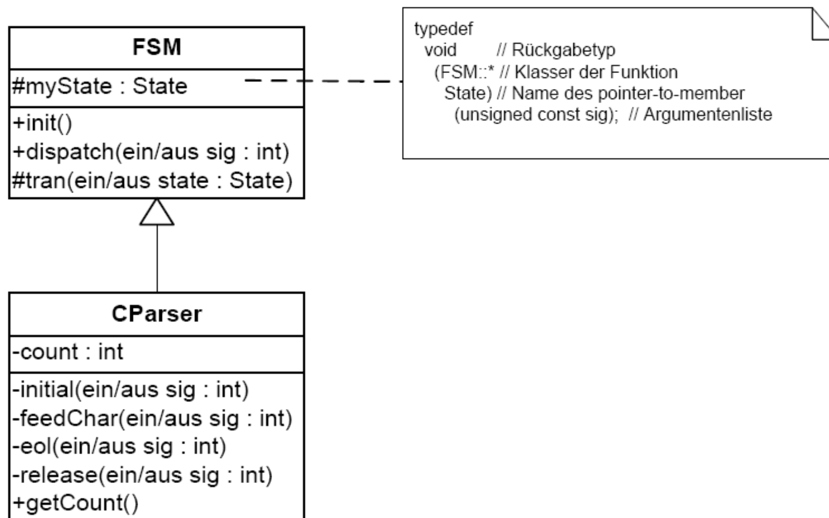

Abbildung 3-10: Klassendiagramm optimal FSM

Ein Unterschied zu den vorherigen Techniken ist der Pseudozustand `Initial`, dieser ist für die Initialisierung des Automaten und den Wechsel nach `Empty` zuständig. Wie schon zuvor wird die Funktion `dispatch(unsigned const sig)` als Funktion zur Ereignisverarbeitung verwendet. Sie arbeitet ähnlich wie die aus dem `StateTable` Ansatz, jedoch wird das Ereignis als Parameter der aufgerufenen `Pointer-to-Member` Funktion übergeben. Ein Zustandswechsel erfolgt durch den Aufruf von `tran(State target)`. Die zuvor definierte Variable (`typedef` einer `Pointer-to-Member Funktion`) vom Typ `State` wird überschrieben.

```
class Fsm {
public:
   typedef void (Fsm::*State) (unsigned const sig);
   Fsm(State initial) : myState(initial) {}
   virtual ~Fsm() {}
   void init() { (this->*myState)(0); }
   void dispatch(unsigned const sig) { (this->*myState)(sig); }
protected:
   void tran(State target) { myState = target; }
   #define TRAN(target_) tran(static_cast<State>(target_))
   State myState;
};
```

Abbildung 3-11: Quellcodeausschnitt optimaler FSM

Für die Ereignisverarbeitung innerhalb der Zustände werden `switch-case` Anweisungen genutzt, wie bei der Technik aus Abschnitt 3.1. Auf diese Art und Weise wird geprüft, welches Ereignis aufgetreten ist. Es handelt sich nicht mehr um eine geschachtelte `switch-case` Anweisung, sondern um eine einfache `switch-case`

Anweisung. Der Zugriff auf die Daten kann ohne Probleme erfolgen, indem die Daten in der gleichen Klasse gehalten werden, in der auch die Ereignisfunktionen definiert sind. Ein Beispiel für die Reaktion im Zustand `Collect` bei Eintreffen eines Ereignisses ist in Abbildung 3-12 zu sehen. Ein Zustandswechsel erfolgt, wenn das eintreffende Ereignis `EOL_SIG` entspricht.

```
void CParser::collect(unsigned const sig) {
   switch (sig) {
   case EOL_SIG:
      TRAN(&CParser::full);
      break;
   case FEEDCHAR_SIG:
      count++;
      break;
   }
}
```

Abbildung 3-12: Quellcodeausschnitt Ereignisverarbeitung (optimal FSM)

Vorteile dieser Technik:

- Der Ansatz ist sowohl in der Struktur als auch in der Umsetzung einfach.
- Das Verhalten ist durch die Verwendung von Funktionen, die die Zustände bearbeiten, verteilt.
- Bei der Kapselung der Klassen muss auf keine Kompromisse eingegangen werden.
- Es kann direkt auf die Daten zugegriffen werden.
- Die Technik ist speicherschonend, denn nur eine Variable (`myState` Zeiger) muss verwaltet werden.
- Die Wiederverwendung ist einfach, da sich der generische Teil in der Klasse `FSM` befindet (Basisklasse).
- Die Zustandsübergänge sind sehr effizient, nur ein Zeiger muss überschrieben werden.
- Die Geschwindigkeit der Ausführung ist schnell. Eine Stufe der `switch-case` Anweisung fällt im Vergleich zum geschachtelten `switch-case` weg.
- Es ist keine Aufzählung für die Zustände notwendig.
- Änderungen im Automatenmodell sind schnell und einfach umsetzbar.

Nachteile dieser Technik:

- Für Ereignisse wird eine Aufzählung verwendet.

- Es ist immer noch eine Stufe von `switch-case` Anweisungen vorhanden, damit ist die Ausführungsgeschwindigkeit von der Anzahl der Fälle abhängig (O(log(n)), n ist die Anzahl der Fälle).
- Diese Technik ist in dieser Form nicht für hierarchische Automaten geeignet.

3.5 Erweiterungen und Spezialisierungen

Viele Zustands-Entwurfsmuster basieren auf dem in 3.3 beschriebenen Muster der GoF. Diese Muster sind oft Spezialisierungen zur Lösung eines bestimmten Problems. Es kommt aber auch vor, dass das Muster der GoF als Grundlage genutzt und ausgebaut wird. Es ist möglich, dass eine neue Technologie zur Umsetzung genutzt oder die Struktur und die Reaktionsweise geändert wird.

In einer Arbeit von Paul Adamczky [Adamcz03] werden einige bekannte Varianten zusammengefasst. Dort sind die meisten Entwurfsmuster Kombinationen aus dem State Design Pattern der GoF und einem weiteren Entwurfsmuster. Ein konkretes Beispiel dafür liefern Jim Odrowski und Paul Sogaard mit dem Muster **Orthogonal State** aus [OdrSog96]. Da es sich lediglich um eine Kombination aus zwei Mustern, mit dem State Pattern als Ausgangpunkt handelt, wird keine neue Technik oder Technologie dargestellt. Aus diesem Grund wird auf dieses Muster hier nicht weiter eingegangen.

Paul Dyson und Bruce Anderson gehen dagegen an das Thema der Zustands-Entwurfsmuster anderes heran. Sie beschäftigen sich in [DysAnd96] zwar ebenfalls mit Erweiterungen und Verfeinerungen des State Design Pattern, bieten jedoch spezielle Lösungen für bestimmte Probleme mit dem State Pattern der GoF als Ausgangspunkt an. So handelt es sich wiederum nicht um neue Techniken, sondern eher um andere Strukturen bezüglich des Aufbaus der Zustandsmuster. Das Verhalten in Abhängigkeit von dem vorliegenden Zustandsautomaten, und nicht die Technik der Muster, steht im Vordergrund. Beispielsweise das **Exposted State** Muster behandelt unter anderem den Fall, dass bestimmte Daten nur in wenigen Zuständen relevant sind. Es soll verhindert werden, dass die Kontextklasse unnötige Daten enthält. Detaillierte Informationen dazu und weitere Spezialisierungen können in der genannten Arbeit nachgelesen werden.

In der Veröffentlichung *„Finite State Machine Patterns“* von Sherif M. Yacoub und Hanny H. Ammar [YacAmm98a] werden spezielle Punkte betrachtet und Lösungen gesucht. Dort sind zum Beispiel die Art des Automaten, die Struktur, die Zustandsübergänge oder die Zustandsinitialisierung von Bedeutung. Es sind jedoch keine neuen Techniken zum Vorschein gekommen.

Von den oben genannten Erweiterungen und Spezialisierungen gibt es sehr viele Varianten. In den erwähnten Arbeiten werden die Bekanntesten zusammengefasst und kurz skizziert.

3.6 Muster für hierarchische Zustandsautomaten (HSM)

Für den Entwurf von hierarchischen Zustandsautomaten gibt es keine Standard-techniken, die als Entwurfsmuster bezeichnet werden können. Die vorhandenen Lösungen sind meist sehr komplex und schwer zu realisieren oder die Zustände mit den zugehörigen Zustandsübergängen sind fest codiert. Aus diesem Grund wird auf diese Technik im Folgenden nur kurz eingegangen.

Einen Ansatz für die Realisierung von hierarchischen Automaten bietet die Veröffentlichung *„A Pattern Language of Statecharts“* [YacAmm98b S.6-8]. Das State Pattern von der GoF wird erneut als Grundlage genutzt, jedoch erfolgt eine Kombination mit dem Composite Pattern. Das resultierende Muster wird **Hierarchical Statechart** genannt. Auf diese Art und Weise ist es möglich, einen hierarchischen Zustandsautomaten aufzubauen. Es wird das Composite Pattern auf das State Design Pattern angewandt. So kann ein Superzustand auch andere Zustände enthalten. Diese Lösung ist aufgrund der Anwendung des Composite Pattern sehr komplex, so dass sich die Frage stellt, ob es sich als Muster für einen Zustandsautomaten eignet. Eine detaillierte Beschreibung des Composite Pattern kann aus [GoF95 S.239-253] entnommen werden.

Eine weitere Möglichkeit bietet M. Samek mit dem **Quantum HSM** aus [Samek02 S.81-130]. Diese Möglichkeit baut auf dem optimalen FSM aus Abschnitt 3.4 auf. Der optimale FSM wird erweitert, die Implementierung eines hierarchischen Zustands-automaten wird möglich. Bei dem Quantum HSM handelt es sich demzufolge um eine ähnliche Technik, wie sie bei dem optimalen FSM genutzt wird. Allerdings werden weitere `Pointer-to-Member` Funktionen benötigt, um die Hierarchie der Zustände realisieren zu können. Die komplette Lösung ist relativ aufwendig und sehr komplex. Da auch diese Variante keinen Einfluss auf die Lösung in dieser Arbeit hat, wird an dieser Stelle auf eine detaillierte Erläuterung verzichtet.

4 Anforderungen an ein State Pattern

Ein Zustands-Entwurfsmuster muss gewisse Anforderungen erfüllen damit es als Grundlage (Muster) für die Implementierung eines Zustandsautomaten genutzt werden kann. Die Anforderungen lassen sich in funktionale Anforderungen, strukturelle Anforderungen und Leistungsanforderungen gliedern. In diesem Kapitel führe ich die genannten Anforderungsgruppen auf und erläutere die Anforderungen. Die in den folgenden Abschnitten beschriebenen Anforderungen habe ich aus der Analyse der existierenden Zustandsmuster und der Definition der Statecharts aus [Harel87] entwickelt.

4.1 Funktionale Anforderungen

Zustandsautomaten besitzen viele Eigenschaften, von denen nicht alle zwingend in einem Zustands-Entwurfsmuster implementiert werden müssen. Zusätzliche Funktionen machen den resultierenden Automaten jedoch komfortabler. Je mehr Funktionen der Automat bereitstellt, umso dynamischer kann das Automatenmodell sein. Anders formuliert, die Implementation ist besser auf das vorhandene Automatenmodell anwendbar. Ich gliedere die funktionalen Anforderungen in zwingende und wünschenswerte Anforderungen. Eine minimale Implementation soll alle zwingenden Anforderungen erfüllen, um einen Zustandsautomaten korrekt zu realisieren. Die wünschenswerten Anforderungen sollten erfüllt werden, um einen Automaten so gut wie möglich zu implementieren. Es sollen letztlich alle Funktionen der Statecharts möglich sein.

A) Zwingende Anforderungen

Alle Anforderungen, die die Realisierung eines Zustandsautomaten erfüllen muss, zählen zu den zwingenden Anforderungen. Nur wenn diese Anforderungen erfüllt sind ist der Einsatz des resultierenden Automaten gerechtfertigt. Eine Bedingung, die die Realisierung eines Zustandsautomaten erfüllen muss ist in [GoF95 S.399] umschrieben. Dort heißt es, dass das Objekt sein Verhalten in Abhängigkeit vom aktuellen Zustand zur Laufzeit ändern können muss. Dies ist ein klassischer Anwendungsfall für den Einsatz des State Design Pattern. Die Reaktion auf ein Ereignis soll die Gleiche sein wie im Modell des Zustandsautomaten. Folgende Eigenschaften sollen dafür umgesetzt werden.

- **Reaktion auf Ereignisse**

  Es sollen alle möglichen Ereignisse, auf die der Zustand reagieren kann, vorhanden und definiert sein. Als Reaktion auf ein Ereignis soll eine Aktion

oder ein Zustandswechsel erfolgen. Die Kombination beider Möglichkeiten ist ebenfalls möglich.

- **Existenz aller Zustände**

  Alle Zustände sollen vorhanden sein und repräsentiert werden, um die auftretenden Ereignisse an diese weiterzuleiten und damit die Grundlage für die richtige Reaktion zu schaffen.

- **Zustandswechsel**

  Wenn als Reaktion auf ein Ereignis ein Zustandswechsel folgt, soll sichergestellt werden, dass dieser ordnungsgemäß vollzogen wird, das heißt, die Technik zur Durchführung des Zustandswechsels muss verlässlich funktionieren.

- **Datenhaltung**

  Die verschiedenen Zustände innerhalb eines Automaten arbeiten auf den gleichen Daten, sie lesen oder ändern diese. Deshalb ist es zwingend erforderlich, einen Mechanismus bereitzustellen, der es ermöglicht, unabhängig vom Zustand, auf die Daten des Automaten zuzugreifen.

Bei einer Erweiterung des Zustands-Entwurfsmuster zur Realisierung eines hierarchischen Automaten müssen zusätzlich zu den oben genannten Anforderungen die folgenden Anforderungen erfüllt werden.

- **Subzustände**

  Es soll ermöglicht werden, Zustände zu schachteln und den Zuständen damit erlauben, Subzustände zu besitzen.

- **Zustandswechsel aus einem Subzustand in einen anderen Subzustand**

  In einem hierarchischen Zustandsautomaten soll es möglich sein, aus einem Subzustand in einen anderen Subzustand zu wechseln und dabei die aktuellen Zustände (geschachtelt) der Reihe nach zu verlassen. Anschließend soll der neue Zustand vom obersten gemeinsamen Zustand aus betreten werden.

- **Verhaltensvererbung an Subzustände**

  Das Prinzip der Verhaltensvererbung ist ein wichtiger Bestandteil von hierarchischen Automaten und soll als ein Grundprinzip realisiert werden. Eine Erläuterung der Verhaltensvererbung bezüglich hierarchischer Automaten ist im Abschnitt 2.1.3 zu finden.

Es gibt die Möglichkeit einen hierarchischen Automaten parallel oder sequentiell zu betreiben. Zwischen diesen Möglichkeiten muss bereits vor der Realisierung gewählt werden.

- **Sequentieller Betrieb**

  Im sequentiellen Betrieb gibt es nur einen Thread, es kann immer nur einen aktiven Zustand geben.

- **Paralleler Betrieb**

  In dieser Variante des Betriebs können verschiedene Subzustände gleichzeitig betreten und Aktionen in ihnen gleichzeitig und nebeneinander ausgeführt werden. Diese Variante erfordert den Einsatz von mehreren Threads.

B) Wünschenswerte Anforderungen

Zusätzlich zu den zwingenden Anforderungen gibt es wünschenswerte Anforderungen, die zur Funktionalität eines Zustandsautomaten beitragen. Dazu gehören Aktionen, die beim Verlassen oder beim Betreten eines Zustandes ausgelöst werden. Bei einem Wechsel von einem Subzustand in einen anderen Subzustand eines anderen Zweiges in der Hierarchie, sollten auf dem Weg vom Quellzustand zum Zielzustand die jeweiligen `entry`- und `exit`-Aktionen ausgeführt werden. Der Einsatz von diesen Funktionen ist ebenso in flachen Zustandsautomaten wünschenswert.

- **Aktion beim Verlassen des Zustandes**

  Es soll in jedem Zustand eine Austrittsfunktion (exit) vorhanden sein.

- **Aktion beim Betreten des Zustandes**

  Es soll in jedem Zustand eine Eintrittsfunktion (entry) vorhanden sein.

Bei der Verwendung eines hierarchischen Automaten ist es sinnvoll, eine Gedächtnisfunktion zu benutzen, da diese in den Statecharts definiert wird. Diese Funktion führt zur Erweiterung der `exit-` und `init-`Funktionen. Beim Verlassen eines Zustandes soll die Gedächtnisfunktion des direkten Superzustandes aufgerufen werden, damit sich dieser den letzten aktiven Subzustand merken kann. Bei der Initialisierung eines Zustandes, sollte geprüft werden, ob sich der aktive Zustand bereits einen schon zuvor betretenen Subzustand gemerkt hat, oder ein Standardzustand zum Betreten definiert ist. Zudem sollten alle Zustände mit einer Initialfunktion initialisiert werden.

- **Gedächtnis im Zustand**

  Die Gedächtnisfunktion sollte in jedem Zustand, der Subzustände besitzt, vorhanden sein. Außerdem besagt die Definition der Statecharts, dass beim Vorhandensein eines History-Zustandes die Initialaktion nicht ausgeführt wird.

- **Initialisierung der Zustände**

  Beim ersten Betreten des Zustandes ist es sinnvoll, diesen durch Ausführung einer Initialaktion zu initialisieren.

4.2 Strukturelle Anforderungen (Modifikationen)

Der Sinn eines Entwurfmusters ist es, eine einfache und elegante Lösung für eine spezifische Anwendung zu beschreiben, so schreibt es E. Gamma in [GoF95 S.1]. Dem Benutzer soll es möglich sein, das Prinzip, welches hinter dem Muster steht, einfach anzuwenden. In diesem konkreten Fall soll das Zustands-Entwurfsmuster dem Entwickler helfen, das entworfene Zustandsautomatenmodell zu implementieren.

Daneben soll es möglich sein, bei Veränderungen im Automatenmodell den bestehenden implementierten Automaten anzupassen. Die erforderlichen Änderungen sollten möglichst gering, unkompliziert und an möglichst wenig Stellen vorzunehmen sein.

Des Weiteren soll der Benutzer des Musters in der Lage sein, beliebig viele Zustände, Ereignisse und Aktionen zu definieren und zu benutzen. Sollte es an dieser Stelle zu Einschränkungen kommen, kann es sein, dass ein komplexes Modell eines Zustandsautomaten nicht mit dem Entwurfsmuster umgesetzt werden kann.

Ebenfalls gehört die Einhaltung von Programmierstandards, wie zum Beispiel die Kapselung von Klassen, zu den Anforderungen, die es zu erfüllen gilt.

Folglich sollen folgende Anforderungen durch ein Zustands-Entwurfsmusters erfüllt werden.

- **einfache Benutzung (Umsetzung)**
- **leichte Verständlichkeit**
- **einfache Veränderung des Verhaltens**
- **Erstellung von Zuständen, Ereignissen und Aktionen beliebiger Anzahl für Automaten beliebiger Komplexität**
- **Einhaltung von Programmierstandards**

4.3 Leistungsanforderungen

Bei dem Einsatz eines Zustandsautomaten werden nicht selten zeit- oder speicherkritische Anwendungen implementiert. Speziell in Echtzeitsystemen ist die Geschwindigkeit der Ausführung eines Programms ein wichtiger Faktor. Zudem steht in Echtzeitsystemen, aufgrund ihres Einsatzortes, des zur Verfügung stehenden Platzes und der Energie, oft nur wenig Speicher zur Verfügung. Diese Faktoren sollen beim Entwurf des Zustands-Entwurfsmusters ebenso beachtet werden wie die strukturellen und funktionalen Anforderungen.

Um den Speicher zu schonen, ist es zweckmäßig, die unnötige Erzeugung von Instanzen einer Klasse zu vermeiden und diese erst zu erzeugen, wenn sie benötigt werden.

Zusätzlich sollten unnötige Zustandsübergänge vermieden werden. Ein weiterer Aspekt zur Effizienz der Laufzeit ist die Verarbeitung der Ereignisse, wie in [Samek02 S.26-27] beschrieben. Wenn in einem Automaten viele Zustandswechsel erfolgen und diese viel Laufzeit benötigen, hat dies eine hohe Gesamtlaufzeit zur Folge. Deshalb ist bei der angewandten Technologie für den Zustandswechsel darauf zu achten, dass sie eine möglichst kurze Ausführungszeit mit sich bringt.

Demzufolge sollen folgende Grundprinzipien bei der Erstellung eines Zustands-Entwurfsmusters beachtet werden.

- **Schonung des Speichers**
- **Laufzeit, besonders bei der Ereignisverarbeitung, möglichst gering halten**

5 Design – Implementation

Der Kern des entwickelten Zustands-Entwurfsmusters ist die verwendete Technologie zur Umsetzung der Wechsel zwischen den Zuständen. Für die Durchführung des Zustandswechsels wird der in Abschnitt 2.3.1 erläuterte Operator `Placement-new` genutzt. Grundsätzlich ist das neue Entwurfsmuster an das State Design Pattern der GoF aus Abschnitt 3.3 angelehnt. Ich habe jedoch einige Modifikationen und Erweiterungen vorgenommen.

Vorerst habe ich ein Muster zum Erstellen von flachen endlichen Automaten entwickelt, um die zwingenden Anforderungen eines Zustands-Entwurfsmusters zu erfüllen. Nach der erfolgreichen Fertigstellung dieses minimalen Musters habe ich es nach und nach erweitert. Dadurch entstand ein Entwurfsmuster für hierarchische Automaten, welches zudem die History-Funktionalität unterstützt. Zeitgleich ist mir die Erstellung eines Quellcodegenerators gelungen, mit dem es möglich ist, aus einer XML-Datei einen hierarchischen Automaten zu erzeugen. In dieser XML-Datei wird das Modell des zu implementierenden Automaten definiert. Die Erstellung des Automaten mit dem Quellcodegenerator erfolgt nach dem hier entwickelten Entwurfsmuster. Der Generator hat die Aufgabe, exemplarisch zu zeigen, dass es ohne großen Aufwand möglich ist, einen Generator zur automatischen Codeerstellung nach diesem Entwurfsmuster, zu implementieren.

5.1 Grundprinzip der Technologie

Wie auch im State Design Pattern von Gamma angewandt, wird jeder Zustand durch eine Klasse dargestellt. Die Ereignisse werden unter Benutzung von Funktionen realisiert. Für jedes Ereignis ist eine Funktion nötig. Weiterhin besteht eine Kontextklasse, die einen Zeiger auf eine abstrakte Zustandsklasse hält. Von dieser abstrakten Zustandsklasse werden alle konkreten Zustände abgeleitet. Dadurch kann jedes auftretende Ereignis direkt von der Kontextklasse an den konkreten Zustand delegiert werden. Die Reaktion und der eventuelle Zustandswechsel finden im konkreten Zustand statt. Damit das Ganze sicher funktioniert, müssen in der Kontextklasse und in der abstrakten Zustandsklasse, die gleichen Ereignisfunktionen deklariert werden. Die eigentliche Ereignisverarbeitung findet folglich unter Verwendung von Laufzeitpolymorphie und Delegation statt. Eine detaillierte Erläuterung des State Design Pattern der GoF und das zugehörige Klassendiagramm sind dem Kapitel 3.3 zu entnehmen.

Der wesentliche Unterschied zum State Design Pattern der GoF besteht im Übergang von einem Zustand zu einem anderen Zustand. Bei dieser Technik wird der Zustands-

wechsel direkt vollzogen, während das State Design Pattern dafür eine Funktion vorsieht. Wie bereits erwähnt nutze ich den Operator `Placement-new`, um den Zustand zu wechseln. Mit dessen Hilfe kann die Adresse, auf die der `this`-Zeiger des aktiven Zustandes zeigt, überschrieben werden. Ein Quellcodebeispiel für die Nutzung des Operators `Placement-new` für einen Zustandswechsel ist in der Abbildung 5-1 dargestellt. Der `this`-Zeiger enthält den Zeiger auf die Tabelle der virtuellen Funktionen (`*vtbl`), welcher die ersten 4 Byte des Objektes belegt. Jede Klasse besitzt eine virtuelle Funktionstabelle, falls sie virtuelle Funktionen besitzt. In dieser Tabelle werden den virtuellen Funktionszeigern (`*pfct`) die Adressen der entsprechenden Realisierungen zugeordnet. Die Funktionsaufrufe werden auf diese Art und Weise an die jeweilige Spezialisierung weitergegeben. Als Resultat sind die virtuellen Funktionszeiger auf den entsprechend ausführbaren Quellcode im Codesegment gerichtet. Eine Zeichnung zum Verständnis der Thematik befindet sich in der Abbildung 5-2. Der gestrichelte Pfeil in der Abbildung stellt den Zeigerwert nach dem Wechsel dar. Wenn der Zeiger auf die virtuelle Funktionstabelle mit einer neuen Adresse überschrieben wird, kann das mit einem Wechsel in den neuen Zustand gleichgestellt werden. In diesem Fall ist diese Adresse, die Adresse der virtuellen Funktionstabelle eines anderen Zustandes. Nach dem Überschreiben der Adresse, werden alle Funktionsaufrufe an die virtuelle Funktionstabelle des neuen Zustandes delegiert. Die dort enthaltenen virtuellen Funktionszeiger zeigen auf den ausführbaren Code des neuen Zustandes.

```
#include „StateOld.h"
#include „StateNew.h"

void StateOld::change2New()
{
   new (this) StateNew;
}
```

Abbildung 5-1: Beispiel Zustandswechsel mit Operator Placement-new

Wie zuvor beschrieben, entsprechen die ersten 4 Byte eines Objektes dem Zeiger auf die virtuelle Funktionstabelle. Beim Anlegen des ersten Zustandes werden lediglich 4 Byte auf dem dynamischen Speicher (Heap) reserviert, weil ein konkreter Zustand keine Daten, sondern nur virtuelle Funktionen enthält. Das bedeutet, dass der Speicherbedarf trotz zunehmender Anzahl von Zuständen gleich bleibt. Die explizite Verwendung eines Destruktors, wie in Abschnitt 2.3.1 beschrieben, ist nicht nötig, da immer auf dem gleichen Speicherbereich gearbeitet wird. Der Zustandswechsel erfolgt sehr schnell, weil es sich hierbei um nur einen Aufruf handelt. Da der Speicherbereich bereits beim Start des Automaten reserviert wurde, findet keine weitere Reservierung statt und der Vorgang nimmt so keine weitere Zeit in Anspruch.

Diese Variante des Zustandswechsels ist infolgedessen speicherschonend, leistungsfähig und sicher zu gleich. Folglich werden durch diese Technik nicht nur die Leistungsanforderungen an ein Zustands-Entwurfsmusters, sondern auch die funktionalen Anforderungen bezüglich des Punktes Zustandswechsel erfüllt.

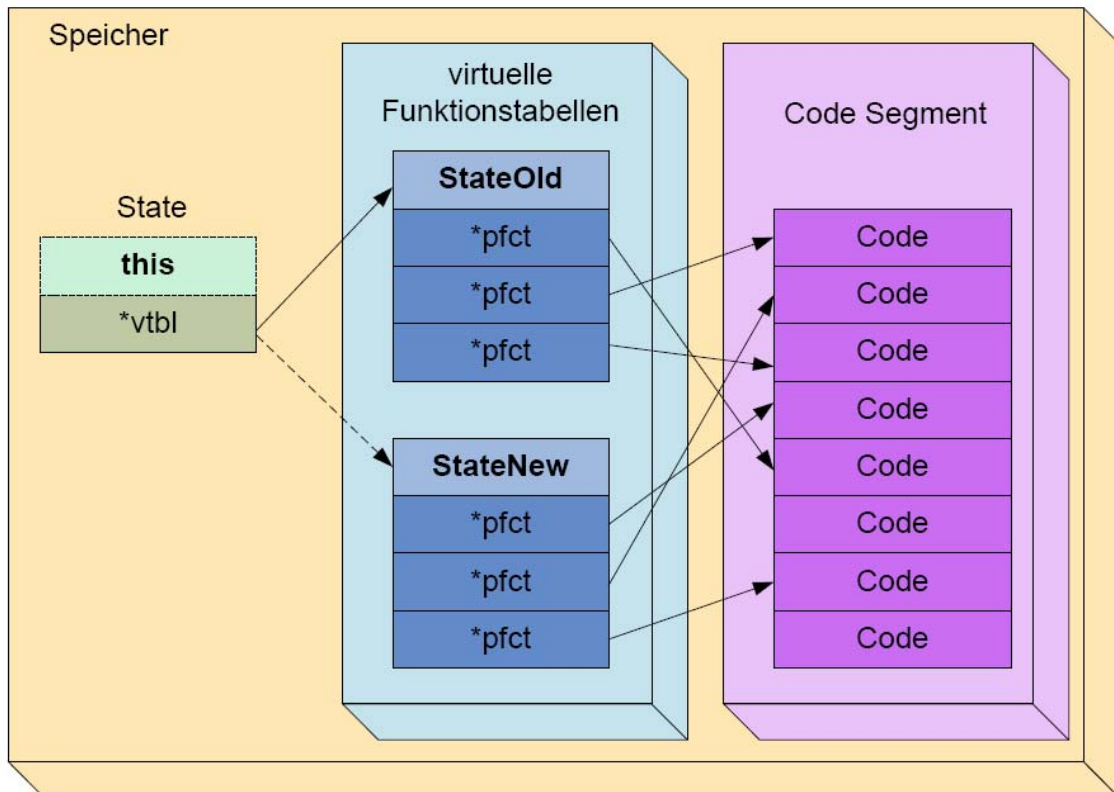

Abbildung 5-2: virtuelle Funktionstabellen

Der einfachste Fall ist ebenfalls in dem Beispiel aus Abbildung 5-1 implementiert. Der aktive Zustand ist `StateOld`, mit dem Ereignis `change2New` soll ein Zustandswechsel von `StateOld` nach `StateNew` vollzogen werden. Mit Hilfe des Operators `Placement-new` wird dem Zeiger (`this`), der auf die virtuelle Funktionstabelle zeigt, ein neuer Wert zugewiesen. Dieser Wert stellt die Adresse der virtuellen Funktionstabelle des Zustandes `StateNew` dar. Zur Bekanntmachung der Klasse `StateNew` in der Klasse `StateOld` genügt es, die Header-Datei der Klasse `StateNew` mit einzubinden.

5.2 Flacher endlicher Zustandsautomat (FSM)

Bei der prototypischen Entwicklung des Zustands-Entwurfsmusters habe ich zunächst einen flachen endlichen Zustandsautomaten aufgebaut. Dieser eignete sich ausgezeichnet dazu, die Zustandsübergänge an einem konkreten Beispiel zu testen.

Zudem diente er der Weiterentwicklung des Musters in Punkto Struktur und Datenhaltung.

Für die Entwicklung des flachen Automaten wählte ich ein einfaches Beispiel. Es sollte nicht zu komplex sein, jedoch mehrer Zustände, Ereignisse und damit verbundene Reaktionen beinhalten. Für den ersten Entwicklungsschritt eignete sich das Modell des Zustandsautomaten aus Abbildung 5-3. Dieser Zustandsautomat wurde bereits in der Veröffentlichung „*On the Implementation of Finite State Machines*“ [GurBos99] verwandt.

Der abgebildete Automat dient zum zeilenweisen Einlesen von Zeichen (Text). Es wird Zeichen für Zeichen eingelesen. Der Startzustand ist `Empty`, sobald das erste Zeichen eingelesen wurde, wechselt der Automat von `Empty` nach `Collect`, dazu dient das Ereignis `feedChar`. In diesem Zustand verweilt der Automat, bis er achtzig Zeichen gelesen hat und als Signal dafür das Ereignis `eol` (Zeilenende erreicht) empfängt. Es wird schließlich der Zustand `Full` betreten, indem der Automat sich solange aufhält, bis das Ereignis `release` (neue Zeile beginnen) auftritt. Der Automat gelangt wieder in den Startzustand `Empty` und kann eine neue Zeile eingelesen. Um diese Funktionalität herzustellen, müssen eingelesene Zeichen gespeichert werden. Zudem muss es einen Zählmechanismus geben, um festzustellen wie viele Zeichen bereits eingelesen wurden. Diese Daten werden in der Regel von allen Zuständen benötigt, deshalb müssen sie in der Kontextklasse gehalten werden.

In diesem Modell sind alle Punkte vorhanden, die zu den zwingenden funktionellen Anforderungen eines Zustandsmusters zählen. Auf die weiteren Anforderungspunkte wird in den folgenden Abschnitten eingegangen.

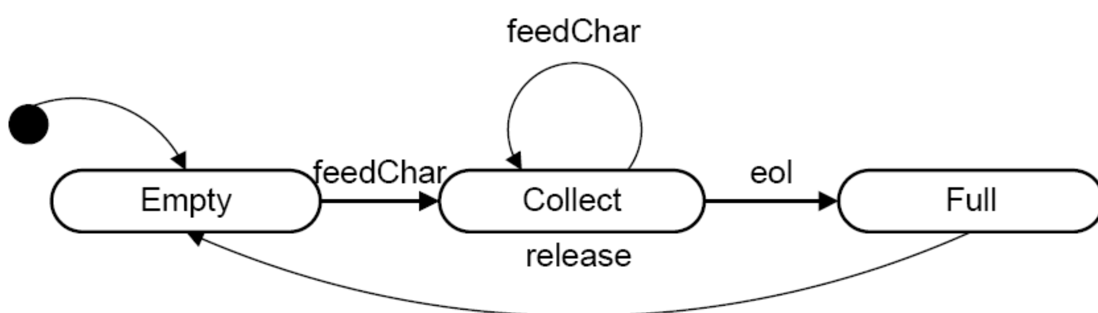

Abbildung 5-3: Zustandsautomat für ersten Entwicklungsschritt

5.2.1 Struktur

Die Struktur entspricht in den Grundzügen der Struktur des State Design Pattern der GoF, dessen Hauptprinzip die Verwendung von Laufzeitpolymorphie für die Reaktion auf Ereignisse ist. Ich habe hierbei das Grundprinzip aus Abschnitt 5.1 angewandt, welches eine Struktur nach dem Klassendiagramm in Abbildung 5-4 zur Folge hat. Auffällig ist, dass in den konkreten Zuständen nur die für diesen Zustand relevanten Ereignisse (Funktionen) implementiert werden. Wenn für einen Zustand ein Ereignis aufgerufen wird, obwohl es nicht definiert ist, wird das Standardverhalten, wie es in der Klasse `State` definiert ist, aufgerufen. Es ist erkennbar, dass in der Klasse `Context` zusätzlich zu dem Zeiger auf eine `State`-Instanz, weitere Attribute gehalten werden. Die Klasse `Context` entspricht hierbei der Kontextklasse des Musters. Die Attribute werden für die Datenhaltung benötigt. Es werden Templates genutzt, um die Datenhaltung zu ermöglichen. Die Funktionsweise der Datenhaltung wird in Abschnitt 5.2.3 detailliert erläutert.

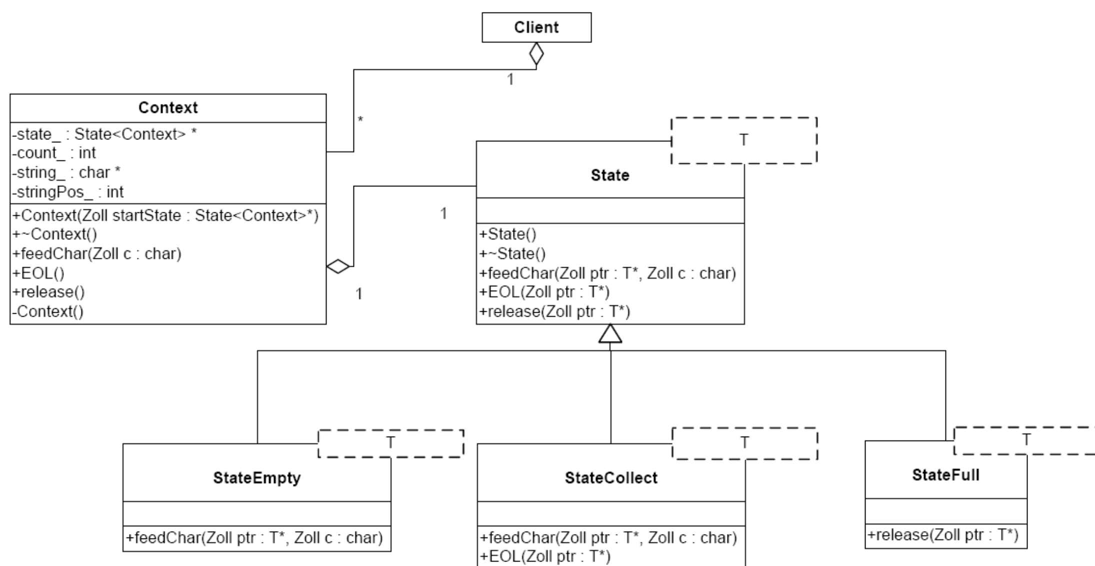

Abbildung 5-4: Klassendiagramm für den konkreten flachen endlichen Automaten

Die Reaktionen auf Ereignisse werden durch Funktionen realisiert. Dazu müssen die Ereignisse durch den Anwender des Automaten erkannt und weitergegeben werden. Der Anwender muss lediglich eine Instanz der Kontextklasse erzeugen, mit deren Hilfe er die Ereignisse an den Zustandsautomaten delegiert (Funktionsaufruf). Dies erfolgt analog zu der vorgestellten Variante der GoF. Da hier ebenfalls für jeden Zustand eine Klasse erstellt wird, ist es unproblematisch beliebig viele Zustände zu nutzen. Bei der Erzeugung der Instanz der Kontextklasse wird als Parameter der zuvor erstellte Startzustand benötigt. Der Default-Konstruktor ist als `private` deklariert und wird dadurch vor unerlaubter Nutzung geschützt.

Für jede Klasse werden eine Header-Datei, eine CPP-Datei und eine TPP-Datei verwendet. Die TPP-Datei enthält die Funktionsrümpfe der Template-Funktionen und somit die Implementation der Aktionen und der Zustandswechsel. Wenn in einer Funktion die Template-Funktionalität genutzt wird, muss diese in der TPP-Datei definiert werden. Die Definition der übrigen Funktionen, sofern andere Funktionen vorhanden sind, erfolgt in der CPP-Datei. In der Header-Datei wird wie üblich die Zustands-Klasse definiert. Durch die Benutzung der Template-Funktionalität muss eine Besonderheit beachtet werden, die in der TPP-Datei implementierten Funktionen werden extra am Ende der Header-Datei eingebunden (Bsp. Abbildung 5-5).

```
template <class T>
class StateEmpty : public State<T>
{
   . . .
};

#include "StateEmpty.tpp"

#endif
```

Abbildung 5-5: Beispiel Einbinden einer TPP-Datei

Alle Funktionen sind mit dem Parameter `t` vom Typ `T*` ausgestattet, damit es möglich ist in jeder Funktion auf die Daten der Kontextklasse zuzugreifen (siehe Abschnitt 5.2.3) und den Parameter erneut weiterzugeben. Der Parameter `ptr` entspricht dem `this`-Zeiger des Objektes der Kontextklasse. Der Zeiger wird jeweils bei der Delegation des auftretenden Ereignisses, von der Kontextklasse an die konkrete Zustandsklasse weitergegeben. In der Abbildung 5-6 ist ein Beispiel für die Delegation zu sehen. Der übergebene `this`-Zeiger wird in den konkreten Zuständen als Parameter `t` gehandelt.

```
void Context::EOL()
{
   state_->EOL(this);
}
```

Abbildung 5-6: Quellcodeausschnitt Delegation Ereignis Kontext zu Zustand

5.2.2 Reaktion – Zustandsübergang

Als Reaktion auf ein aufgetretenes Ereignis soll eine Aktion, ein Zustandswechsel oder auch beides auftreten. Allgemein muss bei der Reaktion in einem flachen endlichen Automaten zwischen zwei Fällen unterschieden werden. Die Erläuterung der Reaktionen erfolgt beispielsweise in dem in Abbildung 5-3 dargestellten Automaten.

- **der Zielzustand ist ein anderer als der aktive Zustand**

  Je nach Definition des Zustandsautomaten sollte eine Aktion ausgeführt werden. Eine Aktion befindet sich in Form von ausführbarem Code direkt in der Ereignisfunktion. Nach Ausführung der Aktion kann der Zustandswechsel vollzogen werden. In dem hier verwandten Beispiel tritt das Ereignis `EOL` auf, während der Zustand `Collect` aktiv ist. Es wird eine Aktion ausgeführt und in den Zustand Full gewechselt. Die Aktion in diesem Beispiel beschränkt sich auf eine Ausgabe in der Kommandozeile.

```
template <class T>
void StateCollect<T>::EOL(T* ptr)
{
   cout << "sig EOL" << endl;  // Aktion
   new (this) StateFull<T>; // Zustandswechsel
}
```

  Abbildung 5-7: Quellcodeausschnitt Aktion + Zustandswechsel im FSM

- **der Zielzustand ist der aktive Zustand**

  In diesem Fall wird kein Zustandswechsel vollzogen, weil er für die Funktionalität unnötig wäre. Es genügt die definierte Aktion auszuführen, dies erfolgt wie im vorangegangenen Fall.

5.2.3 Datenhaltung mit Template-Klassen

Indem die Daten des Zustandsautomaten in der Kontextklasse gehalten werden, können sie von allen Zuständen genutzt werden. Um die Daten für die Zustände zugängig zu machen, bedarf es Zugriffsfunktionen in dieser Klasse. Zugriffsfunktionen gehören nicht zu dem Muster, weil sie, je nach Problem und den erforderlichen Daten, unterschiedlich sind. Daher unterliegen sie der Verantwortung des Benutzers des Musters.

An sich ist es unproblematisch, gemeinsame Daten in der Kontextklasse verfügbar zu machen. Zwischen den Klassen ist bereits eine Bindung von der Kontextklasse in Richtung Zustände vorhanden, deshalb sollte eine weitere Kopplung verhindert werden. Zur Lösung dieses Problems schien die Verwendung von Templates geeignet. Diese Möglichkeit für den Einsatz in Zustandsautomaten ist in der Literatur in dieser Weise nicht zu finden. Aus diesem Grund musste diese Idee zunächst von mir ausgearbeitet werden.

Reine Template-Funktionen sind für die Lösung dieses Problems nicht geeignet, weil sie nicht virtuell sein können (siehe 2.3.2). Die Zustandsklassen werden als Template-Klassen (Vorlagenklassen) genutzt, wie in Abschnitt 2.3.2 beschrieben. Unter Verwendung dieser Technik, muss den Zustandsklassen nicht bekannt sein, um welchen

Typ es sich bei der Kontextklasse handelt. Sie arbeiten ausschließlich mit einem Zeiger vom Typ des Vorlagenparameters. Mit dem Zeiger kann der Zugriff auf die Daten erfolgen. Im unten aufgeführten Beispiel ist der Vorlagenparameter vom Typ `T`, infolgedessen kann mit dem Zeiger `ptr` auf die Funktionen der Kontextklasse zugegriffen werden.

```
template<class T>
void StateEmpty<T>::feedChar(T* ptr, char c)
{
     cout << "SIGNAL feedChar: " << c << endl;
     ptr->incrementCount();
     char* string = ptr->getString();
     ptr->setStringPos(0);
     string[0] = c;
     ptr->incrementStringPos();
     new (this) StateCollect<T>;  //Zustandswechsel
}
```

Abbildung 5-8: Quellcodeausschnitt Anwendung Template für Datenzugriff

In der Kontextklasse ist bei der Definition des Zustandszeigers auf die Angabe des Vorlagenparameters zu achten. Dabei wird der Typ der Kontextklasse selbst als Vorlagenparameter benutzt. Ein Beispiel für die Erzeugung des Zeigers ist in Abbildung 5-9 zu finden.

```
class Context
{
   . . .
private:
   // Zeiger auf den aktuellen Zustand
   State<Context>* state_;
   . . .
};
```

Abbildung 5-9: Quellcodeausschnitt Definition Zeiger mit Vorlagenparameter

Bei der Instantiierung des ersten konkreten Zustandes (Startzustand) wird dem Zeiger `state_` die Adresse des konkreten Zustandes im Heap zugewiesen. Als Vorlagenparameter wird der Typ der Kontextklasse selbst genutzt. In dem aufgeführten Beispiel (Abbildung 5-10) ist `Context` der Typ (Klassenname) des Vorlagenparameters.

```
Context::Context(State<Context>* startState)
:state_(startState)
,count_(0)
,string_(new char[80])
,stringPos_(0)
{}
```

Abbildung 5-10: Quellcodeausschnitt Instantiierung mit Vorlagenparameter

Wenn ein Zustandwechsel verübt wird, ist die Angabe eines Vorlagenparameters ebenso nötig wie bei der Instantiierung. In diesem Fall wird ebenfalls ein Typ als Vorlagenparameter angegeben, doch der Typ der Kontextklasse ist nicht bekannt. So wird als Typ der Typ des Template-Parameters angegeben. In der Abbildung 5-8 wird ein Zustandswechsel unter Angabe des Vorlagenparameters `T` durchgeführt. Der Parameter wurde bereits bei der Definition der Klasse, in der Header-Datei, festgelegt.

5.3 Ansatz eines hierarchischen Zustandsautomaten (HSM)

Nach der Fertigstellung des Zustandsmusters für einen flachen Zustandsautomaten, war das nächste Ziel die Entwicklung eines Musters für die Realisierung eines hierarchischen Automaten. In dieser Erweiterung habe ich die wünschenswerten funktionellen Anforderungen aus Kapitel 4.1 realisiert. Das zuvor vorgestellte Muster für die Erstellung eines flachen Automaten konnte als Grundlage für diese Erweiterung genutzt werden, um die gewonnenen Erkenntnisse weiter zu verwenden. Deshalb wird in diesem Abschnitt ausschließlich auf die Unterschiede und auf die Erweiterungen zum flachen Automaten eingegangen.

Zunächst bedarf es eines Modells für einen hierarchischen Zustandsautomaten, an dem alle funktionalen Anforderungen nachvollzogen werden können. Der Automat muss beliebige Zustände, und diese wiederum Subzustände besitzen, um alle Möglichkeiten der Zustandsübergänge darzustellen. Für diesen Punkt ist es zusätzlich notwendig, dass sich in einem Zustand mehrere Subzustände befinden und genügend Ereignisse vorhanden und definiert sind. Der hierarchische Automat in Abbildung 5-11 erfüllt die genannten Bedingungen und war als Grundlage und als Beispiel für die Entwicklung des erweiterten Entwurfsmusters geeignet. Bei diesem Automatenmodell handelt es sich um ein erweitertes Beispiel aus dem Buch *„Practical Statecharts in C/C++"* [Samek02 S.95]. Ich habe das Modell um einen zusätzlichen Zustand, weitere Ereignisse und die flache Gedächtnisfunktion erweitert. In dem Automaten wird kein Verhalten aus einer bestimmten Anwendung dargestellt, er dient ausschließlich zur Veranschaulichung der Problematik. Obwohl das Verhalten relativ komplex ist, ist der Automat aufgrund der einfachen Zustandsnamen und Ereignisbezeichnung leicht verständlich.

Der abgebildete Automat verfügt über sieben Zustände. Der oberste Zustand (Top-State) ist `S0`. Alle weiteren Zustände sind Subzustände von `S0` und erben somit das Verhalten von `S0`. Des Weiteren gibt es acht Ereignisse (a-h). Diese Ereignisse lösen die Reaktionen in Form von Zustandswechseln und Aktionen aus. Alle Zustände können betreten und wieder verlassen werden. Daneben sind in dem Modell `entry`- und `exit`-Aktionen integriert. Eine weitere wünschenswerte Anforderung ist die Ausführung einer Initialfunktion beim Betreten eines Zustandes. Diese Funktionalität ist in dem verwendeten Modell ebenso wie die Gedächtnisfunktion vorhanden. Es wird die flache Gedächtnisfunktion genutzt. Nur bei Zuständen, die mehrere Subzustände auf der

gleichen Ebene haben, ist die flache Gedächtnisfunktion sinnvoll. Das genaue Verhalten des Beispielautomaten kann aus dem Modell entnommen werden.

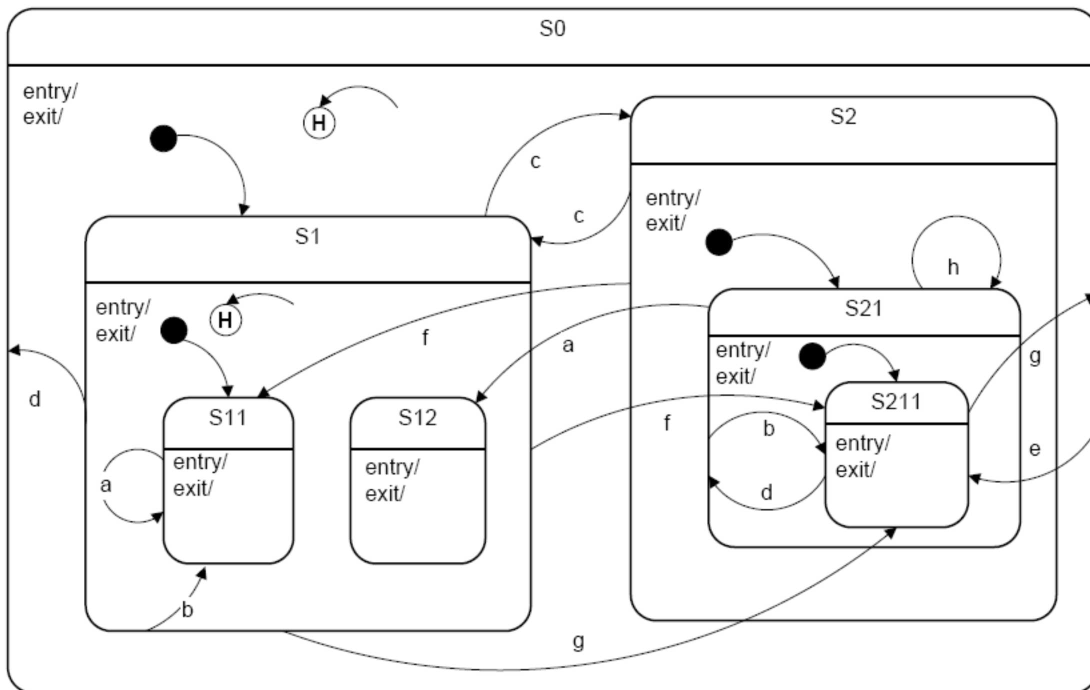

Abbildung 5-11: Hierarchischer Zustandsautomat für zweiten Entwicklungsschritt

5.3.1 Struktur

Zusätzlich zur Struktur des zugrunde liegenden Zustands-Entwurfsmusters für flache Automaten werden die Subzustände von anderen konkreten Zuständen abgeleitet. Der oberste Zustand wird immer von der abstrakten Zustandsklasse abgeleitet. Die Klasse, die den obersten Zustand darstellt, ist die einzige Klasse, die direkt von der abstrakten Zustandsklasse erbt. Alle weiteren Zustände erben von dieser Klasse oder von bereits abgeleiteten Zustandsklassen. Für ein besseres Verständnis ist diese Struktur in dem Klassendiagramm, in der Abbildung 5-13 abgebildet. Im Klassendiagramm wird die Analogie zur Schachtelung der Zustände deutlich. Im Speziellen bedeutet dies, dass jeder Subzustand von seinem direkten Superzustand abgeleitet ist. Daher wird nicht nur die Hierarchie eingehalten, sondern auch das Verhalten (Vererbung) an die Subzustände weitergegeben. Auf diese Art und Weise verhält sich jeder Subzustand wie sein Superzustand, zuzüglich des eigenen Verhaltens. Wenn ein aktiver Zustand selbst keine Reaktion auf ein bestimmtes Ereignis vorsieht, kann es jedoch sein, dass einer seiner Superzustände dies tut. Die detaillierte Funktionsweise der Verhaltensvererbung ist dem Abschnitt 2.1.3 zu entnehmen.

Um eine korrekte Funktion der exit-Funktionalität auszuführen, muss jeder Subzustand explizit durchlaufen werden, bis der Zielzustand erreicht ist. Wenn ein Zustand nicht auf ein Ereignis reagieren kann, dafür aber einer der Superzustände, reicht es nicht, direkt die Reaktion des reaktionsfähigen Zustandes auszuführen. Es soll zuvor die exit-Funktion jedes Zustandes bis zum reagierenden Zustand ausgeführt werden, einschließlich der Reaktion des Zustandes, der reagieren kann. Diese funktionale Anforderung erfordert das Definieren aller Ereignisfunktion in jedem Zustand. Das bedeutet, alle Zustände müssen die Ereignisfunktion von jedem Ereignis implementieren. Dies trifft ebenso zu, wenn der Zustand laut Modell nicht auf das Ereignis reagiert. Wenn ein Zustand auf ein bestimmtes Ereignis nicht reagiert, muss trotzdem die exit-Funktion ausgeführt werden. Danach kann der Wechsel in den Superzustand durchgeführt werden, um dort eine mögliche Reaktion auf das eingetretene Ereignis auszuführen.

Im Beispiel aus der Abbildung 5-12 ist das Verhalten des Zustandes S12 beim Auftreten des Ereignisses sigE (e) sichtbar. Im Modell ist keine Reaktion dieses Zustandes auf das Ereignis e vorgesehen, aus diesem Grund muss zuerst in den Superzustand gewechselt werden. Zunächst wird die exit-Funktion aufgerufen, welche in jedem Zustand definiert ist. Nach dem Wechsel in den Superzustand wird das Ereignis sigE erneut aufgerufen. Aufgrund des vorherigen Zustandswechsels reagiert dann der Superzustand von S12, der Zustand S1, auf das Ereignis. Der Aufruf von sigE(t) wird an die virtuelle Funktionstabelle von der Klasse StateS1 delegiert, weil an der Adresse, auf die der this-Zeiger gerichtet ist, die Adresse der virtuelle Funktionstabelle von der Klasse StateS1 steht.

```
template <class T>
void StateS12<T>::sigE(T* t)
{
   exit(t);
   new (this) StateS1<T>;
   sigE(t);
}
```

Abbildung 5-12: Quellcodeausschnitt Reaktion im Hierarchischen Automaten

Die Funktionen zur Realisierung von Eintritts-, Austritts- und der Gedächtnisfunktion sind in jedem Zustand implementiert. Zusätzlich enthalten alle konkreten Zustände eine Funktion zur Initialisierung des Zustandes, die während dem Betreten des Zustandes ausgeführt wird. Es ist möglich, ein Ereignis zu empfangen auf das keiner der Zustände in der Hierarchie reagieren kann. Das kann bei der Ausführung jedoch erst im obersten Zustand festgestellt werden. Wenn sich der Automat im obersten Zustand befindet, wurden bereits alle Zustandswechsel und die dazugehörigen Austrittsfunktionen durchgeführt. Der Benutzer muss auf dieses Fehlverhalten hingewiesen werden, um Folgefehler zu verhindern. Für diesen Zweck steht die Funktion failure zur

Verfügung. Im einfachsten Fall genügt es, den Fehler auf der Kommandozeile auszugeben. Für die Meldung dieses Fehlers genügt es, die Funktion `failure` im obersten Zustand zu implementieren.

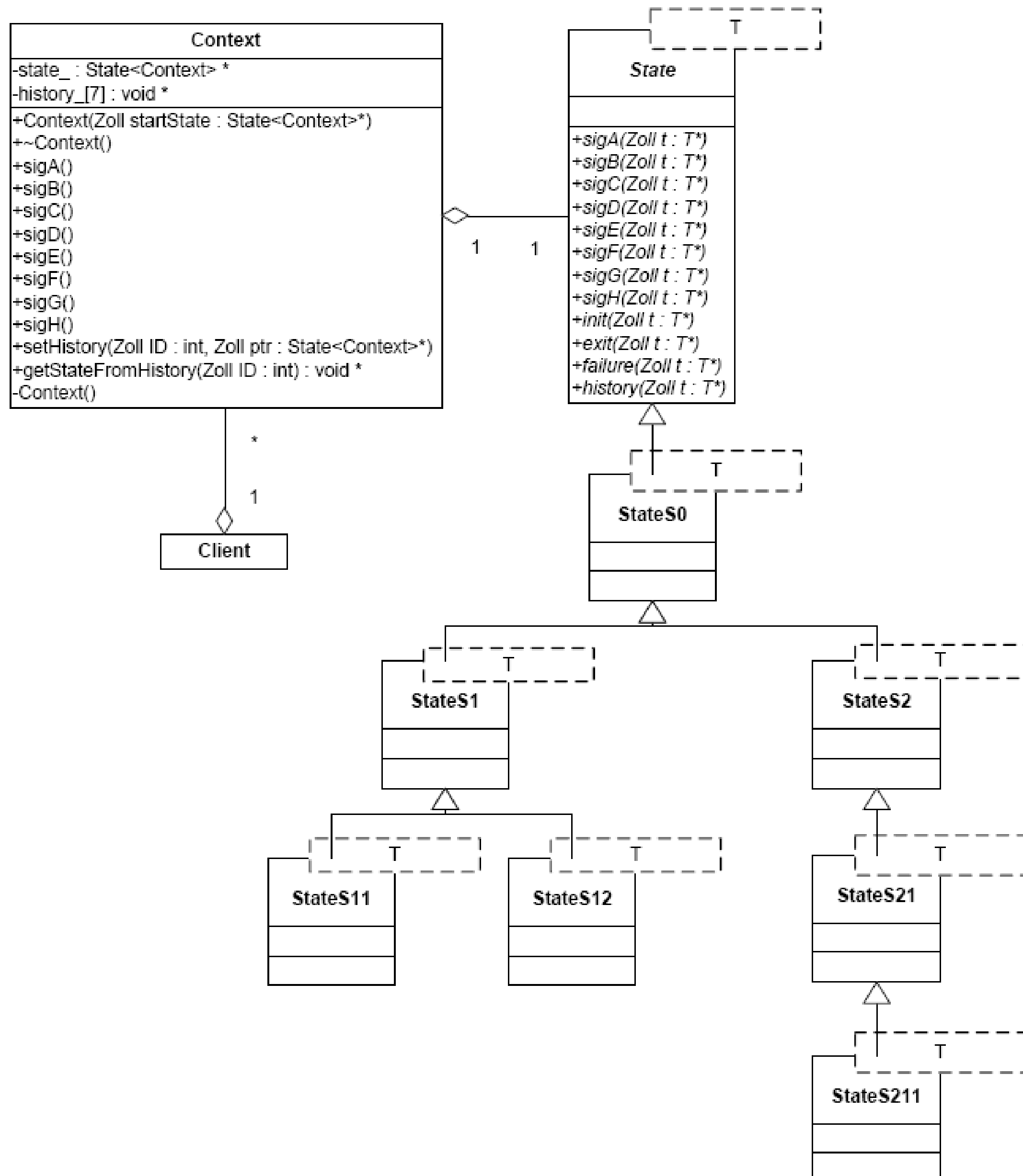

Abbildung 5-13: Klassendiagramm für konkreten hierarchischen Automaten

5.3.2 Zustandsübergänge

Aus den wünschenswerten funktionalen Anforderungen an einen Zustandsautomaten (Abschnitt 4.1) ist zu entnehmen, dass beim Verlassen eines Zustandes die `exit`-Funktion des zu verlassenden Zustandes ausgeführt werden muss. Anschließend darf

der Zustandswechsel vollzogen werden. Beim Betreten eines Zustandes soll die `entry`-Funktion, und somit die Initialfunktion, dieses Zustandes ausgeführt werden. In der Initialfunktion können zustandsspezifische Einstellungen vorgenommen werden. Durch Benutzung dieser Funktion kann in jedem Zustand ein Initialzustand, gemäß den Anforderungen, definiert werden. Dabei tritt folgende Besonderheit auf. Wenn ein Zustand sich mit Hilfe der Gedächtnisfunktion bereits einen vorherigen Subzustand gemerkt hat, wird dieser direkt betreten. Die zustandsspezifischen Einstellungen im zuletzt aktiven Zustand werden in diesem Fall nicht vorgenommen. Unabhängig von dem Gedächtnis des Zustandes, ist auf das Aufrufen der Initialfunktion des Folgezustandes zu achten. In dem aufgeführten Beispiel in Abbildung 5-14 ist die Initialfunktion von Zustand `S1` abgebildet. Egal ob der neue Zustand ein Initialzustand oder History-Zustand ist, die Initialfunktion wird aufgerufen.

```
template <class T>
void StateS1<T>::init(T* t)
{
  cout << "init S1" << endl;
  void* history =
  t->getStateFromHistory_(StateS0<T>::StateID::StateS1_ID);
  if(history != 0)
  {
    memcpy(this, &history, 4);
    cout << "back to history" << endl;
  }
  else
  {
    new (this) StateS11<T>;
    entry(t);
  }
  init(t);
}
```

Abbildung 5-14: Quellcodeausschnitt Initialfunktion

Wie bei flachen Automaten gibt es hier verschiedene Arten der Zustandswechsel, allerdings ist die Anzahl der Möglichkeiten für Zustandswechsel hierbei größer. In der Literatur [Samek02 S.112] ist eine Auflistung der Möglichkeiten gegeben. Daraus lassen sich die vier folgenden Möglichkeiten zusammenfassen. Meine Überlegungen beinhalten zudem die notwendigen Aktionen während der verschiedenen Zustandswechsel. Die genannten Beispiele beziehen sich auf das Modell in Abbildung 5-11.

- **der Zielzustand ist der aktive Zustand**

  Es muss lediglich die `exit`-Funktion aufgerufen, der Zustandswechsel vollzogen und die `entry`-Funktion aufgerufen werden. Diese Möglichkeit ist analog zum Verlassen und sofortigen Wiedereintritt in einen Zustand. Ein `history`-Aufruf zum Speichern des letzten Zustandes ist nicht nötig. Im

konkreten Beispiel entspricht dieser Fall der Reaktion auf das Ereignis `h`, während der Zustand `S21` aktiv ist.

- **der Zielzustand ist ein Subzustand von dem aktiven Zustand**

  Der Automat wandert vom aktiven Zustand, die Hierarchie abwärts, bis zum Zielzustand. Es werden jeweils die `entry`-Funktionen der betretenen Zustände ausgelöst. Der `init`-Aufruf erfolgt erst im Zielzustand. Ein Beispiel dafür ist das Auftreten des Signals `b` im aktiven Zustand `S21`. Hierbei ist der Zielzustand laut Automatenmodell der Zustand `S211`.

- **der Zielzustand ist ein Superzustand von dem aktiven Zustand**

  Der Subzustand und alle darüber liegenden Zustände sollen der Reihe nach verlassen werden, bis der Zielzustand erreicht ist. Hierbei sollen die `history`- und die `exit`-Funktionen der durchlaufenen Zustände ausgeführt werden. Der Wechsel vom Zustand `S211` nach Zustand `S0` mit `g` stellt diese Möglichkeit dar.

- **der Zielzustand ist ein Subzustand in einer anderen Verzweigung der Hierarchie als der aktive Subzustand**

  Es sollen alle Zustände im aktuellen Zweig der Reihe nach verlassen werden, bis ein gemeinsamer Superzustand erreicht ist. Von da an werden alle Zustände der Reihe nach betreten, bis der Zielzustand erreicht ist. Bei dieser Möglichkeit gibt das Klassendiagramm einen guten Überblick. In dem Beispiel entspricht diese Möglichkeit dem Übergang von `S11` nach `S211` als Reaktion auf das Signal `g`.

5.3.3 History-Funktionalität (flache History)

Die Implementation der Gedächtnisfunktion ist eine wünschenswerte funktionale Anforderung. Es ist jedoch interessant zu prüfen, ob die Integration in das entwickelte Entwurfsmuster und den damit entwickelten Zustandsautomaten möglich ist. Im Rahmen dieser Arbeit habe ich die Entwicklung auf die Funktionalität zur Umsetzung der flachen History beschränkt.

Grundsätzlich soll diese Funktionalität genutzt werden, wenn ein Subzustand verlassen wird. Es muss der direkt darüber liegende Superzustand seinen letzten aktiven Subzustand speichern. Das erfordert das Senden einer Nachricht an den Superzustand beim Verlassen des aktiven Zustandes. Für diesen Zweck eignet sich die Nutzung von Vererbung, indem die Funktion `history` des Superzustandes aufgerufen wird. Der Aufruf erfolgt jeweils in der `exit`-Funktion des aktiven Zustandes. Es soll der Parameter `t` übergeben werden, weil dieser den `this`-Zeiger auf das Objekt der Kontextklasse enthält. Das ist notwendig, um den Zustand in der Kontextklasse zu speichern (siehe Abschnitt 5.2.3). Für diesen Zweck stelle ich die Funktion **`setHistory(int ID, State<Context>* ptr)`** in der Kontextklasse bereit. Die

erforderlichen Parameter sind die ID und der this-Zeiger (vtbl-Zeiger) auf den aktiven Zustand. Die ID wird einer Aufzählung (engl.: Enumeration) aus der abstrakten Basisklasse der Zustände entnommen. In dieser Aufzählung sind alle Zustände enthalten, so dass jede ID genau einem Zustand zugeordnet ist. Ein beispielhafter Aufruf der setHistory Funktion von Zustand S1, ist in Abbildung 5-15 dargestellt.

```
template <class T>
void StateS1<T> ::history(T* t)
{
   cout << "speichere history S1" << endl;
   t->setHistory(State<T>::StateID::StateS1_ID, this);
}
```

Abbildung 5-15: Quellcodeausschnitt History-Funktion

In dem entwickelten Muster sind keinerlei Daten in den Zuständen vorgesehen, es soll ausschließlich das Verhalten der Zustände des Modells in den konkreten Zuständen realisiert werden. Unter dieser Voraussetzung ist es möglich, das Speichern der Zustände, bezüglich der Gedächtnisfunktion auf den Zeiger der virtuellen Funktionstabelle der aktiven Zustandsklasse zu beschränken.

```
void setHistory(int ID, State<Context>* ptr)
{
   history_[ID] = *((void**)ptr);
}

void* getStateFromHistory_(int ID)
{
   return history_[ID];
}

private:
   void* history_[NUMBEROFSTATES];
```

Abbildung 5-16: Quellcodeausschnitt Zustand speichern in Kontextklasse

In Abbildung 5-16 ist ein Quellcodeausschnitt aus der Kontextklasse dargestellt. Er beinhaltet die Funktionen zum Speichern und Zurückgeben des Zeigers, der auf die virtuelle Funktionstabelle des zu speichernden Zustandes gerichtet ist. Zusätzlich ist die Variable history_ mit aufgeführt, in der es für jeden Zustand einen Platz geben muss, denn in diesem Array vom Typ void-pointer werden die Zeiger auf die virtuellen Funktionstabellen aller Zustände gespeichert. Die Zuordnung erfolgt über die zuvor bereits erwähnte ID. In der Variablen history_ wird lediglich die Adresse der virtuellen Funktionstabelle des History-Zustandes gespeichert. Der übergebene Parameter ptr entspricht dem this-Zeiger des zu speichernden Zustandes, jedoch ist die Speicherung der Adresse der virtuellen Funktionstabelle komplizierter. Nur durch das Vornehmen einer Typumwandlung (engl.: type-cast) kann die virtuelle

Funktionstabelle gespeichert werden. Für diesen Zweck muss der Zeiger `ptr` zunächst in den Typ `void**` umgewandelt werden, um später an den direkten Zeiger auf die virtuelle Funktionstabelle zu gelangen. Durch die folgende Anwendung des Dereferenzierungsoperators wird die Adresse der virtuellen Funktionstabelle ausgelesen und in dem Array `history_` gespeichert. Mit Hilfe des in Abbildung 5-17 dargestellten Debugger-Ausschnittes kann der beschriebene Vorgang nachvollzogen werden. Es ist ersichtlich, dass der Zeiger `ptr` dem `this`-Zeiger entspricht, dessen Adresse ständig gleich ist. Die Adresse des Zeigers `__vfptr` wird jedoch bei jedem Zustandswechsel geändert und zeigt auf die virtuelle Funktionstabelle des aktiven Zustandes. Bei der Dereferenzierung ohne vorherige Umwandlung würde nicht die Adresse der Tabelle, sondern der Zustand zurückgegeben werden.

Überwachen 1

Name	Wert	Typ
⊟ history_	0x0012fea4	void * [7]
[0]	0x0044f284 const StateS2<class Context>::`vftable'	void *
[1]	0x0044f710 const StateS12<class Context>::`vftable'	void *
[2]	0x00000000	void *
[3]	0x00000000	void *
[4]	0x0044f368 const StateS21<class Context>::`vftable'	void *
[5]	0x0044f450 const StateS211<class Context>::`vftable'	void *
[6]	0x00000000	void *
⊟ ptr	0x00321400	State<Context> *
⊟ __vfptr	0x0044f710 const StateS12<class Context>::`vftable'	*
[0]	0x004198f2 StateS12<class Context>::`vector deleting destructo	*
[1]	0x0041939d StateS12<class Context>::sigA(class Context *)	*
[2]	0x00419929 StateS12<class Context>::sigB(class Context *)	*
[3]	0x00419d43 StateS12<class Context>::sigC(class Context *)	*
[4]	0x00419528 StateS12<class Context>::sigD(class Context *)	*
[5]	0x00419082 StateS12<class Context>::sigE(class Context *)	*
[6]	0x00419d66 StateS12<class Context>::sigF(class Context *)	*
[7]	0x004194d8 StateS12<class Context>::sigG(class Context *)	*
[8]	0x004196ef StateS12<class Context>::sigH(class Context *)	*

Abbildung 5-17: Debug-Ausschnitt vtbl und history

Mit der Funktion **`getStateFromHistory(int ID)`** wird der History-Zustand eines beliebigen Zustandes wieder abgerufen. Mit der `ID` wird lediglich die Zustandsnummer des aktiven (aufrufenden) Zustandes übergeben. Als Rückgabetyp wird wiederum der Typ `void*` verwendet, weil der Datentyp keine Rolle spielt, nur die Adresse ist von Bedeutung. Ein Beispiel für einen Aufruf dieser Funktion ist in Abbildung 5-14 gegeben. Der Zeiger des History-Zustandes wird in der Variablen `history` gespeichert. Der `this`-Zeiger, wird mit Hilfe der Funktion `memcpy` überschrieben. Nach der Ausführung von `memcpy` wurde der Zustandswechsel in den History-Zustand bereits vollzogen. Bei der Anwendung der Funktion `memcpy()` werden die ersten 4 Byte von dem aktiven Zustand überschrieben. Das bedeutet, dass die Adresse der virtuellen Funktionstabelle durch die Adresse der virtuellen Funktionstabelle des History-Zustandes ersetzt wird.

Die Initialisierung der Daten erfolgt in dem Konstruktor der Kontextklassen (Klasse `Context`). In dem Konstruktor wird der Startzustand gesetzt, der dem übergebenen Zustand entspricht. Zudem werden im Konstruktor alle Plätze von dem Array `history_` mit `0` belegt. Das ist notwendig damit der Automat feststellen kann, ob bereits ein Zustand im Zuge der Gedächtnisfunktionalität gespeichert wurde (siehe Abbildung 5-14). Wenn noch kein Zustands als History-Zustand gespeichert ist, wird die Initialaktion wie gewohnt ausgeführt.

5.4 HSMXml2Pattern Generator für hierarchische Automaten

Hier wird der exemplarische Ansatz eines Quellcodegenerators beschrieben. Der Generator hat die Aufgabe, einen Zustandsautomaten nach dem entwickelten Muster für die Erstellung eines hierarchischen Automaten zu erzeugen. Den Generator selbst habe ich in C++ implementiert. Die daraus generierten Quellcodedateien sind ebenfalls für einen C++ Compiler vorgesehen. Als Entwicklungsumgebung habe ich das Microsoft Visual Studio .NET 2003 ® genutzt.

Das Ziel eines vollständigen Quellcodegenerators wäre es, dem Anwender des Entwurfsmusters die Anwendung soweit wie möglich zu erleichtern. Der Anwender sollte nur das Automatenmodell entwerfen und dieses dann in den Generator eingeben. Die Eingabe erfolgt über eine XML-Datei. Ein vollständiger Quellcodegenerator ist im Rahmen dieser Arbeit nicht notwendig, weil er für das Thema von nebenläufiger Bedeutung ist. Es soll lediglich die Möglichkeit dargelegt werden, einen nutzbaren Quellcodegenerator, zur Implementation des entwickelten Musters, zu erstellen.

Durch den von mir implementierten exemplarischen Ansatz ist es möglich, nach Angabe des gewünschten Zustandes für diesen die nötigen Dateien zu erstellen. Die Erzeugung der Dateien für die Kontextklasse ist nicht implementiert. Es wird kein kompletter Automat erzeugt, sondern nur konkrete Zustände.

5.4.1 XML-Datei

Um ein State Pattern zu generieren, müssen zunächst die Daten des Zustandsautomaten eingegeben werden. Damit diese Daten wieder verwendet werden können, empfiehlt es sich, diese in eine Datei zu schreiben. Dies führt zu einer Vereinfachung möglicher Erweiterungen des Entwurfsmusters zu einem späteren Zeitpunkt. Für diesen Zweck eignet sich eine XML-Datei, weil sich diese mit Hilfe der in Kapitel 2.4 beschriebenen Mittel relativ einfach parsen und somit verarbeiten lässt. Die Grundstruktur der Datei ist in der Abbildung 5-18 zu sehen. Die Grundlage für diese Struktur stammt aus der Veröffentlichung *„On Implementation of Finite State Machines“* [GurBos99]. Ich habe die Struktur jedoch verändert und erweitert, weil ein hierarchischer Automaten

implementiert werden sollte, im Gegensatz zur Implementation eines flachen endlichen Automaten in der genannten Veröffentlichung.

Die XML-Datei soll alle notwendigen Daten enthalten. Typischerweise benötigt ein Zustandsautomat Informationen über die Zustände, Ereignisse und Transitionen. Zudem können, je nach Detaillierung des Automaten Informationen hinzugefügt werden.

Dieser spezielle Generator benötigt, zusätzlich zu den Informationen des Automaten, weitere Informationen. Es muss bekannt sein, welches der erste beziehungsweise der Startzustand ist. Um im Projekt für Übersicht zu sorgen, wird ein Verzeichnis zur Quellcodeerzeugung zur Verfügung gestellt. In diesem Verzeichnis befinden sich die generierten Quellcodedateien des Entwurfsmusters nach der Codeerzeugung. Dieser Punkt macht eine Erweiterung der Struktur der XML-Datei, gegenüber der Vorlage, notwendig.

```
<?xml version="1.0" ?>
<nsp entrystate="S11">
   <genCodeDir dir="genCode" />
   <states>
   . . .
   </states>
   <events>
   . . .
   </events>
   <transitions>
   . . .
   </transitions>
</nsp>
```

Abbildung 5-18: Gesamtstruktur XML-Dokument

Im Bereich der Zustandsdefinition (Knoten `states`) werden alle Zustände des Zustandsautomaten definiert. Für jeden Zustand wird der Name und falls vorhanden der Superzustand und der Initialzustand angegeben. Das Element `state` verwendet zur Definition des Zustandsautomaten und des Initialzustandes die Attribute `name` und `initstate`. Hier ist darauf zu achten, dass der zuerst definierte Zustand den obersten Zustand in der Hierarchie darstellt. Es werden also alle weiteren Zustände von ihm abgeleitet, womit sie auch sein Verhalten erben. Es genügt den Namen des Zustandes anzugeben ohne Zusätze wie zum Beispiel das Präfix `State`. Ein Auszug des Bereichs der Zustandsdefinition ist in der Abbildung 5-19 zu finden. Bei den Attributen `superstate` und `initstate` handelt es sich um Erweiterungen, die in flachen Automaten nicht notwendig sind.

```
<states>
   <state  name="S0"
         initstate="S1"/>
   <state   name="S1"
         superstate="S0"
         initstate="S11"/>
   <state   name="S11"
         superstate="S1"
         initstate=""/>
   <state   name="S2"
         superstate="S0"
         initstate="S21"/>
</states>
```

Abbildung 5-19: Zustandsdefinition in der XML-Datei

```
<events>
   <event name="sigA"/>
   <event name="sigB"/>
   <event name="sigC"/>
   <event name="sigD"/>
   <event name="sigE"/>
   <event name="sigF"/>
   <event name="sigG"/>
   <event name="sigH"/>
</events>
```

Abbildung 5-20: Ereignisdefinition in der XML-Datei

Im Bereich der Ereignisdefinition (Knoten `events`) gibt es die Möglichkeit Ereignisse zu definieren, damit der Zustandsautomat auf diese Ereignisse reagieren kann. Alle Ereigniselemente werden dem Elternknoten `events` untergeordnet. Für jedes Ereignis wird ein XML-Element `event` mit dem Attribut `name` benötigt. Das Attribut soll den Ereignisnamen enthalten (siehe Abbildung 5-20). Dieser Name wird im Programmcode eine Funktion darstellen, mit deren Hilfe auf dieses Ereignis reagiert werden kann.

Die Angabe der Übergänge erfolgt sowohl intern als auch extern im Knoten `transitions`. Dieser Knoten besitzt Elemente vom Typ `transition`, welche verschiedene Attribute besitzt. Das Attribut `sourcestate` gibt den Quellzustand an, während `targetState` den Zielzustand angibt, der nach der gewünschten Aktion `action` betreten werden soll. Bei der Angabe einer Aktion handelt es sich um ein Stück Quellcode, welches in der angegebenen Datei (Pfadangabe wenn nicht im gleichen Ordner) definiert wird. Diese Aktion wird in dem entwickelten Quellcode des Zustandsautomaten eingebettet. Hierbei trägt der Anwender die Verantwortung über die Syntax und die Semantik des Quellcodes der Aktion. Abschließend wird das Attribut `event` und somit das Ereignis angegeben, bei dem der Übergang ausgelöst werden soll. Ein Auszug der Transitions-Definition ist in der Abbildung 5-21 zu finden.

Damit das Parsen der Datei nicht unnötig kompliziert wird, sind die einzelnen Knoten nicht tiefer geschachtelt. Es liegen ausschließlich Elemente mit Attributen vor, weitere interne Elemente sind nicht vorgesehen.

Bei der Verwendung des entwickelten `XML2Pattern`-Generators ist unbedingt auf die genaue Einhaltung der beschriebenen Konventionen zu achten. Auf die Verwendung einer Dokument Typ Definition (DTD) wurde verzichtet, weil es sich lediglich um einen exemplarischen Generator handelt,und dieser möglichst einfach gehalten werden sollte.

```
<transitions>
      <transition sourcestate="S0"
               targetstate="S211"
               event="sigE"
               action="actions\sigE.code"/>
      <transition sourcestate="S1"
               targetstate="S1"
               event="sigA"
               action="actions\sigA.code"/>
      <transition sourcestate="S1"
               targetstate="S11"
               event="sigB"
               action="actions\sigB.code"/>
. . .
</transitions>
```

Abbildung 5-21: Transitions-Definition in der XML-Datei

5.4.2 Struktur

Der entwickelte Generator besteht aus der Klasse `HSMXml2Pattern` und benutzt die Klassen `HSMXmlState` und `HSMXmlTransition`. Für die Verwendung des DOM-Parsers von Xerces müssen diverse Header-Dateien eingebunden werden. Als Grundgerüst für das Parsen von XML-Dokumenten kann das Beispiel [XDOMPar] der Xerces Internetseite als Vorlage genutzt werden. In diesem Beispiel sind alle benötigten Header-Dateien bereits eingebunden. Es müssen lediglich die benötigten Programm-bibliotheken (DLL's) kopiert werden. Eine genaue Beschreibung der Konfiguration befindet sich im Anhang, in der mit Doxygen[8] erstellten Dokumentation des Quellcodegenerators. In der Abbildung 5-22 ist ein Klassendiagramm mit den oben erwähnten Klassen, deren Attributen und Eigenschaften dargestellt. Die Klasse `HSMXmlState` dient zur Speicherung der Zustandsinformationen. Zur Erfassung aller relevanten Daten der Zustandsübergänge wird die Klasse `HSMXmlTransition` verwandt. Mit Hilfe dieser Klassen kann auf die verschiedenen Zustände und

[8] Doxygen – Softwarewerkzeug zum Erstellen einer HTML – Dokumentation des Quelltextes

Übergänge zugegriffen werden. Die Verarbeitung der Daten aus der XML-Datei wird in der Klasse `HSMXml2Pattern` realisiert.

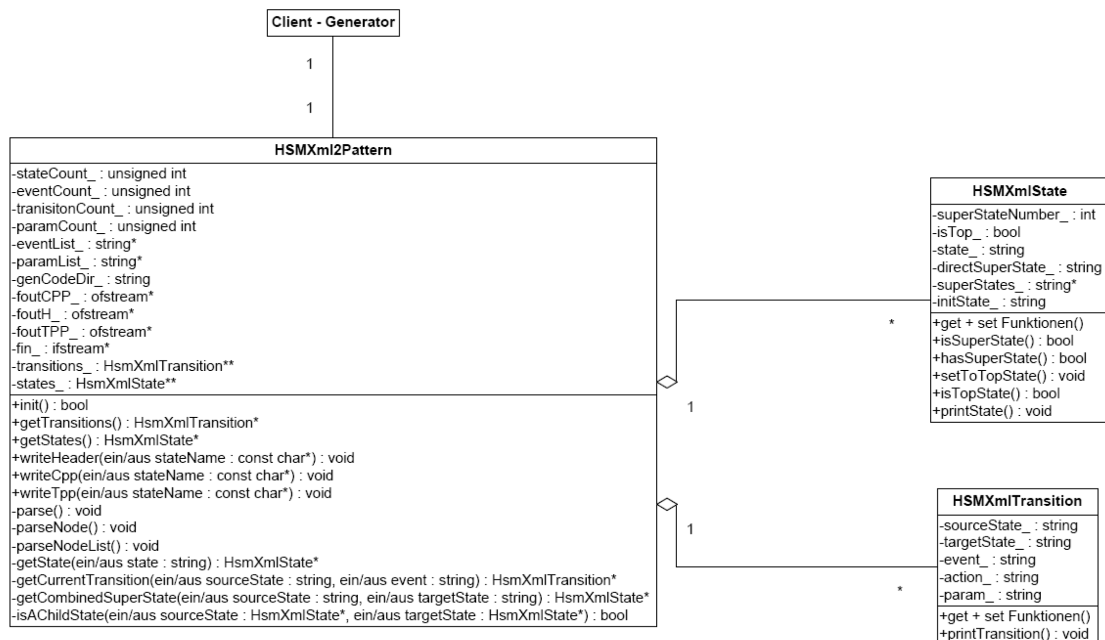

Abbildung 5-22: Klassendiagramm Quellcodegenerator

5.4.3 Verarbeitung der Daten

Um eine Klasse, die einen Zustand repräsentiert, erzeugen zu können, müssen die Daten über den Zustand selbst (Name, Initialzustand und Superzustand) bekannt sein. Zudem müssen die Ereignisse, auf die reagiert werden soll, bekannt sein, damit die entsprechenden Funktionen implementiert werden können. Die Ereignisse werden als `Strings` gespeichert, um später auf sie zugreifen zu können. Die Verwirklichung der Zustandsübergänge als Reaktion auf Ereignisse erfordert zusätzliche Informationen über die Transitionen. In der in Abschnitt 5.4.1 beschriebenen XML-Datei sind die benötigten Daten sequentiell aufgeführt. Mit Hilfe der entwickelten Klassen aus Abschnitt 5.4.2 ist es möglich, alle notwendigen Daten nacheinander zu erfassen, um sie für die spätere Verwendung bereitzustellen. So werden beim Parsen die Daten in den bereitgestellten Klassen gespeichert. Die im nächsten Abschnitt beschriebene Erzeugung des Codes findet demzufolge nach dem Parsen statt, wenn alle Daten aus der XML-Datei bereits gespeichert wurden.

5.4.4 Codeerzeugung

Für jeden Zustand werden aufeinander folgend drei Dateien erzeugt. Für die Erzeugung der Header-, Template- und Cpp-Quellcodedateien sind die Funktionen `writeHeader`,

`writeTPP` und `writeCPP` zuständig. In der Header-Datei wird die Zustandsklasse deklariert und alle notwendigen Dateien werden eingebunden. Am Ende der Header-Datei wird zusätzlich die zugehörige Template-Datei, in der die Funktionen der Template-Klasse definiert werden, eingebunden. Die Funktionsweise der Funktionen ist ähnlich. Zunächst wird unter der Verwendung eines `ofstream`[9] die jeweilige Datei angelegt. Mit dem daraus entstehenden Zeiger auf den Ausgabestrom kann direkt in die Datei geschrieben werden. Hierbei kommen die gespeicherten Zustands- und Transitionsdaten zum Einsatz. Diese sind notwendig, um die Zustände mit den korrekten Funktionen auszustatten. Der Kopf der Dateien, mit Ausnahme des Klassennamens und der Superklasse, ist in den verschiedenen Klassen identisch.

Die verschiedenen Möglichkeiten bei den einzelnen Transitionen führen zu einer nicht unkomplizierten Implementation der Zustandsübergänge in den Template-Dateien. Es sind die in Abschnitt 5.3.2 erläuterten Fälle zu unterscheiden. Um herauszufinden, um welchen Fall es sich handelt, wurden verschiedene Hilfsfunktionen implementiert. Des Weiteren sind verschiedene Abfragen nötig. Im Folgenden sind die verschiedenen Fälle und die zugehörigen Abfragen aufgelistet.

- **der Zielzustand ist der aktive Zustand**

  Abfrage: `if(targetStateString == currentState->getState())`

  Lediglich die `exit-` und `entry-`Funktionen werden integriert.

- **der Zielzustand ist ein Subzustand von dem aktiven Zustand**

  Abfrage: `else if(isAChildState(targetState, currentState))`

  In diesem Fall ist eine tiefe Schachtelung der Zustände möglich. Es ist sinnvoll, vom Zielzustand aufwärts zum aktiven Zustand zu gehen. Die Route zum Ziel beziehungsweise die Zustände werden der Reihenfolge nach gespeichert. Letztendlich können die gespeicherten Zustände in umgekehrter Reihenfolge abgearbeitet und in die Quellcodedatei eingetragen werden.

- **der Zielzustand ist ein Superzustand von dem aktiven Zustand**

  Abfrage: `else if(isAChildState(currentState, targetState))`

  Auch hier ist eine tiefe Schachtelung der Zustände möglich. Es muss jeder Zustand bis zum Zielzustand aufgeführt werden, ähnlich wie im vorangegangenen Fall. Vom aktuellen Zustand wird in der Hierarchie nach oben gegangen, bis der Zielzustand erreicht ist. Eine Speicherung der Zustände ist aufgrund der direkten Verarbeitung der Zustände nicht erforderlich.

---

[9] ofstream – C++ Klasse für die Ausgabe in eine Datei

- **der Zielzustand ist ein Zustand in einer anderen Verzweigung der Hierarchie**

  Abfrage: `else`

  Hier trifft keiner der vorherigen Fälle zu. Es muss bis zum nächsten gemeinsamen Superzustand hinauf gewechselt werden, um von da an bis zum Zielzustand in der Hierarchie hinunter zu wechseln. Hierbei werden die entsprechenden `exit-`, `entry-` und `init`-Funktionen integriert. Das Vorgehen dabei gleicht einer Kombination der beiden vorherigen Fälle.

In den behandelten Fällen werden diverse Hilfsfunktionen und Hilfsvariablen verwendet, um die richtigen Zustände und die der Situation entsprechenden Aktionen zu integrieren. Zum Beispiel die Variable `combinedSuperState` gibt Aufschluss über den nächsten gemeinsamen Zustand vom Zielzustand und dem aktiven Zustand. Die Variable wird mit der Funktion `getCombinedSuperState` aktualisiert. Erst die Kombination von solchen Hilfsfunktionen mit algorithmischen Werkzeugen (Schleifen) ermöglicht die Codeerzeugung.

5.4.5 Test

Der Test des Codegenerators ist ebenso einfach wie nützlich. Für den Test ist es sinnvoll, ein Testprojekt anzulegen, welches eine Kopie vom aktuellen Projekt des entwickelten Zustands-Entwurfsmusters ist. Die verwendete XML-Datei muss mit den Daten des verwendeten Automatenmodells gespeist werden. Der Generator wird mit der Angabe eines bestimmten Zustandes gestartet (Zustand, für den Quellcode erzeugt werden soll). Durch das Kompilieren des Projektes wird geprüft, ob der Generator bezüglich der Syntax korrekt arbeitet. Bei Erfolg ist der nächste Testschritt das Testen der Funktionalität. Die erzeugten Dateien können in das Testprojekt kopiert und die ursprünglichen Dateien des gewählten Zustandes überschrieben werden. Der Automat wird im Testprojekt ausgeführt, über die Tastatur können die Signale von `a-h` eingegeben werden. Es folgen die Reaktionen, die anhand des Automatenmodells überprüft werden können, als Ausgaben auf der Konsole.

Bei Änderungen im Design des Entwurfmusters, im Zuge der Erweiterung, konnten Änderungen an vielen Stellen auch einfach im Generator vorgenommen werden. Durch das Erzeugen der Dateien für alle Zustände, konnten diese im Testprojekt genutzt werden. Im Zuge der Erweiterung, konnten beispielsweise Designänderungen des Entwurfsmusters mit Hilfe des Generators schnell und einfach durchgeführt werden. Dieses Vorgehen war möglich, weil der Generator zeitgleich mit dem Muster entwickelt wurde.

5.4.6 Erweiterungen

Es bedarf einiger Erweiterungen, um einen vollständigen Quellcodegenerator fertig zu stellen. In dem exemplarischen Generator wird auf das Erstellen der Kontextklasse verzichtet, weil die Erstellung der Zustandsklassen und die damit verbundenen Zustandswechsel bei der Entwicklung von höherer Priorität waren. Zudem liegt beim Aufwand einer Projekterstellung nach dem Zustands-Entwurfsmuster der Schwerpunkt bei der Erstellung der Zustandsklassen.

Das Ziel eines vollständigen Quellcodegenerators wäre die Erstellung eines kompletten Zustandsautomaten in C++. Zur Erzeugung des Automaten müssten lediglich der Pfad der XML-Datei und der Ausgabeordner für die Dateien angegeben werden. Es würden alle definierten Zustände aus der XML-Datei erstellt werden. Zusätzlich wäre es denkbar, anzugeben, ob der zu erstellende Automat die Gedächtnisfunktion unterstützen soll. Dabei wäre es sinnvoll, anzugeben, ob es sich um eine flache oder tiefe Gedächtnisfunktion handeln soll. Um den Generator leichter nutzbar zu machen, ist der Einsatz einer grafischen Benutzerschnittstelle (engl.: Grafical User Interface GUI) denkbar. Dort könnte zusätzlich die Festlegung der Eigenschaften der Gedächtnisfunktion erfolgen.

Abschließend darf auf keinen Fall die Fehlerbehandlung vergessen werden, denn bei Systemen, die mit Benutzereingaben arbeiten, treten häufig Fehler auf. Falsche Eingaben müssen erkannt und dem Anwender mitgeteilt werden, damit dieser darauf reagieren kann. Ebenso durch Fehler in der Struktur der XML-Datei können Fehler bei der Datenverarbeitung auftreten. Diese Fehler müssen ebenfalls erkannt und gemeldet werden, um eine zuverlässige Anwendung des Generators zu gewährleisten.

6 Auswertung und Vergleich mit analysierten Techniken

In diesem Kapitel erfolgt die Auswertung des entwickelten Zustandsmusters. Es werden die Vor- und Nachteile der entwickelten Technik dargestellt und mit den Eigenschaften der analysierten Techniken verglichen. Aufgrund von fehlenden Standardtechniken für hierarchische Automaten, erfolgt eine Einschränkung des Vergleichs hinsichtlich der Implementationen mit flacher Struktur.

Es findet eine Einschränkung des Vergleichs auf die Techniken für die Erstellung flacher Zustandsautomaten statt, weil es keine Standardmuster für den Entwurf von hierarchischen Automaten gibt.

Grundsätzlich besitzt das entwickelte Muster ähnliche Eigenschaften wie das State Design Pattern (SDP) der GoF aus Kapitel 3.3. Aus diesem Grund hat die von mir entwickelte Lösung die gleichen Vorteile wie das SDP.

Auch in der entwickelten Lösung sind die Zustandsübergänge sehr effizient. Es wird ebenfalls, wie beim SDP, ein Zeiger überschrieben. Hier handelt es sich um den Zeiger auf die virtuelle Funktionstabelle, während bei der Lösung der GoF der Zustandszeiger in der Kontextklasse überschrieben wird. Zudem ist das Prinzip des neu entwickelten Automaten einfacher zu verstehen und speicherschonender als das der GoF. So sind bis auf eine Ausnahme alle Nachteile des SDP, aufgrund der Anwendung der angewandten Struktur und Technik, nicht mehr vorhanden. Diese Eigenschaft kann nur als Nachteil bezeichnet werden, wenn der Zustandsautomat nach der Fertigstellung verändert wird.

- Um auf neue Ereignisse reagieren zu können, müssen diese zur Schnittstelle `State` (abstrakte Zustandsklasse) und zur Kontextklasse hinzugefügt werden.

In den folgenden Abschnitten gehe ich detaillierter auf die Verbesserungen meiner Lösung in Bezug auf die Lösung der GoF ein. Zudem werde ich die anderen analysierten Techniken ansprechen und diese an geeigneten Punkten mit meiner Lösung vergleichen.

6.1 Struktur

Durch die Verwendung des Operators `Placement-new` für die Wechsel der Zustände untereinander ist eine extra Funktion für den Zustandswechsel überflüssig. In der Lösung der GoF muss für die geschützte (`protected`) Funktion eine Freundschaft (Schlüsselwort `friend`) zu den Zustandsklassen aufgebaut werden. Dies ist zum Schutz gegen unerlaubte Nutzung erforderlich. Dank der Einsparung der Zustandswechselfunktion wird die Kapselung der Klassen, in der von mir entwickelten Lösung, nicht aufgebrochen. Somit werden Programmierstandards eingehalten und das Verständnis

der Struktur vereinfacht. Der Aufbau kann gut an dem vorhandenen Klassendiagramm nachvollzogen werden. Das macht eine Realisierung nach diesem Prinzip relativ einfach möglich.

Die Struktur der anderen Techniken lässt sich kaum mit der entwickelten Lösung vergleichen. Bei den geschachtelten `switch-case` Anweisungen aus Kapitel 3.1 handelt es sich zudem um eine Lösung, die auch prozedural angewendet werden kann. Die Struktur ist leicht verständlich und nachzuvollziehen. Es ist jedoch kaum möglich, Code wieder zu verwenden. Das Prinzip der Realisierung mit einer Zustandstabelle aus Kapitel 3.2 ist ebenso leicht verständlich. Die Initialisierung der Zustandstabelle erschwert jedoch die Umsetzung erheblich. Wenn strikt nach dem vorgestellten Beispiel vorgegangen wird, ist die Realisierung aber dennoch in kurzer Zeit möglich. Die von M. Samek entwickelte Lösung des optimalen FSM aus Kapitel 3.4 wird ebenfalls durch eine einfache Struktur dargestellt. Eine einstufige `switch-case` Anweisungsschicht übernimmt die Ereignisbehandlung. Diese Variante ist leicht nachzuvollziehen und umsetzbar. Die Verwendung von `Pointer-to-Member` Funktionen stellt jedoch für unerfahrene Entwickler möglicherweise ein Problem dar.

Ein weiterer Vorteil der Nutzung der neuen Technik tritt bei der Haltung der Zustände auf. Das Muster der GoF sieht die Verwendung des Singleton-Musters aus [GoF95 S.157-166] vor, um benötigte Instanzen der Zustände erst bei der Nutzung der Zustände erstellen zu müssen. Zudem wird das Singleton-Muster verwendet, um ausschließlich eine Instanz jedes Zustandes zu erzeugen. In der hier entwickelten Variante wird auf die Anwendung des zusätzlichen Musters verzichtet. Instanzen der vorhandenen Zustände müssen nicht erzeugt werden, weil die Zustände nur virtuelle Funktionen enthalten. Die virtuellen Funktionen jedes Zustandes werden durch die virtuelle Funktionstabelle des jeweiligen Zustandes repräsentiert. Der ausführbare Code der Funktionen liegt, wie bereits in Kapitel 5.1 beschrieben, im Codesegment. Bei den anderen der analysierten Techniken ist eine Instantiierung der Zustände, bedingt durch die Struktur, ebenfalls nicht notwendig.

Die Methode mit den geschachtelten `switch-case` Anweisungen ist die einfachste Lösung hinsichtlich Struktur und Verständnis. Im Bereich der objektorientierten Ansätze hat die neue Technik einen Vorteil gegenüber der Struktur des Musters der GoF, sie ist einfacher und verständlicher. Der optimale FSM ist nicht so leicht verständlich wie die Struktur der neu entwickelten Technik. Ebenso die Zustandstabelle ist aufgrund der Initialisierung der Tabelle komplizierter als die entwickelte Lösung.

6.2 Leistung

Ein wichtiger Punkt bei dem Einsatz von Zustandsautomaten ist die Nutzung des Speichers. Speziell bei dem Einsatz in Echtzeitsystemen spielt der Speicherbedarf eine

große Rolle. In Echtzeitsystemen ist aber auch die Geschwindigkeit der Ausführung eine nicht zu vernachlässigende Größe. Bei dem Vergleich des Speicherbedarfs und der Laufzeit der verschiedenen Techniken werden ausschließlich Komponenten, die für den Betrieb des jeweiligen Zustandsautomaten notwendig sind, berücksichtigt. Optionale Aktionen werden nicht mit einbezogen.

6.2.1 Laufzeit

Die Ereignisverarbeitung spielt im Hinblick auf die Ausführungszeit bei Zustandsautomaten eine wichtige Rolle. Durch den Prozess der späten Bindung (Laufzeitpolymorphie) erfolgt die Ereignisverarbeitung sowohl bei dem Muster der GoF als auch bei meinem Ansatz sehr schnell. Die Komplexität für diese Art der Ereignisverarbeitung liegt bei O(1). Das bedeutet, die Zeit der Ereignisverarbeitung ist unabhängig von anderen Faktoren und somit immer konstant. Dies ist ebenso bei der Realisierung der Zustandstabelle der Fall. Dieser Ansatz bietet ebenfalls eine sehr schnelle Ereignisverarbeitung, aber M. Samek ist in [Samek02 S.68] der Meinung, die Technik ist nicht so effektiv wie die späte Bindung. Bei den geschachtelten `switch-case` Anweisungen sieht es anders aus. Durch die zweistufige Technik wird relativ viel Zeit benötigt, um auf Ereignisse zu reagieren. Die Anzahl der Zustände und möglichen Ereignisse beeinflussen die Laufzeit. Die Komplexität liegt bei dieser Lösung bei O(log n), wobei n das Produkt aus der Anzahl der Zustände und der Anzahl der Ereignisse ist. Der optimale FSM verhält sich ähnlich, es wird jedoch eine Stufe bei der Ereignisverarbeitung eingespart. Die Komplexität liegt ebenfalls bei O(log n), jedoch entspricht n lediglich der Anzahl der möglichen Ereignisse, weil die Laufzeit der Ereignisverarbeitung nur noch von der Anzahl der Ereignisse abhängt.

Ein weiterer Punkt in Bezug auf die Laufzeit ist die Realisierung der Zustandswechsel. Diesen Punkt absolvieren alle Techniken sehr schnell, es gibt geringe Unterschiede bezüglich der Anzahl der aufzurufenden Funktionen. Grundsätzlich wird bei allen Varianten lediglich ein Zeiger oder eine Variable überschrieben. Es handelt sich fast immer um die Variable, die den Zustand hält. Bei der Technik der GoF und der des Operators `Placement-new` wird der Unterschied deutlich. Während bei der Verwendung des Operators `Placement-new` der Zustandswechsel direkt vollzogen wird, muss bei der GoF eine zusätzliche Funktion aufgerufen werden. Der Unterschied ist minimal jedoch vorhanden. Es könnte der Eindruck entstehen, dass die zusätzliche Reservierung von Speicher Zeit kostet. Durch den Einsatz des Operators `Placement-new` wird jedoch keine zusätzliche Reservierung des Speichers vorgenommen, somit kostet der Zustandswechsel keinesfalls mehr Zeit als bei der Technik der GoF.

In Punkto Laufzeit besitzen die Techniken der GoF, des Operators `Placement-new` und der Zustandstabelle die gleiche Komplexität (O(const)). Dadurch sind ihre Ausführungszeiten geringer als die der beiden anderen Techniken. Da die Methode mit dem

Operator `Placement-new` keine weiteren Funktionen für den Zustandswechsel benötigt, ist sie meines Erachtens bezüglich der Laufzeit die am besten geeignete Methode.

6.2.2 Speicher

Die Methode der GoF erfordert es, alle vorhandenen Zustände mit einer Instanz vorzuhalten. Die Verwendung des Singleton-Musters ermöglicht es, die Instanzen erst zu erzeugen wenn es nötig ist. Das ändert jedoch nichts an der Tatsache, dass für jeden Zustand Speicher genutzt wird. Die Instantiierung der Zustände muss bei meiner Lösung nicht explizit vorgenommen werden. Wie in Kapitel 5.1 beschrieben, wird lediglich die Adresse der virtuellen Funktionstabelle des aktiven Zustandes mit der Adresse der virtuellen Funktionstabelle des Zielzustandes überschrieben. Da die Zustände keine Daten enthalten, hat das zur Folge, dass nur 4 Byte für den Zeiger auf die virtuelle Tabelle im Heap reserviert werden. Bei der Lösung der GoF werden ebenfalls lediglich 4 Byte pro Zustand benötigt, weil das Prinzip der Zustandsdarstellung das Gleiche ist. Die Reservierung von 4 Byte erfolgt für jeden Zustand des Automaten. Die Speicherbelastung erscheint minimal, jedoch sollte beachtet werden, dass die Anzahl der Zustände in komplexen Systemen sehr hoch sein kann. Bei der von mir entwickelten Lösung ist der Speicherbedarf völlig unabhängig von der Anzahl der Zustände. Er beträgt also immer 4 Byte. Nur beim Start des Automaten wird einmal Speicherplatz für einen Zustand (Startzustand) belegt. Die Zielzustände nehmen beim Zustandswechsel den Platz des aktiven Zustandes ein. Bei beiden Varianten kommen 4 Byte aus der Kontextklasse hinzu. Diese werden benötigt, um den Zeiger für den aktiven Zustand zu halten.

Bei der Variante der Zustandstabelle aus 3.2 müssen keine Zustände instanziiert werden, da die Unterscheidung anhand der Zustandstabelle realisiert wird. Der Speicherbedarf lässt sich aufgrund dessen nicht direkt vergleichen. Jedoch wird durch die Zustandstabelle viel Speicher benötigt. Diese Tabelle wächst mit zunehmender Anzahl an Zuständen und Ereignissen. Bei einer hohen Anzahl von Zuständen und Ereignissen wird der Speicherbedarf um einiges höher sein als 4 Byte, bereits eine Zelle der Tabelle beansprucht 8 Byte. Die 8 Byte setzen sich wie folgt zusammen.

- Jede Zelle enthält eine Struktur `Tran`. Ein Quellcodeausschnitt mit dieser Struktur ist in Abbildung 6-1 zu sehen. Die `Pointer-to-Member` Funktion sowie die Variable `nextstate` benötigen jeweils 4 Byte. Das ergibt pro Zelle 8 Byte.

```
typedef void (StateTable::*Action)(); // Pointer-to-member
struct Tran {
   Action action;
   unsigned int nextState;
};
```

Abbildung 6-1: Quellcodeausschnitt Struktur Tran (Zustandstabelle)

Des Weiteren werden zusätzlich drei Variablen vom Typ `int` (jeweils 4 Byte) und ein Zeiger auf die Zustandstabelle benötigt. Dies führt zu einer weiteren Belastung von 16 Byte. So ergibt sich insgesamt ein Speicherbedarf von 16 Byte und zusätzlich 8 Byte pro Zelle der Tabelle. Die Anzahl der Zellen ergibt sich aus der Multiplikation der Anzahl der Zustände und der möglichen Ereignisse.

Der optimale FSM von M. Samek benötigt sehr wenig Speicherplatz. Es werden lediglich 4 Byte für die Zustandsvariable (`myState` Zeiger) reserviert. Auch die Methode unter Verwendung der geschachtelten `switch-case` Anweisungen erfordert einen Speicherplatz von nur 4 Byte, weil die Zustände nicht instanziiert werden müssen. Sie liegen als Aufzählung vor. Die 4 Byte entsprechen der Variablen, die den aktiven Zustand vorhält.

Das Diagramm in Abbildung 6-2 stellt den Speicherbedarf der verschiedenen Techniken dar. In dem Szenario wird davon ausgegangen, dass der zu realisierende Automat acht Zustände besitzt, in denen zwölf verschiedene Ereignisse auftreten können. Die von mir entwickelte Technik (`Placement-new`) benötigt 8 Byte und ist damit auf dem zweiten Platz. Das SDP steht mit einem Speicherverbrauch von 36 Byte auf dem dritten Platz, während die Technik mit der Zustandstabelle, mit einem Speicherbedarf von 768 Byte, den höchsten Speicherbedarf benötigt. Den geringsten Speicherplatz verbrauchen in diesem Szenario der optimale FSM und die Technik der geschachtelten `switch-case` Anweisungen, diese benötigen nur 4 Byte. Der Speicherbedarf des optimalen FSM, der Technik der geschachtelten `switch-case` Anweisungen und meine Lösung ist von der Anzahl der Zustände unabhängig, während der Speicherbedarf der beiden anderen Techniken davon abhängig ist.

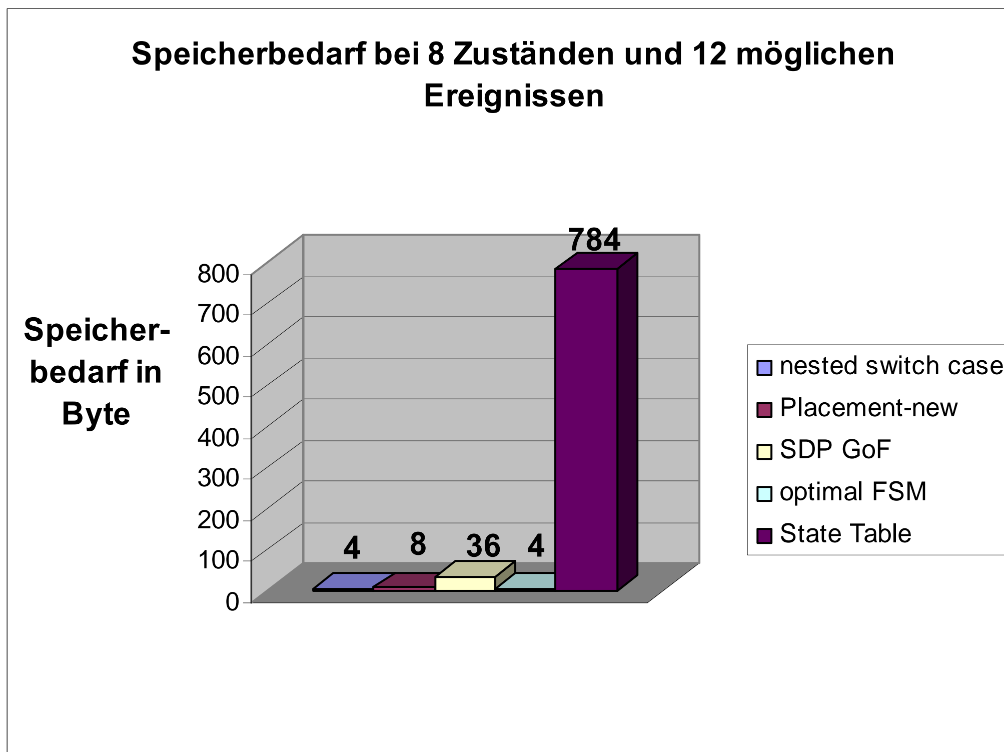

Abbildung 6-2: Diagramm Speicherbedarf

Die Techniken des optimalen FSM und die der geschachtelten `switch-case` Anweisungen sind in Punkto Speicherbedarf die besten Lösungen. Mit einem um nur 4 Byte höheren Speicherverbrauch, ist meine Lösung jedoch eine gute Alternative.

6.3 Funktionalität

Die neu entwickelte Realisierung des Zustandsmusters kann mit der gleichen Funktionalität wie die bereits vorhandenen Techniken dienen. Zudem wird zusätzlich die Möglichkeit eingeräumt, hierarchische Automaten umzusetzen. Alle zwingenden Anforderungen werden in meiner Lösung berücksichtigt und erfüllt. In der Erweiterung des Musters zur Realisierung eines hierarchischen Zustandsautomaten werden später zusätzlich die wünschenswerten Anforderungen, sowohl für flache als auch für hierarchische Automaten, erfüllt. Die Umsetzung eines hierarchischen Automaten ist ohne großen Aufwand realisierbar, was sich in der Struktur anhand des Klassendiagramms aus Kapitel 5.3.1, erkennen lässt. Es gibt jedoch einen Nachteil bei dem Einsatz eines hierarchischen Automaten. Es müssen fest codierte Übergangsketten verwendet werden, um die korrekten `exit`- und `entry`-Funktionen der Zustände auszulösen. Doch wie bereits in Kapitel 5.3.2 beschrieben, ist das nicht sehr aufwendig, da die jeweils zu realisierende Übergangskette durch das Automatenmodell ohnehin bekannt ist.

Bezüglich der Funktionalität ist es ohne Bedeutung welches Modell gewählt wird, weil alle vorgestellten Muster die aufgestellten zwingenden funktionalen Anforderungen erfüllen. Das von mir entwickelte Muster ist jedoch vorzuziehen, wenn ein hierarchischer Automat entwickelt werden soll, da diese Möglichkeit lediglich von meiner Lösung und der Zustandstabelle unterstützt wird. Der Aufwand ist bei meiner Lösung geringer als bei dem Ansatz der Zustandstabelle, auch das Vorhandensein der `History`-Funktion spricht für die neu entwickelte Lösung.

6.4 Wiederverwendbarkeit

Wie bereits erwähnt ist der Aufbau eines Zustandsautomaten nach dem entwickelten Prinzip ist sehr einfach. Es ist dem Entwickler möglich, in sehr kurzer Zeit einen Zustandsautomaten zu implementieren. Die leichte Verständlichkeit des Musters trägt zur Wiederverwendbarkeit bei.

Ein Problem, das häufig beim Einsatz von Zustandsautomaten auftritt, ist die Änderung des Zustandsautomatenmodells. Das Modell ist die Grundlage für den gesamten Automaten. Die Realisierung des Automaten soll exakt dem modellierten Verhalten entsprechen. Änderungen am Automatenmodell nach der Fertigstellung der Implementation, haben Änderungen an der Implementation zur Folge. Diese Änderungen sind abhängig von der Art der Änderung am Automatenmodell. Folgende Änderungen am Automatenmodell kommen häufig vor.

- Zustände hinzufügen / entfernen / ändern
- Ereignisse hinzufügen / entfernen / ändern
- Zustandsübergänge hinzufügen / entfernen / ändern

Eine Änderung der Zustände zieht fast immer eine Änderung an den Zustandsübergängen mit sich. So sind die meisten Änderungen am Modell nur mit großem Aufwand in der vorhandenen Implementation umzusetzen. Hinzu kommt die unterschiedliche Struktur der verschiedenen Implementationstechniken. Das führt zu verschiedenen, notwendigen Modifikationen abhängig von der angewandten Technik.

Bei allen vorgestellten Techniken ist es relativ aufwendig den Code an Modelländerungen anzupassen. Das von mir entwickelte Muster hat trotz der vereinfachten Struktur keine Vorteile gegenüber dem SDP der GoF. An verschiedenen Bereichen im Code müssen Änderungen vorgenommen werden. Durch die Einteilung der Zustände in Klassen ist der Vorgang aber überschaubar. Die Technik der geschachtelten `switch-case` Anweisungen ist am wenigsten für solche Änderungen geeignet, weil an vielen Bereichen im Code Änderungen notwendig sind. Es können sich schnell Fehler einschleichen. Ähnlich sieht es bei dem optimalen FSM aus. Durch die Einsparung einer `switch-case` Stufe sind weniger Änderungen erforderlich, dennoch gibt es viele

Bereiche im Code, an denen die Änderungen notwendig sind. Beim Ansatz der Zustandstabelle müssen nur wenige Stellen im Code nachgearbeitet werden. Es ist jedoch erforderlich, die komplette Zustandstabelle neu zu erstellen. Diese Technik ist aufgrund der aufwendigen und komplizierten Initialisierung der Zustandstabelle für die Umsetzung der Änderungen des Automatenmodells auch nicht besonders gut geeignet

Im Folgenden wird kurz erläutert, welche Änderungen bei meiner Technik oder ebenso bei der Technik der GoF vorzunehmen sind, wenn ein Zustand und ein zusätzliches Ereignis hinzugefügt wird. Beim Hinzufügen eines Zustandes, muss zunächst eine neue Klasse erstellt werden. Außerdem sind die Transitionen zum Erreichen des neuen Zustandes hinzuzufügen. Bei der Erstellung eines zusätzlichen Ereignisses ist eine Änderung der abstrakten Zustandsklasse und der Kontextklasse erforderlich, weil die Ereignisse in Form von Funktionen realisiert werden. Das neue Ereignis ist als virtuelle Funktion in die Zustandsklassen aufzunehmen, die auf dieses Ereignis reagieren soll.

6.4.1 Quellcodegenerator

Der von mir entwickelte Quellcodegenerator aus Abschnitt 5.4 verfolgt das Ziel, den Quellcode für einen hierarchischen Automaten zu erstellen. Es werden alle benötigten Daten durch eine XML-Datei bereitgestellt. Dieser Generator kann ebenso für die Umsetzung von Änderungen des Automatenmodells genutzt werden. Hierzu müssen nur die Daten in der XML-Datei an das veränderte Automatenmodell angepasst werden. Anschließend kann durch den Generator eine aktuelle Implementation des Automaten erzeugt werden. Es ist auch möglich, Aktionen mit in den Prozess der Generierung einzubauen. Durch die Anwendung des Generators ist der Aufwand der Umsetzung von Änderungen am Automatenmodell minimal.

Der Generator ist in der vorgestellten Form ausschließlich für die Erstellung von hierarchischen Automaten geeignet. In einer der ersten Entwicklungsschritte entstand ein Prototyp für die Erzeugung flacher Automaten, der weiterentwickelt wurde. Um ihn wieder für die Erzeugung eines flachen Automaten zur Verfügung zu stellen ist nur wenig Aufwand nötig. Dieser Generator bietet somit eine gute Möglichkeit, das von mir ausgearbeitete Muster flexibel und effizient einzusetzen.

7 Fazit und Ausblick

Durch die Implementation eines zunächst flachen und später hierarchischen Zustandsautomaten unter der Verwendung des Operators `Placement-new` wird in dieser Arbeit (Kapitel 5) gezeigt, dass es möglich ist, diese Technologie im Rahmen eines Zustands-Entwurfsmusters einzusetzen. Zudem wird gezeigt, dass unter der Verwendung dieser Technologie alle in Kapitel 4 aufgestellten Anforderungen erfüllt werden. Es ergeben sich keine Nachteile gegenüber den vorgestellten Techniken. Während der Auswertung der Implementation ergaben sich sogar Vorteile durch die Anwendung des Operators `Placement-new`.

Bei den in dieser Arbeit vorgestellten Techniken liegt der Fokus auf dem Einsatz in Echtzeitsystemen. Die Eignung der Techniken ist von den in Kapitel 6 aufgeführten Punkten abhängig. Die Auswertung der verschiedenen Techniken ergibt, dass die Methode der Anwendung der geschachtelten `switch-case` Anweisungen im Bereich der Struktur durch die Einfachheit sehr leicht verständlich ist. Ebenso in den Punkten Speicherbedarf und Funktionalität überzeugt diese Technik. Bei der Laufzeit hingegen liegen andere Techniken im Vorteil. Die Technik der Zustandstabelle benötigt im Vergleich zu anderen Techniken wesentlich mehr Speicher, deswegen sind andere Techniken vorzuziehen. Der optimale FSM benötigt sehr wenig Speicherplatz und ist strukturell leicht zu erschließen, jedoch wird die Laufzeit durch die Verwendung einer `switch-case` Stufe erhöht. Das von der GoF vorgeschlagene Muster benötigt mehr Speicher als der optimale FSM, ist aber bezüglich der Laufzeit sehr gut geeignet. Durch den Aufbruch der Kapselung der Klassen und die Vorhaltung aller Zustände, ist es jedoch sinnvoll diesem Muster das von mir entwickelte Muster vorzuziehen. Die Technik des Operators `Placement-new` liegt beim Speicherbedarf knapp hinter dem optimalen FSM und ist damit geringer als bei dem SDP. Die Laufzeit meiner Technik weist eine ebenso gute Leistung wie das Muster der GoF auf. Zusammenfassend komme ich zu dem Schluss, dass das entwickelte Muster durchaus eine sehr gute Alternative zu den analysierten Techniken ist.

Um das Muster weiter zu entwickeln, könnte versucht werden, ein Framework für dieses Muster zu erstellen. Dies würde die Anwendung weiterhin vereinfachen. Weitere Aspekte zur Optimierung des Musters sollten die Realisierung der *tiefen* `History`-Funktion und eine erweiterte Fehlerfunktion sein. Die Fehlerfunktion sollte es ermöglichen, im Fehlerfall zum Ausgangszustand zurückzukehren. Um diese neue Methode auch in anderen Programmiersprachen einsetzen zu können, wäre es sinnvoll nach Möglichkeiten zu suchen, mit denen eine Realisierung des Zustandsübergangs nach diesem Prinzip in Sprachen wie C# und Java möglich ist.

Glossar

Application Programming Interface (deutsch: Programmierschnittstelle): Eine Programmierschnittstelle ist die Schnittstelle, die von einem Softwaresystem weiteren Programmen zur Verfügung gestellt wird.

Dynamik Link Library (deutsch: Programmbibliothek): Eine unter Microsoft Windows verwendete Programmbibliothek, diese hat meistens die Endung `dll`.

Document Object Model (deutsch: Dokumentobjekt - Model): Das Dokument-objekt-Model ist eine Programmierschnittstelle (API) für den Zugriff auf HTML- oder XML-Dokumente. Sie wird vom World Wide Web Consortium definiert.

Document Typ Definition (deutsch: Dokument Typ Definition): Es handelt sich um eine Deklaration für XML-Dokumente, die die Struktur eines solchen Dokuments festlegt.

Enumeration (deutsch: Aufzählung): Ein Aufzählungstyp ist ein Datentyp mit einem endlichen Wertebereich. Alle Werte des Aufzählungstyps werden bei der Deklaration des Typs als Name definiert.

Event Handler (deutsch: Ereignisverarbeiter): Ein Event Handler verarbeite beziehungsweise delegiert Signale oder Ereignisse, die in einem System auftreten.

Framework (deutsch: Rahmenwerk): Als Framework bezeichnet man eine Menge von verknüpften Klassen, welche zusammen ein wieder verwendbares und erweiterbares Gerüst für die Entwicklung von Software eines bestimmten Typs bilden.

Gang of Four (deutsch: Die Viererbande): Die „Gang of Four“ setzt sich aus Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides zusammen. 1995 veröffentlichten sie das Buch *"Design Patterns - Elements of Reusable Object-Oriented Software"*, ein Standardwerk im Bereich Software Engineering über Entwurfsmuster.

Polymorphism (deutsch: Polymorphie): Eine Variable muss nicht nur einen bestimmten Datentyp repräsentieren, es ist auch der Verweis auf andere Klassen möglich. Voraussetzung ist, dass die Variable oberhalb in der Vererbungshierarchie der Klasse des Objektes steht [PomPre04].

State Machine (deutsch: Zustandsautomat): Ein Zustandsautomat verarbeitet interne und externe Ereignisse indem er Aktionen und beziehungsweise oder Transitionen als Reaktion auf die Ereignisse ausführt.

Literaturverzeichnis

[Adamcz03] Adamczyk, Paul: *The Anthology of the Finite State Machine Design Patterns.* EuroPLoP'03. – URL http://pinky.cs.uiuc.edu/~padamczy/docs/plop03.pdf

[Alexan03] Alexandrescu, Andrei: *Modernes C++ Design: Generische Programmierung und Entwurfsmuster angewendet*. Mitp-Verlag/Bonn 2003. – ISBN 3-8266-1347-3

[Dougla99] Douglass, Bruce Powel: *Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns.* Addison Wesley, 1999. – ISBN 0-201-49837-5

[DysAnd96] Dyson, Paul; Anderson, Bruce: *State Patterns.* EuroPLoP'96. – URL http://www.cs.wustl.edu/~schmidt/europlop-96/papers/paper29.ps

[GoF95] Gamma, Erich; Helm, Richard; Johnson, Ralph und Vlissides, John: *Design Patterns: Elements of Object Oriented Software*. Addison-Wesley, 1995. – ISBN 0-201-63361-2

[GurBos99] van Gurp, Jilles; Bosch, Jan: *On the Implementation of Finite State Machines* Proceedings of the 3rd Annual LASTED International Conference Software Engineering and Applications (SEA '99) pp 172-178, 1999 Scottsdale, Arizona. –URL http://publications.jillesvangurp.com/fsm-sea99.pdf

[Harel87] Harel, David: *Statecharts: A Visual Formalism for Complex Systems*. Sci. Comput. Programming 8, 1987, 231-274. – URL http://www.wisdom.weizmann.ac.il/~dharel/SCANNED.PAPERS/Statecharts.pdf

[Harel98] Harel, David und Politi, Michal: *Modeling reactive systems with statecharts: The statement Approach*. McGraw-Hill, 1998. – ISBN 0-07-026205-5

[HarMea04] Harold, Elliote und Means, Scott: *XML in a Nutshell*. O,Reilly Verlag, S. 331 – 355, 2004. – ISBN 3-89721-339-7

[Kahlbr98] Kahlbrandt, Bernd: *Software-Engineering: Objektorientierte Software-Entwicklung mit der Unified Modeling Language.* Springer – Verlag, 1998. – ISBN 3-540-63309-x

[Kalten05] Kaltenhäuser, Heiner. Unterlagen PLP: Laufzeitsystem. HAW Hamburg, 2005. – URL http://www.cpt.haw-hamburg.de/~kaltenhaeuser/Unterlagen%20PLP/Laufzeitsystem/

[Meyers98] Meyers, Scott: *Effektiv C++ programmieren: 50 Wege zur Verbesserung Ihrer Programme und Entwürfe.* Addison Wesley, 1997. – ISBN 3-8273-1305-8

[OdrSog96] Odrowski, J., and P. Sogaard: *Pattern Integration - Variations of State*. EuroPLoP96. – URL http://www.cs.wustl.edu/~schmidt/PLoP-96/odrowski.ps.gz

[OMG04] Object Management Group: *UML Superstructure Specification, v2.0*, 2004. – URL http://www.omg.org/docs/formal/05-07-04.pdf

[PomPre04] Pomberger, Gustav und Pree, Wolfgang: *Software Engineering Architektur-Design und Prozessorientierung*. Carl Hanser Verlag, 2004. – ISBN 3-446-22429-7

[Samek02] Samek, Miro Ph.D.: *Practical Statecharts in C++ - Quantum Programming for Embedded Systems*. CMPBooks, 2002. – ISBN 1-57820-110-1

[SanCam95] Sane, Aamod und Campell,Roy: *Object Oriented State Machines: Subclassing, Composition, Delegation and Genericity*. OOPSLA'95. – URL http://gkarte.isst.fhg.de/~mgrosse/Lehre/Papiere/Komposition/SaneCampbell.pdf

[Schmidt98] Schmidt, Douglas: *Framework Design Rules.* 1998. – URL http://www.cs.wustl.edu/~schmidt/rules.html

[SeGuWa94] Selic, Bran; Gullekson, Garth; Ward, Paul T.: *Real Time Object Oriented Modeling*. John Wiley & Sons, 1994. – ISBN 0-471-59917-4

[Stroustrup97] Stroustrup, Bjarne: *The C++ Programming Language: Third Edition.* Addison Wesley, 1997. – ISBN 0-201-88954-4

[XercesDL] Xerces C++ Downloading – URL http://www.apache.org/dist/xml/xerces-c/binaries/xerces-c_2_7_0-windows_2000-msvc_60.zip

[XercesXML] The Apache XML Project – URL http://xml.apache.org/xerces-c/

[XDOMPar] XercesDOMParser: Constructing a Xerces DOM Parser. – URL http://xml.apache.org/xerces-c/program-dom.html#XercesDOMParser

[YacAmm98a] Yacoub, Sherif M. und Ammar, Hany H.: *Finite State Machine Patterns.* EuroPLoP, 1998. – URL http://www.coldewey.com/europlop98/Program/Papers/Yacoub.ps

[YacAmm98b] Yacoub, Sherif M. und Ammar, Hany H.: *A Pattern Language of Statecharts.* PLoP, 1998. – URL http://jerry.cs.uiuc.edu/~plop/plop98/final_submissions/P22.pdf

Anhang A:

Dieser Arbeit ist eine CD-ROM beigefügt. Die elektronische Version dieser Bachelorarbeit mit dem Dateinamen `Bachelorarbeit_NicoManske.pdf` befindet sich, in Form einer PDF-Datei, im Verzeichnis `Dokumentation`. In dessen Unterverzeichnis `Abbildungen` befinden sich alle verwendeten Abbildungen in Form von JPEGs oder MS Visio-Dateien. Die Doxygen-Dokumentation für den Quellcodegenerator befindet sich ebenso wie die Doxygen-Dokumentation der flachen und hierarchischen Automaten (Prototypen) im Unterverzeichnis D`oxygen`.

Des Weiteren befindet sich im Hauptverzeichnis `Quellcode` der Quellcode für die Prototypen der entwickelten Zustandsautomaten. Mit der Datei `NewStatePattern.sln` kann die komplette Projektmappe im Visual Studio .NET 2003 geöffnet werden. Die Projekte `FSMStatePattern`, `HsmStatePattern`, `HSMTest` und `HsmXml2Pattern` befinden sich in dieser Projektmappe. Alle Projekte könne als Startprojekt festgelegt werden.

Die beiden Prototypen für den flachen und den hierarchischen Automaten sind vom Typ „Windows Anwendung" und können durch das ausführen der zugehörigen Dateien (`FSMStatePattern.exe` und `HsmStatePattern.exe`) gestartet werden. Beim Start des flachen Automaten (`FSMStatePattern.exe`) muss die einzulesende Datei angegeben werden. Der Prototyp des hierarchischen Automaten ist interaktiv und wartet auf Ereignisse (a-h). Mit der Eingabe eines „q" kann dieser Automat beendet werden. Der Prototyp des flachen Zustandsautomaten arbeitet nicht interaktiv, er liest nur Text aus einer Datei ein. Der Generator ist vom Typ „Windows Anwendung" und kann ebenfall über die `.exe`-Datei gestartet werden. Für die Ausführung des Generators sind einige Einstellungen vorzunehmen. Diese Einstellungen sind in der Doxygen-Dokumentation und in der Datei `HsmXml2PatternGen.cpp` als Kommentar beschrieben.

Versicherung über Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach § 22 Abs. 4 ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 17.03.2006 ____________________________

Ort, Datum Unterschrift