Diplomarbeit

Zustandsautomaten zur Vorgangssteuerung in WAM-Anwendungen

Klaus Müller
Rübenkamp 245
22337 Hamburg
4kmuelle@informatik.uni-hamburg.de
Matrikel-Nr. 4725063

Mai 2001

Erstbetreuung : Prof. Dr. Heinz Züllighoven
Zweitbetreuung : Prof. Dr. Winfried Lamersdorf

Fachbereich Informatik
Arbeitsbereich Softwaretechnik
Universität Hamburg
Vogt-Kölln-Straße 30
22527 Hamburg

Erklärung:

Hiermit versichere ich, diese Arbeit selbständig und unter ausschließlicher Zuhilfenahme der in der Arbeit aufgeführten Hilfsmittel erstellt zu haben.

Hamburg, den 18.05.2001

Klaus Müller
Rübenkamp 245
22337 Hamburg
Matrikel-Nr. 4725063

Betreuung:
Prof. Dr. Heinz Züllighoven (Erstbetreuer)
Prof. Dr. Winfried Lamersdorf (Zweitbetreuer)

Prof. Dr. Heinz Züllighoven
Arbeitsbereich Softwaretechnik (SWT)
Fachbereich Informatik
Universität Hamburg
Vogt-Kölln-Straße 30
22527 Hamburg

Prof. Dr. Winfried Lamersdorf
Arbeitsbereich Verteilte Systeme (VSYS)
Fachbereich Informatik
Universität Hamburg
Vogt-Kölln-Straße 30
22527 Hamburg

Danksagung

Ich möchte mich bei Prof. Dr. Heinz Züllighoven für die Erstbetreuung, hilfreiche Tips und zahlreiche Denkanstöße bedanken. Prof. Dr. Winfried Lamersdorf danke ich für die Zweitbetreuung.
Bei Stefan Roock und Henning Wolf vom Arbeitsbereich SWT bedanke ich mich für intensive Diskussionen, Literaturhinweise und Kritik.
Des weiteren danke ich allen Teilnehmern des JWAM-Seminars insbesondere Christian Beis und Martin Lippert für ihre Zeit und Kritik.
Dr. Reinhard Budde, Axel Poigné, Karl-Heinz Sylla von der GMD schulde ich Dank für die Zeit, die sie sich genommen haben, um sich meine Ideen zum Thema Zustandsautomaten anzuhören und mir viele Tips zu geben.
Auch möchte ich meiner Familie und Maren Reißberg danken, die mich bei meiner Arbeit immer unterstützt haben.
Darüber hinaus danke ich all denen, die mir bei der Erstellung dieser Arbeit geholfen haben.

Inhaltsverzeichnis

Kapitel 1: Einleitung

1.1 Motivation

Der *Werkzeug-Automat-Material-Ansatz*[1] strebt die Entwicklung anwendungsorientierter Software an. Software dieser Art zeichnet sich durch eine hohe Gebrauchsqualität aus. Gebrauchsqualität bedeutet hierbei, daß sich die Funktionalität des Systems an den Aufgaben des Anwendungsbereichs orientiert, das System eine benutzergerechte Handhabung realisiert und sich die im System festgelegten Abläufe und Schritte an die jeweilige Anwendungssituation anpassen lassen. Das Ziel ist es, den Anwender in seiner Arbeit möglichst gut durch ein Softwaresystem zu unterstützen[2].

Es gibt verschiedene Gruppen von Anwendern, die jeweils eine andere Art von Arbeit verrichten oder über unterschiedliche fachliche Qualifikationen verfügen und daher unterschiedliche Formen der Unterstützung benötigen. Hieraus folgt, daß auch die Softwaresysteme zu ihrer Unterstützung jeweils anders aufgebaut sein müssen. Im WAM-Ansatz wird dies beim Entwurf eines Systems durch die Verwendung verschiedener Leitbilder berücksichtigt, welche die verschiedenen Typen von Arbeitsplätzen charakterisieren.

Zum einem gibt es das Leitbild des *Arbeitsplatzes für eigenverantwortliche Expertentätigkeit*[3], welches auf die Unterstützung von Expertenarbeit abzielt. Experten verfügen über das zur Erledigung ihrer Aufgaben notwendige Fachwissen und die Kenntnis darüber, welche Reihenfolge bei der Aufgabenerledigung die günstigste ist. Bei ihrer Tätigkeit kann es vorkommen, daß diese Arbeitsvorgänge aufgrund von veränderten Rahmenbedingungen angepaßt werden müssen. Das Hauptaugenmerk eines Softwaresystems zur Unterstützung dieses Arbeitsplatztyps muß also auf einer entsprechend flexiblen Handhabung liegen. Folglich sollten in einem solchen System keine Abläufe vorgegeben werden. Es ist leicht nachvollziehbar, daß eine solche Art von Softwaresystem sehr komplex in ihrer Benutzung sein kann, da der Anwender für eine sinnvolle Verwendung viel Hintergrundwissen über fachliche Zusammenhänge haben muß. Von Experten kann allerdings erwartet werden, daß sie über die entsprechenden Kenntnisse verfügen bzw. sich diese aneignen.

Die Flexibilität in der Handhabung soll sich nicht nur in der Benutzungsoberfläche widerspiegeln, sondern auch in den einzelnen *Komponenten*[4] eines Softwaresystems. Diese sollen möglichst so konzipiert sein, daß sie keine konkreten Abläufe für ihre Benutzung vorgeben.

Neben Systemen, bei denen die Flexibilität im Vordergrund steht, gibt es noch solche, bei denen es im Interesse des Anwenders ist, die Einhaltung von Bearbeitungsvorgängen durch ein System sicherzustellen, um Fehlbedienungen zu vermeiden. Hierbei handelt es sich beispielsweise um Anwendungssysteme, die eine Dienstleistung für solche Anwender zugänglich machen sollen, die über kein Expertenwissen verfügen. Anwendungen dieser Art findet man häufig im World Wide Web, da sich dieses gut dazu eignet, eine Dienstleistung einem großen Anwenderkreis zugänglich zu machen. Betrachtet man solche Web-Anwen-

---

1 Dieser Ansatz wird im Folgenden auch kurz als WAM-Ansatz (WAM = Werkzeug Automat Material) bezeichnet. Einen Überblick über diesen Ansatz findet man bei [Züllighoven 1998].

2 Die Verwendung des Begriffs *Unterstützung* im WAM-Ansatz wird in Abschnitt 2.1.1.1 näher erläutert.

3 vgl. Abschnitt 2.1.1.1 **Arbeitsplatz für eigenverantwortliche Expertentätigkeit**

4 Der Begriff *Komponente* wird in dieser Arbeit immer als *„eine benannte, (wieder)verwendbare Einheit eines Softwaresystems*“ verstanden, welche *„ihre Dienste über eine Schnittstelle zur Verfügung*“ stellt ([Bäumer 1997], Seite 30). Auf eine andere Bedeutung, wie zum Beispiel die Verwendung des Begriffs nach [Szyperski 1997], wird im Bedarfsfall ausdrücklich hingewiesen werden.

dungen wie Homebanking- oder E-Commerce-Systeme[5], so haben die Benutzer eines solchen Systems zwar eine recht genaue Vorstellung von dem Ergebnis der jeweiligen Dienstleistung, es kann aber nicht erwartet werden, daß sie wissen, welche Arbeitsschritte zum Erreichen dieses Ziels notwendig sind. In einem solchen Fall wird zum Beispiel das Leitbild des *Selbstbedienungsautomaten*[6] angewandt. Eine auf der Basis dieses Leitbildes entwickelte Anwendung macht einem Benutzer eine genau festgelegte Dienstleistung verfügbar. Bei der Benutzung sorgt das System dafür, daß der Anwender nur solche Handlungen ausführen kann, die zu dem jeweiligen Stand des Bearbeitungsvorganges möglich sind. Um eine solche Verwendung zu ermöglichen, muß in dem Softwaresystem ein Plan des Bearbeitungsvorgangs vorhanden sein. Dieser Plan legt fest, wann welcher Bearbeitungsschritt in einer Anwendung möglich ist. Dabei sollte der Anwender möglichst immer selbst entscheiden können, ob er einen Vorgang abbricht oder ihn bis zum Ende ausführt.

1.2 Fragestellung

Aus der oben beschriebenen fachlichen Anforderung ergibt sich die technische Fragestellung, wie ein solcher Plan zur Vorgangssteuerung konstruktiv umgesetzt werden kann.
In der Regel können Entwickler hierfür die jeweils aktuellen Werte der Variablen und Objekte einer Anwendung verwenden. Diese bilden in ihrer Gesamtheit den Zustand einer Anwendung. Aufgrund dieses Zustandes kann bestimmt werden, welche fachlichen Aktionen gerade möglich sind.
Dieses stellt sich in der Implementierung so dar, daß eine Aktion erst möglich ist, wenn eine oder mehrere Variablen bestimmte Werte angenommen haben. Dies ist eine gängige Möglichkeit, zustandsabhängiges Verhalten zu modellieren. Allerdings ist ein solches implizites Zustandsmodell nur sehr schwer in seiner Gesamtheit zu überblicken und zu verstehen, wenn es über die gesamte Anwendung verteilt wird. Bestehen nur geringe Abhängigkeiten zwischen den Arbeitsschritten, ist dieses Verfahren noch relativ gut anwendbar. Sobald die Abhängigkeiten jedoch an Komplexität zunehmen, geht die Übersicht in der Regel verloren.
Ein weiterer Nachteil eines impliziten Zustandsmodells ist, daß damit kein fachlicher Benutzungsvorgang wiedergegeben wird, sondern nur geprüft wird, ob ein Wert einer Vorgabe entspricht. Eigentlich soll aber ausgedrückt werden, daß in einem Vorgang zum Beispiel eine Handlung vor oder nach einer anderen Handlung erfolgen soll oder daß zum aktuellen Stand der Bearbeitung eine Handlung nicht möglich ist. Zudem ist es in einem impliziten Zustandsmodell schwer möglich, den Zusammenhang zwischen technischen Variablenwerten und fachlichen Zuständen herzustellen. Dies erschwert das Verständnis des Entwurfs deutlich.
Implizite Zustandsmodelle sind offensichtlich nicht ausreichend, um fachliche Vorgänge zu modellieren. Folglich muß das Zustandsmodell selbst explizit modelliert werden. Dies bedeutet, sowohl die fachlichen Zustände als auch die Zustandsübergänge explizit zu benennen und zu modellieren. Das so gewonnene Modell muß nun in einer entsprechenden Softwarekomponente, welche im Folgenden als Zustandsautomat bezeichnet wird, realisiert werden. Aus den oben genannten Darstellungen ergibt sich die zentrale Fragestellung dieser Arbeit, wie ein solcher Zustandsautomat entworfen werden kann und wie er sich verwenden läßt.

[5] Beispiele für E-Commerce-Systeme finden sich in Abschnitt 3.2.1 **Die Containervoranmeldung** und Abschnitt 3.2.2 **Das Traffix-Projekt.**
[6] vgl. Abschnitt 2.1.1.4 **Selbstbedienungsautomat**

1.3 Zielsetzung

Ziel dieser Arbeit ist es, im Rahmen des WAM-Ansatzes ein Konzept zu entwickeln, mit dem der Entwurf und die Entwicklung von Zustandsautomaten zur Vorgangssteuerung unterstützt werden kann. Das Ergebnis dieser Arbeit soll also der Entwurf eines generischen Grundgerüstes zur Konstruktion von Zustandsautomaten sein.
Daneben soll in dieser Arbeit ein mögliches Vorgehensmodell für die Modellierung von Vorgängen und die Umsetzung dieser Modelle in einen entsprechenden Zustandsautomaten erarbeitet werden.

1.4 Grundlagen der Arbeit

Der nachfolgende Abschnitt gibt einen kurzen Überblick über das Umfeld und die Grundlagen dieser Arbeit.

1.4.1 Der Methodenrahmen

Der Werkzeug-Automat-Material-Ansatz ist ein *Methodenrahmen*[7], welcher eine grundlegende Sichtweise der Softwareentwicklung verkörpert. Der WAM-Ansatz strebt, wie schon erwähnt, die Entwicklung anwendungsorientierter Software an.
Der Ansatz selbst bezieht sich auf bestimmte Klassen von Anwendungsbereichen und gibt Richtlinien für die Auswahl von Methoden, Werkzeugen und Organisationsformen für den Entwurf und die Erstellung von Softwaresystemen vor.
Der WAM-Ansatz bildet die konzeptionelle Grundlage dieser Arbeit. Die verwendeten Konzepte, Methoden und Vorgehensweisen wurden diesem Ansatz entlehnt, welcher in Kapitel 2 noch detaillierter erläutert wird.

1.4.2 Das JWAM-Rahmenwerk

Das JWAM-Rahmenwerk (siehe [JWAM 2001]) ist ein nach dem WAM-Ansatz entworfenes Grundgerüst für die Entwicklung von interaktiven Anwendungen, welches auf der objektorientierten Programmiersprache[8] basiert.
Da heutige Anwendungen aufwendiger zu erstellen sind und gleichzeitig die Zeit, welche für die Erstellung zur Verfügung steht, immer geringer wird, kann kaum eine Anwendung noch völlig neu erstellt werden. Dies bedeutet, daß grundlegende Basisfunktionalitäten oder deren Grundlagen schon vorliegen müssen, um diesen Anforderungen gerecht zu werden. Das *Rahmenwerk*[9] stellt Komponenten mit grundlegenden, immer wieder benötigten Eigenschaften zur Verfügung. Diese Komponenten können dann von Anwendungsentwicklern an ihre jeweiligen Einsatzkontexte angepaßt werden. Ein Schwerpunkt ist die Unterstützung bei der Konstruktion von Werkzeugen und Materialien. Daneben werden noch Komponenten zur Verfügung gestellt, welche die Grundlagen für die Erstellung von *Persistenzmechanismen*[10] oder die Erstellung von *Desktop-Umgebungen*[11] ermöglichen. Darüber hinaus gibt es weitere Komponenten wie beispielsweise die *Fachwerte*[12], welche Schwächen der Programmiersprache Java durch Spracherweiterungen ausgleichen. Eine weitergehende Erläuterung des Rahmenwerkes findet in Abschnitt 2.2 statt.

---

7 nach [Gryczan 1996]
8 JDK 1.3 (siehe [JAVA 2000])
9 Der Begriff *Rahmenwerk* wird in Kapitel 2 definiert.
10 siehe [Havenstein 1999]
11 siehe [Lippert 1999]
12 siehe [Müller 1999]

Das JWAM-Rahmenwerk in seiner aktuellen Version[13] bildet die konstruktive Grundlage dieser Arbeit.

1.5 Gliederung der Arbeit

Dieser Abschnitt beschreibt die grundsätzliche Gliederung der Kapitel und den Aufbau dieser Arbeit.

Kapitel 2: Beschreibung des softwaretechnischen Kontextes
In diesem Kapitel werden einige der wichtigsten Konzepte und Begriffe des WAM-Ansatzes beschrieben und erläutert, um die Basis für das Verständnis der nachfolgenden Kapitel zu schaffen.

Kapitel 3: Das Problemfeld – WAM-Anwendungen im World Wide Web
Als nächster Schritt werden die Gründe für die Verwendung von Zustandsautomaten in Web-Anwendungen ausgeführt.
Aus diesem Grund wird in diesem Kapitel unter Einbeziehung einiger Anwendungsbeispiele geklärt, welche Anforderungen an Web-Anwendungen vor dem Hintergrund des Werkzeug-Automat-Material-Ansatzes gestellt werden. Für dieses Vorgehen wird zunächst beschrieben, was Web-Anwendungen ausmacht und welche Probleme sich bei ihrer Konstruktion ergeben. Des weiteren werden Web-Applikationen in den WAM-Rahmen eingeordnet, um hieraus Erkenntnisse darüber zu gewinnen, um welchen Typ von Arbeitsplatz es sich hier handelt und welche Charakteristika diesen Typ kennzeichnen.
Im weiteren Verlauf des Kapitels wird beschrieben, welche der Konstruktionsprobleme bzw. -anforderungen von Web-Anwendungen sich durch Verwendung von Komponenten des JWAM-Rahmenwerks oder durch Designentscheidungen lösen lassen. Darüber hinaus wird gezeigt, welche Probleme weiter bestehen bleiben bzw. welche neuen Probleme durch die Verwendung von JWAM-Komponenten aufgeworfen werden und wie diese durch den Einsatz von Zustandsautomaten gelöst werden können.

Kapitel 4: Beschreibungsmöglichkeiten für Zustandsmodelle
Dieses Kapitel erläutert die Notwendigkeit, zur Erstellung von Zustandsmodellen einen Formalismus zu verwenden, um diesen als Basis für den Entwurf eines Zustandsautomaten zu nehmen. Des weiteren wird erläutert, warum für diese Arbeit der Statechart-Formalismus als Beschreibungsmittel ausgewählt wurde. Neben der Einführung von Begriffen, Syntax und Konzepten sowie dem Vergleich mit endlichen Automaten wird hier exemplarisch für ein Teilsystem der Containervoranmeldung ein Statechart-Diagramm erstellt.

Kapitel 5: Entwurf des Zustandsautomaten
Dieses Kapitel zeigt, wie sich die mit Statecharts erstellten Zustandsmodelle auf einen Zustandsautomaten abbilden lassen. Innerhalb dieses Rahmens wird die Frage beantwortet, welche Komponenten bzw. Konzepte aus [Gamma 1996] oder [Yacoub 1998] für die noch zu entwickelnden Komponenten verwendet werden können bzw. welche Entwurfsmuster oder Konzepte sich in ihrer gegenwärtigen Form nicht für Zustandsautomaten eignen. Dies bedeutet natürlich auch, die Vor- und Nachteile dieser Entwurfsmuster zu analysieren. Hierauf aufbauend wird beschrieben, welche Entwurfsmuster oder Konzepte in den Entwurf übernommen werden können. Des weiteren werden die in dieser Arbeit entwickelten

[13] Version 1.5 (siehe [JWAM 2001])

Komponenten erläutert. Darüber hinaus wird in diesem Kapitel anhand eines Beispiels die Verwendung der entwickelten Softwarekomponenten erläutert.

Kapitel 6: Diskussion und Ausblick
In Kapitel 6 wird diskutiert, in welchen anderen Anwendungen oder Komponenten des WAM-Ansatzes ein Zustandsautomat noch Verwendung finden kann und welche Konsequenzen dies jeweils hat.

Kapitel 7: Zusammenfassung
In diesem Kapitel werden die wesentlichen Aspekte des in dieser Arbeit entworfenen Konzeptes zur Realisierung von Zustandsautomaten zur Vorgangssteuerung zusammengefaßt. Des weiteren werden an dieser Stelle die noch offenen Punkte bzw. Fragen, die durch diese Arbeit aufgeworfen wurden, geschildert.

1.6 Zur Arbeit

Taucht ein neuer Begriff im Text zum ersten Mal auf, wird dies durch eine *kursive Schreibweise* kenntlich gemacht. Wörtliche Zitate aus der Literatur werden ebenfalls *kursiv* markiert. Werden längere Textpassagen oder Definitionen zitiert, erfolgt dies eingerückt im eigenen Absatz.
Da der überwiegende Teil der für diese Arbeit erfolgten Implementationen mit der objektorientierten Programmiersprache *Java*[14] vorgenommen wurde, entsprechen alle Code-Auszüge der Syntax dieser Sprache. Programmcode, Objekt- und Klassennamen werden durch die Schriftart `Courier New` kenntlich gemacht und entsprechen dem JWAM-Styleguide[15].
Werden Codebeispiele im Text verwendet, so zeigen diese der Übersichtlichkeit halber nur einen Ausschnitt der jeweiligen Schnittstelle bzw. des Codes. Für alle Bezeichnungen und Bezeichner, welche allgemeine Konzepte des Entwurfs beschreiben, werden englische Namen verwendet. Für die konkreten fachlichen Konzepte eines Anwendungsbereiches werden die dort verwendeten Begriffe übernommen.
Interaktions-, Objekt- und Klassendiagramme entsprechen der zur Zeit gebräuchlichen *UML*[16]-Notation.
Diese Arbeit wurde nicht nach den Regelungen der neuen Rechtschreibung, welche sich in der Einführung befinden, sondern nach den alten Regeln abgefaßt, um den Leser nicht mit ungewohnten Schreibweisen zu verwirren.
Als Zugeständnis an die üblichen Lesegewohnheiten werden im Folgenden in der Regel maskuline Bezeichner verwendet, womit natürlich auch immer die weibliche Leserschaft gemeint ist.

[14] siehe [JAVA 2000]
[15] siehe [JWAM 2001]
[16] UML (= Unified Modelling Language) Version 1.3 (siehe [Booch 1999] bzw. [UML 1998])

Kapitel 2: Beschreibung des softwaretechnischen Kontextes

In diesem Kapitel werden einige der wichtigsten Konzepte und Begriffe des WAM-Ansatzes erläutert, um die Basis für das Verständnis der nachfolgenden Kapitel zu schaffen.

2.1 Der WAM-Ansatz

Die folgende Erläuterung der Kernkonzepte sowie einzelner Begriffe des WAM-Ansatzes basiert auf [Züllighoven 1998], in dem die unterschiedlichen Arbeiten zu diesem Thema zusammengefaßt werden, und beschäftigt sich nur mit den für diese Arbeit relevanten Teilen des Ansatzes.
Der WAM-Ansatz ist eine Sammlung von Methoden zur Entwicklung von Software, wobei allerdings kein konkretes Vorgehen vorgegeben wird, sondern der Anwender dieses Ansatzes die Möglichkeit hat, die für seine Anforderungen geeignete Entwicklungsmethode aus den dort beschriebenen Entwurfs- und Konstruktionstechniken selbst zusammenzustellen. Der Ansatz vereinigt Bestandteile wie *Leitbilder*[17] mit *Entwurfsmetaphern*[18], anwendungsorientierte Dokumentationstypen und eine evolutionäre Vorgehensweise mit Prototyping.
Im WAM-Ansatz wird das Ziel verfolgt, die Entwicklung anwendungsorientierter Software zu unterstützen, welche sich durch eine hohe Gebrauchsqualität auszeichnet. Gebrauchsqualität definiert sich hiernach darüber, daß sich die Funktionalität eines Systems an den Aufgaben der Anwender orientiert und die im System festgelegten Abläufe und Schritte sich an verschiedene Anwendungssituationen anpassen lassen. Hierbei soll die Handhabung des Systems aber immer noch benutzergerecht sein.
Die zentrale Idee des Ansatzes ist es, die Gegenstände und Konzepte eines Anwendungsbereiches als Grundlage des zu entwickelnden Softwaresystems zu nehmen. Dadurch fällt es den späteren Anwendern eines Systems leichter, mit dem System umzugehen. Zudem haben Entwickler den Vorteil, daß sie die Anwendungskonzepte einfacher erkennen und modellieren können.
Für den eigentlichen Entwurf werden dem Anwender des Ansatzes verschiedene Leitbilder zur Verfügung gestellt. Diese Leitbilder sollen als Grundlage für die Entwicklung von *Benutzungsmodellen* der jeweiligen Anwendungssysteme verwendet werden.
Unter dem Begriff *Benutzungsmodell* wird im weiteren Verlauf der Arbeit folgendes verstanden:

---

[17] **Definition: Leitbild**
Ein Leitbild ist eine benannte, mit Absicht eingenommene grundsätzliche Sichtweise. Es ist eine Orientierung, die von Menschen angenommen wird, anhand der sie einen Abschnitt von Realität wahrnehmen, wie Entwickler und Benutzer bei der Systementwicklung wechselseitig miteinander umgehen und beschreibt die Gestaltungsziele bei der Softwareentwicklung.
[Gryczan 1996], Seite 223-224

[18] **Definition: Entwurfsmetapher**
Eine Entwurfsmetapher ist eine bildhafte Vorstellung, die ein Leitbild fachlich und konstruktiv konkretisiert. Eine Entwurfsmetapher hat sowohl eine fachliche als auch eine technische Interpretation. Die fachliche Interpretation ist die Interpretation in der Umgangskategorie. Die technische Interpretation ist die Interpretation in der Maschinenkategorie.
[Gryczan 1996], Seite 222

> **DEFINITION: BENUTZUNGSMODELL**
>
> *Ein Benutzungsmodell ist ein fachlich orientiertes Modell darüber, wie Anwendungssoftware bei der Erledigung der anstehenden Aufgaben im jeweiligen Einsatzkontext benutzt werden kann.*
>
> *Das Benutzungsmodell umfaßt eine Vorstellung von der Handhabung und Präsentation der Software, aber auch von den fachlichen Gegenständen, Konzepten und Abläufen, die von der Software unterstützt werden. Es ist sinnvoll, ein Benutzungsmodell auf der Grundlage eines Leitbilds mit Entwurfsmetaphern zu realisieren.*
>
> [Züllighoven 1998], Seite 71

2.1.1 WAM - Leitbilder

Ein Leitbild wird in diesem Zusammenhang als eine *„mit Absicht eingenommene grundsätzliche Sichtweise“* (aus [Gryczan 1996], Seite 223) verstanden, welche als Orientierungsrahmen für Entwickler und Anwender bei der Softwareentwicklung dient. Der WAM-Ansatz umfaßt verschiedene Leitbilder. Am häufigsten verwendet wird hierbei das Leitbild des *Arbeitsplatzes für eigenverantwortliche Expertentätigkeit*. Neben diesem gibt es noch weitere Leitbilder im WAM-Kontext, wie den *Funktionsarbeitsplatz für eigenverantwortliche Expertentätigkeit*, den *Gruppenarbeitsplatz für eigenverantwortliche kooperative Aufgabenerledigung* und den *Selbstbedienungsautomaten*. Die ersten beiden genannten Leitbilder ähneln dem oben beschriebenen Expertenarbeitsplatz, da auch hier Experten in ihrem täglichen Arbeitsablauf von einem System unterstützt werden sollen. Der Selbstbedienungsautomat hingegen stellt einen deutlich anderen Typ von Leitbild dar. Im Nachfolgenden werden die einzelnen Leitbilder nun erläutert.

2.1.1.1 Arbeitsplatz für eigenverantwortliche Expertentätigkeit

Diesem Leitbild und auch allen anderen WAM-Leitbildern, liegt die *unterstützende Sichtweise* zugrunde. Der Begriff *Unterstützung* hat hierbei die folgende Definition:

> **DEFINITION: UNTERSTÜTZUNG**
>
> *Unterstützung menschlicher Arbeit bedeutet, daß Benutzer von Softwaresystemen als Experten ihres Arbeitsgebietes verstanden werden und daß in ihrem Arbeitshandeln Computer als Arbeitsmittel eingesetzt werden. Ein charakteristisches Merkmal dieser Sichtweise ist, daß die Initiative bei der Computeranwendung vom Benutzer ausgeht und daß die Kontrolle über den Ablauf der Arbeitsschritte beim Benutzer liegt.*
>
> [Gryczan 1996], Seite 226

Die Anwender eines Systems werden als Experten ihres jeweiligen Fachgebietes aufgefaßt, die von einem System in ihrer Tätigkeit unterstützt werden sollen. Des weiteren muß sich das Benutzungsmodell eines Systems an die jeweilige Anwendungssituation anpassen lassen. Die Charakteristika eines solchen Systems lassen sich wie folgt beschreiben:

- *Experten erledigen häufig wechselnde Aufgaben von hoher Komplexität.*
- *Die Arbeit erfordert einen hohen Grad an Qualifikation, Kenntnissen und Erfahrung, der nicht formalisiert werden kann.*

- *Die Erledigung von Aufgaben orientiert sich zielgerichtet an Arbeitsmitteln und -gegenständen und nicht an vorgegebenen Routinen.*
- *Die situative Auswahl von Arbeitsschritten wird durch die vorhandenen Arbeitsmittel unterstützt und führt zu jeweils adäquaten Ergebnissen.*
- *Pläne werden zur Orientierung und nicht als ausführbare Handlungsvorschrift betrachtet.*
- *Die Kontrolle über die Handlung bleibt beim Menschen.*
  [Züllighoven 1998], Seite 81

Die Benutzer eines solchen Systems werden wie folgt charakterisiert:

- *Benutzer sind Fachleute in ihrem Anwendungsgebiet.*
- *Der Umgang mit neuen Softwarewerkzeugen und -materialien ist für Benutzer zunächst ungewohnt. Er erfordert Übung und Wissen.*
- *Da die betrachteten Aufgaben situationsabhängig und eigenverantwortlich erledigt werden müssen, dürfen die dabei verwendeten Softwarewerkzeuge keine durchgängigen Arbeitsabläufe implementieren.*
- *Je nach Ausbildungsumfang der Benutzer und Häufigkeit des Einsatzes können Werkzeuge unterschiedlich komplex in Funktionalität und Handhabung gestaltet sein.*
  [Züllighoven 1998], Seite 91

Damit grenzen sich nach dem WAM-Ansatz entworfene Systeme von anderen Systemen ab, denen eine *ablaufsteuernde Sichtweise* zugrunde liegt. Bei solchen ablaufsteuernden Systemen liegt die Kontrolle über den Ablauf von Arbeitshandlungen bzw. Entscheidungen beim System. Der Mensch wird nur zur Dateneingabe benötigt. Der Begriff der *Ablaufsteuerung* wird im weiteren Verlauf der Arbeit wie folgt verwendet:

> **DEFINITION: ABLAUFSTEUERUNG**
>
> *Ablaufsteuerung bedeutet, Operationen zu implementieren, die menschliches Arbeitshandeln regeln und kontrollieren. Die Kontrolle über den Ablauf der Arbeitsschritte des Menschen liegt bei der Maschine. Generell wird bei der Automatisierung das Ziel verfolgt, menschliches Arbeitshandeln durch Maschinen zu ersetzen oder bis auf notwendige Dateneingaben (Input) zu reduzieren. Dazu werden Pläne und Vorschriften verwendet, die im Idealfall durch Algorithmen auf Maschinen implementiert werden.*
>
> [Gryczan 1996], Seite 221

Bei [Züllighoven 1998] werden Leitbilder nach den unterschiedlichen Wahlmöglichkeiten bei der Gestaltung der Systeme unterschieden. Hierbei gibt es drei Freiheitsgrade:

- **Grad der Verkopplung von Mensch und Maschine**, d.h. im WAM-Kontext der Umfang, in dem die Aufgaben von der Anwendungssoftware unterstützt werden.
- **Produktdistanz** meint das Ausmaß, in dem eine Dienstleistung oder ein Produkt vom Anwender eigenverantwortlich realisiert werden kann.
- **Prozeßverkopplung** bedeutet, inwieweit die einzelnen Arbeitsabläufe durch die Software vorgegeben sind und in einem Gesamtvorgang (im Sinne von Workflow) eingebunden sind.

Abbildung 2.1: Freiheitsgrade bei der Gestaltung von Softwaresystemen nach [Frei 1996]

Nachdem den in Abbildung 2.1 dargelegten Kriterien ergibt sich für den oben beschriebenen Arbeitsplatztyp folgende Eingruppierung: Dieser Arbeitsplatztyp zeichnet sich durch eine mittlere Verkopplung, eine geringe Produktdistanz und eine minimale Verkettung aus.

2.1.1.2 Funktionsarbeitsplatz für eigenverantwortliche Expertentätigkeit

Funktionsarbeitsplätze[19] zeichnen sich dadurch aus, daß wenige vorgegebene Aufgaben in häufiger Wiederholung erledigt werden müssen. Hierfür ist es notwendig, daß die Arbeiten mit hoher Geschwindigkeit erledigt werden können. Bei diesen Arbeitsplätzen steht eine einfache Benutzbarkeit im Vordergrund, dennoch handelt es sich um einen Arbeitsplatz, mit dem Experten eines Anwendungsgebietes unterstützt werden sollen.
Die Softwarewerkzeuge, welche für eine solche Anwendung erstellt werden, können sehr viel besser an die Arbeitsabläufe angepaßt werden, da hier nur wenige spezialisierte Aufgaben zu erledigen sind. Für solche Arbeitsplatztypen kann es sinnvoll sein, einen Steuerungsautomaten zu bauen, welcher bei der Abfolge der Arbeitsschritte immer das richtige Softwarewerkzeug zum richtigem Moment öffnet. Daneben muß es aber auch möglich sein, in Fällen, in denen es fachlich notwendig ist, vom herkömmlichen Arbeitsablauf abzuweichen. Die Werkzeuge an sich haben den Charakter von Spezialwerkzeugen, welche einen hohen Grad an Fachwissen verlangen.
Werden die in Abbildung 2.1 aufgeführten Freiheitsgrade auf diesen Arbeitplatztyp angewendet, stellt man fest, daß eine hohe Verkopplung von Mensch und Maschine vorhanden ist. Die Produktdistanz ist gering. Die Prozeßverkettung hingegen ist im normalen Arbeitsablauf etwas höher als beim oben erwähnten Expertenarbeitsplatz.

2.1.1.3 Gruppenarbeitsplatz für eigenverantwortliche kooperative Aufgabenerledigung

Dieser Arbeitsplatztyp[20] dient dazu, viele Experten unterschiedlicher Ausrichtungen in ihrer Arbeit zu unterstützen. Dabei geht es vor allem darum, daß es ihnen ermöglicht wird, ihre eigenen Tätigkeiten mit denen anderer Experten zu koordinieren. In der Regel arbeiten die Anwender hierbei zusammen an der Erledigung einer gemeinsamen Aufgabe, wobei jeder eine Teilaufgabe zum richtigen Zeitpunkt, in Abstimmung mit den anderen Beteiligten zu erledigen hat. Hierbei kann es auch vorkommen, daß die fachliche Qualifikation der Beteiligten sowie deren IT-Kenntnisse unterschiedlich sind. Für ein Softwaresystem bedeutet dies, daß neben der Anforderung, eine Vielfalt von Aufgaben zu unterstützen, auch erhebliche sicherheitstechnische Anforderungen erfüllt werden müssen. Bestimmte Teile eines solchen Softwaresystems dürfen also nur berechtigten Anwendern zugänglich gemacht werden.
Der Grad der Verkopplung von Mensch und Maschine und die Prozeßdistanz sind gering, während die Prozeßverkettung zwischen einem sehr geringen und einem mittleren Grad schwankt.

2.1.1.4 Selbstbedienungsautomat

Ein Selbstbedienungsautomat[21] ist zunächst kein Arbeitsplatz im engsten Sinne des Wortes, da er nur gelegentlich im Rahmen einer Dienstleistung von einem Anwender bzw. Kunden in Anspruch genommen wird.
Der Selbstbedienungsautomat macht dem Anwender eine genau festgelegte Dienstleistung einer Anwendungsdomäne verfügbar. Bei der Benutzung des Automaten ist der Anwender zwar für sein Handeln selbst verantwortlich, aber da er keine vertieften Kenntnisse über

[19] vgl. [Züllighoven 1998], Seite 94-95
[20] vgl. [Züllighoven 1998], Seite 96-97
[21] vgl. [Züllighoven 1998], Seite 97-98

den Anwendungsbereich und unter Umständen auch kein Wissen im Umgang mit dieser Kategorie von Softwaresystemen hat, muß er die Leistung trotzdem auf schnelle, sichere und einfache Weise in Anspruch nehmen können. Daraus folgt, daß der Anbieter des Dienstes eine erhöhte Sorgfalts- und Informationspflicht dem Anwender gegenüber hat. Dies muß sich in einer geeigneten Führung des Benutzers bei der Nutzung des Werkzeuges und in einer ausreichenden Information über die Vorgänge im System zeigen. Eine solche Benutzerführung wird durch einen Zustandsautomaten zur Vorgangssteuerung realisiert. Dabei sollte der Anwender im Sinne der unterstützenden Sichtweise immer selbst entscheiden können, ob er einen Vorgang abbricht oder ihn bis zum Ende ausführt. Er muß auch immer über den Zustand des Systems informiert werden. Es muß ihm also bekannt gemacht werden, ob seine Handlungen erfolgreich waren oder ob sie fehlgeschlagen sind. Falls Letzteres der Fall ist, muß er natürlich eine Information darüber erhalten, warum die Dienstleistung nicht erbracht werden konnte.
Ein Beispiel für einen derartigen Arbeitsplatztyp ist eine Homebanking-Anwendung. Hierbei hat der Kunde eine sehr genaue Vorstellung von der Dienstleistung, die erbracht werden soll (zum Beispiel das Vornehmen einer Überweisung), allerdings fehlt ihm das Detailwissen über die notwendigen Materialien (hier: Konto bzw. Überweisungsformular).
Bei der Arbeit an einem solchen Arbeitsplatz sind Informationen darüber, ob ein Teilschritt erfolgreich war oder nicht, wichtig für die Benutzbarkeit des Werkzeuges.
Der Grad der Verkopplung von Mensch und Maschine ist entsprechend den Anforderungen an diesen Arbeitsplatztyp sehr hoch. Die Prozeßdistanz ist gering, da der Kunde wissen muß, welche Dienstleitung erbracht wird und was dies für ihn bedeutet. Die Prozeßverkettung ist hoch, da von den vorgegebenen Abläufen kaum abgewichen werden kann.

2.1.2 WAM - Entwurfsmetaphern

Um die verwendeten Leitbilder weiter zu konkretisieren, werden verschiedene Entwurfsmetaphern eingeführt. Diese beschreiben jeweils ein Konzept oder eine Komponente eines Anwendungssystems durch einen Gegenstand der Anwendungswelt. Die zentralen Entwurfsmetaphern des WAM-Ansatzes sind *Werkzeug*, *Automat* und *Material*. Da sie grundlegend für den gesamten Ansatz sind, werden sie nachfolgend erläutert.
Neben diesen drei namensgebenden Entwurfsmetaphern beinhaltet der WAM-Ansatz natürlich noch weitere Metaphern wie *Arbeitsumgebung*, *Fachwerte*, *Vorgangsmappe*, *Laufzettel*, *Fachliche Services* usw. Von diesen Entwurfsmetaphern wird in diesem Abschnitt nur noch der Begriff der Arbeitsumgebung erläutert. Das Konzept der *Fachlichen Services* wird in Abschnitt 2.1.4 erläutert, nachdem in Abschnitt 2.1.3 bei der Werkzeugkonstruktion die Grundlagen zum Verständnis dieses Abschnittes gelegt worden sind. Alle anderen Begriffe werden bei Bedarf erklärt.

2.1.2.1 Material

Unter einem Material wird ein Gegenstand verstanden, welcher ein fachliches Konzept oder einen Gegenstand einer Anwendungsdomäne darstellt. Diese Gegenstände werden über ihr fachliches Verhalten modelliert. Ein Beispiel für ein solches Material ist ein Konto, wie es im Kontext von Büroarbeit häufig vorkommt. Eine fachliche Operation wäre in diesem Fall das Einzahlen eines Geldbetrages auf dieses Konto.

> **DEFINITION: MATERIAL**
>
> *Ein Material ist ein Gegenstand, der im Rahmen einer Aufgabe Teil des Arbeitsergebnisses wird. Materialien werden durch Werkzeuge und Automaten bearbeitet und verkörpern fachliche Konzepte. Sie müssen für eine Bearbeitung geeignet sein.*
>
> [Züllighoven 1998], Seite 86

2.1.2.2 Werkzeug

Um Materialien für einen Anwender nutzbar zu machen, muß es eine Möglichkeit geben, diese zu verändern bzw. zu sondieren. Hierzu werden Werkzeuge verwendet. Werkzeuge vergegenständlichen also Handlungen oder Tätigkeitsfolgen an Materialien. Es sind folglich Gegenstände, mit denen Materialien verändert und sondiert werden können. Der Umgang mit Werkzeugen erfordert vom Anwender, daß er sich mit diesem Werkzeug vertraut macht.

> **DEFINITION: WERKZEUG**
>
> *Ein Werkzeug ist ein Gegenstand, mit dem Menschen im Rahmen einer Aufgabe Materialien verändern oder sondieren können. Werkzeuge eignen sich meist für verschiedene fachliche Zwecke und für die Arbeit an unterschiedlichen Materialien.*
>
> [Züllighoven 1998], Seite 84

2.1.2.3 Automat

Der Arbeitsplatz eines Menschen enthält Dinge, die sich nicht als Material oder als Werkzeug charakterisieren lassen. Der Benutzer löst an einem solchen Gegenstand eine Aktion aus und dieser produziert ein Ergebnis. Der Benutzer hat hierbei nicht die Möglichkeit, in den eigentlichen Vorgang einzugreifen. Gegenstände dieser Art werden mit der Metapher des *Automate*n charakterisiert. Ein Beispiel für einen solchen Gegenstand ist ein Drucker.

> **DEFINITION: AUTOMAT**
>
> *Ein Automat ist ein Arbeitsmittel im Rahmen einer Aufgabe, um Material zu bearbeiten. Automaten erledigen lästige Routineaufgaben als eine definierte Folge von Arbeitsschritten mit festem Ergebnis ohne weitere äußere Einflüsse.*
>
> [Züllighoven 1998], Seite 89

2.1.2.4 Arbeitsumgebung

Nach [Gryczan 1996] findet qualifizierte fachliche Arbeit immer an einem gesonderten Ort statt, wo alle notwendigen Materialien und die zur Bearbeitung notwendigen Werkzeuge zur Verfügung stehen. Die Materialien und Werkzeuge sind dabei jeweils so angeordnet, daß sie den Anwender in seiner Tätigkeit unterstützen. Diese Anordnung sollte sich an die jeweiligen Erfordernisse der zu erledigenden Aufgaben anpassen lassen.
Veränderungen an Materialien durch Werkzeuge sollten für den Anwender immer sofort sichtbar gemacht werden, um ihm eine unmittelbare Auskunft darüber zu geben, welche Auswirkungen sein Handeln hat.

> **DEFINITION: ARBEITSUMGEBUNG**
>
> *Die Arbeitsumgebung ist ein Ort, wo Werkzeuge, Automaten und Materialien, die bei der Erledigung von Aufgaben griffbereit sein müssen, fachlich motiviert angeordnet werden müssen. Die eigentliche Arbeit findet am Arbeitsplatz statt. Zur Umgebung gehören noch die Orte, die unmittelbar zugänglich sind.*
>
> [Züllighoven 1998], Seite 87

2.1.2.5 Zusammenhang

Um den Zusammenhang zwischen Werkzeugen, Materialien und Automaten nochmals deutlich zu machen, wird bei [Roock&Wolf 1998] der Begriff des *Gegenstands* eingeführt.

> **DEFINITION: GEGENSTAND**
>
> *Ein Gegenstand ist etwas, mit dem Menschen umgehen. Werkzeuge, Materialien und Automaten sind Gegenstände.*
>
> [Roock&Wolf 1998], Seite 11

Die nachfolgende Abbildung 2.2 zeigt die Beziehungen der eben geschilderten Entwurfsmetaphern unter Verwendung des Begriffs des Gegenstands:

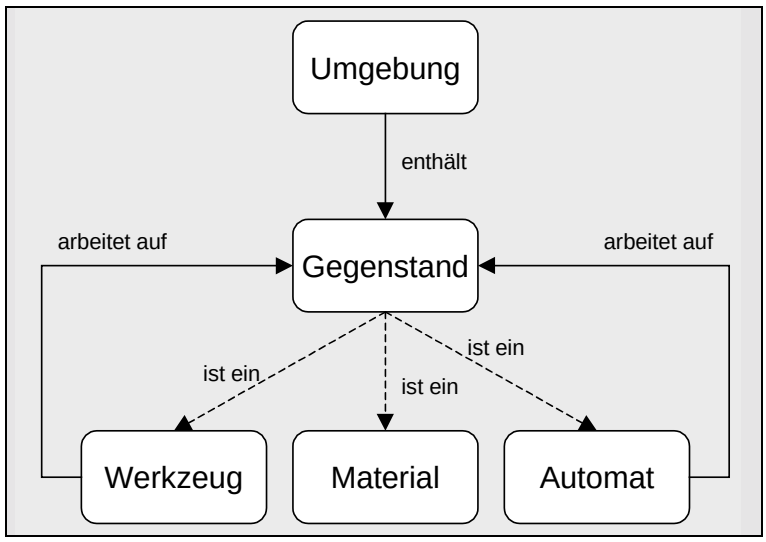

Abbildung 2.2: Zusammenhang von Werkzeug, Material, Automat und Arbeitsumgebung[22]

2.1.3 Werkzeugkonstruktion

Nachdem im vorigen Abschnitt die Entwurfsmetapher des Werkzeugs als solche erläutert wurde, wird nun geschildert, wie (Software-)Werkzeuge nach dem WAM-Ansatz konstruiert[23] werden.

Ein Softwarewerkzeug präsentiert dem Benutzer ein Material. Durch die Funktionalität des Werkzeugs wird das Material für einen Anwender zugänglich gemacht. Die Funktion beschreibt also, wozu das Werkzeug fachlich gut ist. Die Interaktion beschreibt, wie das Werkzeug gehandhabt wird, wie es sich dem Anwender präsentiert und wie es das zu bearbeitende Material darstellt. Bei der Konstruktion von Werkzeugen (siehe [Züllighoven

[22] vgl. [Roock&Wolf 1998]
[23] vgl. [Züllighoven 1998], Seite 234-301

1998], Seite 234ff) werden Werkzeuge in einen interaktiven und in einen funktionellen Bereich aufgeteilt. Dieses Prinzip wird im WAM-Ansatz als *Trennung von Interaktion und Funktion* bezeichnet. Beide Bereiche des Werkzeugs sollen weitgehend unabhängig voneinander modelliert und konstruiert werden können, so daß es möglich ist, den interaktiven Bereich zu verändern, ohne Anpassungen am Funktionsteil vornehmen zu müssen. Durch die Anwendung dieses Konstruktionsprinzips sind so konstruierte Systeme einfacher zu warten bzw. anzupassen, und Teilsysteme können einfacher wiederverwendet werden. Darüber hinaus wird die Verständlichkeit des Entwurfs verbessert.
Um dieses Prinzip umzusetzen, müssen fachliche Funktionalität und Benutzerinteraktion getrennt werden. Der funktionelle Teil wird als *Funktionskomponente* (Abkürzung: FK) bezeichnet. Der interaktive Teil wird mit der Bezeichnung *Interaktionskomponente* (Abkürzung: IAK) versehen.
Die Funktionskomponente ist der bewirkende und sondierende Teil des Werkzeugs. In ihr wird die fachliche Funktionalität festgelegt. Die Funktionskomponente bearbeitet das Material und kennt den Anwendungskontext des Werkzeugs. Des weiteren verwaltet die Funktionskomponente das Werkzeuggedächtnis, welches den aktuellen Arbeitszustand beinhaltet. Das Werkzeug entscheidet aufgrund des Werkzeuggedächtnisses, welche Arbeitsschritte als nächstes möglich sind.
Die Interaktionskomponente legt die Benutzungsschnittstelle des Werkzeugs fest. Sie nimmt die durch Benutzeraktionen ausgelösten Ereignisse entgegen, ruft die Funktionskomponente auf und steuert die Repräsentation an der Oberfläche.
Ein Werkzeug ist technisch als ein reaktives System aufgebaut. Explizite Aktionen eines Benutzers, zum Beispiel durch einen Mausklick oder einen Tastenanschlag, lösen immer eine Reaktion des Werkzeugs aus. Die Aktionen werden als ein Strom von Ereignissen am Werkzeug registriert. Die Interaktionskomponente setzt die Ereignisse in Aufrufe der Funktionskomponente um. Die Funktionskomponente bearbeitet in geeigneter Weise das Material. Damit ist die primäre Kontollflußrichtung in einem Werkzeug festgelegt. Abbildung 2.3 zeigt eine vereinfachte Darstellung eines Werkzeuges nach dem WAM-Ansatz.

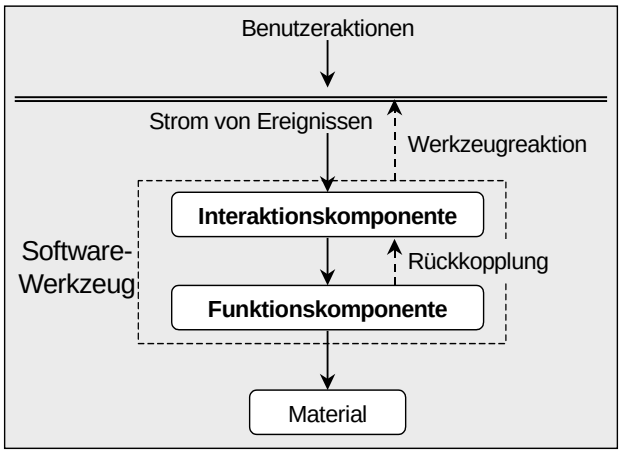

Abbildung 2.3: Konstruktion von Werkzeugen nach dem WAM-Ansatz[24]

Da Funktions- und Interaktionskomponente ständig Informationen über die Bearbeitung eines Materials durch den Benutzer austauschen müssen, es aber keine zyklische Benutzt-Beziehung zwischen beiden geben soll, müssen Funktions- und Interaktionskomponente so miteinander kombiniert werden, daß die Interaktionskomponente ihre Funktionskomponente kennt und benutzt, die Funktionskomponente aber möglichst keinerlei Kenntnis von ihrer Interaktionskomponente hat. Um dies zu realisieren, wird das *Observer-Pattern* (siehe [Gamma 1996], Seite 293ff) verwendet. Die Funktionskomponente des Werkzeugs benachrichtigt die Interaktionskomponente bei einer Zustandsänderung des von ihr bearbeiteten Materials, so daß diese das veränderte Material geeignet darstellen kann.
Jedes Werkzeug hat eine Form der Präsentation und eine Form der Handhabung. Die Form der Präsentation bestimmt, auf welche Art und Weise das Werkzeug für einen Anwender

[24] aus [Züllighoven 1998], Seite 247

dargestellt wird. Die Handhabung bestimmt den Umgang mit dem Werkzeug. Die *Trennung von Handhabung und Präsentation* eines Werkzeugs mildert das Problem der Abhängigkeit einer Anwendung von konkret verwendeten Oberflächenbibliotheken oder GUI[25]-Toolkits. Damit eine Interaktionskomponente von den konkreten Handhabungsformen und dem verwendeten Fenstersystem abstrahieren kann, benutzt sie für die Handhabung und Präsentation sogenannte Interaktionstypen. In diesen werden verschiedene Arten der Handhabung eines Werkzeugs gekapselt. Die Interaktionskomponente des Werkzeugs geht nur mit den Interaktionstypen um. Dies hat zur Folge, daß bei Änderung des GUI-Toolkits nur die betroffenen Interaktionstypen angepaßt werden müssen.

Werkzeuge können miteinander kombiniert werden, um so aus einfachen Werkzeugen Kombi-Werkzeuge für komplexe Handlungen aufzubauen (siehe Abbildung 2.4). Hierbei werden den Sub-Werkzeugen Teilaufgaben übergeben. Dies wird als *Werkzeugkomposition* bezeichnet. Die Werkzeugkomposition regelt das Zusammenspiel von Sub-Werkzeugen und ihren übergeordneten Kombi-Werkzeugen. Durch dieses Konstruktionsprinzip wird die Übersichtlichkeit eines Entwurfs verbessert. Zudem können Sub-Werkzeuge in anderen Anwendungskontexten wiederverwendet werden.

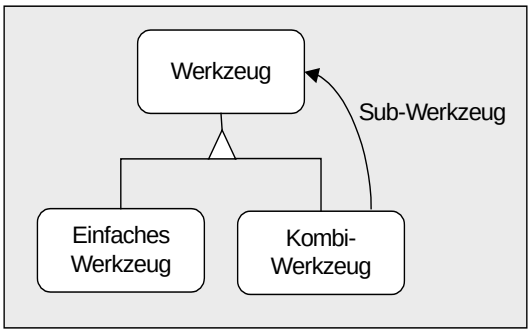

Abbildung 2.4: Werkzeugkomposition[26]

2.1.4 Fachliche Services

Fachliche Services sind ein von [Otto&Schuler 2000] entworfenes Konzept zur Kapselung *fachlichen Wissens*. Das Ziel hierbei ist die Schaffung eines allgemeinen Konzeptes, welches sich für unterschiedlichste Benutzungsschnittstellen eignet.
Unter dem Begriff *fachliches Wissen* wird in diesem Zusammenhang folgendes verstanden:

> **DEFINITION: FACHLICHES WISSEN, FACHLICHE FUNKTIONALITÄT, GESCHÄFTSLOGIK**
>
> *Unter fachlicher Funktionalität verstehen wir das in eine konkrete Implementation gefaßte Anwendungswissen der Entwickler [..]. Die fachliche Funktionalität birgt fachliches Wissen (auch: Fachwissen), und auch der Umgang mit fachlicher Funktionalität erfordert ein gewisses Maß Fachwissen. Im Umfeld komplexer Unternehmenssoftware nennt man fachliche Funktionalität auch Geschäftslogik (engl. Business Logic).*
>
> [Otto&Schuler 2000], Seite 45

Das JWAM-Rahmenwerk wurde ursprünglich zur Konstruktion von Werkzeugen für Desktop-Anwendungen[27] entworfen. Allerdings ist bei verschiedenen Projekten, welche mit dem Rahmenwerk durchgeführt wurden, deutlich geworden, daß unterschiedlichste Benutzungsschnittstellen wie Desktop oder Webtop[28] zu unterstützen sind. Hierfür reicht die ursprüngliche Werkzeugkonstruktion nicht mehr aus. Werden für die unterschiedlichen Be-

[25] GUI = Graphical User Interface (vgl. [Lippert 1999], Seite 4)
[26] aus [Züllighoven 1998], Seite 280
[27] **Definition: Desktop-Anwendung** (siehe [Otto&Schuler 2000], Seite 20)
[28] vgl. Abschnitt 3.1 **WAM-Anwendungen im World Wide Web**

nutzungsschnittstellen eigene Komponenten entwickelt, tritt der Effekt auf, daß fachliches Wissen mehrfach implementiert wird (siehe Abbildung 2.5). Hierdurch wird die Weiterentwicklung und die Wartung der jeweiligen Systeme erschwert.

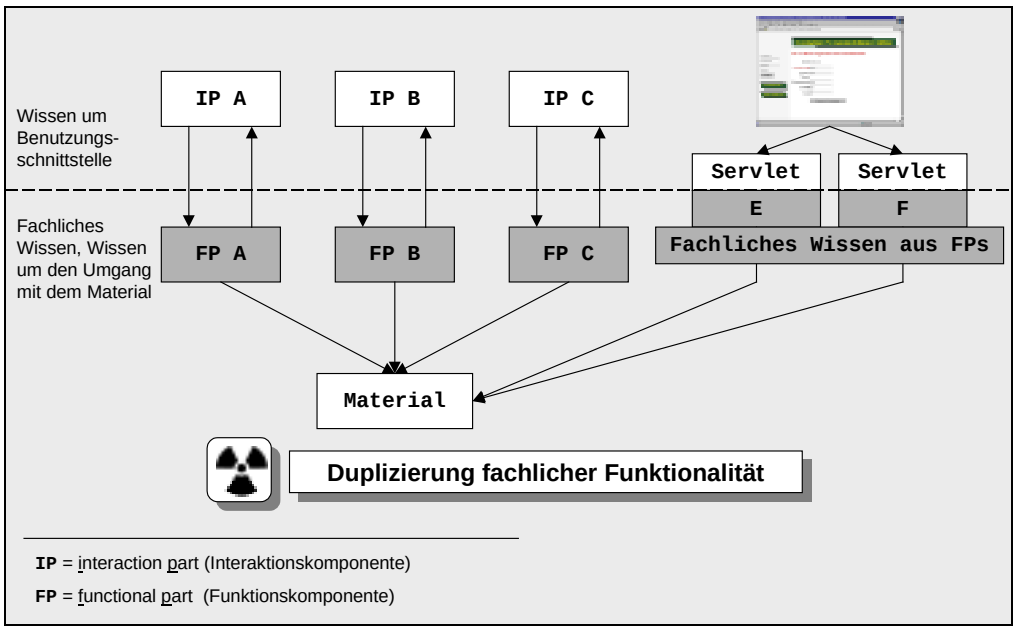

Abbildung 2.5: Duplizierung fachlicher Funktionalität bei der Unterstützung unterschiedlicher Benutzungsschnittstellen[29]

Um dieses Problem zu lösen, wurde das Konzept der Fachlichen Services eingeführt. Nachfolgend wird eine Definition dieses Konzepts gegeben:

> **DEFINITION: FACHLICHER SERVICE**
>
> *Ein Fachlicher Service faßt kohärente Funktionalität und Wissen eines Anwendungsbereichs unabhängig von einer Benutzungsschnittstelle zusammen, kapselt sie und stellt sie als Dienstleistungsbündel zur Verfügung.*
>
> *Die Dienstleistungen eines Fachlichen Services werden über Aufträge von Konsumenten in Anspruch genommen.*
>
> [Otto&Schuler 2000], Seite 67

Eine Dienstleistung wird also durch das Benennen eines Auftrages initiiert. Ein Auftrag wird wie folgt definiert:

> **DEFINITION: AUFTRAG, KONSUMENT**
>
> *Ein Auftrag ist die Benennung einer zu erbringenden Leistung. Der den Auftrag erfüllende Dienst hat einen gewissen Mindestumfang oder eine bestimmte minimale Komplexität. Ein Auftrag wird von einem Konsumenten vergeben und ist nicht ohne diesen denkbar.*
>
> [Otto&Schuler 2000], Seite 44

Unter dem Begriff Dienstleistung[30] wird in diesem Zusammenhang folgendes verstanden:

[29] vgl. [JWAM 2001]

[30] Die Begriffe *Service*, *Dienst* und *Dienstleistung* werden synonym verwendet (vgl. [Otto&Schuler 2000]).

> **DEFINITION: DIENSTLEISTUNG , SERVICE, ANBIETER**
>
> *Ein Service erbringt auf Anforderung Leistungen, die durch Aufträge benannt werden. Er übernimmt für den Konsumenten die Verantwortung in der Ausführung der Aufträge. Wie ein Service eine Leistung erbringt, ist für den Konsumenten nicht erkennbar. Als Gegenstück zum Konsumenten nennen wir einen Service Anbieter einer Leistung.*
>
> [Otto&Schuler 2000], Seite 45

Die nachfolgend aufgeführten Anforderungen beschreiben die Eigenschaften von Fachlichen Services:

- *Fachliche Services sollen das fachliche Wissen eines Anwendungsbereichs zusammengefaßt und gekapselt anbieten.*
- *Fachliche Services sollen das fachliche Wissen eines Anwendungsbereichs handhabungs- und präsentationsunabhängig anbieten.*
- *Fachliche Services sollen das fachliche Wissen eines Anwendungsbereichs mehrbenutzerfähig und verteilt anbieten.*
- *Fachliche Services sollen das fachliche Wissen eines Anwendungsbereichs mit einem der Aufgabe und technischen Umgebung angemessenen Zustands- und Sitzungsmodell anbieten.*
- *Fachliche Services sollen das fachliche Wissen eines Anwendungsbereichs wiederverwendbar und austauschbar anbieten.*
  [Otto&Schuler 2000], Seite 75-81

Fachliche Services können von unterschiedlichen Benutzungsschnittstellen (siehe Abbildung 2.6) verwendet werden. Sie werden also nie direkt von einem Anwender benutzt, sondern stets durch Werkzeuge oder andere softwaretechnische Konstrukte verwendet. Diese kapseln das Wissen um die Besonderheiten der jeweiligen Benutzungsschnittstelle.

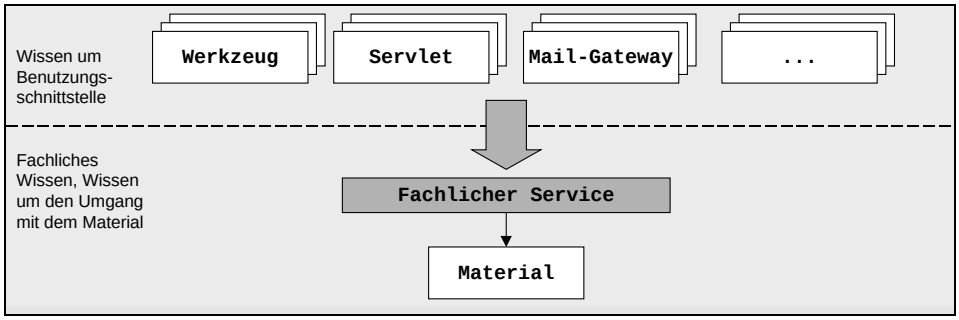

Abbildung 2.6: Verwendung Fachlicher Services durch unterschiedliche Benutzungsschnittstellen[31]

Fachliche Services werden aufgrund der durch sie angebotenen Dienste in zwei grundsätzliche Ausprägungen unterteilt:

- *Bereitstellung von Material und Umgang: Fachliche Services dieser Art bieten Dienste, die Handhabung und oft auch Aufbewahrung von Gegenständen der Anwendung betreffen.*

[31] nach [JWAM 2001]

- *Vergegenständlichung von Vorgängen: Fachliche Services dieser Art kapseln fachliches Wissen über komplexe Zusammenhänge, Implikationen und Zuständigkeiten von Vorgängen in der Anwendungswelt.*
  [Otto&Schuler 2000], Seite 160

Diese Ausprägungen sind allerdings kaum in ihrer reinen Form zu finden. In der Regel sind Fachliche Services Mischformen der beiden oben genannten Formen.

2.2 Das JWAM-Rahmenwerk

Bei dem JWAM-Rahmenwerk handelt es sich um eine Umsetzung der Ideen und Konzepte des WAM-Ansatzes unter Verwendung der objektorientierten Programmiersprache Java. Unter einem Rahmenwerk wird im WAM-Ansatz folgendes verstanden:

> **DEFINITION: RAHMENWERK**
>
> *Ein Rahmenwerk stellt eine softwaretechnische Lösung für eine Reihe verwandter Probleme zur Verfügung. Statisch besteht es aus einer Menge von Klassen, die, bezogen auf das zu lösende Problem, das Zusammenspiel (Kontrollfluß) einer Gruppe von Objekten zur Laufzeit eines Softwaresystems festlegen. Im Unterschied zu Entwurfsmustern sind Rahmenwerke implementierte Einheiten. Sie ermöglichen somit die Wiederverwendung von Analysen, Entwurfsentscheidungen, Architekturen und Implementierungen. Ein Rahmenwerk definiert dabei auch die Stellen, an denen seine Funktionalität erweitert und angepaßt werden kann.*
>
> [Bäumer 1997], Seite 93

Das JWAM-Rahmenwerk stellt allgemeine Basiskomponenten für die Erstellung interaktiver Softwaresysteme zur Verfügung. In dieser Arbeit kann nur ein kurzer Überblick über den Aufbau und die Implementierung gegeben werden[32]. Falls auf die Eigenschaften oder Implementierungen bestimmter Elemente Bezug genommen wird, werden diese dann gesondert erläutert.

2.2.1 Die Grundidee

Die Entwicklung von Softwaresystemen ist durch die Anforderung, immer komplexere und anspruchsvollere Arbeitsabläufe zu unterstützen und durch verbesserte Möglichkeiten zur Gestaltung (zum Beispiel durch verbesserte Fenstersysteme zur Gestaltung der Benutzeroberfläche), extrem aufwendig geworden. Daneben muß auch immer auf die sich verändernden Rahmenbedingungen, wie neue Fenstersysteme oder neue fachliche Anforderungen, reagiert werden. Aus diesem Grund sind Baisisfunktionalitäten, welche in vielen Projekten benötigt werden, als allgemein verwendbare, generische Komponenten in einem Rahmenwerk zusammengefaßt. Unter „allgemein verwendbar“ wird in diesem Zusammenhang die Abstraktion von konkreten Anwendungsdomänen verstanden. Das Rahmenwerk ist somit kein Application-Framework[33].
Für das Rahmenwerk wurden die im WAM-Ansatz enthaltenen Konzepte mit der objektorientierten Programmiersprache Java implementiert. Es wurde darauf geachtet, eine mög-

[32] Eine umfassendere Betrachtung des Rahmenwerkes findet man unter [JWAM 2001]. Darüber hinaus finden sich bei [Bleek 1999a], [Bleek 1999b], [Bleek 1999c], [Gryczan 1999], [Gryczan 2000a], [Gryczan 2000b] weitere Informationen über das JWAM-Rahmenwerk.

[33] vgl. [Gryczan 1999]

lichst in allen Kontexten der Softwareentwicklung, welche auf die Entwicklung eines Anwendungssystemes Einfluß nehmen können, skalierende Architektur zu schaffen. Bei den Kontexten handelt es sich, wie in Abschnitt 2.1.4. geschildert, um fachliche Konzepte, verwendete Technologie[34] sowie Handhabung und Präsentation.

2.2.2 Die Architektur

Das Rahmenwerk ist als Schichtenarchitektur[36] entworfen worden. Die einzelnen Schichten beinhalten softwaretechnische Komponenten, deren Funktionalität sie ihren übergeordneten Schichten als Dienstleistung zur Verfügung stellen. Hierbei haben die untergeordneten Schichten keinerlei Kenntnis von der darüberliegenden Schicht. Diese Schichten werden als *Handhabungs- und Präsentations-*, *Technologie-* und *Spracherweiterungsschicht* bezeichnet. Abbildung 2.7 zeigt die vereinfachte Übersicht der Struktur des Rahmenwerkes.

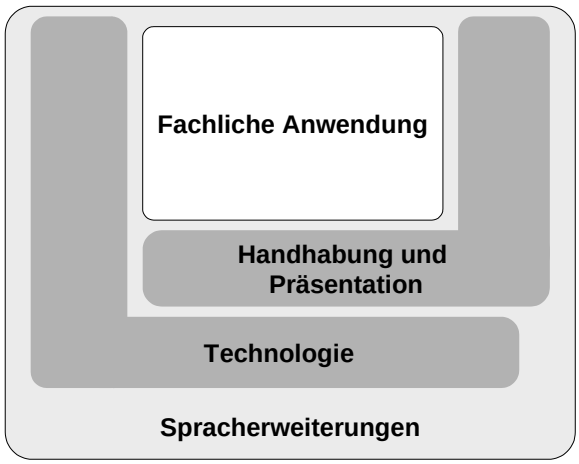

Abbildung 2.7: Die Schichten des JWAM-Rahmenwerks in der Übersicht [35]

In der Spracherweiterungsschicht sind für den WAM-Ansatz benötigte Konzepte der objektorientierten Programmierung wie Fachwerte und Vertragsmodell enthalten, welche nicht von der Programmiersprache selbst oder in einer Klassenbibliothek vorgegeben werden.
Die Technologieschicht kapselt Technologien wie Persistenzmechanismen oder die konkrete GUI-Bibliothek, um die Austauschbarkeit und Anpaßbarkeit der verwendeten Produkte zu gewährleisten.
Die Handhabungs- und Präsentationsschicht beinhaltet Konzepte wie die Material- und Werkzeugkonstruktion. Diese Schicht kapselt die Basisimplementationen dieser Konzepte, welche sich dann an einen konkreten Anwendungsfall anpassen lassen.
In diesen Rahmen kann, wie in der Abbildung dargestellt, eine konkrete fachliche Anwendung eingepaßt werden.

2.3 Zusammenfassung

In diesem Kapitel wurden die für diese Arbeit wichtigen Begriffe und Konzepte für das Verständnis des WAM-Ansatzes vorgestellt. Es wurden unter anderem die verschiedenen Leitbilder für die einzelnen Arbeitsplatztypen erläutert. Des weiteren wurden die zentralen Entwurfsmetaphern des WAM-Ansatzes wie Werkzeug, Automat und Material beschrieben und die Zusammenhänge zwischen den Leitbildern dargelegt. Darüber hinaus wurde das Konzept der Werkzeugkonstruktion und das der Fachlichen Services geschildert.
Anschließend wurden die Kernidee und die Struktur des JWAM-Rahmenwerks dargelegt, um ein grundlegendes Verständnis des technologischen Umfeldes zu schaffen.

---

[34] **Definition: Technologie**
Technologie bezeichnet das konzeptionelle Wissen der Entwickler um die verwendete Technik. Wird das Modell des Anwendungssystems erstellt, muß sich dieses in einem expliziten oder impliziten Modell der verwendeten Technik niederschlagen.
[Züllighoven 1997], Seite 157

[35] vgl. [JWAM 2001] und [Bleek 1999]

[36] vgl. [Züllighoven 1998] und [Bäumer 1997].

Im folgenden Kapitel wird das Problemfeld dieser Arbeit erläutert und unter Beachtung der soeben dargelegten Begriffe und Konzepte des WAM-Ansatzes untersucht.

Kapitel 3: Das Problemfeld – WAM-Anwendungen im World Wide Web

In den letzten Jahren nahm die Nutzung des World Wide Web[37] immer stärker zu. Damit einhergehend wurden und werden auch immer mehr Dienstleistungen, zum Beispiel im Bereich des E-Commerce, über das Web für einen großen Anwenderkreis verfügbar gemacht. Auch in verschiedenen Projekten, welche mit dem JWAM-Rahmenwerk realisiert wurden, wurde der Bedarf nach Anwendungen für das World Wide Web deutlich. Gleichzeitig wurde die Frage gestellt, wie solche Anwendungen aufgebaut sind und wie sich die Entwicklung solcher Anwendungen durch ein Rahmenwerk unterstützen läßt.
In diesem Kapitel wird unter Einbeziehung einiger Anwendungsbeispiele geklärt, welche Anforderungen an Web-Applikationen vor dem Hintergrund des Werkzeug-Automat-Material-Ansatzes gestellt werden. Für dieses Vorgehen wird zunächst beschrieben, was Web-Anwendungen ausmacht und welche Probleme sich bei ihrer Konstruktion ergeben. Des weiteren werden Web-Applikationen in den WAM-Rahmen eingeordnet, um hieraus Erkenntnisse darüber zu gewinnen, um welchen Typ von Arbeitsplatz es sich in diesem Fall handelt und welche Charakteristika ihn kennzeichnen. Ziel dieses Vorgehens ist es, die Gründe für die Verwendung von Zustandsautomaten zur Realisierung von Vorgangssteuerungen aufzuzeigen.

3.1 Anwendungen im World Wide Web

Das World Wide Web ist Anfang der neunziger Jahre als einfacher Informationszugang für eine große Anzahl von Anwendern entworfen worden (siehe [Berners-Lee 1989]). In der Anfangszeit sollten hauptsächlich textuelle Inhalte und Grafiken für die Nutzer zugänglich gemacht werden. Zusätzlich sollte es die Möglichkeit geben, Dokumente durch Verweise, sogenannte *Hyperlinks*, miteinander zu vernetzen, so daß eine Navigation durch Dokumentstrukturen möglich wurde.
Hierfür wurden auch die technischen Hilfsmittel wie die erste Version der *Hypertext Markup Language (HTML)* entworfen. HTML ist eine sogenannte Auszeichnungssprache, welche unter Zuhilfenahme der *Standard Generalized Markup Language (SGML)* definiert wird. Das Prinzip von Auszeichnungssprachen ist es, nur die logischen Bestandteile eines Dokumentes zu beschreiben, nicht aber deren konkretes Aussehen. Nähere Informationen zur aktuellen Version 4.01, auf die im weiteren stets Bezug genommen wird, finden sich unter [HTML 1999]. HTML-Dokumente werden durch einen Browser auf dem Arbeitsplatz des jeweiligen Anwenders repräsentiert. Das Dokument wird hierbei in der Anzeigefläche des Browsers dargestellt. Darüber hinaus hat ein einfaches HTML-Dokument keinerlei Zugriff auf Fensterelemente oder andere Ressourcen des Betriebssystems[38]. Die Anzeigefläche eines Web-Browsers wird bei [Otto&Schuler 2000] als *Webtop* bezeichnet. [Otto&Schuler 2000] führen für die Klasse von Anwendungen, welche auf der Anzeigefläche eines Browsers dargestellt wird, den Begriff der *Webtop-Anwendung* ein.

[37] Im Folgenden auch mit WWW (= World Wide Web) abgekürzt.
[38] Durch auf Java-Applets, JavaScript usw. basierenden HTML-Seiten kann die Trennung von Browser und Betriebssystem allerdings teilweise aufgehoben werden.

> **DEFINITION: WEBTOP-ANWENDUNG**
>
> *Eine Webtop-Anwendung läuft auf einem entfernten Server. Sie überträgt ihre Oberfläche als HTML-Seite auf Anforderung im HTTP-Protokoll[39] an den Client, einen Web-Browser. Der Web-Browser visualisiert die Oberfläche der Webtop-Applikation. Eine Webtop-Applikation kann nicht von sich aus mit dem Client kommunizieren.*
>
> [Otto&Schuler 2000], Seite 22

Die einzigen Anwendungen, welche in den Anfängen des World Wide Web zu Beginn der neunziger Jahre angeboten wurden, waren Suchmaschinen, um nach spezifischen Informationen oder Seiten im gesamten World Wide Web suchen zu können.
Später wurden die technischen Möglichkeiten soweit verbessert, daß auch zum Beispiel komplexere Oberflächenelemente, wie Tabellen oder Frames, einfach gestaltet werden konnten. Daneben wurde auch immer mehr Funktionalität durch Skriptsprachen wie *JavaScript* (siehe [Flanagan 1998]) oder *Cascading Style Sheets* (CSS) (siehe [CSS 1998]) ermöglicht. Daneben gibt es noch eine Vielzahl anderer Skriptsprachen in verschiedensten Versionen.
Gleichzeitig nahm auch die Bandbreite an Angeboten im World Wide Web zu. Es wurden nicht mehr nur einfache HTML-Dokumente zur Verbreitung von Informationen oder Suchmaschinen angeboten, sondern auch eine Vielzahl von Dienstleistungen wie Online-Shopping oder Chat-Rooms.
Bevor nachfolgend anhand von Beispielen Webtop-Anwendungen näher untersucht werden, wird in den folgenden Abschnitten geschildert, welche Charakteristika solche Anwendungen aufweisen.

3.1.1 Technik und Ablauf von Webtop-Anwendungen

Webtop-Anwendungen werden in ihrem Aufbau stark von der verwendeten Technik des HTTP-Protokolls und dem damit zusammenhängenden Ablauf bei der Verwendung geprägt. Der Ablauf einer typischen Webtop-Anwendung läßt sich folgendermaßen beschreiben:
Durch Eingabe einer *URL*[40] wird auf einem Server die Anwendung gestartet. Hierfür wird dem Server die Anfrage übertragen. Der Server reagiert nun, indem er entweder selbst oder durch Benutzung der Webtop-Anwendung übertragene Daten verarbeitet und als Antwort entweder ein HTML-Dokument generiert oder auf ein schon bestehendes Dokument verweist. Dieses Dokument wird nun dem Anwender vom Browser präsentiert, woraufhin er weitere Eingaben machen und diese wiederum abschicken kann oder einfach eine andere URL eingibt und somit eine andere Seite wählt.
Folglich gibt es neben der Möglichkeit, mit dem Benutzer über textuelle oder graphische Hyperlinks oder den durch HTML unterstützten Widgets (ein- und mehrzeilige Eingabefelder, Markierungskästchen, Auswahlknöpfe, Auswahllisten und Schaltflächen) in den generierten HTML-Seiten zu interagieren, keine andere Form der Interaktion.
Durch die Technik wird also ein bestimmtes Request-Response-Interaktionschema (siehe Abbildung 3.1) vorgegeben. Das heißt, zu einem bestimmten Zeitpunkt sind immer nur bestimmte Aktionen möglich. Die Schritte, welche hierbei gemacht werden, sind dabei eher grob. Ein solcher Schritt ist beispielsweise das Ausfüllen eines Formulars. Nach dem Ausfüllen wird das Formular über eine explizite Aktion des Anwenders abgeschickt. Erst

[39] HTTP = Hypertext Transfer Protocol (siehe [Fielding 1997])
[40] URL = Uniform Resource Locator (siehe [Berners-Lee 1994])

dann kann die Anwendung reagieren. Dies bedeutet, die Anwendung kann immer nur dann aktiv werden, wenn der Anwender einen neuen Request angestoßen hat. Vorher gibt es für die Anwendung keine Möglichkeit, Informationen über Änderungen im System an den Anwender weiterzugeben. Webtop-Anwendungen sind also rein reaktive Systeme.

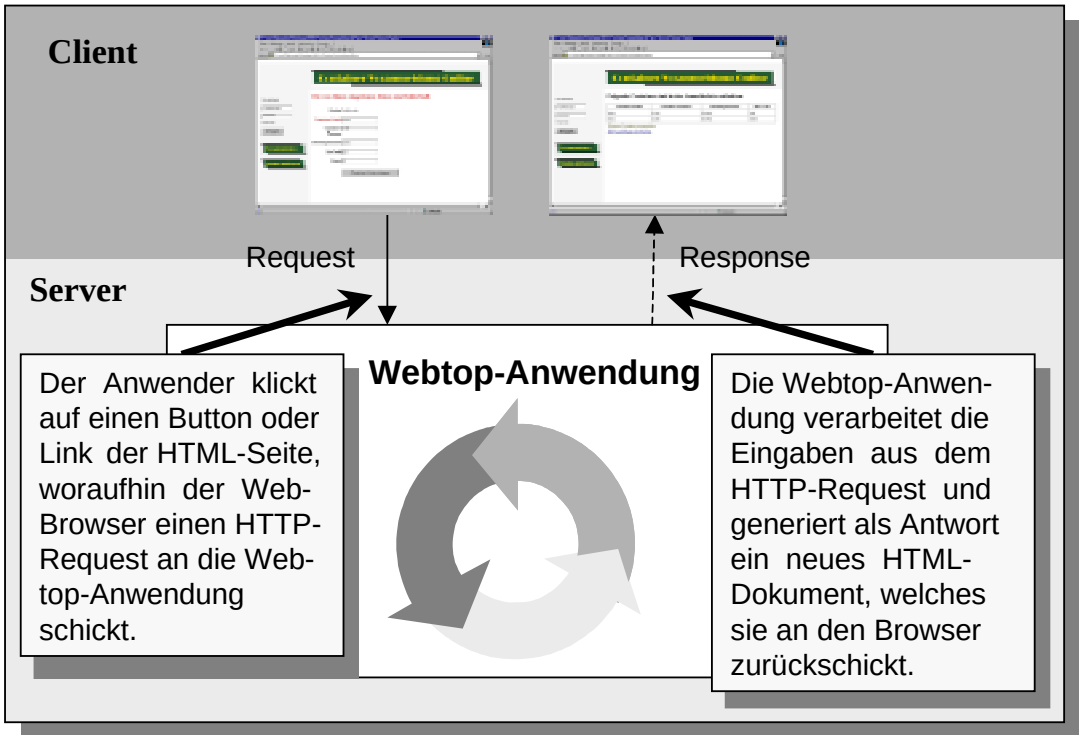

Abbildung 3.1: Request-Response-Interaktionsschema für Webtop-Anwendungen

Da das verwendete HTTP-Protokoll gedächtnis- und zustandslos[41] ist, müssen Informationen zur Identifikation der Anwender oder bestimmte andere Werte der Webtop-Anwendung bei jedem Request neu vorgelegt werden. Oftmals kodiert man dazu die Daten über versteckte Felder, sogenannte *hidden fields,* welche nicht im Browser angezeigt werden, in das HTML-Dokument hinein. Allerdings lassen sich seit der Einführung von *Servlets* [Servlets 2000], *Java Server Pages*[42] [JSP 2000], *Cookies* [Cookies 2000], *URL-Rewriting* [Roßbach 2000] und *Active Server Pages* [Homer 1999] Sitzungen (*Sessions*) nachbilden. Die Verwendung von Cookies kann aber zu Problemen führen, da diese abschaltbar sind. Das URL-Rewriting ist in jedem Fall anwendbar, führt aber zu einer zusätzlichen Belastung des Servers.
Die technischen Möglichkeiten in diesem Bereich werden jedoch ständig verbessert, so daß damit zu rechnen ist, daß in Zukunft diese Probleme nicht mehr auftreten werden.

3.1.2 Aufbau von Webtop-Anwendungen

Nun stellt sich allerdings die Frage, wie eine Webtop-Anwendung[43] aufgebaut ist. Eine Webtop-Anwendung kann aus einem bzw. mehreren Programmteilen, wie zum Beispiel

[41] Durch die Verwendung von JavaScript oder versteckten automatischen Requests des Clients (*Client-Pull*), läßt sich für den Anwender der Anschein erwecken, als würde sich dieser rein reaktive Charakter von Webtop-Anwendungen aufheben (siehe [Otto&Schuler 2000]).

[42] JSP = Java Server Pages

[43] Anwendungen, die zwar im Browser ablaufen, aber unabhängig von einem Server sind, werden nicht zu den Webtop-Anwendungen gezählt (vgl. [Otto&Schuler 2000]). In diese Kategorie fallen *Java-Applets*

Servlets, Java Server Pages oder *CGI-Skripten* [Münz 1999] bestehen. Daneben gibt es noch viele andere Skriptsprachen wie PHP, Perl oder Active Server Pages, auf welche aber nicht gesondert eingegangen wird.
Nach [Otto&Schuler 2000] gibt es zwei wesentliche Verfahren zur Umsetzung von Webtop-Anwendungen (siehe Abbildung 3.2):

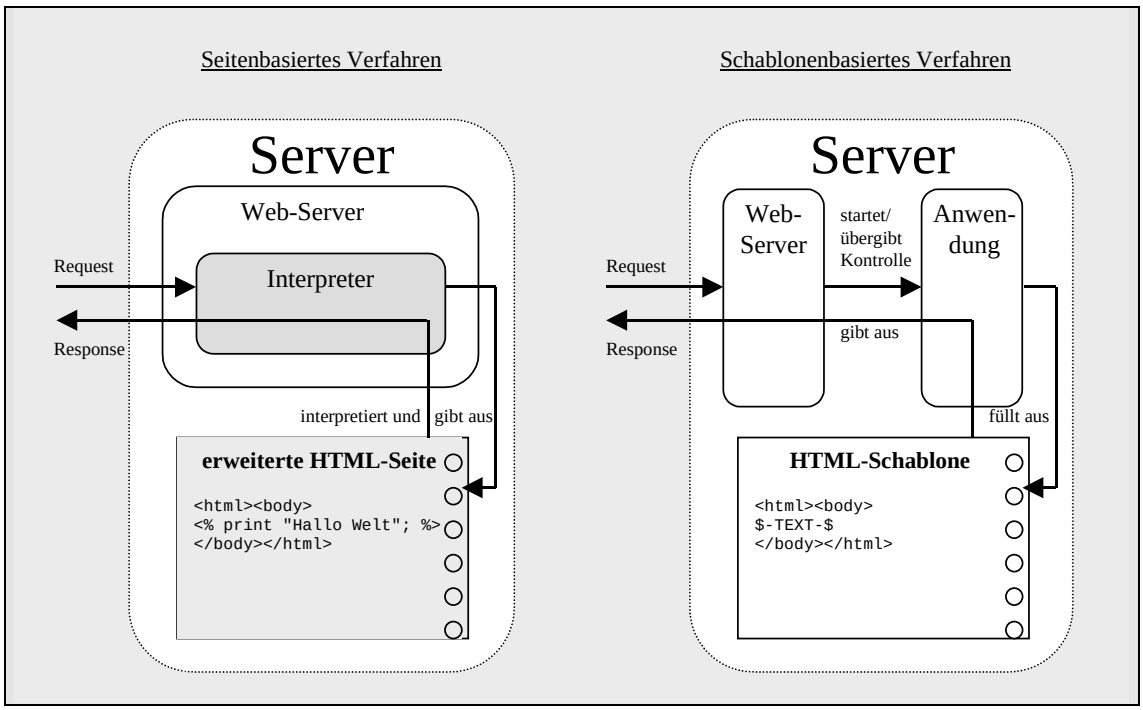

Abbildung 3.2: Verfahren zur Umsetzung von Webtop-Anwendungen[44]

Das erste ist das *seitenbasierte Verfahren,* bei dem der Entwickler die Anwendungslogik in den HTML-Code hineinschreibt. Der Programmcode für die Anwendungen ist durch besondere Textmarken gekennzeichnet, so daß ein Web-Server den Programmtext erkennen kann und die HTML-Seite entsprechend aufbaut. Diese Seite, welche nur den HTML-Code enthält, wird dann zum Anwender übertragen.
Das zweite ist das *schablonenbasierte Verfahren*, bei dem der Web-Server bei einem Request ein Programm startet, welches die Anwendungslogik ausführt und die Ergebnisse in eine HTML-Schablone an entsprechend gekennzeichneten Stellen einfügt. Das Ergebnis ist wiederum eine reine HTML-Seite. Zu den schablonenbasierten Verfahren zählen Servlets und CGI-Skripte.
Beide Verfahren gibt es allerdings kaum in ihrer reinen Form, sondern sie werden oft miteinander vermischt. Häufig verwendete Kombinationen sind *JSP/Servlets* oder *Embedded Perl/Perl CGI.*

3.1.3 Probleme bei der Konstruktion von Webtop-Anwendungen

Im Folgenden werden einige generelle Probleme der Konstruktion von Webtop-Anwendungen beschrieben.
Ein grundsätzliches Problem ist die Verteilung des Wissens über fachliche Zusammenhänge bei Webtop-Anwendungen. Hierbei kann es sich zum Beispiel um das Wissen zur Bear-

(siehe [Münz 1999]), *ActiveX*-Anwendungen (siehe [Chappel 1996]) und *JavaScript*-Code (siehe [Flanagan 1998]).

[44] aus [Otto&Schuler 2000], Seite 33

beitung von Materialien handeln. Bei dem seitenbasierten Verfahren ist immer ein fachlich zusammengehöriger Teil auf einer Seite zusammengefaßt. Von Bedeutung ist hierbei, in welcher Reihenfolge die Seiten abgearbeitet werden, damit immer die notwendigen Informationen für die weitere Arbeit des Anwenders vorhanden sind. Darum muß der Ablauf solcher Anwendungen oftmals stark vorgegeben werden. Wird eine Funktionalität mehrfach benötigt, muß sie mehrfach implementiert werden. Schablonenbasierte Verfahren können diesen Effekt abmildern, indem ein Programm für mehrere Requests zuständig ist, aber auch hier wird der Charakter einer am Ablauf orientierten Anwendung stets deutlich.
Bei dem seitenbasierten Verfahren tritt zusätzlich das Problem auf, daß Funktion und Interaktion bzw. Präsentation und Handhabung miteinander vermischt werden. Der Nachteil hierbei besteht darin, daß diese Teile derart von einander abhängig sind, so daß sie nicht separat weiterentwickelt bzw. gewartet werden können. Mit einer solchen Konstruktion wird ein gängiges Prinzip der Softwareentwicklung verletzt, wonach Funktion und Interaktion zu trennen sind, um die Wartbarkeit und Weiterentwicklungsfähigkeit von Software zu verbessern[45]. Hinzu kommt, daß bei dem seitenbasierten Verfahren die Vermischung von HTML-Code mit den Fragmenten einer Programmiersprache zu schwer lesbaren und unverständlichen Programmtexten führen kann. Besonders deutlich wird dieser Umstand, wenn die Fragmente sehr umfangreich und stark mit dem HTML-Code verschachtelt sind.
Bei den schablonenbasierten Verfahren kann dieser Effekt durch einen entsprechenden Aufbau des Programms vermieden werden, indem die funktionellen Anteile getrennt realisiert werden. Allerdings ist bei den schablonenbasierten Verfahren der Aufwand, welcher für eine geeignete graphische Präsentation einer Seite zu treiben ist, relativ hoch. Gerade diese graphische Präsentation ist aber ein wichtiger Faktor im World Wide Web. Dies gilt insbesondere für Anwendungen, welche im Bereich des Online-Shopping angesiedelt sind, in welchem die Kunden meist über eine ansprechende Gestaltung der Web-Seiten gewonnen werden. Die Gestaltung der graphischen Präsentation obliegt oftmals Web-Designern. Diese sind zwar Experten auf dem Gebiet der graphischen Gestaltung, kennen sich aber in der Regel nicht auf dem Gebiet der Programmierung aus. Somit ist das schablonenbasierte Verfahren besonders ungeeignet, da die Web-Designer hierfür über erhebliche Programmierfähigkeiten verfügen müßten. Schon bei den seitenbasierten Verfahren haben Web-Designer Schwierigkeiten, die dynamischen Anteile zu verstehen und sich darüber klar zu werden, wie die Seiten nachher im Browser aussehen.
Bei der Konstruktion von Webtop-Anwendungen tritt zudem das Problem auf, daß die Anwendungen aus den Requests nur Zeichenketten als Eingaben ziehen können bzw. auch nur Zeichenketten als Ausgabe in die HTML-Seiten schreiben können. Um an die Eingaben zu gelangen, müssen diese Daten aus dem Request herausgefiltert werden. Bei diesem Vorgang kann der Typ eines Eingabefeldes nur über den Bezeichner erkannt werden. Es gibt also keinerlei Typsicherheit. In den Anwendungen muß deshalb darauf geachtet werden, die Eingabefelder so zu benennen, daß eine geeignete Konvertierung geleistet werden kann. Das Wissen über die entsprechenden Konvertierungen ist mitunter sehr umfangreich, und sollte - wenn möglich - zentral zur Verfügung gestellt werden.
Ein weiteres, sehr schwerwiegendes Problem ist das mehrfache, unbeabsichtigte Auslösen ein und derselben Aktion. So darf in einer Online-Banking-Anwendung nicht der Fall eintreten, daß durch mehrfaches Klicken auf einen Link oder Button eine Überweisung auch genauso oft ausgeführt wird. Ebenso darf eine solche Anwendung durch die eben geschilderte Problematik nicht in einen inkonsistenten Zustand geraten, so daß der Anwender nicht mehr weiterarbeiten kann.

[45] vgl. Abschnitt 2.1.3 **Werkzeugkonstruktion**

Daneben gibt es in den Browsern die Möglichkeit, unter Ausnutzung des Caching-Mechanismus[46] der Web-Browser, einige Seiten „zurückzublättern“ und die „alten“ Seiten zur Anzeige zubringen. Bei einem solchen Vorgang darf es nicht vorkommen, daß von einer solchen „alten“ Seite eine Aktion ausgeführt werden kann, wodurch die Anwendung in einen anderen, mitunter inkonsistenten Systemzustand wechselt.
Neben den bisher genannten Problemen gibt es bei Webtop-Anwendungen, welche aus mehreren Programmteilen bestehen, noch das Problem des Weiterleitens auf eine geeignete Folgeseite. Bei größeren Anwendungen kann es durchaus passieren, daß eine der Folgeseiten nicht mehr besteht oder umbenannt wurde. Die Wartung von solchen, über mehrere Seiten verteilten Anwendungen gestaltet sich ebenfalls schwierig, da hier alle Programmteile auf Verweise zu anderen Teilen der Anwendungen durchsucht werden müssen. Dieses Problem ließe sich aber durch verbesserte technische Unterstützung, wie beispielsweise bessere Entwicklungstools, beheben. Allerdings bleibt weiterhin das Problem bestehen, daß unter dieser Verteilung von Wissen die Übersichtlichkeit und Verständlichkeit der Anwendung leidet.

3.2 Anwendungsbeispiele

Nachdem im vorangegangenen Abschnitt grundsätzlich das Umfeld beschrieben und einige wichtige Begriffe eingeführt wurden, werden im Folgenden zwei typische Beispiele beschrieben, um den fachlichen Hintergrund für diese Arbeit zu erläutern. Die beiden vorgestellten Projekte, welche sich aus unterschiedlichen Anwendungskontexten ergeben haben, stehen stellvertretend für eine Reihe von Anwendungsfällen.

3.2.1 Die Containervoranmeldung

Bei der Containervoranmeldung handelt es sich um ein System zum Erfassen von Containerdaten bei der Hamburger Hafen & Lager AG (HHLA).
Bevor ein Container umgeschlagen werden kann, müssen die Daten des jeweiligen Containers wie Containernummer, An- oder Auslieferdatum erfaßt werden, so daß im Vorfeld geklärt werden kann, welche Ressourcen für Lagerung bzw. Be- und Entladen notwendig sind. So muß zum Beispiel geklärt werden, wann ein Container für welchen Kunden per Schiff oder LKW angeliefert bzw. abgeholt wird. Diese Daten werden von den jeweiligen Reedern und Spediteuren zur Verfügung gestellt.
Die Voranmeldungen gingen bisher größtenteils per Fax ein und mußten bei der HHLA unter großem zeitlichen und personellen Aufwand in das EDV-System übernommen werden.
Um bei diesem Verfahren Kosten zu sparen, soll die Voranmeldung nun so gestaltet werden, daß der gesamte Vorgang über das Internet abgewickelt werden kann. Hierfür ist eine geeignete Webtop-Anwendung zu erstellen, welche diese Dienstleistung für die Kunden verfügbar macht. Zusätzlich soll diese Anwendung so gestaltet sein, daß Anwender mit den unterschiedlichsten Wissensständen mit diesem System sicher umgehen können, um fehlerhafte Eingaben zu vermeiden.

[46] Jeder heutzutage gebräuchliche Internet-Browser verfügt über einen Cache-Speicher, der es ihm erlaubt, angezeigte Web-Seiten und -dateien (beispielsweise Grafiken oder HTML) temporär in gesonderten Verzeichnissen zu speichern. Hierdurch ist es möglich, eine schon besuchte und wieder verlassene Seite erneut zur Anzeige zu bringen, ohne sie neu aus dem WWW zu laden. Dadurch erreicht man einen Geschwindigkeitsvorteil, da es in der Regel länger dauert, eine Seite aus dem WWW zu laden, als sie von der Festplatte zu laden. Dabei wird allerdings der Nachteil in Kauf genommen, daß sich die Seite im Web von der im Speicher unterscheiden kann.

Das Benutzungsmodell[47] dieser Webtop-Anwendung sollte dann wie folgt aussehen:
Zunächst startet der Anwender seinen Browser und gibt die entsprechende URL ein. Anschließend identifiziert er sich über die Eingabe seines Benutzernamens und des dazugehörigen Paßwortes. Hiernach hat er die Möglichkeit, über die Auswahl eines entsprechenden Buttons oder Links auf eine Seite mit einem Anmeldeformular zu gelangen. Hier kann er seine Angaben zu dem anzumeldenden Container in ein Voranmeldungsformular eintragen. Nach dem Ausfüllen kann die Voranmeldung über das Betätigen eines Buttons oder Links in eine Liste von Voranmeldungen übernommen werden. Vor dem Eintragen werden die Eingaben von der Anwendung auf ihre Korrektheit überprüft. Sind die Angaben nicht korrekt, wird eine zweite Seite angezeigt, auf der die fehlerhaften Angaben markiert sind und der Anwender die Möglichkeit hat, seine Angaben zu korrigieren. Sind die Eingaben korrekt, wird die Voranmeldung in die Liste von zu versendenden Anmeldungen eingefügt. Hiernach hat der Anwender die Auswahl zwischen dem Ausfüllen einer weiteren Anmeldung oder dem Versenden der Liste mit den bisher gemachten Voranmeldungen. Während des Ausfüllens der Voranmeldungen soll die Möglichkeit bestehen, die Liste der Voranmeldungen anzuschauen. Voraussetzung hierfür ist allerdings, daß schon Voranmeldungen in der Liste vorhanden sind. Sobald der Benutzer alle Voranmeldungen eingegeben und diese abgeschickt hat, kann er die Anwendung über ein explizites Abmelden verlassen. Falls der Benutzer die Anwendung während des Ausfüllens einer Voranmeldung oder des Betrachtens der Liste der Voranmeldungen verlassen möchte, hat er jederzeit die Möglichkeit dazu.

3.2.2 Das Traffixx-Projekt

Die Idee des Traffixx-Projektes ist es, einen elektronischen Marktplatz für den öffentlichen Personennahverkehr (ÖPNV) zu schaffen, bei dem alle ÖPNV-Unternehmen unter einer Adresse im World Wide Web erreichbar sind.
Die verschiedenen Dienstleistungen gliedern sich in unterschiedliche Bereiche, welche als Module bezeichnet werden. Diese Module teilen sich in die Bereiche Ticketverkauf, Versteigerung von Sonderfahrzeugen und anderen Investitionsgütern sowie Ausschreibungen auf.
Der hier zu betrachtende Bereich ist der Ticketmanager, mit dem es den Kunden dieser Unternehmen ermöglicht werden soll, sich die angebotenen Dienstleistungen einfach über das Internet verfügbar zu machen. Der Kunde kann die Dienstleistung wahrnehmen, ohne eine Niederlassung der Anbieter aufsuchen zu müssen. Im Vordergrund steht der Verkauf von Zeitkarten. Daneben sollen aber auch Sonderleistungen wie Charter- und Sonderfahrten, die Versteigerung von Fundsachen und der Verkauf von Fanartiklen und Souvenirs vereinfacht werden. Die Anbieter der Dienstleistung können Personal einsparen, welches sonst diese Dienstleistung erbracht hat.
Das Benutzungsmodell läßt sich wie folgt beschreiben:
Die Kunden sollen sich, um die Dienstleistungen wahrnehmen zu können, auf die Traffixx-Seite begeben und dort die gewünschten Angebote auswählen. Wenn sie einen Artikel erwerben möchten, können sie ihn in ihren virtuellen Warenkorb packen. Natürlich soll es auch möglich sein, Angebote wieder aus dem Warenkorb zu entfernen. Nachdem der Kunde seine Auswahl getroffen hat, kann er die gewählten Angebote bezahlen, indem er die notwendigen Informationen wie zum Beispiel seine Kreditkartennummer angibt. Diese werden dann von der Anwendung validiert. Waren die Angaben richtig, wird dem Kunden

[47] Auf eine detailliertere Schilderung von Szenarien oder Systemvisionen wurde verzichtet, da für den hier betrachteten Rahmen ein kurzer Überblick ausreichend ist.

zunächst eine Bestätigung zugesandt. Anschließend erhält er die erworbenen Waren zugesandt.

3.2.3 Einordnung der Beispiele in den WAM-Ansatz

Bevor versucht wird, die Anforderungen für ein solches System, wie es im vorangegangenen Abschnitt beschrieben wurde, zu ermitteln, sollte an dieser Stelle bestimmt werden, welches der verschiedenen Leitbilder des WAM-Ansatzes anwendbar ist.
Vergleicht man die Leitbilder mit den Beschreibungen der Beispiele im Kapitel 3.2 stellt man schnell fest, daß an dieser Stelle am besten das Leitbild des Selbstbedienungsautomaten paßt.
Das Leitbild des Arbeitsplatzes für eigenverantwortliche Expertentätigkeit[48] ist nicht geeignet, da hier einige wichtige Charakteristika, welche dieses Leitbild ausmachen, nicht anwendbar sind. Eine der wichtigsten Eigenschaften dieses Arbeitsplatztyps ist die Tatsache, daß es sich bei den Anwendern um Experten handelt, die ihren Arbeitsablauf möglichst frei gestalten können sollen. Dies ist bei den in Abschnitt 3.2 genannten Beispielen nicht der Fall.
Bei den hier vorliegenden Beispielen soll eine Vielzahl von Nutzern mit unterschiedlichem Fachwissen an einem Arbeitsplatz arbeiten. Man muß beispielsweise also davon ausgehen, daß die Kunden nicht mit allen Verfahrensweisen und Arbeitsschritten bei der HHLA vertraut sind. Sie müssen zwar wissen, daß eine Dienstleistung angeboten wird, welche es ermöglicht Container anzumelden, aber sie müssen keine Kenntnis über konkrete Materialien (zum Beispiel das Anmeldeformular) oder Werkzeuge (zum Beispiel ein Formulareditor zum Ausfüllen einer Anmeldung) haben. Auch müssen sie nicht wissen, wann welcher Bearbeitungsschritt erfolgen muß. Ähnlich verhält es sich bei den Anwendern des Traffixx-Projektes.
Nun kann natürlich argumentiert werden, daß jedes neue System Übung und Fachwissen erfordert. Aber gegen dieses Argument spricht die Tatsache, daß die Anwender mitunter nur einmal im Monat mit dem System arbeiten müssen und sich somit die notwendige Routine erst spät einstellt. Des weiteren hat das System die Aufgabe, die Kommunikation zwischen Kunde und Anbieter einer Dienstleistung zu regeln. Dies bedeutet, falls der Anbieter nicht in der Lage ist, ein für den Kunden einfach zu bedienendes System zu erstellen, wird dieser sich unter Umständen nach anderen Anbietern umsehen. Davon abgesehen sind die möglichen Kosten und der Aufwand, welche durch Fehlbedienungen entstehen können, nicht zu vernachlässigen. Hinzu kommt, daß ein solches System auch eingesetzt werden soll, um Fachpersonal zu entlasten. Die Kunden sollen eine Dienstleistung immer in Anspruch nehmen können, ohne daß sie Fachpersonal hinzuziehen müssen. Treten bei der Erbringung dieser Dienstleistung aber ständig Probleme auf, wird das Fachpersonal durch das System nicht entlastet sondern - im Gegenteil - stärker belastet.

> **FAZIT:**
>
> *Bei den hier betrachteten Webtop-Anwendungen handelt es sich um **Selbstbedienungsautomaten** im Sinne des WAM-Ansatzes.*

3.2.3.1 Anforderungen aus dem Leitbild

Wenn nun ein solcher Selbstbedienungsautomat umgesetzt werden soll, sind die nachfolgend aufgeführten Anforderungen[49] zu erfüllen.

---

[48] vgl. Abschnitt 2.1.1.1 **Arbeitsplatz für eigenverantwortliche Expertentätigkeit**
[49] vgl. Abschnitt 2.1.1.4 **Selbstbedienungsautomat**

Der erste Punkt, nämlich welche Aufgabe grundsätzlich zu erfüllen ist, ergibt sich schon aus der Bezeichnung „Selbstbedienungsautomat“:

> **ANFORDERUNG:**
>
> *Der Selbstbedienungsautomat soll einer Gruppe von Anwendern, welche keine Experten des jeweiligen Anwendungsgebietes sind, eine Dienstleistung verfügbar machen.*

Da die Anwender keine Experten bzw. nicht mit der konkreten Arbeitsumgebung vertraut sind, müssen die notwendigen Materialien einer Dienstleistung zugeordnet werden, so daß der Anwender, wenn er eine Dienstleistung ausgewählt hat, automatisch alle Materialien erhält.
Auch wenn die Anwender über die notwendigen Materialien verfügen könnten, kann nicht davon ausgegangen werden, daß sie wissen, wie diese zu bearbeiten sind und wann welche Bearbeitungsschritte möglich sind. Deshalb muß dieses Wissen im Automaten vorhanden sein. Daneben sollten, wenn dies fachlich möglich ist, komplexe Handlungen zusammengefaßt werden, um den Anwender zu entlasten.

> **ANFORDERUNG:**
>
> *Der Selbstbedienungsautomat muß das Wissen darüber kapseln, welche Materialien notwendig und wie diese zu bearbeiten sind, d.h. er muß Dienstleistungen vergegenständlichen.*

Daneben muß der Selbstbedienungsautomat den Anwender, im Sinne der unterstützenden Sichtweise, durch die Benutzung des Systems führen. So darf er dem Anwender nur die in dem jeweiligen Systemzustand sinnvoll anwendbare Funktionalität der Dienstleistung zur Verfügung stellen. Es bedeutet aber nicht, daß die Arbeitsschritte des Anwenders vom Automaten im Sinne einer Ablaufsteuerung vorgegeben werden. Die Entscheidung, welcher Schritt als nächster ausgeführt wird ist, bleibt beim Anwender, die Kontrolle darüber, welche Schritte möglich sind, liegt beim Automaten.

> **ANFORDERUNG:**
>
> *Der Selbstbedienungsautomat muß den Anwender durch den Vorgang der Benutzung des Systems führen.*

Um eine solche Vorgangssteuerung für den Selbstbedienungsautomaten zu realisieren, wird ein Zustandsmodell, über welches die Systemzustände explizit gemacht werden, benötigt. In den meisten softwaretechnischen Systemen werden Systemzustände über die Werte der Variablen des jeweiligen Systems implizit definiert. Für den vorliegenden Fall ist dies nicht ausreichend, weil der Systemzustand auch nach außen für den Anwender explizit gemacht werden muß, da er ansonsten keine Möglichkeit hat, zu verstehen, warum er eine bestimmte Funktionalität gerade nutzen bzw. nicht nutzen kann.

> **ANFORDERUNG:**
>
> *Der Selbstbedienungsautomat muß ein Zustandsmodell besitzen, in welchem Zustände und Zustandsübergänge explizit gemacht werden.*

Damit ein Anwender einen Selbstbedienungsautomaten sinnvoll nutzen kann, darf dieser nur Handlungen zulassen, die in dem aktuellen Zustand möglich sind. Versucht der Anwender eine Operation auszuführen, welche im aktuellen Zustand nicht möglich ist, muß deren Ausführung verweigert werden.

> **ANFORDERUNG:**
>
> *Der Selbstbedienungsautomat muß seinen Systemzustand sowie Zustandsänderungen einem Anwender gegenüber deutlich machen, indem er nur Operationen zuläßt, welche in dem jeweiligen Systemzustand möglich sind.*

Die folgende Abbildung 3.3 faßt die Anforderungen von Selbstbedienungsautomaten nochmals zusammen.

- Vergegenständlichung von Dienstleistungen
- Kapselung des Wissens über Materialien
- Muß ein explizites Zustandsmodell besitzen, um im Sinne einer Vorgangssteuerung Bearbeitungszustände zu verdeutlichen

Abbildung 3.3: Anforderungen von Selbstbedienungsautomaten

3.3 Offene Fragen zur Konstruktion von Web-Anwendungen

Im Folgenden werden die Probleme und Anforderungen, welche bei der Konstruktion von Selbstbedienungsautomaten für Webtop-Anwendungen auftreten, zusammengefaßt und als Fragestellungen formuliert. Grundlage hierfür sind die Abschnitte 3.1.3 und 3.2.4
Um die Forderungen nach einer Vorgangssteuerung zu erfüllen, müssen die Zustände des Benutzungsmodells für die jeweilige Anwendung explizit gemacht werden. Es dürfen nur solche Operationen ausgeführt werden, welche in dem aktuellen Zustand möglich sind.
Ein Zustandsmodell ist auch aus technischer Sicht sinnvoll, da immer sichergestellt werden muß, daß sich das System in einem fachlich konsistenten Zustand befindet. Es darf zum Beispiel nicht vorkommen, daß von „veralteten“ Seiten Aktionen ausgelöst werden können.

> **OFFENE FRAGE:**
>
> *Wie läßt sich ein explizites Zustandsmodell realisieren, welches die fachliche Funktionalität kapselt und somit die fachliche Konsistenz der Anwendung wahrt?*

Es stellt sich des weiteren die Frage, wie die zu entwerfende Komponente mit den bereits vorliegenden Komponenten, wie zum Beispiel Materialien, interagiert. Hierzu muß geklärt werden, wo und wie das fachliche Wissen zur Bearbeitung von Materialien modelliert wird. Das Wissen über die Bearbeitung eines Materials sollte möglichst nur an einer Stelle im Entwurf modelliert werden.

> **OFFENE FRAGE:**
>
> *Wo und wie wird das fachliche Wissen in der Anwendung modelliert?*

Es sollte beim Entwurf darauf geachtet werden, Funktion und Interaktion[50] zu trennen, um auch für die hier vorliegende Klasse von Anwendungen einen hohen Grad an Wiederverwendbarkeit, Wartbarkeit und Verständlichkeit zu erreichen.

> **OFFENE FRAGE:**
>
> *Wie können Interaktion und Funktion getrennt modelliert werden?*

Ähnlich der Werkzeugkonstruktion[50] muß ebenfalls die Trennung von Präsentation und Handhabung eingehalten werden. Wie jedes Werkzeug hat eine Webtop-Anwendung eine Form der Präsentation und eine Form der Handhabung und kann folglich auch entsprechend unterteilt werden. Die Trennung ist gerade in Hinblick auf die oben genannten Probleme bei der Entwicklung von Webtop-Anwendungen wichtig, bei denen oft Änderungen an der Oberfläche - sprich dem HTML-Code - vorgenommen werden, sich die Handhabung aber nicht ändert.

> **OFFENE FRAGE:**
>
> *Wie können Präsentation und Handhabung getrennt modelliert werden?*

Daneben müssen vom Benutzer im Browser ausgelöste Ereignisse von der Anwendung als solche erkannt und identifiziert werden. Dieses *Event-Handling* ist notwendig, damit die richtige fachliche Aktion ausgeführt werden kann.

> **OFFENE FRAGE:**
>
> *Wo wird das Wissen über das Event-Handling angesiedelt?*

Um diese Aktionen ausführen zu können, ist es natürlich notwendig, vom Anwender eingegebene Daten aus dem Request zu extrahieren und für die Ausführung der Aktionen aufzubereiten. Aufbereitung heißt in diesem Fall, die Werte, bei welchen es sich um einfache Zeichenketten handelt, ihren korrekten fachlichen Werten bzw. Materialien zuzuordnen.

> **OFFENE FRAGE:**
>
> *Wo wird das Wissen über die Konvertierung von Daten angesiedelt?*

Nachdem die Anwendung auf das vom Benutzer ausgelöste Ereignis reagiert hat, muß sie als Antwort dem Anwender eine neue Web-Seite zur Verfügung stellen, auf welcher die Ergebnisse seiner Aktion dargestellt werden. Hierzu muß aufgrund des Systemzustandes die Folgeseite bestimmt werden. Dieses Wissen, hier auch als *Dispatching* bezeichnet, sollte möglichst nur an einer Stelle lokalisiert sein, damit es zu keinen Inkonsistenzen kommt und die Wartung bzw. der Austausch der Seiten möglichst einfach ausführbar ist.

> **OFFENE FRAGE:**
>
> *An welcher Stelle lokalisiert sich das Wissen über die Abfolge der Seiten?*

Die Abbildung 3.4 zeigt eine Übersicht der oben geschilderten Konstruktionsprobleme[51]:

[50] vgl. Abschnitt 2.1.3 **Werkzeugkonstruktion**

[51] Konstruktionsprobleme werden in der Abbildung als grau unterlegte Kästchen dargestellt.

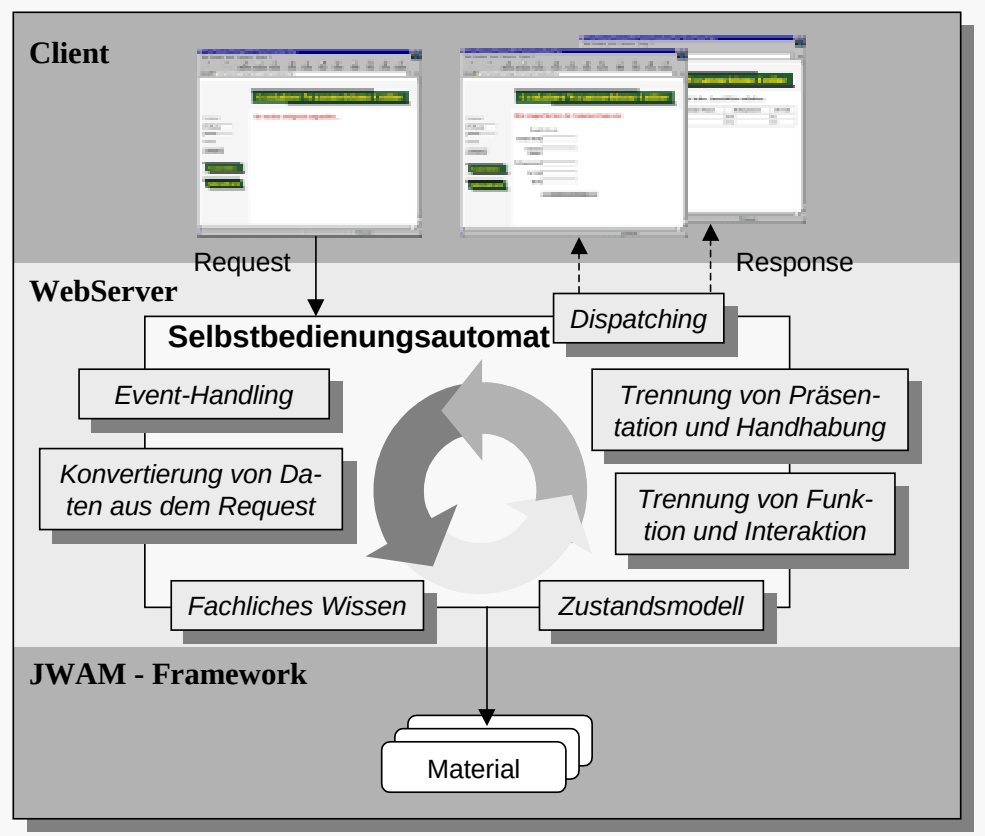

Abbildung 3.4: Konstruktionsprobleme für den Entwurf von Selbstbedienungsautomaten

3.4 Entwurf eines Selbstbedienungsautomaten für Webtop-Anwendungen

Dieser Abschnitt beschreibt, welche der in Abschnitt 3.3 genannten Konstruktionsprobleme für Webtop-Anwendungen sich durch Anwendung von Komponenten des JWAM-Rahmenwerks bzw. von Konstruktionskonzepten des WAM-Ansatzes lösen lassen.
Es wird gezeigt, wie sich JWAM-Komponenten in eine solche Anwendung einfügen und welche Probleme weiter bestehen bleiben bzw. welche neuen Probleme durch die Verwendung aufgeworfen werden.
Um die jeweiligen Entwurfsentscheidungen zu erläutern, wird gegebenenfalls das aus Abschnitt 3.2.1 bekannte Beispiel der Containervoranmeldung verwendet.

3.4.1 Trennung von Funktion und Interaktion

Wie schon bei der Beschreibung der Konstruktionsprobleme erläutert, ist es sinnvoll, Funktion und Interaktion voneinander zu trennen. Bei der Umsetzung der Komponenten im funktionellen Teil des hier vorliegenden Selbstbedienungsautomaten sollte darauf geachtet werden, diese möglichst ohne Wissen über ihre Präsentation zu modellieren. In diesem Fall heißt dies, daß das Wissen über das HTTP-Protokoll, Sessions oder die Repräsentation im Browser mittels HTML im interaktiven Teil angesiedelt werden muß.
Im Folgenden werden die einzelnen Entwurfsprobleme dem Funktions- bzw. dem Interaktionsteil zugeordnet.
Zu den funktionellen Anteilen gehört zum einen das fachliche Wissen des jeweiligen Anwendungsbereiches. Das fachliche Wissen umfaßt das Wissen darüber, welche Materialien zu bearbeiten und welche Bearbeitungsschritte möglich sind. Zum anderen gehört hierzu

das Zustandsmodell, welches das Wissen über den aktuellen Bearbeitungszustand der Materialien und des sie bearbeitenden Selbstbedienungsautomaten kapselt.
Zum interaktiven Teil zählen, wie bei der Werkzeugkonstruktion auch, die Bereiche Präsentation und Handhabung. Diesem Teil ist folglich das Problem der Trennung von Handhabung und Präsentation zuzuordnen. Darüber hinaus muß in diesem Teil das Event-Handling geleistet werden, da hierbei die vom Anwender ausgelösten Ereignisse für den Funktionsteil aufbereitet und an diesen weitergeleitet werden. Neben den Ereignissen umfaßt diese Aufbereitung auch die Konvertierung und Aufbereitung von Standarddatentypen aus dem Request in fachlich motivierte Werte der Anwendungsdomäne.
Abbildung 3.5 zeigt die entsprechende Zuordnung der verbleibenden Konstruktionsprobleme:

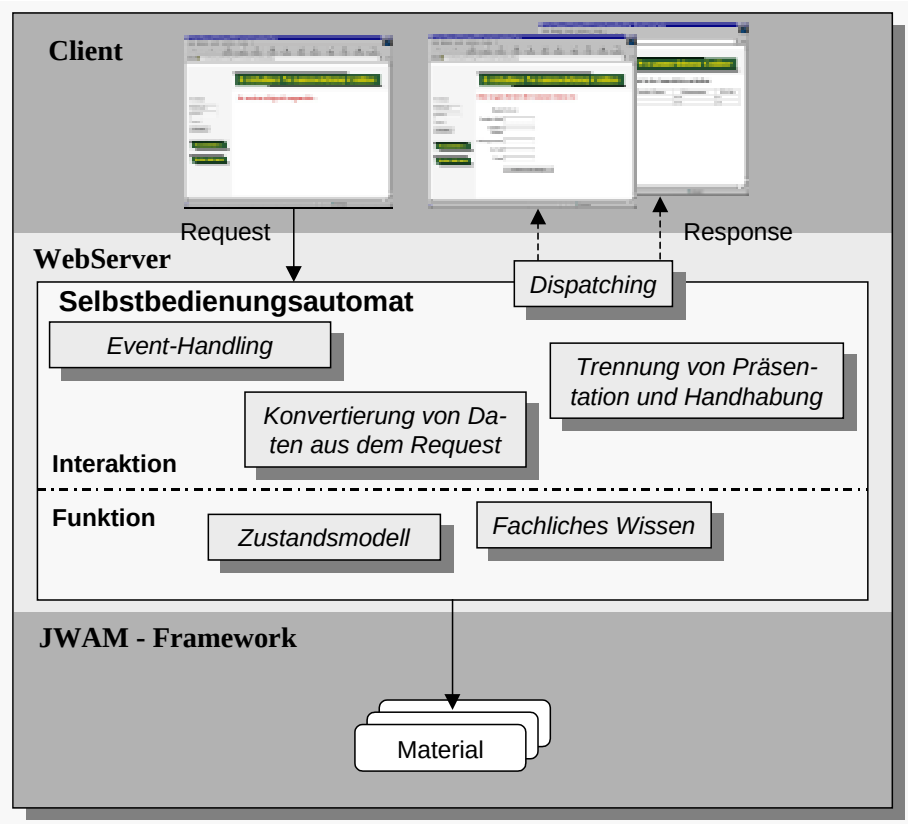

Abbildung 3.5: Trennung von Funktion und Interaktion

3.4.2 Kapselung fachlichen Wissens

Im obigen Abschnitt, welcher die Trennung in Funktion und Interaktion erläuterte, wurde das fachliche Wissen über den Anwendungsbereich dem Funktionsteil zugeordnet. Es stellt sich die Frage, an welcher Stelle und in welcher Form dieses Wissen modelliert wird.
Betrachtet man die in Kapitel 2 aufgeführten Konzepte des WAM-Ansatzes, so stellt man fest, daß die Fachlichen Services[52] eine Lösungsmöglichkeit für dieses Problem bieten. Wie in Abbildung 2.6 dargestellt, können Fachliche Services die fachliche Funktionalität einer Anwendungsdomäne an einer Stelle zentral zur Verfügung stellen, ohne daß die Funktionalität hierfür mehrfach implementiert werden muß.

[52] vgl. Abschnitt 2.1.4 **Fachliche Services**

Für den Selbstbedienungsautomaten bedeutet dies, daß das Wissen über den Umgang mit den Materialien aus der Anwendung herausgezogen und in einem Fachlichen Service zur Verfügung gestellt wird (siehe Abbildung 3.6).

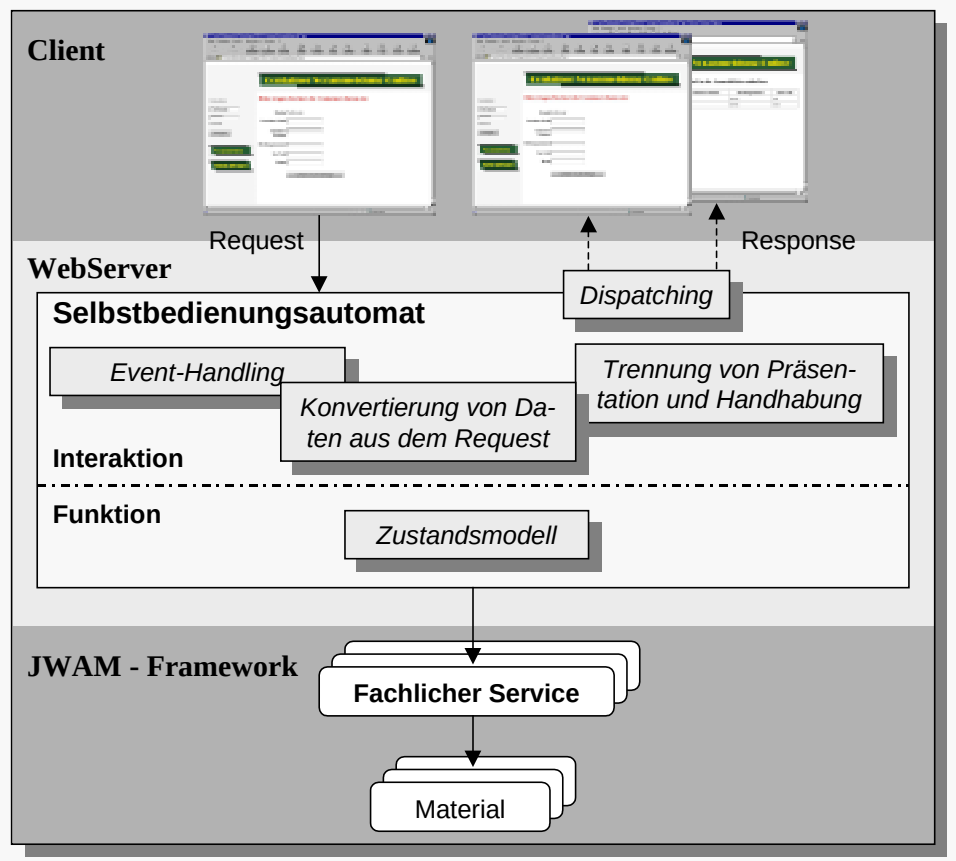

Abbildung 3.6: Verwendung von Fachlichen Services im Entwurf

Beim Entwurf der Fachlichen Services stellt sich aber die Frage, wie diese dimensioniert werden sollen. Es ist sicherlich keine zufriedenstellende Lösung, die gesamte fachliche Funktionalität eines Anwendungsfalles, wie der Containervoranmeldung, in einem Fachlichen Service zur Verfügung zu stellen. Der entstandene Service wäre dann mit großer Wahrscheinlichkeit nur für diesen einen Anwendungsfall nutzbar. Dadurch wäre die Forderung aus Abschnitt 2.1.4 nach Wiederverwendbarkeit Fachlicher Services nicht mehr erfüllt. Es ist also in solchen Fällen besser, kleinere Services zu entwerfen, welche nur einen Teil der Funktionalität für einen Anwendungsfalles beinhalten. [Otto&Schuler 2000] formulieren dieses Problem als Frage im Ausblick ihrer Arbeit:

> *Wo und anhand welcher Kriterien schneidet man die Grenzen zwischen verschiedenen Fachlichen Services?*
> [Otto&Schuler 2000], Seite 157

Bei [Züllighoven 1998] (Seite 42ff) werden, basierend auf [Meyer 1997], Kriterien und Regeln festgelegt, die eine Entwurfs- und Konstruktionseinheit erfüllen soll. Hierzu gehört unter anderem, daß für eine Konstruktionseinheit das *„Prinzip der maximalen Kohäsion“* gelten muß. Der Begriff „Kohäsion“ meint hierbei den *„Inneren Zusammenhalt“* zwischen den Merkmalen einer Konstruktionseinheit. Nimmt man dies als Grundlage, läßt sich vielfach eindeutig bestimmen, welche Dimension ein Fachlicher Service haben sollte. Für das Beispiel der Containervoranmeldung läßt sich dies mit Hilfe des eben genannten Kriteri-

ums recht einfach beantworten. Es gibt hier den Bereich der Benutzerverwaltung, der die Identifikation der Anwender regelt, und den Bereich des Umgangs mit Containervoranmeldungen, der die Bearbeitung und den Versand von Voranmeldungen regelt. Unter Umständen könnte man den zweiten Bereich noch aufteilen, allerdings handelt es sich beim Versenden um keine sehr komplexe Funktionalität, so daß eine Unterteilung nicht notwendig ist. Für die Modellierung bedeutet dies, daß ein Service zur Benutzerverwaltung (Login-Services) und zum anderen ein Service zur Bearbeitung und Verwaltung von Voranmeldungen (Voranmelde-Service) umgesetzt werden muß.
Wie bei der Einführung der Fachlichen Services schon geschildert, weisen diese verschiedene Ausprägungen auf. Einige dienen der Bereitstellung von Materialien und dem Umgang mit diesen, während andere zur Vergegenständlichung von Vorgängen verwendet werden. Für beide Ausprägungen gibt es den Fall, daß sie zustandsbehaftet sind. Die folgenden Zitate aus [Otto&Schuler 2000] belegen dies:

> *Dient ein Material und Umgang bereitstellender Fachlicher Service auch als Aufbewahrungsort für die Gegenstände der Anwendung [...] so ist er zustandsbehaftet. Nach außen ändert sich sein Zustand, wenn man von ihm verwaltete Materialien durch ihn verändern lässt.*
> [Otto&Schuler 2000], Seite 71

> *Vorgänge können in Fachlichen Services oft effektiv repräsentiert werden, wenn dieser das Modell einer Benutzer-Sitzung umsetzt. Der Benutzer wendet sich dann zum Anstoßen eines Vorgangs an den Fachlichen Service und dieser „kennt" den Benutzer bis zum Ende des Vorgangs. In diesem Fall kann auch die Vergegenständlichung des Vorgangs durch ein Material nach außen wegfallen.*
> [Otto&Schuler 2000], Seite 72

Das Wissen über das Zustands- bzw. Sitzungsmodell kann sich nach [Otto&Schuler 2000] entweder in dem Fachlichen Service selbst oder in der diesen benutzenden Komponente befinden. Da schon ein Zustandsmodell in der die Fachlichen Services benutzenden Komponente vorliegt, können hier einfach die Zustandsmodelle der jeweiligen Fachlichen Services integriert werden.
Im Normalfall tritt bei der Verwendung mehrerer Fachlicher Services ein Problem auf, wenn diese, wie oben geschildert, zusammen eine Dienstleitung realisieren sollen. Haben die Services Zustandsmodelle, müssen diese synchronisiert werden, um einen gemeinsamen Benutzungsvorgang zu ermöglichen. Betrachtet man das Beispiel der Containervoranmeldung, so muß der Login-Service, welcher die Zugriffe der Anwender auf das System regelt, auf den Voranmelde-Service, welcher Voranmeldungen auf ihre Korrektheit prüft und das Versenden von Voranmeldungen übernimmt, abgestimmt sein, damit eine fachlich einwandfreie Verwendung gegeben ist. Hieraus ergibt sich allerdings auch die folgende Frage:

> *Wie spielen mehrere Fachliche Services sinnvoll zusammen, ohne die Anforderung nach Kapselung fachlichen Wissens zu verletzen?*
> [Otto&Schuler 2000], Seite 157

Da allerdings die Entscheidung getroffen wurde, die Zustandsmodelle aus den Fachlichen Services in dem Zustandsmodell des Selbstbedienungsautomaten zu realisieren, ist dieses Problem bereits gelöst.

3.4.3 Trennung von Präsentation und Handhabung

Eine Trennung von Präsentation und Handhabung ist in dem vorliegenden Fall einfach möglich, indem die Aufgaben Event-Handling, Konvertierung von Daten und Dispatching der Handhabung zugeordnet werden. Die Präsentation umfaßt hierbei nur die Darstellung der Anwendung im Browser. Darüber hinaus sollte dort keinerlei Kenntnis über die Handhabung der Anwendung vorhanden sein.
Die Präsentation erfolgt entsprechend dem schablonenbasierten Verfahren[53] durch Java Server Pages[54]. Diese eignen sich besonders für die Repräsentation, da mit ihnen relativ einfach komplexe HTML-Seiten dynamisch generiert werden können. Der Anteil des Java-Quellcodes, der in ihnen verwendet werden muß, ist minimal und bezieht sich nur noch auf die Präsentation. Da die fachliche Funktionalität vollständig durch den funktionellen Teil abgedeckt wird, kann in der Hauptsache mit HTML-Code gearbeitet werden. Java-Quellcode wird nur dann benötigt, wenn es darum geht, Seiten dynamisch zu erstellen. Somit wird der Anteil von Java-Quellcode auf ein Minimum reduziert, wodurch die JSP-Dateien erheblich an Übersichtlichkeit und Lesbarkeit gewinnen. Für Seiten, deren Präsentation keine dynamischen Anteile benötigt, können auch nur aus HTML-Code bestehende Dateien verwendet werden. Bei der Gestaltung dieser Seiten sollte darauf geachtet werden, daß die Möglichkeit zum Auslösen von Ereignissen nur dann vorhanden sein sollte, wenn dies im zugehörigen Zustandsmodell auch möglich ist. Dadurch, daß die Seiten dynamisch erstellt werden können, ist dies problemlos möglich.
Zur Realisierung der Handhabung werden Servlets[55] verwendet. Servlets bieten die gesamte benötigte Grundfunktionalität für das Verwalten von technischen Benutzungssitzungen (Sessions)[56]. In ihnen kann fast ausschließlich mit der Programmiersprache Java gearbeitet werden, so daß an dieser Stelle die volle Mächtigkeit einer objektorientierten Sprache genutzt werden kann. Dies ist insofern von Nutzen, da mit Vererbung gearbeitet werden kann. Das heißt, alles was sich von einem konkreten Anwendungsfall abstrahieren läßt, kann durch eine Basisklasse zur Verfügung gestellt werden. In einer entsprechenden Subklasse muß nur Bezug auf den konkreten Anwendungsfall genommen werden; der Vorgang zur Abarbeitung eines Request bleibt immer gleich. So kann das Event-Handling und das Konvertieren von Daten immer nach dem gleichen Schema ablaufen. Das einzige was hierfür vorhanden sein muß, ist eine Zuordnung von der String-Repräsentation eines Objektes, Wertes oder Ereignisses zu seinem entsprechenden fachlichen Gegenstück. Gleiches gilt für das Dispatching, für welches nur Listen mit einer Zuordnung von Systemzustand zur entsprechenden HTML- bzw. JSP-Datei geführt werden müssen.
Durch die Trennung von Präsentation und Handhabung ist es zudem möglich, nur ein Servlet[57] für die gesamte Anwendung zu benutzen. Das Wissen über das Event-Handling, Auslesen und das Dispatching ist folglich nur an einer Stelle im System vorhanden. Hierdurch können die Probleme, die durch eine Verteilung des Handhabungswissens entstehen, vermieden werden.
Die beiden oben geschilderten Komponenten verhalten sich bei ihrer Verwendung wie folgt:

---

[53] vgl. Abschnitt 3.1.2 **Technik und Aufbau von Webtop-Anwendungen** und Abschnitt 3.1.3 **Probleme bei der Konstruktion von Webtop-Anwendungen**

[54] siehe [JSP 2000]

[55] siehe [Servlets 2000]

[56] Für die Realisierung eines fachlichen Benutzungsvorgangs sind Servlets nicht geeignet, da sie durch ihren sehr technischen Charakter immer von Web-Technologien wie dem HTTP-Protokoll abhängig sind.

[57] Dieses Servlet, welches die Handhabung einer Anwendung modelliert, wird im Folgenden als Dispatcher bezeichnet.

Der Benutzer macht im Browser seine Eingaben und löst dann ein Ereignis aus. Die Ereignisse werden im Quellcode als Parameter in einem Link oder in einem hidden field angegeben. Wird der entsprechende Link oder Button betätigt, wird der Bezeichner des Ereignisses als Parameter mit einem Request übergeben. Das Servlet nimmt den Request entgegen und hat nun die Aufgabe, die Eingaben und das Ereignis zu extrahieren. Anschließend muß das Ereignis mit den Eingaben an die Komponente übergeben werden, die das Zustandsmodell realisiert. Diese Komponente sorgt dafür, daß die notwendigen Operationen an den Fachlichen Services ausgeführt werden und der Zustand entsprechend den Ergebnissen dieser Operationen gesetzt wird. Im Dispatcher muß anschließend, entsprechend des aufgetretenen Ereignisses und dem Zustand der Anwendung, eine Folgeseite ausgewählt und zur Anzeige gebracht werden. Bei der Folgeseite kann es sich, wie schon erwähnt, um eine Java Server Page oder um eine HTML-Datei handeln.
Die nachfolgende Abbildung zeigt die Ergebnisse des eben geschilderten Abschnittes in der Übersicht:

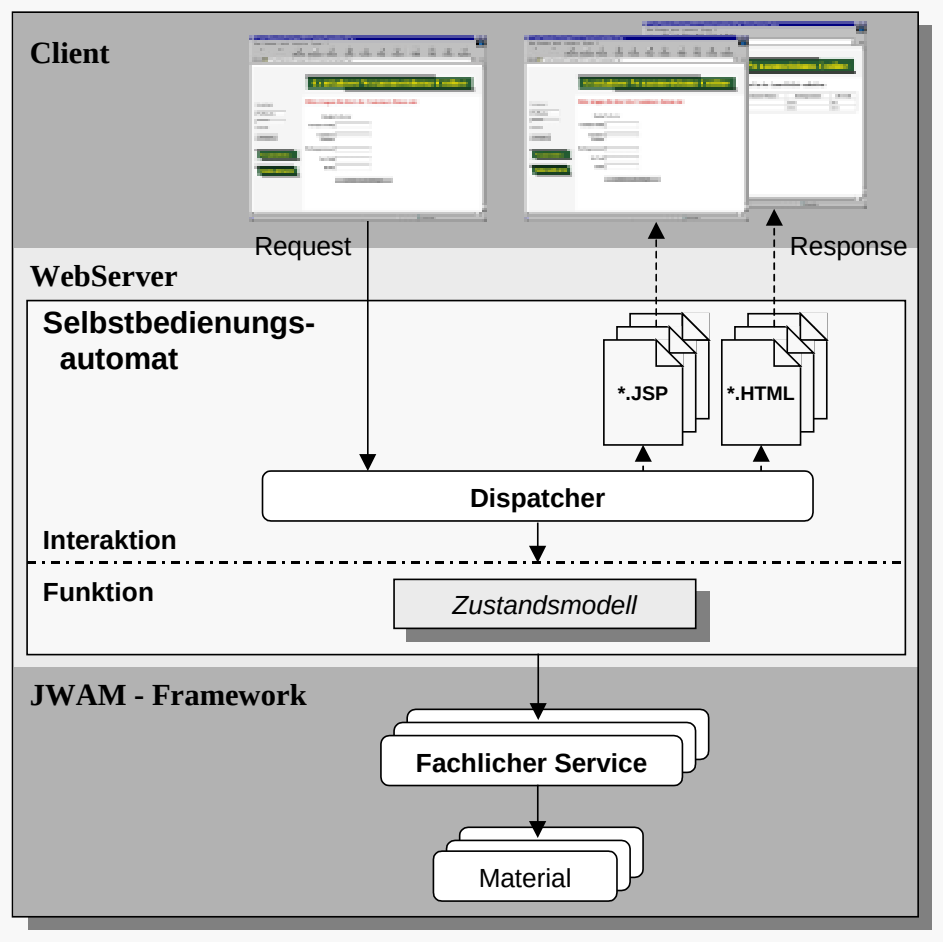

Abbildung 3.7: Verwendung von Dispatcher und JSP/HTML im Entwurf

3.5 Aufgaben des Zustandsautomaten

Nachdem in Abschnitt 3.4 alle mit den Komponenten bzw. Konzepten des WAM-Ansatzes lösbaren Konstruktionsprobleme aufgeführt und die entsprechenden Lösungsmöglichkeiten geschildert wurden, beschreibt der folgende Abschnitt die Aufgaben eines Zustandsautomaten, welche sich aus den verbleibenden Konstruktionsproblemen und den beim Entwurf des Selbstbedienungsautomaten getroffenen Designentscheidungen ergeben.

Der Zustandsautomat soll den internen Bearbeitungszustand einer Anwendung kapseln und bei allen Arbeitsschritten die fachliche Konsistenz wahren. In diesem Fall bedeutet das, daß auf die Funktionalität Fachlicher Services nur über diesen Zustandsautomaten zugegriffen werden kann. Ein Methodenaufruf an einen Fachlichen Service kann also nur erfolgen, wenn der aktuelle Zustand des Zustandsautomaten dies auch zuläßt. Wird ein Ereignis am Zustandsautomaten registriert, welches eine Änderung von Materialien zur Folge hat, so muß im Automaten die entsprechende Methode am Fachlichen Service aufgerufen werden. Die Methode wird ausgeführt, und der Automat ändert entsprechend seinen Zustand. Darüber hinaus muß der Zustandsautomat verschiedene Fachliche Services so miteinander synchronisieren, daß ein gemeinsames, fachlich sinnvolles Benutzungs- und Zustandsmodell eingehalten wird.
Die nachfolgende Definition faßt die Aufgaben eines Zustandsautomaten zusammen:

> **DEFINITION: ZUSTANDSAUTOMAT**
>
> *Der Zustandsautomat hat die Aufgabe, über ein explizites Zustandsmodell den internen Bearbeitungszustand einer oder mehrerer Softwarekomponenten nach außen deutlich zu machen und nur solche Operationen an ihnen zulassen, die im aktuellen Zustand möglich sind. Ändert sich durch eine Operation der Bearbeitungszustand der Komponenten, so ändert auch der Zustandsautomat seinen Zustand entsprechend.*
>
> *Der Zustandsautomat wahrt bei allen Arbeitsschritten die anwendungsfachliche Konsistenz des Systems.*

3.6 Zusammenfassung

In diesem Kapitel wurden Web-Anwendungen als Anwendungskontext für Vorgangssteuerungen eingeführt.
Zunächst wurden die Eigenschaften von Webtop-Anwendungen und die generellen Probleme bei ihrer Konstruktion analysiert. Des weiteren wurden Webtop-Anwendungen anhand von ausgewählten Anwendungsbeispielen in den WAM-Ansatz eingeordnet. Für die untersuchten Beispiele wurde festgestellt, daß es sich um Selbstbedienungsautomaten handelt. Darüber hinaus wurde gezeigt, daß es aus Gründen der Benutzerführung und der Einhaltung eines fachlich korrekten Benutzungsmodells nötig ist, Vorgänge über Zustandsmodelle abzubilden.
Anschließend wurde, unter Verwendung der Konstruktionsprinzipien des WAM-Ansatzes, beispielhaft ein Selbstbedienungsautomat für Webtop-Anwendungen erstellt. An dem Selbstbedienungsautomaten wurde gezeigt, wie sich eine Komponente, welche ein Zustandsmodell realisiert, in den Entwurf einfügen sollte. Bei der Umsetzung der fachlichen Funktionalität wurde die Designentscheidung getroffen, die Zustandsmodelle Fachlicher Services in das Zustandsmodell des Selbstbedienungsautomaten zu integrieren. Hierdurch war es möglich, Fachliche Services miteinander zu synchronisieren, ohne die Forderung nach der Kapselung fachlichen Wissens zu verletzen.
Des weiteren wurde im letzten Abschnitt eine Zusammenfassung der Aufgaben eines Zustandsautomaten gegeben.

Kapitel 4: Beschreibungsmöglichkeiten für Zustandsmodelle

Bevor ein konkreter Lösungsvorschlag für die Implementation eines Zustandsautomaten erläutert wird, wird im Folgenden untersucht, wie sich Zustandsmodelle beschreiben lassen und welche Konsequenzen dies für den Entwurf eines Zustandsautomaten hat.
Hierbei wird die Frage geklärt, warum der Aufwand, der für eine solche Beschreibung zu leisten ist, vertretbar ist, und welche Vorteile die Spezifikation eines Zustandsmodells durch einen Formalismus mit sich bringt.

4.1 Gründe für die Verwendung eines formalen Beschreibungsmittels

Zunächst stellt sich die Frage, warum ein Formalismus zur Beschreibung von Zustandsmodellen sinnvoll bzw. warum er essentiell wichtig ist, um einen solchen Zustandsautomaten modellieren zu können.
Im vorigen Kapitel wurde festgestellt, daß Selbstbedienungsautomaten über ein explizites Zustandsmodell mit entsprechenden Zustandsübergängen verfügen sollten, um deutlich zu machen, in welchem Zustand sich das System befindet und welche Arbeitsschritte in einem bestimmten Zustand möglich sind. Durch dieses Modell und einen entsprechenden Zustandsautomaten, der dieses Modell realisiert, soll der Anwender durch die Benutzung des Systems geleitet werden. Daneben kann ein solches Zustandsmodell dazu verwendet werden, um für den Entwickler klarzustellen, wann welche Arbeitsschritte erfolgen und welche Resultate möglich sind. Auf diese Weise können typische Probleme bei der Entwicklung von Webtop-Anwendungen vermieden bzw. entschärft werden. Hierzu gehört zum Beispiel, daß darauf zu achten, bei jedem Arbeitsschritt alle notwendigen Informationen vorzuhalten bzw. dafür zu sorgen, daß diese Informationen in vorangegangenen Arbeitsschritten eingegeben wurden. Daneben muß auch garantiert werden, daß das Zustandsmodell eingehalten wird. Es darf keine Möglichkeit geben, durch unsachgemäße Handlungen des Anwenders oder durch Fehler bei der Modellierung des Systems, in einen vom Entwickler nicht vorhergesehenen Systemzustand zu kommen.
Durch die Verwendung eines Formalismus können Fehler bei der Modellierung vermieden werden, und aufbauend auf dem entstandenen formalen Modell, kann ein entsprechender Zustandsautomat erstellt werden.

4.2 Wahl des formalen Beschreibungsmittels

Nachdem die Frage nach der Notwendigkeit eines formalen Beschreibungsmittels bei der Modellierung von Selbstbedienungsautomaten erörtert wurde, soll nun geklärt werden, welches Beschreibungsmittel für die Modellierung von Selbstbedienungsautomaten am geeignetsten ist. Dieses Beschreibungsmittel sollte wohlstrukturiert, eindeutig lesbar und möglichst leicht verständlich sein. Darüber hinaus sollte es möglichst einfach erweiter- und wartbar sein.
Im Vorfeld dieser Arbeit wurde die Verwendung von *Statecharts* oder von endlichen Automaten diskutiert. Letztendlich fiel die Wahl des Beschreibungsmittels auf die Statecharts

nach Harel (siehe [Harel 1987a]), da diese über einige Vorteile gegenüber anderen Formalismen verfügen und auch zunehmend in der Softwareentwicklung[58] verwendet werden.
Im Folgenden soll in die zum Verständnis notwendigen Grundlagen des Statechart-Formalismus eingeführt und ein beispielhaftes Statechart-Diagramm, für das im letzten Kapitel erwähnte Beispiel der Containervoranmeldung, erstellt werden. Anschließend wird der Statechart-Formalismus mit dem der *endlichen Automaten*[59] verglichen, um zu zeigen, welche Vorteile Statecharts gegenüber endlichen Automaten haben[60].

4.3 Grundlegende Eigenschaften und Begriffe der Statecharts

Der anschließende Abschnitt, welcher auf [Laue&Lietke 1998] basiert, soll einen kurzen Einblick in Begriffe und die Notation von Statecharts geben. Eine ausführlichere Beschreibung mit einem Überblick über das gesamte Spektrum dieses Formalismus ist bei [Laue&Lietke 1998] oder bei [von der Beck 1994] zu finden.
Bei den Statecharts handelt es sich um einen in den achtziger Jahren entwickelten graphischen Formalismus (siehe [Harel 1987a], [Harel 1988]) zur Beschreibung komplexer Systeme. Dieser Formalismus ist eine Erweiterung der endlichen Automaten. Grundsätzlich weisen Statecharts dieselbe Mächtigkeit wie endliche Automaten auf. Darüber hinaus läßt sich jeder Statechart in einen äquivalenten endlichen Automaten überführen.
Statecharts beschreiben reaktive Systeme. Reaktive Systeme werden im Gegensatz zu transformierenden Systemen, welche nur Funktionen berechnen, über externe Ereignisse beeinflußt. Sie stehen also ständig zu ihrer Umwelt in Beziehung.
Im Folgenden sollen zunächst grundlegend die graphische Syntax und einige wichtige Begriffe der Statecharts erläutert werden. Anschließend wird dann auf einige Besonderheiten zur Interpretation von Statechart-Diagrammen eingegangen.

4.3.1 Syntax

Die grundlegenden Elemente der Statecharts sind Zustände. Sie beschreiben den Status eines Systems während einer bestimmten Zeitspanne. Ihre Darstellung in Statechart-Diagrammen erfolgt durch abgerundete Rechtecke, wie das Beispiel in Abbildung 4.1 zeigt. Sie werden dabei in der Regel mit lateinischen Großbuchstaben beschriftet.

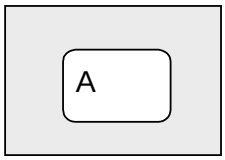

Abbildung 4.1: Darstellung eines Zustandssymbols

Eine wichtige Eigenschaft von Statecharts ist die Möglichkeit, Zustandshierarchien zu bilden. Diese werden dadurch gebildet, daß Zustände über Subzustände verfügen können. Solche Beziehungen werden durch graphischen Einschluß (siehe Abbildung 4.2) ausgedrückt. Hierarchien erlauben es, zusammengehörige Zustände zu erkennen. Darüber hinaus ist hierdurch die Möglichkeit gegeben, Statecharts weiter zu verfeinern. Es ist also möglich, mittels eines Top-Down-Verfahrens zuerst einen groben Entwurf eines

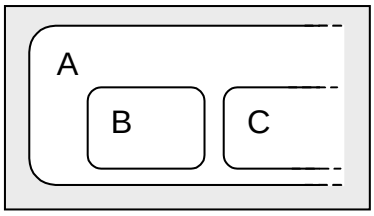

Abbildung 4.2: Darstellung einer Hierarchiebeziehung

[58] Statecharts werden zum Beispiel in der UML verwendet, um Zustandsmodelle von Systemen oder Objekten abzubilden (siehe [Booch 1999]).
[59] Die Grundkonzepte endlicher Automaten werden in dieser Arbeit als bekannt vorausgesetzt. Ausführlichere Erläuterungen zum Formalismus der endlichen Automaten finden sich unter anderen bei [Hopcraft 1994] (Seite 13-58).
[60] vgl. [Harel 1987b]

Zustandsmodells zu entwerfen und diesen dann zu konkretisieren. Es können aber auch nachträglich Zustände zusammengefaßt werden, wenn sie eine gemeinsame Eigenschaft aufweisen. Mit den Statecharts ist also auch eine Bottom-Up-Entwicklung möglich.
Jedem Zustand eines Statecharts wird einer der Typen XOR, AND oder BASIC[61] (oder auch *Basiszustand*, dies bedeutet, daß er keine Subzustände besitzt) zugeordnet (siehe Abbildung 4.3 bis 4.5).

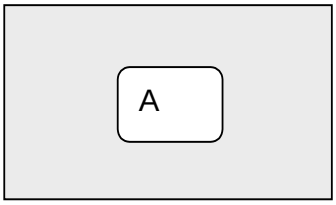

Abbildung 4.3: Basiszustand

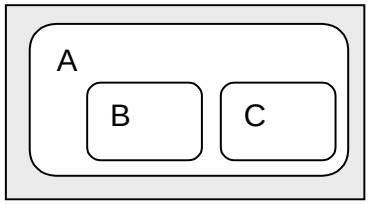

Abbildung 4.4: XOR-Dekomposition

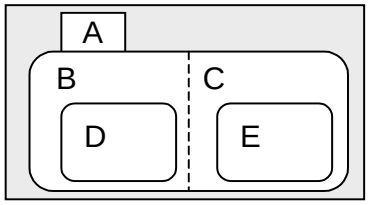

Abbildung 4.5: AND-Dekomposition

Verschiedene Zustände können also in genau einer der Beziehungen *ausschließend*, *orthogonal* oder *hierarchisch* zueinander stehen. Diese Beziehungen lassen sich durch die Darstellung eines Statecharts als Zustandsbaum (siehe Abbildung 4.6) gut deutlich machen. Auch die Hierarchieebenen des Statechart-Diagramms lassen sich gut erkennen. Für den besonderen Zustand ROOT, der die oberste Ebene der Systembeschreibung darstellt, gilt, daß er kein Subzustand eines anderen Zustands ist.

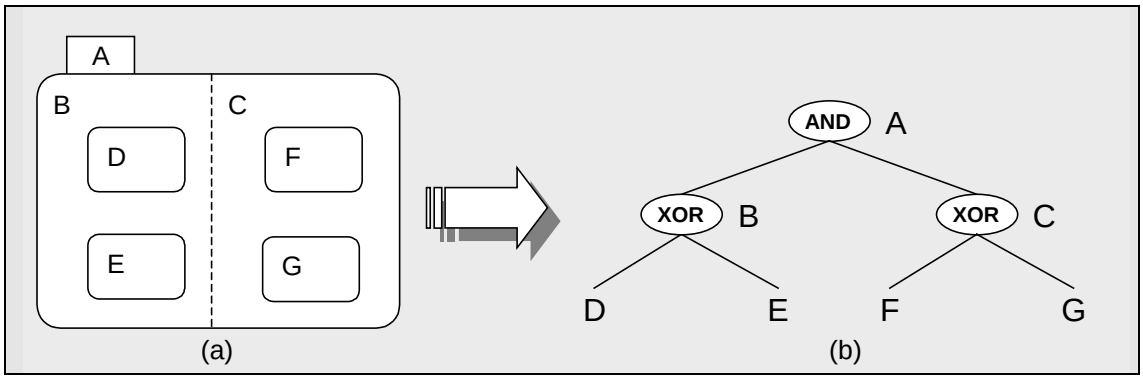

Abbildung 4.6: Darstellung eines Statechart-Diagramms (a) als Zustandsbaum (b)

Zustände, welche in einer Exklusiv-Oder-Beziehung (XOR) zueinander stehen, schließen sich gegenseitig aus. Dies bedeutet, wenn der übergeordnete Zustand betreten wird, kann nur einer seiner Subzustände ebenfalls eingenommen werden. Im Falle einer XOR-Dekomposition spricht man folglich von einander ausschließenden Subzuständen. Die Möglichkeiten, den initialen Zustand zu bestimmen, werden im Folgenden noch beschrieben.
Der Statechart-Formalismus stellt neben der bisher vorgestellten Möglichkeit der XOR-Dekomposition zur Darstellung von Zuständen die Möglichkeit einer AND-Dekomposition zur Verfügung. Durch sie wird festgelegt, daß alle Subzustände eines Zustandes gleichzeitig eingenommen werden, wenn dieser betreten wird. Die Subzustände eines mittels der AND-Dekomposition verfeinerten Zustandes nennt man *orthogonal*. In solchen Zuständen ablaufende Vorgänge können unabhängig voneinander ausgeführt werden. Orthogonalität kann, wie auch die XOR-Dekomposition, auf jeder Hierarchieebene eingesetzt werden, um Zustände genauer zu spezifizieren. Die graphische Repräsentation von Orthogonalität erfolgt

[61] Im weiteren Verlauf der Arbeit werden Zustandstypen durch die folgende SCHREIBWEISE gekennzeichnet.

mit Hilfe gestrichelter Linien, welche die einzelnen AND-Komponenten voneinander trennen (siehe Abbildung 4.5).
Für die Beschreibung der Menge von Zuständen, in denen ein System zu einem bestimmten Zeitpunkt verweilt, gibt es verschiedene Bezeichnungen. Die hier gegebene Definition von *Zustandskonfiguration* berücksichtigt die einzelnen Varianten, indem eine weitere Aufgliederung des Begriffs vorgenommen wird. Es erfolgt eine Trennung zwischen *partiellen* und *maximalen* Zustandskonfigurationen sowie eine Unterscheidung anhand des Zustandstyps, der in den Zustandskonfigurationen zulässig ist. Eine partielle Zustandskonfiguration ist eine relativ zum Zustand ROOT orthogonale Menge von Zuständen beliebiger Ebenen. Wenn diese Menge maximal ist, bezeichnet man sie kurz als Zustandskonfiguration. Wenn eine (partielle) Zustandskonfiguration nur aus Basiszuständen besteht, wird sie (partielle) *Basiskonfiguration* genannt. Zusätzlich kann der Begriff der *vervollständigten* Konfiguration eingeführt werden. Diese enthält neben den Zuständen der Basiskonfiguration auch die entsprechenden Zustände der höheren Ebenen.
Die Zustandsübergänge werden durch beschriftete Pfeile dargestellt, die Zustände beliebiger Hierarchieebenen miteinander verbinden können (siehe Abbildung 4.7). Die Beschriftung eines Pfeils besteht immer aus einem Bezeichner in lateinischen Kleinbuchstaben für das *Ereignis*, welches den tatsächlichen Übergang bewirkt. Solche Ereignisse basieren auf sogenannten primitiven Ereignissen, welche auf dem Verlassen oder dem Einnehmen von Zuständen und der Änderung des Ergebnisses von *Bedingungen* oder von *Ausdrücken* beruhen. Durch logische Verknüpfungen können die Ereignisse beliebig kombiniert werden.

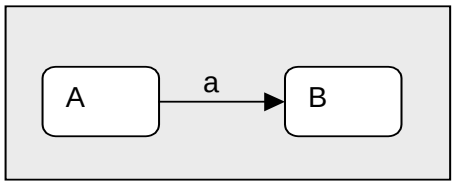

Abbildung 4.7: Darstellung eines Zustandsübergangs

Bedingungen werden von einer Basismenge ausgehend definiert. Diese kann zum einen aus sogenannten primitiven Bedingungen bestehen, zum anderen aus Bedingungen, welche zum Beispiel prüfen, ob sich das System in einem bestimmten Zustand befindet oder ob eine bestimmte Aktivität läuft. Weitere Bedingungen lassen sich erstellen, indem man die Verknüpfungen $\vee$, $\wedge$ und $\neg$ auf durch $=$, $\neq$, $\leq$, $\geq$, $<$ oder $>$ verbundene Ausdrücke oder Bedingungen anwendet. Für die in Aktionen und Ereignissen verwendbaren Bedingungen werden wie für Ausdrücke Ereignisse generiert, wenn sich ihr Wert ändert. Hierbei bedeutetet zum Beispiel das Ereignis *true( )*, daß ein Ausdruck wahr wird, während *false( )* für eines steht, welches den Wahrheitswert falsch annimmt.
Die Menge der Ausdrücke wird auf der Basis der natürlichen Zahlen und einer Menge von Variablen gebildet, indem algebraische Operationen auf schon erstellte Ausdrücke angewendet werden. Der entsprechende Pfeil wird zusätzlich mit einem Bezeichner in griechischen Kleinbuchstaben für diese Bedingung versehen, welcher in runden Klammern hinter dem Ereignisbezeichner aufgeführt wird. Die Abbildung 4.8 zeigt ein Beispiel für einen solchen Zustandsübergang.

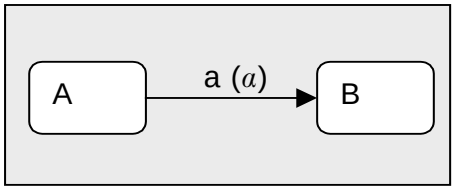

Abbildung 4.8: Zustandsübergang mit Bedingung

Neben der Möglichkeit auf Ereignisse zu reagieren, können in Statecharts Ereignisse durch Aktionen ausgelöst werden. Aktionen werden hierbei als Geschehnisse von extrem kurzer Dauer betrachtet. Bei den Aktionen lassen sich zwei Arten unterscheiden:

Zum einen gibt es Aktionen, welche unmittelbar auf die Umgebung des reaktiven Systems wirken und zum anderen solche, die Zustandsübergänge in orthogonalen Komponenten auslösen. Die Aktionen der ersten Kategorie sind Ausgaben des Systems. Sie bestehen aus primitiven Ereignissen und Zuweisungen von Bedingungen an primitive Bedingungen oder von Ausdrücken an Variablen. Die Aktionen der zweiten Gruppe sind von den einzelnen Statecharts selbst generierte Ereignisse, die in allen orthogonalen Komponenten registriert werden. Registrieren bedeutet, daß ein Zustandsübergang ausgelöst wird. Der Bereich, in dem auf Aktionen reagiert wird, ist immer auf den entsprechenden Statechart begrenzt. Einzelne Aktionen sowie Aktionsgruppen, welche durch Semikola getrennt werden (siehe Abbildung 4.9), können für die Beschriftung von Zustandsübergängen eingesetzt werden. Indem man einen Zustand zusätzlich mit einem der Ausdrücke *entry* oder *exit* versieht, kann eine Aktion dann ausgeführt werden, wenn der betreffende Zustand betreten bzw. verlassen wird (siehe Abbildung 4.10).

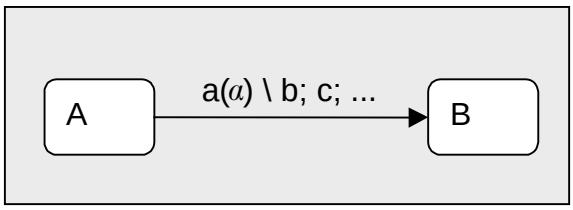

Abbildung 4.9: Darstellungsmöglichkeiten von Aktionen

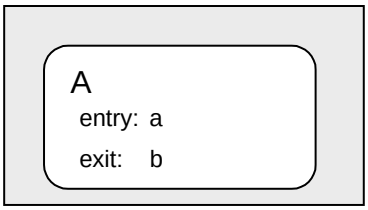

Abbildung 4.10: Darstellungsmöglichkeiten von Aktionen

Durch die Verwendung von Aktionen ist es, wie oben beschrieben, möglich, in orthogonalen Komponenten Zustandsübergänge auszulösen. Auf diese Weise können orthogonale Komponenten miteinander kommunizieren und es wird eine Synchronisation zwischen diesen Komponenten eines Statecharts ermöglicht. Diese Kommunikation wird auch als *broadcast-communication* bezeichnet. Abbildung 4.11 zeigt ein Beispiel für einen solchen Vorgang.

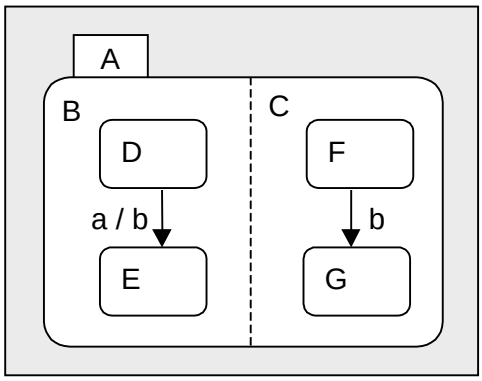

Abbildung 4.11: Auslösen eines Zustandsübergangs in einer orthogonalen Komponente

Nachdem die Syntax von Zustandsübergängen beschrieben wurde, soll im Folgenden eine formale Definition von Zustandsübergängen gegeben werden. Hiernach definiert sich ein Zustandsübergang, auch als *Transition* bezeichnet, folgendermaßen:

> **DEFINITION: TRANSITION (ZUSTANDSÜBERGANG)**
>
> *Eine Transition t besitzt eine Quellmenge X von Zuständen und eine Zielmenge Y, die sowohl Zustände als auch History-Symbole umfassen kann. Ihre Beschriftung l, die als label bezeichnet wird, besteht aus einem Ereignis-Aktionspaar a. Häufig wird auch das Tripel (X, l, Y) als Transition bezeichnet. Transitionen können zur Ausführung ausgewählt werden, wenn sich das System in X befindet und das Ereignis e eintritt; bei dem zugehörigen Zustandsübergang wird X verlassen, die Aktion a ausgeführt und Y betreten.*
>
> nach [Laue&Lietke 1998], Seite 42

Neben den bisher gezeigten Zustandsübergängen gibt es noch einige zusätzliche Möglichkeiten zur Darstellung von Zustandsübergängen, welche die Abbildung einiger häufig vorkommender Anwendungsfälle erheblich einfacher machen.

Statische Reaktionen sind eine einfache Darstellung für einen Spezialfall von Transitionen (siehe Abbildung 4.12). Eine statische Reaktion erhält man, indem die Transition wieder direkt in den Zustand übergeht. Dies bedeutet, daß der Zustand vor Beginn der Transaktion gleich dem Folgezustand ist. Bei einer Vielzahl derartiger Transitionen in einem Zustand kann die Anzahl der orthogonalen Komponenten eines Statecharts verhältnismäßig klein gehalten werden. Ansonsten müßten zusätzliche Zustände in zusätzlichen orthogonalen Komponenten eingeführt werden, welche ein derartiges Verhalten beschreiben.

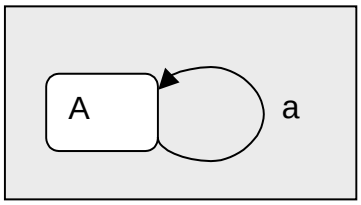

Abbildung 4.12: Darstellung einer statischen Reaktion

Da die beschriebenen Aktionen allerdings als augenblicklich betrachtet werden, während reale Systeme stets durch Abläufe gekennzeichnet sind, die eine bestimmte Zeitdauer beanspruchen, ist eine zusätzliche Erweiterung des Formalismus erforderlich. Hierzu wurde der Begriff der *Aktivität* eingeführt, mit dem Vorgänge bezeichnet werden, welche Zeit benötigen. Es ist zu beachten, daß Aktivitäten selbst zunächst nicht durch Statecharts spezifiziert werden. Aktivitäten können, wenn sie rein reaktiver Natur sind, durch Statecharts beschrieben werden, so daß Hierarchien von Statecharts und Aktivitäten denkbar sind. In Statecharts werden in der Regel nur die Anfangs- und die Endzeitpunkte einer Aktivität spezifiziert. Hierfür werden die Ausdrücke *start( )* und *stop( )* verwendet. Aktivitäten können auf zwei verschiedene Arten in Statechart-Diagrammen benutzt werden. Zunächst gibt es die Möglichkeit, Aktivitäten nach dem Vorbild eines *Mealy-Automaten*[62] zu verwenden. Hierbei werden die Aktivitäten über Zustandsübergänge gestartet bzw. beendet (siehe Abbildung 4.13). Daneben können Aktivitäten gemäß des Vorbildes eines *Moore-Automaten*[63] in einem Statechart modelliert werden. Mit dem Einnehmen eines Zustandes wird, wie in Abbildung 4.14 gezeigt, eine Aktivität gestartet und mit dem Verlassen des Zustandes beendet. Der Bezeichner *throughout* stellt eine Abkürzung für den Fall dar, daß eine Aktivität nur während eines bestimmten Zustands abläuft.

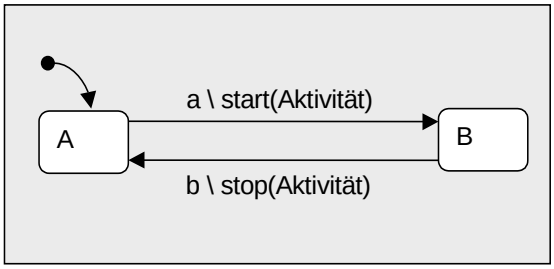

Abbildung 4.13: Steuerung von Aktivitäten über Zustandsübergänge

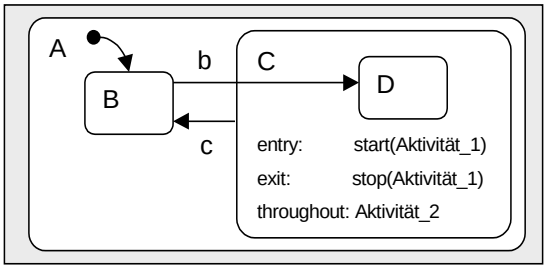

Abbildung 4.14: Steuerung von Aktivitäten über Zustandsübergänge

[62] Bei Mealy-Automaten werden Ausgaben mit einem Zustandsübergang assoziiert. Das heißt, wenn ein Zustandsübergang durchgeführt wird, erfolgt eine Ausgabe (vgl. [Hopcraft 1994], Seite 45).

[63] In Moore-Automaten werden die Ausgaben einem Zustand zugeordnet. Sobald ein Zustand eingenommen wird, wird eine Ausgabe produziert (vgl. [Hopcraft 1994], Seite 44).

Zusätzlich zu Zustandsübergängen gibt es die Möglichkeit, den Folgezustand beim Übergang in einen mittels der XOR-Dekomposition verfeinerten Zustand über s*elektive* und *bedingte Eintritte* festzulegen. Solche Eintritte können verwendet werden, wenn es besondere Beziehungen zwischen Zuständen und Ereignissen oder zwischen Zuständen und Bedingungen gibt.

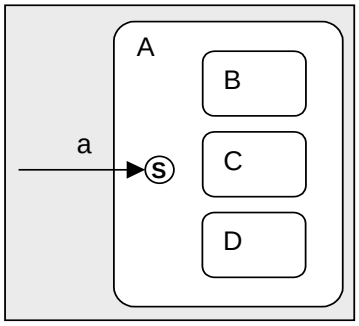

Abbildung 4.15: Selektiver Eintritt

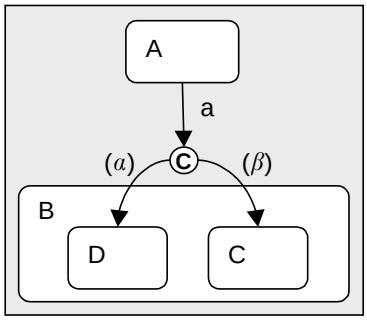

Abbildung 4.16: Bedingter Eintritt

Der selektive Eintritt wird verwendet, wenn es eine 1:1-Beziehung zwischen der Menge der denkbaren Ereignisse, welche einen Zustandsübergang auslösen können, und der Menge der möglichen Folgezustände gibt. Hierbei kann man jedes der Ereignisse als Auswahl aus einer Menge von Optionen ansehen, die durch die einzelnen Zustände repräsentiert werden. Die Zugehörigkeit eines Ereignisses zu einem bestimmten Zustand ist anhand der gewählten Bezeichnungen erkennbar (siehe Abbildung 4.15). Für das Beispiel in der Abbildung könnte ein Zustandsübergang mit „aB“ bezeichnet sein. Das System hätte nach der Durchführung dieses Zustandsübergangs die Konfiguration {A, B}.
Der bedingte Eintritt (Abbildung 4.16) ist für Fälle vorgesehen, in denen ein Ereignis einen Zustandsübergang auslöst, wobei der neu einzunehmende Subzustand eines übergeordneten Zustandes jedoch davon abhängt, ob bestimmte Bedingungen erfüllt sind. Das Diagramm gewinnt dadurch an Übersichtlichkeit, da Sinnzusammenhänge zwischen bedingten Zustandsübergängen kenntlich gemacht werden können.

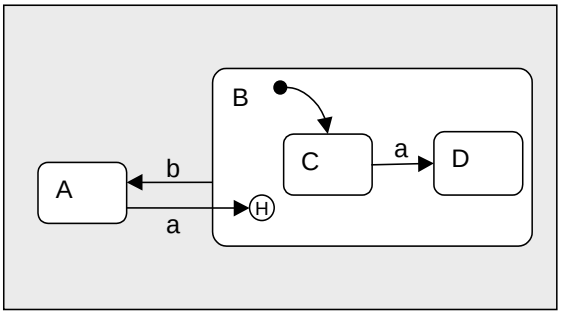

Abbildung 4.17: History-Funktion[64]

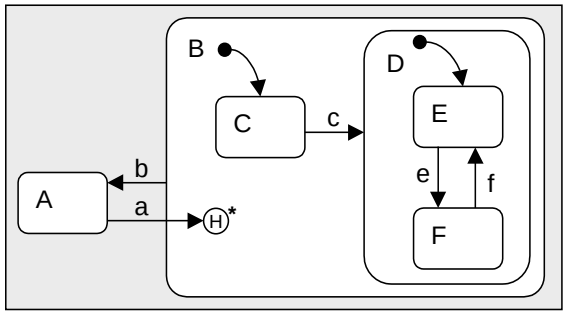

Abbildung 4.18: History-Funktion über mehrere Hierarchieebenen[64]

[64] Zur Verwendung von History-Funktionen gibt es unterschiedliche Varianten. In dieser Arbeit wird nur die gebräuchlichste Variante aufgeführt. Weitere Varianten finden sich bei [von der Beck 1994] oder [Laue&Lietke 1998].

Mit Hilfe einer *History*-Funktion kann derjenige Zustand zum Folgezustand erklärt werden, in dem sich das System befand, als der unmittelbar übergeordnete Zustand zuletzt verlassen wurde. Das Symbol Ⓗ kennzeichnet eine solche Funktion, welche nur für die Zustandsebene gilt, auf der sie auftritt (siehe Abbildung 4.17). Das Symbol Ⓗ* (siehe Abbildung 4.18) wird verwendet, wenn auch auf die Zustände aller untergeordneten Ebenen Bezug genommen werden soll.

Über die bisher genannten Möglichkeiten hinaus, kann ein *Default*-Zustand (siehe Abbildung 4.19) unter den einander ausschließenden Subzuständen bestimmt werden, der eingenommen wird, sofern kein anderer Zustand durch einen Pfeil explizit als Folgezustand gekennzeichnet wird.

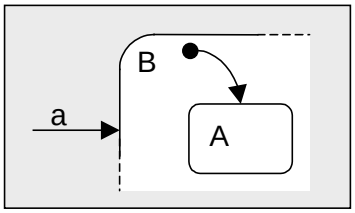

Abbildung 4.19: Notation eines Default-Zustands

4.3.2 Semantik

Leider gibt die graphische Darstellung von Statecharts keine eindeutige Interpretation vor. Dies bedeutet, zu jedem Statechart kann es unterschiedliche Deutungen geben, was dazu führt, daß es aufgrund dieser Mehrdeutigkeiten zu Mißverständnissen bei der Verwendung kommen kann. Aus diesem Grund muß eine Semantik für Statecharts festgelegt werden. In der Literatur werden hierbei der *STATEMATE*-Ansatz (siehe [Harel 1996] oder [Harel 1998]) und der Ansatz der *Microsteps* (siehe [Pnueli 1991]) als die wichtigsten Semantiken[65] genannt.

Allen Semantiken ist der Begriff des *Schrittes* oder *Step* gemein. Darunter versteht man das Erhalten eines oder mehrerer Ereignisse aus der Umgebung, das Vornehmen einer entsprechenden Zustandsänderung und die Generierung eines Ausgabeergebnisses. Allerdings unterscheiden sich die Semantiken in ihrer genauen Definition eines Schrittes.

Der Grundgedanke der Microstep-Semantik ist, daß die durch einen Zustandsübergang hervorgerufenen Veränderungen, wie Zuweisungen an Variablen, Start von Aktivitäten oder das Auslösen von Ereignissen, noch im gleichen Schritt wirksam werden und weitere Zustandsübergänge auslösen können. Eine auf diese Weise gebildete Kette von Zustandsübergängen wird als Step bezeichnet. Für einen solchen Step muß zum einem die Forderung gelten, daß die Menge seiner Zustandsübergänge konsistent ist. Das heißt, die in ihr enthaltenen Transitionen und die durch sie ausgelösten Aktionen müssen strukturell konsistent sein. Dies bedeutet, daß es keine mehrfachen Wertzuweisungen an primitive Variablen oder Bedingungen geben darf. Zum anderen soll die Ausführung der Kettenreaktion keine Zeit beanspruchen, so daß die nächste Zustandskonfiguration im folgenden Schritt vorliegt und keine Unterbrechung durch extern generierte Ereignisse ermöglicht wird. Trotz der Zeitlosigkeit der Kettenreaktion, soll die Reihenfolge der einzelnen Transitionen während der Ausführung Einfluß auf die nächste Zustandskonfiguration haben.

Von außen läßt sich ein solcher Schritt nicht unterbrechen. Es lassen sich nur die ursprüngliche und die neu entstandene Zustandskonfiguration, die während des Schrittes vom Statechart generierten Ereignisse sowie die neuen Werte für die primitiven Bedingungen und Variablen wahrnehmen.

Der STATEMATE-Ansatz basiert auf der grundsätzlichen Überlegung, daß die durch einen Zustandsübergang hervorgerufenen Veränderungen erst im nächsten Schritt Auswirkung haben sollen und somit keine weiteren Zustandsübergänge im aktuellen Schritt auslö-

[65] Auf formale Details dieser Semantiken wird in dieser Arbeit verzichtet, da sie nur wenig zum Verständnis beitragen und den Rahmen der Arbeit sprengen würden. Einen detaillierten Überblick findet man bei [von der Beck 1994]. Dort werden diese Ansätze und sowie einige weitere Semantiken für Statecharts vorgestellt.

sen können. Reaktionen auf externe und interne Ereignisse, wie Zustandswechsel oder auftretende Veränderungen wie Wertänderungen von Variablen, können erst nach Beendigung des Steps registriert werden. Tritt also ein Ereignis ein, so existiert es nur bis zum Ende des nach seinem Eintreten beginnenden Steps. Die Berechnungen innerhalb eines Steps basieren immer auf der Situation am Anfang des Schrittes. In einem Schritt wird immer eine maximale Menge konfliktfreier Transitionen und statischer Reaktionen ausgeführt. Diese Konfliktfreiheit bezieht sich hierbei auf die zu verlassenden Zustände und nicht auf die Wertzuweisung an Variablen.
In dieser Arbeit werden die Statecharts im Sinne der Semantik der Microsteps verwendet, da diese gegenüber dem STATEMATE-Ansatz einige Vorteile bietet. Hierbei stehen die einfache Verifizierbarkeit und Modellierbarkeit der Stacharts als Kriterien im Vordergrund.
Nach der Microstep-Semantik konstruierte Statecharts können im Vorfeld einfacher verifiziert werden, da in ihnen immer genau festgelegt ist, wann eine Transition erfolgt und was diese zur Folge hat. Hieran kann auch durch externe Änderungen wie neue Ereignisse aus der Systemumgebung nichts geändert werden. Beim STATEMATE-Ansatz ist die Verifikation erheblich schwerer, da generierte Ereignisse und Zuweisungen erst im nächsten Schritt wirksam werden. Es könnte dann der Fall eintreten, daß durch hinzugekommene externe Ereignisse bestimmte Zustandsübergänge nicht stattfinden. Das bedeutet zwar, daß sie schneller auf ihre Änderungen in der Umwelt reagieren können, aber bei einer Prüfung müssen erheblich mehr Variablen beachtet werden.
Neben den Vorteilen bei der Verifizierbarkeit sind nach der Microstep-Semantik konstruierte Systeme einfacher zu modellieren, da in ihnen die Berechnungsschritte weitgehend festgelegt sind. Zudem können so entworfene Systeme, die entsprechend verifiziert wurden, leichter miteinander kombiniert werden, da sich auch in diesem Fall nichts an ihrem Verhalten ändern kann. Bei einem nach der STATEMATE-Semantik entworfenen System ist dies nicht sichergestellt. Werden hier mehrere Statecharts zu einem neuen System zusammengefügt, kann es passieren, daß sich diese nicht mehr wie gewünscht verhalten. Deshalb müssen solche Statecharts dann erneut verifiziert werden.

4.4 Verwendung von Statecharts

In diesem Abschnitt wird gezeigt, wie sich Statecharts zur Modellierung von Zustandsmodellen verwenden lassen. Bevor beispielhaft ein Statechart-Diagramm erstellt wird, soll zunächst erläutert werden, wie ein solches Diagramm zu erstellen ist und worauf bei der Erstellung zu achten ist.

4.4.1 Skizzierung eines Vorgehensmodells zur Erstellung von Statecharts

Um ein System durch ein Statechart zu modellieren, wird bei [Booch 1999] (Seite 331ff) das folgende Vorgehen empfohlen. Dabei handelt es sich um eine Anleitung zu einem Top-Down-Vorgehen, bei dem anhand von Szenarien, welche dort als *use cases* bezeichnet werden, ein Zustandsmodell eines Systems erstellt wird.
Zunächst muß das Umfeld des zu modellierenden Systems oder Objektes und seine Interaktion mit diesem Umfeld bestimmt werden. Hierzu werden die use cases verwendet, in denen dieser Kontext und der Umgang mit dem System beschrieben wird.
Im Anschluß daran werden der Anfangszustand und der Endzustand bestimmt, um das System abzugrenzen. Falls es für diese Zustände Vor- oder Nachbedingungen gibt, sollten diese zu Beginn festgelegt werden, damit sie für die weitere Modellierung als Grundlage dienen können.

Im nächsten Schritt sollten die Zustände des Systems oder Objektes in Abhängigkeit von Umgebung und Lebenszyklus des Systems bzw. Objektes festgelegt werden. Hierbei sollte mit Zuständen begonnen werden, welche sich auf einer möglichst hohen Ebene der Modellierung befinden. Diese Zustände können dann in späteren Modellierungsschritten noch durch die Verwendung von Subzuständen verfeinert werden. Dabei ist es günstig, die Zustände nach ihrem Vorkommen im Lebenszyklus des Objektes anzuordnen.
Anschließend muß definiert werden, welche Ereignisse welche Zustandsübergänge auslösen. Beim Einfügen der Transitionen sollte darauf geachtet werden, daß die Übergänge immer nur von einer konsistenten Zustandskonfiguration in die nächste konsistente Konfiguration erfolgen. Mit konsistenten Zustandskonfigurationen sind hier sowohl syntaktisch als auch fachlich korrekte Zustandsmengen gemeint. Danach können gegebenenfalls die Zustandsübergänge mit Aktionen[66] versehen werden.
Die eben genannten Schritte können mehrfach durchlaufen werden. Vor Abschluß der Modellierung sollte noch geprüft werden, ob sich der Statechart zum Beispiel unter Ausnutzung von Orthogonalität oder Hierarchiebildung noch weiter vereinfachen läßt.
Nachdem die Modellierung abgeschlossen ist, sollte noch kontrolliert werden, ob alle Zustände unter gewünschten Kombinationen von Ereignissen erreichbar sind. Des weiteren sollte darauf geachtet werden, daß es keine Zustände gibt, aus denen es keine Möglichkeit gibt, in einen gültigen Endzustand zu gelangen. Bei allen Schritten zur Erstellung eines Statecharts muß natürlich die in Abschnitt 4.3.2 erläuterte Semantik eingehalten werden.

4.4.2 Erstellung eines Statecharts zur Containervoranmeldung

Nachdem in den vorigen Abschnitten die Grundlagen von Statecharts und ein Vorgehen zur Modellierung vorgestellt wurden, soll nun für das Beispiel der Containervoranmeldung ein Statechart-Diagramm erstellt werden[67]. Hierbei wird das in Kapitel zwei beschriebene Szenario als Vorlage genommen. An dieser Stelle wird allerdings nur ein Teil des Gesamtsystems modelliert, da eine vollständige Beschreibung den Rahmen dieses Kapitels sprengen würde.

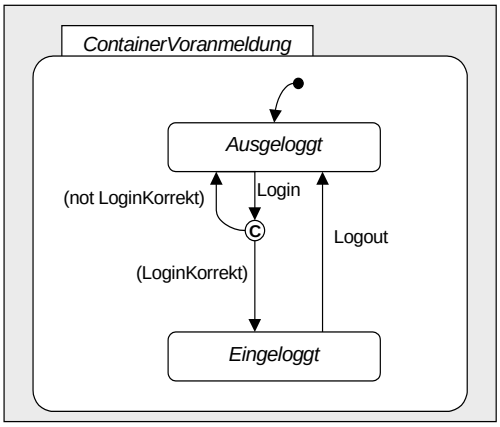

Abbildung 4.20: Statechart der Containervoranmeldung, Grobstruktur

Der Kunde soll laut Szenario die Möglichkeit haben, sich zu Beginn der Sitzung bei der Webtop-Anwendung anzumelden, um sich als zugriffsberechtigter Anwender zu identifizieren. Für die Erstellung des Statecharts bedeutet dies, daß es einen Zustand *Eingeloggt*[68] und einen Zustand *Ausgeloggt* geben muß. Initial ist das System im Zustand *Ausgeloggt*. Wird nun vom Anwender das Ereignis *Login* durch das Ausfüllen der Anmeldemaske und das Abschicken ausgelöst, geht das System in den Zustand

[66] Bei der Verwendung von Aktionen hat es sich im Verlauf der Arbeit als günstig erwiesen, diese stets im Sinne von Mealy-Automaten (siehe [Hopcraft 1994], Seite 44-47) an die Zustandsübergänge zu binden, anstatt sie, wie in Abbildung 4.14 gezeigt, beim Betreten oder Verlassen von Zuständen auszulösen. Dies liegt zum einen daran, daß diese Verwendungsweise näher an der Grundidee von Statecharts liegt, wonach nur Zustandsübergänge Veränderungen am System durchführen und Zustände einen passiven Charakter haben. Zum anderen verbessert sich die Lesbarkeit der Statecharts dadurch erheblich.

[67] Das gleiche Vorgehen wäre natürlich auch für das Traffixx-Projekt anwendbar.

[68] Um Zustände und Zustandsübergänge vom übrigen Text zu unterschieden, werden diese durch eine *kursive Schreibweise* hervorgehoben.

Eingeloggt über. Voraussetzung dafür ist, daß Anwendername und Paßwort korrekt waren, ansonsten verbleibt das System im Ausgangszustand (siehe Abbildung 4.20).

Da der Anwender über die Systemanmeldung hinaus die Möglichkeit haben soll, Container anzumelden bzw. sich schon angemeldete Container anzuschauen, muß der Zustand *Eingeloggt* noch verfeinert werden. Aus diesem Grund wird dieser Zustand in einen orthogonalen Zustand mit den Subzuständen *Anmeldebogen* und *Voranmeldungsliste* überführt (siehe Abbildung 4.21). Nun müssen diese beiden Zustände genauer spezifiziert werden.

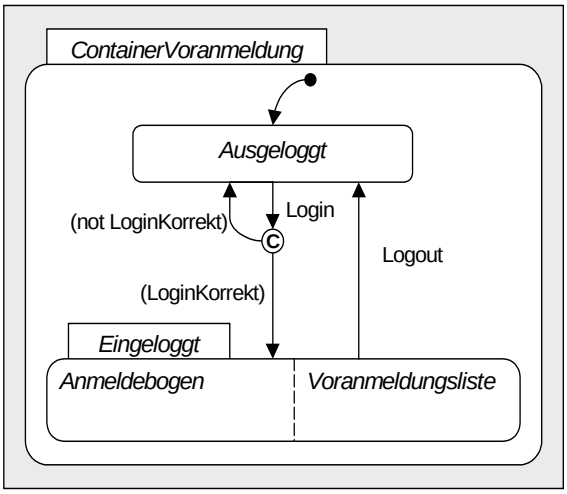

Abbildung 4.21: Statechart der Containervoranmeldung, 1. Verfeinerungsschritt

Da direkt nach dem Einloggen noch keine Containeranmeldung erfolgt ist, befindet sich das Teilsystem *Anmeldebogen* des Statecharts in dem Subzustand *NichtVorhanden*. Dieser Zustand ist also der Default-Zustand des XOR-Zustandes *Anmeldebogen*. Über das Auslösen des Ereignisses *NeuerAnmeldebogen* geht das Teilsystem in den Subzustand *InBearbeitung* über. Dieser Zustandsübergang umfaßt die Generierung eines Materials vom Typ Anmeldebogen und die Bereitstellung dieses Materials für den Anwender in einer geeigneten Bearbeitungsmaske. Der Anwender kann nun seine Angaben in dieses Formular eintragen und dieses über das Auslösen des Ereignisses *Container-Eintragen* mittels des Betätigens eines Buttons oder Links abschicken. Sind die Eingaben nicht korrekt, geht das System in den Zustand *InKorrektur* über. Hier werden dem Anwender die fehlerhaften Eingaben präsentiert, und er hat die Möglichkeit, seine Eingaben zu korrigieren und wiederum abzuschicken. Waren die Eingaben korrekt, wird die Voranmeldung des Containers mittels des Zustandsüberganges *ContainerEintragen* in die Liste der Voranmeldungen übernommen. Das Teilsystem *Anmeldebogen* wird wieder in seinen Anfangszustand *NichtVorhanden* versetzt.

Das Teilsystem der *Voranmeldungsliste* ist defaultmäßig im Zustand *Leer*. Wird im zur *Voranmeldungsliste* orthogonalen Teilsystem *Anmeldebogen* die Aktion *ContainerEintragen* mit der Bedingung, daß die Eingaben korrekt waren, ausgeführt, wird das Ereignis *InListeEintragen* generiert. Als Reaktion geht das Teilsystem *Voranmeldungsliste* in den Zustand *Gefüllt* über. Dieser Vorgang kann beliebig oft erfolgen, was über eine statische Reaktion modelliert wird. In dem Zustand *Gefüllt* ist es nun möglich, sich die Liste der bisher vorgenommenen Voranmeldungen über das Ereignis *ListeAnzeigen* anzeigen zu lassen. Sind alle Voranmeldungen erfolgt, kann über das Ereignis *ListeAbschicken* die Liste der Voranmeldungen endgültig übernommen werden. Vorbedingung ist allerdings, daß das System in seiner aktuellen Konfiguration den Basiszustand *NichtVorhanden* enthält. Ansonsten würde dies bedeuten, daß noch eine Voranmeldung bearbeitet wird. Nachdem die Liste abgeschickt wurde, kann über das Ereignis *Logout* die Anwendung verlassen werden oder es können weitere Voranmeldungen vorgenommen werden. Zusätzlich zu den bisher beschriebenen Möglichkeiten soll der Anwender auch die Möglichkeit haben, die Anwendung trotz nicht vollständig ausgefüllter Voranmeldungen oder einer nicht abgeschickten Liste zu verlassen. Hierzu wird ein weiterer Zustandsübergang namens *Abbrechen* hinzugefügt, welcher ohne Prüfung von Vorbedingungen einen Zustandsübergang von *Eingeloggt* nach *Ausgeloggt* möglich macht.

Die Abbildung 4.22 zeigt den vollständigen Statechart für das hier betrachtete Teilsystem der Containervoranmeldung.

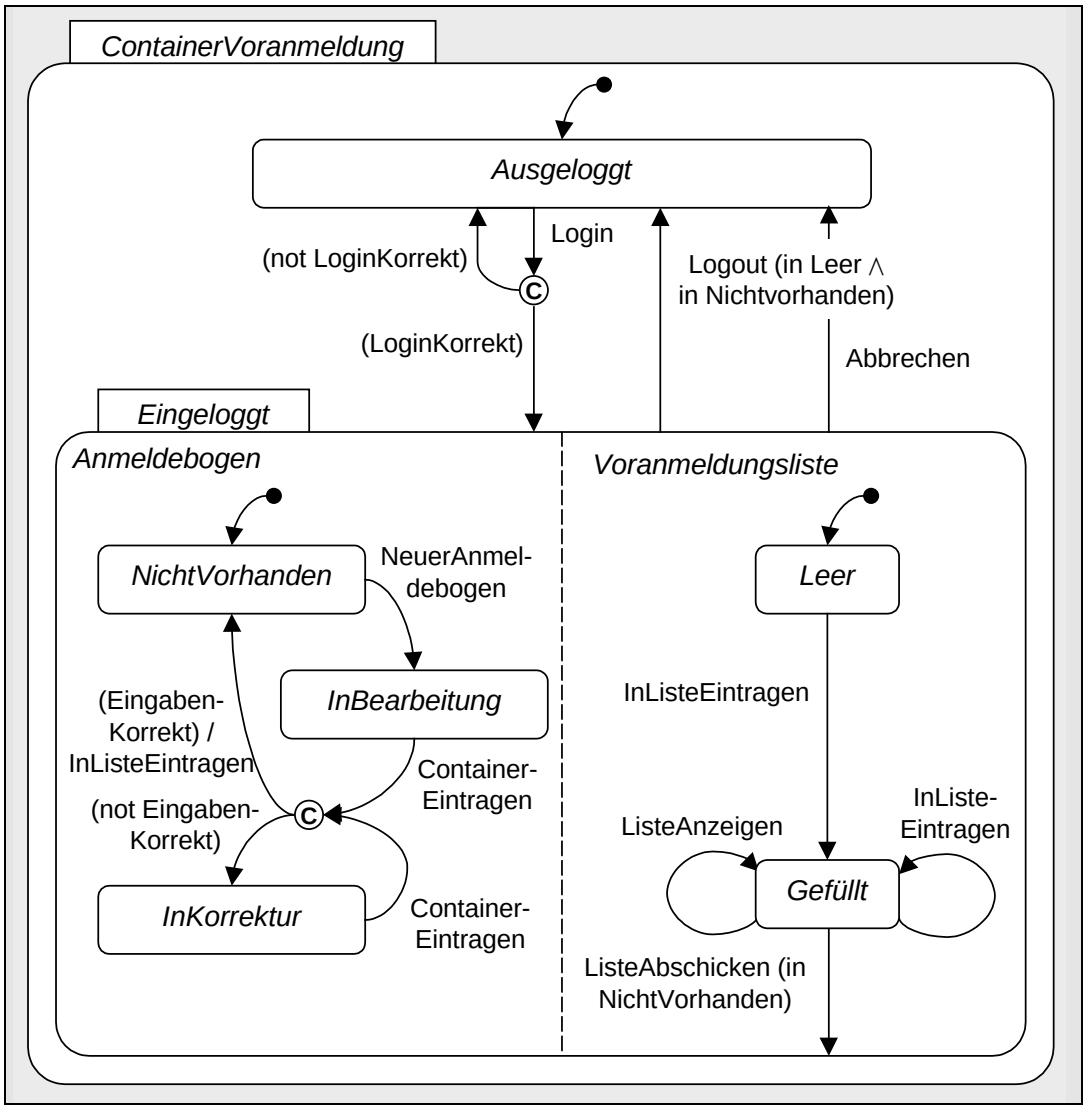

Abbildung 4.22: Statechart der Containervoranmeldung

4.5 Vergleich von Statecharts und endlichen Automaten

Vergleicht man die im vorigen Kapitel vorgestellten Statecharts mit endlichen Automaten, werden einige wesentliche Unterschiede deutlich.

Die Erstellung von endlichen Automaten benötigt einen erheblichen Aufwand. Einen einmal erstellten Automaten nachträglich noch zu ändern oder zu verfeinern, ist nur unter großem zeitlichen Aufwand möglich. Dies liegt darin begründet, daß endliche Automaten kein Konzept für Hierarchien oder Modularität haben. Statecharts hingegen verfügen über diese Optionen, wodurch es möglich ist, die erstellten Statecharts nachträglich unter geringem Aufwand zu verfeinern. Daneben gibt es die Möglichkeit, auf recht einfache und überschaubare Art Zustände nachträglich zusammenzufassen. Die Konzeption des Statechart-Formalismus ermöglicht es im Gegensatz zu den endlichen Automaten, Top-Down- oder Bottom-Up-Verfahren bei der Modellierung zu verwenden.

Des weiteren gibt es bei den endlichen Automaten kein Konzept, durch welches sich Nebenläufigkeit ausdrücken läßt, da sie rein sequentiell sind. Statecharts verfügen sehr wohl über die Möglichkeit, Nebenläufigkeit durch orthogonale Zustände darzustellen.
Bei den endlichen Automaten müssen je Ereignis, welches identische Übergänge aus einer Anzahl verschiedener Zuständen auslöst, für jeden dieser Zustände Pfeile gezeichnet werden. Dadurch erhöht sich die Anzahl der zu zeichnenden Zustandsübergänge gegenüber einem äquivalenten Statechart erheblich. Diese hohe Anzahl von Pfeilen in dem entsprechenden endlichen Automaten verringert dessen Lesbarkeit deutlich. Werden die zu beschreibenden Systeme größer, nimmt die Zahl der darzustellenden Zustände bei den endlichen Automaten erheblich zu. Die Anzahl der Zustände nimmt bei einem linearen Wachstum des Systems exponentiell zu. Dies resultiert aus der Tatsache, daß bei diesem Formalismus immer alle Zustände explizit dargestellt werden müssen. Bei Statecharts kann dieser Effekt durch Hierarchiebildung und die Verwendung von orthogonalen Zuständen verringert werden. Möglich ist dies allerdings nur, falls das zu modellierende System dies zuläßt. Orthogonalität kann zu einer eventuell drastischen Verkleinerung der Zustandsmenge führen. Man denke hier zum Beispiel an zwei orthogonale Komponenten mit jeweils eintausend Subzuständen. Würde man diese durch einen endlichen Automaten modellieren, hätte dies im schlechtesten Fall ein Zustandsdiagramm mit einer Million Zuständen zur Folge. Zusätzlich kann mit Hilfe orthogonaler Zustände die Beschränkung auf Sequentialität der herkömmlichen endlichen Automaten aufgehoben und Nebenläufigkeit dargestellt werden. Wie in Abschnitt 4.3.1 festgestellt, können orthogonale Zustände aufeinander Bezug nehmen. Dies kann zu einer erheblichen Verkleinerung der Diagramme führen.
Betrachtet man die Lesbarkeit von endlichen Automaten, wird gerade bei größeren Systemen deutlich, daß sie nur schwer lesbar sind. Von Laien sind solche Systembeschreibungen in der Regel kaum zu lesen oder zu interpretieren. Durch die Unübersichtlichkeit kann es bei komplexen Systemspezifikationen, welche einen hohen Grad an Fehlerfreiheit erfordern, zu Fehlern bei der Erstellung kommen, welche auch bei einem anschließenden Test nicht erkannt werden. Betrachtet man hingegen die Statechart-Diagramme, so sind diese um ein Vielfaches einfacher zu lesen als ein äquivalenter endlicher Automat, indem sie das visuelle Erscheinungsbild der Zustandsdiagramme durch die Bildung von Hierarchien verbessern. Diese Hierarchien ermöglichen es, fachlich zusammengehörige Teile des zu modellierenden Systems auch im Diagramm zu gruppieren. Bei endlichen Automaten ist dies nur über geeignete Bezeichner für die Zustände möglich, was aber dazu führt, daß die Bezeichner mitunter sehr lang werden, was wiederum die Übersichtlichkeit des Diagramms verringert.
Um die eben genannten Punkte zu verdeutlichen, wurde der Statechart der Containervoranmeldung (siehe Abbildung 4.22) in einen äquivalenten endlichen Automaten überführt. Abbildung 4.23 zeigt das zugehörige Zustandsdiagramm dieses Automaten. Wie man gut sehen kann, hat sich die Anzahl der Zustände erhöht. Gleichzeitig hat sich auch die Zahl der Pfeile vervielfacht, welche gleiche Zustandsübergänge bezeichnen. Des weiteren sind die Informationen über die fachliche Zusammengehörigkeit der Teilsysteme verlorengegangen.

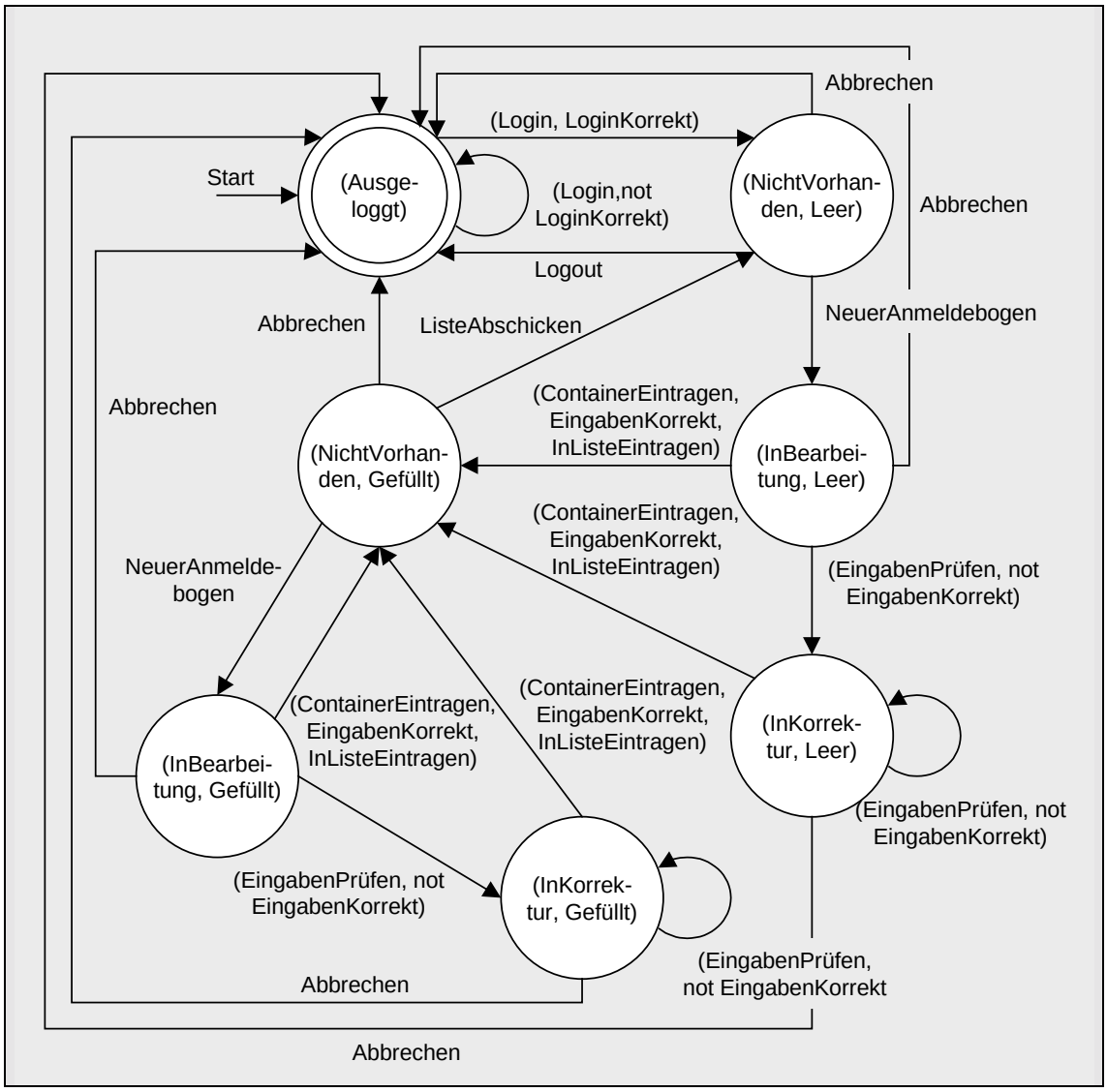

Abbildung 4.23: Die Containervoranmeldung als endlicher Automat[69]

In der nachfolgenden Abbildung 4.24 werden die in diesen Abschnitt genannten Punkte nochmals zusammengefaßt:

[69] Verwendete Notation für Zustandsdiagramme endlicher Automaten (aus [Hopcraft 1994]):
Zustand: ○ Endzustand: ◎ Startzustand: Start → ○ Zustandsübergang : Eingabe / Ausgabe →

Statecharts	Endliche Automaten
• Leichter verständlich als andere formale Modelle	• Nur schwer lesbar
• Leicht wartbar	• Hoher Aufwand bei Erstellung und Wartung
• Schrittweise verfeinerbar	• Nicht schrittweise verfeinerbar
• Orthogonalität ist einfach zu beschreiben	• Nebenläufigkeit nur schwer darstellbar
• Durch die Möglichkeit zur Darstellung von Orthogonalität ist die Anzahl der Zustände reduzierbar	• Bei linearem Wachstum des Systems wächst die Anzahl der Zustände exponentiell
• Bieten die Möglichkeit zur Strukturierung des Diagramms durch die Bildung von Hierarchien und Verwendung von Orthogonalität.	• Strukturierung der Diagramme nur durch Anordnung der Elemente oder Wahl der Bezeichner möglich

Abbildung 4.24: Vergleich von Statecharts und endlichen Automaten

4.6 Zusammenfassung

Mit den Statecharts liegt ein mächtiges, formales Beschreibungsmittel vor, welches sich im Gegensatz zu endlichen Automaten gut dazu eignet, die dynamischen Aspekte reaktiver Systeme in knapper, ausdrucksmächtiger und eindeutiger Form zu beschreiben.
Die wichtigsten Eigenschaften von Statecharts sind ihr reaktiver Charakter, ihre Fähigkeit zur Modellierung von Zustandshierarchien und die Möglichkeit zur Realisierung von Orthogonalität.
Mit Hilfe des in Abschnitt 4.4.1 skizzierten Vorgehensmodells ist es möglich, aus Szenarien ein Zustandsmodell, welches das Benutzungsmodell eines zu entwickelnden Systems abbildet, zu erstellen.
Im nachfolgenden Kapitel wird nun die Frage diskutiert, wie sich die hier beschriebenen Zustandsmodelle in einen Zustandsautomaten umsetzen lassen und wie die Eigenschaften von Statecharts sinnvoll für den Entwurf genutzt werden können.

Kapitel 5: Entwurf des Zustandsautomaten

Das nächste Kapitel beschreibt die Konstruktion des Zustandsautomaten zur Vorgangssteuerung, um das Benutzungsmodell in Form eines Zustandsmodells und eines zustandsabhängigen Verhaltens zu modellieren.
Wie in Abschnitt 3.5 festgelegt, soll der Zustandsautomat den internen Bearbeitungszustand kapseln und bei allen Arbeitsschritten die fachliche Konsistenz wahren. In diesem Fall bedeutet dies, daß auf die Funktionalität Fachlicher Services nur über den Zustandsautomaten zugegriffen werden kann. Ein Methodenaufruf an einen Fachlichen Service kann also nur erfolgen, wenn der aktuelle Zustand des Zustandsautomaten dies auch zuläßt. Wird ein Ereignis am Zustandsautomaten registriert, welches eine Änderung von Materialien zur Folge hat, so muß im Automaten die entsprechende Methode am Fachlichen Service aufgerufen werden. Die Methode wird ausgeführt und der Automat ändert seinen Zustand entsprechend.
Zur Konstruktion solcher Zustandsautomaten sollen zunächst verschiedene Ansätze aus der Literatur untersucht werden. Hierauf aufbauend wird dann ein Entwurf für das JWAM-Rahmenwerk entwickelt, mit dem sich die durch Statecharts spezifizierten Zustandsmodelle umsetzen lassen. Dabei sollen allerdings nur solche Eigenschaften der Statecharts modelliert werden, welche für die hier betrachteten Anwendungsfälle notwendig sind, um keine „Technologie auf Vorrat"[70] zu produzieren.

5.1 Diskussion verschiedener Entwurfsmuster

Zum Problemfeld der Modellierung von Zuständen und Zustandsmodellen in objektorientierten Sprachen gibt es eine Vielzahl von Veröffentlichungen. Bei [Gamma 1996] wird hierzu das *State-Pattern* als Entwurfsmuster eingeführt. In anderen Veröffentlichungen, wie zum Beispiel [Ran 1996] oder [Dyson 1998], wurde dieses Entwurfsmuster erweitert und an die Anforderungen der jeweils zu modellierenden Zustandsmodelle angepaßt. Bei [Yacoub 1998] wird das State-Pattern speziell für die Umsetzung von Statecharts angepaßt. Im Nachfolgenden sollen nun stellvertretend für die verschiedenen Ansätze das ursprüngliche State-Pattern nach [Gamma 1996] und die Erweiterung von [Yacoub 1998] erläutert und auf ihre Verwendbarkeit zur Umsetzung des hier vorliegenden Zustandsmodelles und eine Integration in das JWAM-Rahmenwerk untersucht werden.

5.1.1 Das State-Pattern nach Gamma

Das State-Pattern nach [Gamma 1996] (Seite 305ff) ist ein Entwurfsmuster zur Beschreibung objektbasierter Verhaltensmuster. Hierbei bleibt nach außen die Schnittstelle eines Objektes gleich, während sich das Verhalten des Objektes in Abhängigkeit von seinem Zustand ändert.

5.1.1.1 Die Struktur des State-Pattern

Konstruktiv wird dies gelöst, indem von außen auf ein Kontextobjekt zugegriffen wird. Dieses Objekt definiert für den jeweiligen Klienten die Schnittstelle. Der Klient kann nur über diese Schnittstelle auf Funktionen zugreifen. Um nun das veränderbare Verhalten zu modellieren, verweist das Kontextobjekt auf ein Zustandsobjekt, in welchem das Verhalten

[70] Im Rahmen des WAM-Ansatzes und in den Dokumentationen zum JWAM-Rahmenwerk wird Technologie auf Vorrat als problematisch angesehen, da sich diese nicht an einer realen Anwendungssituation auf ihre Verwendbarkeit überprüfen läßt. Des weiteren „veralten" solche auf Vorrat entworfenen Komponenten schnell, da sie in der Regel nicht intensiv gewartet werden. Dies bedeutet, daß sie, wenn sie zur Anwendung kommen, praktisch neu entworfen werden müssen (siehe [JWAM 2001]).

in einem bestimmten Zustand definiert ist. Je nach Zustand des Systems gibt es folglich ein Zustandsobjekt. Um allen Zustandsobjekten dieselbe Schnittstelle zu geben, wird eine abstrakte Basisklasse eingeführt, von der alle Zustandsobjekte erben. Dem Kontextobjekt sind alle Zustandsobjekte nur über diese Basisklasse bekannt. Die nachfolgende Abbildung 5.1 zeigt die Grundstruktur des State-Pattern.

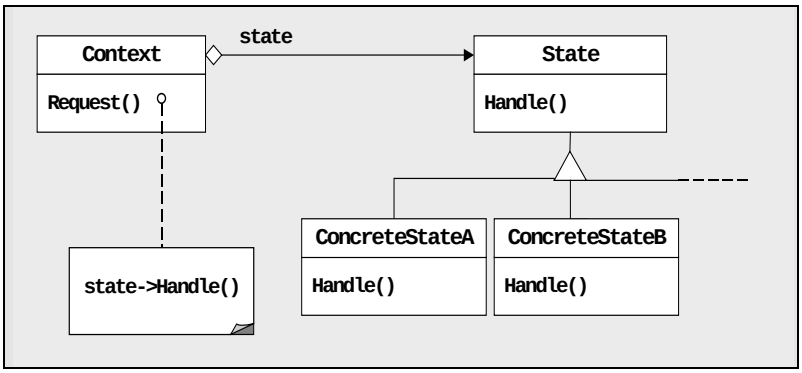

Abbildung 5.1: Struktur des State-Pattern[71]

Ruft ein Klient am Kontextobjekt eine Methode auf, wird der Aufruf an das zur Zeit aktuelle Zustandsobjekt weitergeleitet. Falls es für die Bearbeitung des Aufrufes notwendig ist, kann sich das Kontextobjekt selbst als Argument an ein Zustandsobjekt mitgeben. Hierdurch kann ein Zustandsobjekt auf die Methoden des Kontextobjektes zugreifen.
Wird durch die Ausführung einer Methode ein Zustandsübergang ausgelöst, setzt das Zustandsobjekt am Kontextobjekt den aktuellen Zustand auf den Folgezustand. Zustandsübergänge können aber auch vom Kontextobjekt selbst vorgenommen werden. Dies geschieht zum Beispiel bei der Initialisierung des Systems.

5.1.1.2 Bewertung des State-Pattern

Prüft man das State-Pattern auf seine Nutzbarkeit für die Umsetzung eines Zustandsautomaten unter Berücksichtigung der Modellierung durch Statecharts, stellt man fest, daß die Verwendung dieses Entwurfsmusters einige wesentliche Einschränkungen mit sich bringt. So ist das State-Pattern dafür entworfen worden, flache Zustandsmodelle, wie sie von endlichen Automaten bekannt sind, abzubilden. Aus diesem Grund können einige wichtige Eigenschaften von Statecharts nicht oder nur ungenügend umgesetzt werden.
Solange die Zustandsdiagramme nur Basiszustände enthalten, ist das State-Pattern ohne Einschränkungen anwendbar. Enthält ein zu modellierender Statechart allerdings hierarchische Komponenten, kann diese Eigenschaft der Statecharts nur mühsam modelliert werden. Es ist zwar möglich, über Vererbung Zustandshierarchien zu bilden, allerdings bedeutet dies, daß für jede mögliche Zustandskonfiguration zusätzliche, konkrete Klassen erstellt und gewartet werden müssen. Diese verfügen dann nur über eine geringe Funktionalität und dienen lediglich dazu, Methodenaufrufe weiterzuleiten. Gruppierungen, wie sie in Abbildung 4.22 zur Darstellung fachlicher Zusammengehörigkeit und zum besseren Verständnis der Statecharts eingesetzt werden, sind nur über Bezeichner herstellbar. Orthogonale Komponenten lassen sich nur durch die Einführung zusätzlicher Zustandsobjekte für jede mögliche Zustandskonfiguration innerhalb der orthogonalen Komponente abbilden. Dies hat, wie in Abschnitt 4.5 für endliche Automaten gezeigt, den Effekt, daß sich die Zahl der zu implementierenden Zustandsobjekte erheblich erhöht.
Des weiteren bedingt das State-Pattern sehr umfangreiche Schnittstellen an den Zustandsobjekten. Es müssen immer alle generell möglichen Methodenaufrufe angeboten werden,

[71] aus [Gamma 1996], Seite 306

egal ob sie zur Zeit ausführbar sind oder nicht. An der Schnittstelle der Objekte ist also nicht ersichtlich, welche Methoden gerade ausführbar oder inaktiv sind. Dies muß explizit über sondierende Operationen erfragt werden. Zusätzlich muß für den Fall, daß eine Methode aufgerufen wird, obwohl der durch sie modellierte Zustandsübergang gerade nicht möglich ist, ein entsprechendes Defaultverhalten festgelegt und implementiert werden. Dies kann in der Basisklasse erfolgen, wodurch sich der Aufwand bei der Implementierung deutlich verringern läßt.
Ein weiteres Problem ist die Möglichkeit zur Modifikation von Methoden. Methoden können auf allen Ebenen der Vererbungshierarchie überschrieben werden. Hierdurch kann der Fall eintreten, daß eine Methode in unterschiedlichen Teilbäumen der Vererbungshierarchie überschrieben wird. Dies hat den Effekt, daß Funktionalität dupliziert wird. Je Aufruf, welcher identische Zustandsübergänge aus verschiedenen Zuständen auslöst, muß die entsprechende Methode in dem jeweiligen Zustandsobjekt implementiert werden. Dies verringert die Übersichtlichkeit und die Wartbarkeit des Entwurfs deutlich, denn bei jeder Änderung oder Erweiterung des Zustandsmodells müssen immer alle Zustandsobjekte überprüft werden. Abhilfe kann in diesem Fall nur geschaffen werden, indem per Konvention festgelegt wird, daß Methoden nur in genau einer Ebene der Vererbung überschrieben werden dürfen.
Ferner ist die Verwendung von zyklischen Referenzen problematisch. Dadurch, daß den Zustandsobjekten das übergeordnete Kontextobjekt bekannt gemacht wird, wird das Prinzip der minimalen Kopplung[72] verletzt.
Des weiteren fällt bei diesem Entwurfsmuster noch eine konzeptionelle Schwäche auf. Es werden das gesamte Zustandsmodell und die aktuelle Zustandskonfiguration konzeptionell nicht voneinander getrennt. Dies resultiert aus der Tatsache, daß in diesem Entwurfsmuster keine Modellierung von Zustandskonfigurationen stattfindet. Bei einem flachen Zustandsmodell ist dies noch zu verkraften. Handelt es sich aber um ein Zustandsmodell mit einer Hierarchie, führt dies leicht zu einem schwer verständlichen Entwurf, da immer auf jeder Ebene unterschieden werden muß, welcher Subzustand gerade aktuell ist.
Außerdem werden auch andere Strukturen von Statecharts nicht modelliert, wodurch das Prinzip der direkten Abbildung verletzt wird. So gibt es für Aktionen und Ereignisse keine Entsprechung in diesem Entwurfsmuster.
Neben all diesen Schwächen dieses Entwurfsmusters gibt es noch das Problem, daß das Zustandsmodell und die Zustandsübergänge nicht nach außen explizit gemacht werden. Die Begründung für diese Tatsache liegt darin, daß in dem Entwurfsmuster die ausdrückliche Designentscheidung getroffen wurde, das Zustandsmodell vor den Anwender zu verbergen.
Zusammengefaßt bedeutet dies, daß das State-Pattern sich nur für Systeme mit einem flachen Zustandsmodell, wenigen Zuständen und wenigen Zustandsübergängen eignet. Für derartige Systeme stellt es allerdings eine einfache und leicht umsetzbare Lösung dar. Für eine Umsetzung von Zustandsmodellen, welche mit Hilfe von Statecharts erstellt wurden, ist dieses Entwurfsmuster ohne umfangreiche Anpassungen nicht zu verwenden, da wichtige Eigenschaften nicht modelliert werden können.
Darüber hinaus eignet sich dieser Ansatz nicht für die Integration in ein Rahmenwerk, da für eine Umsetzung viele an ein konkretes Zustandsmodell angepaßte Klassen benötigt werden. Es läßt sich also kaum Funktionalität abstrahieren, welche in ein Rahmenwerk

[72] *Die Bindungen zwischen den Einheiten sollten möglichst reduziert werden, insbesondere zyklische Benutzung sollte vermieden werden.*
aus [Züllighoven 1998], Seite 43

übernommen werden kann. Das bedeutet, daß die Unterstützung, welche durch ein Rahmenwerk erfolgen kann, minimal ist.

5.1.2 Das erweiterte State-Pattern nach Yacoub

Das State-Pattern, wie es bei [Yacoub 1998] beschrieben ist, ist eine Erweiterung des State-Pattern nach [Gamma 1996] zu einem an die Bedürfnisse von Statecharts angepaßten Entwurfsmuster. Hierfür wird das Entwurfsmuster um Komponenten zur Modellierung von Ereignissen, Aktionen, Hierarchien, Orthogonalität, History-Funktionen und Broadcasting-Mechanismen erweitert.

5.1.2.1 Die Struktur des erweiterten State-Pattern

Grundsätzlich werden in diesem Entwurfsmuster die Konzepte des ursprünglichen State-Pattern beibehalten. Hierzu zählen das Kontextobjekt, welches hier als `Interface` bezeichnet wird, sowie die abstrakte Zustandsklasse, die als `AState` bezeichnet wird. Ebenso wird eine zyklische Benutzt-Beziehung zwischen Zustandsobjekt und Kontextobjekt verwendet, welches sich hier sogar in einer expliziten Referenzierung des Kontextobjektes niederschlägt. Genauso werden auch Zustandsübergänge mit einem direkten Setzen des nachfolgenden Zustandes am Kontextobjekt modelliert. An der Kontextklasse `Interface` werden nun nicht mehr direkt Methoden aufgerufen und dieser Aufruf an das jeweilige Zustandsobjekt weitergegeben, sondern es wird auf eingehende Ereignisse reagiert. Die Methode `Event_Dispatcher` nimmt die von dem jeweiligen Klienten erzeugten Ereignisse an und ruft die zugehörige Methode am Zustandsobjekt auf. Abbildung 5.2 zeigt die Struktur des erweiterten State-Pattern:

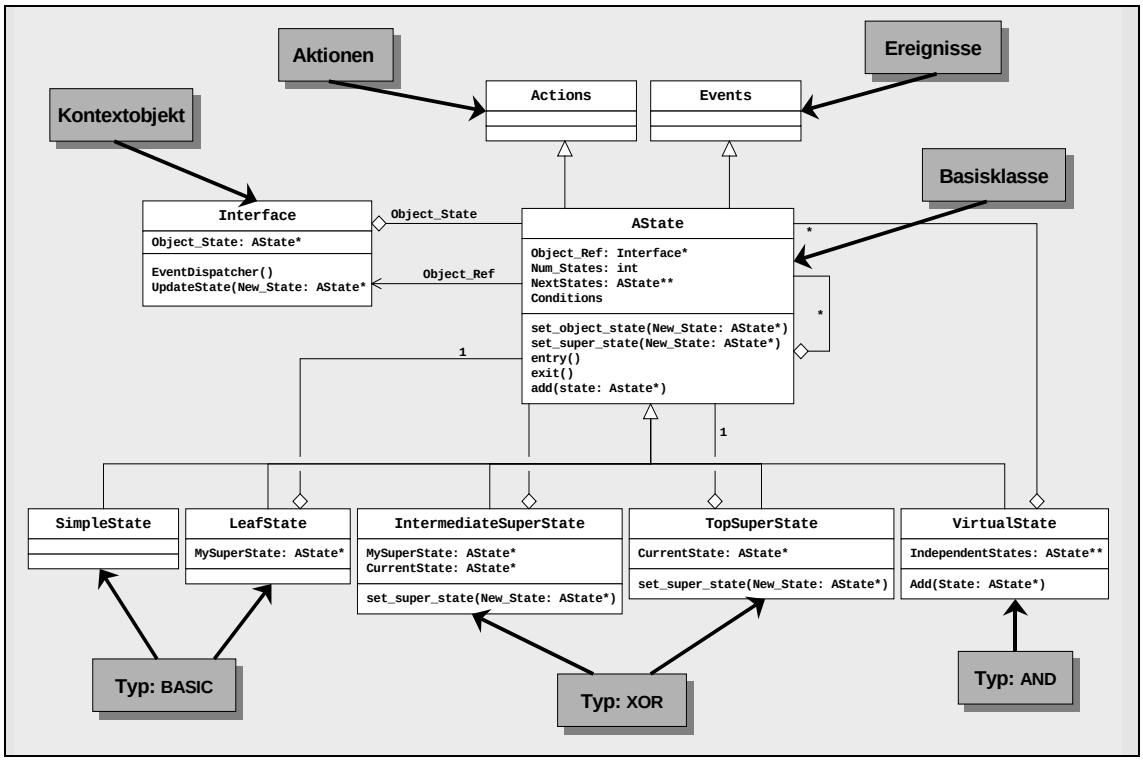

Abbildung 5.2: Struktur des erweiterten State-Pattern[73]

Daneben werden die folgenden Anpassungen an Statecharts vorgenommen:

[73] nach [Yacoub 1998]

Zunächst werden die Klassen `Actions` und `Events` eingeführt, von denen die Klasse `AState` erbt, um die Schnittstelle dieser Klasse festzulegen.
Die Klasse `Events` beinhaltet die Deklaration der Methoden zu allen in einem konkreten Statechart vorkommenden Ereignissen. Der Einfachheit halber wurden die Methoden nach den sie auslösenden Ereignissen benannt. In diesen Methoden werden Zustandsübergänge und der Aufruf von Aktionen an der Klasse `Actions` gekapselt. Beschreibt eine Methode ein für alle Zustände gültiges Verhalten, wird sie in dieser Klasse implementiert, ansonsten wird sie als virtuelle Methode deklariert und in den konkreten Zustandsklassen, welche von `AState` erben, definiert. Dort wird auch das für Zustandsübergänge notwendige Wissen über Folgezustände und Vor- und Nachbedingungen gekapselt.
Die Klasse `Actions` umfaßt alle in einem Statechart vorkommenden Aktionen. Diese dienen dazu, Variablen zu verändern oder Ausgaben zu produzieren. Die Aktionen sind in der Regel als statische Methoden implementiert, um in allen Klassen ein gleiches Verhalten zu garantieren. In den konkreten Zustandsklassen können bei Bedarf noch zusätzliche Aktionen definiert werden.
Die Klassen `Actions` und `Events` müssen für jedes zu modellierende Zustandsmodell jeweils neu implementiert werden.
In der Klasse `AState` wurde die Methode `set_object_state` zum Setzen des Folgezustandes eingeführt. In dieser Methode wird an der Klasse `Interface` mit `UpdateState` der Zustand neu gesetzt. Um den Folgezustand für jede Klasse festzulegen, wird für alle Subklassen die Möglichkeit zur Bestimmung eines Satzes von Folgezuständen durch eine Liste von Zuständen gegeben. Zusätzlich werden in der Klasse `AState` und in deren Subklassen noch Bedingungen für Zustandsübergänge gekapselt. Diese werden dann in den von `Events` ererbten Methoden verwendet. Die Methoden `entry` und `exit` werden bei Zustandsübergängen gerufen, wenn ein Zustand betreten bzw. verlassen wird. An dem betretenen Zustand wird die Methode `entry` ausgeführt, um diesen zu aktivieren. An dem jeweiligen Vorgängerzustand wird dementsprechend die Methode `exit` aufgerufen, um das Zustandsobjekt zu deaktivieren.
Neben diesen Erweiterungen, in den aus dem ursprünglichen State-Pattern stammenden Komponenten, wird auch ein Konzept für die Umsetzung der Hierarchieeigenschaften von Statecharts realisiert. Hierfür werden für die Zustandstypen ROOT, BASIC und XOR besondere Klassen eingeführt. Diese Klassen werden durch Verfeinerungen der Klasse `AState` gebildet.
Die Klasse `SimpleState` modelliert die Zustände vom Typ BASIC, also Zustände, welche über keine Subzustände verfügen. Die Funktionalität dieser Klasse ist vollständig in `AState` enthalten. `LeafState` modelliert ebenfalls Zustände vom Typ BASIC und verfügt über die gleiche Funktionalität wie `SimpleStates`. Zusätzlich enthält diese Klasse noch eine Referenz namens `MySuperState` auf ihren übergeordneten Zustand, um an diesem gegebenenfalls Zustandsveränderungen vorzunehmen.
Die Zustandsklasse XOR wird in der Klasse `TopSuperState` modelliert. Diese Zustandsklasse bildet eine Kapsel um untergeordnete Zustände und leitet Ereignisse an die entsprechenden Subzustände weiter. In dieser Klasse kann ein für alle Subzustände gültiges Verhalten implementiert werden. Das Wissen darüber, welcher Subzustand gerade aktiv ist, wird über die Exemplarvariable `CurrentState` abgebildet. Diese verweist auf den aktuellen Subzustand.
Zustände vom Typ XOR werden auch durch die Klasse `IntermediateSuperState` abgebildet. Diese Klasse vereinigt in sich die Funktionalität von `TopSuperState` und `LeafState`.
Neben der Fähigkeit zur Modellierung von Hierarchien wird das State-Pattern noch um die Möglichkeit zur Abbildung von Orthogonalität (AND-Dekomposition) ergänzt. Hierzu wird

die Klasse `VirtualSuperstate` eingeführt. Diese Klasse kapselt untergeordnete Zustandsklassen vom Typ `IntermediateSuperState` und `TopSuperState`. Ebenso wie die eben genannten Klassen leitet `VirtualSuperstate` Ereignisse an ihre Subzustände weiter. Sie hat aber im Gegensatz zu diesen keinen aktuellen Zustand, da, wenn ein Zustand dieses Typs eingenommen wird, immer alle Subzustände aktiv sind. Mit der Fähigkeit zu orthogonalem Verhalten kann auch ein Broadcasting-Mechanismus realisiert werden. In diesem Fall wird in einem Zustandsübergang einfach ein Ereignis generiert und an das `Interface` übermittelt. Dieses leitet dann auf dem üblichen Weg das Ereignis weiter.
Eine History-Funktionalität kann über Verwendung der Variable `CurrentState` realisiert werden. Diese Variable wird beim Betreten eines `SuperStates` einmal gesetzt und später beim Verlassen des Zustandes nicht zurückgesetzt, so daß beim erneuten Betreten des jeweiligen Zustandes der alte Subzustand wieder zur Verfügung steht.

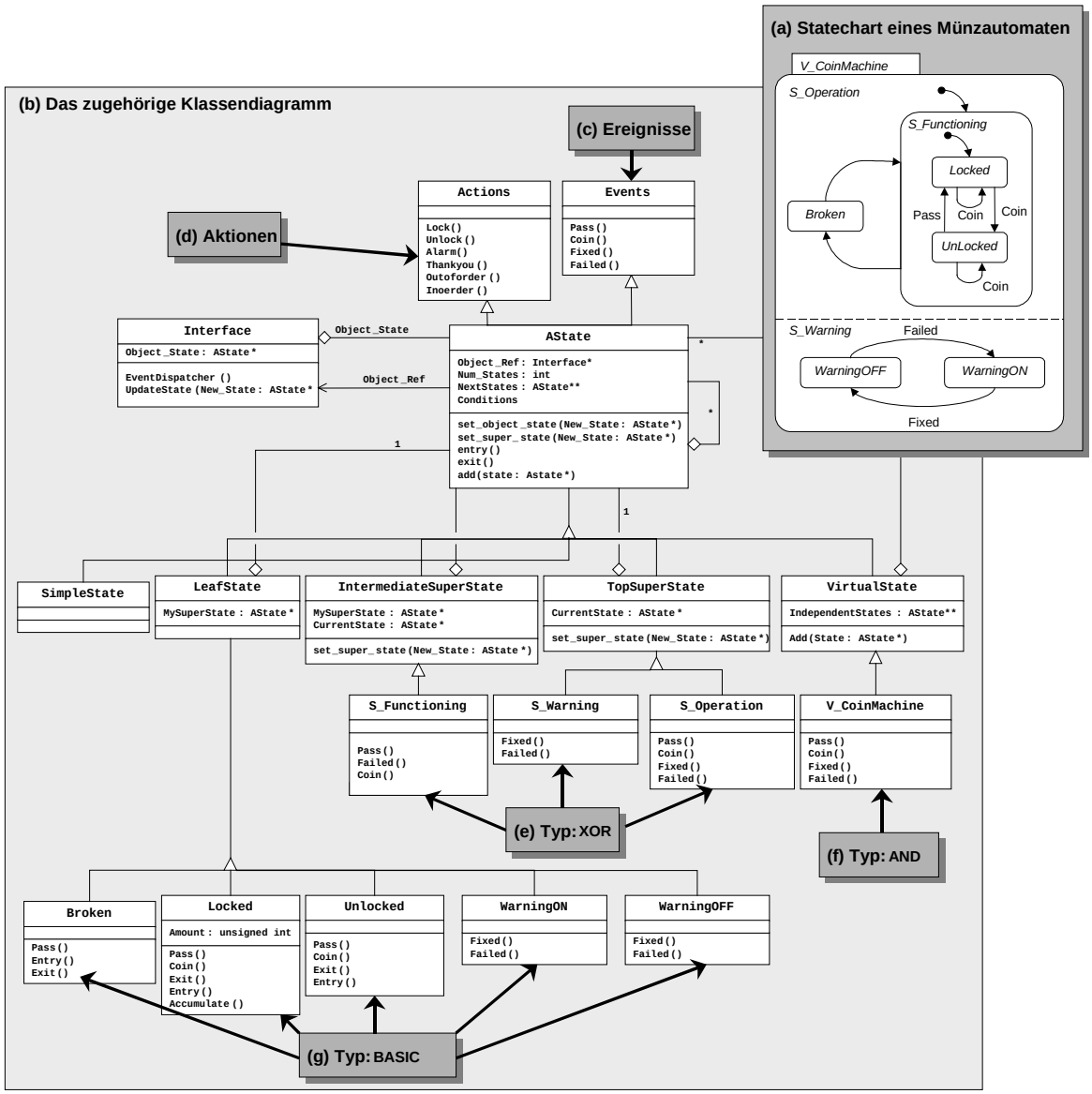

Abbildung 5.3: Verwendung des erweiterten State-Pattern[74]

[74] vgl. [Yacoub 1998]

Um dieses umfangreiche Entwurfsmuster weiter zu erläutern, wird anhand eines Beispiels die Verwendung beschrieben. Abbildung 5.3 zeigt anhand des Statecharts eines Münzautomaten[75] (a), wie ein Klassendiagramm (b) für einen konkreten Anwendungsfall aussieht. Die Ereignisse (c) und Aktionen (d) werden in den Klassen `Actions` und `Events` eingefügt. Darüber hinaus werden Klassen zur Modellierung der konkreten Zustände (e) - (g) ergänzt.

5.1.2.2 Bewertung

Mit dem erweiterten State-Pattern ist man in der Lage, ereignisgesteuerte Systeme zu bauen, in welchen sich fast alle Eigenschaften von Statecharts abbilden lassen.
Allerdings fällt auch bei diesem Entwurfsmuster auf, daß es keine konzeptionelle Trennung von Zustandsmodell und der aktuellen Zustandskonfiguration gibt. Dies führt, wie schon erläutert, zu einem schwer verständlichen Entwurf, da auf jeder Ebene unterschieden werden muß, welcher Subzustand gerade aktuell ist. Dies erschwert die Verwendbarkeit eines solchen Systems durch andere Entwickler.
Gleiches gilt für die enorme Anzahl an konkreten Zustandsklassen. Wie oben aufgeführt, gibt es auch in diesem Fall so viele konkrete Klassen, wie es Zustände gibt.
Bei den Schnittstellen, welche in `Actions` vordefiniert werden, verhält es sich genauso. Es wird immer die gesamte Breite der Schnittstelle angeboten. Die Methoden, welche gerade nicht aktiv sind, bieten nur ein Defaultverhalten an. Ein Entwickler kann nur über zusätzliche, sondierende Operationen herausfinden, welche Methoden gerade aktiv sind. Das Zustandsmodell und die Zustandsübergänge werden also nicht explizit voneinander getrennt dargestellt.
Ein weiteres Problem ist die Verwendung von Mehrfachvererbung. Im WAM-Kontext wird auf die Verwendung von Mehrfachvererbung verzichtet, da dies zu unübersichtlichen Entwürfen führt, weil oftmals nicht eindeutig ersichtlich ist, was von welcher Oberklasse geerbt wird. Werden in den Oberklassen zudem gleichnamige Bezeichner für Methoden verwendet, sind in der Regel Fehler bei der Implementierung vorprogrammiert. Hinzu kommt, daß die in dieser Arbeit verwendete Programmiersprache Java keine Mehrfachvererbung erlaubt.
Wie beim State-Pattern nach [Gamma 1996] werden auch hier zyklische Referenzen zwischen den Zustandsobjekten und dem Kontextobjekt verwendet, was den gleichen negativen Effekt bedingt. Durch den Umstand, daß Zustandsobjekte für Subzustände immer wissen müssen, welches das übergeordnete Zustandsobjekt ist, liegt auch an dieser Stelle eine zyklische Benutzt-Beziehung vor.
Bei der Modellierung von Zustandsübergängen besteht die Möglichkeit, daß für ein und dieselbe Methode in verschiedenen Zustandsklassen unterschiedliche Implementationen möglich und sogar notwendig sind. Dies hat zur Folge, daß das Wissen darüber, was bei einem Zustandsübergang passiert, auf mehrere Klassen verteilt wird. Dies erschwert die Wartung des Systems und kann zu Fehlern führen, wenn bei einer Änderung nicht das Verhalten an allen Stellen, wo der betreffende Zustandsübergang modelliert wird, angepaßt wird.
Zusammenfassend läßt sich sagen, daß das erweiterte State-Pattern über die Möglichkeit zur Modellierung aller wichtigen Eigenschaften von Statecharts verfügt. Ähnlich dem oben genannten Ansatz eignet sich dieses Entwurfsmuster nicht für eine Integration in ein Rahmenwerk, da auch hier viele konkrete Klassen benötigt werden. Das, was sich an Funktionalität in ein Rahmenwerk übernehmen ließe, ist zwar deutlich mehr als bei [Gamma

[75] aus [Yacoub 1998]

1996], aber die Unterstützung, welche durch ein Rahmenwerk erfolgen kann, wäre immer noch nur rudimentär.

5.1.3 Fazit

Zusammenfassend läßt sich sagen, daß sich beide Entwurfsmuster nicht für eine direkte Implementation im Rahmen dieser Arbeit eignen. Sie bieten zwar verwendbare Ansätze zum Bau von Zustandsautomaten, doch sie haben beide einige entscheidende Nachteile, welche dem WAM-Ansatz und der Integration in ein Rahmenwerk entgegenstehen. Aus diesem Grund wird im Folgenden, basierend auf den in Abschnitt 5.1 gewonnenen Ergebnissen, ein eigenes Entwurfsmuster für Zustandsautomaten erarbeitet.

5.2 Entwurf des Zustandsautomaten

In diesem Abschnitt soll gezeigt werden, wie der erste Entwurf eines Zustandsautomaten für das JWAM-Rahmenwerk aussieht. Zur Erläuterung des Entwurfs sollen in mehreren Schritten einzelne Eigenschaften der Statecharts aufgegriffen und modelliert werden. Nachfolgend wird zunächst die Grundidee des für den Entwurf des Zustandsautomaten erläutert

5.2.1 Grundidee für den Entwurf des Zustandsautomaten

Wie bei den oben beschriebenen Entwurfsmustern deutlich wurde, ist es problematisch, das Zustandsmodell und die aktuelle Zustandskonfiguration in einer Komponente zu modellieren. Bei diesen Entwurfsmustern wird nur über ein Attribut der Zustandsobjekte kenntlich gemacht, ob ein Zustandsobjekt zur aktuellen Konfiguration gehört. Ein weiterer Nachteil dieser Modellierung sind die zyklischen Benutzt-Beziehungen der Zustandsklassen untereinander. Diese sind bei einem solchen Ansatz allerdings notwendig, um Zustandsübergänge realisieren zu können.
Es stellt sich die Frage, wie ein Zustandsautomat beschaffen sein muß, um diese Probleme zu vermeiden.
Betrachtet man Zustände und ihre Beziehungen untereinander, stellt man fest, daß sich diese zur Laufzeit eines Systems nicht verändern. Zustände und Zustandsmodell werden zu Beginn des Lebenszyklus des Systems festgelegt und danach nicht mehr verändert oder angepaßt. Genaugenommen wird beides schon während des Entwicklungsprozesses festgelegt und bleibt auch nach der Beendigung eines Lebenszyklus des Systems erhalten. Jeder Zustand hat einen Bezeichner, welcher diesen Zustand eindeutig in der Menge der Zustände benennbar macht. Dieser Bezeichner ändert sich auch zur Laufzeit nicht. Darüber hinaus steht ein Zustand zu anderen des jeweiligen Zustandsraumes in genau festgelegten Beziehung. Wie in Abschnitt 4.3.1 erläutert, kann dies eine hierarchische oder orthogonale Beziehung sein. Diese Beziehung bleibt während der gesamten Laufzeit bestehen. Gleiches gilt für die Zustandsübergänge. Die Zustandsübergänge, welche von einem Zustand aus grundsätzlich möglich sind, sind immer gleich. Was also vorliegt, ist eine statische Beschreibung des Systems. Betrachtet man die Zustände vom theoretischen Standpunkt aus, ist ein Zustand nur ein Element aus einer Menge von Werten. Folglich können Zustände als Werte angesehen werden.
Wie eben angeführt, ändern auch Zustandsübergänge ihr Verhalten während des Lebenszyklus eines Systems nicht. Dies bedeutet, bei gleicher Ausgangslage produzieren sie immer exakt denselben Folgezustand. Sie verändern also die Konfiguration eines Systems entsprechend den in ihnen festgelegten Regeln.

So gesehen, sind Konfigurationen die einzigen Teile eines Zustandsautomaten, die während des Lebenszyklus eines Systems manipuliert werden können. Konfigurationen sind, wie in Abschnitt 4.3.1 erläutert, immer Teilmengen der Gesamtheit aller Zustände.
Faßt man die obigen Abschnitte zusammen, kann festgestellt werden, daß ein Zustandsautomat in drei Komponenten unterteilbar ist. Als erstes gibt es einen statischen Anteil, zu welchem Zustände und das Zustandsmodell gerechnet werden können. Den zweiten Teil bilden die Zustandsübergänge als verändernde Komponenten. Der letzte Teilbereich wird durch die verschiedenen möglichen Konfigurationen gebildet.
Neben der Unterteilung des Zustandsautomaten läßt sich noch ein weiterer Schluß aus den eben genannten Punkten ziehen, nämlich die Benutzt-Beziehungen der einzelnen Komponenten untereinander. Das Zustandsmodell hat Kenntnis über alle möglichen Zustände des Zustandsautomaten. Die Konfigurationen kennen den Zustandsbaum bzw. die einzelnen Zustände. Zustandsübergänge verwenden Konfigurationen. Der einzige Punkt, der noch offen bleibt und erst später beantwortet werden kann, ist die Frage nach dem Zusammenhang von Zuständen und Zustandsübergängen. Normalerweise müßten die Zustände Kenntnis über Zustandsübergänge haben. Würde man dies allerdings so umsetzen, hätte man wiederum das Problem zyklischer Benutzt-Beziehungen:
Abbildung 5.4 zeigt die Komponenten eines Zustandsautomaten und ihre Zusammenhänge in der Übersicht:

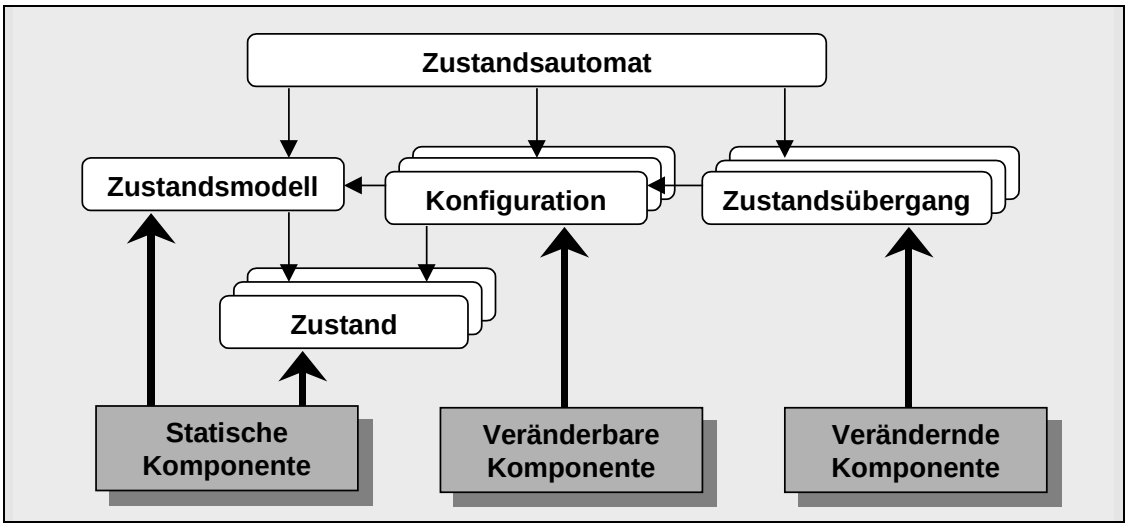

Abbildung 5.4: Komponenten eines Zustandsautomaten

Im weiteren Verlauf der Arbeit soll gezeigt werden, wie sich diese Komponenten modellieren lassen. Zunächst soll erläutert werden, wie Zustände abgebildet werden können. Aufbauend darauf sollen die Hierarchieeigenschaften von Statecharts modelliert werden. Anschließend wird gezeigt, wie Konfigurationen in das Modell integriert werden können. Mit den Zuständen und den Konfigurationen als Basis wird die Umsetzung von Zustandsübergängen geschildert. Den Abschluß bildet die Beschreibung des Zustandsautomaten.

5.2.2 Modellierung von Zuständen und Zustandshierarchien

Wie im vorigen Abschnitt beschrieben, können Zustände als Werte angesehen und modelliert werden. Das bedeutet für die Implementierung in einer objektorientierten Programmiersprache wie Java, daß Zustände durch Klassen abgebildet werden können, deren Objekte nach der Erzeugung nicht mehr verändert werden dürfen. Diese Objekte verfügen über einen eindeutigen Bezeichner, eine Beschreibung der möglichen Zustandsübergänge

und das Wissen über die Beziehung zu den anderen Zuständen. Jedes Objekt darf hierbei natürlich nur einmal im System vorhanden sein, da jeder Zustand nur ein einziges Mal im Zustandsraum vorkommt. Hierbei können aber nicht alle Zustände mit ein und derselben Klasse abgebildet werden, da es erhebliche Unterschiede in der Verwendung der Zustandstypen gibt. Die verschiedenen Typen von Zuständen wie BASIC, XOR oder AND, welche in Abschnitt 4.3.1 beschrieben wurden, müssen auf ihre Eigenschaften untersucht und zu Klassen zusammengefaßt werden. Man könnte also mit drei Klassen die gesamte Zustandshierarchie eines Statecharts abbilden. Hierdurch wird das oben geschilderte Problem der „inflationären" Vermehrung von Zustandsklassen, von denen jede nur ein Exemplar besitzt, vermieden.
Das JWAM-Rahmenwerk verfügt über das Konstrukt der Fachwerte. Mit diesem Konstrukt als Grundlage lassen sich die oben genannten Anforderungen realisieren. Im Folgenden soll erläutert werden, was Fachwerte sind und wie sie mit dem JWAM-Rahmenwerk realisiert werden können.

5.2.2.1 Fachwerte im JWAM-Rahmenwerk

Fachwerte[76] sind ein Konzept, mit dem in objektorientierten Programmiersprachen Werte abgebildet werden können.
Bei [Züllighoven 1998] und in anderen Veröffentlichungen[77] wird die Notwendigkeit zur Modellierung von Werten betont. Werte werden benötigt, um fachliche Größen zum Abzählen, Ordnen oder zur Identifikation aus einem Anwendungsbereich abbilden zu können. Bei [Züllighoven 1998] wird folgende Definition für Fachwerte gegeben:

> **DEFINITION: FACHWERT**
>
> *Ein Fachwert ist ein benutzerdefinierter Wert. Er repräsentiert Werte im Anwendungsbereich. Ein Fachwert ist ein Werttyp mit einem definierten Wertebereich und festgelegten Operationen. Seine innere Repräsentation ist verborgen.*
>
> *Fachwerte werden in objektorientierten Sprachen als Klassen definiert. Wichtig ist, daß die Exemplare eines Fachwertes immer Wertsemantik[78] besitzen, d.h. daß ein einmal gesetzter Wert nicht mehr verändert werden kann.*
>
> [Züllighoven 1998], Seite 62

Leider gibt es in Sprachen wie Java oder C++ kein Konzept für Werte. Standarddatentypen, wie zum Beispiel `int` oder `String`[79], können keine fachlichen Konzepte abbilden. Objekte können zwar fachliche Konzepte abbilden, aber sie verhalten sich nicht wie Werte. Abbildung 5.5 zeigt die grundlegenden Unterschiede zwischen Werten und Objekten.

---

[76] Im JWAM-Rahmenwerk werden Fachwerte oftmals unter ihrem englischen Namen *domain values* verwendet. Um dies in der Programmierung kenntlich zu machen, beginnen die Namen von Fachwertklassen immer mit dem Präfix „`dv`".

[77] Weitere Veröffentlichungen zu diesem Thema: [Müller 1999], [Bleek 1999c], [JWAM 2001]

[78] **Definition: Wertsemantik**
Variablen haben einen Wert und können einen anderen Wert zugewiesen bekommen und auf Gleichheit geprüft werden.
aus [Müller 1999], Seite 9

[79] Standarddatentypen der Programmiersprache Java

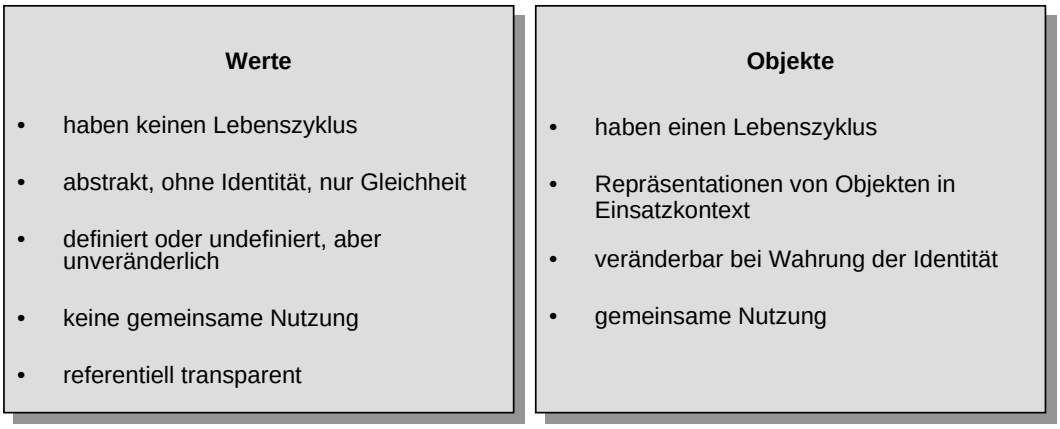

Abbildung 5.5: Unterschiede zwischen Werten und Objekten[80]

Aus diesem Grund wird im JWAM-Rahmenwerk unter Verwendung des *Flyweight/Factory-Pattern* (siehe [Gamma 1996], Seite 87ff, Seite 107ff und Seite 195ff) ein Konzept angeboten, welches es erlaubt, mit den Mitteln objektorientierter Sprachen Werte zu modellieren. Mit diesem Konzept kann Wertsemantik garantiert werden. Hierfür wird der Ansatz der unveränderlichen Objekte verwendet. Bei diesem Ansatz werden Wertobjekte nur einmal bei ihrer Erzeugung gesetzt. Um den bei einem solchen Vorgehen normalerweise auftretenden hohen Speicherverbrauch zu vermeiden, wird das oben genannte Entwurfsmuster verwendet. Auf diese Weise werden bei den Fachwertklassen die Konstruktoren für andere Klassen verborgen, so daß sie nur noch innerhalb der Klasse bekannt sind. Die *Factory* (in diesem Zusammenhang auch als Fachwertfabrik bezeichnet), welche als *Singleton*[81] realisiert wird, übernimmt die Erzeugung von Fachwertobjekten. Bei der Erzeugung wird von der Factory geprüft, ob ein Wert bereits vorliegt. Ist dies nicht der Fall, wird das entsprechende Wertobjekt neu erzeugt. Im anderen Fall wird eine Referenz auf das schon erzeugte Objekt zurückgeliefert. So kann zugesichert werden, daß jeder Wert nur einmal erzeugt wird und es somit nur eine minimale Anzahl von Objekten einer Fachwertklasse gibt. Bevor ein Wert erzeugt wird, wird die vorliegende Repräsentation eines Fachwertes an der Fabrik über geeignete Methoden auf ihre Gültigkeit geprüft.
Technisch wird dies so gelöst, daß die Factory als *inner class*[82] der jeweiligen Fachwertklasse realisiert wird, damit Factory und Wertobjekt eine konstruktive Einheit bilden. Dies erleichtert die Entwicklung und Wartung erheblich. Im Rahmenwerk wird mit dem Interface `DomainValue` und der Klasse `DomainValueImpl` eine grundlegende Implementation für Fachwerte angeboten. welche sich über Vererbung zu ihren jeweiligen Fachwertklassen erweitern lassen.
Nähere Informationen zur Implementation der Fachwerte sind in der Spezifikation des JWAM-Rahmenwerks (siehe [JWAM 2001]) zu finden.

[80] aus [Müller 1999], Seite 8

[81] *Singleton-Pattern* (siehe [Gamma 1996], Seite 157ff): Dieses Entwurfsmuster garantiert, daß es von einer Klasse nur ein Objekt gibt. Um dies zu erreichen, wird der eigentliche Konstruktor der Klasse verborgen und ist nur über eine statische Erzeugungsmethode der Klasse zugreifbar. Diese Methode sorgt dafür, daß nur ein Exemplar erzeugt wird. Liegt bereits ein Exemplar dieser Klasse vor, wird eine Referenz auf das bereits erzeugte Objekt zurückgeliefert.

[82] In der Programmiersprache Java ist es seit Version 1.1 möglich, eine Klasse in einer anderen Klasse als sogenannte *inner class* zu definieren.
siehe [Flanagan 1997], Seite 110-112

5.2.2.2 Implementation

Wie eben ausgeführt wurde, eignen sich Fachwerte gut, um die Zustandsklassen mit einer minimalen Anzahl von Objekten bei Wahrung des initialen Objektzustandes zu modellieren. Im weiteren Verlauf soll erläutert werden, wie die einzelnen Fachwertklassen aufgebaut sein müssen, um die Hierarchieeigenschaften von Statecharts abzubilden. Abbildung 5.6 zeigt den in Kapitel 4.5 erstellten Statechart zur Containervoranmeldung als Zustandsbaum. Diese Struktur muß nun in eine Softwarekomponente überführt werden.

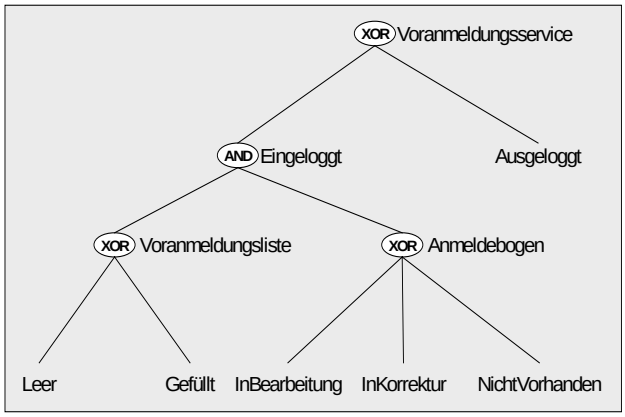

Abbildung 5.6: Zustandsbaum der Containervoranmeldung

In Abbildung 5.6 wird nochmals deutlich, daß es drei Typen von Zuständen gibt. Zunächst sollen gemeinsame Eigenschaften der einzelnen Typen beschrieben werden, welche sich für die Realisierung in einer gemeinsamen Basisklasse eignen. Diese Basisklasse wird als `dvState` bezeichnet. Da die Grundlage dieses Entwurfs eine Implementation unter Zuhilfenahme von Fachwerten ist, erbt die Basisklasse von `DomainValue`. Da diese Basisklasse nur der Bereitstellung gemeinsamer Eigenschaften dienen soll und kein eigenständiges Konzept der Statecharts beschreibt, wird sie als abstrakt deklariert. Dies bedeutet, daß kein Objekt dieser Klasse direkt instanziiert werden kann.

5.2.2.2.1 Die Basisklasse

Wie schon im vorangegangenen Abschnitt ausgeführt, verfügt jeder Zustand über einen ihn eindeutig bezeichnenden Namen. Dies ist eine Eigenschaft, welche alle Zustände im Zustandsbaum gemeinsam haben. Daneben können von allen Zuständen aus Zustandsübergänge ausgeführt werden. Auf welche Weise dies umzusetzen ist, wird nachfolgend bei der Modellierung der Zustandsübergänge gezeigt. Es soll nur festgehalten werden, daß auch dies eine gemeinsame Eigenschaft ist.
Die Schnittstelle[83] der Klasse `dvState` sieht folgendermaßen aus:

```
public abstract class dvState extends DomainValueImpl
{
  public String getName()
   {...}

  protected String      _name;

  // Factory inner class
  public abstract static class Factory extends DomainValueImpl.Factory
  {
    protected static boolean isNameValid (String name)
     {...}
  }
}
```

[83] Es wird bei diesem und auch bei den nachfolgenden Code-Auszügen, keine vollständige Schnittstellenbeschreibung der Fachwertklassen gegeben, um die Darstellung übersichtlicher zu gestalten. Hierzu werden einige Standardmethoden der Fachwerte weggelassen und nur die fachlich relevanten Methoden beschrieben.

Sie umfaßt zum einen die Fachwertfabrik `Factory`. Diese inner class übernimmt bei Fachwerten normalerweise die Erzeugung und die Gültigkeitsprüfungen. Da die vorliegende Klasse abstrakt ist, enthält sie keine Methoden zur Erzeugung von Objekten. Allerdings kann hier schon eine Gültigkeitsprüfung (`isNameValid`) für die Bezeichner der Zustände definiert werden. Die eigentliche Klasse `dvState` enthält ebenfalls keinen Konstruktor, da sie als abstrakt deklariert ist. Sie verfügt nur über einen Bezeichner `_name`[84], welcher die Bezeichnung eines Zustandes beinhaltet und die zugehörige Ausgabemethode `getName`. Methoden und Attribute für die Umsetzung von Zustandsübergängen werden später noch ergänzt.
Nach der Modellierung der gemeinsamen Eigenschaften der Zustandstypen, werden im Folgenden die einzelnen Typen betrachtet.

5.2.2.2.2 Der Basiszustand

Die Eigenschaften der Zustände vom Typ BASIC sind mit der Basisklasse fast vollständig modelliert. Es muß nur eine instanziierbare Klasse (`dvBasicState`) mit einem Konstruktor und den entsprechenden Methoden an der Fachwertfabrik eingeführt werden. Das nachfolgend aufgeführte Code-Beispiel zeigt einen Ausschnitt der Schnittstelle von `dvBasicState`:

```
public class dvBasicState extends dvState
{
   protected dvBasicState (String name, dvBasicState.Factory
        creatingFactory)
    {...}

 // Factory inner class
   public static class Factory extends dvState.Factory
   {
     public static boolean isValidValue (String name)
      {...}
     public static dvBasicState valueOf (String name)
      {...}
   }
}
```

Gegenüber der Basisklasse ist die hier gezeigte Klasse instanziierbar und verfügt über einen entsprechenden Konstruktor. Die zugehörige Fachwertfabrik ist um die Methode `valueOf`, zur Erzeugung von Fachwertobjekten, und um die Methode `isValidValue`, zur Validierung der übergebenen Repräsentation des Fachwertes, ergänzt worden.

5.2.2.2.3 Die Basisklasse für hierarchische Zustandstypen

Die Typen AND und XOR verfügen beide über Subzustände, welche von einem beliebigen Zustandstyp sein können. Allerdings ist der Umgang mit den Zuständen unterschiedlich. Aus diesem Grund kann als Erweiterung der bisherigen Basisklasse eine zusätzliche abstrakte Klasse namens `dvCompositeState` definiert werden, von der AND- und XOR-Zustandsklassen erben.

```
public abstract class dvCompositeState extends State
{
  public dvState[] getSubstates()
   {...}
  public boolean hasState(String name)
   {...}
  public boolean hasState(dvState s)
   {...}
```

[84] Im Styleguide des JWAM-Rahmenwerks gilt die Konvention, daß die Benennung von Klassenvariablen dem folgenden Schema folgt: „`_variable`“

```
  public boolean hasSubstate(String name)
   {...}
  public boolean hasSubstate(dvState s)
   {...}
  public State getSubstate(String name)
   {...}

  protected Hashtable     _substates;

  // Factory inner class
  public abstract static class Factory extends dvState.Factory
  {
    protected static boolean areSubStatesValid(dvState[] substates)
    {...}
  }
}
```

Die Klasse hat als zusätzliche Exemplarvariable eine Liste von Subzuständen (`_substates`). Die Funktionalität der Fabrik wird um die Methode `AreSubStatesValid` erweitert. Diese prüft, ob eine Menge von Zuständen gültig ist. Das heißt, es wird geprüft, ob alle Zustände definiert sind und ob die Anzahl der Subzustände mindestens eins beträgt. Daneben erhält die Klasse noch einige sondierende Methoden, mit welchen zum Beispiel nach Subzuständen gesucht werden kann.
Die Methode `getSubstates` liefert die Subzustände als Array zurück, während `getState` ein bestimmtes, über den Bezeichner spezifiziertes Zustandsobjekt zum Ergebnis hat. Dies ist möglich, da sichergestellt werden kann, daß Bezeichner eindeutig sind. Es kann auch über einen Bezeichner nach einem Zustand gesucht werden. Die beiden Varianten von `hasState` durchsuchen die Liste der direkt untergeordneten Subzustände nach einem Zustand, während `hasSubstate` rekursiv den gesamten untergeordneten Teil des Zustandsbaumes absucht. Die Methode `getSubstate` durchsucht den untergeordneten Zustandsbaum nach einem Zustand mit der Bezeichnung `name` und gibt diesen als Ergebnis zurück.
Betrachtet man nun die Zustandstypen XOR und AND, so unterscheiden sich beide in der Eigenschaft, daß, wenn ein Zustand des Typs XOR betreten wird, nur ein Subzustand aktiv sein darf, während bei orthogonalen Zuständen immer alle direkt untergeordneten Subzustände aktiv sind. Des weiteren haben Zustände vom Typ XOR die Eigenschaft, daß einer ihrer Subzustände als Defaultzustand definiert werden kann. Das bedeutet, dieser Zustand ist beim Betreten des Zustandes immer aktiv, sofern kein anderer Subzustand direkt angegeben wird.

5.2.2.2.4 Der Zustandstyp AND

Bildet man die im vorigen Abschnitt geschilderten Punkte auf Klassen ab, so erhält man für den Typ AND die folgende Schnittstelle:

```
public class dvOrthogonalState extends dvCompositeState
{
  protected dvOrthogonalState(  String                    name,
                                dvState[]                 substates,
                                dvOrthogonalState.Factory creatingFactory)
   {...}

  // Factory inner class
  public static class Factory extends dvCompositeState.Factory
  {
    public static dvOrthogonalState valueOf(String name, dvState[]substates)
     {...}
    public static boolean isValidValue (String name, dvState[] substates)
     {...}
   }
}
```

Die Klasse `dvOrthogonalState` unterscheidet sich insofern von ihrer Basisklasse, als daß es sich hierbei um keine abstrakte Klasse handelt. Das bedeutet, an der Fachwertfabrik sind Methoden zur Erzeugung von Fachwertobjekten (`valueOf`) und zur Validierung (`isValidValue`) vorhanden. Darüber hinaus gibt es auch einen Konstruktor zur eigentlichen Erzeugung von Objekten.

5.2.2.2.5 Der Zustandstyp XOR

Die den Typ XOR modellierende Klasse `dvExclusiveState` hat die folgende Schnittstelle:

```
public class dvExclusiveState extends dvCompositeState
{
  public dvState getDefaultSubstate()
   {...}
  protected dvExclusiveState(String                name,
                             dvState[]             substates,
                             vdState               defaultState,
                             ExclusiveState.Factory creatingFactory)
  {...}

  protected dvState      _defaultState;

  // Factory inner class
  public static class Factory extends dvCompositeState.Factory
  {
    public static dvExclusiveState valueOf (String    name,
                                            dvState[] substates,
                                            dvState   defaultState)
     {...}
    public static boolean isValidValue (String    name,
                                        dvState[] substates,
                                        dvState   defaultState)
    {...}
   }
}
```

Die Klasse `dvExclusiveState` verfügt zusätzlich zu Konstruktor und den Methoden an der Fachwertfabrik noch über die Exemplarvariable `_defaultState`. Diese gibt an, welcher der Subzustände defaultmäßig verwendet werden kann. Nach den in Abschnitt 4.2.1 vorgestellten Verwendungsmöglichkeiten von Defaultzuständen muß dieser Zustand zwar nicht vorhanden sein, aber es hat sich gezeigt, daß es die Verständlichkeit der Statechart-Diagramme erhöht, wenn ein solcher Zustand immer gesetzt ist. Außerdem wird die Bestimmung von initialen Zustandskonfigurationen erheblich erleichtert.

5.2.2.2.6 Zusammenhang

Um den Zusammenhang zwischen der Basisklasse für alle Zustandstypen `dvState`, der Basisklasse für hierarchische Zustandstypen `dvCompositeState`, der Klasse für Basiszustände `dvState`, der Klasse für XOR-Zustände `dvExclusiveState` und der Klasse für AND-Zustände `dvOrthogonalState` deutlich zu machen, werden in der nachfolgenden Abbildung 5.7 die Beziehungen zwischen den Klassen in einem Klassendiagramm dargestellt:

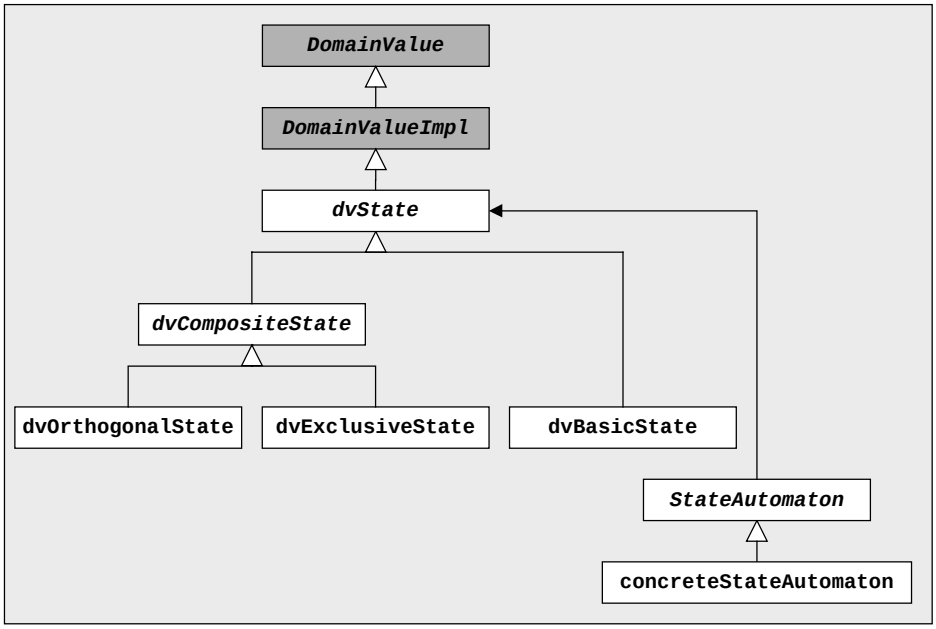

Abbildung 5.7: Abbildung der Zustandstypen BASIC (`dvBasicState`), AND (`dvOrthogonalState`) und XOR (`dvExclusiveState`)[85]

5.2.3 Modellierung von Konfigurationen

Nachdem im vorangegangenen Abschnitt der Zustandsbaum modelliert wurde, muß nun eine Möglichkeit gefunden werden, Konfigurationen abzubilden. Aus diesem Grund muß, wie in Abbildung 5.8 dargestellt, die Möglichkeit geschaffen werden, solche Ausschnitte eines Zustandsbaumes in einer Komponente des Entwurfs zu modellieren.

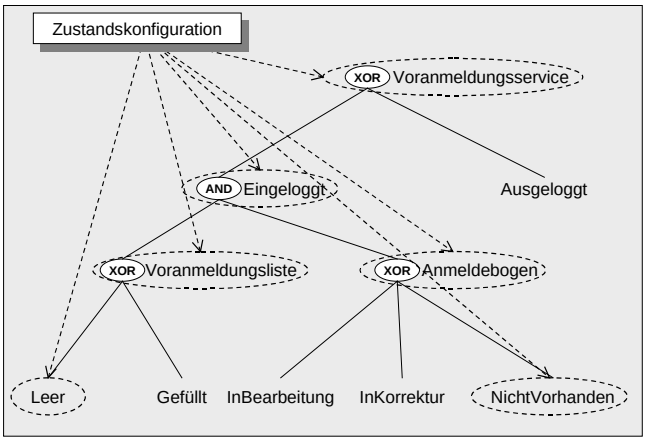

Abbildung 5.8: Beispiel für eine Zustandskonfiguration

Konfigurationen sind, wie in Abschnitt 4.3.1 ausgeführt, ein Teil der Gesamtmenge von Zuständen eines Systems. Konfigurationen werden nach den in ihnen enthaltenen Elementen in verschiedene Arten von Konfigurationen unterteilt. Dies können zum Beispiel Basiskonfigurationen, vervollständigte Konfigurationen oder partielle Konfigurationen sein. Für diesen Entwurf sind partielle und vervollständigte Konfigurationen von besonderem Interesse.

Vervollständigte Konfigurationen bieten die Möglichkeit zu prüfen, ob sich ein System in einer syntaktisch korrekten Zustandskonfiguration befindet. Vervollständigte Konfigurationen enthalten alle Elemente der Zustandsmenge, welche zu einem bestimmten Zustand gehören. Sie enthalten also neben den Basiszuständen auch die entsprechenden Zustände der höheren Ebenen. In Abbildung 5.8 wird eine solche Konfiguration dargestellt. Eine Kon-

[85] Klassen aus dem JWAM-Rahmenwerk oder aus Java-Klassenbibliotheken werden grau unterlegt, um deutlich zu machen, daß sie nicht Ergebnis dieses Entwurfs sind.

figuration kann einfach auf ihre Vollständigkeit getestet werden, indem zunächst geprüft wird, ob der Zustand ROOT, sprich der oberste Knoten des Zustandsbaumes, enthalten ist. Ist er nicht enthalten, ist die Konfiguration nicht vollständig. Nun wird ausgehend vom ROOT-Zustand geprüft, ob zu jedem Zustand der Typen XOR und AND auch die entsprechenden Subzustände vorhanden sind. In diesem Fall bedeutet dies, wenn ROOT vom Typ XOR ist, so muß genau einer seiner Subzustände in der Konfiguration enthalten sein. Sind mehrere enthalten, ist die Konfiguaration syntaktisch nicht korrekt. Ist ROOT vom Typ AND, müssen alle seine Subzustände enthalten sein. Für die enthaltenen Subzustände muß, wenn sie vom Typ XOR oder AND sind, die eben geschilderte Überprüfung ebenfalls erfolgen. Handelt es sich bei einem Element um einen Basiszustand, ist keine Überprüfung notwendig. Die Überprüfung ist beendet, sobald alle Elemente zugeordnet werden konnten. Kann ein Element nicht zugeordnet werden, ist die Konfiguration ebenfalls nicht vollständig.
Dies ist allerdings nur eine syntaktische Prüfung. Semantische Korrektheit kann nur über eine fachlich korrekte Modellierung eines Systems durch einen Statechart sichergestellt werden. Ist der erstellte Statechart fachlich fehlerhaft, so nützt auch diese Syntaxprüfung nur insofern, als daß Rückschlüsse auf ein fehlerhaft modelliertes Teilsystem gezogen werden können.
Gerade bei großen Zustandsbäumen ist es mühsam, immer mit der vervollständigten Konfiguration umzugehen, wenn nur ein Teil des Zustandsbaumes oder auch nur ein einziger Zustand von Interesse ist. Hierfür lassen sich partielle Konfigurationen verwenden, da diese auch nur Teile des Zustandsbaumes enthalten können. Natürlich müssen auch diese syntaktisch korrekt sein.
Im Folgenden wird erläutert, wie sich die oben geschilderten Eigenschaften von Konfigurationen geeignet modellieren lassen. Da es sich bei Konfigurationen um Mengen handelt, muß zunächst eine Möglichkeit gefunden werden, diese abzubilden. Die Programmiersprache Java bietet zwar eine Implementation für Mengen an, aber diese eignen sich für diesen Zweck nicht, da sie keine Wertsemantik bieten können. Diese ist aber erforderlich, da durch Konfigurationen nicht nur dynamische Informationen über ein System ausgedrückt werden, sondern sie auch dazu dienen, statische Zusammenhänge zu beschreiben. Es muß also sichergestellt werden, daß eine von einem Zustandsübergang[86] referenzierte Konfiguration nicht verändert wird. Eine solche Konfiguration enthält statische Informationen darüber, wie ein System vor und nach der Zustandsänderung konfiguriert sein muß. Würde eine solche Konfiguration zur Laufzeit verändert, wäre völlig offen, welche Konfiguration ein System nach einem Zustandsübergang eingenommen hätte. Es könnte nicht einmal sichergestellt werden, daß ein Zustandsübergang aus einer gültigen Konfiguration heraus möglich ist. Konfigurationen dürfen also nach der Erzeugung nicht verändert werden. An dieser Stelle könnte mit Kopien gearbeitet werden, welche dann jeweils entsprechend verändert werden. Dies wäre zwar eine Möglichkeit zur Lösung des Problems, aber ein solches Vorgehen kann zu einem enormen Mehrverbrauch an Systemressourcen führen. Eine bessere Lösung wäre die Modellierung von Konfigurationen mittels Fachwerten. Diese bieten neben referentieller Transparenz auch einen minimalen Verbrauch an Systemressourcen. Die Elemente eines solchen Konfigurationsobjektes sind wiederum Fachwerte, so daß auch hier keine Verletzung des Prinzips der Wertsemantik auftreten kann.

[86] Der Zusammenhang von Konfigurationen und Zustandsübergängen wird bei der Modellierung der Zustandsübergänge erläutert werden.

5.2.3.1 Implementation

Die Klasse `dvConfiguration` modelliert Konfigurationen als Fachwerte. Nachfolgend wird die Schnittstelle dieser Klasse gezeigt:

```
public class dvConfiguration extends DomainValueImpl
{
  public dvConfiguration add(dvConfiguration set);
   {...}
  public dvConfiguration add(dvState[] values)
   {...}
  public dvConfiguration add(dvState value)
   {...}
  public dvConfiguration remove(dvConfiguration set)
   {...}
  public dvConfiguration remove(dvState[] values)
   {...}
  public dvConfiguration remove(dvState value)
   {...}
  public boolean contains(dvState value)
   {...}
  public boolean contains(String value)
   {...}
  public boolean contains(dvState[] values)
   {...}
  public boolean contains(dvConfiguration values)
   {...}
  public boolean isEmpty()
    {...}
  public boolean isComplete(dvState root)
    {...}
  protected dvConfiguration( dvState[]                states,
                             dvConfiguration.Factory  creatingFactory)
   {...}

  private Set _States;

  // Factory inner class
  public static class Factory extends DomainValueImpl.Factory
  {
   public DomainValue createEmptySet ()
    {...}
   public static dvConfiguration valueOf (dvState[] States)
    {...}
   public static boolean isValidValue (dvState[] States)
    {...}
  }
}
```

Da Mengen Zusammenfassungen bestimmter, wohlunterschiedener Elemente ohne Wiederholung[87] sind, muß bei der Erzeugung der Fachwertobjekte vom Typ `dvConfiguration` und bei der Prüfung auf Gültigkeit darauf geachtet werden, daß jeder Wert nur einmal in der Liste der Werte vorkommt. Darüber hinaus muß geprüft werden, ob die übergebene Zustandsmenge syntaktisch korrekt ist. Das heißt, es darf nur ein Subzustand eines Zustands vom Typ XOR in der Menge der übergebenen Zustände vorhanden sein, da die Konfiguration sonst syntaktisch nicht korrekt aufgebaut wäre. Gleiches gilt natürlich auch für Methoden, deren Ergebnis ein neuer Fachwert des Typs `dvConfiguration` ist.
Neben den erzeugenden und validierenden Methoden an der Fachwertfabrik und dem Konstruktor stehen Methoden zu den verschiedenen Operationen wie zur Bildung von Vereinigungsmenge (`add`) oder Differenzmenge (`remove`) mit verschiedenen Argumentlisten zur

[87] vgl. [Hopcraft 1994]

Verfügung. Des weiteren kann geprüft werden, ob ein Objekt, eine Konfiguration oder eine Menge von Zuständen in der betrachteten Konfiguration enthalten ist.
Die Methode `isComplete` prüft, ob die vorliegende Konfiguration vollständig ist. Als Parameter wird der ROOT-Zustand übergeben. Gemäß dem oben erläuterten Schema werden dann Elemente der Konfiguration überprüft.
Im Zustandsautomaten werden die Konfigurationen dazu verwendet, sich den aktuellen Systemzustand zu merken. Hierzu erhält der Zustandsautomat eine Exemplarvariable namens `_currentConfiguration`.
Eine weitere Frage im Zusammenhang mit der Verwendung von Konfigurationen im Zustandsautomaten ist die Bestimmung der Anfangskonfiguration eines Systems. Die naheliegendste Lösung wäre, eine Konfiguration im Zustandsautomaten explizit zu erzeugen. Daneben gibt es die Möglichkeit, aus dem Zustandsbaum eine Konfiguration abzuleiten. Dies ist allerdings nur möglich, wenn alle Zustände des Typs XOR einen Defaultzustand haben. Hierfür wird die Klasse `dvCompositeState` um die abstrakte Methode `getInitialConfiguration` ergänzt. In den Subklassen `dvOrthogonalState` und `dvExclusiveState` wird diese Methode entsprechend implementiert. In der Klasse `dvOrthogonalState` werden alle Subzustände und, wenn vorhanden, deren Subzustände als initiale Konfiguration zurückgeliefert. In der Klasse `dvExclusiveState` hingegen werden nur der Defaultzustand und dessen Subzustände der initialen Konfiguration hinzugefügt.
Der konkrete Zustandsautomat ruft an `_root` die Methode `GetInitialConfiguration` auf und erhält als Ergebnis die initiale Konfiguration. Hierdurch entfällt der explizite Aufbau der initialen Konfiguration im Zustandsautomaten.
Abbildung 5.9 zeigt, wie sich Konfigurationen in das Klassendiagramm einordnen:

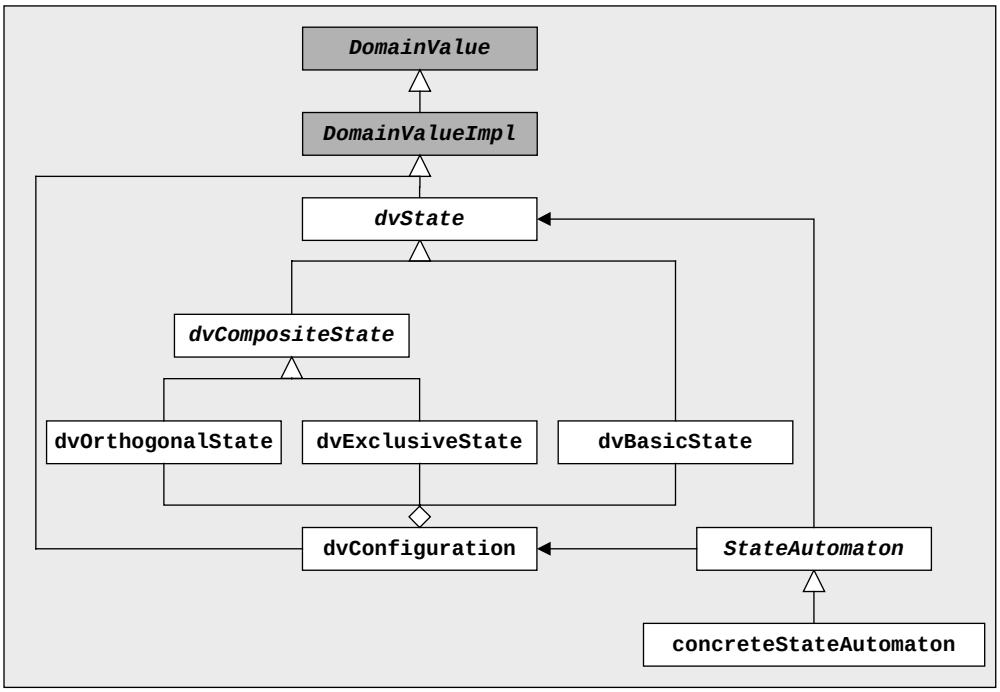

Abbildung 5.9: Modellierung von Konfigurationen (`dvConfiguration`)

5.2.4 Modellierung von Zustandsübergängen

Nachdem Zustandsbaum und Konfigurationen modelliert worden sind, müssen nun Zustandsübergänge und Ereignisse in den Entwurf integriert werden.

Zustandsübergänge, wie sie in Abschnitt 4.2.1 beschrieben wurden, besitzen eine Quell- und eine Zielmenge von Zuständen. Das heißt, die Quellmenge ist die (Teil-)Konfiguration des Systems vor der Zustandsänderung. Das System muß sich in dieser Konfiguration befinden, damit der Zustandsübergang stattfinden kann. Die Zielmenge ist die Konfiguration des Systems nach dem Schalten. Auslöser für einen Zustandsübergang ist ein Ereignis. Wird ein Ereignis ausgelöst und ist das System in einem Zustand, in dem es einen Zustandsübergang durchführen kann, wird die Konfiguration des Systems geändert. Neben dieser Zustandsänderung können Aktionen ausgeführt werden, bei denen es sich entweder um das Auslösen von Ereignissen, das Starten oder Stoppen von Aktivitäten oder um die Manipulation von Variablen handelt. Orthogonale Zustandsübergänge werden hierbei „sofort“[88] ausgeführt. Zustandsübergänge bzw. Folgen von Zustandsübergängen sind ununterbrechbar. Wenn also ein Ereignis eintritt, wird der zugehörige Zustandsübergang mit allen orthogonalen Übergängen abgearbeitet. Erst dann kann auf das nächste Ereignis reagiert werden.
Nach dieser Beschreibung von Zustandsübergängen stellt sich die Frage, wie Zustandsübergänge für den vorliegenden Zustandsautomaten modelliert werden sollen.
Betrachtet man die eben genannte Beschreibung von Zustandsübergängen, lassen sich diese als Funktionen ansehen. Dies bedeutet, daß sie einen Satz von Argumenten erhalten und daraus, anhand der in ihnen festgelegten Regeln, eine Ausgabe produzieren. Daraus folgt, daß Zustandsübergänge nichts über den Zustandsautomaten wissen müssen. Sie erhalten vom Zustandsautomaten eine Eingabe und produzieren eine Ausgabe, die dann vom Zustandsautomaten entsprechend ausgewertet werden kann.
Ein Zustandsübergang braucht die aktuelle Zustandskonfiguration und das auslösende Ereignis als Eingaben. Darüber hinaus werden aber noch die Materialien, Fachwerte und Standarddatentypen benötigt, auf denen oder mit denen fachliche Operationen ausgeführt werden sollen. Als Ausgabe erhält der Zustandsautomat eine veränderte Zustandskonfiguration, die dann als aktuelle Zustandskonfiguration übernommen werden muß und einen veränderten Satz an Materialien.
Für den Entwurf folgt daraus, daß der Zustandsautomat zusätzlich zu seiner Konfiguration über Ereignisse und Materialien verfügen muß. Ereignisse erhält der Zustandsautomat von außen. Er muß dann dafür sorgen, daß, sobald ein Ereignis registriert wird, der zugehörige Zustandsübergang angestoßen wird. Der Zustandsautomat hat dafür Sorge zu tragen, daß Zustandsübergänge nur dann ausgeführt werden, wenn dies auch laut Statechart möglich ist. Ereignisse und Zustandsübergänge können also getrennt modelliert werden.
Nachdem erläutert wurde, was ein Zustandsübergang als Eingaben benötigt, wird im Folgenden geklärt, wie es zu einer Ausgabe kommt.
Nachdem ein Zustandsübergang seine Eingaben erhalten hat, muß zunächst geprüft werden, ob alle Vorbedingungen erfüllt sind. Es muß hierbei sichergestellt werden, ob alle für den Zustandsübergang notwendigen Materialien oder Werte vorliegen. Ist dies nicht der Fall, wird so verfahren, daß der Quellzustand auch wieder der Zielzustand ist. Die Materialien werden dann unverändert zurückgeliefert. Ob eine Konfiguration vorliegt, aus der der betreffende Zustandsübergang aufgerufen werden darf, wird nicht geprüft, da der Zustandsautomat sowieso nur Zustandsübergänge anstoßen darf, die sich aus der aktuellen Konfiguration ableiten.
Sind alle Bedingungen erfüllt worden, können Aktionen ausgeführt werden. An dieser Stelle werden nun die Fachlichen Services verwendet. Diesen werden die entsprechenden Materialien und Werte übergeben. In diesen Fachlichen Services werden nun Operationen auf den Materialien ausgeführt. Um ihnen umgehen zu können, muß in den jeweiligen Zu-

[88] nach Semantik der Microsteps (vgl. Abschnitt 4.2.2 **Semantik**)

standsübergängen das Wissen um diese Services und ihre Schnittstellen vorhanden sein. Abhängig von den Ergebnissen der Aktionen und dem Prüfen der Bedingungen wird dann die Zielkonfiguration bestimmt.
Nun gibt es noch den Fall, daß bei Zustandsübergängen Ereignisse generiert werden können, um Zustandsübergänge in einer orthogonalen Komponente auszulösen. In diesem Fall wird, anstatt ein Ereignis zu generieren, welches dann vom Zustandsautomaten erst wieder verarbeitet werden muß, der betreffende orthogonale Zustandsübergang direkt verwendet. Dieser erhält die veränderte Konfiguration und den veränderten Materialienpool des auslösenden Zustandsüberganges. Hieraus generiert er seine Ausgabe, die an den ersten Zustandsübergang übergeben und von diesem an den Zustandsautomaten weitergereicht wird. Bei dieser Art der Modellierung von Zustandsübergängen ist es von enormer Wichtigkeit, die im Abschnitt 4.3.2 formulierte Semantik einzuhalten. Ansonsten könnte es an dieser Stelle zu Mehrfachzuweisungen oder zyklischen Zustandsübergängen kommen. Die Abbildung 5.11 zeigt eine schematische Darstellung eines solchen Zustandsüberganges:

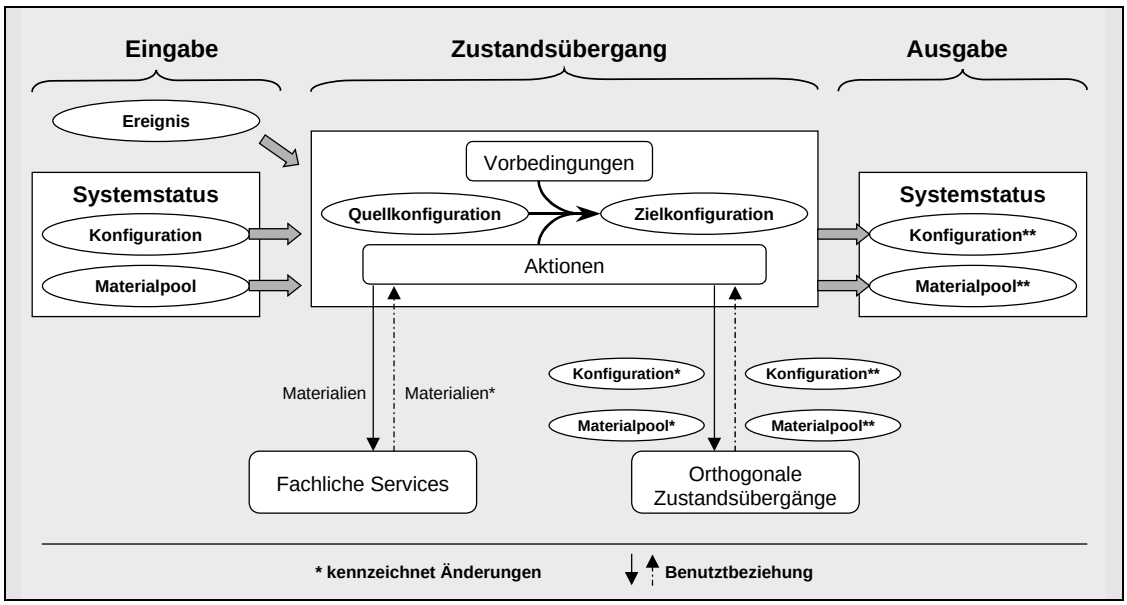

Abbildung 5.11: Schematische Darstellung eines Zustandsübergangs

Im Zusammenhang mit Zustandsübergängen gibt es noch den Fall, daß ein Ereignis am Zustandsautomaten registriert wird, welches nicht zum fachlichen Systemzustand paßt. In dem hier vorliegenden Beispiel der Containervoranmeldung kann dies passieren, falls ein Anwender in einem Browser unter Ausnutzung des Caching-Mechanismus einige Seiten „zurück geht“ und dort ein Ereignis auslöst. In diesem Fall verbleibt der Zustandsautomat in seiner aktuellen Konfiguration. Er muß allerdings über seinen Ergebniswert nach außen bekannt machen, daß er nicht schalten konnte. Diese Information muß von dem den Zustandsautomaten verwendenden Objekt ausgewertet werden.
Nachdem die Frage geklärt wurde, was einen Zustandsübergang ausmacht und was er leistet, soll nun erläutert werden, wie sich dieses im Entwurf abbilden läßt und gleichzeitig die bei den oben genannten Entwurfsmustern auftretenden Probleme vermieden werden können.
Ein gravierender Nachteil der in Abschnitt 5.1 geschilderten Entwurfsmuster ist, daß das Wissen über Zustandsübergänge auf viele Klassen verteilt ist. Zudem kann es durch unüberlegtes Überschreiben von Methoden zu fachlichen Inkonsistenzen kommen. Um dieses Problem zu lösen, wird für jeden Zustandsübergang eine Klasse geschaffen. In dieser Klas-

se wird das gesamte Wissen über alle durch diesen Zustandsübergang durchgeführten Zustandsänderungen gekapselt. Dies hat allerdings den unvermeidbaren Nachteil, daß die Zahl der Klassen erheblich zunimmt. Um aber die Zahl der instanziierbaren Objekte zu begrenzen, können diese Klassen als Singletons[89] implementiert werden. Dies ist möglich, da Zustandsübergänge Funktionen sind. Funktionen verhalten sich immer gleich. Wird dieselbe Funktion mehrmals mit der gleichen Parameterliste aufgerufen, so ist das Ergebnis immer gleich. Folglich ist es zwingend, daß alles, was zur Durchführung des Zustandsüberganges notwendig ist, beim Aufruf als Parameter übergeben werden muß. Allerdings kann man sich hier auf sich dynamisch verändernde Informationen, wie die aktuelle Konfiguration des Zustandsautomaten, beschränken. Statische Elemente, wie zum Beispiel der Zustandsbaum sowie die Quell- und Zielmenge, können schon bei der Erzeugung der Singleton-Objekte übergeben werden.

5.2.4.1 Implementation

Nachdem das Konzept von Zustandsübergängen erläutert wurde, werden im Folgenden die für eine entsprechende Umsetzung notwendigen Klassen beschrieben.

5.2.4.1.1 Ereignisse und Ereignismengen

Ereignisse werden durch die Fachwertklasse `dvEvent` abgebildet. Da dies als ein sehr einfacher Fachwert ist, welcher nur einen Bezeichner, die entsprechenden Zugriffsoperationen und Konstruktoren enthält, wird die zugehörige Klasse nicht weiter erläutert. Um mit Mengen von Ereignissen besser umgehen zu können, wird der Fachwert `dvEventSet` eingeführt. Diese Klasse implementiert einen Satz von Mengenoperationen ähnlich denen von `dvConfiguration`. Allerdings gibt es in dieser Klasse keine besonderen Konsistenzbedingungen außer der, daß jedes Element nur einmal vorhanden sein darf.
Jedem Zustand werden Ereignisse zugeordnet, die, von ihm ausgehend, einen Zustandsübergang auslösen können. Hierfür wird die Basisklasse `dvState` um eine Exemplarvariable vom Typ `dvEventSet` ergänzt, welche die zugeordneten Fachwerte vom Typ `dvEvent` enthält. Hinzu kommen noch entsprechende Erweiterungen an den Konstruktoren der Basisklasse sowie deren Subklassen. Des weiteren werden Methoden ergänzt, welche den Umgang mit Objekten vom Typ `dvEventSet` ermöglichen. Hierzu zählen die Methoden `hasEvent` und `getEvents`. Die Methode `hasEvent` prüft, ob dem jeweiligen Zustand ein bestimmtes Ereignis zugeordnet ist. Die Methode `getEvents` liefert als Ergebnis ein Objekt vom Typ `dvEventSet`, welches eine Liste aller Ereignisse eines Zustands beinhaltet.
Der Zustandsautomat hat nun die Möglichkeit, aus seiner aktuellen Konfiguration die Ereignisse zu bestimmen, welche zur Zeit Zustandsübergänge auslösen können. Hierzu läßt er sich von den Zustandsobjekten ihre Ereignismengen geben und fügt diese über die Bildung einer Vereinigungsmenge zusammen. Zu welchem Zustandsübergang ein Ereignis gehört, läßt sich jetzt einfach über eine Liste bestimmen, in der diese Zuordnung festgehalten ist. Für diese Liste gilt die Bedingung, daß jedes Ereignis und jeder Zustandsübergang nur genau einmal auftreten dürfen. Ansonsten wäre es möglich, daß es zu einem Ereignis zwei verschiedene Zustandsübergänge gibt. Diese Liste ist im Zustandsautomaten enthalten.
Die nachfolgende Abbildung 5.10 zeigt die Zusammenhänge zwischen den Klassen als Übersicht:

[89] siehe [Gamma 1996], Seite 157ff

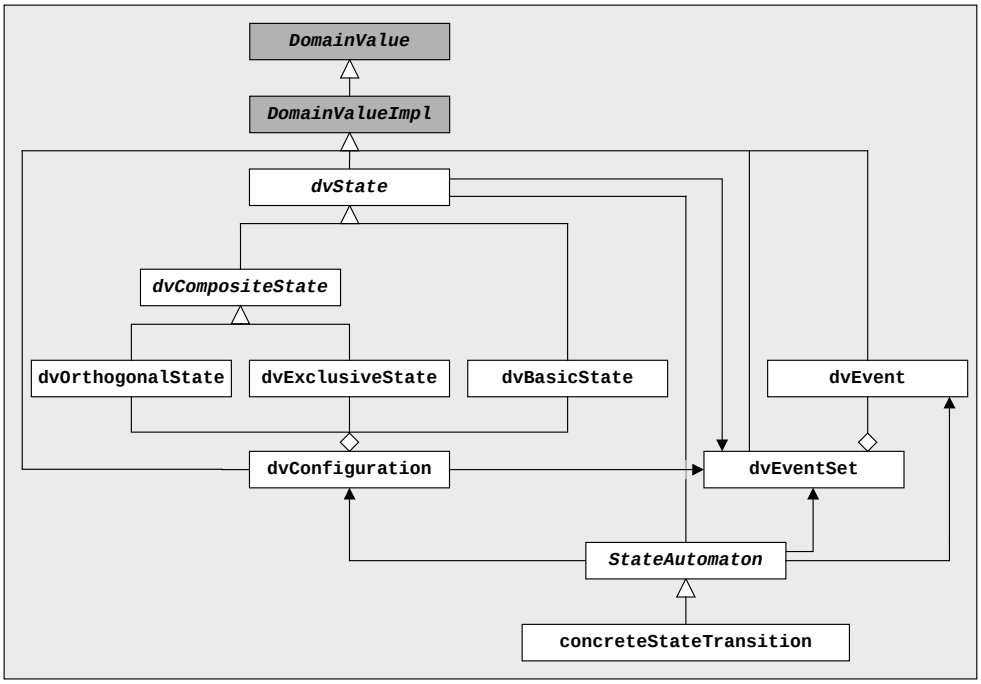

Abbildung 5.10: Modellierung von Ereignissen (`dvEvent`) und Ereignismengen (`dvEventSet`)

5.2.4.1.2 Behälter für Materialien

Neben den Ereignissen erhält ein Zustandsautomat von außen auch Eingaben in Form von Materialien oder Fachwerten, welche er an den jeweiligen Zustandsübergang weitergeben muß. Hierfür wurde eine weitere Hilfsklasse `Materialpool` eingeführt, welche alle verwendeten Materialien und Fachwerte beinhaltet. Da sie nur dazu dient, Materialien und Fachwerte mit der Umgebung auszutauschen, wird sie nicht näher erläutert.

5.2.4.1.3 Zustandsübergänge

Das Interface `StateTransition` und die Klasse `StateTransitionImpl` realisieren generalisierbare Methoden für die Implementierung von Zustandsübergängen. Im Folgenden wird zunächst die Schnittstelle des Interface `StateTransition` gezeigt. Anschließend wird die Schnittstelle der abstrakten Klasse `StateTransitionImpl` aufgeführt.

```
public interface StateTransition
{
  public boolean checkPreconditions( dvConfiguration config,
                                     Materialpool    materials);

  public dvStateTransitionResult execute(dvConfiguration config,
                                         Materialpool    materials);
}

public abstract class StateTransitionImpl implements StateTransition
{
  public boolean checkPreconditions(dvConfiguration config,
                                    Materialpool    materials)
  {...}
 public abstract dvStateTransitionResult execute(dvConfiguration config,
                                                 Materialpool    materials)
  {...}

  protected dvConfiguration _currentConfig;
  protected dvConfiguration    _nextConfig;
  protected dvState                _root;
}
```

Für einen bestimmten Anwendungsfall müssen die Methoden von `StateTransitionImpl` in den Subklassen noch mit dem entsprechenden Wissen über den Zustandsübergang und die zu benutzenden Fachlichen Services gefüllt werden. Konkrete Zustandsübergänge werden als inner classes des jeweiligen Zustandsautomaten realisiert und erben von `StateTransitionImpl`.
In der Basisklasse `StateTransitionImpl` ist nur eine Folgekonfiguration `_nextConfig` vorhanden, da dies für den Standardfall ausreichend ist. Sollten mehrere Folgekonfigurationen möglich sein, müssen diese in den Subklassen eingeführt werden. Die Methode `checkPreconditions` liefert defaultmäßig den Wert `true` und muß in den Subklassen entsprechend überschrieben werden.
`Execute` ist die Methode, welche den eigentlichen Zustandsübergang berechnet. Sie erhält die aktuelle Konfiguration und die Materialien als Argumente und berechnet aus diesen ein Ergebnis. Da sie abstrakt ist, muß sie, wie oben erwähnt, in den Subklassen entsprechend überschrieben werden. Das heißt, es muß für den konkreten Anwendungsfall bestimmt werden, welche Fachlichen Services mit welchen Materialien aufgerufen werden und welches die Zielkonfiguration ist. Eventuell müssen auch noch orthogonale Zustandsübergänge ausgeführt werden. Die berechneten Resultate müssen dann als Ergebnis an den Zustandsautomaten weitergeleitet werden.
Zum einfacheren Umgang mit den Ergebnissen wird noch die Klasse `dvStateTransitionResult` eingeführt. Da diese Klasse nur eine Kapsel für die Ergebnisse von Zustandsübergängen darstellt, wird sie nicht näher erläutert. Sie enthält neben den Materialien noch die Quell- und Zielkonfiguration und die Information darüber, ob der Zustandsübergang erfolgt ist.
Abbildung 5.12 stellt die Zusammenhänge zwischen den Klassen dar:

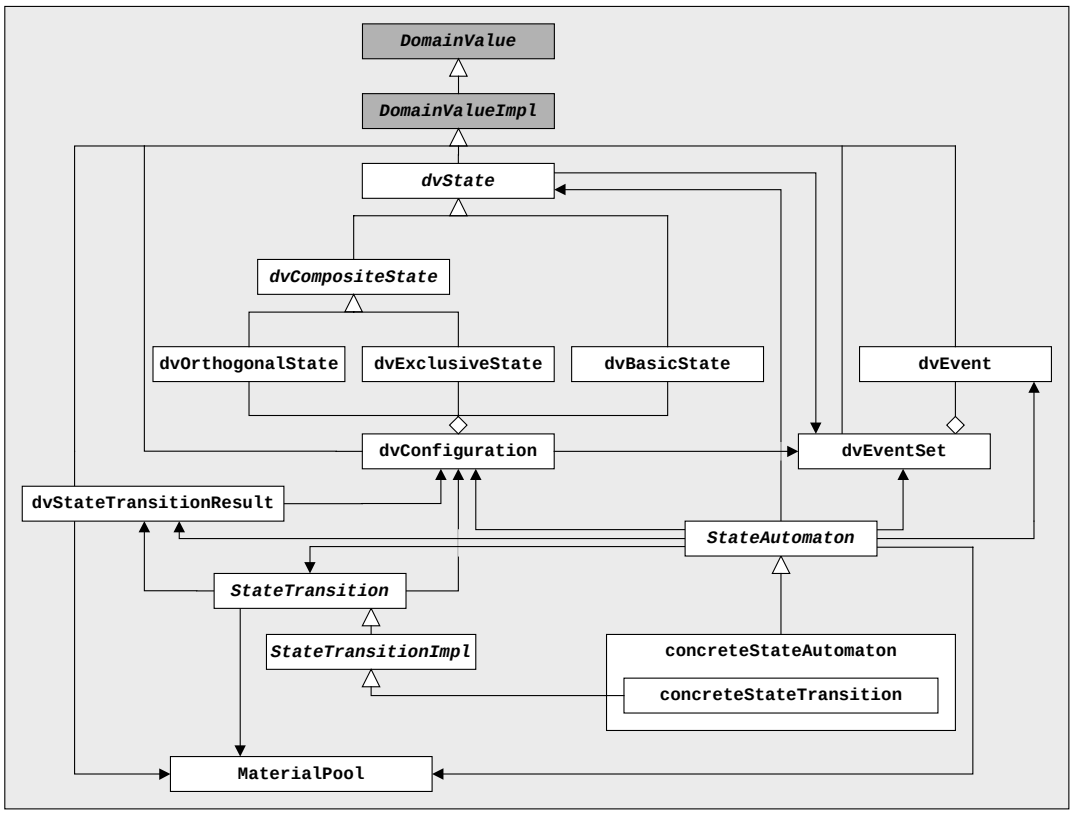

Abbildung 5.12: Modellierung von Zustandsübergängen

5.2.5 Der Zustandsautomat

Nachdem die Beschreibung der Komponenten eines Zustandsautomaten erfolgt ist, wird im Folgenden der Zustandsautomat selbst beschrieben. Da der Zustandsautomat und seine Aufgaben schon vielfach bei der Erläuterung der anderen Komponenten erwähnt wurden, ist an dieser Stelle eine kurze Schilderung des Zustandsautomaten zum Verständnis ausreichend.
Der Zustandsautomat hat die Aufgaben, für die einzelnen Komponenten nach außen eine einheitliche Schnittstelle zu bieten. Darüber hinaus hat er die Aufgabe, den Zustandsbaum aufzubauen und diesen für die Verwendung durch Zustandsübergänge und Konfigurationen bereitzustellen. Des weiteren muß er das Wissen über die aktuelle Konfiguration beinhalten.
Außerdem muß der Zustandsautomat in der Lage sein, aus dieser Konfiguration die Ereignisse zu bestimmen, welche zur Zeit Zustandsübergänge auslösen können. Hierzu läßt er sich von den Zustandsobjekten ihre Ereignismengen geben und fügt diese über die Bildung einer Vereinigungsmenge zusammen.
Zu welchem Zustandsübergang ein Ereignis gehört, läßt sich jetzt einfach über eine Liste bestimmen, in der diese Zuordnung festgehalten ist. Für diese Liste gilt die Bedingung, daß jedes Ereignis und jeder Zustandsübergang nur genau einmal auftreten dürfen. Ansonsten wäre es möglich, daß es zu einem Ereignis zwei verschiedene Zustandsübergänge gibt.
Beim Entwurf der Zustandsübergänge wurde festgestellt, daß der Zustandsautomat über Ereignisse und Materialien verfügen muß. Ereignisse und Materialien erhält der Zustandsautomat von außen. Wenn ein Ereignis registriert wird, muß der Zustandsautomat den zugehörigen Zustandsübergang aufrufen. Er darf allerdings nur solche Zustandsübergänge aufrufen, die im der aktuellen Zustandskonfiguration ausführbar sind. An den Zustandsübergang werden die Materialien, die aktuelle Zustandskonfiguration und das auslösende Ereignis als Argumente übergeben. Als Ausgabe erhält der Zustandsautomat eine veränderte Zustandskonfiguration, die dann, nach einer Prüfung auf syntaktische Korrektheit, als neue Konfiguration übernommen werden muß. Darüber hinaus erhält er die bearbeiteten Materialien, die er an die ihn benutzende Instanz weitergeben muß.

5.2.5.1 Implementation

Für diesen Entwurf wurde vorgesehen, daß es einen abstrakten Zustandsautomaten gibt, welcher allgemeingültige Methoden zur Verfügung stellt und einen konkreten Zustandsautomaten für jedes zu modellierende System.
Im Folgenden wird die Schnittstelle der abstrakten Basisklasse für Zustandsautomaten gezeigt:

```
public abstract class StateAutomaton
{
  public StateAutomaton()
   {...}
  public dvConfiguration getCurrentConfiguration()
   {...}
  public boolean hasEvent(dvEvent event)
   {...}
  public boolean hasState(dvState state)
   {...}
  public Hashtable execute(dvEvent event, Materialpool materials)
   {...}
  protected void setConfiguration(dvConfiguration configuration)
   {...}
  protected void checkConfiguration(dvConfiguration configuration)
   {...}
```

```
    protected StateTransition getStateTransition(dvEvent event)
      {...}
    protected CompositeState getRoot()
      {...}

    private   dvConfiguration _CurrentConfiguration;
    protected CompositeState  _Root;
    protected Hashtable       _StateTransitionList;
  }
```

Die Variable `_Root` bezeichnet den Zustand, der die Wurzel des Zustandsbaumes (siehe Abschnitt 4.3.1) bildet. Der Aufbau eines Zustandsbaumes beginnt in der untersten Ebene. Die erzeugten Zustandsobjekte der untersten Ebene werden dann zu den Zuständen der nächsten Ebene zusammengefügt. Dies geschieht, bis die Wurzel des Baumes erreicht ist. Der Aufbau muß allerdings in einem konkreten Zustandsautomaten erfolgen, da nur dort das Wissen über die Zustände des jeweiligen Anwendungssystems vorhanden ist.
Aus dem Zustandsbaum wird auch die initiale Konfiguration des Systems bestimmt. Dies geschieht, wie in Abschnitt 5.2.3 beschrieben, mit Hilfe der Methode `getInitialConfiguration`.
Die Methode `setConfiguration` ist für die Manipulation der Variablen `_CurrentConfiguration` zuständig, welche die aktuelle Konfiguration des Zustandsautomaten abbildet. Die Methode `getCurrentConfiguration` liefert die aktuelle Konfiguration als Ergebnis zurück. Mit der Methode `checkConfiguration` kann eine Konfiguration auf ihre syntaktische Korrektheit geprüft werden. Hierzu wird an der Konfiguration die Methode `isComplete` aufgerufen, mit der ermittelt werden kann, ob eine Konfiguration vollständig ist.
In der Liste `_StateTransitionList` ist die Zuordnung von Ereignissen zu Zustandsübergängen zu finden. Diese Liste muß, ähnlich der Variablen `_ Root`, in einer Subklasse implementiert werden. Um diese Liste zu füllen, müssen alle inner classes, die Zustandsübergänge realisieren, und Ereignisse instanziiert werden.
Zu welchem Zustandsübergang ein Ereignis gehört, läßt sich über die Methode `getStateTransition` feststellen, welche die Liste `_StateTransitionList` nach dem betreffenden Ereignis durchsucht und, wenn sie fündig wird, den zugehörigen Zustandsübergang als Ergebnis zurückliefert.
Mit den Methoden `hasState` kann festgestellt werden, ob ein Zustand Teil der aktuellen Konfiguration ist. Mit `hasEvent` kann überprüft werden, ob ein Ereignis zum aktuellen Zeitpunkt einen Zustandsübergang auslösen kann.
Eine History-Funktion läßt sich in diesem Ansatz über die Einführung eines zusätzlichen Attributes an der Klasse `StateAutomaton` realisieren, in welcher die vorangegangene Zustandskonfiguration gespeichert wird. Allerdings wird hier darauf verzichtet da dies keine allgemein benötigte Funktionalität ist.

5.3 Verwendung des Zustandsautomaten

Im Nachfolgenden wird gezeigt, wie der eben entworfene Zustandsautomat verwendet wird. Um dies zu verdeutlichen, wird ein Zustandsautomat in den aus Kapitel 3 bekannten Selbstbedienungsautomaten für Web-Anwendungen integriert. Als Anwendungsfall wird das aus Abschnitt 3.2.1 bekannte Beispiel der Containervoranmeldung gewählt. Hierbei benutzt das Dispatcher-Servlet, welches die Handhabung des Selbstbedienungsautomaten beinhaltet, den Zustandsautomaten. Dieser wiederum benutzt die Fachlichen Services zum Ausführen fachlicher Operationen.
Abbildung 5.13 zeigt wie sich der Zustandsautomat in den Entwurf eingliedert:

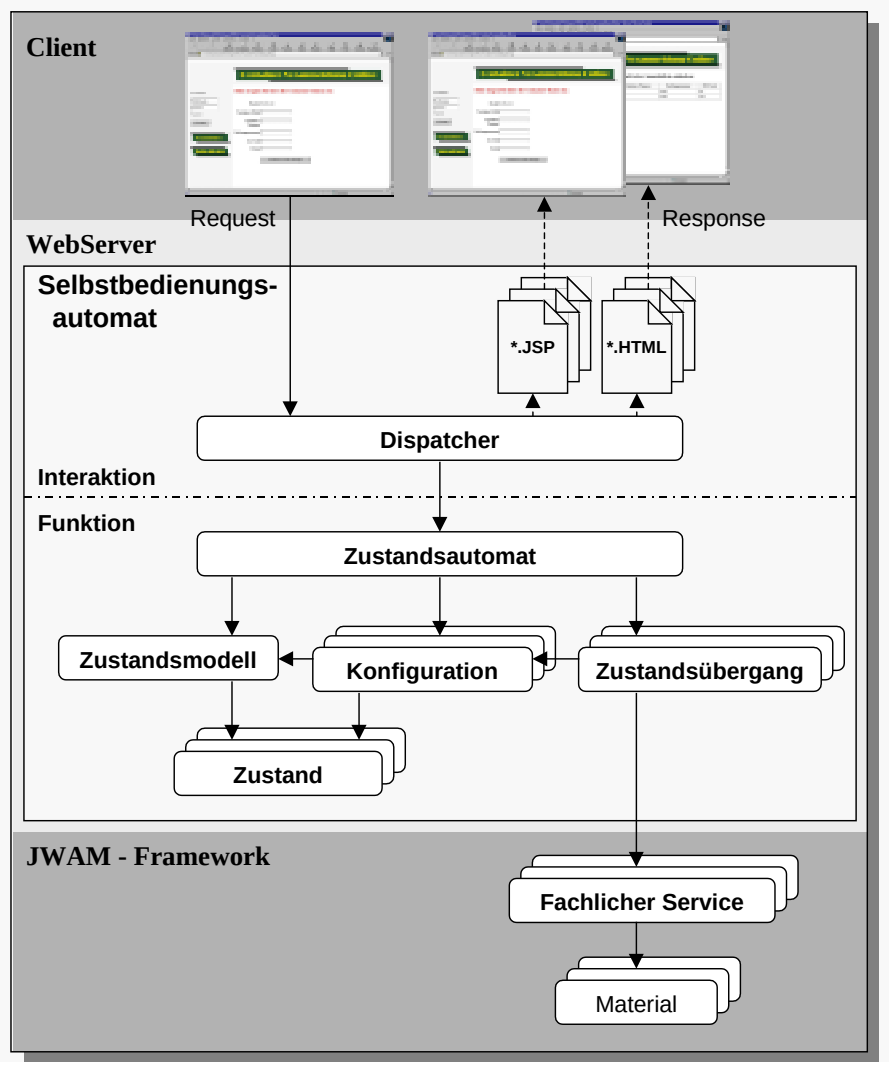

Abbildung 5.13: Verwendung des WAM-Zustandsautomaten im Entwurf

Bevor der Aufruf eines Zustandsüberganges geschildert wird, soll nachfolgend kurz die Umsetzung des Dispatchers erläutert werden, um ein besseres Verständnis des anschließend beschriebenen Zustandsüberganges zu ermöglichen.
Die generelle Funktionalität zum Abarbeiten von Requests[90] wird durch die abstrakte Klasse `Dispatcher` realisiert, welche von `HttpServlet` erbt. Hierdurch verfügt sie über die Grundfunktionalität zum Umgang mit Request und Response (`doPost` und `doGet`). Zusätzlich wird sie um Methoden ergänzt, welche das Extrahieren von Ereignissen und Daten (`readRequest`), das Aufrufen von Aktionen (`execute`) und das Weiterleiten zur Folgeseite (`getNextPage`) übernehmen. Die Schnittstelle dieser Klasse stellt sich folgendermaßen dar:

```
public abstract class Dispatcher extends HttpServlet
{
   public void doPost(HttpServletRequest   request,
                      HttpServletResponse  response)
                  throws IOException, ServletException
   {...}
```

[90] vgl. Abschnitt 3.4.3 **Trennung von Präsentation und Handhabung**

```
    public void doGet(HttpServletRequest   request,
                      HttpServletResponse  response)
                    throws IOException, ServletException
      {...}
    public String execute(Materialpool materials)
      {...}
    public void readRequest(HttpServletRequest request,
                            HttpSession        session)
      {...}
    public String getNextPage(dvStateTransitionResult result)
      {...}
    public abstract Hashtable getMap();

   protected static Hashtable _Map = null;
}
```

Für den jeweiligen Anwendungsfall müssen entsprechende Subklassen der Basisklasse gebildet werden, welche Kenntnis über den Kontext haben. Die Subklassen haben also das Wissen über einen konkreten Zustandsautomaten und das Wissen über die Folgeseiten als Reaktion auf Benutzeraktionen. Die Methode `getMap` entsprechend des konkreten Anwendungsfalles implementiert werden. Hierfür muß die Liste `_Map` entsprechend mit den Zuordnungen von Java Server Page zu Konfiguration und Ereignis gefüllt werden.
Abbildung 5.14 zeigt die Zusammenhänge zwischen dem `Dispatchers` und seiner Subklasse `concreteDispatcher` und dem Zustandsautomaten mit seinem Komponenten als Klassendiagramm:

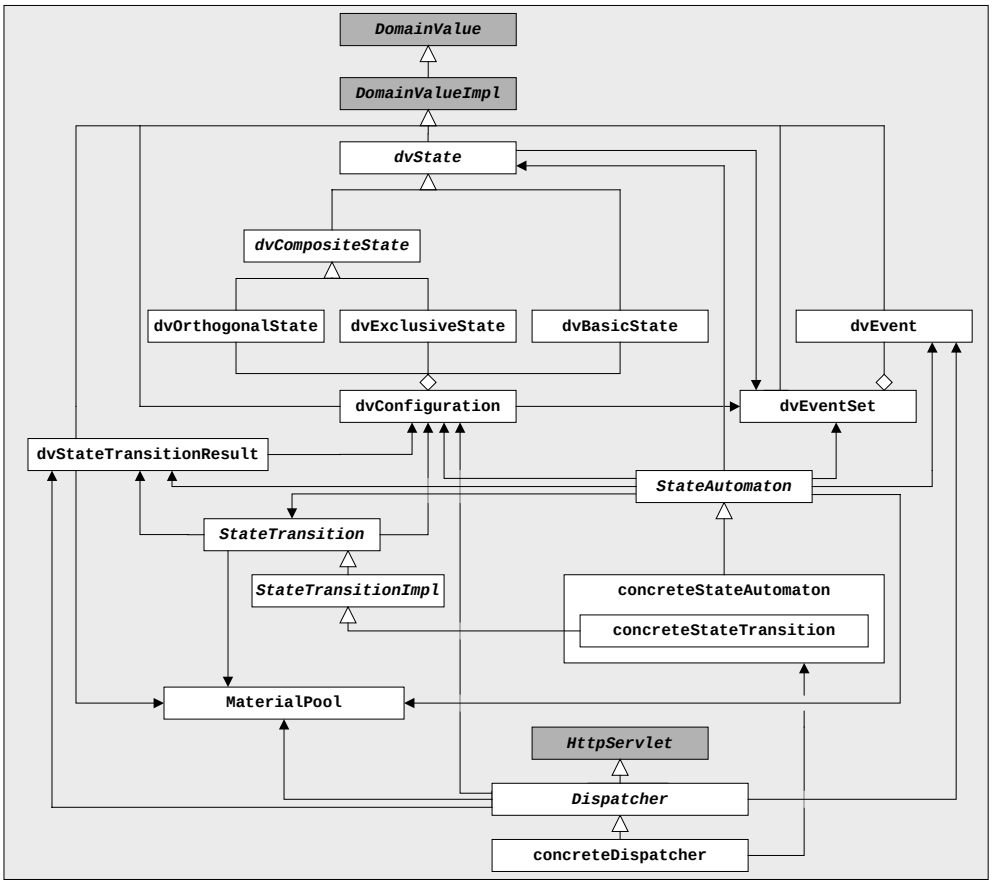

Abbildung 5.14: Integration der Klasse `Dispatcher` und ihrer Subklasse `concreteDispatcher`

Der Ablauf eines Aufrufes ist in Abbildung 5.15 beschrieben. Die Methode `doPost` nimmt einen Request entgegen. Aus dem Request wird die Session ausgelesen. Mit `readRequest` werden alle Daten einer `session` ausgelesen. Hierbei handelt es sich um den Zustandsautomaten, das vom Anwender ausgelöste Ereignis und die zugehörigen Materialien, Fachwerte sowie Standarddatentypen. Alle diese Objekte, bis auf den Zustandsautomaten, werden in einem Objekt vom Typ `Materialpool` verwaltet. Um auf ein Ereignis zu reagieren, wird die Methode `execute` mit dem Materialpool `materials` als Argument aufgerufen. Diese entnimmt aus `material` das Ereignis und ruft am Zustandsautomaten die Methode `execute` mit dem Ereignis und dem Materialienpool als Argumenten auf. Der Zustandsautomat führt nun den entsprechenden Zustandsübergang durch. Als Ergebnis erhält der `Dispatcher` über ein Objekt vom Typ `dvStateTransitionResult` (in der Abbildung als `result` bezeichnet) den veränderten Materialienpool. Anschließend wird mit der Methode `getNextPage` die Folgeseite bestimmt. Hierfür wird über die aktuelle und die vorangegangene Konfiguration sowie über das Ereignis, welches in dem Ergebnis des Zustandsüberganges (`result`) festgehalten ist, der Name einer Java Server Page oder einer HTML-Seite bestimmt. Diese Zuordnung ist in einer Exemplarvariablen `_Map` festgehalten. Mit diesem Ergebnis wird über `GetRequestDispatcher` die Folgeseite als Response an den Browser des Anwenders gesendet und dort zur Anzeige gebracht.

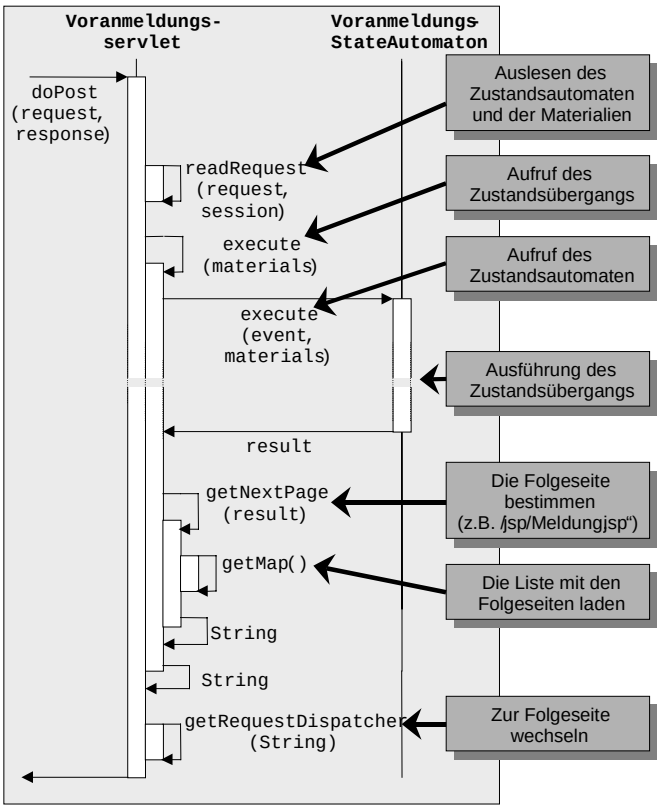

Abbildung 5.15: Interaktionsdiagramm zur Verwendung des `Dispatchers`

Nachdem der Aufruf eines Zustandsüberganges an einem Zustandsautomaten durch einen Selbstbedienungsautomaten erläutert wurde, soll nachfolgend exemplarisch geschildert werden, wie ein Zustandsübergang abläuft. Grundlage für den hier gezeigten Zustandsübergang ist der Statechart aus Abbildung 3.22.

Am Zustandsautomaten `VoranmeldungsStateAutomaton` wird die Methode `execute` mit `materials` und `event` als Argumenten aufgerufen. In dieser Methode wird über `getStateTransition` unter Angabe des Ereignisses `event` der zugehörige Zustandsübergang ermittelt. Voraussetzung ist, daß der Zustandsübergang möglich ist. In dem hier betrachteten Beispiel wird angenommen, daß der Übergang möglich ist. An dem Zustandsübergang `ContainerAnmelden` wird die Methode `Execute` aufgerufen. Ihr wird die aktuelle Konfiguration `currentConfiguration` des Zustandsautomaten und der Materialienpool `materials` übergeben. Zunächst werden mit `checkPreconditions` die Vorbedingungen geprüft. Anschließend wird aus dem Materialienpool das Material `Voranmeldung` herausgefiltert. Mit diesem Material wird am Fachlichen Service `AnmeldeService` die Operation `Check` gerufen. Liefert diese Methode `true` zurück, wird der orthogonale Zu-

standsübergang `InListeEintragen` aufgerufen. Hierbei wird ihm anstelle der Anfangskonfiguration eine entsprechend dem Zustandsübergang `ContainerAnmelden` geänderte Konfiguration übergeben. Ähnlich wie bei dem eben genannten Zustandsübergang `ContainerAnmelden` werden nun die Vorbedingungen geprüft und anschließend Operationen am Fachlichen Service ausgeführt. Nach deren Ausführung wird das nicht mehr benötige Material der Voranmeldung aus `materials` entfernt. Als Ergebnis des Zustandsübergangs wird ein Objekt vom Typ `StateTransitionResult` namens `result` erzeugt. Bei der Erzeugung werden `materials`, Quell- und Zielkonfiguration als Argumente übergeben. Das Objekt `result` wird zunächst an die aufrufende Methode und anschließend an den Zustandsautomaten als Ergebnis zurückgeliefert. Der Zustandsautomat läßt sich von diesem die Folgekonfiguration geben und prüft, ob die Konfiguration korrekt ist. Ist sie korrekt, weist sie der Zustandsautomat seiner eigenen Konfiguration zu. Danach wird `result` an die aufrufende Klasse zurückgeliefert.
Abbildung 5.16 zeigt den eben beschriebenen Ablauf als Interaktionsdiagramm:

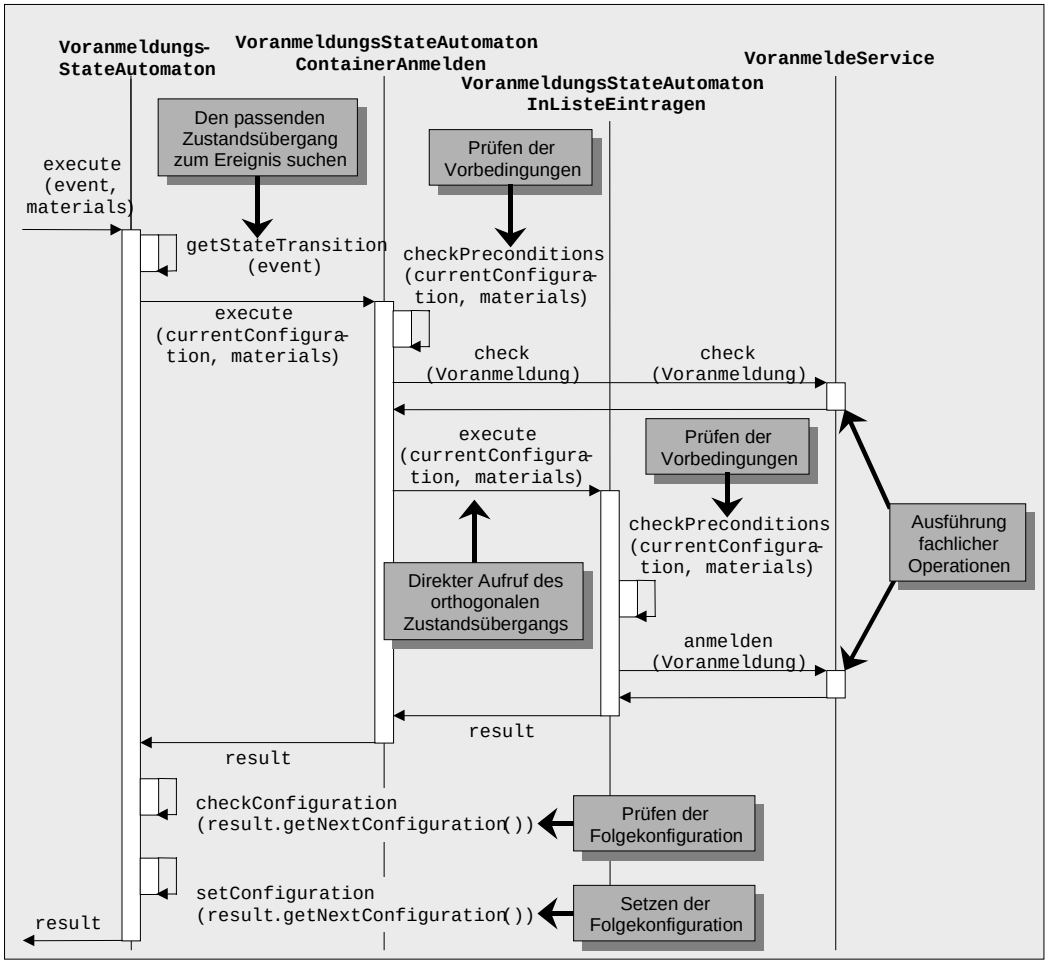

Abbildung 5.16: Interaktionsdiagramm zur Ausführung eines Zustandsübergangs

5.4 Zusammenfassung

Im folgenden Abschnitt werden die Ergebnisse dieses Kapitels zusammengefaßt.
Zunächst wurde untersucht, welche der verschiedenen Entwurfsmuster aus der Literatur sich für eine Modellierung der in Kapitel 4 eingeführten Statecharts eignen. Es wurde fest-

gestellt, daß sich die untersuchten Entwurfsmuster des State-Pattern und des erweiterten State-Pattern nicht für eine Umsetzung in einen Zustandsautomaten eignen.
Anschließend wurde der Entwurf des Zustandsautomaten beschrieben und die Designentscheidungen erläutert.
Bei der Modellierung des Zustandsautomaten wurde darauf geachtet, daß die Benutzung von Komponenten nur in einer Richtung erfolgt, um die Abhängigkeiten der Komponenten untereinander zu reduzieren. Somit konnte vermieden werden, daß zwischen den Komponenten zyklische Benutzt-Beziehungen bestehen.
Es wurde bei der Umsetzung nur eine minimale Anzahl an Klassen verwendet, um einer „inflationären“ Zunahme von Klassen vorzubeugen, wie sie bei den vorher untersuchten Entwurfsmustern vorkommen kann. Dies konnte durch die Verwendung von Fachwerten und Singletons realisiert werden.
Mit dem Zustandsautomaten können Konzepte wie Hierarchien und Orthogonalität modelliert werden. Da der Automat hierarchisch aufgebaut ist, kann er leicht erweitert oder verfeinert werden. Mit Zustandsautomaten können bestimmte Teile von Vorgängen als orthogonale Abläufe modelliert werden.
In dem Zustandsautomaten ist eine explizite Trennung von Zustandsmodell, aktuellem Zustand und Zustandsübergängen realisiert worden, wodurch der Entwurf gegenüber anderen Arten der Modellierung erheblich an Verständlichkeit gewinnt. Der aktuelle Zustand wird immer explizit gemacht. Über diesen ist auch jederzeit bestimmbar, welche Zustandsübergänge möglich sind.
Das Wissen über Zustandsübergänge wurde an einer Stelle im Entwurf konzentriert, wodurch die Verständlichkeit des Entwurfs und die Wartbarkeit der softwaretechnischen Umsetzung erhöht wird.
Es wurde gezeigt, wie bei Zustandsübergängen Fachliche Services verwendet oder manipuliert werden. Da Änderungen an Materialien nur über Fachliche Services erfolgen können, kann somit die Einhaltung einer fachlich korrekten Verwendung von Materialien und Fachlichen Services im Rahmen des gesamten Arbeitsvorganges garantiert werden.
Neben dieser Kapselung wurden der Zustandsautomat und sein Zustandsmodell dazu verwendet, die Repräsentation der Anwendung im Browser zu steuern. Beim Dispatching wurde die jeweilige Folgeseite aus Quell- und Zielkonfiguration sowie aus dem Ereignis eines Zustandsüberganges bestimmt.
Des weiteren ist es gelungen, mit dem Zustandsautomaten und seinen Komponenten ein allgemein gültiges Konzept für Zustände, Zustandshierarchien, Konfigurationen und Zustandsübergänge zu schaffen. Für die Zustände und Konfigurationen müssen in einen konkreten Anwendungsfall nur die Objekte für die betreffenden Werte erzeugt werden. Das heißt, es müssen den Zuständen nur Name, Ereignisse und Subzustände zugeordnet werden. Diese Zustandsobjekte können dann einfach zu Konfigurationen zusammengefaßt werden. Die Implementationsarbeit beim Einsatz von Zustandsautomaten besteht lediglich darin, den konkreten Zustandsautomaten und die notwendigen Zustandsübergänge für den jeweiligen Anwendungsfall anzupassen.

Kapitel 6: Diskussion und Ausblick

Nachdem im vorangegangenen Kapitel die wesentlichen Aspekte des Entwurfs vorgestellt wurden, soll an dieser Stelle diskutiert werden, ob und wie sich ein solcher Zustandsautomat zur Vorgangssteuerung noch für andere WAM-Komponenten eignet.
Zunächst einmal kann festgestellt werden, daß jedes Objekt[91] über ein Benutzungs- bzw. Zustandsmodell[92] verfügt. Hierbei kann es sich um ein explizites (zum Beispiel als endlicher Zustandsautomat) oder um ein implizites (auf Basis der Werte der Exemplarvariablen) Zustandsmodell handeln. Operationen, welche vom Zustand eines Objektes abhängen oder diesen verändern können, dürfen nur dann verwendet werden, wenn das Objekt in dem entsprechenden Zustand ist, ansonsten würde dieses Objekt in einen inkonsistenten Zustand wechseln. Es wird somit eine bestimmte Reihenfolge für Operationen festgelegt. Bei [Züllighoven 1998] wird dies als *Protokoll* bezeichnet. Nun ist es sicherlich übertrieben, für jedes Objekt ein explizites Zustandsmodell für die Benutzung aufzustellen und dies über einen Zustandsautomaten zu realisieren, da der Aufwand für eine solche Umsetzung den für die Realisierung eines impliziten Zustandsmodells sicherlich übersteigt. Grundsätzlich besteht aber die Möglichkeit, dies zu tun. Bevor weitere Verwendungsmöglichkeiten geschildert werden, werden daher Kriterien für den Einsatz von Zustandsautomaten formuliert.

6.1 Kriterien für die Verwendung von Zustandsautomaten

Die in diesem Abschnitt geschilderten fachlichen Kriterien[93] stellen die Grundlage für eine weitere Verwendung von Zustandsautomaten zur Vorgangssteuerung dar. Im Nachfolgenden sollen zunächst die notwendigen Bedingungen für den Einsatz von Zustandsautomaten genannt werden, welche unabhängig vom fachlichen Einsatzkontext gültig sein müssen. Anschließend werden die vom fachlichen Einsatzkontext abhängigen Kriterien ausgeführt.
Eine notwendige Grundvoraussetzung für den Einsatz ist die Tatsache, daß es sich bei dem zu modellierenden System um eines handelt, welches nur beim Eintreffen externer Ereignisse aktiv wird und dann entsprechend reagiert.

> **ERGEBNIS:**
>
> *Es muß sich bei dem zu modellierenden System um ein rein reaktives System handeln.*

Dieses Kriterium an sich ist aber wenig aussagekräftig, da es für fast alle JWAM-Komponenten zutrifft. Deshalb sind weitere Kriterien notwendig.
Ein weiteres notwendiges Kriterium ist, daß die abzubildenden fachlichen Vorgänge schematisierbar sind. Hierunter wird verstanden, daß sie als immer wiederkehrende Abläufe identifizierbar sind. In jedem Zustand eines Vorganges sollte klar sein, welche Arbeitsschritte als nächstes möglich sind. Die Entscheidung, welcher Schritt ausgeführt wird, obliegt der benutzenden Instanz.

---

[91] vgl. [Züllighoven 1998], Seite 28

[92] siehe Abbildung 5.5: **Unterschiede zwischen Werten und Objekten**

[93] Zu den fachlichen Kriterien, welche für die Verwendung von Zustandsautomaten sprechen, können noch technische Gründe kommen. In dieser Arbeit sind dies Probleme wie die Zustandslosigkeit von Webtop-Anwendungen und die Verwendung des Caching-Mechanismus. Da dies in der Regel sehr spezielle Probleme sind, wird hier auf eine Auflistung verzichtet.

Trifft dies nicht zu, ist keine Notwendigkeit für die Verwendung von Zustandsautomaten gegeben.

> **ERGEBNIS:**
>
> *Zustandsautomaten zur Vorgangssteuerung sind überall dort sinnvoll zu verwenden, wo zur Wahrung fachlicher Konsistenz schematische Abfolgen von Arbeitsschritten formuliert und eingehalten und Bearbeitungszustände über ein explizites Zustandsmodell ausgedrückt werden müssen.*

Der Vorgang, zu dessen Beschreibung Zustandsautomaten herangezogen werden, muß hinreichend komplex sein. Bestehen nur geringe Abhängigkeiten zwischen den Arbeitsschritten, ist eine Verwendung von Zustandsautomaten nicht notwendig.

> **ERGEBNIS:**
>
> *Die zu modellierenden Vorgänge sollten hinreichend komplex sein.*

Darüber hinaus muß die Ununterbrechbarbarkeit der Arbeitsschritte, die durch Zustandsübergänge modelliert werden, gegeben sein. Kann dies nicht garantiert werden oder ist dies aus fachlichen oder technischen Gründen nicht möglich, sollte auf den Einsatz von Zustandsautomaten verzichtet werden.

> **ERGEBNIS:**
>
> *Die Arbeitsschritte des zu modellierenden Vorganges müssen ununterbrechbar sein.*

Die nachfolgend genannten Kriterien beziehen sich auf den konkreten Einsatzkontext und leiten sich unmittelbar aus der Art der Verwendung von Zustandsautomaten in dieser Arbeit ab.
Zustandsautomaten können verschiedene Komponenten miteinander synchronisieren, so daß diese ein gemeinsames Verhalten zeigen. Die einzelnen Komponenten müssen keinerlei Kenntnis voneinander haben und können somit unabhängig voneinander entworfen werden. Hierdurch verringert sich die Komplexität dieser Komponenten, da nicht mehr Bezug auf den Zustand einer korrespondierenden Komponente genommen werden muß.

> **ERGEBNIS:**
>
> *Zustandsautomaten können zur Komposition und Synchronisation unterschiedlicher Komponenten verwendet werden, so daß diese keinerlei Kenntnis voneinander haben müssen und somit unabhängig voneinander entworfen werden können.*

Durch Zustandsautomaten können auch fachliche Funktionalitäten eingeschränkt oder reglementiert werden, ohne daß dies in der entsprechenden Komponente abgebildet werden muß. So ist es denkbar, daß ein Material oder ein Fachlicher Service in ein und demselben Anwendungssystem in unterschiedlichen konkreten Benutzungssituationen ein anderes Verhalten haben soll, ohne daß dieses Wissen in der Komponente realisiert werden soll. Der Zustandsautomat bietet nun die Möglichkeit, dieses Wissen von der jeweiligen Kom-

ponente getrennt zu modellieren und dieses Wissen anderen Komponenten zur Verfügung zu stellen.

> **ERGEBNIS:**
>
> *Zustandsautomaten können zur Einschränkung des Benutzungsmodelles einer Komponente verwendet werden, ohne die jeweilige Komponente zu verändern.*

6.2 Verwendungsmöglichkeiten

Nun stellt sich die Frage, wo sich explizite Zustandsmodelle und deren Realisierungen durch Zustandsautomaten zur Vorgangssteuerung noch verwenden lassen und an welcher Stelle von dieser Möglichkeit abzuraten ist.
Die hier aufgezeigten Möglichkeiten sollen dabei nicht die bisherigen Methoden zur Modellierung von Vorgängen ersetzen, sondern eine Alternative zu diesen bieten. Die Entscheidung, welche Möglichkeit konkret angewendet wird, liegt beim Entwickler.
Die nachfolgend geschilderten exemplarischen Verwendungsmöglichkeiten sind Beispiele, in welchen Fällen eine Verwendung noch sinnvoll wäre.

6.2.1 Fachliche Services

Bei der weiteren Arbeit mit Zustandsautomaten wäre es wünschenswert, wenn auch Fachliche Services ein explizites Zustandsmodell hätten, wie dies bei [Otto&Schuler 2000] gefordert wird[94]. Hiernach verfügen Fachliche Services, die Materialien und Methoden zum Umgang mit diesen zur Verfügung zu stellen oder die eine Dienstleistung realisieren, über ein Zustandsmodell.
Das Wissen über das Zustand- bzw. Sitzungsmodell kann sich nach [Otto&Schuler 2000] entweder in dem Fachlichen Service selbst oder in der diesen benutzenden Komponente befinden. In dieser Arbeit wurde das Zustandsmodell über den Zustandsautomaten in die den Fachlichen Service benutzende Komponente integriert. Ein solches Modell ließe sich aber auch problemlos über Zustandsautomaten in einen Fachlichen Service integrieren. Der Zustandsautomat würde dann den Zugriff auf die Materialien kapseln (siehe Abbildung 6.1). Hierdurch wäre es dann möglich, den Vorgang an eventuell veränderte Anforderungen anzupassen, ohne den Fachlichen Services zu ändern. Darüber hinaus wäre es bei der Modellierung von Fachlichen Services und Zustandsautomaten möglich, Seiteneffekte oder zyklische Benutzungsbeziehungen von Fachlichen Services schon bei der Entwicklung zu erkennen und diese dann entsprechend zu entkoppeln. In diesem Fall könnten auch für das Zusammenspiel von Fachlichen Services untereinander bzw. für deren Verwendung in einem Zustandsautomaten, wie er hier entwickelt wurde, Aussagen über das Verhalten des Gesamtsystems gemacht werden. Gleichzeitig wäre es einfacher, Zustandsautomaten zu bauen, da man die Zustandsmodelle der Fachlichen Services in das Zustandsmodell integrieren könnte.

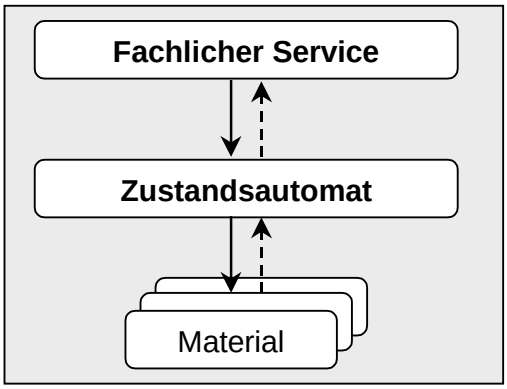

Abbildung 6.1: Verwendung von Zustandsautomaten in Fachlichen Services

[94] vgl. Abschnitt 2.1.4 **Fachliche Services** und Abschnitt 3.4.2 **Kapselung fachlichen Wissens**

Des weiteren würden gerade Fachliche Services von einem expliziten Zustandsmodell und der Beschreibung ihres Benutzungsmodells durch Statecharts profitieren, da es sich bei diesen Services um Komponenten handelt, mit denen fachliche Funktionalität einem großen Kreis von Entwicklern schnell und einfach zur Verfügung gestellt werden soll. Ein explizites Zustandsmodell fördert die leichtere Verständlichkeit und Benutzbarkeit eines Services, da immer genau festgelegt wird, wann welcher Fall eintritt.

Zustandsautomaten wurden in dieser Arbeit zur Komposition und Synchronisation unterschiedlicher Fachlicher Services verwendet, um für diese einen fachlichen Zusammenhang zu erstellen, ohne daß dies in den Fachlichen Services bekannt sein muß. Das heißt, daß die Kapselung fachlichen Wissens bewahrt[95] wurde. Eine solche Komposition könnte auch für die Konstruktion eines neuen Fachlichen Services aus verschiedenen anderen Services, welche Teilfunktionalitäten bereitstellen, genutzt werden. Über den Zustandsautomaten könnten die benutzten Services zu einem neuen Service zusammengefügt werden, ohne daß die Forderung nach Kapselung des fachlichen Wissens bei Fachlichen Services verletzt wird (siehe Abbildung 6.2).

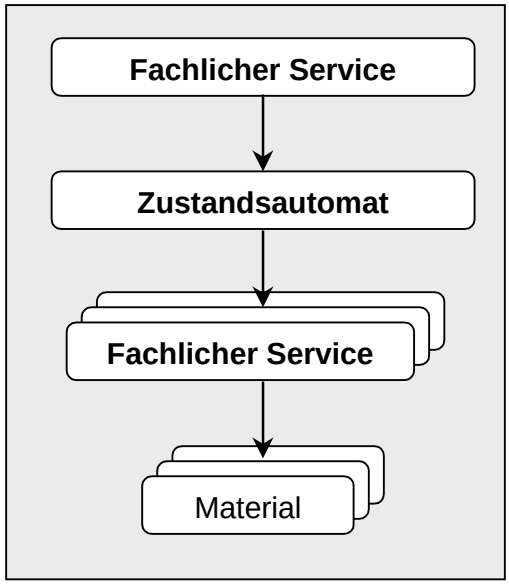

Abbildung 6.2: Komposition Fachlicher Services

6.2.2 Werkzeuge

Für Werkzeuge[96] gilt, daß sie möglichst keine Vorgänge vorgeben sollten. Ein Einsatz von Zustandsautomaten bei der Konstruktion von Werkzeugen ist daher in der Regel nicht angebracht.

Zwar haben Werkzeuge einen Werkzeugzustand und es kann logische Abhängigkeiten zwischen Methodenaufrufen geben, aber die Handhabung sollte möglichst flexibel[97] sein. Daher liegt das Augenmerk auf der Anpaßbarkeit. Das folgende Zitat verdeutlicht diese Forderung:

> *[...] Werkzeuge definieren aber keine Funktionsketten oder Abläufe. Ein Werkzeug wird zwar für eine bestimmte Aufgabe oder Teilaufgabe modelliert. Da die konkreten Arbeitssituationen aber unterschiedlich sein können, kann die genaue Reihenfolge von Arbeitsschritten nicht festgelegt werden. [...] Soll ein Werkzeug den Anwender bei einer Tätigkeit unterstützen, müssen wir darauf achten, daß der Anwender darüber entscheiden kann, in welcher Reihenfolge er welche Operation des Werkzeugs aufruft, und wann er seine Arbeit unterbricht. Dies macht die Flexibilität eines Werkzeugs aus.*
> [Züllighoven 1998], Seite 173

Wie oben bereits geschildert, kann es aber den Fall geben, daß Abhängigkeiten bestehen. In diesem Fall könnten Zustandsautomaten als Teil der Funktionskomponente verwendet

[95] vgl. [Otto&Schuler 2000], Seite 157
[96] vgl. [Züllgihoven 1998], Abschnitt 2.1.2.2 **Werkzeug** und Abschnitt 2.1.3 **Werkzeugkonstruktion**
[97] vgl. Abschnitt 2.1.1.1 **Arbeitsplatz für eigenverantwortliche Expertentätigkeit**

werden, da mit ihnen immer klar ist, wann eine Operation ausführbar ist und in welchem Zustand sich ein System befindet. Es wäre auch problemlos möglich, einen Zustandsautomaten mit einem Vertragsmodell zu koppeln und an der Benutzungsschnittstelle den Zustand über sondierende Operationen sichtbar zu machen. Allerdings sollten, wie bei den oben genannten Kriterien geschildert, die Vorgänge eine gewisse Komplexität aufweisen. Gibt es nur schwache Abhängigkeiten in dem Werkzeug, wie zum Beispiel, daß alle Methodenaufrufe von einer einzigen Vorbedingung abhängen, ist die Forderung nach einer hinreichenden Komplexität nicht gegeben.
Eine Verwendungsmöglichkeit von Zustandsautomaten wären Werkzeuge, die zur Unterstützung eines Arbeitsplatz nach dem Leitbild des Funktionsarbeitsplatzes für eigenverantwortliche Expertentätigkeit[98] entworfen werden. In diesem Bereich finden sich viele Anwendungsfälle, bei denen es darum geht, Routinevorgänge zu unterstützen, die in steter Folge wiederholt werden müssen. Hierbei könnte ein Zustandsautomat nützlich sein, indem er diese Vorgänge abbildet und dafür sorgt, daß die Abfolge der Bearbeitungsschritte eingehalten wird. Allerdings müßte für den Einsatz in diesem Kontext noch geklärt werden, wie sich ein Zustandsautomat in bestimmten Ausnahmesituationen, welche eine Abweichung von Routinehandlungen notwendig machen, verhält. Es wäre denkbar, den Zustandsautomaten vorübergehend zu deaktivieren und nach Behebung der Probleme wieder in Betrieb zu nehmen. In diesem Fall wäre noch das Problem zu lösen, wie sichergestellt werden kann, daß sich Zustandsautomat und Werkzeug nach der Inbetriebnahme wieder synchronisieren.

6.2.3 Automaten

Grundsätzlich ist eine Spezifikation von Automaten im Sinne des WAM-Ansatzes mit Statecharts und eine Modellierung als Zustandsautomat möglich. Allerdings ist ein solches Vorgehen nicht sinnvoll, da Automaten im WAM-Ansatz nach dem Anstoßen ohne weiteres Zutun einer benutzenden Instanz ablaufen und ein Ergebnis produzieren[99]. Ein solches Verhalten entspricht nicht dem Charakter der in dieser Arbeit diskutierten Zustandsautomaten, welche reaktive Systeme beschreiben, sondern eher dem eines Zustandsüberganges innerhalb eines Zustandsautomaten. Aus diesem Grund bietet es sich an, Automaten in Zustandsautomaten zu verwenden, wo sie genau diesen einen Schritt ausführen.

6.2.4 Materialien

In der Regel ist von der Verwendung von Zustandsautomaten bei der Modellierung von Materialien abzusehen, da in diesem Fall ein bestimmter Umgang mit dem Material festgeschrieben wird. Dies hat zum einen den Effekt, daß ein Material nur noch in einem sehr engen Rahmen zu verwenden wäre. Ändert sich in einem solchen Fall der Benutzungsvorgang, müssen alle auf diesem Material arbeitenden Werkzeuge oder Fachlichen Services angepaßt werden. Im anderen Fall müßte das Benutzungsmodell so „locker“ formuliert werden, daß es keinen Nutzen mehr bringt oder es müßten schlimmstenfalls alle möglichen Zustände und Anwendungsfälle beschrieben werden, die sich aus der Kombination der möglichen Fachwerte eines Materials ergeben. Es ist also folglich besser, Vorgänge außerhalb der Materialien zu modellieren.

[98] vgl. Abschnitt 2.1.1.2 **Funktionsarbeitsplatz für eigenverantwortliche Expertentätigkeit**
[99] vgl. Abschnitt 2.1.2.3 **Automat**

Allerdings gibt es noch Fälle, bei denen Materialien, welche Vorgänge modellieren, zur Unterstützung *kooperativer Arbeit*[100] eingesetzt werden. Bei diesen Fällen kann ein Zustandsautomat eingesetzt werden, wenn man die oben genannten Nachteile in Kauf nimmt.
Bei kooperativer Arbeit lassen sich die Regeln und Arbeitsabläufe zu einem Vorgang zusammenfassen. Die Idee hierbei ist es, einen Vorgang zu vergegenständlichen und den Anwendern als Material zur Verfügung zu stellen. Ein solches Material wird als *Prozeßmuster*[101] bezeichnet. Beispiele für Prozeßmuster sind *Laufzettel* oder *Vorgangsmappen*. Beide spiegeln den aktuellen Stand der Bearbeitung in einem kooperativen Arbeitsablauf durch ihren Zustand wieder.
Ein Laufzettel ist ein Arbeitsgegenstand, der von einem Anwender für einen speziellen Vorgang erstellt wird. Dazu gehört die Festlegung, wer für welche Tätigkeiten zuständig ist, welche Reihenfolge zwischen den Arbeitsschritten besteht, welche Dokumente benötigt werden und welchen Bearbeitungszustand solche Dokumente haben.
Vorgangsmappen sind fachliche *Behälter*[102], die Materialien aufnehmen, verwalten, ordnen und herausgeben können. Sie unterstützen kooperative Arbeit, indem sie Vorgänge vergegenständlichen.
Bei einer Modellierung eines solchen Materials könnte der Vorgang einfach in ein Zustandsmodell übertragen und in einen Zustandsautomaten implementiert werden. Änderungen des Bearbeitungszustandes lassen sich über Zustandsübergänge abbilden. Mit einem Zustandsautomaten wäre sogar eine nebenläufige Bearbeitung der Materialien möglich. Betrachtet man beispielsweise eine Kreditakte, welche als Vorgangsmappe einen Bearbeitungsvorgang vergegenständlicht. Zu einer Kreditakte gehören verschiedene Dokumente wie zum Beispiel Kreditanträge. Im Zustandsautomaten könnte deren Verhalten und ihre Abhängigkeiten untereinander über orthogonale Komponenten modelliert werden, ohne daß sich dies im Material niederschlagen muß. Solche Materialien könnten dann erheblich einfacher und kontextunabhängiger modelliert werden, da der Bearbeitungszustand des Vorganges im Automaten abgebildet wird.
Allerdings müßte bei dieser Art der Modellierung eines kooperativen Arbeitsvorganges immer von dem Material, welches das Prozeßmuster umsetzt, kontrolliert werden, was mit den von ihm verwalteten Materialien geschieht. Eine Möglichkeit diese Kontrolle zu realisieren, wäre, nur Kopien der Materialien zur Bearbeitung herauszugeben und diese Kopien erst dann als neue Originale zu akzeptieren, wenn die an ihnen vorgenommenen Änderungen dem Zustandsmodell des Bearbeitungsvorganges entsprechen. Eine andere Möglichkeit wäre, die Manipulation der Materialien nur über die Vorgangsmappe bzw. den Laufzettel zuzulassen. Dies würde allerdings dem Charakter solcher Materialien widersprechen, denn welche Vorgangsmappe ermöglicht die Bearbeitung der in ihr enthaltenen Gegenstände. In diesem Fall wäre es angebracht, statt eines Materials einen Fachlichen Service zur Vorgangsbearbeitung zu verwenden, da durch diesem eine Form der Bearbeitung, wie sie oben beschrieben wurde, möglich ist.

---

[100] **Definition: Kooperative Arbeit**
Kooperative Arbeit ist Arbeit, bei der verschiedene Personen mit zwischen ihnen ausgehandelten Konventionen zur Koordination ein Ergebnis anstreben, das nur gemeinsam erreicht werden kann.
aus [Gryzcan 1996], Seite 223

[101] **Definition: Prozeßmuster**
Ein Prozeßmuster ist ein gemeinsames Material zur Vergegenständlichung eines kooperativen Arbeitsprozesses. Durch das Prozeßmuster werden Verantwortlichkeiten von Personen oder Rollenträgern und Tätigkeiten festgelegt. Ein Prozeßmuster besteht aus der Angabe der Abhängigkeiten von und zwischen Tätigkeiten, die bei der kooperativen Arbeit zu erledigen sind und dazu notwendigen Dokumente.
aus [Gryczan 1996], Seite 225

[102] vgl. **Definition: Behälter** (siehe [Züllighoven 1998], Seite 89)

6.3 Zusammenfassung

Als Zusammenfassung dieses Kapitels läßt sich sagen, daß neben dem, in den vorigen Kapiteln beschriebenen, Selbstbedienungsautomaten für Webtop-Anwendungen, besonders die Fachlichen Services von einer Verwendung von Zustandsautomaten profitieren würden. Zum einen lassen sich mit Zustandsautomaten Fachlichen Services miteinander synchronisieren und zum anderen können die Zustandsmodelle der Services durch einen Zustandsautomaten realisiert werden.
Für Werkzeuge und Materialien eignen sich Zustandsautomaten nur in einigen wenigen Fällen und auch dann sind mit der Verwendung wesentliche Einschränkungen verbunden. Im Zusammenhang mit Automaten lassen sich Zustandsautomaten nur dann einsetzen, wenn die Automaten als Teil eines Zustandsüberganges angesehen werden.

Kapitel 7: Zusammenfassung der Arbeit

In diesem Kapitel werden die wesentlichen Aspekte des in dieser Arbeit entworfenen Konzeptes zur Realisierung von Zustandsautomaten zur Vorgangssteuerung zusammengefaßt.
In der Motivation zu dieser Arbeit wurde ausgeführt, daß es bei der Entwicklung von anwendungsorientierter Softwaresysteme nach dem WAM-Ansatz unter bestimmten fachlichen Umständen notwendig ist, eine Vorgangssteuerung in einem System zu realisieren.
Anschließend wurde ein grundlegender Überblick über den WAM-Ansatz und seine für diese Arbeit relevanten Konzepte gegeben, um das Verständnis der nachfolgenden Kapitel zu erleichtern. Des weiteren wurde das JWAM-Rahmenwerk als technisches Umfeld eingeführt.
Hierauf aufbauend wurden Web-Anwendungen als Anwendungskontext für Vorgangssteuerungen eingeführt. Zunächst wurden die generellen Probleme bei der Konstruktion von Webtop-Anwendungen analysiert. Bei der Einordnung von ausgewählten Anwendungsbeispielen für Webtop-Anwendungen in den WAM-Ansatz wurde gezeigt, daß es sich bei diesen um Selbstbedienungsautomaten handelt. Bei Selbstbedienungsautomaten ist es aus fachlichen Gründen, wie der Benutzerführung, notwendig, Vorgänge durch Zustandsmodelle abzubilden.
Anschließend wurde beispielhaft ein Selbstbedienungsautomat unter Verwendung der Konstruktionsprinzipien des WAM-Ansatzes erstellt und die Aufgaben eines Zustandsautomaten spezifiziert. Der Zustandsautomat hat die Aufgabe, den internen Bearbeitungszustand einer oder mehrerer Komponenten zu kapseln und bei allen Arbeitsschritten deren fachliche Konsistenz zu wahren[103].
Als Grundlage für den Entwurf und die Spezifikation von Zustandsautomaten wurden Statecharts ausgewählt. Diese eignen sich gut als formales Beschreibungsmittel für das fachliche Benutzungsmodell von Zustandsautomaten, da sie durch ihre Möglichkeit zur Bildung von Hierarchien und die Verwendung von Orthogonalität knappe, ausdrucksmächtige und eindeutige Beschreibungen ermöglichen. Sie sind darüber hinaus verhältnismäßig leicht verständlich, einfach zu verfeinern und zu neuen Systemen zu kombinieren.
Zur Modellierung des Benutzungsmodells wurde ein Zustandsautomat konstruiert, welcher die Verwendung Fachlicher Services kapselt und somit die Einhaltung einer fachlich korrekten Verwendung im Rahmen des gesamten Arbeitskontextes garantiert. Diese Kapselung bzw. Gruppierung ermöglicht die Synchronisation der Fachlichen Services untereinander. Hierdurch ist es möglich kleinere Fachliche Services zu bauen, da der Zustandsautomat den Zusammenhang herstellt und für die Konsistenz des Systems sorgt.
Bei dem Zustandsautomaten handelt es sich um einen rein reaktiven, hierarchischen Automaten, welcher sich streng deterministisch verhält. An dieser Stelle soll betont werden, daß es sich nicht um eine genaue Implementation von Statecharts handelt, sondern daß der Zustandsautomat auf einer Beschreibung von Benutzungsmodellen mittels Statecharts beruht und ein ähnliches Verhalten wie diese zeigt. Ähnliches Verhalten bedeutet in diesem Fall, daß die Semantik des hier vorliegenden Automaten denen der Statecharts zwar ähnlich ist, sich aber in einigen Punkten von diesen unterscheidet. Da der Automat hierarchisch aufgebaut ist, kann er leicht erweitert oder verfeinert werden.
Des weiteren ist in dem Automaten eine explizite Trennung von Zustandsmodell, aktuellem Zustand und Zustandsübergängen realisiert worden. Am Zustandsautomaten wird über den Zustand immer explizit gemacht, welche Zustandsübergänge möglich sind.

[103] vgl. Abschnitt 3.5 **Definition Zustandsautomat**

Neben dieser Kapselung wurden der Zustandsautomat und sein Zustandsmodell dazu verwendet, die Repräsentation der Anwendung im Browser zu steuern. Beim Dispatching wurde unter Benutzung des Zustandsautomaten aus der funktionalen Komponente die jeweilige Folgeseite aus Quell- und Zielkonfiguration sowie dem Ereignis bestimmt. Dadurch ist es einfach möglich, einen Systemzustand einer geeigneten Repräsentation zuzuordnen.
Zum Abschluß der Arbeit wurden weitere Anwendungsmöglichkeiten für Zustandsautomaten diskutiert. Hierfür wurden zunächst verschiedene Kriterien für eine Verwendung erarbeitet und anschließend auf verschiedene Konzepte des WAM-Ansatzes angewandt. Hierbei hat sich gezeigt, daß sich Zustandsautomaten besonders für die Verwendung im Umgang mit Fachlichen Services eignen.
Abschließend läßt sich sagen, daß der Zustandsautomat und seine Komponenten ein allgemein gültiges Konzept für Zustände, Zustandshierarchien, Konfigurationen und Zustandsübergänge realisieren. Die Implementationsarbeit beim Einsatz von Zustandsautomaten besteht lediglich darin, den konkreten Zustandsautomaten und die notwendigen Zustandsübergänge für den jeweiligen Anwendungsfall anzupassen. Ansonsten müssen nur noch zu den fachlichen Zuständen bzw. Konfigurationen entsprechende Objekte der zugehörigen Zustands- bzw. Konfigurationsklassen erzeugt werden.

7.1 Offene Fragen und Probleme

An dieser Stelle sollen Probleme und Fragen, die sich im Verlauf der Arbeit ergeben haben, geschildert werden.
In dieser Arbeit wurde von der Möglichkeit zur Verwendung von Vererbung nur in wenigen Fällen Gebrauch gemacht. Dies geschah immer nur dann, wenn die allgemeingültigen Eigenschaften einer abstrakten Basisklasse in einer Subklasse für einen konkreten Anwendungsfall verfeinert wurden. Nun stellt sich aber die Frage, welche Konsequenzen es hätte, wenn eine Subklasse, welche einen konkreten Zustandsübergang oder einen Zustandsautomaten implementiert, wiederum durch eine Subklasse verfeinert wird. Könnte man hier Vererbung einsetzen, hätte man die Möglichkeit, auf einfache Weise bestimmte Vorgänge eines Anwendungsbereiches für einen anderen zu adaptieren, ohne diese völlig neu entwerfen zu müssen. Da es aber bei den Statecharts kein Konzept für Vererbung gibt, müßte hierfür ein eigenes Konzept erarbeitet werden. Bei der Verwendung von Vererbung würde sich allerdings die Frage stellen, ob sich die Aussagen über das Verhalten einer Basisklasse auf die jeweilige Subklasse übertragen läßt, oder ob das Verhalten bedingt durch eventuelle Verfeinerungen ein völlig anderes wäre.
Bei der Erstellung von Zustandsdiagrammen fiel immer wieder auf, daß diese zwar recht einfach anzufertigen und zu verstehen sind, aber das Testen auf syntaktische und semantische Korrektheit verhältnismäßig viel Zeit in Anspruch nimmt. Solch inkonsistentes Verhalten mußte in dieser Arbeit durch entsprechende Überlegungen beim Entwurf gefunden und korrigiert werden. In diesem Zusammenhang sollte über Möglichkeiten des automatisierten Testens von Zustandsmodellen nachgedacht werden. Hierdurch könnte zum Beispiel ein zyklisches Schalten von Zustandsübergängen einfacher erkannt werden.
Auch sollte über die Erstellung eines Editors zum Entwurf und zur Realisierung von Zustandsmodellen nachgedacht werden.

Literaturverzeichnis

[Bäumer 1997] D. Bäumer: *Softwarearchitekturen für die rahmenwerksbasierte Konstruktion großer Anwendungssysteme*, Dissertationsschrift, Universität Hamburg – Fachbereich Informatik, 1997.

[Berners-Lee 1989] T. Berners-Lee: *Information Management: A Proposal*, http://www.w3.org/History/1989/proposal.html, 1989.

[Berners-Lee 1994] T. Berners-Lee, L. Masinter, M. McCahill: *RFC 1738: Uniform Resource Locators (URL)*, http://www.w3.org/Addressing/, 1994.

[Bleek 1999a] W.-G. Bleek, G. Gryczan, C. Lilienthal, M. Lippert, S. Roock, W. Strunk, H. Wolf, H. Züllighoven: *Frameworkbasierte Anwendungsentwicklung (Teil 2): Die Konstruktion interaktiver Anwendungen*, OBJEKTspektrum 2/99, März/April 1999.

[Bleek 1999b] W.-G. Bleek, M. Lippert, S. Roock, W. Strunk, H. Züllighoven: *Frameworkbasierte Anwendungsentwicklung (Teil 3): Die Anbindung von Benutzungsoberflächen und Entwicklungsumgebungen an Frameworks*, OBJEKTspektrum 3/99, Mai/Juni 1999.

[Bleek 1999c] W.-G. Bleek, C. Lilienthal, H. Züllighoven**:** *Frameworkbasierte Anwendungsentwicklung(Teil 4): Fachwerte*, OBJEKTspektrum 5/99, September/Oktober 1999.

[Booch 1999] G. Booch, J. Rumbaugh, I. Jacobsen: *The Unified Modelling Language User Guide*, Addison Wesley, 1999.

[Chappel 1996] D. Chappell: *Understanding ActiveX and Ole*, Microsoft Press, September 1996.

[CSS 1998] W3C: *CSS-Style-Sheets, Specification - Version 2.0,* http://www.w3.org/TR/PR-CSS2, 1998.

[Cookies 2000] Netscape Communications Corporation: *Persistent client state HTTP cookies*, http://www.netscape.com/newsref/std/cookie_spec.html, Dezember 2000.

[Dyson 1998] D. Dyson, B. Anderson: *State Patterns*, in R. Martin, D. Riehle, F. Buschmann: *Pattern Languages of Design 3*, Addison Wesley Longman Inc, 1998.

[Fielding 1997] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, T. Berners-Lee: *Hypertext Transfer Protocol, HTTP 1.1. , RFC 2068*, http://www.cis.ohio-state.edu/rfc/ rfc2068.html, 1997.

[Flanagan 1997] D. Flanagan: *Java in a Nutshell, 2nd Edition*, O'Reilly, 1997.

[Flanagan 1998] D. Flanagan: *JavaScript: The Definitive Guide*, O'Reilly, 1998.

[Frei 1996] F. Frei, M. Hugentobler, A. Alioth, W. Duell, L. Ruch: *Die kompetente Organisation*, 2. Auflage, Zürich: vdf Hochschulverlag der ETH Zürich, 1996.

[Gamma 1996] E. Gamma, R. Helm, R. Johnson, J. Vlissides: *Design Patterns - Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1996.

[Gryczan 1996] G. Gryczan: *Prozeßmuster zur Unterstützung kooperativer Tätigkeit* , Dt. Universitätsverlag, 1996.

[Gryczan 1999] G. Gryczan, C. Lilienthal, M. Lippert, S. Roock, H. Wolf, H. Züllighoven: *Frameworkbasierte Anwendungsentwicklung (Teil 1)*, OBJEKTspektrum 1/99, Januar/Februar 1999.

[Gryczan 2000a] G. Gryczan, S. Roock, H. Wolf, H. Züllighoven: Guido Gryczan, Andreas Havenstein, Stefan Roock, Ingrid Wetzel, Heinz Züllighoven: *Frameworkbasierte Anwendungsentwicklung (Teil 5): Unterstützung von Kooperation mit persistenten fachlichen Behältern*, OBJEKTspektrum 1/2000, Januar/Februar 2000.

[Gryczan 2000b] G. Gryczan, S. Roock, H. Wolf, H. Züllighoven: *Framework-basierte Anwendungsentwicklung (Teil 6): Frameworkentwicklung und -einsatz organisieren*. OBJEKTspektrum 2/2000, März/April 2000.

[Harel 1987a] D. Harel: *Statecharts: A Visual Formalism for Complex Systems*, Science Computer Program, Vol. 8, 1987.

[Harel 1987b] D. Harel, A. Pnueli, J. P. Schmidt, R. Sherman: *On the formal semantics of statecharts,* Proceedings of the Symposium on Logic in Computer Science, IEEE Computer Society Press, 1987.

[Harel 1988] D. Harel: *On Visual Formalism*, Communications of the ACM, No.5, 1988.

[Harel 1996] D. Harel, A. Naamad. *The statemate semantics of statecharts*, ACM Transactions on Software Engineering and Methodology, October 1996.

[Harel 1998] D. Harel, M. Politi: *Modelling Reactive Systems with Statecharts: The Statemate Approach*, McGraw-Hill, 1998.

[Havenstein 1999] A. Havenstein: *Unterstützung Kooperativer Arbeit durch eine Software-Registratur*, Diplomarbeit, Universität Hamburg – Fachbereich Informatik, 1999.

[Homer 1999] A. Homer, D. Sussman, B. Francis, G. Reilly, D. Esposito, A.Chiarelli, B. Kropog, C. McQueen, G. Nolan, S. Robinson, J. Schenken, K. Tegel: *Professional Active Server Pages 3.0*, Wrox Press, 1999.

[Hopcraft 1994] J. E. Hopcraft, J. Ullman: *Einführung in die Automatentheorie, Formale Sprachen und Komplexitätstheorie*, Addisson-Wesley (Deutschland), 1994.

[HTML 1999] W3C: *HTML 4.01 Specification*, http://www.w3.org/TR/REC-html40, 1999.

[Java 2000] JavaSoft: *Java™ 2 SDK, Standard Edition v. 1.3* , http://www.javasoft.com/j2se/1.3/, Dezember 2000.

[JSP 2000] JavaSoft: *Java Server Pages – Spezification 1.2* , http://www.javasoft.com/products/jsp, Dezember 2000.

[JWAM 2001] Apcon WPS: *JWAM-Framework*, http://www.jwam.de, Februar 2001.

[Laue&Lietke 1998] A. Laue, M. Lietke: *Eine Einführung in Statecharts vor dem Hintergrund der objektorientierten Anwendungsentwicklung* , Studienarbeit, Universität Hamburg – Fachbereich Informatik, 1998.

[Lippert 1999] M. Lippert: *Die Desktop-Metapher in Systemen nach dem Werkzeug- und Material-Ansatz*, Diplomarbeit, Universität Hamburg – Fachbereich Informatik, 1999.

[Meyer 1997] B. Meyer: *Object-Oriented Software Construction*, New York, London, Prentice-Hall, 2nd Edition, 1997.

[Müller 1999] K. Müller: *Konzeption und Umsetzung eines Fachwertkonzepts*, Studienarbeit, Universität Hamburg – Fachbereich Informatik, 1999.

[Münz 1999] S. Münz, W. Nefzger: *HTML 4.0 Handbuch, 3. Auflage*, Franzis` Verlag (München), 1999.

[Otto&Schuler 2000] M. Otto, N. Schuler: *Fachliche Services: Geschäftslogik als Dienstleistung für verschiedene Benutzungsschnittstellen-Typen* , Diplomarbeit, Universität Hamburg – Fachbereich Informatik, 2000.

[Pnueli 1991] A. Pnueli, M. Shalev: *What is in a step: On the semantics of statecharts*, in T. Ito, A. R. Meyer: *Theoretical Aspects of Computer Software*, Nr. 526 of Lecture Notes in Computer Science, Springer-Verlag, Berlin Heidelberg New York, 1991.

[Ran 1996] A. Ran: *MOODS: Models for Object-Oriented Design of State* , in J. M. Vlissides, J. O. Coplien, N. L. Kerth*: Pattern Languages of Design 2*, Addison Wesely Longman Inc, 1996.

[Roock&Wolf 1998] S. Roock, H. Wolf: *Die Raummetapher zur Entwicklung kooperationsunterstützender Softwaresysteme für Organisationen*, Diplomarbeit, Universität Hamburg – Fachbereich Informatik, 1998.

[Roßbach 2000] P. Roßbach, H. Schreiber: *Java Server und Servlets: Portierbare Web-Applikationen effizient entwickeln* , 2. Auflage, Addison Wesley Verlag, München, 2000.

[Servlets 2000] JavaSoft: *Servlet – Specification 2.3,* http://www.javasoft.com/products/servlets, Dezember 2000.

[Szyperski 1997] C. Szyperski: *Component Software: Beyond Object-Oriented Programming*, Addison Wesley, Reading, 1997.

[UML 1998] UML Ressource Center: http://www.rationel.com/uml/documentation.html, 1998.

[von der Beck 1994] M. von der Beck: *A comparison of statecharts variants*, in *Formal Techniques in Realtime and Fault-Tolerant Systems*, Springer Verlag, Berlin Heidelberg New York, 1994.

[Yacoub 1998] S. M. Yacoub, H. Ammar: *A Pattern Language of Statecharts*, Abstract, Pattern Languages of Programs Conference (PloP-98), 1998.

[Züllighoven 1998] H. Züllighoven, D. Bäumer, W.-G. Bleek, U. Bürkle, I. Dörre, M. Franz, G. Gryczan, J. Hess, R. Knoll, A. Krabbel, C. Lilienthal, D. Megert, D. Riehle, P. Rieth, S. Roock, W. Siebirsky, T. Slotos, W. Strunk, D. Weske, I. Wetzel, F. Wiegand, H. Wolf, M. Wulf : *Das objektorientierte Konstruktionshandbuch*, dpunkt.verlag, 1998.

Abbildungsverzeichnis