Lösungsskizze 7: Dynamische Modellierung mit UML

Inhaltsverzeichnis

1 Einleitung

1.1 Übersicht

Ehe ich Lösungen für die einzelnen Aufgaben in Kurzform skizziere, möchte ich einige Bemerkungen zu Notation und Bedeutung (Interpretation) der einzelnen Diagrammtypen machen. Letztere können zumindest teilweise als Anleitung für ein System zur Erstellung von Diagrammen verstanden werden.

Es handelt sich bei diesen Bemerkungen nicht um das in den Anmerkungen zur Aufgabenstellung versprochene Tutorial zur Erstellung von Zustandsdiagrammen, das nach meinem Dafürhalten so dringend nötig wäre: Zwar werde ich (in informeller Weise) die offenen Probleme ansprechen und andeuten, in welcher Richtung meines Erachtens eine Lösung zu suchen ist, jedoch sind diese Ausführungen sicher nicht ausreichend und benötigen eine präzise formale Fundierung.

1.2 Zur Bedeutung von Zustandsdiagrammen

1.2.1 Was ist ein Zustand?

Die Terminologie der Zustandsdiagramme kommt aus der Theorie der endlichen Automaten: Dort stehen Knoten für einen Zustand des Automaten, die Kanten für Übergänge, die beim Eintreffen von äußeren *Ereignissen* oder *Eingabesymbolen* vollzogen werden.

Der unmittelbare Wert von Zustandsdiagrammen in Entwurf und Systemanalyse liegt darin, die vom momentanen Zustand abhängige *Reaktion* eines Systems oder von Systemteilen in anschaulicher (d. h. für den Kunden verständlicher) Weise sichtbar zu machen. Jedoch kann der Zustandsbegriff aus der Theorie der endlichen Automaten nicht ohne Änderungen übernommen werden: Zwar ist jedes System prinzipiell ein endlicher Automat, aber tendenziell eher mit vielen Zuständen, während man im Diagramm der Übersichtlichkeit halber ja eher wenige Zustände zu sehen wünscht (s. Teillösung A.1).

Der Ausweg besteht darin, dass man die Knoten eines Diagramms als *Äquivalenzklassen* von Zuständen auffasst. Das Zustandsdiagramm (bzw. der dadurch beschriebene Automat) ist also eine *Projektion* des sehr viel komplizierteren Automaten mit sehr vielen Zuständen, der das beschriebene System „in Wahrheit“ ist. Unter Umständen sind mehrere solcher Projektionen nötig, um das System mit zufriedenstellender Genauigkeit zu beschreiben. Gleichzeitig ist es so, dass ein Übergang fast immer einer Sequenz von Verarbeitungszuständen entspricht (und nicht, wie in der Theorie der endlichen Automaten, als ein praktisch instantanes Kippen von einem Zustand in den anderen beschrieben wird).

1.2.2 Spezifikation von Übergängen

Behandeln wir zuerst Systeme, die keine eigene innere Dynamik haben, sondern quasi passiv auf Ereignisse von außen warten, welche dann eine Verarbeitungssequenz auslösen, die in einem Endzustand in der einen oder anderen Zustandsmenge endet, in dem dann wieder passiv auf das nächste Ereignis gewartet wird. Betrachten wir hier die Übergänge, die in der angesprochenen Form von Zustandsmodelle auftreten. Diese Übergänge werden immer durch ein als systemextern angenommenes Ereignis ausgelöst, es kann sich dabei um Interrupts, Tastendrücke, eintreffende Serverbotschaften oder Methodenaufrufe, praktisch um jede Art von Signal oder Eingabe handeln.

Die Ereignisse können also Daten mit sich tragen. Diese können wir, wenn nötig hinter dem Ereignissymbol in Klammern notieren (siehe Teillösung A.1): FUEGE-EIN(a).

In deterministischen Systemen darf die Reaktion eines Systems, d. h. ob ein Übergang stattfindet, und der Endzustand eines Überganges, nur von seinem Zustand und vom Ereignis (seiner Art und den Daten, die es mit sich trägt) abhängen. Entspricht ein Knoten nur genau einem „realen“ Zustand, so lässt sich ein Übergang immer charakterisieren durch Anfangs- und Endzustand (Einzeichnen der Kante in das Diagramm), sowie die Ereignisse (Eingabemenge), welche den Übergang auslöst. Wenn jedoch, wie in unserem Fall, ein Knoten gleich einer Äquivalenzklasse von Zuständen entspricht, erweist sich die Situation als geringfügig komplizierter: Jetzt kann es durchaus sein, dass nicht bei allen Zuständen der Äquivalenzklasse die gleiche Reaktion erfolgt. Puristen würden nun sicher gerne die Äquivalenzklasse aufspalten, dies ist aber für den vorgenannten Zweck der Projektion auf eine leichter zu überblickende Situation eher kontraproduktiv, und es sei nur darauf hingewiesen, dass dies (die Aufspaltung der Äquivalenzklasse) auch nicht immer sinnvoll möglich ist, wie sich der geneigte Leser anhand der Lösungsskizze von Teilaufgabe A.1 vergewissern kann.

Es ist also nötig eine Bedingung hinzuzufügen, unter der der Übergang erfolgt, was nichts anderes ist, als eine Definition der Teilmenge der Äquivalenzklasse des Knotens, aus der heraus der Übergang erfolgt. Wir notieren diese *guard condition* in einer eckigen Klammer hinter der Ereignismenge (siehe Teillösung A.1):

$$\text{ENTFERNE}(a)[\mathit{artikelanzahl} = 1] \,.$$

Ebenso kann die Bedingung eine Untermenge von Ereignissen spezifizieren. Es sei hier darauf aufmerksam gemacht, dass es hier ein Vollständigkeitskriterium gibt: Spezifiziert das Diagramm die Reaktion auf alle möglichen Ereignisse für alle Zustände der Äquivalenzklasse? Unglücklicherweise ist die UML-Definition, soweit ich das nachvollziehen kann, recht vage im Bezug auf die Bedeutung unspezifizierter Fälle – bedeuten sie, dass im Sinne einer Vorbedingung solche Situationen nicht auftreten können und dürfen, oder dass das System solche Ereignisse ignoriert, oder aber, dass es diese sehr wohl bearbeitet, es aber nur zu Zustandsänderungen innerhalb der Äquivalenzklasse kommt, die für die Übergänge nicht signifikant sind?

Da diese Dinge derart unklar sind, empfehle ich, für die beiden letzteren Fälle explizit entsprechende Übergänge einzuzeichnen, in denen die Endknoten identisch sind mit den Ausgangsknoten, oder gewisse Verarbeitungsprozesse pauschal in Fußnoten zuzusichern (etwa im diesem Stil: „In den Zuständen X, Y und Z werden Serverbotschaften immer bearbeitet und zwar in der in Dokument 101 beschriebenen Weise.“).

Übergänge charakterisieren Reaktionen des beschriebenen Systems. Diese bestehen aber nicht nur aus Zustandsänderungen, sondern auch aus Ausgaben, allgemeiner *Effekten*, die das System auf seine Um- und Außenwelt ausübt. Deshalb ist es üblich an den Übergängen *Operationen* zu notieren, die im Zuge des Überganges ausgeführt werden (siehe Skizze zur Teilaufgabe B.4): $\text{ABBRUCH}/\mathit{widerrufe\text{-}anfrage}()$.

Mit diesen Operationenannotationen ist es auch möglich, eine bestimmte Teilmenge der Äquivalenzklasse des Endknotens als Ziel des Übergangs zu identifizieren, indem man eine Operation angibt, welche einen Teil des Systemzustandes verändert (A.1):

$$\text{ENTFERNE}(a)/\text{entferne}(a) \,.$$

Hier ist die Methode leider unvollständig: Eigentlich würde man es her wohl in den meisten Fällen vorziehen, eine Nachbedingung, welche den möglichen Endzustand einschränkt, aufzuschreiben, beispielsweise $\text{ENTFERNE}(a)/\textbf{Post}\colon a \notin \{\mathit{Posten}\}$, aber dies ist nicht vorgesehen.

Nachdem, wie bereits erwähnt, Übergänge sowieso *Verarbeitungssequenzen* entsprechen, ist es meines Erachtens auch sinnvoll, die Annotationen an den Übergängen, d. h. Ereignismengen, Bedingung und Operation bzw. Effekt auch mehrfach anzureihen (siehe

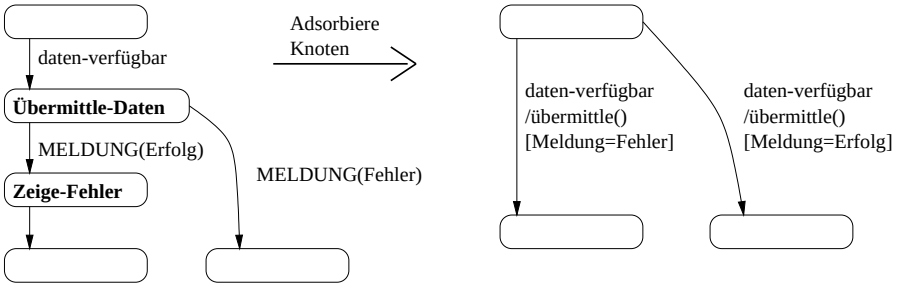
Lösungsskizze B.3). Der Nutzen besteht vor allem darin (was z. B. B.3 illustriert), dass die Einführung sinnloser Pseudoknoten, die nur als Ankerpunkt für Bedingungen dienen, dadurch vermieden werden kann. Man muss diese Sequenzen aus Ereignissen und Bedingungen wohl so lesen, dass es sich dabei um lineare Ablaufszenarien handelt: Entweder treffen die entlang des Übergangs notierten Ereignismengen und Bedingungen für einen gegebenen Fall zu, oder wir müssen eine andere Kante des Graphen suchen, deren Ablaufsequenz bei gegebenem Anfangszustand einen Übergang zu beschreiben vermag.

1.2.3 Aktive Systeme

Betrachten wir nun *aktive* Systeme. Damit bezeichne ich hier Systeme, die eine innere Dynamik besitzen, d. h. deren Zustand sich ständig ändert. Praktisch alle zeitgenössischen datenverarbeitenden Systeme sind in Wahrheit von dieser Art (wenn sie nichts zu tun haben, iterieren sie in der *idle loop*), die in vielen Fällen unterstellte Passivität (Warten auf Eingabe) ist bereits eine Idealisierung. Auch aktive Systeme lassen sich durch Zustandsdiagramme beschreiben. Am eben erläuterten Konzept ändert sich prinzipiell nichts, jedoch muss nun eine neue Art von Ereignissen berücksichtigt werden: Die *inneren* Ereignisse, d. h. Bedingungen, die durch das System selbst hervorgebracht werden.

Unter *inneren Ereignissen* verstehe ich Vorfälle, die nicht von außen auf das System einwirken, sondern mit dem Erreichen eines bestimmten Verarbeitungszustands durch das stetig fortarbeitende System identifiziert werden können. In der Regel wird dies sogar eine bestimmte Teilmenge innerhalb der Äquivalenzklasse sein, welche einem Knoten zugehört. Das System ändert, wie erwähnt, ja ständig „spontan“ seinen Zustand (zählt beispielsweise einen Timer hoch). Bei Eintritt einer bestimmten Bedingung („Timer überschreitet einen bestimmten Zählerstand“) – diese Bedingung entspricht einer Zustandsteilmenge – wird eine Verarbeitungssequenz ausgelöst, die in einem Übergang in eine andere Äquivalenzklasse endet („Abbruch des Vorgangs, Timeout“). Die *inneren* Ereignisse sind also der Eintritt gewissen Bedingungen an den „realen“ Zustand (den Mikrozustand) eines dynamischen Systems. Sie werden wie die äußeren Ereignisse an der Übergangskante notiert, müssen aber, wie erläutert, als Aussage über den Mikrozustand verstanden werden (B.2): TIMEOUT/*widerrufe-belegung*().

Da nun im allgemeinsten Fall die Knoten des Zustandsdiagrammes für Mengen von Zuständen stehen, die Kanten für Folgen von Zuständen, und sowohl während des Übergangs als auch während des Aufenthaltes in den Knoten ständig Verarbeitungsprozesse ablaufen (d. h. Operationen ausgeführt werden, Zustände sich ändern und Effekte auf die Außenwelt ausgeübt werden), ist es nicht *immer* eindeutig, wann ein Verarbeitungsprozess (wie etwa „Übermittlung der Daten an den Server“) durch einen Knoten, und wann durch eine Übergangskante modelliert werden soll. Ich halte es für recht wahrscheinlich, dass sich dafür ohne eine präzise formale Fundierung dieses Diagrammtyps keine exakten Regeln angeben lassen. Man kann sich jedoch von dem grundsätzlichen Gedanken leiten lassen, dass Knoten für Abläufe mit einer gewissen Dauer stehen, die von äußeren Ereignissen unterbrochen werden können, während Übergänge ununterbrechbar und mit einer gewissen Zwangsläufigkeit ablaufend gedacht werden müssen. Es folgt daraus, dass unterbrechbare Prozesse immer durch Knoten dargestellt werden *müssen* (Beispielsweise, wenn die „Übermittlung von Daten an den Server“ durch eine Benutzeraktion abgebrochen werden kann). Andererseits können Knoten, aus denen Übergänge nur durch synchrone äußere Ereignisse ausgelöst werden, immer in die Kanten adsorbiert werden (beispielsweise, wenn die „Übermittlung von Daten an den Server“ nicht unterbrechbar ist, jedoch die weitere Verarbeitung (der Zustand nach vollendeter Übermittlung) von einer Statusmeldung des Servers abhängt).



Es sei schließlich noch erwähnt, dass der Unterscheidung zwischen passiven und aktiven Systemen, die hier vor allem aus didaktischen Gründen erfolgte, eine gewisse Beliebigkeit innewohnt: Aktive Systeme können als passive Systeme mit einer extern gedachten „Uhr“ als Ereignisquelle modelliert werden, andererseits sind passive Systeme als Idealisierungen aktiver Systeme denkbar, deren Verarbeitungsprozesse in den interessierenden Zuständen letzten Endes immer auf Eingaben warten. Beide Perspektiven sind also in einem gewissen Maß gegeneinander austauschbar, welche schließlich gewählt wird, hängt – ebenso wie die Frage auf welchen endlichen Automaten man projizieren wird – von pragmatischen Gesichtspunkten ab, d. h. welche Systemeigenschaften vorrangig sichtbar gemacht werden sollen.

1.2.4 Ein- und Ausgänge

Bis hierher wurde nicht darauf eingegangen, welche Bedeutung den Ein- und Ausgängen des Diagramms zuzuschreiben ist. Ich will dies nun nachholen. In den (qualitativ) billigeren Varianten der UML-Literatur werden beide häufig als die Erzeugung eines Objektes bzw. dessen Vernichtung interpretiert. Dass dies sicher zu kurz greift, erkennt man an folgender Überlegung: Ein echter OO-Entwurf gibt in der Regel übersichtliche Teilausschnitte des Systems wieder. Es mag sein, dass man in einer Projektion des Systems an einem gewissen Punkt das Interesse an einem Objekt verliert – was z. B. die Bearbeitung einer Bestellung betrifft, etwa dann, wenn die Warensendung an den Kunden ausgeliefert ist. Andererseits kann man kaum mit Sicherheit sagen, dass das Objekt an dieser Stelle bereits aus der Existenz tritt: Die Buchhaltung mag für Bilanzierungs- und Steuerzwecke durchaus noch für einige Jahre an dem Datensatz interessiert sein, so dass das betreffende Objekt solange in der Datenbank weiter existieren muss.

Es lässt sich also in einem gegebenen Vorgang kaum sagen, dass das Objekt an einer bestimmten Stelle zerstört wird, da es andere Prozesse und Vorgänge geben mag, die weiterhin Referenzen auf das Objekt halten. Dies muss denn auch als Grund gesehen werden, warum viele objektorientierte Sprachen mit einer *Garbage Collection* ausgerüstet sind, die die Objekte genau dann beseitigt, wenn sie nicht mehr erreichbar sind, also die Entscheidung, wann ein Objekt zerstört werden kann oder gar muss, vollkommen aus der Entscheidung der Entwickler (Entwerfer wie Implementatoren) nehmen.

In analoger Weise ist es auch schwierig, den Eingang eines Diagrammes immer als die Erzeugung eines Objektes zu interpretieren, da man möglicherweise nur einen späteren Ausschnitt aus dem Werdegang eines Objektes modellieren möchte, oder im gegebenen Kontext gar keine Aussage über den Entstehungsmechanismus oder die Herkunft eines Objektes treffen will. Die „traditionelle“ Interpretation der Ein- und Ausgänge von Zustandsdiagrammen ist also problematisch und auf jeden Fall mit Vorsicht zu betrachten. Es ist aber sicher, zu sagen, dass mit dem Eingang ins Diagramm ein Interesse an einem *bestimmten* Objekt, einer Konstellation von Objekten oder an einer Untermenge von Systemzuständen etabliert wird, und dass mit dem Übergang in den Ausgang des Diagramms dieses Interesse für den Zweck des modellierten Problemausschnittes als erloschen gelten muss.

1.2.5 Fragen der Notation

Zum Schluss der Ausführungen über Zustandsdiagramme seien noch einige Bemerkungen zur Notation angefügt: UML muss nach der Lage der Dinge eher als eine semiformale Methode gelten, wenn es auch möglich erscheint, Teile der UML mit entsprechendem Aufwand zu einer formalen Methode auszubauen. Als semiformale Methode teilt sie mit anderen semiformalen Methoden eine Eigenschaft, die zugleich Segen und Fluch bedeuten kann: Es ist möglich, Teile der Spezifikation zu verkürzen oder wegzulassen, was es zum einen ermöglicht, bestimmte Festlegungen erst in späten Phasen des Entwurfs zu treffen, was zum anderen aber auch heisst, dass es normalerweise fast unmöglich ist, sich der Konsistenz und Zuverlässigkeit eines Entwurfs auch systematisch zu versichern, da die entsprechenden Regeln nicht existieren.

Wie aus den vorangegangenen Ausführungen hervorgeht, müssten Knoten in Zustandsdiagrammen eigentlich mit einer Charakterisierung der betreffenden Äquivalenzklassen (in Form von Prädikaten über den Systemzustand) beschriftet sein, ebenso müssten die Ereignisse an den Übergängen eigentlich präzise charakterisierte Untermengen von vorher definierten Ereignisgrundmengen sein, und so weiter. Für all das hat man in der Praxis häufig noch nicht genug Informationen, stattdessen kommen (hoffentlich) aussagekräftige Kurzbeschriftungen in (fast) natürlicher Sprache zum Einsatz (siehe Teillösung B.3). Unter bestimmten Umständen kann dies durchaus als ein Pluspunkt zugunsten der Methode gezählt werden. Zur Präzisierung empfiehlt sich dann, dem Diagramm in späteren Phasen des Entwurfs State-, Transition- und Event-Dictionaries beizufügen, die die Bedeutung der Kurzbeschriftungen präzisieren[1].

Sofern man sich bei den Prädikaten, welche die den Knoten zugeordneten Zustandsmengen beschreiben, bzw. denen, die die Bedingungen der Übergänge angeben, für eine formale Notation entscheidet, muss eigentlich geklärt werden, welchen Geltungsbereich die dort verwendeten Bezeichner haben. Soweit mir bekannt existieren bezüglich dieser Geltungsbereiche keine standardisierten Regeln, aber es ist möglich, dass die Regeln der OCL (Object Constraint Language), soweit sie auf Zustandsdiagramme angewendet werden können, solche zu liefern vermögen.

1.3 Zur Bedeutung von Ablaufdiagrammen

1.3.1 Probleme

Ablaufdiagramme notieren in anschaulicher Weise *typische* Abläufe in Teilen eines Softwaresystems.

Die *OMG Unified Modeling Language Specification, Version 1.4* (UML 1.4, 2001) formuliert das so:

> The behavioral parts of UML describe the valid sequences of valid system instances that may occur as a result of both external and internal behavioral effects.

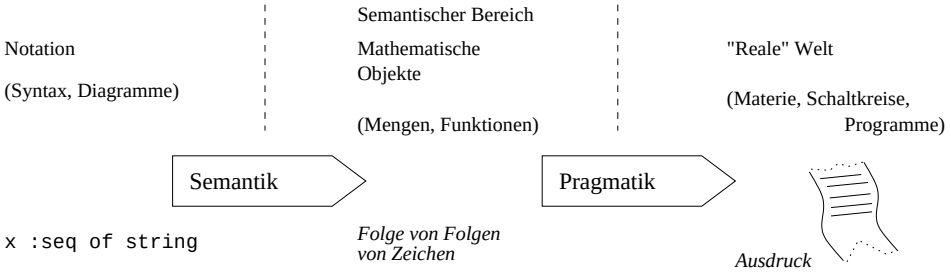
Dem wäre auch gleich hinzuzufügen, dass ein Ablaufdiagramm in der Regel nur einen Teil des Systems beschreiben kann, so dass die UML Spezifikation in dieser Hinsicht nicht ganz exakt ist.

Versucht man Ablaufdiagrammen eine exakte Bedeutung unterzulegen, so stößt man auf Schwierigkeiten – wie Widersprüche und Auslassungen in der Literatur, die sich auch oft

[1] In der UML gibt es diese Dictionaries nicht, man kann aber wie immer „Notizen“ für diesen Zweck verwenden.

auf den anschaulichen Gehalt der Diagramme zurückziehen[2]. Ich möchte deshalb kurz ausholen, um Fragen der Notation von prinzipiellen Fragen der Bedeutung (d. h. der semantischen Bereiche) zu trennen.

1.3.2 Allgemeine Bemerkungen zu Semantik und Pragmatik



Ich lege folgendes Modell zugrunde: Jede Spezifikationstechnik besitzt eine Notation, diese kann formal sein, etwa durch Syntax definiert, aus einer bestimmten Art von Diagrammen bestehen, oder auch informell in natürlicher Sprache über bestimmte Konzeptbereiche bestehen. Elemente aus einer Untermenge dieser Notation – nämlich die wohlgetypten und konsistenten Dokumente in dieser Notation – können als Aussagen über reale System angewandt werden, die dann für ein konkretes System entweder wahr sind (System entspricht der Spezifikation) oder falsch (System erfüllt die Spezifikation nicht). In der Regel geschieht dies in zwei Schritten: Im ersten werden den Dokumenten der Notation wohldefinierte, mathematische Objekte zugewiesen (z. B. Mengen oder Funktionen). Diese Zuweisung ist in der Informatik als *Semantik* der Notation bekannt. Im zweiten Schritt müssen die Objekte irgendwie mit den Objekten der realen Welt identifiziert werden (z. B. mit der Interpretation einer Folge als Ausgabe eines Druckers). Die meist ungeschriebenen, informellen Regeln, nach denen diese Identifikation erfolgt, ist als *Pragmatik* bekannt. Da hier eine Idealisierung der „realen“ Welt involviert wird, ist es unmöglich, diesen Schritt vollständig zu präzisieren. In der Regel bleibt die Pragmatik eine sich dynamisch entwickelnde Kultur, die mündlich oder schriftlich in natürlicher Sprache überliefert werden muss.

1.3.3 Der semantische Bereich

Wenn wir also hier über die Bedeutung von Ablaufdiagrammen sprechen wollten, müssen wir die Notation beschreiben, den semantischen Bereich festlegen, eine Zuordnung zwischen Notation und semantischen Objekten finden und eventuell einige Worte zur Pragmatik verlieren. Wir können dies allerdings im vorliegenden Fall nicht vollständig tun, da wir nicht frei sind, eine eigene Notation zu entwerfen, und in der UML die Sachlage alles andere als klar ist[3]. Stattdessen werde ich erläutern, was gesichert erscheint, und ansonsten auf die Variationen in der Literatur verweisen.

Was ein Ablaufdiagramm notiert, d. h. der *semantische Bereich* der Notation, ist im Wesentlichen eine *Menge* möglicher Abläufe, beschrieben durch Botschaften, die zwischen Objekten ausgetauscht werden. Im Bereich des *Model Checking* sind solche Mengen als *event traces* bekannt und werden dort exakter, aber etwas weniger anschaulich, mittels der universelleren *path expressions* notiert.

[2] Und diese Anschaulichkeit ist vollkommen unzureichend, sowohl für die Zwecke exakter Wissenschaft, als auch für die praktische Aufgabe, dunkle Punkte eines Entwurfes zwangsweise ins Licht zu rücken und technisch komplexe Streitfragen in Bezug auf einen Entwurf entscheidbar zu machen.

[3] Dies wird schon allein durch die stetige Folge von Veröffentlichungen belegt, in denen versucht wird, eine formale Semantik der UML-Diagramme zu entwerfen. Die in der UML-Spezifikation selbst gegebene Beschreibung stützt sich stark auf natürliche Sprache und darüberhinaus auf ein „selbst-gemachtes“ Objektmodell: Im besten Fall wird dadurch die UML mit Mitteln der UML beschrieben.

Man muss sich das so vorstellen, als ob man an dem laufenden Softwaresystem (z. B. mittels eines Debuggers) laufend die ausgetauschten Botschaften aufzeichnen würde. Sodann benötigt man zusätzlich eine Handhabe, aus der Trace des Gesamtsystems eine relevante Untermenge herauszulösen, z. B. alle Botschaften, die von einer Botschaft einer bestimmten Art (etwa *login*()) ausgelöst werden. Dies entspricht der Bildung eines Systemausschnitts. Wenn wir die so erhaltene reduzierte Trace in der durch das Ablaufdiagramm beschriebenen Menge möglicher Abläufe wiederfinden, dann gestattet das Ablaufdiagramm die von uns festgestellten Geschehnisse. Wenn nicht, dann muss unsere Spezifikation ein anderes Ablaufdiagramm enthalten, dass die empirisch oder im Gedankenexperiment erhaltene Folge von Ereignissen spezifiziert – oder wir müssen urteilen, dass sich das betrachtete System entgegen der Spezifikation verhält.

Zwei Dinge sind hier anzumerken: Zum einen müssen in der UML alle Ablaufdiagramme (die mit demselben Einstieg beginnen) additiv zusammengenommen werden: Ein einzelnes Diagramm würde beispielsweise nur den regulären Ablauf beschreiben, andere wiederum die Ausnahme- und Fehlerfälle. Ein einzelnes Ablaufdiagramm beschreibt, welches Geschehen auf jeden Fall als legitim gelten soll, ohne Aussagen zu machen über Geschehen, das in diesem Diagramm nicht enthalten ist – erst wenn sich dieses Geschehen in *keinem* der Ablaufdiagramme der gesamten Spezifikation des Systems finden lässt, kann gefolgert werden, dass es illegitim ist[4]. Zum zweiten benötigt man, wie bereits angesprochen, eine Handhabe, mit der man sagen kann, dass eine Teilgeschehen der beschriebenen Art beginnt. Diese Handhabe ist wohl mehr der Pragmatik zuzurechnen, man sollte sich bei jedem Ablaufdiagramm drüber im Klaren sein, wann und unter welchen Umständen das beschriebene Geschehen beginnt. Mit der Anwendung dieser Kriterien könnten dann die Objekte und Botschaften aus dem Ablaufdiagramm mit konkreten Objekten und Botschaften aus dem Gesamtsystem identifiziert werden. Es empfiehlt sich, diese Kriterien in natürlicher Sprache zum Ablaufdiagramm hinzuzufügen. Dies erübrigt sich allerdings, wenn das Kriterium in der Erzeugung bestimmter Klassen oder dem Auftreten aller Botschaften einer bestimmten Art besteht.

1.3.4 Notation und Semantik

Wenden wir uns nun der Notation und Semantik von Ablaufdiagrammen zu. Leider besteht in diesem Punkt in der von mir zu Rate gezogenen Literatur wenig Einigkeit, Exaktheit oder Verständlichkeit[5]. Ich werde nun kurz eine Übersicht über diese Literatur und ihre Aussagen geben, und dann einige (unvollständige) Empfehlungen zur Anfertigung von Ablaufdiagrammen.

Die UML Spezifikation *OMG Unified Modeling Language Specification, Version 1.4* (UML 1.4, 2001) sollte eigentlich die ultimative Quelle für UML-Notationsfragen sein. Leider handelt es sich um ein extrem hierarchisch aufgebautes Dokument, das objektorientierte Begriffe verwendet, um Syntax und Semantik von UML-Diagrammen zu definieren. Es ist also recht schwer, Antworten auf einzelne Detailfragen zu extrahieren, ohne das ge-

[4] Andererseits legt man Spezifikationen häufig so an, dass sie deterministisch sind. Dann hat man häufig schon anhand eines Teils der Ablaufdiagramme die Möglichkeit, bestimmte Abläufe als illegitim zu klassifizieren, nämlich dann, wenn man erkennt, dass ihre Legitimierung (etwa durch weitere Ablaufdiagramme) das System indeterministisch machen würde.

[5] Ich muss hier schon etwas verärgert bemerken, dass es ja eigentlich möglich sein sollte, diese Dinge in wenigen Seiten kurz und bündig abzuhandeln, unter Verzicht auf vage Einlassungen irgendwelcher Art. Ablaufdiagramme beschreiben Folgen von Ereignissen in einer ähnlichen Weise wie reguläre Ausdrücke Folgen von Symbolen beschreiben. Letzteres wird in einschlägigen Büchern (Aho u. a., 1986) auf wenigen Seiten (94-96) abgehandelt, dies sollte eigentlich auch für die Notation der UML Ablaufdiagramme möglich sein. Im vorliegenden Text bin ich leider nicht frei, von den Grundlagen her neu zu beginnen, sondern muss fragen, was in der UML vorliegt, eine Frage, die ich ohne vollständige Aufarbeitung der UML-Spezifikation (566 Seiten) und möglicherweise diverse Sekundärliteratur, wozu mir offen gestanden Neigung und Zeit abgehen, noch immer nicht vollständig beantworten kann.

samte Dokument aufzuarbeiten. Die dort angeführten Beispiele für Ablaufdiagramme illustrieren Fallunterscheidungen durch Aufspaltung der Methodenaufrufe und Annotation der Fallbedingungen. Auf die Bedeutung unvollständiger, d. h. nicht alle Fälle abdeckender Bedingungen wird nicht eingegangen[6]. Weiterhin fehlt die Illustration von Notationen zur Wiederholung, auch sind diese im laufenden Text nicht erwähnt. Es ist allerdings möglich, dass die Erklärungen von Wiederholungskonstrukten in allgemeinen Erläuterungen zu den Interaktionsdiagrammen (zu denen die Ablaufdiagramme in der UML gezählt werden) enthalten ist.

UML in a Nutshell (Alhir, 1998) illustriert die Anwendung von *guard conditions*, die hier so interpretiert werden, dass bei einem Nichtzutreffen der Bedingung die Folgeaktionen einfach unterbleiben, das Diagramm aber anwendbar bleibt[7]. Weiterhin werden zwei Notationen für die Wiederholung von Abläufen gezeigt, allerdings ohne die sonst übliche $*$-Notation. Diese Abweichung mag der Tatsache geschuldet sein, dass dieses Buch sich an der Version 1.1 der UML orientiert.

In *Fundamentals of Object-Oriented Design in UML* (Page-Jones, 2000) wird ein Beispiel für ein Ablaufdiagramm angeboten, das meines Erachtens schlicht falsch ist: Ohne Guard-Conditions werden verschiedene alternative, sich gegenseitig abschließende Abläufe zu einem Diagramm gemischt, ohne dass klar ist, welche Botschaft zu welchem Ablauf gehört[8].

Object-Oriented Methods: Principles & Practice hat sich in anderen Fragen der Objektorientierung als eine zuverlässige und vor allem präzise Quelle erwiesen. Leider ist Graham im Bezug auf Ablaufdiagramme wenig hilfreich, da er sie selbst kaum einsetzt. Wo er auf Ablaufdiagramme eingeht (Seite 265) befürwortet er eine Erweiterung im Sinne der Entwicklungsmethode *Catalysis*, in welcher Ablaufdiagramme zur hierarchischen Abstraktion von Use Cases eingesetzt werden, d. h. horizontale Linien (Ereignisse im Diagramm) bedeuten nicht nur den Austausch einzelner Botschaften, sondern möglicherweise komplette Transaktionen, die Beteiligten sind allgemein *Actors* bzw. geeignete Abstraktionen von Objektaggregaten, also nicht mehr notwendig einzelne Instanzen von Klassen.

Zusammenfassend muss man also sagen, dass die Ablaufdiagramme wohl – ich vermute, gerade *wegen* ihrer täuschende Anschaulichkeit – die Stiefkinder der objektorientierten Literatur sind. Dabei liegen die Problem allerdings vollkommen im Bereich der Notation, da die prinzipielle Bedeutung eines Ablaufdiagrammes klar ist: Es notiert (man könnte sagen: generiert) eine Menge von Event-Traces, die als legitimes Verhalten des spezifizierten Systems gelten sollen.

Bei einfachen Diagrammen gibt es hier keinen Raum für Missverständnisse aber bezüglich der Bedeutung von Guard-Conditions und Wiederholungsnotationen bleiben einige Fragen offen. Ich kann nicht für mich in Anspruch nehmen, diese aufzuklären – die letztendliche Referenz in diesen Fragen sollte eigentlich auch die UML Spezifikation (UML 1.4, 2001) sein die ich hier nicht aufarbeiten möchte. Stattdessen werde ich die offenen Fragen kurz andeuten und Vorschläge machen, wie diese Graubereiche in der Praxis umgangen wer-

[6] In der Tat ist das auf Seite 3-105 abgedruckte Diagramm unvollständig. Der Fall $x = 0$ ist nicht berücksichtigt, und es ist schlicht unklar, ob dieser Fall (a) nicht auftreten kann, (b) in einem anderen Diagramm spezifiziert werden wird, oder (c) schlicht vergessen wurde. Ich möchte Letzteres nicht ganz ausschließen, da das ganze Dokument nicht von bester Qualität ist, was schon damit beginnt, dass das Dokument die Version **1.4** der UML darlegt, der ziemlich umfangreiche Hinweis *GENERAL USE RESTRICTIONS* jedoch mit den Worten „The owners of the copyright in the UML specifications version **1.3** hereby grant you ... ".

[7] Die alternative Interpretation, die sich hier angeboten hätte, wäre, dass das Diagramm unter diesen Umständen (Nichtzutreffen der Guard-Condition) nicht anwendbar ist, also der konkrete Ablauf, dessen Legitimität bewertet werden soll, in einem anderen Diagramm gefunden werden muss.

[8] Um hier Gerechtigkeit walten zu lassen, sei erwähnt, dass Meilir Page-Jones, der Autor des genannten Buches, den genauen Ablauf des Geschehens durch Beifügung eines in Pseudo-Code skizzierten Algorithmus zu klären versucht. Offensichtlich sieht er das Ablaufdiagramm mehr als einen Katalog aller Botschaften, die zwischen zwei (oder mehreren?) Objekten ausgetauscht werden können, die genaue Abfolge müsste dann mit anderen Mitteln geklärt werden. Ich bin mir allerdings verhältnismäßig sicher, dass so etwas – trotz des Buchtitels – nicht im Sinne der UML ist.

den können. Für das in dieser Lösungsskizze wiedergegebene Ablaufdiagramm sind die Ausführungen allerdings kaum relevant, da es die in Frage stehenden Konstrukte kaum enthält.

1.3.5 Vermeidung von Grauzonen

Bei den Wiederholung notierenden Konstrukten muss der Geltungsbereich geklärt sein, also welcher Teil des Ablaufs unter welchen Bedingungen wie oft wiederholt wird. Wird die Wiederholung mit dem Stern „∗“ an einer Botschaft notiert, so schlage ich vor, dass sich diese Wiederholung (praktisch notwendigerweise) auf alle direkt oder indirekt kausal abhängigen Ereignisse erstreckt. Wie oft, bzw. unter welchen Bedingungen wiederholt wird, kann als Fußnote (Notiz) an der betreffenden Botschaft notiert werden. Zuweilen sind direkt oder indirekt ausgelöste Ereignisse nicht strikt kausal verknüpft (treten nicht unter allen Umständen auf), oder stehen zu ihren Auslösern nicht in einem 1:1-Verhältnis. In solchen Fällen sollte man dies durch eine Notiz am betreffenden Ereignis dokumentieren (z. B. „Alle 5-10 eintreffende Datenpakete wird ein aufsummierter Report ans Backend übermittelt“). Lässt sich dagegen die Wiederholung nicht an einer einzigen Botschaft festmachen – etwa wenn zwei Botschaftenarten *A* und *B* aufgrund einer Protokolldefinition immer paarweise auftreten, aber dieses Paar wiederholt werden kann, dann wird man nicht umhin können, eine entsprechende Notation (Kasten, vertikale Linie), wie von Alhir (1998) illustriert, über beide Botschaften zu zeichnen und die Wiederholungsbedingungen an dieses graphische Element zu notieren.

Im Bezug auf die Guard-Condition ist unklar, ob man diese als Bedingung für die Anwendbarkeit des Diagramms oder als Bedingung für die Fortsetzung der Ereignisfolge zu interpretieren hat (wie Alhir (1998) vorschlägt). Ich tendiere dazu der ersteren Interpretation den Vorzug zu geben, da sie die Darstellung verschiedener Fälle (etwa den regulären Ablauf versus die Fehlerbehandlung) in getrennten Diagrammen stark vereinfacht. Man kann jedoch leicht jede Doppeldeutigkeit vermeiden, indem man alle Fälle an den jeweiligen Orten vollständig abdeckt: Den Abbruch der Ereigniskette kann man durch eine entsprechende Notiz fordern, und den Verweis auf ein getrenntes Diagramm, in dem Sonderfälle behandelt werden, kann man durch einen Pfeil ins Leere notieren zusammen mit einer UML-Notiz, die auf das entsprechende Diagramm verweist (Siehe UML 1.4, 2001, Seite 3-110).

1.4 Nachbedingungen und Zusicherungen

Beschreibt man eine Methode als Operation auf dem Objektmodell[9] mit Vor- und Nachbedingungen, so gibt dies zwar an, was die Bedingungen für den Aufruf einer Methode sind und wie der Zustand der Systemdaten nach der Rückkehr aus dem Methodenaufruf beschaffen ist, jedoch nicht, was während des Methodenaufrufs geschieht, insbesondere nicht, welche Botschaften während der Abarbeitung der Methode an andere Objekte geschickt werden. Diese Information liefern Ablaufdiagramme. In diesem Sinne stehen sie näher an der Implementation als die Constraints des statischen Modells dies vermögen.

Die Vorbedingungen sagen aus, wie ein Objekt (und möglicherweise sein Kontext) bei Aufruf einer Methode beschaffen sein kann. Daraus kann aber kein Aussage abgeleitet werden, welchen Zustand dieses Objekt hat, wenn es nun während der Abarbeitung der aufgerufenen Methode Botschaften an andere Objekte verschickt, da zwischen beiden Ereignissen durchaus Verarbeitungsprozesse stattgefunden haben können. Aus diesem Grund kann es

[9] In vielen Fällen wird ja nicht nur das Objekt, dem die Methode angehört, verändert, sondern auch andere Objekte, die mit ihm direkt oder indirekt assoziiert sind. In solchen Fällen ist die Methode eine Operation auf einem Ausschnitt des Objektmodells, nicht auf dem Objekt allein.

sich als nützlich oder notwendig erweisen, in Ablaufdiagrammen an den Botschaften, welche ein Objekt ins „Hinterland“ schickt, Zusicherungen zu notieren, und zwar einmal über den Zustand des sendenden Objektes selbst, möglicherweise das gesamte Objektmodell und zum anderen über die Beschaffenheit der ausgesandten Botschaft, die möglicherweise strikter ist, als die Vorbedingungen des empfangenden Objektes vorschreiben. Möchte man Vollständigkeit der abgedeckten Fälle und Konsistenz der beschriebenen Situation analysieren, sind solche Zusicherungen unumgänglich.

1.4.1 Grenzen: Multithreading, Activation-Frames

Streng genommen gehört zum Zustand eines Objektes auch der Activation-Frame gerade noch laufender Methodenaufrufe. Das kann dann eine Rolle spielen, wenn ein Objekt sich direkt oder indirekt wieder selbst aufruft (eine Botschaft schickt).

Zeitgenössische Notationen zur OO-Modellierung stellen für Aussagen über gerade laufende Methodenaufrufe kein Vokabular zur Verfügung. Zeitgenössische OO-Sprachen haben keine explizite Behandlung von Activation-Frames als Zustandsteile der Objekte. Beides ist meines Erachtens der Grund, warum Objektorientierung zur Zeit mit echtem feingranularen Multithreading nicht immer gut zusammenwirkt und indirekte rekursive Aktivierung der Objekte schlecht behandelt werden können. Um dies zu beheben, muss entweder der Activation-Frame laufender Methodenaufrufe formalisiert und als ein Teil des Objektzustandes begriffen werden, oder ein auf autonomen objektinternen Prozessen basierendes Objektmodell entworfen werden, in dem auch die Rückgabe von Ergebnissen durch das Senden von Botschaften (statt durch Abbau des Callstack) geschieht.

1.5 Eine abschließende Warnung vor „Spaghettidiagrammen“

Den Interaktionsdiagrammen der OO-Methoden (das sind Zustands-, Ablauf- und Kollaborationsdiagramme) haftet eine spezifische Gefahr an, nämlich dass, wenn der Entwerfer nicht acht gibt, sehr leicht „Spaghettidiagramme“, analog zum „Spaghetticode“ in der unstrukturierten Programmierung mit *goto*, entstehen. Der Grund dafür ist meines Erachtens der gleiche: Während die strukturierte Programmierung Notationselemente besitzt, die eine hierarchische Ordnung der Programmelemente erzwingt, hat die unstrukturierte Programmierung das nicht. Ganz analog fehlen in den Interaktionsdiagrammen hierarchisierende Notationselemente fast ganz. Verschärft wird diese Problem zudem durch die Tatsache, dass die objektorientierten Methoden eine nicht-hierarchische Gestaltung des Systems bevorzugen, man muss sogar fast sagen, dass dies geradezu die Philosophie der Objektorientierung ist (gleichberechtigte Teile, kein Hauptmodul).

Das soll nun nicht heißen, dass es unmöglich ist, mit der Notation der UML (oder verwandter Methoden) übersichtliche Entwürfe zu erstellen. Diese Übersicht zu bewahren, und immer wieder Diagramme und Entwurfsteile einzufügen, die einerseits Details wegabstrahieren, zum anderen die weiträumige Gestaltung des Systems zusammenfassen, ist aber die Aufgabe des Entwerfers: Dies wird nicht von der Notation erzwungen.

Insbesondere sollte man sich vor der Versuchung hüten, Zustandsdiagramme als Flußdiagramme im klassischen Sinn zu missbrauchen[10]. Beherzigt man dies Warnung und vermeidet diese Probleme, so sind diese Diagramme ein probates Mittel alle möglichen Abläufe –

[10] Diese Warnung kommt nicht von ungefähr: Über die Hälfte der bereits „besseren“ Literatur – d. h. Bücher der Art *Professionelle UML-Programmierung in 7 Tagen optimal lernen, Gold Edition* werden hier schon gar nicht mehr gezählt – macht meines Erachtens genau diesen Fehler. Nicht, dass an der Steinzeit der Programmierung viel falsch gewesen wäre: Aber diese Methoden, Flußdiagramme, kamen eigentlich deshalb außer Mode, weil sie für den Entwurf größerer System nicht mehr so richtig taugten.

vom Lebenslauf eine Objektes bis zur Bedienerführung – im angemessenen Detaillierungsgrad zu notieren, wie hoffentlich aus den folgenden Lösungsskizzen ersichtlich ist.

2 Aufgabe A: Fingerübungen

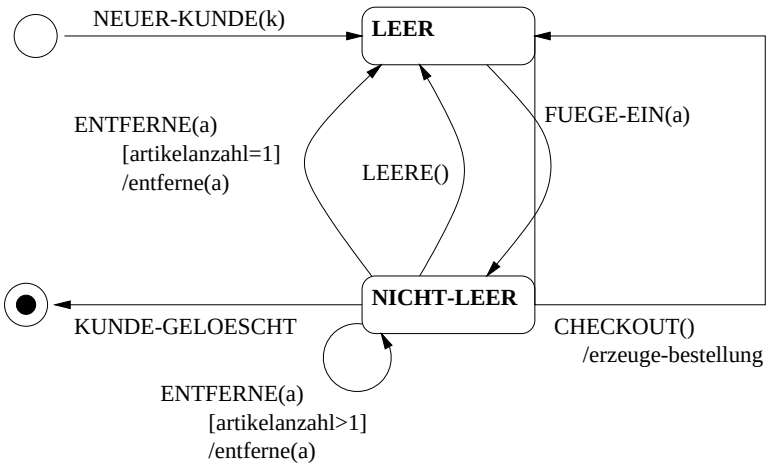
2.1 Einfaches Zustandsdiagramm (A.1)

Warenkorb
posten: Posten(0,*) artikelanzahl { 1 }
leere() entferne(p:Posten) fuege-ein(p:Posten) checkout()

1: Die Anzahl der im Warenkorb enthaltenen Posten, d. h. für alle w :*Warenkorb* gilt:

$$w.artikelanzahl = \text{card}\{p : Posten \mid p \text{ enthalten in } w\}$$

Das nun folgende Zustandsdiagramm gibt den *Lebenslauf* (*life cycle*) von Instanzen der obigen Klasse wieder. Diese Klasse ist unterschiedlich von der in Lösungsskizze 6 (Teach-SWT@Tü, 2002) gegebenen, aber beide sind vereinbar in dem Sinne, dass eine Klasse angegeben werden kann, welche die Charakteristika (Attribute) beider Klassen miteinander vereinbart, und von der dementsprechend die beiden unterschiedlichen, angebenen Klassen Ansichten oder Projektionen sind. Die Assoziation zu Instanzen der Klasse *Posten* ist als mehrwertiges Attribut geschrieben, wie dies von Graham u. a. (2001) propagiert wird.

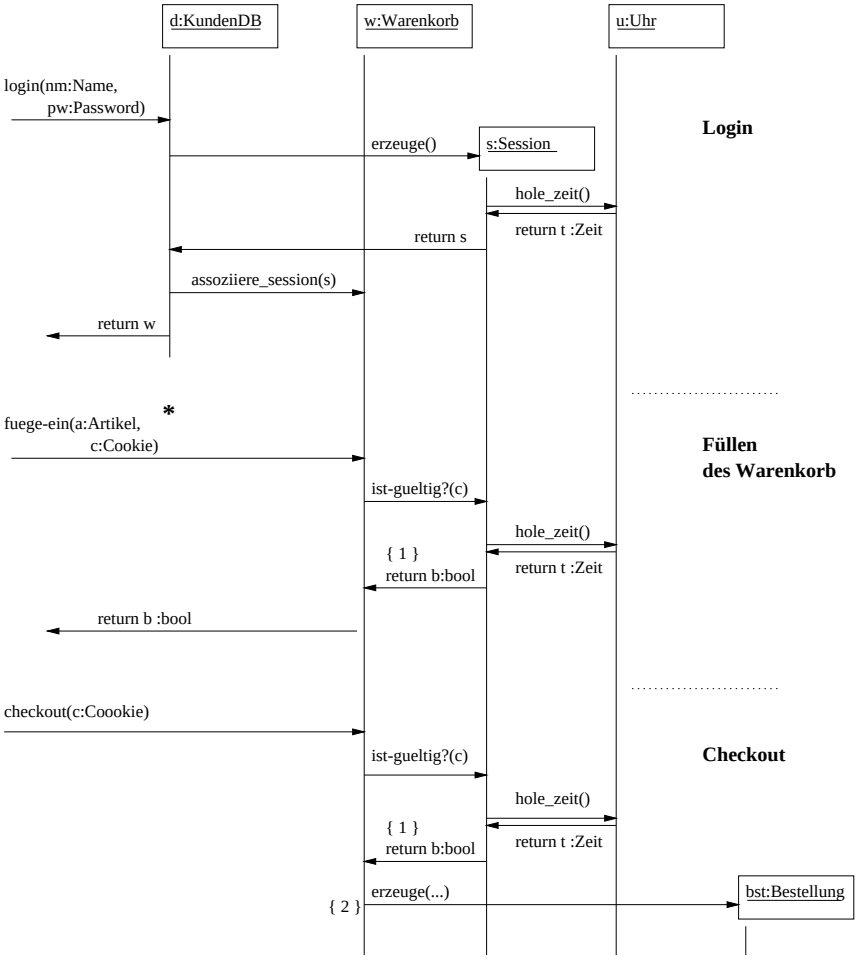


Zustände:

LEER: $artikelanzahl = 0$

NICHT-LEER: $artikelanzahl > 0$

2.2 Ablaufdiagramm (A.2)

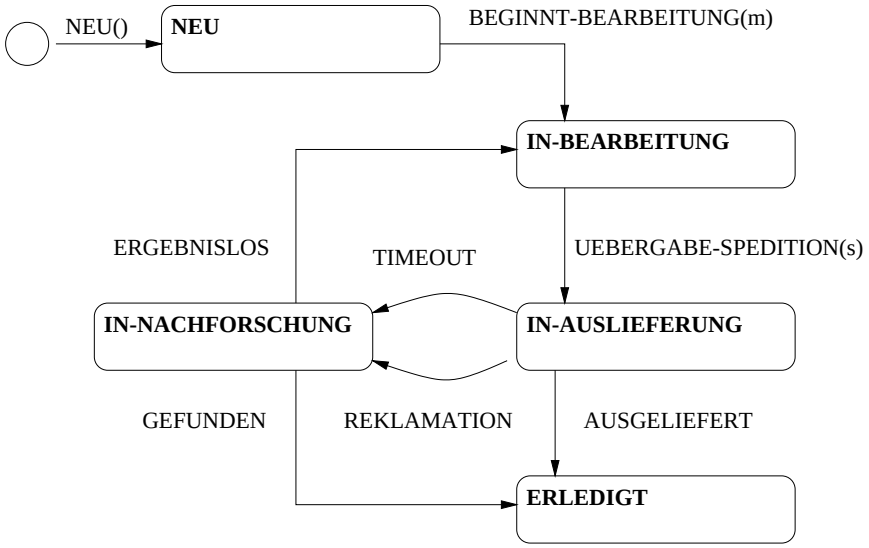


1: *b* ist **true** genau dann, wenn *t-s.letzter-Zugriff* $<$ *s.timelimit* **und** $c = s.cookie$.

2: Nur, wenn $b =$ **true**.

Dieses Diagramm kann auf verschiedene Arten variiert werden. Zum einen kann es an den rechts im Diagramm angedeuteten Schnittlinien in getrennte Diagramme für die Teilszenarien *Login*, *Füllen des Warenkorbes* und *Checkout* zerlegt werden. Zum anderen können Objekte, die weiter hinten in der Aufruffolge stehen, im Diagramm unterdrückt werden. Dies wäre hier beispielsweise für *u* :*Uhr* oder *s* :*Session* möglich. Selbstverständlich würden diese Objekte aber in den Bedingungen an den Aufruflinien (und im zugrundeliegenden Klassenmodell) wieder erwähnt werden müssen. Beide Modifikationen entsprechen der Bildung von engeren Teilansichten des Systems.

2.3 Komplexeres Zustandsdiagramm: Abarbeitung einer Bestellung (A.3)



Zustände:

NEU: $\#B.Bearbeiter = 0$.

IN-BEARBEITUNG: $\#B.Bearbeiter = 1$ **und** $\#B.Paket = 0$.

IN-AUSLIEFERUNG:
$\#B.Paket = 1$ **und nicht** *B.zugestellt* **und nicht** *B.reklamiert*.

ERLEDIGT: *B.zugestellt*.

IN-NACHFORSCHUNG: *B.reklamiert*.

Dabei steht „#“ hier für die Multiplizität (d. h. Anzahl) der in der betreffenden Rolle assoziierten Objekte. Wurde im Klassendiagramm kein Rollenname angegeben, so ist der Rollenname der Name der Klasse der assoziierten Objekte.

Dem in Lösungskizze 6 (TeachSWT@Tü, 2002) gezeigten Klassendiagramm müssen offensichtlich noch weitere Attribute der Klasse *Bestellung* hinzugefügt werden, damit es als Grundlage für das hier gezeigte Zustandsdiagramm dienen kann, namentlich die booleschen Attribute *B.zugestellt* und *B.reklamiert*.

Ereignisse:

BEGINNT-BEARBEITUNG(*m*): Der Mitarbeiter *m* übernimmt die Bearbeitung und trägt dies ins System ein.

UEBERGABE-SPEDITION(*s*): Das Paket wurde an die Spedition *s* übergeben. Das Ereignis ist genau genommen die Tatsache, dass diese Behauptung vom Mitarbeiter ins System eingetragen wird.

TIMEOUT: Der Sachbearbeiter *m* stellt fest, dass das Paket auch nach 2 Wochen noch nicht als ausgeliefert gemeldet ist.

REKLAMATION: Der Kunde reklamiert, dass das Paket nicht eingetroffen ist.

GEFUNDEN: Das Paket wurde doch noch gefunden und von der Spedition als an den Kunden ausgeliefert gemeldet.

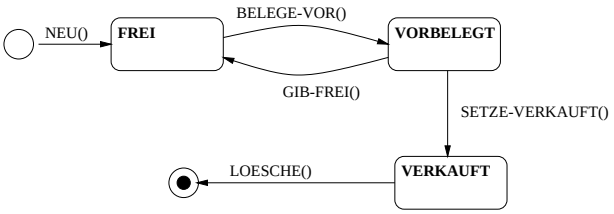
ERGEBNISLOS: Die Nachforschung der Spedition verläuft ergebnislos: Weder ist das Paket dem Kunden zugestellt worden, noch kann die Spedition das Paket finden und sieht sich demnach außerstande, es dem Kunden doch noch auszuliefern.

AUSGELIEFERT: Das Paket wird von der Spedition als ausgeliefert gemeldet.

Es gibt keine Zerstörung des Objektes, dessen Lebenskauf hier beschrieben ist. Zumindest was die hier behandelte Problemsicht betrifft, bleibt das Objekt auf ewig archiviert.

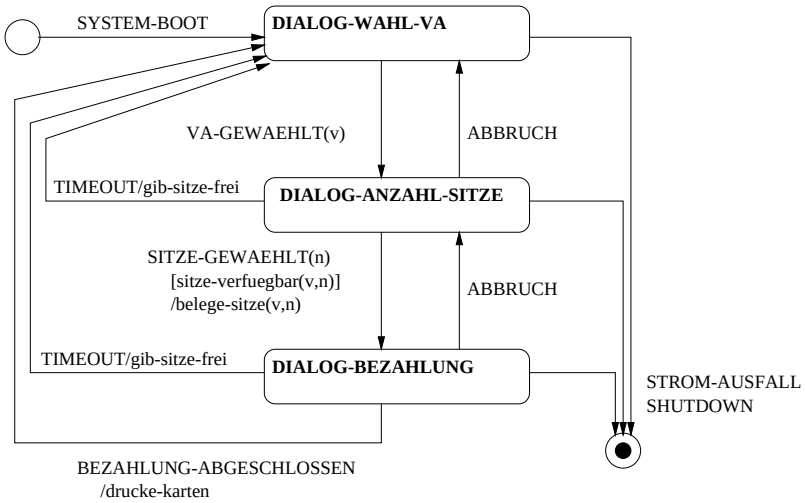
3 Ansichten eines Automaten zum Verkauf von Theaterkarten

3.1 Lebenslauf eines Platzes (B.1)



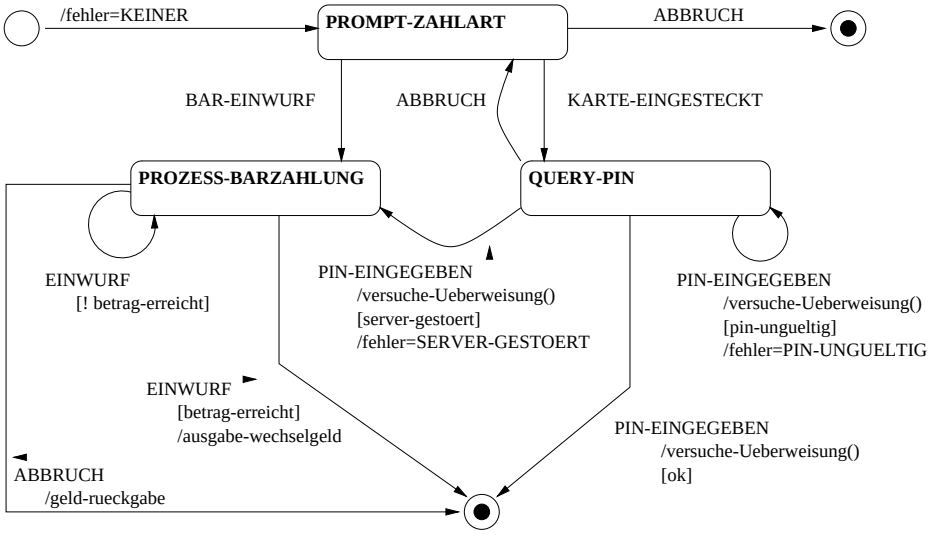
Zur Bedeutung von Operationen (die hier die Ereignisse stellen) und Zuständen siehe Aufgabenstellung.

3.2 Benutzerführung – Eine Anwendung für Ablaufdiagramme (B.2)



Der Diagrammausgang könnte hier auch weggelassen werden, seine Darstellung gibt wenig bis keine Information. Die Angabe *belege-sitze*(v,n) ist keine Operation auf der Aussen- oder Umwelt des System, sondern verrät uns vor allem, in welchen Unterzustand des Systems der Übergang erfolgt.

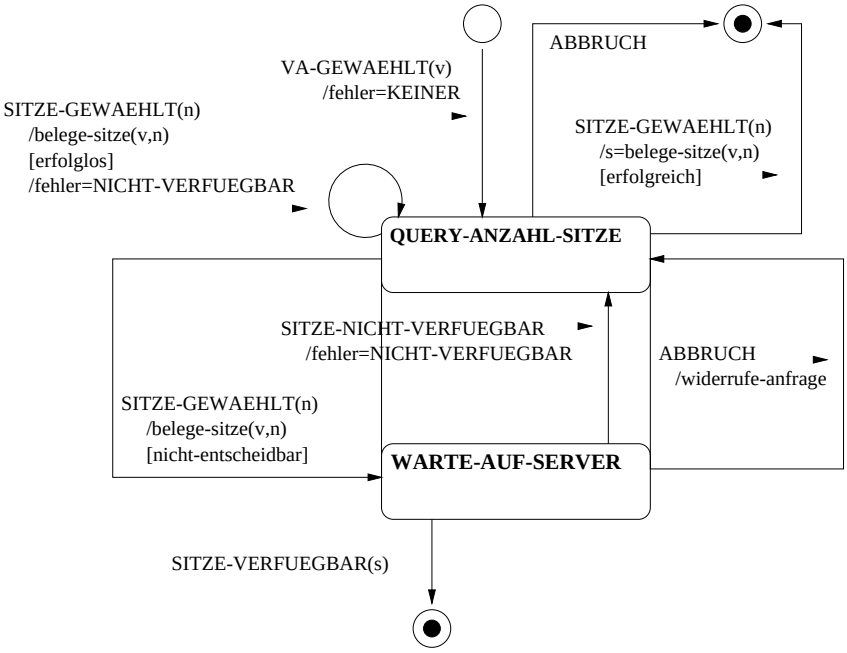
3.3 Bezahlvorgang (B.3)



Dieses Diagram beschreibt die Details des Prozesses DIALOG-BEZAHLUNG. Ein TIMEOUT ist ein internes Ereignis, das als eingetreten gilt, wenn der Benutzer für eine gewisse Zeit (dies muss noch genauer spezifiziert werden) keine Eingabe trifft. In jedem der Teilzustände dieses Prozesses führt ein TIMEOUT zum Abbruch des Prozesses (Verlassen des Diagramms), und in dem übergeordneten Diagrammen (B.2) gilt das TIMEOUT-Ereignis als ausgelöst. Das Ereignis TIMEOUT wurde aus Gründen der Übersichtlichkeit nicht in das Diagramm eingetragen.

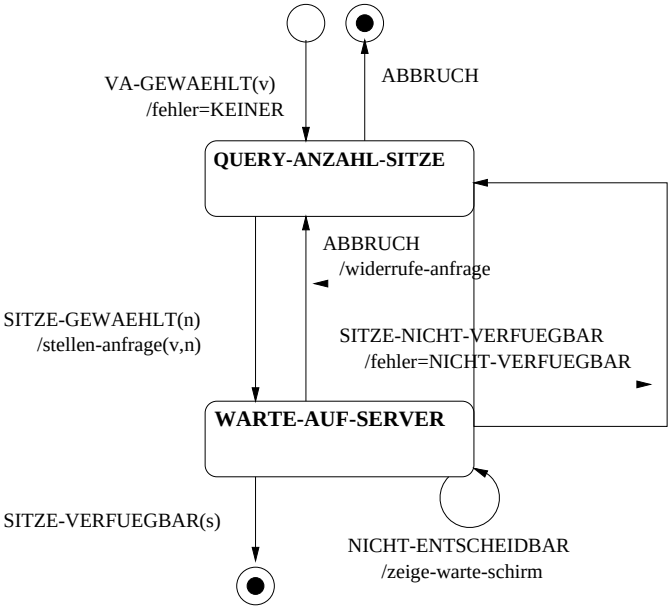
In den Dialogschirmen dieses Prozesses wird, wenn *fehler* gesetzt ist, ein entsprechender Meldetext eventuell mit Erläuterungen angezeigt. Der Text ist gegebenenfalls in einem späteren Entwurfsschritt auszuarbeiten.

3.4 Buchung von Sitzen (B.4)



Das vorangehende Diagramm gibt die der Aufgabestellung entsprechende Lösung wieder. Während der Erstellung dieser Lösungsskizze hat sich mir eine weitere Variante geradezu aufgedrängt, die allerdings eine leichte Änderung des Protokolls zwischen Client und Server suggeriert. Im obigen Diagramm wird angedeutet, dass der Server auf die Anfrage zuerst immer sofort eine Antwort liefert (synchron, dies könnte auch ein RPC-Aufruf sein), die in den Bedingungen *erfolgreich*, *erfolglos* oder *nicht-entscheidbar* resultiert. Im Fall der Bedingung *nicht-entscheidbar* muss in eine Wartezustand eingetreten werden, der entweder durch einem Abbruch oder das Eintreffen einer asynchronen Serverbotschaft SITZE-VERFUEGBAR(*s*) oder SITZE-NICHT-VERFUEGBAR beendet wird.

Dagegen setzt das alternative Diagramm vollkommen auf ein asynchrones Protokoll: Eine Anfrage zum Server wird abgesetzt, dann tritt das System in den Wartezustand, der durch Abbruch oder Botschaften der obigen Art beendet wird.



Beide Diagramme implementieren gleichwertige Lösungen. Da zudem (in der UML) nicht wirklich Genaues über die Bedeutung der an Übergängen und in Knoten ablaufenden Prozesse bekannt ist, ist der Unterschied zwischen synchroner und asynchroner Antwort im Wesentlichen in den Augen des Betrachters: Beide Diagramme könnten auch dasselbe System beschreiben.

Die Veranstaltung *v* ist in beiden Diagrammen eine Konstante.

Literatur

[Aho u. a. 1986] AHO, Alfred V. ; SETHI, Ravi ; ULLMAN, Jeffrey D.: *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986. – ISBN 0-201-10088-6

[Alhir 1998] ALHIR, Sinan S.: *UML in a Nutshell: A Desktop Quick Reference*. Erste Auflage. O'Reilly, 1998. – ISBN 1-56592-448-7

[Graham u. a. 2001] GRAHAM, Ian ; O'CALLAGHAN, Alan J. ; WILLS, Alan C.: *Object-oriented methods: principles & practice*. Erste Auflage. Addison-Wesley, 2001 (The Addison-Wesley object technology series). – ISBN 0-201-61913-X

[Page-Jones 2000] PAGE-JONES, Meilir: *Fundamentals of object-oriented design in*

UML. Addison-Wesley, 2000 (Addison-Wesley object technology series). – ISBN 0-201-69946-X

[TeachSWT@Tü 2002] LEYPOLD, M E.: Lösungsskizze 6: Statische OO-Modelle mit UML. Sand 13, D 72076 Tübingen, 2002 (Materialien zur Softwaretechnik). – `handouts/loesung-06.tex`

[UML 1.4 2001] Object Management Group, Inc. (Veranst.): *OMG Unified Modeling Language Specification: Version 1.4*. 2001. – URL `http://www.omg.org/cgi-bin/doc?formal/01-09-67`