Übungsblatt 7

Dynamische OO-Modelle

Ausgabe: 18.6.2002
Abgabe: 27.6.2002, 12:00

Inhaltsverzeichnis

1 Einleitung

1.1 Thema

Das heutige Übungsblatt ist der dynamischen Modellierung gewidmet – d. h. Methoden, die entweder die zeitliche Abfolge von Ereignissen in einem System beschreiben, oder die mögliche Änderung von Zuständen. Wir werden uns dabei auf Zustandsdiagramme (state charts) und Sequenzdiagramme (sequence charts) beschränken. Kollaborationsdiagramme (collaboration charts) sind logisch zu den Sequenzdiagrammen äquivalent, jedoch für bestimmte Zwecke übersichtlicher.

Zum Übungsblatt teilen wir Kopien aus den folgenden Büchern aus:

- Bernd Östereich *Die UML-Kurzreferenz für die Praxis*.
- Ian Graham *Object-Oriented Methods – Principles & Practice* Graham u. a. (2001).

1.2 Bemerkung zu Sequenzdiagrammen

Sequenzdiagramme stellen (bedingt durch die eingeschränkte Syntax) nur einen Ausschnitt der möglichen Vorgänge dar. Typischerweise ist dies ein bestimmtes Szenario (wie: Kauf eine Fahrkarte), verwandte Szenarien müssen – vor allem, wenn sie komplexer sind – in separaten Sequenzdiagrammen dargestellt werden (wie z. B. komplexere Fehlerbehandlungen).

Bemerkung zu Zustandsdiagrammen Die Tatsache, dass Aktivitäten und Prozesse in die Knoten eines Zustandsdiagramms geschrieben werden können, ist verwirrend, da es der Idee eines Zustandes prinzipiell widerspricht. Wenn auch außer Zweifel steht, dass sich dafür mit etwas Sorgfalt sinnvolle Anwendungsfälle finden lassen, möchte ich dennoch empfehlen, dieser Konfusion zu Anfang auszuweichen und sich soweit als möglich auf Diagramme zu beschränken, in denen Aktionen – wie bei Ian Graham – zu den Übergängen notiert werden (statt in den Zuständen), und Zustände als ein "Warten" des Systems auf eine äußeren Eingabe oder ein inneres Ereignis (Ablauf eines Timers oder Terminieren eines Verarbeitungsprozesses) interpretiert werden können.

Zustände in Zustandsdiagrammen beschreiben eigentlich Äquivalenzklassen von Zuständen des Systems oder von Systemteilen.

1.3 Bemerkung zum Stand dieser Anleitung

Hier (oder in einem separaten Handout) sollte eigentlich ein längeres Tutorial stehen, wie Sequenz- und Zustandsdiagramme zu interpretieren sind, und wie die „gute Praxis" aussieht. Beklagenswerterweise macht sich die realexistierende Literatur in dieser Hinsicht sehr rar (anything goes), da, wie es aussieht, jeder davon ausgeht, dass man beim Erstellen solcher Diagramme wenig falsch machen könne. Die in der Literatur abgedruckten Beispiele, die häufig wie Rückfälle in die finstersten Zeiten der Spaghetti-Programmierung mit Goto und Flussdiagrammen anmuten, widerlegen dies eindrucksvoll. Es gibt gute und es gibt schlechte Diagramme der genannten Arten. Die guten Diagramme unterscheiden sich von den schlechten vor allem dadurch, dass sie

- *das beschriebene Systemverhalten sinnvoll und hierarchisch zerlegen,*

- *einen exakten und (innerhalb des Diagramms) einheitlichen Zustandsbegriff zugrundelegen, und*
- *dies mit der Tatsache, dass viele Systeme auch eine innere Dynamik besitzen, vereinbaren können.*

Es rentiert meines Erachtens, zu diesen Punkten klare Regeln zu erarbeiten, da Sequenz- und Zustandsdiagramme für den Ungeübten viele Fallstricke bereithalten, andererseits diese Diagrammarten wirklich ein wertvolles Mittel sind, um Anforderungen von der Kundenseite zu notieren und aufzunehmen oder in einfachen Fällen Nebenläufigkeiten und Echtzeitanforderungen zu analysieren.

2 Aufgabe A: Fingerübungen

Zum Warmwerden sollt Ihr einige Abläufe aus dem Web-Shop, der im letzten Übungsblatt modelliert wurde, in Sequenz- bzw. Zustandsdiagrammen darstellen.

1. Modelliere als Zustandsdiagramm das Leben eines Warenkorbs: Wie bereits im letzten Übungsblatt erläutert, soll der Warenkorb im Web-Shop von *United Clothes* seinen Inhalt auch zwischen verschiedeneren Logins bewahren: Der Warenkorb ist ein persistentes Objekt. Jeder Kunde besitzt genau einen Warenkorb, der eingerichtet wird, wenn der Kunde für den Web-Shop registriert wird, und erst mit der endgültigen Abmeldung des Kunden obsolet wird. Der Warenkorb agiert als Container für die verschiedenen Posten einer Bestellung, die ein Kunde auf seinem Weg durch den Web-Shop einsammelt. Der Kunde kann alle oder einzelne Posten löschen (verwerfen), oder (während des Checkout) eine Bestellung erzeugen, wobei der Warenkorb natürlich auch geleert werden muss. Zeichne ein Klassendiagramm des Warenkorbs mit den nötigen Operationen und ein Zustandsdiagramm.
2. Zeichne nun ein Sequenzdiagramm, das den den Warenkorb betreffenden Ablauf des Besuchs eines Kunden im Webshop mit anschließendem Checkout beschreibt.
3. Die beim Checkout erzeugte Bestellung durchläuft danach den Lieferprozess: Wir wollen nun (im Gegensatz zur Aufgabenstellung des letzten Blattes) auch betrachten, wie der Fall behandelt wird, dass ein Paket beim Kunden nicht angekommen ist. Eine neu erzeugte Bestellung wird früher oder später von einem Mitarbeiter begonnen zu bearbeiten – der Mitarbeiter trägt sich am Computer als für die Bearbeitung Zuständiger ein. Auch die Übergabe der Lieferung an die Spedition wird im Programm zur Verfolgung des Lieferstatus festgehalten. Meldet die Spedition die Auslieferung, kann die Bestellung archiviert werden. Reklamiert der Kunde, daß die Lieferung nicht angekommen ist oder meldet die Spedition keine Zustellung innerhalb einer gewissen Zeit, muss der Lieferung nachgeforscht werden. Verläuft die Nachforschung ergebnislos, muss eine neue Lieferung zusammengestellt werden. Erstelle nun ein Zustandsdiagramm, dass die Abarbeitung einer Bestellung beschreibt.

Abzugeben sind: Die Diagramme zu den obigen Aufgaben (1 Klassendiagramm, 2 Zustandsdiagramme, 1 Sequenzdiagramm).

3 Aufgabe B: Schon wieder Theaterkarten

3.1 Problembeschreibung

Eines der vergangenen Übungsblätter hatte einen Automaten zum Verkauf von Theaterkarten zum Gegenstand. Zur Modellierung solcher Systeme sind Zustandsdiagramme und Sequenzdiagramme ganz besonders geeignet, da in der Regel das Verhalten dieser interaktiven Systeme als Ihre primäre Leistung wahrgenommen wird (im Gegensatz zu ihrer datenverarbeitenden, die häufig in den Hintergrund tritt). Besonders schön lässt sich die Benutzerführung (d. h. das Fortschreiten durch eine Transaktion) durch Zustandsdiagramme darstellen und schrittweise verfeinern.

Ihr sollt nun im Folgenden Zustandsdiagramme für die Interaktionen des Terminals mit dem Benutzer bzw. dem Datenbankserver erstellen. Als Hintergrundinformation möchte ich zuerst einen kurzen Überblick (im Wesentlichen in Stichworten) über die in Aussicht genommene Architektur geben. Nicht alle Informationen aus diesem Überblick werden zur Durchführung der anschließenden Aufgaben wirklich benötigt.

Eine Besonderheit des Theaterkartenterminals ist es, dass es sich dabei um eine verteilte Anwendung handelt: Die auf den Terminals ausgeführten Instanzen der Verkaufssoftware müssen sich einen gemeinsamen Zustand teilen: Die Datenbank, in der die Veranstaltungen und vor allem deren Belegungszustand gespeichert ist. Verteilte Anwendungen lassen sich objektorientiert zwanglos realisieren, insbesondere wenn man über einen Mechanismus zur *Remote Method Invocation* verfügt, d. h. über einen Mechanismus, bei dem die Methoden von Objekten, die auf einem anderen Rechner leben, ebenso aufgerufen werden können, als wären diese Objekte auf dem lokalen Rechner. Da die Kommunikation sowieso nur über Nachrichten erfolgt, also eben nicht über direkten Zugriff auf Attribute, können lokale und entfernte Objekte nicht wirklich voneinander unterschieden werden. Das Objektmodell wird dementsprechend Server und Client anfangs nicht unterschieden, stattdessen ist das Modell an der Oberfläche nach wie vor nur eine Kollektion von Objekten, welche miteinander über Nachrichten kommunizieren. Bei genauerem Hinsehen (bzw. in einer späteren Entwurfsphase) stellt sich heraus, das ein Teil der Objekte auf dem Server lebt – und damit den “shared state” repräsentiert – und ein anderer Teil der Objekte auf den Terminals lebt (und dementsprechend auf jeden Fall mehrfach instantiiert wird).

Ich werden nun kurz einige Klassen einer möglichen Architektur des Theaterkartenverkaufssystems und ihre Aufgaben aufzählen:

Platz: Ein Platz ist, was letzten Endes verkauft wird. Er kann unbelegt oder bereits verkauft sein, besitzt eine Platznummer und ist immer einer Veranstaltung zugeordnet.

Veranstaltung: Objekte dieser Klasse beschreiben einzelne Veranstaltungen.

Datenbank: Die Datenbank dient als Container für Objekte. Verschiedene Abfragen ermöglichen es, Listen von Veranstaltungen aus der Datenbank herauszuziehen.

Session: Dieses Objekt enthält den jeweiligen den Zustand eines Verkaufsvorgangs. Alle Operationen auf diesen Zustand werden über das Session-Objekt (Belegung, Berechnung des Preises) ausgeführt. Das Session-Objekt repräsentiert das Datenmodell eines Verkaufsvorgangs.

In Diskussionen mit Teilnehmern und studentischen Mitarbeitern hat sich ein alternatives Modell herausgeschält. In diesem werden die Aktionen des Kunden durch das Webfrontend (welches nicht Bestandteil des Modells ist!), in Methodenaufrufe auf das Warenkorbobjekt übersetzt und tragen ein Cookie mit sich. Das Warenkorb-Objekt prüft gegen das alternative Session-Objekt, ob die Operation erlaubt ist.

Der Warenkorb agiert hier also als Wächter über gewisse Operationen auf dem Objektmodell, wobei der Authentisierungsdienst ins Sessionobjekt ausgelagert ist. Konsequente Fortschreibung dieses Entwurfsprinzips führt dazu, dass alle Objekte, die Kontakte des zentralen Objektsystems zur Außenwelt herstellen, solche Kontrollfunktionen übernehmen müssen. Dies ist jedoch mehr im Rahmen der üblichen OO-Entwurfsmuster (Dezentralität ist hier ein Stichwort), als die Kanalisierung aller Operationen durch **ein** *authentisierendes Frontend-Objekt, das hier im ursprünglichen Ansatz „Session" genannt wurde.*

Dialog: Das Dialogobjekt steuert den Ablauf der Transaktion mit dem Benutzer, entscheidet also, welche Meldungen als nächste angezeigt werden, und welche Operationen auf dem Session-Objekt ausgeführt werden.

Screen: Definiert und kapselt das Aussehen einer Schirmanzeige, und die Eingaben (Eingabeereignisse), die während der Anzeige dieses Schirms bearbeitet werden.

InputEvent: Ein Eingabeereignis von Tastatur, Kartenleser, Münzeinheit oder eine asynchrone Nachricht vom Server.

InputQueue: Eine Queue, in der *InputEvents* gespeichert werden. Alle Hardwaretreiber stellen Eingabeereignisse direkt in die *Inputqueue*.

Eine weitere späte Erkenntnis: Die objektorientierte Philosophie impliziert unter anderem das Bestreben, zentrale Dispatch-Loops für „Events" zu vermeiden. RTTI (Run Time Type Information) steht sowieso in einigen objektorientierten Sprachen (beispielsweise OCaml) nicht zur Verfügung. Ohne RTTI und ohne Union-Types ist es auch nicht möglich, zentralen Dispatchcode für InputEvents zu schreiben.

Aus diesem Grund sollte man die Klasse InputQueue wohl besser streichen und stattdessen einen Observerpattern auf den originalen Ereignisquellen verwenden. Es würde dann dementsprechend auch keine gemeinsame Basisklasse „InputEvent" geben.

Es lohnt meines Erachtens, die eben gemachten Behauptungen genau zu durchdenken: Ich halte außer kontraproduktiven Vererbungshierarchieen die unangemessene Zentralisierung und Hierarchisierung von kontrollflussbestimmenden Elementen und der damit automatisch einhergehende Missbrauch von RTTI für den häufigste Fehler in objektorientierten Entwürfen und Implementationen.

Ein prinzipielles Problem entsteht daraus, dass mehrere Terminals (d. h. die Session-Objekte dieser Terminals) parallel auf die Datenbank (bzw. den Server) zugreifen, auf dem die Veranstaltungs-Objekte ausgeführt werden, und der Anforderung, dem Kunden nötigenfalls mehrere zusammenhängende Sitze zu verkaufen. Es genügt nicht, beim Server die Verfügbarkeit der Plätze zu prüfen, dann dem Kunden die Bezahlung abzufordern und schließlich die Plätze zu belegen. Geht man so vor, kann es geschehen, dass in der Zwischenzeit ein anderes Terminal eben diese Plätze belegt und man sich in der peinlichen Situation wiederfindet, dem Kunden zwar bereits Geld abgenommen zu haben, aber eigentlich keine Tickets mehr ausdrucken zu dürfen. Hier liegt eine Race-Condition vor, welche sich aber mit relativ einfachen Mitteln umschiffen lässt: Anstatt zu prüfen, ob Plätze in der gewünschten Zahl vorhanden sind, versucht man diese diese Plätze sofort *vorzubelegen*, was fehlschlägt, wenn keine Plätze mehr vorhanden sind, oder gelingt, wenn Plätze vorhanden sind. Da vorbelegte Plätze durch andere Terminals nicht mehr belegt werden können, kann man nun in aller Ruhe den Bezahlvorgang abwickeln, und dann die Plätze endgültig belegen. Wird der Vorgang abgebrochen, müssen die vorbelegten Plätze natürlich wieder freigegeben werden.

Leider öffnet diese Lösung einem weiteren Problem die Tür: Nehmen wir an, ein Benutzer versucht (eventuell mit der Absicht der Sabotage) eine ganze Veranstaltung leerzukaufen,

verlässt aber das Terminal, sobald er zur Bezahlung aufgefordert wird: Da die Plätze dann schon vorbelegt sind, kann aus keinem Automaten der Stadt mehr ein Ticket zu dieser Veranstaltung verkauft werden, bis der nächste Kunde den Vorgang an eben diesem Automaten abbricht. Zwei Dinge werden daraus klar:

- Nach einer gewissen Zeit sollte der Vorgang automatisch abgebrochen, wenn keine Eingabe mehr erfolgt.
- Es kann sein, dass obwohl keine Plätze mehr verfügbar sind (unbelegt), als auch einige Plätze doch noch verfügbar werden, weil sie nur vorbelegt waren, und der dazugehörige Vorgang abgebrochen wird.

Die Strategie der Vorbelegung von Plätzen kann dann verbessert werden: Anstatt nur zwischen den beiden Fällen zu unterscheiden, dass die gewünschten Plätze verfügbar sind und dem, dass diese nicht mehr verfügbar sind, führt man auch noch den Fall ein, in dem möglicherweise doch noch Plätze verfügbar werden: In diesem Fall gibt das Veranstaltungsobjekt (auf dem Server) an das Terminal statt der gewünschten Menge an Sitzen eine spezielle Nachricht zurück, welche besagt, daß diese Anfrage noch nicht entschieden werden kann. Das Terminal wartet dann entweder auf den Abbruch des Vorgangs durch den Benutzer oder auf eine Benachrichtigung, ob die Sitze vorbelegt werden konnten. Auch die Anfrage muß natürlich zurückgezogen werden, wenn der Vorgang abgebrochen wird.

Eine Frage, die sich die eine oder der andere vielleicht noch stellt: Warum verzögert man die Rückkehr des Methodenaufrufs, welcher die Plätze vorbelegt, nicht einfach solange, bis ein endgültiges Ergebnis geliefert werden kann? Der Grund dafür ist die Behandlung der Abbruch-Taste: Diese muss ja vom Terminal auch während des Wartens auf die Serverantwort bearbeitet werden, zudem wäre es auch wünschenswert, wenn dem Benutzer eine Nachricht angezeigt wird, welche ihn darüber informiert, daß er (u. U.) einige Minuten warten muss, aber durchaus noch eine gewisse Chance besteht, Karten zu erhalten. All dies kann auch mit einem entsprechenden Einsatz von Threads erreicht werden, was aber auch nichts weiter tut, als die “asynchrone” Rückmeldung des Ergebnisses, die wir hier ins Protokoll eingebaut haben, mit anderen Mitteln zu emulieren[1].

3.2 Teilaufgabe B.1: Lebenslauf eines Platzes

Ein *Platz* ist (in meiner Terminologie) immer genau einer Veranstaltung zugeordnet. Was im Theatersaal steht (über mehrere Veranstaltungen hinweg), ist ein *Stuhl*. Die oben erwähnten Belegungszustände sind Attribute eines Platzes.

Beschreibe den Lebenslauf eines Platzes in einem Zustandsdiagramm.

[1] Überhaupt ist das Zusammenspiel von OO mit nebenläufiger Programmierung nicht so einfach. Die Metapher, dass Nachrichten von Objekt zu Objekt verschickt werden, legt ja zuerst einmal nahe, daß alle Objekte nebenläufig ausgeführt werden. Nimmt man den Gedanken jedoch ernst, und versucht auch den Rückgabewert als eine Nachricht zwischen zwei Objekten zu interpretieren (und ebenso zu behandeln), muss man früher oder später feststellen, dass man sich damit Race-Conditions in großer Zahl einhandelt: Was passiert wenn zwischen dem Absenden einer Nachricht und dem Empfang der Antwort eine weitere Nachricht bei einem Objekt eintrifft? In welcher Reihenfolge werden solche Nachrichten zugestellt? Wird sie fallen gelassen? Was, wenn der innere Objektzustand gerade in einem “ungültigen” Zwischenzustand ist (also den zugesicherten Invarianten nicht genügt)? Dass es in den meisten OO-Sprachen als Regel erst mal nur einen Thread gibt, der sich durch die Methodenaufrufe windet und bei der Rückkehr die Antworten mit sich trägt, garantiert, dass man sich über so etwas (das Eintreffen von Nachrichten zur Unzeit) keine Gedanken machen muss. Man kann mit Fug und Recht das Zusammenspiel von OO und Nebenläufigkeit als ungeklärt bezeichnen. Das ist – unter anderem der Grund – warum ich hier die notwendige Nebenläufigkeit von Terminal und Server im Protokoll explizit machen möchte.

3.3 Teilaufgabe B.2: Ablauf eines Kartenverkaufs

In der Übersicht spielt sich der Verkauf einer Karte folgendermaßen ab:

1. Der Benutzer wählt eine Veranstaltung.
2. Der Benutzer wählt die Anzahl der Sitze. Dabei werden die Sitze auf dem Server vorläufig belegt (siehe oben).
3. Der Benutzer bezahlt die Karte.
4. Die Karten werden gedruckt und auf dem Server endgültig belegt.

Es soll dabei möglich sein, jederzeit, d. h. in jedem Schritt des Vorgangs, diesen mit der Abbruchtaste zu beendigen.

Beschreibe den Ablauf eines Verkaufs als Zustandsdiagramm. Berücksichtige dabei, dass nicht jeder der oben genannten Punkte einem Zustand entsprechen muss.

3.4 Teilaufgabe B.3: Verfeinerung: Bezahlung

Die Auswahl der Veranstaltung, dass Auswählen der Anzahl der Sitze und der Teilvorgang Bezahlung sind jeweils aus weiteren Einzelschritten zusammengesetzt und lassen sich wiederum durch eigene Zustandsdiagramme beschreiben. Die Auswahl der Veranstaltung ist dabei vom akademischen Standpunkt aus langweilig, weswegen wir uns nun auf den Ablauf der Bezahlung und die Vorbelegung der Sitze konzentrieren wollen.

Nachdem die Anzahl der Sitze ausgewählt ist (und der Gesamtpreis feststeht), sieht der Benutzer eine Meldung, die ihm eben diesen Gesamtpreis zeigt und ihn zum Bezahlen mit Scheckkarte oder Bargeld auffordert. Je nachdem, ob er nun Bargeld einwirft oder die Scheckkarte einschiebt, fährt das Terminal entsprechend fort: Bei Barzahlung wird mit jeder Münze der noch fehlende Betrag angezeigt, bis der Gesamtpreis entrichtet ist, bei Kartenzahlung wird die PIN erfragt und dann versucht über das Bankingmodul eine Überweisung vorzunehmen. Schlägt die Überweisung fehlt, kann das zwei Gründe haben: Die Verbindung zum Bankserver ist gestört, oder die PIN ist ungültig. Auch diese Fälle müssen sinnvoll behandelt werden.

Sinnvollerweise sollte das Drücken der Abbruchtaste in einigen Teilen dieses Vorgangs zuerst zum Schirm mit der Zahlungsaufforderung zurückführen, statt sofort zum Anfang des Verkaufsvorganges. Beim endgültigen Abbruch des Bezahlvorgangs ist es nötig, die vorbelegten Plätze wieder freizugeben.

Beschreibe den Bezahlvorgang durch ein Zustandsdiagramm.

3.5 Teilaufgabe B.4: Verfeinerung: Vorbelegung von Sitzen

Der Käufer kann mehrere zusammenhängende Sitze buchen. Nachdem der Käufer die gewünschte Anzahl von zusammenhängenden Plätzen eingegeben hat, wird versucht, diese Plätze auf dem Server vorzubelegen. Wie oben erläutert, sind darauf drei Antworten des Servers möglich: Dass keine Plätze mehr verfügbar sind, eine Menge mit den vorbelegten Sitzen oder dass es notwendig ist, noch ein Weilchen zu warten, ehe entschieden werden kann, ob noch Plätze verfügbar sind. Im letzteren Fall ist garantiert, daß der Server eine weitere Nachricht sendet, welche entweder mitteilt, dass (doch) keine Plätze mehr verfügbar sind, oder wieder eine Menge der vorbelegten Plätze enthält.

Während der Wartephase muss es möglich sein, mittels der Abbruchtaste den Vorgang zu beendigen. Dabei muss dann auch die Anfrage an den Server zurückgenommen werden.

Beschreibe die Auswahl von Sitzen mit einem Zustandsdiagramm..

Abzugeben sind: Zustandsdiagramme: Lebenslauf eines Platzes, Übersicht über einen Verkaufsvorgang, Details des Bezahlvorgangs, Details der Auswahl und der Vorbelegung von Plätzen.

Viel Spaß!

Literatur

[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