KAPITEL 3

Eine erste Fallstudie

In diesem Kapitel stellen wir UML vor, die Unified Modeling Language. Wir wollen dabei hier nur so viel von UML und seinen Möglichkeiten, Systeme zu entwerfen und zu spezifizieren, vorführen, um den Kontext für die detaillierteren Kapitel im Anschluss zu setzen. Im nächsten Kapitel werden wir uns mit den Ursprüngen von UML und ihrer Rolle im Entwicklungsprozess beschäftigen. In Teil II werden wir tiefer auf UML eingehen, dabei aber häufig die einfache Fallstudie hier heranziehen.

3.1 Das Problem

Der schwierigste Teil in der Entwurfsphase eines jeden Projekts ist es, die zu lösende Aufgabe überhaupt zu verstehen. Sehen wir uns die folgende Situation an:

> Sie haben den Auftrag erhalten, ein Computersystem für eine Universitätsbibliothek zu entwickeln. Derzeit verwendet die Bibliothek ein Programm aus den Sechzigern, geschrieben in einer überholten Sprache. Damit erledigt sie eine Reihe von einfachen Verwaltungsaufgaben und führt einen Schlagwortkatalog auf Kartenbasis. Von Ihnen wird ein interaktives System erwartet, mit dem beide Tätigkeiten online erledigt werden können.

3.1.1 Anforderungen klären

Die obige Problembeschreibung ist ziemlich vage, aber typisch für erste Anfragen im Frühstadium der Projektdefinition. Bevor man auch nur zusagen kann, ein Design zu entwerfen, ist eine viel genauere Untersuchung der eigentlichen Anforderungen der Benutzer erforderlich. Diese Aufgabe des *Requirement Engineering* ist aus einer Vielzahl von Gründen nicht ganz einfach:

- Unterschiedliche Anwender haben oft unterschiedliche, manchmal sogar widersprüchliche Prioritäten.
- Es ist sehr unwahrscheinlich, dass Anwender klare und einfach auszudrückende Vorstellungen dessen haben, was sie wollen. Es ist beispielsweise für sie sehr schwierig, zu unterscheiden, was ein existierendes System macht und was ein geeignetes System machen muss.
- Man kann sich nur schwer vorstellen, wie man mit einem System arbeitet, von dem man nur die Beschreibung kennt. Ein Anwender wird also denken, die Beschreibung eines Systems wäre schon in Ordnung, obwohl ihr in Wirklichkeit wesentliche Teile fehlen.

- Die Manager, die hauptsächlich mit den Entwicklern reden, haben vielleicht gar nicht so viel direkte Erfahrung mit der Erledigung der Aufgabe wie die Anwender des Systems.

Diskussionsfrage 15

Wie können Sie als Entwickler dazu beitragen, diese Probleme zu überwinden?

Die umfassende Behandlung des Requirement Engineering kann von diesem Buch nicht mit abgedeckt werden; gut dargestellt ist es in [44].

Nach sorgfältigen Untersuchungen kristallisieren sich die folgenden Anforderungen heraus, wie sie durch ein ideales System abgedeckt wären:

- **Bücher und Zeitschriften** In der Bibliothek gibt es Bücher und Zeitschriften. Von Büchern kann es auch mehrere Exemplare geben. Einige Bücher sind nur für Kurzzeitentleihe. Alle anderen Bücher können von jedem Bibliotheksmitglied für drei Wochen ausgeliehen werden. Zeitschriften dürfen nur von Mitarbeitern ausgeliehen werden. Mitglieder der Bibliothek können normalerweise bis zu sechs Einheiten entleihen, Mitarbeiter der Bibliothek dürfen hingegen gleichzeitig bis zu zwölf Einheiten entleihen. Regelmäßig laufen neue Bücher und Zeitschriften ein und alte werden gelegentlich weggeworfen. Am Ende jeden Jahres werden die Zeitschriften des aktuellen Jahrgangs zum Binden weggeschickt.
- **Entleihe** Wichtig ist, dass das System verfolgen kann, wann Bücher und Zeitschriften ausgeliehen und zurückgegeben werden, weil dies auch das aktuelle System schon kann. Das neue System sollte darüber hinaus Mahnungen auswerfen, wenn ein Entleihzeitraum überzogen ist. Eine zukünftige Anforderung kann sein, dass Benutzer die Entleihdauer eines Buches verlängern können, wenn es nicht anderweitig vorbestellt ist.
- **Suche** Vom System sollte ermöglicht werden, dass Anwender nach einem Buch zu einem bestimmten Thema oder von einem bestimmten Autor etc. suchen und prüfen können, ob das Buch zur Entleihe verfügbar ist und, falls nicht, es auch gleich reservieren können. Die Suche ist für jedermann zugänglich. So ist es jetzt schon besser, aber es ist immer noch nicht klar, welche unterschiedlichen Aufgaben es gibt und wer was benötigt.

Diskussionsfrage 16

Welche weiteren Fragen müssten Sie noch stellen?

Diskussionsfrage 17

Welche Nachteile hat es, ein System auf der Basis der oben geschilderten Anforderungen zu entwerfen, ohne weitere Analysen anzustellen?

3.1.2 Das anwendungsfallorientierte Modell

Ein System, das qualitativ hochwertig sein soll, muss den Bedürfnissen der Anwender entgegenkommen.[1] Für die Systemanalyse nehmen wir also einen *anwenderorientierten* Standpunkt ein. Wir identifizieren dabei die Anwender des Systems und die Aufgaben, die sie mit Hilfe des Systems lösen müssen. Wir brauchen aber auch Informationen darüber, welche Aufgaben am wichtigsten sind, damit wir die Entwicklung entsprechend planen können.

Was ist mit „Anwender“ (user) und „Aufgabe“ (task) gemeint? UML verwendet hier die Fachausdrücke *Akteur* (*actor*) und *Anwendungsfall* (*use case*). Ein Akteur ist ein Anwender des Systems *in einer bestimmten Rolle*. (Genau genommen kann ein Akteur auch ein externes

[1] Einige Leute würden hier lieber von „Kunden“ sprechen statt von „Anwendern“ – frei nach dem Motto: „Wer zahlt, schafft an!“ Der Kunde wird aber sicher von den Anwendern informiert, und es muss schon ein sehr seltsamer Kunde sein, der der Idee nachhängt, ein System wäre gut, wenn die Anwender es verfluchen!

System sein, das vom aktuellen System aus betrachtet wie ein Anwender erscheint. Worauf es hier ankommt ist, dass es etwas oder jemand ist, das bzw. der extern zum aktuell betrachteten System ist, mit diesem aber in Interaktion tritt und Anforderungen daran stellt.) So wird unser System beispielsweise einen Akteur `BuchEntleiher` haben, der für eine Person steht, die mit dem System interagiert und ein Buch entleiht. Dabei ist es für uns nicht erkenntlich, ob es ein Bibliotheksmitglied ist oder ein Bibliothekar im Auftrag eines Mitglieds, für unsere aktuellen Zwecke müssen wir das aber auch gar nicht wissen. Ein *Anwendungsfall* ist eine Aufgabe, die ein Akteur mit Hilfe des Systems ausführt, also etwa `Entleihe Buchexemplar`. Beachten Sie, dass sich hinter dieser einfachen Bezeichnung ein ziemlich komplexes Verhalten mit einer Vielzahl von Resultaten verbergen kann. Zum Beispiel erfordert die Entleihe eines Buches die Überprüfung, ob der Entleiher ein Mitglied der Bibliothek ist und ob er oder sie nicht schon die maximale Anzahl von Büchern ausgeliehen hat. Es kann sein, dass der Anwendungsfall endet, ohne dass der Anwender erfolgreich ein Buch ausgeliehen hat, die Benennung des Anwendungsfalles erfolgt aber danach, was im normalen, erfolgreichen Fall geschieht.

Jeder Anwendungsfall sollte im Detail dokumentiert werden, am besten im Aktiv der Dritten Person.[2] Hier ein Beispiel:

- **Entleihe Buchexemplar** Ein `BuchEntleiher` zeigt ein Buch. Das System prüft, ob der potentielle Entleiher auch ein Mitglied der Bibliothek ist und er nicht schon die maximale Anzahl von Büchern ausgeliehen hat. Dieses Maximum ist sechs; für den Fall, dass es ein Bibliotheksmitglied ist, ist das Maximum zwölf. Wenn beide Prüfungen erfolgreich sind, hält das System fest, dass dieses Bibliotheksmitglied dieses Buch ausgeliehen hat. Im anderen Fall verweigert es die Entleihe.

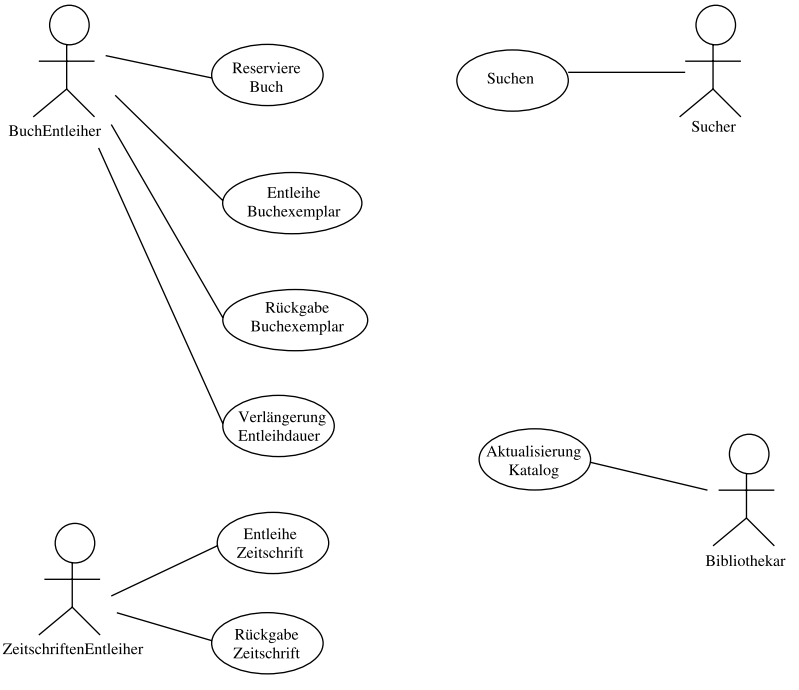
Diskussionsfrage 18

Das Bibliothekssystem in diesem Kapitel ist vereinfacht. Stellen Sie sich Bibliotheken vor, die Sie kennen: Was müsste ein vollständiges System für die Ausführung dieses Anwendungsfalles sonst noch tun oder überprüfen? Erweitern Sie die obige Beschreibung eines Anwendungsfalles entsprechend und denken Sie dabei daran, dass nur Anforderungen enthalten sein sollten, kein Design.

Hinweise zu Benutzerschnittstellen Das Design von Benutzerschnittstellen ist ein sehr großer und hochinteressanter Bereich, gehört jedoch nicht zu diesem Buch. Für den Designer der Benutzerschnittstelle ist es wahrscheinlich sehr wohl von großer Bedeutung, ob ein Buch von einem Bibliotheksmitglied oder einem Bibliothekar ausgeliehen wird. Wir unterstellen hier jedoch, dass es unsere Aufgabe ist, das zugrunde liegende System zu bauen und Funktionalität für eine Benutzerschnittstelle bereitzustellen, die von jemand anderem gebaut wird. Das ist eine nur vernünftige Annahme: Bei Benutzerschnittstellen kommt es viel häufiger vor, dass sie geändert werden müssen, als beim Rest des Systems und eine Trennung zwischen der Benutzerschnittstelle und dem darunter liegenden System erleichtert deren Änderung oder Ersetzung enorm. Darüber hinaus ist es sinnvoll, die Benutzerschnittstelle von jemand entwerfen zu lassen, der Experte auf dem Gebiet ist. Weiterführende Hinweise zum Entwurf von Benutzerschnittstellen finden Sie beispielsweise in [48] oder [16].

[2] Also besser „Das System überprüft“ statt „Überprüfe“ oder „ ... wird vom System überprüft“.

Abbildung 3.1 Das Anwendungsfall-Diagramm für die Bibliothek.



Wir sind jetzt so weit, dass wir unsere Informationen bildlich darstellen können – in einem *Anwendungsfall-Diagramm* (*use case diagramm*) für das System, wie in Abbildung 3.1 gezeigt. Die Notation ist hier selbsterklärend: Strichfiguren stellen Akteure dar, Ovale stehen für Anwendungsfälle und eine Linie zwischen Akteur und Anwendungsfall stellt die mögliche Beteiligung eines Akteurs an einem Anwendungsfall dar.

Die Arbeit mit UML-Diagrammen – sie zu zeichnen, sie gemeinsam zu bearbeiten, sie konsistent zu halten – kann durch ein CASE-Tool (Computer Aided Software Engineering), das UML unterstützt, vereinfacht werden.[3] Für kleinere Systeme tut es aber oft auch schon ein Stück Papier oder die Rückseite des sprichwörtlichen Briefumschlages. Achten Sie darauf, daß Sie Ihre Diagramme nicht zu komplex anlegen – mit einem mächtigen Tool kann das leicht passieren. Ein Diagramm, das zu komplex ist, um es mit der Hand zu zeichnen, ist vermutlich auch zu komplex, um es geistig klar zu erfassen. In einem solchen Fall sollte es aufgeteilt oder auf einem höheren Abstraktionsniveau gezeichnet werden oder sogar beides. Im obigen Beispiel haben wir Tätigkeiten des `Bibliothekar`, wie Bücher aufnehmen oder entfernen oder Zeitschriften versenden, in einem einzigen Anwendungsfall `Aktualisierung Katalog` zusammengefasst. Bei der detaillierten Bearbeitung dieser Funktionalität können wir immer noch mehrere einzelne Anwendungsfälle daraus machen.

Auf dieser Stufe sollte jeder Anwendungsfall dokumentiert werden, zumindest in Stichpunkten. Hier müssen Sie sich nur überlegen, was das System machen soll, nicht, wie es gemacht werden soll. In `Entleihe Buchexemplar` haben wir beispielsweise nicht erwähnt, in welcher Form das System die Information über die Entleihe festhalten soll; wir sagen nur, was festgehalten werden soll.

[3] Wir arbeiten mit Rational Rose, aber es gibt noch viele andere Tools. Auf der englischsprachigen Homepage zu diesem Buch finden Sie dazu einige Links.

? Verfassen Sie kurze Beschreibungen von Anwendungsfällen aus Abbildung 3.1.

Erfinden Sie keine Anforderungen.

Das klingt selbstverständlich – aber wenn Sie Anwendungsfälle genauer studieren, fallen Ihnen sicher noch viele Dinge ein, die das System gut noch erledigen könnte. Es ist sehr wichtig, dass Sie nicht das, was das System tun muss, weil es der Kunde so will, mit dem durcheinander bringen, von dem Sie denken, dass es das System auch noch kann oder können sollte. Es mag hilfreich sein, zu jedem Anwendungsfall eine Liste mit Fragen und zusätzlichen Möglichkeiten zur Diskussion mit dem Kunden anzulegen – bauen Sie aber nie irgendwelche dubiosen Features in die Beschreibung des Anwendungsfalles selbst ein.

3.2 Reichweite (scope) und Iterationen

Jetzt haben wir eine einigermaßen klare Vorstellung davon, wie ein ideales System aussehen könnte. Die Erfahrung hat aber gezeigt, dass es extrem riskant ist, wenn Entwickler denken, sie könnten alles auf einen Schlag erledigen, und das ideale System bereits in der ersten und einzigen Auslieferung herstellten. Will man dieses Risiko in Grenzen halten, setzt man sich besser zum Ziel, in mehreren Schritten oder *Iterationen* zum idealen System zu kommen. Die erste Iteration führt dann zu einem System mit nur der grundlegendsten und wichtigsten Funktionalität; spätere Interationen verbessern dann das System. Im nächsten Kapitel gehen wir weiter auf diesen Punkt ein, aber Sie möchten vielleicht jetzt schon näher darüber nachdenken:

Diskussionsfrage 19

Welche Vorteile und Nachteile sehen Sie im iterativen Ansatz zur Entwicklung des angesprochenen Systems?

Einer der wichtigsten Zwecke von Anwendungsfällen ist es, die Identifikation von zweckmäßigen Trennlinien zwischen Interationen zu unterstützen. Eine Iteration kann so viel über das System liefern, dass bestimmte Anwendungsfälle ausgeführt werden, andere aber nicht.

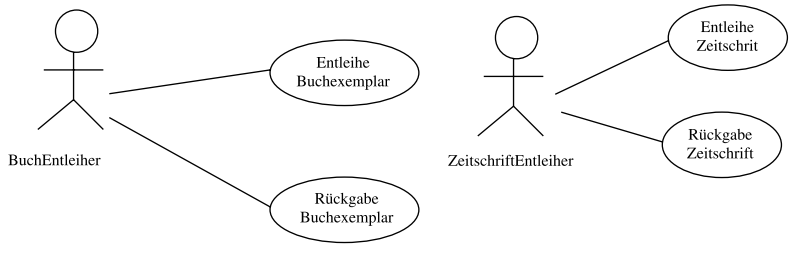
In unserem Fall nehmen wir an, dass wir mit dem Kunden die Prioritäten diskutiert und festgelegt hätten, dass die erste Iteration des Systems die folgenden Anwendungsfälle bieten sollte:

- `Entleihe Buchexemplar`
- `Rückgabe Buchexemplar`
- `Entleihe Zeitschrift`
- `Rückgabe Zeitschrift`

Diese eingeschränkte Version des Anwendungsfall-Diagramms unseres Systems wird in Abbildung 3.2 gezeigt.

Es folgt eine erneute kurze Zusammenstellung der Anforderungen in der ersten Iteration, wobei alles Unwichtige weggelassen wird. In einem echten Projekt kann es erforderlich sein, ein ähnliches Dokument zu erstellen, in dem alle Anwendungsfälle beschrieben werden, die in der ersten Iteration angelegt werden (etwa als Teil der Vertragsunterlagen), oder Sie nehmen sich die Beschreibungen der Anwendungsfälle und stellen nur eine Liste der Namen aller Anwendungsfälle auf.

Abbildung 3.2 Anwendungsfall-Diagramm in der ersten Iteration.



- **Bücher und Zeitschriften** In der Bibliothek gibt es Bücher und Zeitschriften. Von Büchern kann es mehrere Exemplare geben. Einige Bücher sind nur für Kurzzeitentleihe gedacht. Alle anderen Bücher können von jedem Bibliotheksmitglied für drei Wochen ausgeliehen werden. Mitglieder der Bibliothek können normalerweise bis zu sechs Einheiten entleihen, während Mitarbeiter der Bibliothek bis zu zwölf Einheiten zur gleichen Zeit entleihen können. Nur Mitarbeiter der Bibliothek dürfen Zeitschriften entleihen.
- **Entleihe** Das System muss verfolgen können, wann Bücher und Zeitschriften entliehen und zurückgegeben werden, wobei es erzwingt, dass die Regeln, wie oben beschrieben, eingehalten werden.

In einem echten Projekt würden Sie sich zwischendurch auch einmal die Frage nach den zur Verfügung stehenden Mitteln stellen, d.h. in welcher Programmiersprache, mit welchen Tools und Applikationen gearbeitet werden soll. Für die Zwecke dieses Buches nehmen wir an, dass dabei herauskommt, dass das System in einer objektorientierten Sprache implementiert werden soll, und daher für das Design UML eingesetzt wird. Das Thema Datenbank wird hier ignoriert – der Einschub 3.2 gegen Ende dieses Kapitels geht darauf näher ein.

Diskussionsfrage 20

Die von uns vorgeschlagene erste Iteration für dieses System war, wie wir zugeben, von unserem Ziel beeinflusst, in diesem Buch die interessanten OO-Techniken einzuführen, ohne viel Zeit damit zu verbringen, Dinge zu diskutieren, die ausserhalb des Spektrums dieses Buches liegen. Ignoriert man diese Beschränkungen, welche anderen Möglichkeiten für erste Iterationen des Systems zeigen sich dann? Halten Sie diese für besser?

3.3 Klassen identifizieren

In der Sprache der Analyse sprechen wir oft über Abstraktionen im *Haupt-Problembereich* (*key domain abstraction*). Mit dem Wort *Problembereich* (*domain*) ist der Bereich gemeint, mit dem wir arbeiten, hier also beispielsweise die Bibliothek. Wir könnten nun auch von Problembereichs-Klassen sprechen, verwenden aber stattdessen den Ausdruck *Abstraktion*, um damit zu betonen, dass wir nur die Aspekte des Problembereichs beschreiben, die für die Applikation von Bedeutung sind. (Denken Sie dennoch daran, dass wir in Kapitel 2 dargestellt haben, dass alle guten Module, auch Klassen, Abstraktionen sind.) Etwas einfacher ausgedrückt, suchen wir nach den Fakten und Merkmalen der Bibliothek, die in dem System von Bedeutung sind, das wir für ebendiese Bibliothek bauen.

Die richtigen *Klassen* zu identifizieren ist eine der Hauptaufgaben in der objektorientieren Entwicklung. Es ist von zentraler Bedeutung für die Entwicklung von erweiterbarer und wiederverwendbarer Software. Wir beginnen den Prozess der Identifikation der Abstraktionen im Hauptproblembereich mit dem folgenden Ansatz, der auch *Hauptwort-Identifizierungs-Technik* (*noun identification technique*) genannt wird.

Nehmen Sie eine zusammenhängende, kurze Beschreibung der Anforderungen des Systems und unterstreichen Sie seine *Hauptwörter* und *Hauptwort-Ausdrücke*; das heißt, kennzeichnen Sie die Wörter und Ausdrücke, die *Dinge* bezeichnen. Damit erhalten wir eine *Liste der vermutlichen Klassen* (*candidate classes*), die wir noch ein wenig bearbeiten müssen, und haben dann eine erste Liste der Klassen des Systems.

Abbildung 3.3 Hauptwörter und Hauptwort-Ausdrücke in der Bibliothek.

Bücher und Zeitschriften In der Bibliothek gibt es Bücher und Zeitschriften. Von Büchern kann es mehrere Exemplare geben. Einige Bücher sind nur für Kurzzeitentleihe gedacht. Alle anderen Bücher können von jedem Bibliotheksmitglied für drei Wochen entgeliehen werden. Mitglieder der Bibliothek können normalerweise bis zu 6 Einheiten entleihen, während Mitarbeiter der Bibliothek bis zu 12 Einheiten zur gleichen Zeit entleihen können. Nur Mitarbeiter der Bibliothek dürfen Zeitschriften entleihen.

Entleihe Das System muss verfolgen können, wann Bücher und Zeitschriften entliehen und zurückgegeben werden, wobei es erzwingt, dass die Regeln, wie oben beschrieben, eingehalten werden.

Nehmen Sie eine klare, knappe Beschreibung der Anforderungen des Systems und unterstreichen Sie darin die *Hauptwörter* und *substantivischen Ausdrücke*; finden Sie also die Wörter und Wendungen heraus, die Dinge bezeichnen. Dies ergibt eine Liste von *vermutlichen Klassen*, die wir dann bearbeiten und zu einer ersten Klassenliste für das System zurechtstutzen können.

Dies ist nur eine von vielen Techniken, mit denen Sie zur Identifizierung von Klassen arbeiten werden. In Kapitel 5 werden wir noch einige andere Techniken vorstellen, die eingesetzt werden können, um Klassen aufzufinden und zu validieren. In diesem Stadium wollen wir hier nur eine grobe Auflistung der vermutlichen Klassen erhalten.

Wir nehmen also die Sätze aus dem Abschnitt 3.1.1 und unterstreichen darin die Hauptwörter und Hauptwort-Ausdrücke, wie in Abbildung 3.3 zu sehen. (Wir hätten auch die Beschreibungen der Anwendungsfälle verwenden können.)

Als Nächstes sondern wir alle Hauptwörter aus, die ganz offensichtlich aus einer Reihe von Gründen weniger gut als Klassen geeignet sind. (Wenn Sie mit dieser Technik einmal vertrauter sind, werden Sie die Hauptwörter, die offensichtlich ungeeignet sind, gar nicht erst unterstreichen.) Wir betrachten jedes Hauptwort im Singular. In Kapitel 5 werden wir dies allgemeiner und systematischer betrachten. Hier, in diesem speziellen Fall, sondern wir folgende Hauptwörter aus:

- Bibliothek, weil es außerhalb des Bereichs unseres Systems liegt[4];
- Kurzzeitentleihe, weil eine Entleihe genau genommen ein Ereignis ist, eben die Ausgabe eines Buches an einen Benutzer, was nach unserem derzeitigen Ermessen kein brauchbares Objekt für das System sein kann;

[4] Der Frage, ob es ein „Haupt"-Objekt des Systems gibt, wird in Kapitel 5 weiter nachgegangen.

- Mitglied der Bibliothek, weil es redundant ist und das Gleiche bedeutet wie Bibliotheksmitglied, das wir beibehalten und nicht aussondern;
- Woche, weil es eine Zeiteinheit ist, nicht ein Ding;
- Einheit, weil es zu unklar ist; wenn wir es näher untersuchen, sehen wir, dass es für Buch oder Zeitschrift steht;
- Zeit, weil es ausserhalb des Bereichs unseres Systems liegt;
- System, weil es zur Metasprache der Anforderungsbeschreibung gehört und nicht Bestandteil des beschriebenen Problembereichs ist;
- Regel, aus dem gleichen Grund.

Als erster Ansatz einer Liste vermutlicher Klassen bleiben übrig:

- Buch
- Zeitschrift
- Exemplar (von Buch)
- Bibliotheksmitglied
- Mitarbeiter

Ab hier könnten Sie jetzt mit CRC-Karten weiterarbeiten, um die vorläufigen Zuständigkeiten der Klassen festzuhalten und Verbesserungsmöglichkeiten in der Liste auszumachen; wir verschieben dies aber auf Kapitel 5.

Beachten Sie, daß sowohl Bibliotheksmitglied als auch Mitarbeiter Leute sind, die zugleich auch Anwender des zu errichtenden Systems sind. Natürlich müssen Anwender des zu errichtenden Systems nicht immer auch innerhalb des Systems abgebildet werden (und hier haben wir auch die Bibliothekare nicht abgebildet) – man muss in jedem Einzelfall erneut klären, wie weit dies sinnvoll ist. In unserem Fall hier gibt es Beschränkungen, wie viele Bücher ein Bibliotheksmitglied oder ein Mitarbeiter entleihen kann; es ist also klar, dass das System auch Daten über diese Anwender vorhalten muss. Weniger klar ist, welches Verhalten ein Objekt haben muss, das einen Anwender repräsentiert. Hierzu setzen wir eine Technik ein, die oft weiterhelfen kann, aber nicht für jedes System passend ist. Wir führen mit den Objekten des Systems, die Akteure darstellen, Aktionen im Namen dieser Akteure aus. Wenn beispielsweise das Bibliotheksmitglied Jo Bloggs die Kopie eines Buches entleiht, wird die Nachricht `entleihe(dasExemplar)` an das Objekt `Bibliotheksmitglied` geschickt, durch das Jo Bloggs im System abgebildet wird. (Genau genommen wird die Nachricht durch die Benutzeroberfläche versandt, die aber in diesem Kapitel nicht modelliert wird.) Das `Bibliotheksmitglied`-Objekt ist dann zuständig für die Ausführung all dessen, was für die Aufzeichnung (oder Verweigerung) der Entleihe zu tun ist. Dazu sendet es Nachrichten an andere Systemkomponenten.

Dieser Prozess zur Identifizierung von Objekten ist natürlich keine exakte Wissenschaft. Wir hätten die gleichen vermeintlichen Objekte auch aus anderen Gründen ausschließen können. In diesem Stadium müssen wir es auch noch gar nicht absolut fehlerfrei hinbekommen, da wir noch nicht dabei sind, das Design des Systems zu entwerfen – wir *sind* gerade erst soweit, die wichtigsten Objekte der realen Welt innerhalb des Problembereichs (domain) unseres Systems zu identifizieren.

Diskussionsfrage 21

Sind Sie zu einer der getroffenen Entscheidungen anderer Meinung? Warum?

Es ist sehr wichtig, dass klar ist, was mit jedem der Ausdrücke (Klassenname etc.) gemeint ist. Einige Methodiken schreiben sogar vor, dass in diesem Stadium ein *Data-Dictionary-Eintrag* zur Definition jedes Ausdrucks skizziert werden sollte. Nach unserer Erfahrung wird dieser bürokratische Overhead in den meisten Projekten aber weder unterstützt noch benötigt. Wir gehen davon aus, dass jedes Dokument, das Sie zur Festlegung Ihres Designs verwenden (sei es nun ein Whiteboard, die Rückseite eine Briefumschlages, ein richtiges Dokument oder die Datei eines softwarebasierten Designsystems), alle Informationen enthält, die sicherstellen, dass alle am Projekt Beteiligten das gleiche Verständnis der benutzten Ausdrücke haben. In frühen Phasen des Projekts müssen Sie sicherstellen, dass alle darin übereinstimmen, dass ein Ausdruck noch nicht vollständig definiert ist. Was es zu vermeiden gilt, sind Missverständnisse, die entstehen, wenn die Beteiligten zu einzelnen Ausdrücken und Inhalten unterschiedliche Vorstellungen haben.

3.4 Beziehungen zwischen Klassen

Als Nächstes werden wir wichtige reale Beziehungen oder Assoziationen (associations) zwischen unseren Klassen identifizieren und benennen. Wir tun das aus zwei Gründen:

- um unser *Verständnis des Problembereichs* zu klären. Dazu beschreiben wir unsere Objekte im Hinblick darauf, wie sie zusammenarbeiten.
- um die *Kopplung im Zielsystem* zu überprüfen, d.h. sicherzustellen, dass wir bei der Modularisierung unseres Designs sauber vorgehen (siehe Kapitel 1).

Der zweite Punkt hier bedarf noch einer Erklärung. Nimmt man von einem Objekt an, dass es zu einem anderen in enger Beziehung steht, dann ist es wahrscheinlich relativ problemlos, wenn die Klasse, die das eine Objekt implementiert, dies in Abhängigkeit von der Klasse tut, die das andere implementiert, und sie damit eine enge Kopplung eingehen. Dafür gibt es mindestens die folgenden beiden Begründungen:

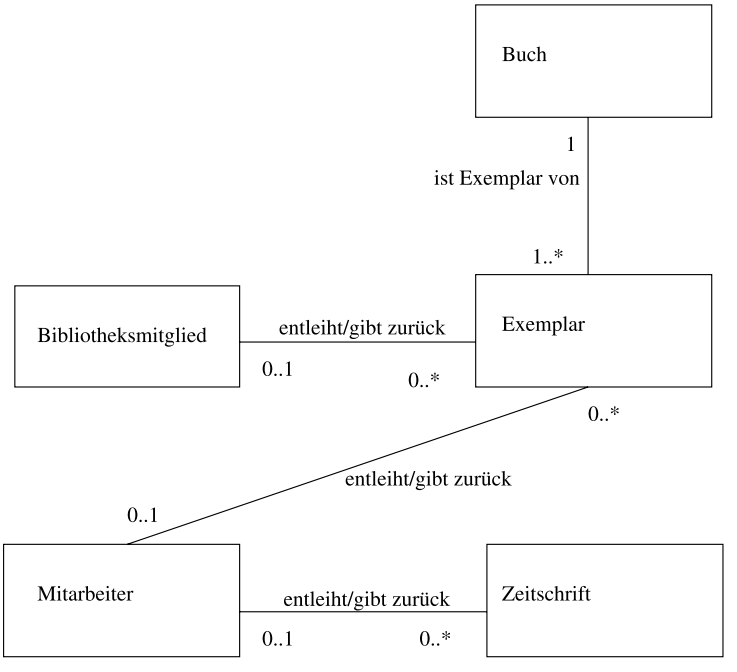
1. Haben Objekte des Problembereichs konzeptuell eine Beziehung zueinander, dann wird eine mit der Wartung betraute Person, die über diese Beziehung weiß, auch eine Abhängigkeit zwischen den entsprechenden Klassen annehmen. Es kann daher ebenfalls angenommen werden, dass diese Person auch auf diese Abhängigkeit achtet, wenn sie Änderungen an einer der Klassen vornimmt. Auf diese Weise ist die Wahrscheinlichkeit, dass aufgrund dieser Abhängigkeit ein Problem auftritt, relativ gering.
2. Sind die Objekte des Problembereichs konzeptionell verwandt, dann ist es wahrscheinlich, dass eine Applikation, die eine der Klassen wiederverwenden kann, auch die andere wiederverwenden kann. Obwohl also jegliche Kopplung in einem System die Wiederverwendung erschwert und komplizierter macht, wird der Effekt hier relativ gutartig sein.

Denken Sie an unsere Behauptung, dass ein Hauptargument für Objektorientierung ist, dass die Struktur eines objektorientierten Systems die Struktur der Wirklichkeit abbildet. Hier ist ein konkretes Beispiel dafür, da wir hier Folgendes deutlich sehen können:

- ein Exemplar *ist ein Exemplar* eines Buches
- ein Bibliotheksmitglied *entleiht/gibt zurück* ein Exemplar
- ein Mitarbeiter *entleiht/gibt zurück* ein Exemplar
- ein Mitarbeiter *entleiht/gibt zurück* eine Zeitschrift

Bildlich ausgedrückt sieht diese Information aus wie in dem UML-Klassenmodell von Abbildung 3.4.

Abbildung 3.4 Erstes Klassenmodell der Bibliothek.



Dieses Klassenmodell enthält genau betrachtet ein paar Informationen mehr, als wir bisher erwähnt haben: Es zeigt die *Multiplizitäten* der Assoziationen. Ein Beispiel: Jedes Exemplar ist ein Exemplar eines einzigen, genau definierten Buches – die Zahl 1 steht also auf der `Buch`-Seite der Assoziation zwischen `Buch` und `Exemplar`. Andererseits kann es aber auch mehr als nur ein Exemplar eines bestimmten Buches geben (gäbe es keine Exemplare, hätte das System auch nichts mit dem Buch zu tun, aber das ist eine Annahme, die näher überprüft werden müsste) – am anderen Ende der Assoziation steht also 1..*. In Kapitel 5 gehen wir näher darauf ein.

Beachten Sie, dass die grafische Darstellung *nichts* darüber aussagt, ob ein `Exemplar`-Objekt etwas über das korrespondierende `Buch`-Objekt weiß, oder umgekehrt oder beides. Die grafische Darstellung sagt also nichts über die Navigierbarkeit der dargestellten Assoziationen.

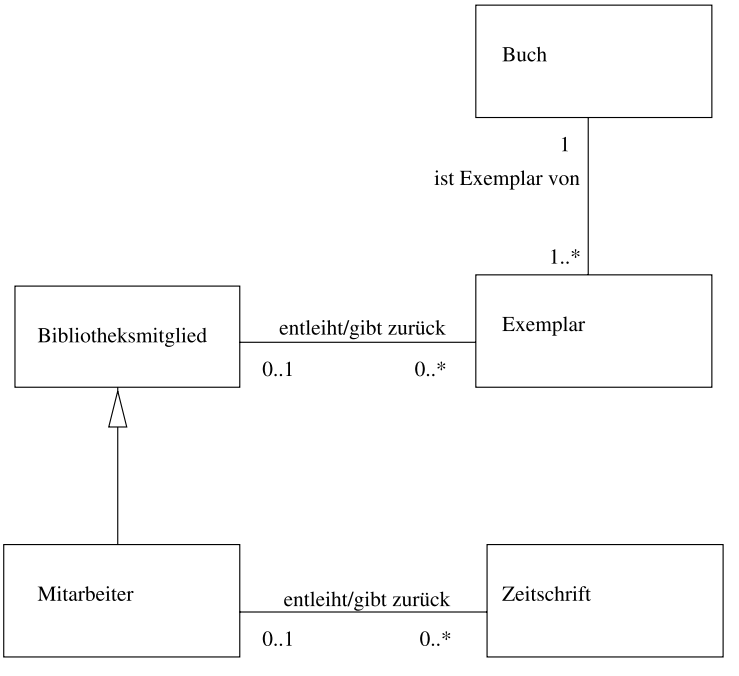
Diskussionsfrage 22

In welcher Richtung müsste die Assoziation Ihrer Meinung nach navigierbar sein, wenn Sie beachten, dass keine unnötige Kopplung eingeführt werden sollte? Überlegen Sie sich das Gleiche noch einmal am Ende dieses Kapitels.

Schließlich stellen wir auch noch fest, dass `Mitarbeiter` die gleichen Assoziationen hat wie auch `Bibliotheksmitglied` und dass dies unsere Intuition bestätigt, nach der ein Mitarbeiter eine Art Bibliotheksmitglied ist. Halten wir dies in unserem Klassendiagramm fest, trägt das zur Klärung unseres Verständnisses der Situation bei: Die Relation zwischen `Bibliotheksmitglied` und `Mitarbeiter` ist eine Generalisierung. Wir könnten diese

Generalisierungsbeziehung mittels Vererbung implementieren (`Mitarbeiter` also als Unterklasse von `Bibliotheksmitglied` anlegen) – diese Designentscheidung basiert aber auf einem tiefergehenden Verständnis des Systems, als wir momentan haben. Setzen wir diese Generalisierung in UML-Notation um, erhalten wir ein Klassenmodell (Abbildung 3.5), das schon viel besser aussieht.

Abbildung 3.5 Das überarbeitete Klassenmodell.



3.5 Das System in Aktion

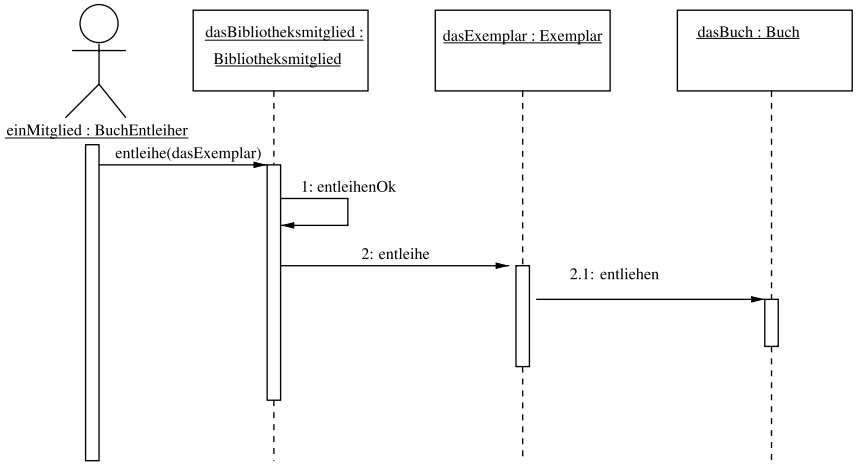
Bis jetzt haben wir die statische Struktur des Systems skizziert, sein dynamisches Verhalten aber noch nicht beschrieben. So wäre beispielsweise noch aufzuzeichnen, wie die Objekte im System zusammenarbeiten und es einem Anwender ermöglichen, ein Buch auszuleihen. Es gibt derzeit noch keinen Zusammenhang zwischen den eingangs erwähnten Anwendungsfällen und den Objekten, aus denen wir das System zusammengestellt haben.

In UML können wir *Interaktionsdiagramme* (*interaction diagram*) verwenden, die zeigen, wie Nachrichten zwischen den Objekten des Systems laufen und einzelne Aktionen, etwa einen bestimmten Anwendungsfall, ausführen. Wir brauchen das nicht für jeden Anwendungsfall durchzuführen und in einigen Projekten wird es überhaupt nicht eingesetzt – Sie sollten ein Interaktionsdiagramm immer dann verwenden, wenn Sie meinen, der Nutzen könnte den Aufwand rechtfertigen. Wenn etwa ein Anwendungsfall besonders kompliziert ist oder es Zweifel gibt, welche von zwei Lösungen die bessere ist, dann kann ein Interaktionsdiagramm helfen, Klarheit zu schaffen.

Als Beispiel wollen wir uns genauer ansehen, was geschieht, wenn ein Bibliotheksmitglied ein Exemplar eines Buches entleiht. In der realen Welt wird das Bibliotheksmitglied wahrscheinlich mit einem tatsächlich vorliegenden Exemplar des Buches in der Hand zu einem

Ausgabeschalter kommen, wo die Benutzeroberfläche des Systems eine Möglichkeit bietet, das Bibliotheksmitglied und das Exemplar des Buches zu identifizieren. (Das kann durch einen Kartenleser oder einen Barcode-Scanner geschehen oder durch eine Person, die die Daten in der Benutzeroberfläche eingibt.) Wir können also damit beginnen, dass wir ein bestimmtes Objekt für das `Bibliotheksmitglied` haben, das wir `dasBibliotheksmitglied` nennen, und ein Objekt für das `Exemplar`, das wir `dasExemplar` nennen.

Abbildung 3.6 Interaktion in einem Sequenzdiagramm.



Erinnern Sie sich, dass wir weiter oben gesagt haben, dass das `Bibliotheksmitglied`-Objekt im Interesse der Bibliotheksmitglieder agiert; die Interaktion beginnt also damit, dass ein `BuchEntleiher`-Akteur eine Nachricht an das Objekt `dasBibliotheksmitglied` schickt. Diese Nachricht enthält, welcher Service verlangt ist: in diesem Fall die Entleihe von `dasExemplar`. Diese Nachricht nennen wir `entleihe(dasExemplar)`. Das System muss nun prüfen, ob `dasBibliotheksmitglied` ein weiteres Buch entleihen darf – dies ist eine der Aufgaben, die von `dasBibliotheksmitglied` auszuführen sind. (Implementiert wird dies wahrscheinlich, indem einfach die Anzahl der von dem Benutzer aktuell ausgeliehenen Dinge, die über ein Attribut der Klasse `Bibliotheksmitglied` festgehalten werden kann, mit der maximal zulässigen Anzahl ausgeliehener Dinge verglichen wird. Darüber entscheiden wir aber mit voller Absicht an dieser Stelle noch nicht.) Darstellen können wir dies, indem wir `dasBibliotheksmitglied` eine Nachricht, etwa `entleihenOK`, an sich selbst schicken lassen.

Als Nächstes müssen wir im System die Information ändern, wie viele Exemplare dieses Buches, nachdem eines ausgegeben wurde, noch in der Bibliothek vorhanden sind. Betroffen davon ist das `Buch`-Objekt, das mit `dasExemplar` über die Assoziation `ist ein Exemplar von` verbunden ist; nennen wir es `dasBuch`. Nehmen wir an, dass `dasBibliotheksmitglied` `dasExemplar` mittels der Nachricht `entleihe` darüber informiert, dass es ausgeliehen wird, und dass `dasExemplar` daraufhin an `dasBuch` die Nachricht, etwa `entliehen()`, schickt, die besagt, dass ein Exemplar des Buches ausgeliehen ist. Mehr muss nicht gemacht werden. All diese Nachrichten werden mit der Antwort, dass kein Fehler vorliegt, erwidert. Die Benutzerschnittstelle wird dem Entleiher aus Fleisch und Blut irgendwie mitteilen, dass alles in Ordnung ist, und dieser wird das tatsächliche Buch dann mitnehmen.

? Hätten wir dieses Verhalten auch noch anders implementieren können? Wäre eine der Alternativen besser gewesen? Warum?

Eine Möglichkeit, dies in UML viel lesbarer als in dem obigen Text zu beschreiben, ist das *Sequenzdiagramm* (*sequence diagram*) aus Abbildung 3.6. Ein Sequenzdiagramm beschreibt einen Ausschnitt aus dem System, typischerweise einen Anwendungsfall oder einen Teil daraus, durch die Darstellung, welche Nachrichten zwischen den Objekten verschickt werden und in welcher Reihenfolge dies geschehen muss (zu lesen ist die Darstellung von oben nach unten).

In einem echten Projekt würden Sie vermutlich bei einer so einfachen Interaktion wie dieser noch kein Sequenzdiagramm entwickeln. Sequenzdiagramme können noch viel mehr aussagen, als hier gezeigt wird; Details hierzu finden Sie in den Kapiteln 9 und 10. In diesem Beispiel etwa müssen die Nachrichten in einer festgelegten Reihenfolge auftreten: Es gibt nur einen *Aktivitätsverlauf* (*thread*) und Aktivitäten können nicht parallel ablaufen. Zudem haben wir uns nur mit der Reihenfolge beschäftigt, in der die Dinge geschehen. Verbindet man die einzelnen Aktivitäten mit Zeiten, ist es auch möglich, sich mit Fragen der Echtzeitverarbeitung zu beschäftigen.

Es gibt noch andere Betrachtungsweisen für diese Interaktionen. In den Kapiteln 9 und 10 werden wir sehen, wie die gleiche Information in einem *Kollaborationsdiagramm* (*collaboration diagram*) dargestellt werden kann. Jedes Vorgehen hat seine eigenen Vorteile.

? Entwickeln Sie ein Sequenzdiagramm für das alternative Szenario, in dem `entleihenOK` ein `false` liefert, weil ein Bibliotheksmitglied bereits die maximale Anzahl an Büchern ausgeliehen hat.

EXKURS 3.1

Design durch Vereinbarung 1

In diesem Exkurs besprechen wir, wie man Entscheidungen über das Verhalten von Operationen detaillierter als nur durch seinen Namen und die Typen seiner Argumente festhalten kann.

Vorbedingungen und Zusicherungen Es ist ein fundamentales Problem, dass Typen in einer Programmiersprache nur eine sehr ungenaue Möglichkeit sind, die Eigenschaften zu beschreiben, die die Attribute und Operationen eines Objekts haben sollten. So hat beispielsweise die Nachricht `entleihenOK` keine Argumente und gibt einen boolschen Wert zurück. Das reicht aber nicht, um die Richtigkeit von `entleihenOK` sicherzustellen. Auch eine Implementierung, bei der immer der Wert `false` zurückgegeben wird, genügt dieser Typspezifikation. Allerdings wird das System mit dieser Implementierung nicht korrekt funktionieren, weil es nie zulässt, dass ein Bibliotheksmitglied ein Buch entleiht. Das Problem hier ist, dass wir zwar in jedem Fall einen boolschen Wert als Rückgabe haben wollen, dies allein aber noch nicht ausreicht. Wir wollen zudem, dass dieser Wert in einer speziellen Weise mit dem Status des Objekts in Verbindung steht.

Wir können alle speziellen Bedingungen oder Einschränkungen (contraints) ausdrücken, indem wir mit *Vorbedingungen* (*preconditions*) oder *Zusicherungen* (*postconditions*) arbeiten.

Eine Vorbedingung beschreibt etwas, was beim Aufruf der Operation erfüllt (true) sein muss – es wäre ein Fehler, die Operation anders aufzurufen. Zu der Vorbedingung kann der aktuelle Zustand des Objekts gehören (die Werte der Attribute) und/oder die Aufrufargumente der Operation. Das heißt, die Vorbedingung beschreibt die Anforderungen der Operation.

Eine Zusicherung beschreibt etwas, was am Ende der Operation erfüllt (true) sein muss, und die Operation ist fehlerhaft implementiert, wenn die Zusicherung sich als falsch herausstellt, obwohl die Vorbedingung erfüllt war. Zur Zusicherung können der Wert, den die Operation zurückgibt, die Argumente der Operation und der Status des Objekts vor und nach der Operation gehören. Das heißt, die Zusicherung beschreibt, was die Operation verspricht.

Stellen Sie sich noch einmal die Operation `entleihenOK` vor, wie sie an `Bibliotheksmitglied` ausgeführt wird. Der Rückgabewert soll wahr sein, solange das Bibliotheksmitglied weniger als die maximal erlaubte Zahl an Büchern ausgeliehen hat, anderenfalls falsch. Wir können zudem sagen, dass eine Implementierung der Operation davon ausgehen kann, dass das Mitglied zu Beginn nicht mehr als die maximal erlaubte Zahl an Büchern ausgeliehen hat. Nehmen wir an, `Bibliotheksmitglied` hätte die Attribute `anzahlDerEntliehenenExemplare : integer` und `maximaleAnzahlEntliehenerExemplare : integer`. Wir könnten das beabsichtigte Verhalten der Operation `entleihenOK` mit einer Vorbedingung und einer Zusicherung normalsprachlich wie folgt dokumentieren:

`entleihenOK`

> pre: `anzahlDerEntliehenenExemplare` ist kleiner oder gleich `maximaleAnzahlEntliehenerExemplare`
>
> post: Rückgabewert ist `true` wenn `anzahlDerEntliehenenExemplare` kleiner als `maximaleAnzahlEntliehenerExemplare`, sonst `false`. Status des Objekts ist unverändert.

Wenn Sie mit vielen Vorbedingungen und Zusicherungen arbeiten, brauchen Sie natürlich eine etwas präzisere Notation als dies hier. Dazu gehören etwa boolsche Ausdrücke einer Programmiersprache oder Aussagen in einer speziellen Sprache. In Kapitel 6, wo noch einmal auf Möglichkeiten eingegangen wird, Einschränkungen (constraints) in UML festzulegen, wird in einem eigenen Exkurs auch auf die darauf spezialisierte Object Constraint Language, OCL, hingewiesen, die sehr gut zusammen mit UML eingesetzt werden kann.

Diskussionsfrage 23

Sind die Argumente, die an eine Operation übergeben werden, oder ihre Rückgabewerte ebenfalls Objekte (was häufig vorkommt) und nicht Werte eines Basistyps wie `integer` oder `boolean`, was sollte dann die Vorbedingung und Zusicherung der Operation über diese Objekte aussagen dürfen?

Klasseninvarianten (class invariants) In unserem Beispiel war die Vorbedingung nicht unbedingt ausschließlich für diese eine Operation wichtig. Es wäre *in jedem Fall* ein Fehler, wenn ein `Bibliotheksmitglied` mehr Bücher ausgeliehen hätte als die zulässige Höchstmenge. Statt dies nun als Vorbedingung jeder Operation einzusetzen,

können wir es viel einfacher als Klasseninvariante (class invariant) festlegen. Das heißt, wir legen als Dokumentation der Klasse `Bibliotheksmitglied` fest, dass bei einem gültigen Objekt dieser Klasse der Wert des Attributs `anzahlDerEntliehenen Exemplare` nicht größer sein darf als der Wert des Attributs `maxAnzahlDer EntliehenenExemplare`. Es ist durchaus denkbar, dass diese Bedingung verletzt wird, wenn sich das Objekt gerade mitten in der Behandlung einer Nachricht befindet, sie muss aber wieder hergestellt sein, wenn die Verarbeitung abgeschlossen ist, und damit immer dann bestehen, wenn eine Nachricht an ein Objekt gesendet wird.

? Wie könnte überprüft werden, ob die Klasseninvariante immer eingehalten wird?

Typ oder Constraint? Es gibt keinen Bedeutungsunterschied zwischen einem Typ und einem Constraint. Sowohl Typ als auch Constraint können etwas aussagen über das, worauf sie sich beziehen. Der Typ eines Objekts beschreibt, welche Attribute und Operationen es hat und welchen Typ *diese* haben. (Weil eine Klassendefinition diese Information ebenfalls gibt, wird der Typ eines Objekts oft mit der Klasse verwechselt; also denken Sie daran, dass die Klasse eines Objekts auch dessen Implementierung spezifiziert.) Ein Constraint – in diesem Fall eine Klasseninvariante – sagt Ihnen noch mehr: beispielsweise könnte sie Informationen über die Verbindung zwischen zwei unterschiedlichen Attributen enthalten.

Auf ähnliche Weise sagt der Typ einer Operation etwas aus über den Typ der Argumente und des Ergebnisses. Ein Constraint auf die Operation hat die Form einer Vorbedingung oder Zusicherung. Genau wie eine Operation nur ausgeführt werden kann, wenn ihr Argumente im richtigen Typ übergeben wurden, kann sie auch nur funktionieren, wenn seine Vorbedingung erfüllt wurde. Und genau wie sie einen Wert im richtigen Typ zurückgeben sollte, muss dieser Rückgabewert auch die Zusicherung erfüllen.

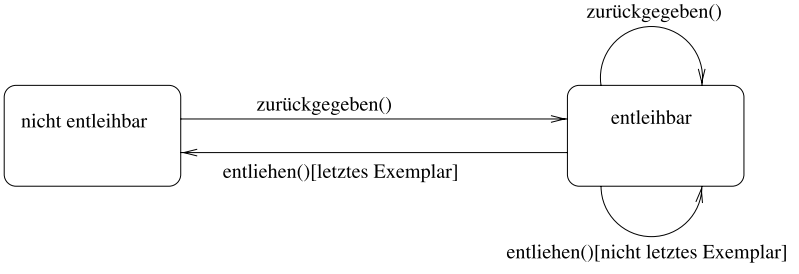
Wenn Sie also angeben möchten, was in einem bestimmten Kontext erlaubt ist, so legen Sie das zunächst lose über das Typensystem Ihrer Programmiersprache fest. Ist das für Ihre Anforderungen nicht präzise genug, so führen Sie noch Constraints ein und drücken damit den Rest dessen aus, was Sie festlegen wollen. Ein Extrem wäre es – etwa bei der Sprache Smalltalk, wo es keine statische Typenfestlegung gibt – alles, was Sie über die passenden Werte eines Ausdrucks sagen wollen, in einem Constraint festzulegen. In Sprachen wie Java und C++, wo es statische Typisierung gibt, können Sie die grundlegenderen Routineüberprüfungen durch den Compiler erledigen lassen. Constraints schreiben Sie nur für die interessanteren Teile.

Der Nachteil von Constraints gegenüber Typen ist, dass die Einhaltung von Constraints normalerweise nicht automatisch überprüft wird, wobei Eiffel hier eine Ausnahme bildet. In mehreren Programmiersprachen können Sie auch Assertions einsetzen. Dies sind boolsche Ausdrücke in Ihrer Programmiersprache, die zur Laufzeit überprüft werden können. Dazu muss ggf. der Debug-Modus eingeschaltet sein.

3.5.1 Änderungen im System: Zustandsdiagramme

Vielleicht ist Ihnen aufgefallen, dass sich der Status des `Buch`-Objektes ändert, wenn ein Exemplar des Buches ausgeliehen wurde. Das `Buch`-Objekt kann sich dann von entleihbar auf nicht entleihbar ändern. (Das ist in diesem Iterationsschritt noch nicht so wichtig, wird aber noch wichtig werden, wenn wir in zukünftigen Iterationen ein Blättern im Katalog oder Reservierungen einbauen.) Aufzeichnen können wir diese Zustandsänderung in einem *Zustandsdiagramm* wie in Abbildung 3.7.

Abbildung 3.7 Zustandsdiagramm der Klasse `Buch`.



Diese Notation zeigt, dass die Frage, ob die Entleihe eines Exemplars des Buches eine Zustandsänderung von `Buch` auslöst, davon abhängt, ob in der Bibliothek noch weitere Exemplare des Buches vorhanden sind.

Diskussionsfrage 24

Haben wir jetzt schon genügend Information festgehalten? Halten Sie es für erforderlich, zu wissen, *wie viele* Exemplare des Buches für die Entleihe verfügbar sind? Welche Fragen würden sich stellen, versuchte man, dies im Modell abzubilden?

3.6 Weitere Arbeiten

Bei der Entwicklung der Sequenz- und Zustandsdiagramme haben wir bestimmte Nachrichten ausgemacht, die unsere Objekte verstehen müssen, d.h. bestimmte Operationen auf unsere Klassen. Dies müssen wir noch ein wenig genauer betrachten und uns mit den anderen Interaktionen des Systems beschäftigen. Wir können auch die in den Objekten gekapselten Daten identifizieren, einschließlich der Daten, die die gezeigten Assoziationen und Referenzen zu anderen Objekten implementieren. Diese ganze Information kann dem Klassenmodell hinzugefügt werden – die Attribute und Operationen einer Klasse können im Klassensymbol aufgelistet werden. Die Besprechung dessen, wie man dies in UML macht, verschieben wir auf Kapitel 5.

Haben wir einmal festgelegt, wie diese Anwendungsfälle realisiert werden, bis zu der Ebene, welche Nachrichten verschickt werden und in welcher Reihenfolge, dann können die Klassen relativ schnell implementiert werden.

Damit sind wir in der ersten Iteration unseres Systems angelangt, wie sie dann auch schon von den Entwicklern und Anwendern getestet werden kann, um mögliche Missverständnisse und Fehler zu identifizieren. Weitere Iterationsschritte in der Entwicklung führen dann zu einem System, das sich dem Ideal immer weiter annähert.

EXKURS 3.2

Persistenz

Bis jetzt haben wir noch nicht über die große Bedeutung von Persistenz gesprochen. Es muss natürlich möglich sein, ein System zu beenden und später wieder zu starten, ohne dass die Informationen über Bibliotheksmitglieder, Bücher und Zeitschriften etc. verlo-

ren gehen. Objekte, die länger existieren müssen als die Laufzeit des Programms, werden *persistent* genannt. Persistente Objekte können auch potentiell von unterschiedlichen Programmen genutzt werden, doch darum geht es hier nicht.

In ganz einfachen Fällen werden Objekte einfach „roh“ in Dateien abgespeichert – ein Objekt kann sich selbst in eine Datei schreiben und kann aus dieser Datei auch wieder rekonstruiert werden. Die Details, wie dies zu tun ist, sind abhängig von der verwendeten Sprache. Viele, aber nicht alle objektorientierten Sprachen bieten hierfür einen Standardmechanismus an, bei denen sich der Entwickler auch über Fragen des Dateiformats keine Gedanken zu machen braucht. Dieses System eignet sich allerdings nicht für Systeme, die eine weniger triviale Objektpersistenz erfordern – es ist zu wenig flexibel und effizient.

? Wie kann diese Art der Speicherung in Ihrer Programmiersprache durchgeführt werden? (Hinweis: Sehen Sie sich in Java das `Serializable`-Interface an, in Smalltalk die Methode `storeOn:aStream` und in C++ Ihre Klassenbibliothek.) Schreiben Sie ein kleines Programm, mit dem Sie ein Objekt anlegen, seinen Zustand ändern, es dann in eine Datei schreiben und das Objekt schließlich aus dieser Datei rekonstruieren.

? Welche Nachteile hat diese Methode? Denken Sie etwa an die Arbeit mit Objekten, die Verweise auf andere Objekte enthalten.

Eine ernsthaftere Option für die Objektpersistenz wäre eine *Datenbank*, die viel mehr bieten kann als nur Persistenz. Eine Datenbank kann beispielsweise auch *Transaktionen* unterstützen – eine Reihe von Änderungen, die nur gemeinsam durchgeführt werden dürfen oder gemeinsam scheitern. Die am weitesten verbreitete Lösung ist die Verwendung einer *relationalen* Datenbank, was aber sicher nicht die beste Lösung ist, weil es große *konzeptionelle Unterschiede* zwischen SQL und einer objektorientierten Programmiersprache gibt. Daten werden dabei in Tabellen abgelegt und müssen erst noch in die Objektdarstellung übersetzt werden, die in objektorientierten Systemen üblich ist. Der Entwickler kann nicht einfach nur bestimmte Objekte als persistent kennzeichnen und erwarten, dass Änderungen daran von Dauer wären. Es gibt jedoch immer mehr Unterstützung durch Software-Tools, und die neuesten Entwicklungen in relationalen Datenbanksystemen (RDBMS), insbesondere bei SQL3, können diese Probleme vereinfachen. Die Alternative ist die Verwendung von objektorientierten Datenbanken (OODBs). Diese unterstützen explizit Dinge wie Objektorientierung und Vererbung und ermöglichen meist auch einen flexibleren Zugriff auf die Daten als dies in RDBMS möglich ist. Noch haben objektorientierte Datenbanksysteme allerdings einen sehr kleinen Marktanteil. Weitere Details zu OODBs gehen über dieses Buch hinaus. Ein gutes und auch sehr gut lesbares Buch dazu ist [31].

? Überlegen Sie sich für die Datenbanken, zu denen Sie Zugang haben, wie diese für die hier beschriebene Anwendung eingesetzt werden könnten.

Diskussionsfrage 25

Welche Faktoren haben Einfluss darauf, ob es wünschenswert sein kann, das beschriebene Bibliothekssystem mit anderen Systemen an der Universität, etwa der Studentenverwaltung, zu integrieren?

ZUSAMMENFASSUNG

Dieses Kapitel beschrieb eine einfache Fallstudie. Dabei wurden die hauptsächlichen Eigenschaften von UML skizzenhaft dargestellt. Teil 2 des Buches geht detailliert auf jeden Diagrammtyp ein, aber da die Diagramme angelegt sind, um zusammen in einem iterativen Entwicklungsprozess eingesetzt zu werden, müssen wir einen Überblick über UML haben, bevor wir in die Details gehen.

Beschrieben haben wir eine einzelne *Iteration* in der Entwicklung des Systems und wir haben die eingesetzten Techniken nicht weiter bewertet. Im nächsten Kapitel sprechen wir über den Entwicklungsprozess.

DISKUSSIONSFRAGEN

3.1 In diesem Kapitel haben wir uns nur auf einen Fall – die Entleihe – konzentriert. Entwerfen Sie ähnliche Diagramme für die anderen Anwendungsfälle.

3.2 Für wie wichtig halten Sie den Aufwand, dieses Modell zu entwerfen? Erinnern Sie sich daran, wie Sie das Problem angegangen hätten, bevor Sie dieses Kapitel gelesen haben, und überlegen Sie sich, in wie weit Sie von diesem neuen Verständnis profitieren können?

3.3 Bibliothekare scheinen hier nur dafür da zu sein, um die Anwendungsfälle der Bibliotheksmitglieder bedienen zu können. Wie könnten wir deren eigene Anwendungsfälle modellieren, wozu auch die Aufnahme und Entfernung von neuen Büchern und weiteren Exemplaren existierender Bücher gehört? Wie weit würde diese Erweiterung unseres Designs eine Änderung des bestehenden Modells erfordern?

3.4 Was halten Sie von unserer bisherigen Behandlung von Zeitschriften?

3.5 Wir haben die Akteure der Angestellten bisher ähnlich wie die Bibliotheksmitglieder modelliert, Angestellte dürfen aber mehr Bücher entleihen als normale Bibliotheksmitglieder. Ist das ein Problem?

3.6 Wie könnte das System mit Büchern umgehen, die aus mehr als einem Band bestehen, oder mit Büchern, zu denen eine CD oder ein Tonband gehören? Zählen Sie einige Optionen auf und überlegen Sie, welche Vor- und Nachteile davon Sie mit Ihren Kunden besprechen würden.