Objektorientierte Modellierung

a) Was bedeutet Objektorientierung?

Objektorientierung bedeutet das Zusammenführen von Daten und Programmen (Funktionen) in Form von Objekten.

- Ein **Objekt** besitzt Eigenschaften, das sind **Attribute** und **Operationen**, die entweder von außen zugreifbar oder nur innerhalb des Objekts sichtbar sind. Objekte kommunizieren durch den Austausch von **Nachrichten** miteinander. Eine von einem Objekt empfangene Nachricht löst in der Regel die Ausführung einer Operation auf dem betreffenden Objekt aus.
- **Objektorientierte Entwicklung** heißt daher, das zu realisierende (Informations-)system als eine Menge kommunizierender Objekte zu modellieren und zu implementieren.

Der Einsatz objektorientierter Konzepte wurde zuerst in **Programmiersprachen** verwirklicht, und zwar bereits mit SIMULA in den späten 60er Jahren. Während aber SIMULA noch in erster Linie für die Entwicklung von Simulationssoftware eingesetzt wurde und daher unverdientermaßen einen geringen Verbreitungsgrad hatte, ist der Siegeszug der objektorientierten Nachfolgesprachen, allen voran Smalltalk, C++ und Java, bereits Geschichte. Zu den Wurzeln der Objektorientierung gehören aber auch die Bereiche des *Artificial Intelligence* und der Datenbanksysteme. Innerhalb der *Artificial Intelligence*-Forschung war es vor allem die **Wissensrepräsentation** mit Sprachen wie KL-ONE, die als Vorläufer der Objektorientierung gelten. Im Bereich der Datenbanksysteme sind es vor allem die Konzepte der (semantischen) **Datenmodellierung,** wie z. B. Aggregation und Generalisierung (vgl. *Entity-Relationship*-Model in Unterkapitel XXX), die sich in den objektorientierten Konzepten wiederfinden.

b) Was bedeutet objektorientierte Modellierung?

Objektorientierte Modellierung bedeutet, ausgehend von der Problemstellung, den Problembereich (*universe of discourse*) als eine Menge kommunizierender und interagierender Objekte zu beschreiben, die sukzessive in den Lösungsbereich, d.h. in das zu implementierende System, übergeführt werden. Dabei umfasst objektorientierte Modellierung sowohl die frühen Phasen der Softwareentwicklung wie Anforderungsbeschreibung und Analyse, als auch die späten Phasen wie Systementwurf und Detailentwurf (vgl. auch Planungs-, Definitions- und Entwurfsphase des Phasenkonzepts in Unterkapitel XXX).

Der entscheidende Vorteil der objektorientierten Modellierung gegenüber der traditionellen strukturierten Analyse und des strukturierten Entwurfs ist dabei die **Durchgängigkeit der Konzepte**. D.h., Objekte und ihre Eigenschaften werden im Problembereich identifiziert, mit Hilfe einer Modellierungssprache modelliert, im Laufe des Entwicklungsfortschritts verfeinert und schließlich in einer (hoffentlich) objektorientierten Implementierungsumgebung (Sprache, Datenbanksystem, Middleware etc.) umgesetzt.

c) Warum UML?

Bis Mitte der 90er Jahre waren weit **über ein Dutzend objektorientierter Modellierungsmethoden** am Markt. Abhängig von der Vergangenheit der jeweiligen Entwickler hatten die Methoden einen starken Datenmodellierungsbezug oder einen starken Programmiersprachenbezug. Die Hauptvertreter waren

- **OMT von Rumbaugh** et al.,
- die **Booch-Methode** und
- **OOSE von Jacobson**.

OMT war ein klarer Vertreter der ersten Gruppe, nämlich mit einer engen Verbindung zur Datenmodellierung. Die Entwickler von OMT wollten vor allem die Modellierung von komplexen Objekten im Sinne einer objektorientierten Erweiterung des *Entity-Relationship*-Modells (vgl. Unterkapitel XXX) unterstützen. Die Booch-Methode hatte aufgrund ihrer Ada-nahen Vergangenheit einen starken Bezug zur dynamischen Modellierung und zu Programmiersprachenkonzepten. Sie war besonders zur Modellierung von Echtzeitsystemen und nebenläufigen Systemen geeignet. OOSE schließlich fiel etwas aus dieser Klassifikation heraus, indem es als der Vertreter der Skandinavischen Schule angesehen werden kann, bei der die Modellierung und die Simulation von Vorgängen der realen Welt im Vordergrund stehen. OOSE wurde ursprünglich zur Modellierung von Telekommunikationssystemen entwickelt und wies daher einen engen Bezug zu den dort eingesetzten Modellierungstechniken auf.

Um die Aufgaben der objektorientierten Modellierung nicht durch eine reine Notationsdiskussion zu überfrachten, hatte die OMG (*Object Management Group*), das wichtigste Standardisierungsgremium für objektorientierte Entwicklung, im Jahre 1996 einen Aufruf zur Spezifikation eines Modellierungsstandards erlassen. Booch, Jacobson und Rumbaugh arbeiteten zu diesem Zeitpunkt bereits an einer Vereinheitlichung ihrer Ansätze. Deren Vorschlag wurde am 17. 11. 1997 als ***Unified Modeling Language*** **(UML)** in der Version 1.1 von der OMG als Modellierungsstandard akzeptiert. Zurzeit (Sommer 2000) existiert UML in der **Version 1.3**, deren Konzepte auf den folgenden Seiten beschrieben werden.

● UML ist eine **grafische Modellierungssprache** zum Spezifizieren, Konstruieren, Visualisieren und Dokumentieren von Softwaresystemen.

Ein solches System wird nicht »in einem Wurf« entwickelt, sondern im Verlauf eines Entwicklungsprozesses stufenweise verfeinert und aufgebaut. Die dabei entstehenden grafischen Repräsentationen beschreiben auf vorgegebenen Abstraktionsniveaus einen bestimmten Aspekt des Problembereichs oder des zu realisierenden Systems, wie z.B. die involvierten Objekte und ihre Beziehungen zueinander, die Interaktionen zwischen den Objekten und die Verteilung auf unterschiedliche Rechner. Diese Diagramme bieten auch Hilfestellung bei der Kommunikation zwischen Entwicklern und Kunden und zwischen den Entwicklern untereinander. Die Softwaretechnik folgt damit dem guten Beispiel anderer technischer Disziplinen, wie z.B. der Elektrotechnik und der Architektur, die sich längst grafischer Modellierungssprachen bedienen. So folgt der Schaltplan eines Videorecorders und der Plan eines Hauses einer international standardisierten Syntax und Semantik und ist damit »für jedermann« lesbar. In diesem Sinn ist UML die **Lingua franca der objektorientierten Softwareentwicklung**, indem es als Modellierungssprache dem Entwickler wesentliche Modellierungskonzepte und eine intuitive grafische Repräsentation derselben für die Konstruktion objektorientierter Softwaresysteme in die Hand gibt. UML unterscheidet **acht Diagrammarten**:

- Anwendungsfalldiagramm
- Klassendiagramm
- Sequenzdiagramm
- Kollaborationsdiagramm
- Zustandsdiagramm
- Aktivitätsdiagramm
- Komponentendiagramm
- Verteilungsdiagramm

Während das Anwendungsfalldiagramm und das Klassendiagramm die **statische Struktur** des zu entwickelnden Systems beschreiben, werden die vier Diagramme Sequenzdiagramm, Kollaborationsdiagramm, Zustandsdiagramm und Aktivitätsdiagramm zur Darstellung der **dynamischen Struktur** herangezogen. Komponentendiagramm und Verteilungsdiagramm werden auch als Implementierungsdiagramme bezeichnet, da sie zur **Visualisierung der Hardware- und Softwaretopologie** auf der Ebene der Implementierung dienen.

Welche Diagramme bei der Entwicklung eines Softwaresystems konstruiert werden, hängt nicht zuletzt auch von der Art des zu entwickelnden Systems und vom zugrundeliegenden Entwicklungsprozess ab. Von UML wird **kein Entwicklungsprozess vorgeschrieben**, vielmehr ist UML mit jedem beliebigen Entwicklungsprozess kombinierbar (so z.B. mit dem Phasenkonzept aus Unterkapitel XXX).

Die acht Diagramme von UML werden anhand der **Entwicklung eines Bibliothekssystems** vorgestellt. Mit dem Bibliothekssystem sollen so unterschiedliche Werke wie Bücher, CD-ROM's und Zeitschriften sowohl beschafft als auch entlehnt werden können.

d) Anwendungsfalldiagramm (*Use Case Diagram*)

Auch wenn in der objektorientierten Entwicklung die identifizierten Objekte, ihre Beziehungen zueinander und ihre Interaktionen im Mittelpunkt stehen, so ist für den Auftraggeber und Benutzer eines (Informations-)systems die Funktionalität dieses Systems am wichtigsten, wie auch immer diese Funktionalität realisiert wird.

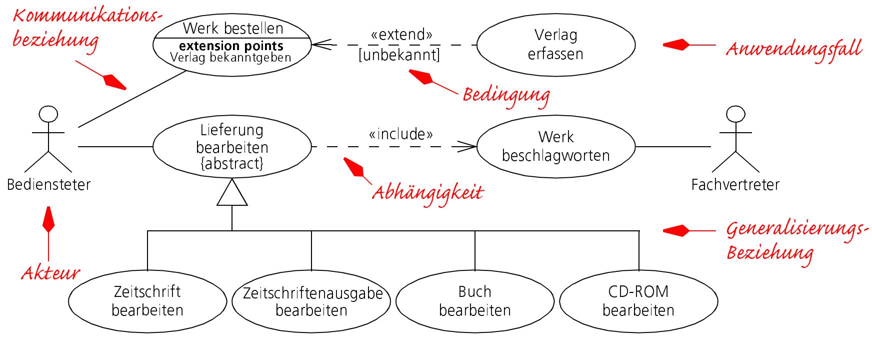
- Das **Anwendungsfalldiagramm** dient zur Beschreibung der geforderten Funktionalität des zu entwickelnden Gesamtsystems in den frühen Phasen der Softwareentwicklung. Es besteht aus einer Menge von Anwendungsfällen und Schnittstellen zur Außenwelt in Form von kommunizierenden Akteuren.

- Ein **Anwendungsfall** (*use case*) beschreibt ein bestimmtes Verhalten, das von dem zu entwickelnden System erwartet wird. Er wird durch eine mit dem Namen des Anwendungsfalls beschriftete Ellipse dargestellt.

Alle Anwendungsfälle zusammen machen die Funktionalität des Gesamtsystems aus. Wer mit dem System interagieren soll, wird durch Akteure festgelegt.

- **Akteure** (*actor*) stehen klar außerhalb des Systems. Sie **benützen das System**, indem sie die Ausführung von Anwendungsfällen initiieren, oder sie **werden vom System benützt**, indem sie selbst Funktionalität zur Realisierung einzelner Anwendungsfälle zur Verfügung stellen. UML kennt für Akteure zwei Darstellungsformen. Entweder das Rechteck, das mit dem Namen des Akteurs und dem Schlüsselwort **«actor»** beschriftet ist oder die piktografische Darstellung in Form eines beschrifteten Strichmännchens.

☐ Anwendungsfalldiagramm `Beschaffung`:
Das zu realisierende Bibliothekssystem muss sowohl die Beschaffung von neuen Werken als auch die Entlehnung von Werken unterstützen. Aus Platzgründen wird im Anwendungsfalldiagramm nur die Beschaffung näher spezifiziert. Zur Beschaffung gehören die Anwendungsfälle `Werk bestellen`, `Verlag erfassen`, sofern er dem System noch nicht bekannt ist, `Lieferung bearbeiten` und das erworbene `Werk beschlagworten`. Die Beschaffung wird von einem Bediensteten der Bibliothek durchgeführt, der sich des Systems bedient (Akteur `Bediensteter`). Zur Beschlagwortung wird auch das Fachwissen eines Fachvertreters herangezogen (Akteur `Fachvertreter`). Beim Bearbeiten der Lieferung gibt es Tätigkeiten, die von der Art des Werks unabhängig sind (z.B. Eingang bestätigen), und werkspezifische Tätigkeiten. Letztere werden in den spezialisierten Anwendungsfällen `Zeitschrift bearbeiten`, `Zeitschriftenausgabe bearbeiten`, `Buch bearbeiten und CD-ROM bearbeiten` behandelt.



Da bei der Modellierung zwischen menschlichen und nicht menschlichen Akteuren unterschieden werden kann, gilt als Konvention, dass die Strichmännchendarstellung für menschliche Akteure (vgl. `Bediensteter` im Anwendungsfalldiagramm) und die Rechtecksdarstellung für nicht menschliche Akteure (vgl. `Drucker` im Sequenzdiagramm) verwendet wird.

- Ein **Schlüsselwort** (*keyword*) dient der textuellen Differenzierung von UML-Konstrukten, z.B. Klassen, Interfaces und Akteure, die aber alle durch dasselbe grafische Symbol (in diesem Fall ein Rechteck) dargestellt werden. Um sie zu unterscheiden, wird ein Schlüsselwort in doppelten Spitzklammern («...») neben dem Namen des jeweiligen UML-Konstrukts vermerkt.

Jeder Akteur muss mindestens eine Kommunikationsbeziehung zu einem Anwendungsfall haben, er kann aber auch mit mehreren Anwendungsfällen kommunizieren.

- Eine **Kommunikationsbeziehung** (*communication relationship*) wird zwischen genau einem Akteur und genau einem Anwendungsfall spezifiziert und ist ungerichtet, d.h. die Kommunikation kann in beiden Richtungen erfolgen.

Darüber hinaus können Anwendungsfälle auch untereinander in Beziehung stehen, wobei man drei Beziehungsarten unterscheidet.

- Die ***include*-Beziehung** (*include relationship*) ist eine gerichtete Beziehung zwischen zwei Anwendungsfällen und besagt, dass das Verhalten vom zu inkludierenden Anwendungsfall in den inkludierenden Anwendungsfall eingefügt wird (entspricht einem Unterprogrammaufruf in der prozeduralen Programmierung). Die *include*-Beziehung wird durch einen gestrichelten **Abhängigkeitspfeil** dargestellt, der mit dem Schlüsselwort **«include»** beschriftet ist.

Streng genommen müsste man daher von einer »*include*-Abhängigkeit« reden, doch hat sich der Begriff der Beziehung in diesem Kontext in der Literatur durchgesetzt. Dies ist auch insofern korrekt, als dass in UML eine Abhängigkeit eine spezielle Form einer Beziehung darstellt.

- Die ***extend*-Beziehung** (*extend relationship*) ist ebenfalls eine gerichtete Beziehung zwischen zwei Anwendungsfällen und besagt, dass das Verhalten des erweiternden Anwendungsfalls in den zu erweiternden Anwendungsfall eingefügt werden kann (aber nicht muss!), vorausgesetzt, dass eine etwaig spezifizierte Bedingung erfüllt ist. Die *extend*-Beziehung wird durch einen mit dem Schlüsselwort *«extend»* beschrifteten Abhängigkeitspfeil dargestellt. Die **Bedingung** (*condition*) wird in eckigen Klammern ebenfalls zur *extend*-Beziehung vermerkt. Für jede *extend*-Beziehung können außerdem eine oder mehrere **Erweiterungsstellen** (*extension points)* im zu erweiternden Anwendungsfall definiert werden, die angeben, wo der erweiternde Anwendungsfall einzufügen ist.

- Bei der **Generalisierungsbeziehung** (*generalisation relationship*) erbt ein Anwendungsfall das gesamte Verhalten inklusive Kommunikationsbeziehungen von einem anderen Anwendungsfall, wobei er das geerbte Verhalten verändern und ergänzen kann. Die Generalisierung wird durch einen Pfeil mit einem nicht ausgefüllten gleichseitigen Dreieck als Spitze notiert, der von dem spezielleren zum generischeren Konstrukt führt.

Generalisierungsbeziehungen können nicht nur zwischen Anwendungsfällen, sondern auch z.B. zwischen Klassen und zwischen Paketen spezifiziert werden (siehe auch weiter unten).

UML erlaubt es, UML-Konstrukte durch so genannte Eigenschaften näher zu beschreiben.

- Eine **Eigenschaft** (*property*) ist entweder vordefiniert oder benutzerdefiniert und wird in geschwungenen Klammern neben dem Namen des betroffenen Konstrukts angeführt.

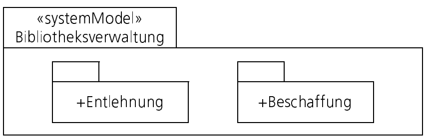
☐ Werden idente Teilaspekte von unterschiedlichen Anwendungsfällen in einem eigenen Anwendungsfall zusammengefasst, der aber für sich genommen keine ausführbare Funktionalität darstellt, so spricht man von einem **abstrakten Anwendungsfall** (*abstract use case*). Ein solcher wird durch Angabe der vordefinierten Eigenschaft **{abstract}** modelliert (vgl. `Lieferung bearbeiten` im Anwendungsfalldiagramm `Beschaffung`).

e) Pakete

Für nichttriviale Problemstellungen können die einzelnen Diagramme schnell sehr unübersichtlich werden. Für solche Fälle kennt UML den Abstraktions- bzw. Strukturierungsmechanismus des Pakets.

- Ein **Paket** (*package*) ermöglicht es, eine beliebige Anzahl von UML-Konstrukten und -Diagrammen zu gruppieren und davon zu abstrahieren. Dabei können Pakete selbst wieder Pakete beinhalten. Ein Paket wird in UML durch ein Rechteck mit aufgesetztem kleinen Rechteck dargestellt (»Karteikarte mit Reiter«). Wird der Paketinhalt nicht gezeigt, so steht der Paketname innerhalb des großen Rechtecks. Wird der Paketinhalt dargestellt, so steht der Paketname im Reiter.

☐ Paket `Bibliotheksverwaltung`:
Das gesamte zu realisierende Bibliothekssystem wird als Paket `Bibliotheksverwaltung` mit dem vordefinierten Stereotyp **«systemModel»** modelliert und enthält alle für die Beschreibung des zu implementierenden Systems relevanten Informationen. Dieses spezielle Paket ist die Wurzel aller geschachtelten Pakethierarchien eines Systems. Es ist das einzige Konstrukt in UML, das selbst nicht als Teil eines anderen Konstrukts auftreten muss. Im Beispiel enthält das Paket `Bibliotheksverwaltung` die beiden Pakete `Entlehnung` und `Beschaffung`. Das Paket `Beschaffung` besteht u.a. aus jenem Teil des Anwendungsfalldiagramms, der im vorangegangenen Beispiel diskutiert wurde. Im Paket `Entlehnung` wiederum sind alle jene Informationen gekapselt, die den Entlehnvorgang genauer spezifizieren.



Die im Beispiel vorkommenden Stereotype stellen einen Erweiterungsmechanismus in UML dar.

- Ein **Stereotyp** (*stereotype*) ermöglicht die Kategorisierung von UML-Elementen. Ein Stereotyp wird textuell durch ein Schlüsselwort (s.o.) oder grafisch durch ein Piktogramm repräsentiert. Bezüglich der textuellen Darstellung wird der Name des Stereotyps in doppelten Spitzklammern (**«...»**) in der Nähe des Namens des betroffenen UML-Elements angegeben.

☐ Das Paket `Bibliotheksverwaltung` wird durch das vordefinierte Stereotyp **«systemModel»** der Paketkategorie Systemmodell zugeordnet.

Damit der Paketinhalt in kontrollierter Weise von außen sichtbar und damit auch zugreifbar wird, werden Sichtbarkeiten der einzelnen Elemente und Beziehungen zwischen den Paketen festgelegt.

● Die **Sichtbarkeit** (*visibility*) wird als Präfix zum Namen des betreffenden Elements hinzugefügt. Sie gibt die Sichtbarkeit des Elements außerhalb seines umschließenden Pakets an:

- **+** ... öffentlich sichtbar,
- **#** ... geschützt sichtbar, d.h. nur für erbende Pakete sichtbar,
- **–** ... außerhalb des Pakets nicht sichtbar (»privat sichtbar«).

Bei den Beziehungen zwischen Paketen unterscheidet man die Genehmigungsabhängigkeit und die Generalisierungsbeziehung.

● Die **Genehmigungsabhängigkeit** (*permission dependency*) eines Pakets von einem anderen Paket impliziert, dass die Elemente des ersten, »abhängigen« Pakets alle Elemente mit ausreichender Sichtbarkeit des zweiten, »unabhängigen« Pakets referenzieren können. Die Genehmigungsabhängigkeit existiert in **zwei Ausprägungen** und wird durch einen Abhängigkeitspfeil mit den Stereotypen **«access»** bzw. **«import»** von einem Paket zu einem anderen zum Ausdruck gebracht.

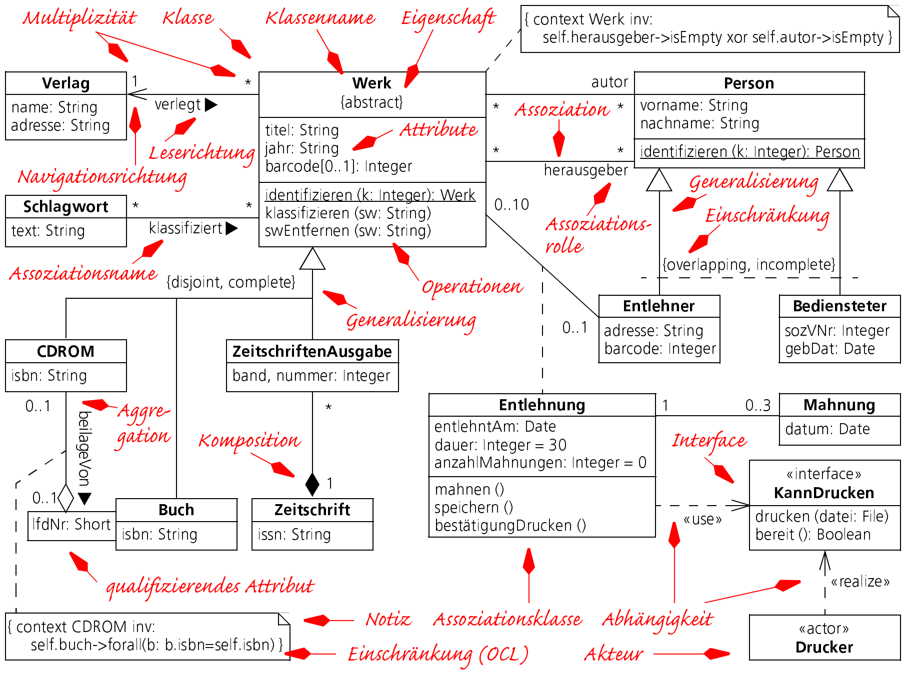
● Die **Generalisierungsbeziehung** (*generalisation relationship*) von einem Paket zu einem anderen bedeutet, dass ersteres alle öffentlich und geschützt sichtbaren Elemente des zweiten Pakets erbt und diese dann wie seine eigenen Elemente referenzieren kann.

f) Klassendiagramm (*Static Structure Diagram*, *Class Diagram*)

● Das **Klassendiagramm** zeigt im Wesentlichen Klassen und deren Beziehungen, kann allerdings auch einige andere »klassenähnliche« Konstrukte enthalten (Interfaces, Klassenschablonen, Typen, Referenzen auf Entwurfsmuster u.v.m.). Darüber hinaus können auch konkrete Instanzierungen, also Objekte und Objektbeziehungen repräsentiert werden, um bestimmte Sachverhalte exemplarisch zu illustrieren.

Auf Grund dieser Mannigfaltigkeit lautet die offizielle Bezeichnung des Klassendiagramms in UML »***static structure diagram***«.

☐ Ausschnitt aus einem Klassendiagramm für das Bibliothekssystem (Detaillierungsniveau gemäß einer frühen Phase des Softwareentwicklungsprozesses):



● Eine **Klasse** (*class*) definiert die strukturellen Eigenschaften (Attribute) und das Verhalten (Operationen) einer Menge gleichartiger Objekte (Instanzen der Klasse). Das Rechtecksymbol für eine **Klasse** ist in Abschnitte untergliedert. Mit Ausnahme des obersten dürfen alle Abschnitte fehlen. Im obersten Abschnitt sind der **Name** und optional **allgemeine Charakteristika** der Klasse vermerkt (Stereotyp textuell oder als Piktogramm, Eigenschaftsangaben in geschwungenen Klammern), der zweite Abschnitt enthält die **Attribute** und der dritte Abschnitt die **Operationen**. Weitere Abschnitte mit frei wählbarer Semantik sind zulässig.

● Eine abstrakte Klasse (abstract class) ist eine Klasse, die nicht instanziert werden kann. Sie wird durch die Eigenschaftsangabe **{abstract}** charakterisiert. Alternativ zur Angabe von {abstract} kann eine abstrakte Klasse auch durch Notierung des Klassennamens in Kursivschrift gekennzeichnet werden.

☐ `Werk` ist im Gegensatz zu `CDROM`, `Buch` und `Zeitschriftenausgabe` nicht instanzierbar und daher als {abstract} markiert.

● Ein **Attribut** ist ein Datenelement, über das jedes Objekt der entsprechenden Klasse verfügt. Kann es für jedes Objekt einen individuellen Wert aufweisen, spricht man genauer von einem **Instanzattribut**, teilen sich alle Objekte einer Klasse denselben Wert des Attributs, handelt es sich um ein **Klassenattribut**. Ein Attribut wird durch seinen **Namen** definiert. Zusätzliche optionale Angaben umfassen den **Datentyp**, die **Multiplizität** (im Standardfall 1), den **Initialwert**, und die **Sichtbarkeit** (**public** oder **+** für öffentliche Sichtbarkeit, **private** oder **–** für private Sichtbarkeit, **protected** oder **#** für geschützte Sichtbarkeit). **Abgeleitete** (berechenbare) **Attribute** werden durch einen vorangestellten Schrägstrich markiert. Am Ende der Spezifikation kann dem Attribut eine Liste mit **Eigenschaftsangaben** hinzugefügt werden. **Klassenattribute** werden in UML unterstrichen.

☐ Die Klasse `Entlehnung` verfügt u.a. über das Attribut `dauer` vom Datentyp `Integer` mit dem Initialwert 30.

☐ Eine vollständigere Attributspezifikation aus einer späteren Phase des Softwareentwicklungsprozesses, um `barcode` aus der Klasse `Wert` als geschütztes, optionales (Multiplizität 0 oder 1) `Integer`-Attribut zu definieren, für das ein einmal zugewiesener Wert nicht mehr verändert werden darf (Eigenschaft {frozen}), würde lauten:

```
# barcode[0..1]: Integer {frozen}
```

● Eine **Operation** stellt eine Dienstleistung dar, die von einem anderen Objekt durch eine Nachricht angefordert werden kann. Operationen werden durch Methoden implementiert. Eine Operation aus dem dritten Abschnitt des Klassensymbols wird ähnlich einem Attribut durch **Sichtbarkeit**, **Name**, und **Eigenschaftsangaben** spezifiziert, wobei zusätzlich eine (eventuell leere) **Parameterliste** angegeben werden muss. Je Parameter sind folgende Angaben möglich: **Name**, **Datentyp**, **Datenflussrichtung** (**in**, **out**, **inout**) und **Standardwert**. Nach der Parameterliste kann der **Ergebnistyp** der Operation angegeben werden. **Klassenoperationen**, d.s. solche, die nicht auf Instanzattributen operieren, werden von **Instanzoperationen** durch Unterstreichen unterschieden.

☐ Die Klassenoperation `identifizieren (k: String): Werk` der Klasse `Werk` sucht aus allen Instanzen der Klasse `Werk` jene mit dem Schlüssel `k` und gibt diese zurück.

● Eine **abstrakte Operation** ist eine Operation einer abstrakten Klasse, deren Implementation an eine Unterklasse (s.u.) delegiert wird. Eine abstrakte Operation wird durch die Eigenschaftsangabe **{abstract}** definiert. Alternativ dazu kann die Signatur der Operation auch kursiv gesetzt werden.

● Ein **Interface** ähnelt einer Klasse, kann aber lediglich Operationen, Generalisierungsbeziehungen und allenfalls hinführende unidirektionale Assoziationen aufweisen. Es spezifiziert durch die Vereinbarung von Operationen gewünschtes Verhalten, das es aber selbst nicht implementieren kann. Dazu bedarf es »echter« Klassen, die mit dem Interface in einer so genannten Realisierungsbeziehung (eine Art Abhängigkeit, s.u.) stehen. Ein Interface wird durch ein Klassensymbol notiert, das i.a. lediglich den Namensabschnitt (mit dem Schlüsselwort **«interface»** versehen) und Operationssignaturen aufweist.

☐ Die Klasse `Entlehnung` benützt das Interface `KannDrucken`, das vom Akteur `Drucker` angeboten wird. Dadurch wird die Klasse von der konkreten Wahl des Akteurs entkoppelt.

Notizen bieten generell die Möglichkeit, UML-Diagramme verbal zu erläutern; ihre Verwendung ist nicht auf Klassen- bzw. Objektdiagramme beschränkt.

● Eine **Notiz** (*note*) wird durch Rechteck mit Eselsohr dargestellt und durch eine gestrichelte Linie mit dem beschriebenen Modellelement verbunden. Als Inhalte von Notizen sind z.B. Kommentare, Einschränkungen, oder Programmcode denkbar.

☐ Im obigen Beispiel-Klassendiagramm finden sich zwei Notizen, die jeweils eine Einschränkung enthalten. Diese Einschränkungen sind in OCL (*Object Constraint Language*) formuliert, die es als Teil des UML-Standards erlaubt, semantische Zusatzbedingungen zu definieren.

● Eine **Assoziation** beschreibt gleichartige Beziehungen zwischen Objekten als Beziehung zwischen den zugehörigen Klassen. **Binäre** (zweistellige) **Assoziationen** werden durch einfache Verbindungslinien (Kanten) zwischen den beteiligten Klassen dargestellt. Folgende zusätzliche Angaben sind u.a. möglich: **Name** der Assoziation (eventuell mit Angabe der **Leserichtung** durch ein schwarzes gleichseitiges Dreieck), Bezeichnung der **Rollen**, welche die beteiligten Objekte im Rahmen der Beziehung spielen, und **Multiplizitäten** der einzelnen Rollen (erlaubte Anzahl von »Partnerobjekten« der beteiligten Klasse). Assoziationskanten können auch gerichtet sein, um explizit anzugeben, in welche Richtung die Navigation von einem Objekt zu seinem Partnerobjekt erfolgen kann. Ungerichtete Kanten signalisieren »keine Angabe über Navigationsmöglichkeiten«.

☐ Die Assoziation zwischen den Klassen `Schlagwort` und `Werk` heißt `klassifiziert`. Sie wird von `Schlagwort` nach `Werk` gelesen (»Ein Schlagwort klassifiziert ein Werk«). Die Multiplizität ihrer beiden (unbenannten) Rollen ist jeweils »*«, was dem Intervall $[0..\infty)$ entspricht.

☐ Zwischen `Werk` und `Person` bestehen zwei unbenannte Assoziationen, die einem `Werk` eine Reihe von Personen als Autoren bzw. als Herausgeber zuordnen (vgl. die beiden Assoziationsrollen). Gemäß einer der Klasse `Werk` zugeordneten Einschränkung werden im gegenständlichen System **entweder** Autoren **oder** Herausgeber zugeordnet, jedoch nicht beides gleichzeitig.

☐ Die Assoziation `verlegt` zwischen `Verlag` und `Werk` kann im Bibliothekssystem nur in **einer** Richtung verfolgt (navigiert) werden: Der Zugriff von einem Werk zu seinem Verlag wird unterstützt, der Zugriff auf alle Werke eines bestimmten Verlags hingegen nicht.

● Bei einer **qualifizierten Assoziation** wird die Menge der Objektbeziehungen durch **qualifizierende Attribute** in disjunkte Teilmengen zerlegt.

☐ Eine CD-ROM kann nicht nur als eigenständiges Werk, sondern auch als Beilage zu einem Buch verwaltet werden. Ein solches Buch kann mehrere CD-ROM-Beilagen aufweisen, durch das qualifizierende Attribut `lfdNr` wird die Multiplizität bei `CDROM` jedoch auf maximal eins reduziert: Einer Buchinstanz und einem bestimmten Wert für `lfdNr` ist genau eine CD-ROM zugeordnet (die Null ist nötig, da viele Bücher gar keine CD-ROM besitzen).

● Eine **Assoziationsklasse** ist einer Assoziation zugeordnet. Auf Ausprägungsebene entsprechen die Objekte einer Assoziationsklasse einzelnen Objektbeziehungen. Diese Beziehungen werden durch die Attribute der Assoziationsklasse näher beschrieben. Eine Assoziationsklasse wird durch ein eigenes Klassensymbol notiert, das durch eine gestrichelte Linie mit der entsprechenden Assoziationskante verbunden ist.

☐ Die Beziehung zwischen `Werk` und `Entlehner` wird als Assoziationsklasse `Entlehnung` modelliert, deren Attribute einen konkreten Entlehnfall charakterisieren.

● Eine **mehrstellige Assoziation** beschreibt eine Beziehung zwischen mehr als zwei Partnern. Sie wird durch eine Raute dargestellt, die mit allen Klassen, die an der Assoziation teilnehmen, durch eine Kante verbunden ist. Der Assoziationsname wird in der Umgebung der Raute notiert.

● Die **Aggregation** (*aggregation*, *part-of relationship*) ist ein Spezialfall der allgemeinen Assoziation. Sie wird auch »Teile-Ganzes-Beziehung« genannt und stellt eine asymmetrische Beziehung zwischen nicht gleichwertigen Partnern dar. Bei Vorliegen einer Aggregationen wird jenes Ende der Assoziationskante, das zur Aggregatklasse, also »zum Ganzen« hinführt, durch eine kleine (nicht ausgefüllte) Raute markiert.

☐ Ein `Buch`-Objekt kann (u.a.) aus mehreren `CDROM`-Beilagen bestehen, die durch das qualifizierende Attribut `lfdNr` durchnummeriert werden.

● Die **Komposition** stellt eine spezielle, strengere Form der Aggregation dar, die durch eine ausgefüllte Aggregationsraute notiert wird. In der Kompositionsbeziehung gelten folgende Einschränkungen für die beteiligten Instanzen:

- Ein Teil darf »Kompositionsteil« höchstens *eines* Ganzen sein.

- Ein Teil existiert höchstens so lange wie sein Ganzes.
- Viele Operationen des Ganzen werden an seine Teile propagiert, typischerweise Kopier- und Löschoperationen.

□ Eine Zeitschrift »besteht aus« ihren einzelnen Ausgaben, letztere können ohne die Zeitschrift nicht existieren.

● Die **Generalisierung** (*generalization*, *is-a*-Beziehung) stellt eine taxonomische Beziehung zwischen einer spezialisierten Klasse (**Unter**- oder **Subklasse**) und einer allgemeineren Klasse (**Ober**-, **Basis**- oder **Superklasse**) dar, wobei die Subklasse die Charakteristika (Attribute, Operationen, Assoziationen) der Superklasse erbt (**Eigenschaftsvererbung**), weitere Merkmale hinzufügen kann, sowie geerbte Operationen anpassen kann. Die Generalisierung wird durch einen Pfeil mit einem gleichseitigen Dreieck als Spitze notiert, der von der Subklasse zur Superklasse führt. Mehrere Generalisierungspfeile, die zur selben Superklasse führen, können zu einem »Mehrfachpfeil« verschmelzen. Diese Darstellungsvariante ist semantisch äquivalent zur entsprechenden Menge von Einzelpfeilen.

□ Die Klassen `CDROM`, `Buch` und `ZeitschriftenAusgabe` werden zur abstrakten Klasse `Werk` generalisiert. Sie verfügen durch Vererbung über die Attribute `titel`, `jahr` und `barcode` sowie über die Operationen `klassifizieren`, `swEntfernen` (zum Hinzufügen und Entfernen von Schlagwörtern) und `identifizieren`.

Die Generalisierungsbeziehung kann durch Angabe von Einschränkungen in unmittelbarer Nähe der Pfeilspitze näher charakterisiert werden. Falls mehrere Pfeile betroffen sind, erfolgt die Zuordnung über eine gestrichelte Linie, die alle betroffenen Pfeile kreuzt. Folgenden Einschränkungen sind vordefiniert:

- **complete** (vollständig, Standardannahme) bzw. **incomplete** (unvollständig): In einer vollständigen Generalisierungshierarchie muss jedes Element der Superklasse gleichzeitig auch Element mindestens einer Subklasse sein. In einer unvollständigen Generalisierungshierarchie wird dies nicht gefordert.
- **disjoint** (disjunkt, Standardannahme) bzw. **overlapping** (überlappend): In einer disjunkten Generalisierungshierarchie kann ein Objekt nur Element höchstens einer Subklasse sein; in einer überlappenden Hierarchie kann es auch mehreren Subklassen angehören.

□ Generalisierungshierarchie von `Person`: {incomplete} ⇒ es kann Personen geben, die weder Bedienstete noch Entlehner sind (nämlich Autoren bzw. Herausgeber). {overlapping} ⇒ es kann Bedienstete geben, die auch Entlehner sind.

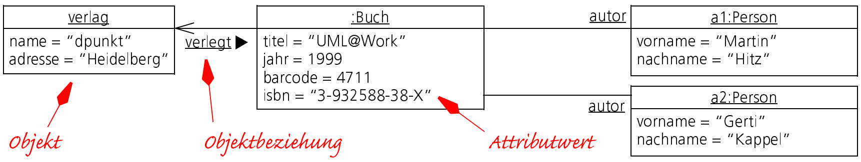
● Steht eine Unterklasse in Generalisierungsbeziehung zu mehreren Oberklassen, spricht man von **Mehrfachvererbung**.

Typ versus Instanz

Instanzen (Objekte, Objektbeziehungen (*links*) etc.) werden von den zugehörigen Typen (Klassen, Assoziationen etc.) in UML durch einen einheitlichen Mechanismus unterschieden: Es wird für die Instanz dasselbe grafische Symbol wie für den Typ benützt, wobei jedoch der Name der Instanz und eine allfällige Typangabe unterstrichen werden. Objekten können darüber hinaus Attributwerte zugeordnet werden.

● Ein Klassendiagramm, das nur Objekte und Objektbeziehungen darstellt, heißt **Objektdiagramm** (*object diagram*).

☐ Ausschnitt aus einem Objektdiagramm für ein konkretes Buch im Bibliothekssystem:



g) Sequenzdiagramm (*Sequence Diagram*)

Zur Darstellung des Systemverhaltens dienen in UML Sequenz-, Kollaborations-, Zustands- und Aktivitätsdiagramme.

- **Sequenz**- und **Kollaborationsdiagramme** beschreiben das so genannte **Interobjektverhalten**, das ist die Art und Weise, wie einzelne Objekte durch Nachrichtenaustausch interagieren, um eine bestimmte Aufgabe zu erfüllen. Sie werden daher auch als **Interaktionsdiagramme** (*interaction diagrams*) bezeichnet.

Ein Sequenzdiagramm dient primär zur Modellierung eines Szenarios, d.h., der exemplarischen Darstellung eines Ablaufs (Anwendungsfall oder Operation). Die Erweiterung zu vollständigen Algorithmen ist möglich (Symbolik für Fallunterscheidungen, Wiederholungen), kann jedoch leicht zu unleserlichen Diagrammen führen.

- Die vertikale Dimension des Sequenzdiagramms entspricht einer ordinal skalierten **Zeitachse**, horizontal werden die an dem Szenario teilnehmenden Objekte angeordnet. Sie werden durch eine gestrichelte »**Lebenslinie**« (*lifeline*) repräsentiert, die mit einem Objektsymbol beginnt.

- **Aktivierungsbalken** (*activation bars*) markieren jene Bereiche, in denen ein Objekt über den Kontrollfluss verfügt, also eine seiner Operationen ausgeführt wird.

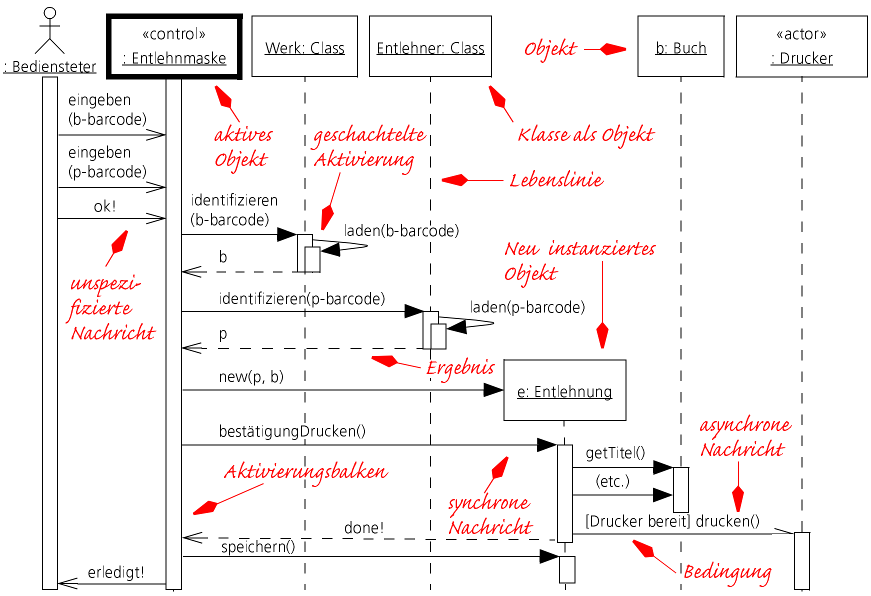
- **Aktive Objekte** besitzen einen durchgängigen Aktivierungsbalken und ein Objektsymbol mit fettem Rand.

- **Nachrichten** (*messages*) die zwischen den Objekten ausgetauscht werden, werden durch Pfeile zwischen den Lebenslinien notiert. Diese Pfeile sind mit dem Namen und den Argumenten der Nachricht beschriftet. Durch entsprechende Pfeilspitzen wird zwischen **synchroner** (Prozeduraufruf), **asynchroner** (Signal) und **unspezifizierter** (nicht näher charakterisierter) **Nachrichtenübermittlung** unterschieden. Über einen gestrichelten Pfeil kann die Rückgabe eines Ergebnisses sichtbar gemacht werden. Nachrichten, die Objekte erzeugen, werden direkt an das Objektsymbol geführt. Wird ein Objekt gelöscht, endet seine Lebenslinie in einem X.

- Eine **Überwachungsbedingung** (*guard condition*, kurz: Bedingung) legt fest, wann eine Nachricht gesendet wird. Die »überwachte« Nachricht wird nur übermittelt, wenn die in eckigen Klammern angegebene Bedingung erfüllt ist. Sollen **Wiederholungen** notiert werden, so sind die zu wiederholenden Nachrichten durch ein Rechteck einzurahmen, über dessen unterem Rand sich Angaben über die Wiederholung befinden.

☐ Sequenzdiagramm für den Anwendungsfall `Buch entlehnen`
Der Akteur kommuniziert mit dem Bibliothekssystem über ein aktives Kontrollobjekt, das die Eingabemaske für die Entlehnung darstellt und den Anwendungsfall steuert. Die Eingaben des Akteurs (Strichcode des Buchs bzw. der entlehnenden Person, Schaltfläche »Ok«), werden als (unspezifizierte) Nachrichten an die Entlehnmaske modelliert. Diese ermittelt zunächst das entsprechende Buch- bzw. Entlehnerobjekt `b` und `p`, indem sie die (synchrone) Operation `identifizieren`

aktiviert, die von den Klassen `Werk` und `Entlehner` als Klassenoperation angeboten wird (man beachte, dass die beiden Klassen als Instanzen der vordefinierten Metaklasse `Class` dargestellt sind und als solche mit Klassenoperationen auf Nachrichten reagieren). Diese Operation lädt das jeweilige durch seinen Strichcode identifizierte Objekt mit Hilfe der eigenen Operation `laden` vom Massenspeicher und gibt die resultierenden Objekte als Funktionswerte zurück. Unter der diesem Szenario zugrundeliegenden Annahme, dass die Identifikation erfolgreich war, legt das Kontrollobjekt eine neue `Entlehnung`-Instanz `e` an. Dieser werden im Zuge ihrer Initialisierung die als Argumente der Operation `new` übermittelten Referenzen auf das Buch und den Entlehner zugeordnet. Danach wird `e` veranlasst, eine entsprechende Bestätigung zu drucken. Zu diesem Zweck eruiert `e` mittels geeigneter Zugriffsoperationen (`getTitel` etc.) die zu druckenden Daten des Buchs. Im Fall, dass der Drucker als »bereit« erkannt wird (eine Alternative dazu ist im vorliegenden Szenario nicht vorgesehen), wird der eigentliche Ausdruck durch die Operation `drucken` veranlasst (asynchron, es wird nicht auf ein Ergebnis gewartet). Schließlich wird das Entlehnungsobjekt `e` gespeichert und der Akteur vom erfolgreichen Abschluss des Anwendungsfalls informiert.



h) Kollaborationsdiagramm (*Collaboration Diagram*)

Als zweiter Vertreter der so genannten Interaktionsdiagramme zeigt auch das Kollaborationsdiagramm die für einen bestimmten Zweck notwendigen Interaktionen zwischen Objekten. Das Kollaborationsdiagramm ist dem Sequenzdiagramm dabei insofern überlegen, als es zusätzlich zu den Interaktionen auch den ihnen zugrundeliegenden **Kontext** in Form von Objektbeziehungen darstellt (Kollaboration, s.u.). Umgekehrt wird darauf verzichtet, die Zeit als eigene grafische Dimension zu modellieren, sodass die Reihenfolge von Nachrichten durch Nummerieren spezifiziert werden muss.

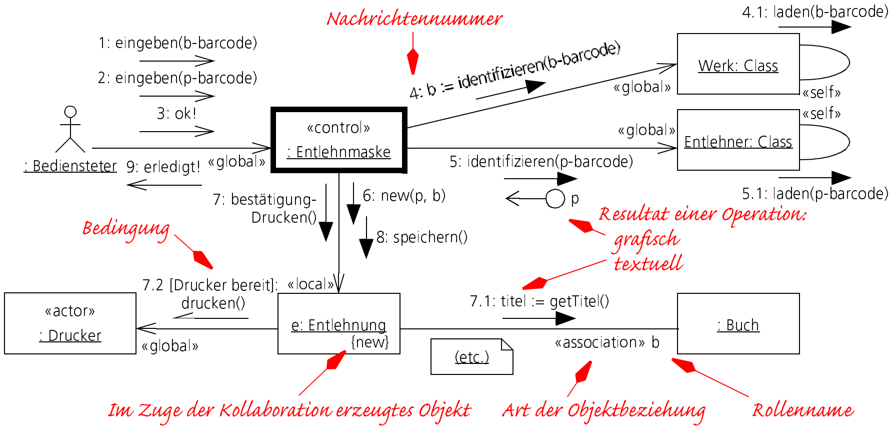
- Eine **Kollaboration** (*collaboration* – manche deutschsprachigen Autoren übersetzen den Begriff zur Vermeidung der historischen Konnotation mit »Kooperation«) wird in UML als Ausschnitt aus der statischen Modellstruktur definiert, der genau jene Modellelemente enthält, die zur Erreichung eines definierten Ziels kooperieren. Dabei werden die Modellelemente auf Instanzebene (als Objekte und Objektbeziehungen) dargestellt. Zusätzlich ist es möglich, in einer Kollaboration auch temporäre Beziehungen anzugeben:

- **«parameter»**: Das Zielobjekt ist Parameter einer Methode.
- **«local»**: Das Zielobjekt ist lokal zu einer Methode.
- **«global»**: Das Zielobjekt ist global verfügbar.
- **«self»**: Das Zielobjekt ist das Objekt selbst.
- **«association»** (Standardangabe, kann entfallen): Die Objektbeziehung entspricht einer gewöhnlichen Assoziation aus dem Klassendiagramm.

● **Gerichtete** Objektbeziehungen dokumentieren eingeschränkte Navigierbarkeit.

● Ein **Kollaborationsdiagramm** entsteht aus einer Kollaboration, indem auf den Beziehungskanten der Nachrichtenaustausch zwischen den beteiligten Objekten dargestellt wird.

☐ Kollaborationsdiagramm für den Anwendungsfall `Buch entlehnen` (vgl. das zuvor gezeigte Sequenzdiagramm mit analoger Semantik)



● Eine **Nachricht** wird durch einen Pfeil und eine textuelle Spezifikation notiert. Die Pfeilspitze gibt die Art des Nachrichtenaustausches an, wobei dieselben Typen wie im Sequenzdiagramm zulässig sind: **synchron**, **asynchron** und **unspezifiziert**. Die **Nachrichtenspezifikation** beginnt im Normalfall mit einer eindeutigen **Sequenznummer** für jede Nachricht. Diese Nummer folgt dem Format einer hierarchischen Klassifikation. Jede Klassifikationsebene entspricht einer Kontrollfluss-Schachtelung, innerhalb derer sequenzielle Nachrichten gemäß ihrer Abfolge mit den Zahlen 1, 2, usw. durchnummeriert werden. Es gilt also für zwei Sequenznummern N.n und N.(n+1) mit gleichem Präfix N, dass die Nachricht N.(n+1) unmittelbar nach N.n erfolgt und dass beide Nachrichten zur Behandlung der Nachricht N dienen. Wird für n anstelle einer Zahl ein Name angegeben, so wird **Nebenläufigkeit** spezifiziert. Liegen ausschließlich nebenläufige Nachrichten vor, werden die Sequenznummern einem »flachen« Nummernkreis entnommen. Bei bedingten Nachrichten wird die **Bedingung** in eckigen Klammern angegeben. Eine **Iteration** wird durch einen Stern nach der Sequenznummer notiert, wobei zusätzlich die Iterationsbedingung angegeben werden kann. Ein optionaler Doppelstrich rechts des Sterns deutet an, dass die einzelnen Schleifendurchgänge **nebenläufig** erfolgen können.

☐ Als 3. Schritt zur Behandlung der Nachricht 2 werden in einer Schleife n Nachrichten übermittelt: `2.3*||[i=1..n]: verarbeite(i)`

Der Sequenznummer kann bei nebenläufigen Nachrichten eine **Synchronisationsbedingung** vorangestellt sein. Diese Bedingung besteht aus einer Liste von Sequenznummern, die Vorgängernachrichten bezeichnen, gefolgt von einem Schrägstrich. Die gegenständliche Nachricht wird erst gesendet, wenn alle Vorgängernachrichten gesendet worden sind.

☐ Die Nachricht `C` (`starteC`) kann nur nach `A` und `B` gesendet werden: `A,B/C: starteC()`

Numerische Sequenznummern **synchronisieren implizit** mit einer einzigen Vorgängernachricht: Eine Nachricht N.(n+1) kann erst erfolgen, nachdem N.n erfolgt ist.

Nachrichten können über **Argumente** verfügen, die in Pseudocode oder einer Programmiersprache notiert werden. **Ergebnisse** einer Nachricht werden durch eine Liste von Namen bezeichnet, die links vom Zuweisungssymbol := stehen. Diese Namen können als Rollennamen für Objektbeziehungen herangezogen werden (vgl. `4: b := identifizieren(b-barcode)` und die Objektbeziehung mit der Rolle `b` für Nachricht 7.1 im obigen Beispiel), als Argumente für weitere Nachrichten dienen, in Bedingungen auftreten etc. Alternativ zur textuellen Notation von Ergebnissen und Argumenten sind auch eigene **Objektfluss-Pfeile** möglich, die mit den Namen der Ergebnisse bzw. mit den Argumentausdrücken markiert sind und entgegen bzw. entlang der Senderichtung der Nachricht gerichtet sind (vgl. das Ergebnis `p` der Operation 5).

Objekte und Objektbeziehungen, die während einer durch ein Kollaborationsdiagramm beschriebenen Interaktion erzeugt bzw. gelöscht werden, können durch die Einschränkung **{new}** bzw. **{destroyed}** gekennzeichnet werden (vgl. die Entlehnung `e`). Für die Kombination {new destroyed} ist die Abkürzung **{transient}** vorgesehen.

● **Aktive Objekte** werden durch die Eigenschaftsangabe {**active**} oder durch einen fetten Symbolrand markiert.

i) Zustandsdiagramm (*State Chart Diagram*)

Zustandsdiagramme beschreiben **Intraobjektverhalten**, d.h., die möglichen Folgen von Zuständen, die ein einzelnes Objekt einer bestimmten Klasse

- während seines »Lebenslaufs«, also von seiner Erzeugung bis zu seiner Destruktion, bzw.
- während der Ausführung einer Operation

der Reihe nach einnehmen kann.

Zustandsdiagramme basieren auf Verallgemeinerungen von endlichen Automaten.

● Ein **Zustandsdiagramm** stellt einen Graph dar, dessen Knoten den Zuständen entsprechen, die von Objekten eingenommen werden können, und dessen gerichtete Kanten die möglichen Zustandsübergänge angeben. Zustandsübergänge werden i.a. von exogenen Stimuli ausgelöst, die als Ereignisse bezeichnet werden.

● Ein **Zustand** (*state*) charakterisiert eine Situation, in der das betrachtete Objekt auf bestimmte äußere Ereignisse in situationsspezifischer Weise reagiert. Objekte verweilen eine gewisse Zeit lang in einem bestimmten Zustand, bis ein **Ereignis** (*event*) einen Zustandsübergang (Transition) auslöst, der dann augenblicklich erfolgt, ohne selbst Zeit in Anspruch zu nehmen. Ein Zustandssymbol im

Diagramm (Rechteck mit abgerundeten Ecken) kann in zwei Abschnitte unterteilt sein. Ein Zustand kann durch einen Namen im oberen Abschnitt charakterisiert sein (unbenannte Zustände gelten grundsätzlich als voneinander verschieden). Der untere Abschnitt kann Angaben über Aktivitäten und so genannte innere Transitionen (s.u.) enthalten.

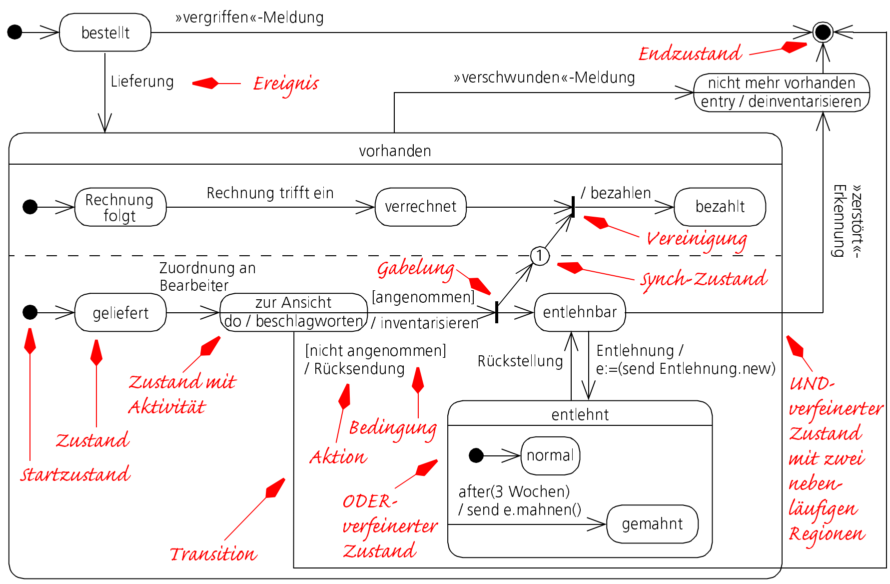
Zuständen können andauernde **Aktivitäten** (*activities*) zugeordnet werden.

- **Aktivitäten** (*activities*) werden ausgeführt, solange sich das Objekt in dem Zustand befindet, dem die Aktivität zugeordnet ist. Eine Aktivität wird im Zustandssymbol nach dem Präfix **do/** angegeben und kann durch ein eigenes Zustandsdiagramm näher beschrieben werden.

Transitionen hingegen können mit so genannten Aktionen assoziiert werden.

- **Aktionen** (*actions*) sind atomare (nicht unterbrechbare) Tätigkeiten, die im Zuge des Zustandsübergangs durchgeführt werden und (konzeptuell) ebenfalls keine Dauer aufweisen.

☐ Bestellwesen
Lebenslauf der Klasse `Werk`:



- **Transitionen** (*transitions*) sind Zustandsübergänge und werden durch einen i.a. beschrifteten Pfeil dargestellt, der einen Zustand mit seinem Folgezustand verbindet. Die Beschriftung enthält folgende Angaben:

  - Das auslösende **Ereignis**, das durch Argumente näher beschrieben sein kann. Sobald das Ereignis auftritt und die zugeordnete Bedingung erfüllt ist (s.u.), erfolgt (**schaltet**) die Transition. Eine allfällige noch andauernde, dem Zustand zugeordnete Aktivität wird dabei unterbrochen. Fehlt die Angabe des Ereignisses, so schaltet die Transition nach Beendigung der Aktivität im Zustand. Tritt ein Ereignis ein, dem im aktuellen Zustand keine Transition zugeordnet ist, wird es ignoriert – es »geht verloren«. Sind dem Ereignis mehrere Transitionen zugeordnet, so schaltet nur eine davon (nichtdeterministisch).

☐ Ereignisse im Beispieldiagramm sind etwa `Lieferung` oder `Rechnung trifft ein`.

☐ Mit dem Ende der Aktivität `beschlagworten` wird der Zustand `zur Ansicht` verlassen.

- Eine optionale **Bedingung**, die die Transition überwacht (*guard condition*). Tritt das Ereignis ein und ist die Bedingung nicht erfüllt, so erfolgt kein Zustandsübergang. Das »nicht konsumierte« Ereignis geht verloren, d.h., die Transition kann nicht ohne neuerliches Ereignis schalten.

  ☐ Der Übergang von `zur Ansicht` nach `entlehnbar` erfolgt nur unter der Bedingung `[angenommen]`.

- **Aktionen**, die im Zuge der Transition auszuführen sind. Einen besonderen Rang nimmt dabei das Senden einer Nachricht ein, wofür eine eigene Syntax vorgesehen ist. Werden mehrere Aktionen angeführt, so entspricht die Reihenfolge der Ausführung der Reihenfolge der Spezifikation.

  ☐ Aktionen sind z.B. `bezahlen`, `Rücksendung`, `e := send Entlehnung.new` (der Klasse `Entlehnung` wird die Nachricht `new` übermittelt, das Ergebnis wird `e` genannt).

● **Innere Transitionen** (*internal transitions*) werden ebenfalls von Ereignissen ausgelöst, verlassen aber den aktuellen Zustand **nicht**. Ihre Spezifikation erfolgt im unteren Abschnitt des Zustandssymbols in der Form **Ereignis / Aktion**. An die Stelle von echten Ereignissen können auch die beiden Pseudoereignisse *entry* und *exit* treten. Die ihnen zugeordneten Aktionen werden immer dann ausgeführt, wenn das Objekt in den Zustand eintritt bzw. ihn verlässt.

☐ Bei Eintritt in den Zustand `nicht mehr vorhanden` wird die Aktion `deinventarisieren` ausgeführt, egal, über welche Transition der Zustand erreicht wurde.

Eine innere Transition, die von demselben Ereignis ausgelöst wird wie eine äußere Transition, **maskiert** die äußere. Sie wird also anstelle der äußeren Transition ausgeführt, wenn das Ereignis eintritt.

UML unterscheidet folgende **Varianten von Ereignissen**:

- ***SignalEvent***: Es wird ein Signal von einem anderen Objekt empfangen. Der Ereignisname bezeichnet das Signal.

  ☐ Zu dieser Kategorie können `»verschwunden«-Meldung` oder `»zerstört«-Erkennung` gezählt werden.

- ***CallEvent***: Es wird eine Nachricht im Sinne eines Operationsaufrufs »empfangen«. Die Notation entspricht jener des Typs ***SignalEvent***, wobei der Ereignisname die auszuführende Operation bezeichnet.

  ☐ `Entlehnung`, `Rückstellung`.

- ***ChangeEvent***: Eine bestimmte Bedingung wird erfüllt. Ein solches Ereignis hat die Form ***when*(logischer Ausdruck)** und tritt ein, sobald der Wert des logischen Ausdrucks von falsch auf wahr wechselt. Ein weiteres Mal kann das Ereignis erst wieder eintreten, wenn der Wert des logischen Ausdrucks zuvor auf falsch gewechselt hat.

- ***TimeEvent***: Eine vorgegebene Zeitspanne seit einem bestimmten Vorgänger-Ereignis ist abgelaufen. Format: ***after*(konstante Zeitspanne, optionales Bezugsereignis)**. Fehlt das Bezugsereignis, bezieht sich die Zeitspanne auf den Zeitpunkt des Eintritts in den aktuellen Vorzustand.

  ☐ `after (1 Sekunde)`, was gleich bedeutend ist mit
  `after (1 Sekunde seit Eintritt in den Vorzustand)`.

  ☐ Im Beispieldiagramm: `after(3 Wochen)`

- **Pseudozustände** sind notationstechnische Hilfszustände, die sofort verlassen werden müssen:

  ● Startzustand bzw. statische Verzweigungsstelle ◯ dynamische Entscheidungsstelle
  | synchronsiationsbalken Ⓚ Synch-Zustand Ⓗ History-Zustand Ⓗ* tiefer History-Zustand

  - Der **Startzustand** (*initial state*) markiert den »Beginn« eines Zustandsdiagramms und besitzt keine eingehenden Transitionen. Die eindeutige ausgehende Transition wird sofort ausgelöst, sobald das System den Startzustand erreicht.
  - ***History*-Zustände** dienen als »Rücksprung-Adressen« in ODER-verfeinerte Zustände (s.u.).
  - Zum Aufbau komplexer Transitionen aus mehreren einzelnen »Segment-Transitionen« dienen **statische Verzweigungsstellen** (*static branch points*), **dynamische Entscheidungsstellen** *(dynamic decision points)* und **Synchronisationsbalken** (*synchronization bars*). Verzweigungs- und Entscheidungsstellen werden in sequenziellen Systemen eingesetzt, Synchronisationsbalken werden beim Übergang von sequenziellen zu nebenläufigen Systemen (Gabelung, bzw. umgekehrt: Vereinigung) verwendet.

    ☐ Die Gabelung und Vereinigung aus dem Beispieldiagramm werden durch Synchronisationsbalken ausgedrückt.
  - **Synch-Zustände** erlauben die Synchronisation von Zustandsübergängen innerhalb eines UND-verfeinerten Zustands (s.u.).
- Der **Endzustand** (*final stat*e; kein Pseudozustand) markiert das Ende des Ablaufs. Er besitzt keine ausgehenden Transitionen. Bei geschachtelten Diagrammen definiert er das Ende der Aktivität des übergeordneten Zustands, auf oberster Ebene entspricht er dem Destruktionspunkt des Objekts.

UML kennt zwei Formen der **Verfeinerung von Zustandsdiagrammen**, die ODER-Verfeinerung und die UND-Verfeinerung.

- Mit der **ODER-Verfeinerung** wird ein so genannter komplexer Zustand (Superzustand; *composite state*) in Subzustände zerlegt, wobei das Objekt immer in **genau einem** der Subzustände ist, sobald es sich im Superzustand befindet (*mutually exclusive disjoint substates*). Diese Form kommt in zwei Varianten vor:
  - Angabe eines getrennten Zustandsdiagramms, das den Superzustand verfeinert, und im Symbol des Superzustands durch Angabe von *include /* gefolgt vom Namen der Verfeinerung referenziert wird.
  - Angabe der Verfeinerung des Superzustands direkt in einem eigenen Abschnitt innerhalb des (i.a. stark vergrößerten) Zustandssymbols.

☐ Der Zustand `entlehnt` sowie beide nebenläufigen Subzustände von `vorhanden` sind ODER-verfeinert.

- Mit der **UND-Verfeinerung** erfolgt die Zerlegung in nebenläufige, **gleichzeitig** aktive Subzustände (*concurrent substates*). Auch die UND-Verfeinerung wird in einem eigenen Abschnitt des Zustandssymbols angegeben, der durch gestrichelte Linien in horizontale Unterabschnitte (**Regionen**; *concurrent regions*) zerlegt wird, die jeweils einem nebenläufigen Subzustand entsprechen und i.a. je eine ODER-Verfeinerung dieses Subzustands enthalten. Innerhalb der einzelnen nebenläufigen Subzustände erfolgen die Zustandsübergänge unabhängig voneinander.

  ☐ Der Zustand `vorhanden` ist in zwei nebenläufige Subzustände UND-verfeinert (die ihrerseits beide ODER-verfeinert sind).

  Verlassen wird ein UND-verfeinerter Zustand durch zwei alternative Mechanismen:

- Sobald in den Zustandsdiagrammen **aller** Subzustände die Endzustände erreicht worden sind, wird der komplexe Zustand über die in diesem Fall eindeutige und **unmarkierte** Transition verlassen. Es erfolgt also eine implizite Synchronisation der Abläufe in den Subzuständen – jeder Subzustand muss »bereit« für den Austritt aus dem Superzustand sein, bevor dieser Austritt erfolgen kann.
- Existiert eine Transition, die direkt vom Zustandsdiagramm eines Subzustands aus dem komplexen Zustand hinausführt, so wird der komplexe Zustand verlassen, sobald dieser Zustandsübergang erfolgt, und zwar ohne Rücksicht auf die anderen nebenläufigen Subzustände, die dadurch ebenfalls verlassen werden.

  □ Im `[nicht angenommen]`-Fall nach Verlassen des Zustands `zur Ansicht` wird der Superzustand `vorhanden` ebenfalls verlassen.

● Zwei nebenläufige Regionen in einer UND-Verfeinerung können durch einen **Synch-Zustand** (*synch state*) synchronisiert werden, womit i.a. Produzent-Konsument-Beziehungen modelliert werden: Ein Synch-Zustand »zählt«, wie oft seine eingehende Transition schaltet. Solange dieser »Zähler« positiv ist, gilt der Pseudozustand als aktiv, d.h., die ausgangsseitige komplexe Transition kann schalten. Im Zuge einer solchen Transition wird der Zähler im Synch-Zustand dekrementiert. Sobald der Zähler null wird, ist die ausgangsseitige komplexe Transition so lange blockiert, bis eingangsseitig ein Zustandsübergang erfolgt und der Zähler wieder erhöht wird.

□ Die Transition von `verrechnet` nach `bezahlt` kann nur schalten, wenn zuvor der Zustandsübergang von `zur Ansicht` nach `entlehnbar` erfolgt ist.

● **»Vererbung« von Transitionen**: Transitionen, die von einem Superzustand zu einem bestimmten Ziel **wegführen**, gelten als für alle Subzustände »dupliziert«, und zwar zum selben Ziel führend. Eine solcherart »geerbte« Transition kann aber lokal redefiniert werden, indem bei einem einzelnen Subzustand eine explizite Transition mit demselben auslösenden Ereignis angegeben wird.

  □ Das Ereignis `after(3 Wochen)` führt von jedem Subzustand von `entlehnt` in den Subzustand `gemahnt` – aus dem Subzustand `normal` heraus entspricht das der ersten Mahnung, aus `gemahnt` heraus jeder weiteren Mahnung.

  □ Das Ereignis »`verschwunden`«`-Meldung` führt von jedem Subzustand von `vorhanden` in den Zustand `nicht mehr vorhanden`.

Transitionen, die zu einem Superzustand **hinführen**, entsprechen Transitionen in den Startzustand der Verfeinerung (bzw. in alle Startzustände aller nebenläufigen Regionen bei UND-Verfeinerungen). Alternativ kann eine solche Transition direkt in einen Subzustand führen (»in die Verfeinerung hineinführen«) – ist die Verfeinerung nicht sichtbar, muss dieser Zielzustand durch einen kurzen senkrechten Strich innerhalb des Symbols des Superzustands notiert werden (**Rumpfzustand** mit **Rumpftransition**; *stubbed state*, *stubbed transition*).

  □ Die Transition, die von `Lieferung` ausgelöst wird, aktiviert die beiden nebenläufigen Startzustände des Superzustands `vorhanden`.

  □ Die dem Ereignis `Entlehnung` entsprechende Transition aktiviert den Startzustand von `entlehnt`.

● Ein **History-Zustand** (*history state indicator*) in einem ODER-verfeinerter Zustand »merkt sich« den zuletzt aktiven Subzustand der Verfeinerung, in der er sich befindet, der im Zuge einer Transition in den History-Zustand automatisch wieder angenommen wird. Durch Verwendung eines **tiefen History-Zustands** wird das »Gedächtnis« auf **alle geschachtelten** Verfeinerungsebenen des komplexen Zustands ausgedehnt.

j) Aktivitätsdiagramm (*Activity Diagram*)

Obwohl bei der objektorientierten Modellierung das Konzept des Objekts und seine Interaktionen mit anderen Objekten im Vordergrund stehen, ist es auch notwendig, **Abläufe** zu beschreiben. Abläufe treten auf gänzlich unterschiedlichen Detaillierungsebenen auf. Beispiele sind die Realisierung einer Operation, die einzelnen Schritte eines Anwendungsfalls oder das Zusammenspiel mehrerer Anwendungsfälle zur Realisierung (von Teilen) des Gesamtsystems. Aktivitätsdiagramme stammen von den Datenflussdiagrammen der strukturierten Analyse ab (s.a. Unterkapitel XXX) und ermöglichen die Beschreibung solcher Abläufe, wobei spezifiziert werden kann,

- was die einzelnen Schritte des Ablaufs tun (durch Angabe einer Bezeichnung des Schritts und der durch ihn manipulierten Objekte),
- in welcher Reihenfolge sie ausgeführt werden und
- wer für sie verantwortlich zeichnet (optional).

Die einzelnen Schritte eines Ablaufs werden als Aktionen oder Subaktivitäten bezeichnet.

● **Aktionen** (*actions*) sind nicht weiter zerlegbare Schritte eines Ablaufs.

● **Subaktivitäten** (*subactivities*) werden in einem eigenen Aktivitätsdiagramm weiter detailliert.

Im Aktivitätsdiagramm werden beide durch Angabe von Zuständen, in denen Aktionen bzw. Subaktivitäten ausgeführt werden, beschrieben.

● Ein **Zustand** (*state*) im Aktivitätsdiagramm wird durch ein Rechteck mit konvexen Vertikalen dargestellt, das den Namen der auszuführenden Aktion bzw. Subaktivität enthält. Die Verfeinerung von Subaktivitäten und damit ihre grafische Unterscheidung zu Aktionen wird durch ein stilisiertes Aktivitätsdiagramm im Zustandssymbol dargestellt.

Falls nicht inhaltlich irreführend werden – einer intuitiveren Schreibweise gehorchend – im folgenden Aktionen und Subaktivitäten zu Aktivitäten und deren Zustände zu Aktivitätszustände zusammengefasst.

Ein Grossteil der Eigenschaften von Zustandsdiagrammen ist auch auf Aktivitätsdiagramme übertragbar. So werden Beginn und Ende eines Ablaufs wie in Zustandsdiagrammen durch einen eindeutigen **Start-** und möglicherweise mehrere **Endzustände** markiert.

● **Kontrollflusskanten** (*control flow*) verbinden Aktivitäten untereinander und werden durch Transitionspfeile dargestellt, die die Ausführungsreihenfolge der einzelnen Schritte festlegen. Der Abschluss einer Aktivität fungiert dabei als impliziter Trigger für die Ausführung der Folgeaktivität. Ereignisse als explizite Auslöser von Transitionen werden nur in Ausnahmefällen modelliert (s.u.). Eine Transition kann jedoch mit einer **Überwachungsbedingung** (*guard*) in eckigen Klammern und weiteren Aktionen annotiert werden. In einem solchen Fall wird die Transition nur dann ausgeführt, wenn die vorangegangene Aktivität abgeschlossen *und* die Bedingung erfüllt ist. Ist dies der Fall, werden die angegebenen Aktionen en passant ausgeführt, bevor die Aktivität, zu der die Transition führt, angestoßen wird. Jeder Aktivität muss mindestens eine **eingehende** und mindestens eine **ausgehende Transition** zugeordnet sein. Sind mehrere ausgehende Transitionen ohne annotierte Bedingungen zugeordnet, wird indeterministisch ein Zustandsübergang ausgeführt. Werden einander wechselseitig ausschließende Bedingungen den ausgehenden Transitionen einer Aktivität annotiert, wird so ein **alternativer Ablauf** modelliert.

● Mithilfe eines Pseudozustands genannt **Entscheidungsknoten** (*decision node*) können alternative Abläufe auch explizit gemacht und komplexe Entscheidungen in Form von Entscheidungsbäumen dargestellt werden. Einem Entscheidungsknoten sind eine oder mehrere eingehende Transitionen und zwei oder mehrere ausgehende Transitionen zugeordnet. Alle ausgehenden Transitionen müssen einander wechselseitig ausschließende Bedingungen tragen, wobei auch die vordefinierte Bedingung `[else]` möglich ist. Ein Kontrollfluss, der durch Entscheidungsknoten aufgespalten wurde, kann bei Bedarf auch durch dasselbe Symbol wieder vereinigt werden. Der Entscheidungsknoten wird in dieser Rolle dann als ***merge*** bezeichnet. Einem *merge* sind zwei oder mehrere eingehende Transitionen und eine ausgehende Transition zugeordnet. Diese ausgehende Transition darf weder ein Ereignis noch eine Überwachungsbedingung tragen.

Mit den Pseudozuständen Gabelung und Vereinigung können auch **nebenläufige Abläufe**, d.h. parallele Teilfolgen von Aktivitäten, dargestellt werden.

● Die **Gabelung** (*fork*) zeigt den Beginn einer Nebenläufigkeit durch einen so genannten Synchronisationsbalken an, von dem mehrere Transitionen wegführen. Transitionen, die eine Gabelung verlassen, können auch Überwachungsbedingungen tragen. Sollte eine solche Bedingung zum Zeitpunkt des Schaltens der Transition nicht erfüllt sein, wird der zugehörige Zweig im Aktivitätsdiagramm nicht aktiviert.

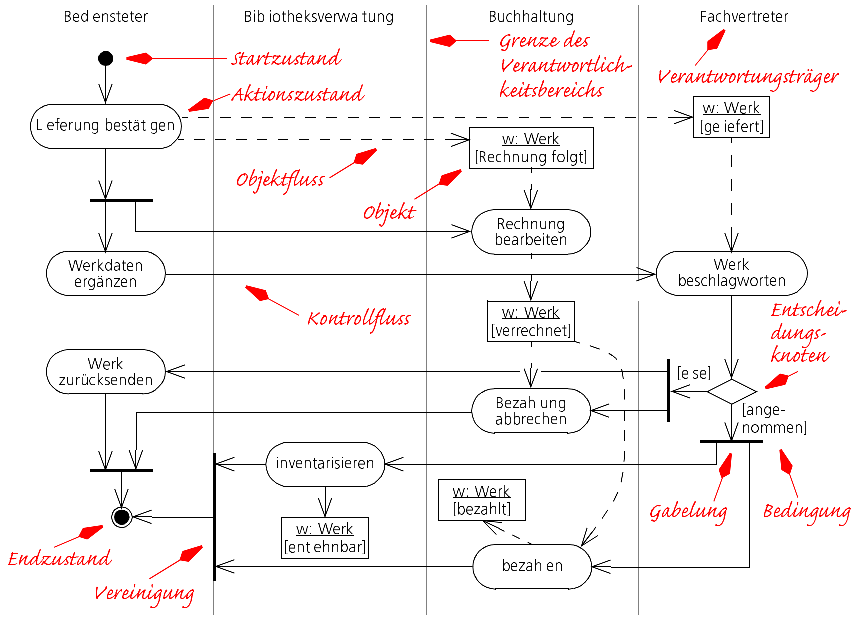
● Analog zeigt die **Vereinigung** (*join*) das Ende einer Nebenläufigkeit durch einen Synchronisationsbalken an, in den mehrere Transitionen münden. Die Vereinigung schaltet, sobald alle über die eingehenden Transitionen verbundenen Aktivitäten beendet worden sind. Ist bei der zugehörigen Gabelung aufgrund einer Überwachungsbedingung ein bestimmter Zweig nicht aktiviert worden, so ist dieser auch für die Vereinigung irrelevant, d.h., es wird »auf ihn nicht gewartet«.

● Aktivitätsdiagramme unterstützen in Anlehnung an Datenflussdiagramme die Modellierung von **Objektflüssen** (*object flows*), indem Objekte entweder als Eingabe oder als Ausgabe einer oder mehrerer Aktivitäten modelliert werden. Abhängig davon wird das entsprechende Objektsymbol, ein Rechteck mit dem unterstrichenen Namen des Objekts und seinem aktuellen Verarbeitungszustand in eckigen Klammern, als Eingabe zu oder als Ausgabe von einer Aktivität spezifiziert, indem eine gerichtete gestrichelte Kante vom Objektsymbol zur verarbeitenden Aktivität bzw. umgekehrt gezeichnet wird. Sofern der Objektfluss den Kontrollfluss zwischen zwei Aktivitäten vollständig spezifiziert, wird nur der Objektfluss dargestellt.

Für die Ausführung der Aktivitäten können verantwortliche Objekte oder Akteure (**Verantwortungsträger**) spezifiziert werden. Da Aktivitätsdiagramme meist relativ früh im Entwicklungsprozess eingesetzt werden, können diese »Objekte« aber auch sehr grob granulare Einheiten sein, z.B. die Abteilungen eines Unternehmens, dessen Geschäftsprozesse beschrieben werden.

● Verantwortungsträger und die von ihnen ausgeführten Aktivitäten werden zu **Verantwortlichkeitsbereichen** (*swimlane*) zusammengefasst, das sind Regionen im Aktivitätsdiagramm, die mit den jeweils verantwortlichen Objekten beschriftet sind und durch vertikale Linien voneinander getrennt werden.

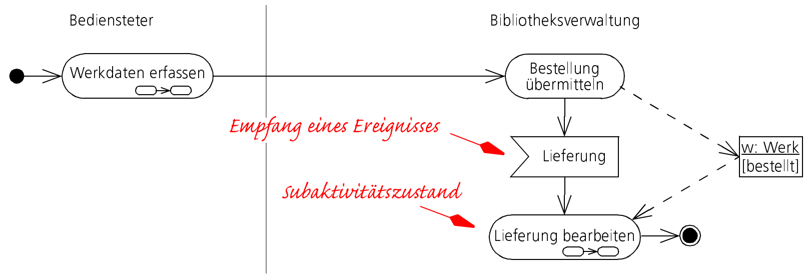
□ Aktivitätsdiagramm für den Anwendungsfall `Lieferung bearbeiten`:
Der Anwendungsfall `Lieferung bearbeiten` ist Teil der Beschaffung (siehe Anwendungsfalldiagramm weiter oben). Zur Bearbeitung der Lieferung sind vier Verantwortungsträger involviert, nämlich ein `Bediensteter`, die `Bibliotheksverwaltung`, die `Buchhaltung` und ein `Fachvertreter`. Die einzelnen Schritte, aus denen der Anwendungsfall `Lieferung bearbeiten` besteht, sind Aktionen, die nicht weiter zerlegt werden. Die Lieferung wird von einem Bediensteten bestätigt, bevor parallel die Werkdaten ergänzt werden und die Rechnung bearbeitet werden kann. Ergibt die Beschlagwortung, dass das Werk nicht benötigt wird, so wird ebenfalls parallel die Bezahlung abgebrochen und das Werk zurückgesandt. Im positiven Fall wird parallel inventarisiert und bezahlt. Als Ergebnis ist das Werk sowohl im Zustand `bezahlt` als auch im Zustand `entlehnbar`.



Aktivitätsdiagramme beschreiben in der Regel einen prozeduralen Ablauf und kein ereignisgesteuertes System. Will man dennoch das Senden bzw. Empfangen von Ereignissen oder Signalen modellieren, werden zwei eigene Knotensymbole verwendet.

- Für das **Empfangen eines Ereignisses** ist ein »Rechteck mit Nut« vorgesehen, das mit dem Namen des Ereignisses beschriftet und mit zwei unmarkierten Transitionen mit einer Vorgänger- und einer Nachfolgeraktivität verbunden ist. Optional kann ein Senderobjekt durch einen gestrichelten Pfeil auf das Ereignissymbol verweisen. Die Vorgängeraktivität wird zuerst abgeschlossen, bevor das spezifizierte Ereignis empfangen werden kann.
- Ein duales Symbol (»Rechteck mit Feder«) ist für das **Senden eines Ereignisses** im Zuge einer Transition vorgesehen.

☐ Aktivitätsdiagramm für den (komplexen) Anwendungsfall `Beschaffung`:
Die Beschaffung wurde bereits im Anwendungsfalldiagramm weiter oben beschrieben und wird nun präzisiert. Die Subaktivität `Lieferung bearbeiten` wurde im vorangegangenen Aktivitätsdiagramm detailliert. Sobald die Lieferung eingetroffen ist, kann sie bearbeitet werden. Dabei dient das zu bearbeitende Werk im Zustand `bestellt` als Eingabe.



k) Komponentendiagramm (*Component Diagram*) und Verteilungsdiagramm (*Deployment Diagram*)

Die bisher vorgestellten Diagrammarten von UML werden von den frühen Phasen der Softwareentwicklung an verwendet. UML bietet mit dem Komponentendiagramm und dem Verteilungsdiagramm aber auch diagrammatische Unterstützung zur Dokumentation des implementierten Systems. Die beiden Diagramme stellen die implementierten Softwarekomponenten, ihre Abhängigkeiten untereinander und ihre Zuordnungen zu einzelnen Knoten einer Hardwaretopologie dar.

- Ein **Komponentendiagramm** ist ein Graph, dessen Knoten Komponenten und dessen Kanten Abhängigkeiten zwischen diesen Komponenten darstellen.

- **Komponenten** (*components*) existieren in UML nur während der Implementierung bzw. Laufzeit des Systems und sind entweder

  - Quellcode-Komponenten,
  - Binärcode-Komponenten oder
  - ausführbare Komponenten.

  Eine Komponente wird durch ein Rechteck mit zwei kleinen, den linken Rand überlagernden Rechtecken dargestellt. Innerhalb des Rechtecks wird der **Komponentenname**, etwaige **Eigenschaften** und ein mögliches **Stereotyp** geschrieben.

  Komponenten können sowohl auf der Typebene als auch auf der Instanzebene existieren. Eine **Komponenteninstanz** wird durch das Komponentensymbol und – in Analogie zum Objekt – mit Namen, mit Namen und Typbezeichner durch Doppelpunkt getrennt oder unbenannt mit Doppelpunkt gefolgt vom Typbezeichner dargestellt. Die gesamte Zeichenkette wird unterstrichen, um sie von einem **Komponententyp** zu unterscheiden. Es macht auch nur Sinn, bei ausführbaren Komponenten zwischen Typ und Instanz einer Komponente zu unterscheiden. Alle anderen Komponentenarten existieren *nur* auf der Typebene.

Eine Komponente bietet eine Menge von **Schnittstellen** (*interfaces*) und deren Realisierung an. Daher kann eine Komponente jederzeit durch eine andere Komponente substituiert werden, die zumindest die Schnittstelle(n) der ursprünglichen Komponente anbietet. Dies entspricht genau der Philosophie der **komponentenorientierten Softwareentwicklung**, die die Spezifikation von angebotenen und benötigten Schnittstellen in den Vordergrund stellt und strikt zwischen Funktionalität, d.h. Schnittstellen, und deren Realisierung unterscheidet.

- Die möglichen Abhängigkeiten zwischen Komponenten sind, neben von UML vordefinierten **Übersetzungsabhängigkeiten zwischen Quellcode-Komponenten** und **Ausführungsabhängigkeiten**

zwischen ausführbaren Komponenten, benutzerdefiniert und abhängig von der verwendeten Sprache und Entwicklungsumgebung. Sie werden durch den bereits bekannten Abhängigkeitspfeil (gestrichelter Pfeil) dargestellt.

Komponentendiagramme werden immer dann eingesetzt, wenn das entwickelte System hinreichend komplex ist. Da damit physische Abhängigkeiten zwischen Komponenten beschrieben werden – im Gegensatz zu logischen Beziehungen zwischen Klassen während der Modellierung –, stellen Komponentendiagramme wichtige Informationen für Konfigurationsmanagement und Versionierung der Software zur Verfügung. **Komponentendiagramme** existieren aber **ausschließlich auf der Typebene** und können daher keine Komponenteninstanzen beinhalten. Solche können nur im Zusammenhang mit einem Verteilungsdiagramm gezeigt werden, indem angegeben wird, auf welchem konkreten Knoten (Prozessor) die Komponenteninstanz ausgeführt wird. Eine Komponenteninstanz kann auch eine geschachtelte Struktur aufweisen, indem sie selbst wieder Komponenteninstanzen, Pakete, aber auch Objekte beinhaltet, die innerhalb dieser Komponente existieren und ausgeführt werden.

☐ So zeigt das Verteilungsdiagramm für das Bibliothekssystem (s.u.) die Komponenteninstanz `beschaffung`, die auf dem Knoten `biblioServer:Host` ausgeführt wird, und selbst die Objekte `verlag` und `buch` enthält. `beschaffung` hat eine Ausführungsabhängigkeit zur am selben Knoten ablaufenden Datenbankschnittstellenkomponente `jdbc`.

● Ein **Verteilungsdiagramm** zeigt die eingesetzte Hardwaretopologie und das zugeordnete Laufzeitsystem in Form von Softwarekomponenten und eingebetteten Objekten. Ein Verteilungsdiagramm ist ein Graph, dessen Knoten Verarbeitungseinheiten, in der Regel Prozessoren, repräsentieren und dessen Kanten Kommunikationsbeziehungen zwischen diesen Verarbeitungseinheiten darstellen. Ein Knoten wird als Quader dargestellt, die Kommunikationsbeziehung zwischen den Knoten als durchgezogene Linie. Die Benennung von Knoteninstanzen erfolgt analog zu Komponenteninstanzen.

Obwohl Verteilungsdiagramme sowohl auf der Typ- als auch auf der Instanzebene definiert sind, werden sie vor allem auf der Instanzebene eingesetzt und beschreiben hier eine konkrete Hardware- *und* Softwarekonfiguration.

Auch können Knoten, wie Komponenten, geschachtelt sein und Pakete oder direkt Komponenten beinhalten. Es können natürlich nur solche Komponenten Knoten zugeordnet werden, die während der Laufzeit existieren und instanzierbar sind, d.h. ausführbare Komponenten sind. Im Verteilungsdiagramm werden neben den Kommunikationsbeziehungen zwischen den Knoten auch die Abhängigkeitsbeziehungen zwischen den Komponenten dargestellt.

☐ Ein mögliches Verteilungsdiagramm für das Bibliothekssystem besteht aus fünf Knoten und vier Kommunikationsbeziehungen zwischen diesen. Am Knoten `biblioServer:Host` läuft das Bibliothekssystem im Wesentlichen in zwei Komponenten, in der Komponente `beschaffung` und der Komponente `entlehnung`. Beide Komponenten sind über die Datenbankschnittstellenkomponente `jdbc` mit dem Datenbankserver (Knoten `dbServer`) verbunden, auf dem das Datenbankverwaltungssystem läuft (nicht explizit dargestellt). Sicherungen der Datenbank werden auf einem Band gespeichert (Knoten `bandlaufwerk` mit benutzerdefiniertem Stereotyp «`device`»), mit dem über eine SCSI-Schnittstelle kommuniziert wird (Kommunikationsbeziehung zwischen `dbServer` und `bandlaufwerk` mit benutzerdefiniertem Stereotyp «`scsi`»). Da das Bibliothekssystem als Web-Anwendung realisiert ist, wird auf die Komponenten `beschaffung` und `entlehnung` über die Web-Server-Komponente `:W3Server` mithilfe eines «`cgi`»-Aufrufs zugegriffen. Im Beispiel wird der Zugriff auf die Beschaffungssoftware (Knoten `beschaffClient:PC`) getrennt vom Zugriff auf die Entlehnsoftware (Knoten `entlehnClient:PC`) modelliert, was aufgrund sicherheitstechnischer Überlegungen sehr leicht vorstellbar ist. In beiden Fällen wird über einen Web-Browser (Komponente `beschaff:W3Browser` bzw. `entlehn:W3Browser`) auf den entsprechenden Web-Server (Komponente `:W3Server` auf dem Knoten `biblioServer:Host`) zugegriffen.