COMPANION WEBSITE
CWS
www.pearson-studium.de

informatik

Harald Störrle

UML 2 für Studenten

Mit UML-Syntax-Poster

PEARSON
Studium

PEARSON Education

PEARSON Studium

Harald Störrle

UML 2 für Studenten

PEARSON Studium

ein Imprint von Pearson Education
München • Boston • San Francisco • Harlow, England
Don Mills, Ontario • Sydney • Mexico City
Madrid • Amsterdam

Diese Teile können ihrerseits wieder rekursiv strukturiert sein (siehe z. B. für den „Boarding-Automat“ Abbildung 6.7) und könnten zunächst z. B. durch Kollaborationen beschrieben werden (siehe Abschnitt 6.4, insbesondere Abbildungen 6.26 bis 6.28).

Anschluss

Ein `Port` (dt.: Anschluss) ist ein Interaktionspunkt einer Klasse, er dient dazu, die gesamte Interaktion mit der Klasse zu kanalisieren, also die Klasse effektiv zu verkapseln: Die UML spricht hier bezeichnenderweise von einem `EncapsulatedClassifier` (in Vorläufern von UML 2.0 wurde ein ähnliches Konzept namens „Capsule“ verwendet). Dazu müssen einerseits sämtliche Interna der Klasse versteckt, im Gegenzug dafür aber *alle* für die Erfüllung einer Rolle charakteristischen Informationen vollständig offen gelegt werden.

Daher umfasst ein Anschluss neben der gebotenen auch die genutzte Schnittstelle, das Protokoll seiner Benutzung, die von ihm repräsentierte Funktionalität und so weiter – in der UML ist ein Anschluss selbst wieder eine (unstrukturierte) Klasse. Im Gegensatz zu einer simplen Schnittstelle bietet ein Anschluss also weitaus mehr Abstraktion und daher eine sehr viel stärkere Kapselung.

Anschlüsse sind Merkmale von verkapselten Klassen, d. h., für Anschlüsse gelten alle Eigenschaften und Notationen, die für alle Merkmale gelten (siehe Abschnitt 5.3). Anschlüsse sind Merkmale von verkapselten Klassen und werden als kleine Quadrate auf dem Rand derjenigen Klasse eingezeichet, zu welcher sie gehören. In Abbildung 6.5 verfügen sowohl das Gesamtsystem AAA über Anschlüsse (z. B. der Anschluss gegenüber „Passagier“) als auch seine Teile (z. B. der Anschluss gegenüber „Bodenpersonal“). Sie können mit voll qualifizierten Namen unterschieden werden (z. B. „AAA.GUI“ vs. „AAA.Abfertigung.GUI“). Da Anschlüsse Merkmale sind, besitzen sie alle Details und Notationen, die für Merkmale im Allgemeinen gelten (siehe Abschnitte 5.2 und 5.3), z. B. Multiplizitäten oder Sichtbarkeiten.

In Abbildung 6.5 tauchen mehrere Arten von Anschlüssen auf.[1]

innerer Anschluss	Der Anschluss „GUI“ von „Abfertigung“ ist direkt mit dem Kontext von AAA verbunden, nämlich mit dem Aktor „Bodenpersonal“. Daher wird „AAA.Abfertigung.GUI“ als innerer Anschluss von AAA bezeichnet. Damit dieser innere Anschluss im Kontext überhaupt sichtbar ist, müssen sowohl „AAA.Abfertigung“ als auch „Abfertigung.GUI“ sichtbar sein. Dies ist zwar zulässig, aber nicht unter allen Umständen erwünscht, denn dadurch wird im Prinzip die Verkapselung von AAA durchbrochen. Dies kann man mit Relaisanschlüssen umgehen.
Relaisanschluss	Ein *Relaisanschluss* ist ein Anschluss, der keine Aufgabe erfüllt, außer zu verhindern, dass innere Anschlüsse nach außen sichtbar gemacht werden müssen. Relaisanschlüsse werden zusammen mit *Delegierungsverbindern* benutzt. Ein Beispiel für einen Relaisanschluss ist „AAA.SQL“ in Abbildung 6.5.

1 Die Differenzierung in unterschiedliche Arten von Anschlüssen wird im UML-Standard nicht explizit vorgenommen, hat sich jedoch in den Vorgänger-Notationen von Architektur-Montagediagrammen bewährt. Sie lässt sich leicht durch Stereotypisierung in der UML verankern (siehe Anhang 11.3).

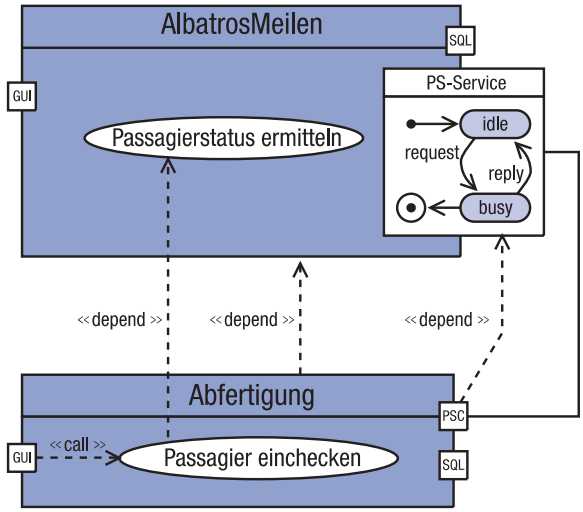
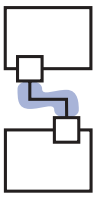


Abbildung 6.8: Schematische Darstellung des Verhaltensaspekts eines Anschlusses als Zustandsautomat in einem eigenen Abteil

Transponderanschluss Ein *Transponderanschluss* steht wie ein Relaisanschluss zwischen Kontext und Anschlüssen der Teile, führt aber eine eigene Funktion aus (z. B. die Umkodierung, Umleitung oder Pufferung von Signalen), nennt man Transponderanschluss. Ein Beispiel für einen Transponderanschluss ist „AAA.GUI" in Abbildung 6.5, der den Zugriff auf „AAA.Buchung.GUI" und „AAA.AlbatrosMeilen.GUI" zusammenfasst.

Das Verhalten von Transponderanschlüssen kann wie jeder andere Merkmal seiner Definition in einem separaten Abteil angezeigt werden (siehe die schematische Darstellung in Abbildung 6.8). Die Verwendung von Zustandsautomaten zur Modellierung von Transponderanschlüssen und Protokollrollen wird in Abschnitt 10.4 gezeigt.[2] Im vorliegenden Beispiel könnte auch die Darstellung in Form eines Dialogablaufes angemessen sein (siehe Abschnitt 10.7).

Verbinder



Um ihre Verkapselung zu gewährleisten, sollten Verbindungen zwischen Teilen ausschließlich über Anschlüsse hergestellt werden, die ihrerseits durch einen `Connector` (dt.: Verbinder) verbunden sind. Verbinder erscheinen in Architektur-Montagediagrammen als durchgezogene Linien ohne Anschriften.

Ob ein Verbinder zwischen zwei Teile eingefügt wird und wenn ja, welche Inhalte er transportieren soll, ist eine fachliche Entscheidung. Zum Beispiel könnte der Grund für den Verbinder zwischen „AlbatrosMeilen" und „Abfertigung" in Abbildung 6.5 darin liegen, dass der späteste zulässige Zeitpunkt des Check-In vom Passagier-Status abhängt: je niedriger der Status, desto früher der Check-In. Also muss zum Check-In

2 Für Anschlüsse kann nur eine Spezialform von Zustandsautomaten (Protokollzustandsautomaten) benutzt werden.

unter Umständen der Status des Passagiers abgefragt werden, wozu in diesem Beispiel nur „AlbatrosMeilen“ imstande sein möge. Daher wurden die Anschlüsse „PSS“ und „PSC“ (für „Passagierstatus-Service“ und „-Client“) eingeführt.

Neben den in Abbildung 6.5 gezeigten einfachen bzw. idealen Verbindern gibt es auch noch komplexe Verbinder, die ein eigenes Verhalten haben (z. B. Pufferung, Verzögerung, Fehler usw.), das z. B. durch einen Zustandsautomaten modelliert wird. Der UML-Standard sieht für komplexe Verbinder keine unterschiedliche Notation vor.

Ein Verbinder stellt nicht notwendigerweise ein zur Laufzeit vorhandenes Konstrukt dar (z. B. die Instanz einer Assoziation in Form eines Attributs), sondern lediglich die Möglichkeit des Nachrichtenaustauschs zwischen zwei zueinander passenden Rollen. Die „Passung“ von Rollen hängt von ihren Schnittstellen und ihrem Verhalten ab (siehe Abschnitt 6.6.4). Die exakte Semantik ist jedoch im Standard ausdrücklich nicht festgelegt.

Abhängigkeiten

UML unterscheidet zwei Arten von Beziehung, die (ungerichtete) `Association` (dt.: Assoziation) und die `DirectedRelationship` (dt.: gerichtete Beziehung). Mit Inklusion, Erweiterung und Generalisierung haben wir bereits einige Formen der gerichteten Beziehungen kennen gelernt. Gerichtete Beziehungen werden in der Regel durch einen gestrichelten offenen Pfeil dargestellt, optional wird die Art der Abhängigkeit in französischen Anführungszeichen („Guillemets“) dazugeschrieben.

Eine andere Form der gerichteten Beziehung ist die `Dependency` (dt.: Abhängigkeit). Abhängigkeit ist ein Sammelbegriff für nicht näher spezifizierte Abhängigkeiten zwischen einem *Dienstnutzer* (engl.: client) und einem *Diensterbringer* (engl.: server) und wird dargestellt als ein gestrichelter offener Pfeil vom Dienstnutzer zum Diensterbringer.

In Abbildung 6.8 sind drei Abhängigkeits-Beziehungen dokumentiert worden. Zum einen die schon am Namen ersichtliche Abhängigkeit zwischen „PS-Client“ und „PS-Service“. Infolgedessen besteht die gleiche Abhängigkeit zwischen den Teilen, die diese Rollen spielen, also zwischen „Abfertigung“ und „AlbatrosMeilen“. Fachlich ausgelöst werden beide jedoch durch die Abhängigkeit zwischen den Nutzfällen „Passagier einchecken“ und „Passagierstatus ermitteln“.

Weitere Arten von Abhängigkeiten werden in ihrem jeweiligen Kontext erläutert, z. B. in den Abschnitten 5.9, 6.5, 6.6 und 6.7.2. Darüber hinaus kann ein Modellierer auch ad hoc weitere Abhängigkeitsbeziehungen durch Stereotypisierung definieren, etwa «`call`» in Abbildung 6.8.

6.3.2 Objekt-Montagediagramm

Montagediagramme kommen auch noch als Objekt-Montagediagramme vor. Hier geht es dann nicht um die Strukturierung von Systemen in Subsysteme, sondern um den Aufbau komplexer Klasse einer feineren Granularität.

Schnittstelle

Ausgangspunkt ist die in Abbildung 6.9 dargestellte Situation. Es gibt je eine Menge von Instanzen von „Client“ und „Server“. Wenn bei einem „Client“ mindestens

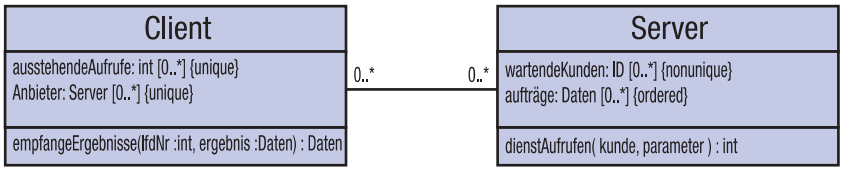


Abbildung 6.9: Eine Implementierung eines einfachen asynchronen Client/Server-Protokolls (Ausgangssituation)

ein „Server“ angemeldet ist, kann in einer ersten Phase ein „Client“ die Operation „dienstAufrufen“ eines „Servers“ aufrufen. Als Parameter wird die eigene Kennung und ein Parameter übertragen. Der „Server“ nimmt den Aufruf entgegen, puffert ihn und liefert als Rückgabewert eine laufende Auftragsnummer für den Aufruf. In einer zweiten Phase wird der Auftrag aus dem Puffer entnommen, bearbeitet und das Ergebnis an den Auftraggeber durch Aufruf der Operation „empfangeErgebnisse“ übertragen, zur Identifikation wird die Auftragsnummer benutzt.

Um Varianten von „Client“ und „Server“ herzustellen, kann man Unterklassen ableiten, die Operationen der Oberklassen nutzen und gegebenenfalls überschreiben. Allerdings könnte dies zur Folge haben, dass scheinbar harmlose Änderungen der Oberklassen verheerende Auswirkungen in den Unterklassen haben (das so genannte *fragile base class problem*). Sauberer ist es daher, das Client/Server-Protokoll durch ein Paar von Schnittstellen zu beschreiben, etwa so wie in Abbildung 6.10. Anschließend können diese Schnittstellen von verschiedensten Klassen implementiert werden, ohne dass es starke Kopplung zwischen den Implementationen gibt. Dadurch wird ein größeres Maß an Abstraktion erreicht und das *fragile base class problem* wird vermieden.

Eine *Schnittstelle* (`Interface` in UML) ist die Deklaration einer Menge von Operationen (ähnlich wie z. B. in Java). Die UML sieht vor, Schnittstllen wie eine stereotypisierte Klasse darzustellen („requireService“ und „provideService“ in Abbildung 6.10).

Zwischen Schnittstellen und Klassen sieht die UML zwei Arten von Beziehungen vor.

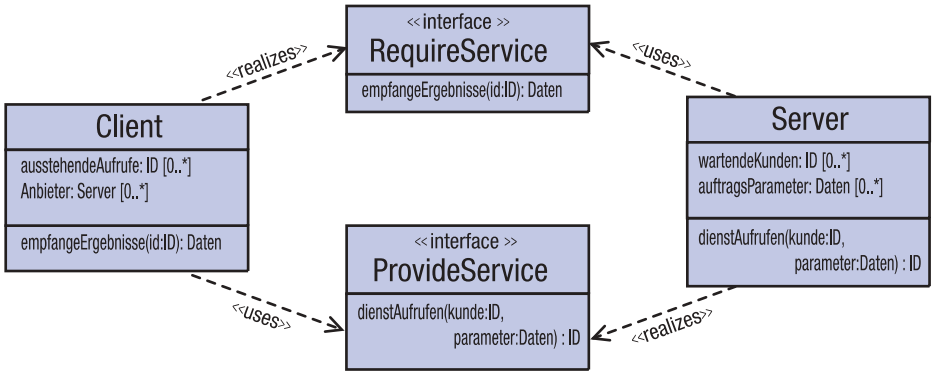


Abbildung 6.10: Verbesserte Modellierung des Client/Server-Protokolls aus Abbildung 6.9: Abtrennung der Schnittstellen „requireService“ und „provideService“

Realisierung	Die Beziehung `Realization` (dt.: Realisierung) stellt den Zusammenhang zwischen einer Klasse und einer von ihr gebotenen Schnittstelle (Kreis) dar. Realisierung wird als gestrichelter Pfeil notiert, der optional das Schlüsselwort «`realize`» trägt. Um Rückwärtskompatibilität sicherzustellen, kann Realisierung auch als gestrichelter Pfeil notiert werden, also wie eine „gestrichelte Generalisierung". Realisierung kann auch zwischen zwei Schnittstellen gelten. Realisierung und Generalisierung sind zwischen Schnittstellen bedeutungsgleich.
Nutzung	Die Beziehung `Usage` (dt.: Nutzung) stellt den Zusammenhang zwischen einer Klasse und einer von ihr genutzten (bzw. geforderten) Schnittstelle dar (Halbkreis). Nutzung wird als offener gestrichelter Pfeil notiert, der das Schlüsselwort «`use`» trägt.

Eine alternative Notation für Schnittstellen, ihre Realisierung und Nutzung ist in Abbildung 6.11 dargestellt. Diese Notation ist nicht nur wesentlich kompakter, sondern auch anschaulicher: Die Dualität von realisierten und genutzten Schnittstellen ist augenfällig (siehe Abbildung 6.12).

Der traditionelle englische Name *lollipop notation* passt nur auf realisierte Schnittstellen und nicht auf die geforderte Schnittstelle. Daher wird in UML 2 der Name *ball-and-socket notation* eingeführt, was hier mit *Kopf/Fassung-Notation* übersetzt wird.

Die Definition der Passung zweier Schnittstellen ist im Standard explizit offen gelassen. Mindestens gilt jedoch, dass die Signatur einer angebotenen Schnittstelle die Signatur einer nachgefragten Schnittstelle vollständig enthalten muss, also die Namen und Typen der jeweils deklarierten Operationen.

Das oben beschriebene *Protokoll* (engl.: protocol) besteht aber nicht nur aus zwei Mengen von Operationen, sondern auch aus einer bestimmten Weise der Benutzung, also einem bestimmten Verhalten. Dieses Protokoll kann z. B. durch Kommentare oder durch Vor- und Nachbedingungen in OCL oder Prosa beschrieben werden (siehe Abbildung 6.13).

Abbildung 6.11: Bessere Darstellung des Modells aus Abbildung 6.10

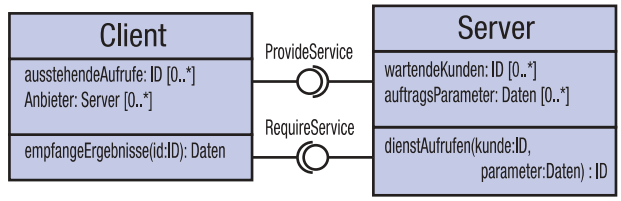


Abbildung 6.12: Bessere Darstellung des Modells aus Abbildung 6.9

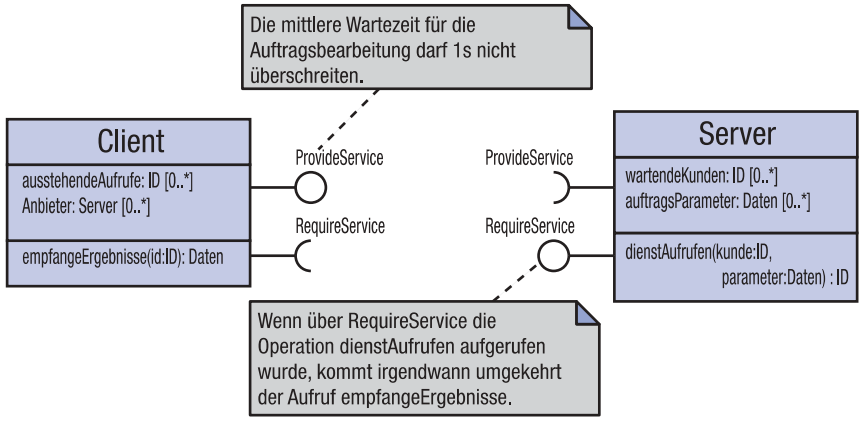


Abbildung 6.13: In der Regel wirken verschiedene Schnittstellen einer Klasse an der Erbringung einer Funktionalität zusammen.

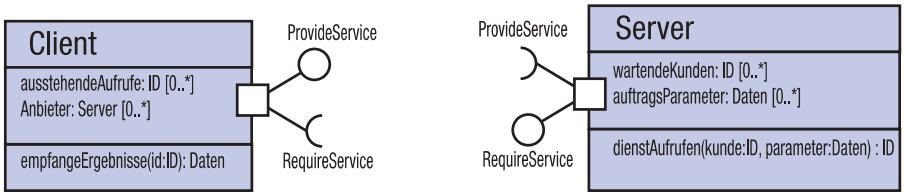


Abbildung 6.14: Weiterentwicklung des Modells aus Abbildung 6.13: Die Schnittstellen werden zu Anschlüssen zusammengefasst und verstärken so die Kapselung der Klassen „Client" und „Server".

In diesem Beispiel wirken je zwei Schnittstellen einer Klasse an der Erbringung einer Funktionalität zusammen. Um diesen übergreifenden Charakter deutlich zu machen, kann eine Menge von Schnittstellen zu einem Anschluss zusammengefasst werden (siehe Abbildung 6.14). So sind die zusammengehörigen Schnittstellen eindeutig aufeinander bezogen. Das Verhalten des Anschlusses – und damit der Schnittstellen – lässt sich durch einen Zustandsautomaten beschreiben, wie bereits in Abbildung 6.8 angedeutet. Umgekehrt zeigt diese Notation auch, wie Anschlüsse um Schnittstellen angereichert werden, auch in Architektur-Montagediagrammen.

Wenn ein Anschluss mehrere Schnittstellen nutzt bzw. fordert, kann als notationelle Vereinfachung darauf verzichtet werden, jede Schnittstelle einzeln einzuzeichnen. Stattdessen können die Namen von geforderten und gebotenen Schnittstellen jeweils als Aufzählung an ein Schnittstellensymbol geschrieben werden (siehe Abbildung 6.15).

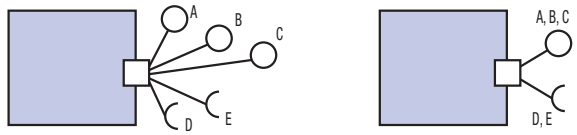


Abbildung 6.15: Zwei gleichwertige Darstellungen eines Anschlusses mit mehreren genutzten bzw. geforderten Schnittstellen

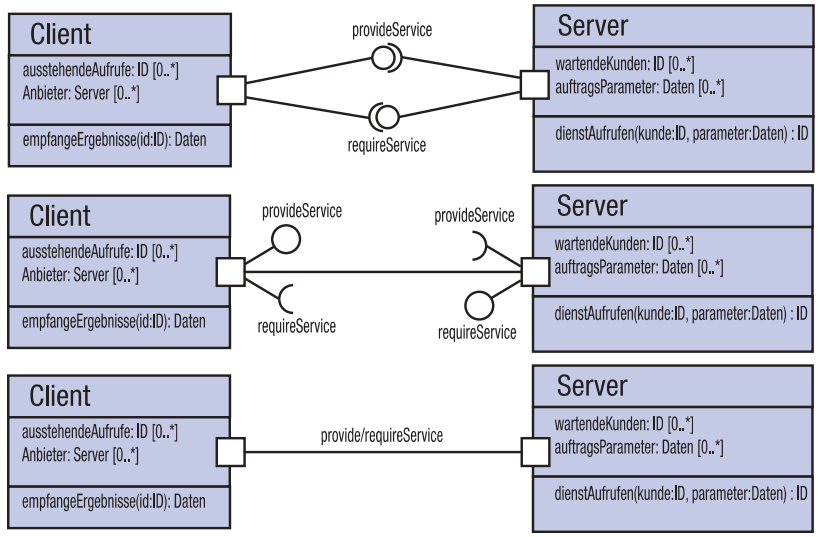


Abbildung 6.16: Ein Paar unidirektionaler Konnektoren (oben) kann der Einfachheit halber durch einen unspezifischen Konnektor ersetzt werden (Mitte) und die explizite Nennung der Schnittstellen entfällt meist (unten).

Verbinder

Wenn zwei Anschlüsse zueinander passende Schnittstellen besitzen, können sie miteinander verbunden werden (siehe Abbildung 6.16, oben). Der Übersichtlichkeit halber wird oft ein Paar unidirektionaler Verbinder durch einen einzelnen (unspezifischen) `Connector` (dt.: Verbinder) ersetzt werden (siehe Abbildung 6.16, Mitte). Verbinder werden wie Assoziationen notiert, der Name wird in der Form [([*Name*]: *Klasse*) | *Name*] angegeben. Zusätzlich können Eigenschaften in der gleichen Form wie für Assoziationen angegeben werden (siehe Abbildung 5.19). Typischerweise werden dann die Schnittstellen nicht mehr eingezeichnet, nur noch der Verbinder (siehe Abbildung 6.16, unten). Eine ausführlichere Beschreibung zu Verbindern folgt in Abschnitt 6.6.3.

6.3.3 Interpretation von Anschlüssen und Verbindern in Java

Die Interpretation von UML-Klassen als Java-Klassen bzw. Datenbank-Tabellen ist recht nahe liegend. Schwieriger wird es, wenn die UML-Konstrukte keine unmittelbare Entsprechung in der gewählten Implementierungssprache besitzen. Wenn es keine augenfällige Entsprechung gibt, bestehen in der Regel mehrere Möglichkeiten der Abbildung. Im Folgenden vergleichen wir als Beispiel zwei mögliche Interpretationen von Anschlüssen und Verbindern in Java.

Interpretation 1

Abbildung 6.17 zeigt ein sehr einfaches Montagediagramm, das nach Java übersetzt werden soll. Insgesamt kann man sagen, dass der Schritt von dem Montagediagramm zu dem Java-Programm relativ groß ist. Daher ist eine unmittelbare Umsetzung von

Abbildung 6.17: Ein einfaches Montagediagramm, das nach Java übersetzt werden soll.

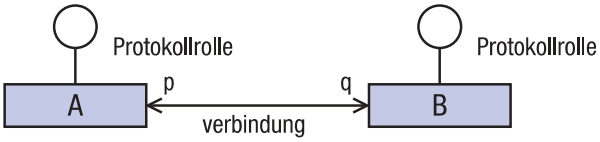


Abbildung 6.18: Zwischenschritt bei der Umsetzung des Montagediagramms von Abbildung 6.17 in das Java-Programm von Abbildung 6.19

```
interface Protokollrolle {
   public void snd(Signal s);
   public void rcv(Signal s);
}
```

Abbildung 6.19: Protokollrolle als Java-Interface

UML-Montagediagrammen in Java-Code unter Umständen schwierig und der Überblick dafür geht verloren, wie die Umsetzung eigentlich definiert ist.

Es bietet sich also an, einen Zwischenschritt einzulegen: zunächst könnte man das Montagediagramm in ein Implementations-Klassendiagramm auflösen und dann dieses Klassendiagramm nach Java übersetzen. Die erste Interpretation ist in Abbildung 6.18 dargestellt.

In Java wird zunächst eine Protokollrolle definiert (siehe Abbildung 6.19). Eine Klasse `Signal` wird dabei vorausgesetzt. Dann werden Komponenten als Klassen interpretiert, wobei verbundene Ports einfach zu Verweisen auf die durch den Port verbundene Komponente werden (siehe Abbildung 6.20).

Interpretation 2

Eine andere mögliche Interpretation zeigt Abbildung 6.21. Hier werden Anschlüsse und Verbinder als eigenständige Objekte realisiert. Abbildung 6.23 zeigt zunächst eine Möglichkeit, Komponenten und Anschlüsse zu realisieren. Nach diesem Schema kann das Montagediagramm aus Abbildung 6.17 als das Objektdiagramm aus Abbildung 6.22 interpretiert werden.

Abbildung 6.24 definiert zusätzlich eine Klasse `BinärVerbinder` und definiert darauf aufbauend eine Klasse `System`. Offenbar ist diese zweite Variante schon recht komplex. Es gibt zahlreiche Details, die anders hätten gelöst werden können, und es sind auch gänzlich andere Abbildungen denkbar, z. B. könnte man hier auch das Beobachter-Muster einsetzen, wobei der Verbinder dem Subjekt entspräche.

Ohne den Zwischenschritt über das Klassendiagramm aus Abbildung 6.21 würde zumindest für die zweite Interpretation der Übergang vom Montagediagramm zum

```
public class A implements Protokollrolle {
  B q;
  Signal buffer;

  public void snd(Signal s) {
     q.rcv(s);
  }

  public void rcv(Signal s) {
    buffer = s;
  }
  ...
}

public class B implements Protokollrolle {
  A p;
  final static int bufsize = 9;
  int pos = 0;
  Signal [] buffer = new Signal[bufsize];


  public void snd(Signal s) {
     q.rcv(s);
  }

  public void rcv(Signal s) {
    buffer[pos] = s;
    pos = pos++ % bufsize;
  }
  ...
}
```

Abbildung 6.20: Anschlüsse und Verbinder in Java (Interpretation 1)

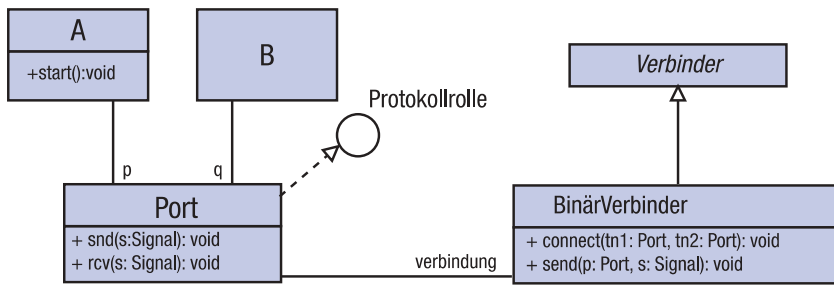


Abbildung 6.21: Zwischenschritt bei der Umsetzung des Montagediagramms von Abbildung 6.17 in das Java-Programm der Abbildungen 6.23 und 6.24

Java-Code sehr schwierig werden. Viel einfacher ist es, die Code-Generierung aus Implementations-Klassendiagrammen zu automatisieren und die Übersetzung von Montagediagrammen in Klassendiagrammen nur teilweise zu automatisieren oder ganz manuell vorzunehmen.

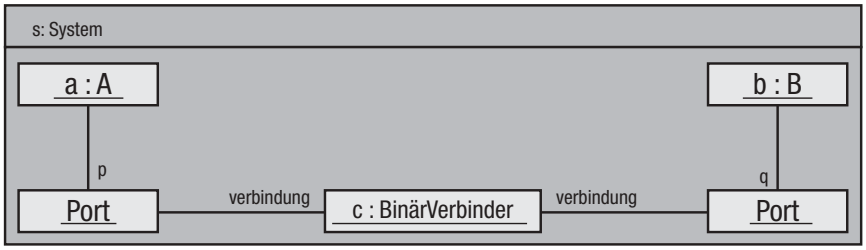


Abbildung 6.22: Instantiierung des Systems aus Abbildung 6.21.

```
public class A {
  Protokollrolle p;

  public void start() {
    new Signal("hallo!") s;
    p.snd(s);
    ...
  }
  ...
}

public class Port implements Protokollrolle {
  private Verbinder verbindung;
  public void connect (Verbinder verbinder) {
    verbindung = verbinder;
  }
  public void snd(Signal s) {
    verbindung.send(self, Signal s);
  }
  ...
}

public class B {
  Protokollrolle q;
  ...
}

class BinärVerbinder {
  private Port[1] teilnehmer;

  public void connect(Port tn1, Port tn2) {
    teilnehmer[0] = tn1;
    tn1.connect(self);
    teilnehmer[1] = tn2;
    tn2.connect(self);
  }

  public void send(Port p, Signal s) {
    if (p == teilnehmer[0])
         {teilnehmer[1].rcv(s);}
    else {teilnehmer[0].rcv(s);}
  }
  ...
}
```

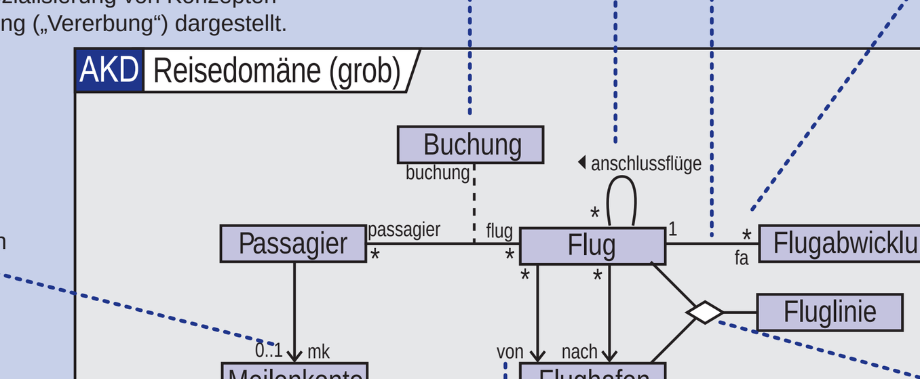
Abbildung 6.23: Komponenten, Anschlüsse und Verbinder in Java (Interpretation 2)

UML 2.0 Syntax

© Harald Störrle, www.pst.ifi.lmu.de/~stoerrle

Klassendiagramme

Analyse-Klassendiagramme stellen Geflechte von Konzepten dar, wobei Konzepte durch Klassen und ihre Beziehungen durch Assoziationen dargestellt werden. Verallgemeinerung und Spezialisierung von Konzepten werden durch Generalisierung („Vererbung") dargestellt.



benannte ungerichtete Assoziation mit Leserichtung

Assoziationsklasse

ungerichtete Assoziation

Die Multiplizität gibt an, wie viele Objekte eines Typs an einer Assoziation teilnehmen können. Die Multiplizität wird als Intervall aus minimaler und maximaler Zahl von Teilnehmern angegeben (sind beide gleich, entfällt der obere Wert). Der Wert * gibt eine unbeschränkte Multiplizität an.

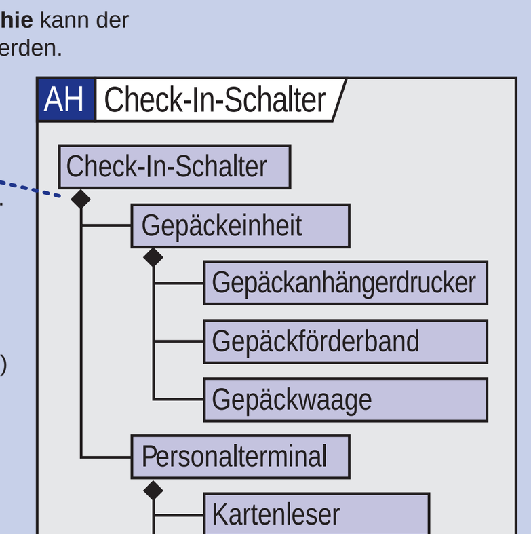
Assoziationen können durch Namen und Pfeile als **navigierbar** ausgezeichnet werden.

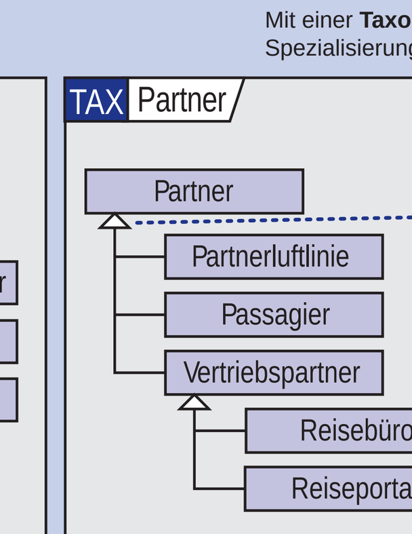
ternäre Assoziation

benanntes Assoziationsende (Rolle)

Mit einer **Aggregationshierarchie** kann der statische Aufbau verdeutlicht werden.

Mit einer **Taxonomie** wird eine Generalisierungs- bzw. Spezialisierungshierarchie von Begriffen aufgebaut.





Komposition ist eine Teil-Ganzes-Beziehung („by value").

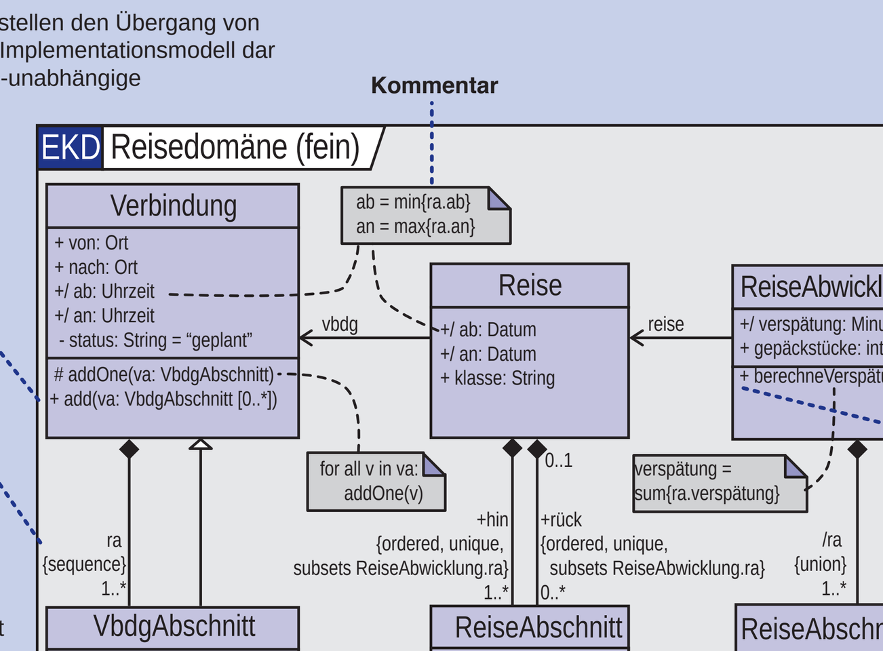
Im Gegensatz dazu ist eine **Assoziation** ein reiner Verweis („by reference").

Die **Aggregation** (weiße Raute) ist undefiniert, und sollte vermieden werden.

Die **Generalisierungsbeziehung** kann in unterschiedlichen Kontexten z.B. als Generalisierung, Spezialisierung, Vererbung, oder Subtypisierung interpretiert werden.

Entwurfs-Klassendiagramme stellen den Übergang von einem Analysemodell zu einem Implementationsmodell dar und dokumentieren Technologie-unabhängige Entwurfsentscheidungen.

Kommentar



Abgeleitete Merkmale werden durch einen Schrägstrich angezeigt.

Multiplizität eines Attributs

Merkmale können durch Eigenschaftswerte näher beschrieben werden.
- Die Eigenschaften **unique**/ **nonunique**, **ordered/unordered**, **seq**, **sequence**, und **bag** beziehen sich auf die Menge der Objekte in einem Merkmal,
- **readonly/unrestricted** bezieht sich auf Sichtbarkeit und Änderbarkeit,
- **composite** legt fest, dass der Teil nur in einem Ganzen existieren kann, nicht losgelöst;
- **redefines *Attr***, **subsets *Attr***, **union** beziehen sich auf andere Merkmale der Klasse.

Klassen besitzen **Merkmale**, die sich nach statischen Merkmalen (Attribute und Assoziationen) und dynamischen Merkmalen (Operatinen) untergliedern.

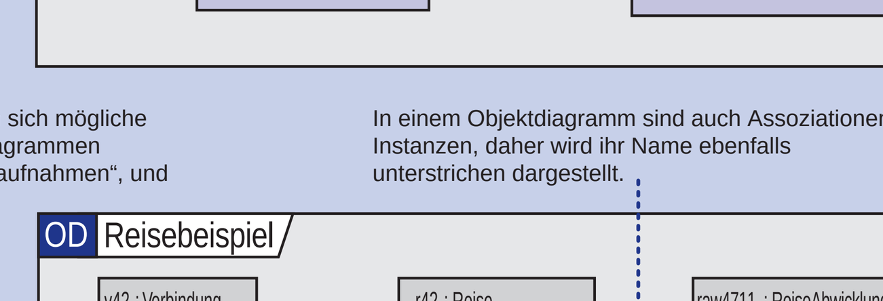
Attribut mit **Typ** und **Initialwert**

Operation („Methode")

Die **Sichtbarkeit** von Merkmalen einer Klasse kann einen der folgenden Werte annehmen:
+ (public) allgemein sichtbar
# (package) innerhalb des Pakets sichtbar
~ (friend) wird durch permission-Abhängigkeit sichtbar gemacht
- (private) nur für Objekte dieser Klasse selbst sichtbar

Klassenattribute („static" in Java und C++) können für die Klasse instantiert werden.

Klassen und Anschlüsse können durch **Schnittstellen** näher beschrieben werden. Dabei kann zwischen gebotenen und genutzten Schnittstellen unterschieden werden.



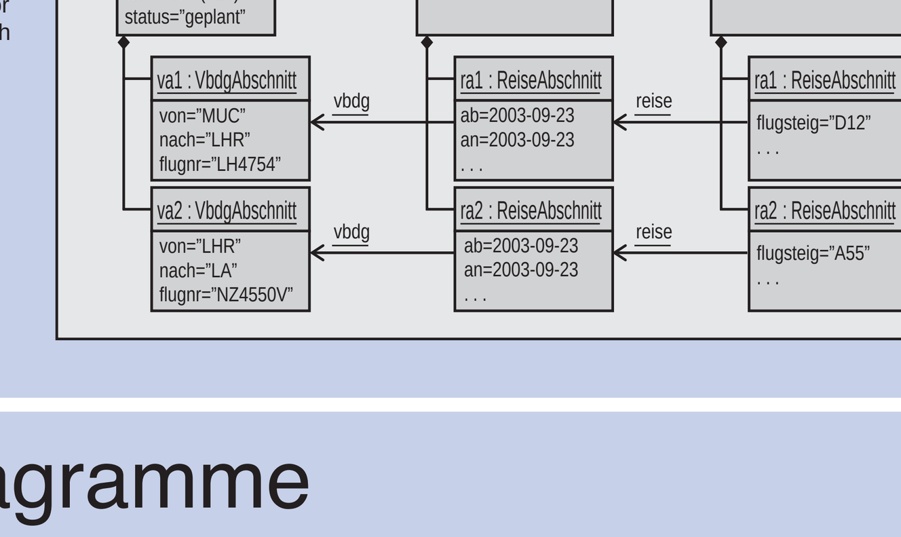
genutzte **Schnittstelle** (Benutzungs-Beziehung, «uses»)

gebotene **Schnittstelle** (Realisierungs-Beziehung, «realizes»)

Assoziationen und Attribute sind semantisch gleichwertig, daher können beide durch die gleichen Eigenschaftswerte beschrieben werden.

Mit **Objektdiagrammen** lassen sich mögliche Instantierungen von Klassendiagrammen darstellen, also quasi „Momentaufnahmen", und damit Systemzustände.

In einem Objektdiagramm sind auch Assoziationen Instanzen, daher wird ihr Name ebenfalls unterstrichen dargestellt.



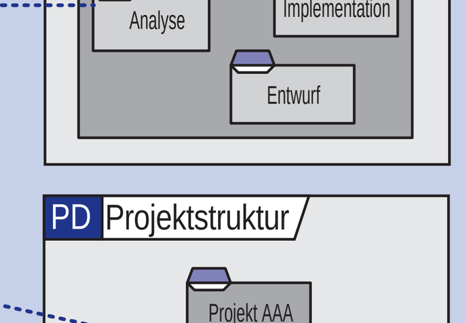
Objekte (in der UML-Terminologie „InstanceSpecification") sind Instanzen von Klassen. Sie werden durch einen (optionalen) Instanznamen vor dem Klassennamen und durch Unterstreichung angezeigt.

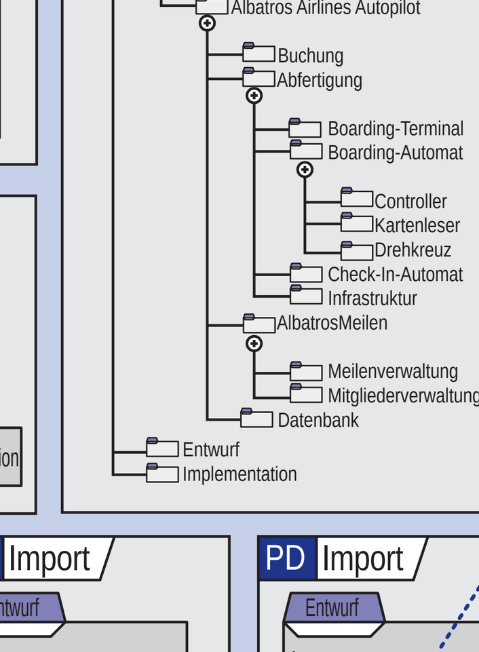
Anders als in Klassendiagrammen besitzen die Attribute in Objektdiagrammen konkrete Werte.

Paketdiagramme

Paketdiagramme stellen die Gruppierung von Modellelementen zu Namensräumen dar. Paketdiagramme werden gleichermaßen zur Beschreibung von Projektinfrastrukturen und Software verwendet.

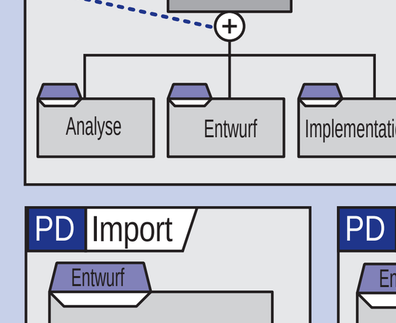
Für große Aggregationshierarchien bietet sich eine baumförmige Darstellung in Anlehnung an gängige Werkzeuge an.





Pakete werden als stilisierte Hängeregistermappen mit einer Lasche links oben dargestellt. Der Name eines Pakets wird auf der Lasche oder auf der Mappe selbst dargestellt.

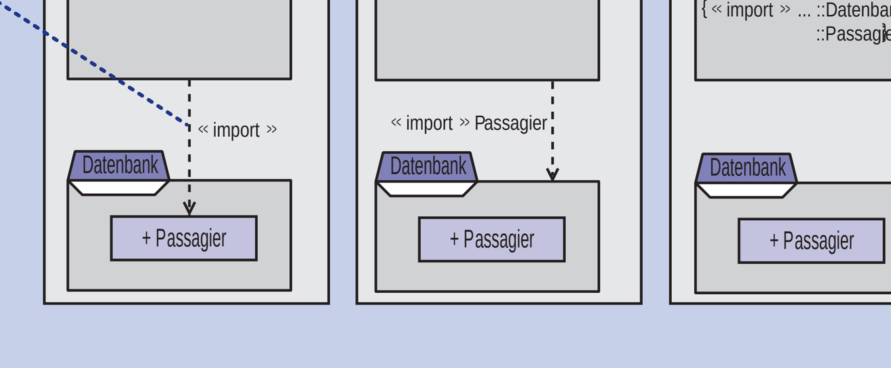
Pakete können geschachtelt sein, was entweder durch die graphische Schachtelung (links) oder die Aggregationsassoziation (rechts) angezeigt wird.



Die **Aggregationsbeziehung** für Pakete ähnelt der Komposition von Klassen

In einem Paket sind alle Elemente **sichtbar** die
- in diesem Paket **definiert** werden;
- aus einem anderen Paket **importiert** werden;
- in einem übergeordneten (!) Paket **sichtbar** sind.

Verschiedene Pakete können durch **Importbeziehungen** miteinander in Beziehung gesetzt werden.



Um gezielt einzelne Elemente (z.B. Klassen oder Pakete) sichtbar (und damit zugreifbar) zu machen, kann der **Einzelimport** benutzt werden. Im Bild rechts sind drei notationelle Varianten von Einzelimport dargestellt: das Paket Entwurf kann jeweils auf das Element Passagier zugreifen.

Soll ein Element nicht weiter exportiert werden dürfen, wird es mit «**access**» importiert.

Die Bestandteile **qualifizierter Namen** werden durch „::" voneinander getrennt.

Um den gesamten Inhalt eines Pakets auf einen Schlag sichtbar zu machen, kann der **Pauschalimport** benutzt werden. Er wird angezeigt durch eine Beziehung auf ein exportierendes Paket mit dem Stereotyp «**import**» aber ohne qualifizierenden Elementnamen (nicht dargestellt).

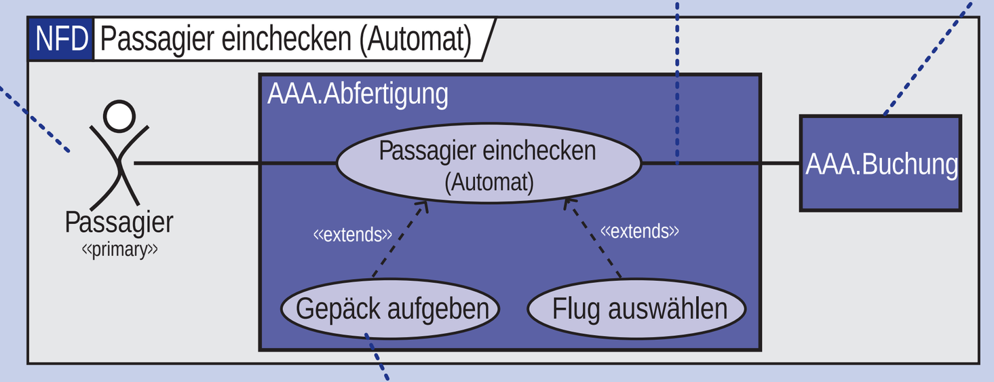
Nutzfalldiagramme

Nutzfalldiagramme sind vor allem als Übersichtsdarstellungen nützlich. Sie stellen einerseits Elemente eines Systemkontextes dar (also Akteure und Nachbarsysteme), andererseits Geschäftsprozesse bzw. Nutzfälle des Systems. Daneben können auch Abhängigkeiten zwischen Geschäftsprozessen bzw. Nutzfällen mit Inklusions- und Erweiterungsbeziehungen angezeigt werden.

Die **Assoziation** zeigt an, dass der Akteur bzw. das Nachbarsystem an diesem Geschäftsprozess teilnimmt.

Nachbarsystem

Der **Akteur** Passagier ist Teil des Systemkontextes. Das Stereotyp «primary» drückt aus, dass dieser Akteur den Geschäftsprozess auslöst.

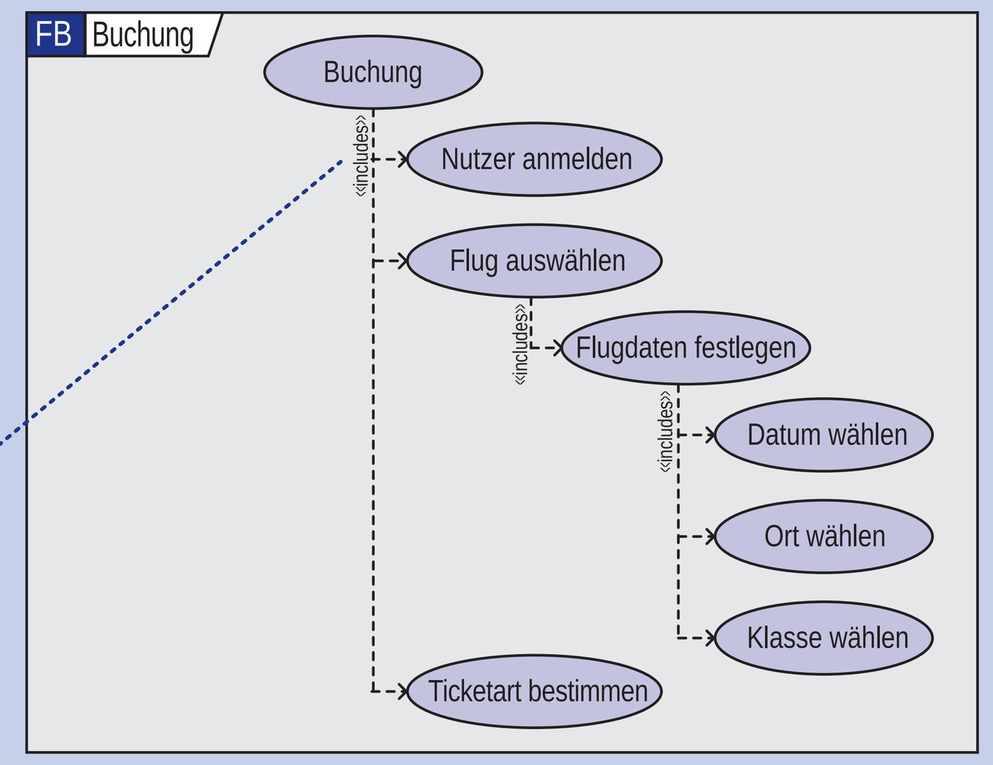


Das Layout eines Nutzfalldiagramms sollte die Leserichtung ausnutzen, und nach Möglichkeit den auslösenden Akteur links darstellen, und aufgerufene Teil- bzw. eingebundene Sonderfälle unten.

Eine optionale Variante eines Geschäftsprozesses bzw. Nutzfalles kann durch einen **Erweiterungsfall** („Sonderfall") modelliert werden. Im zu erweiternden Geschäftsprozess bzw. Nutzfall wird dazu ein **Erweiterungspunkt** definiert (hier nicht dargestellt), an dem der Sonderfall durch eine **Erweiterungs-Beziehung** angebunden wird.

Ein **Funktionsbaum** stellt die Aufrufabhängigkeiten zwischen Geschäftsprozessen bzw. Notfällen dar. Diese Darstellung ist vor allem in der Dokumentation geeignet, weniger zum Entwurf.

Analog zu Klassen können auch für Akteure und Nutzfälle Kompositionshierarchien und Taxonomien gebildet werden.



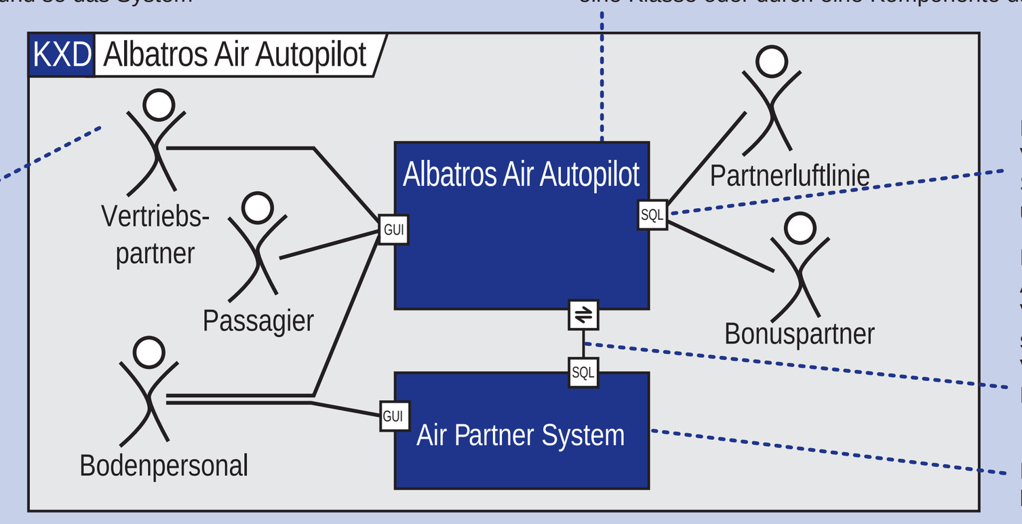
Ein Teilprozess bzw. eine Teilfunktion, die einmal oder mehrmals im Verlauf eines Geschäftsprozesses bzw. Nutzfalles aufgerufen werden kann, wird durch eine **Inklusions-Beziehung** dargestellt.

Durch das Layout des Funktionsbaums kann die **Reihenfolge** der Aufrufe (teilweise) wiedergegeben werden. Durch die Einrückung kann gleichzeitig die **Hierarchieebene** des Aufrufs angezeigt werden.

Montagediagramme

Ein **Kontextdiagramm** dient dazu, die Umgebung eines Systems zu definieren, und so das System abzugrenzen.

Ein **System** in einem Kontextdiagramm kann je nach Anwendung durch eine Klasse oder durch eine Komponente dargestellt werden.



Ein **Akteur** ist eine Rolle, die mit einem System interagiert. Diese Rolle kann von einem Menschen, einer Gruppe oder Organisation, aber auch von einer Maschine bzw. einem Nachbarsystem übernommen werden (in dem Fall empfiehlt es sich, eine stereotypisierte Notation zu verwenden).

Ein **Anschluss** stellt die Verbindung zwischen zwei Systemen bzw. einem System und einem Akteur dar.

Ein **Verbinder** verbindet Anschlüsse von Systemen. Verbinder können rein passiv sein oder mit eigenem Verhalten modelliert sein (z.B. Fehler oder Verzögerung).

Der Begriff **Nachbarsystem** bezieht sich immer auf ein „Haupt"-System, und beschreibt oft dessen Einbettung in existierende Systemlandschaften.

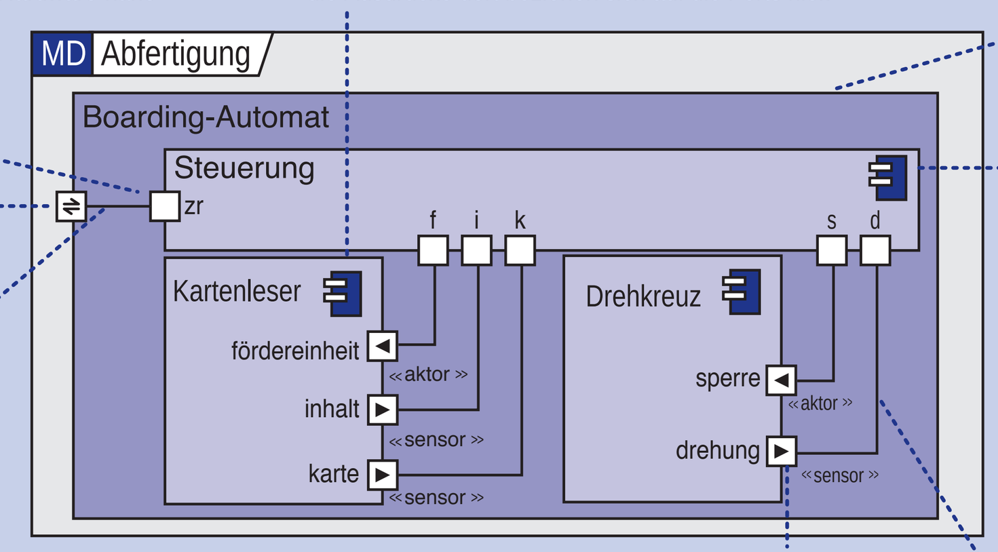
Mit einem **Montagediagramm** kann die Grobstruktur eines Systems beschrieben werden. In dieser Verwendung lassen sich auch Hardware-nahe Strukturen gut darstellen (so begründet sich z.B. auch der Name „Anschluss").

Das Verhalten der Komponente „Kartenleser" ist im gleichnamigen Zustandsautomaten rechts unten beschrieben, die Ausdrücke dort beziehen sich auf die Ports hier.

innerer Anschluss

Ein **Relaisanschluss** leitet Signale unverändert weiter (im Gegensatz zu einem **Transponderanschluss**).

Delegierungsverbinder verbinden innere und Relaisanschlüsse.



Genauso ist auch **Subsystem** ein relativer Begriff: Subsysteme können ihrerseits aus Subsystemen bestehen.

Komponenten werden durch Rechtecke mit dem Komponentensymbol in der rechten oberen Ecke dargestellt. Eine Komponente in UML 2 stellt einen „modularen Teil eines Systems dar und kapselt ihren Inhalt". Eine Komponente durchläuft den gleichen Lebenszyklus wie Software im Allgemeinen, ist also eine Einheit des Entwurfs und der Herstellung wie der Installation und Nutzung.

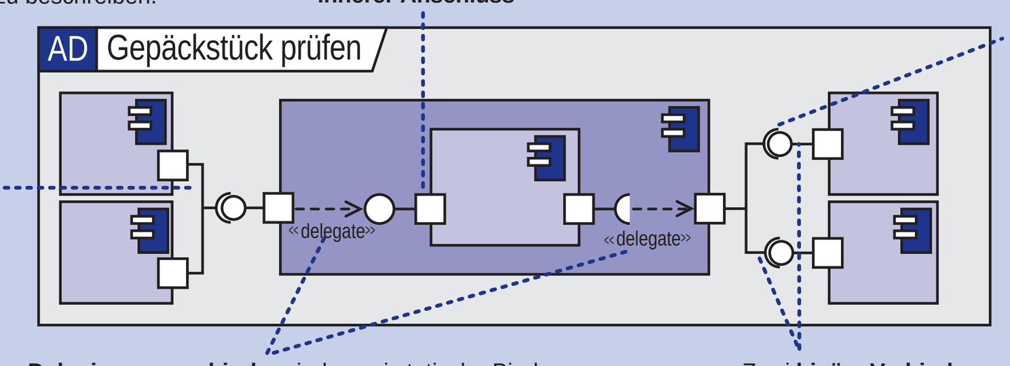
Unidirektionaler Anschluss

Verbinder

Auf einem noch feineren Detailgrad können **Montagediagramme** dazu benutzt werden, die Struktur zusammengesetzter Klassen zu beschreiben.

innerer Anschluss

Ein **ternärer Verbinder**



Anschlüsse können durch gebotene und genutzte Schnittstellen charakterisiert werden.

Delegierungsverbinder sind quasi statische Bindungen.

Zwei **binäre Verbinder**

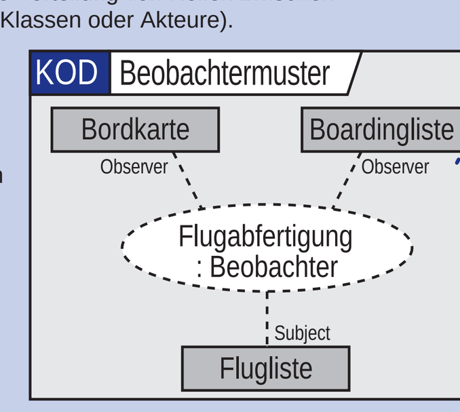
Kollaborationen

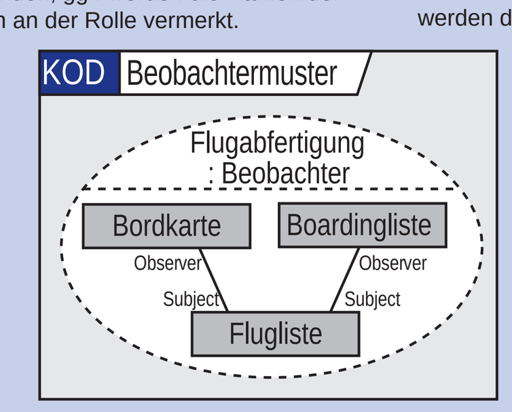
Kollaborationen definieren einen strukturellen Zusammenhang und eine Verteilung von Rollen zwischen einigen Elementen (z.B. Klassen oder Akteure).

Insbesondere in der Dokumentation der Anwendung von Architekturstilen und Entwurfsmustern werden Kollaborationen eingesetzt.

Die **Rollen** der Kollaboration werden mit dieser durch gestrichelte Linien verbunden, ggf. werden die Namen der Rollen an der Rolle vermerkt.

Alternativ können die Rollen innerhalb der Kollaboration angezeigt werden, Verbindungen zwischen den Rollen werden durch Assoziationen dargestellt.





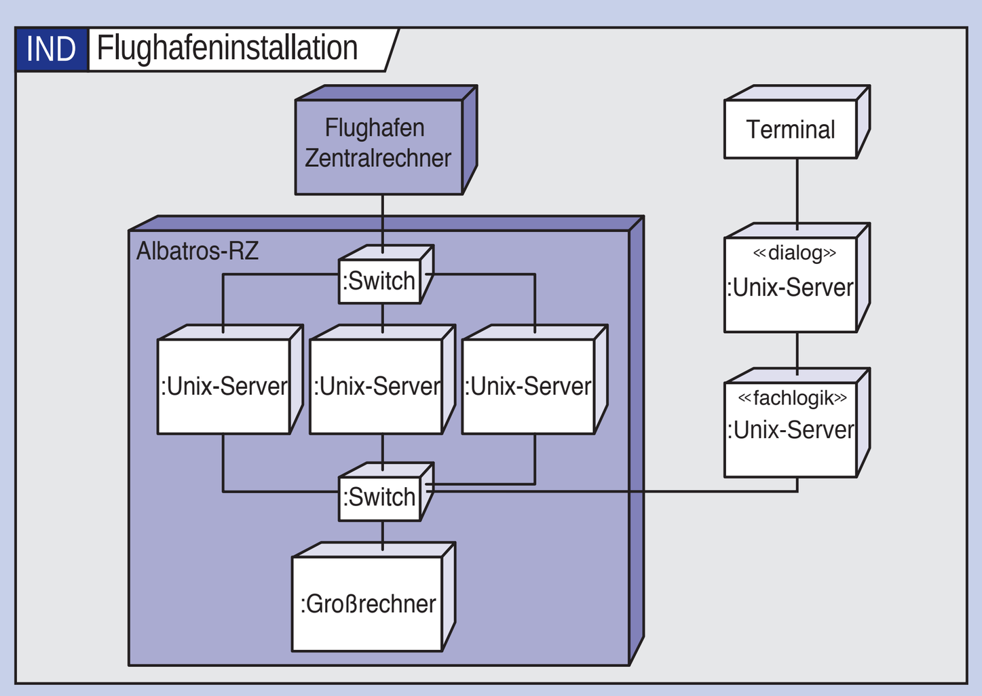
Eine Kollaboration wird als gestrichelte Ellipse dargestellt, in der die Namen der Instanz und des Typs der Kollaboration dargestellt werden.

Installationsdiagramme

Systemstrukturdiagramme stellen den physischen Aufbau eines Systems dar, also die Rechner, Netzwerke usw., die in einer Installation vorkommen.

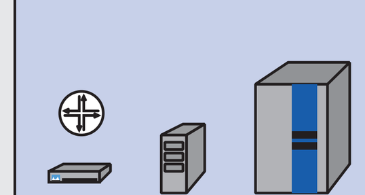
Installationsdiagramme können auf Typ- und auf Instanzebene beschrieben werden.

Ein Knoten repräsentiert einen Rechner oder eine sonstige aktive Komponente in einer Installation. Knoten können ineinander geschachtelt werden.



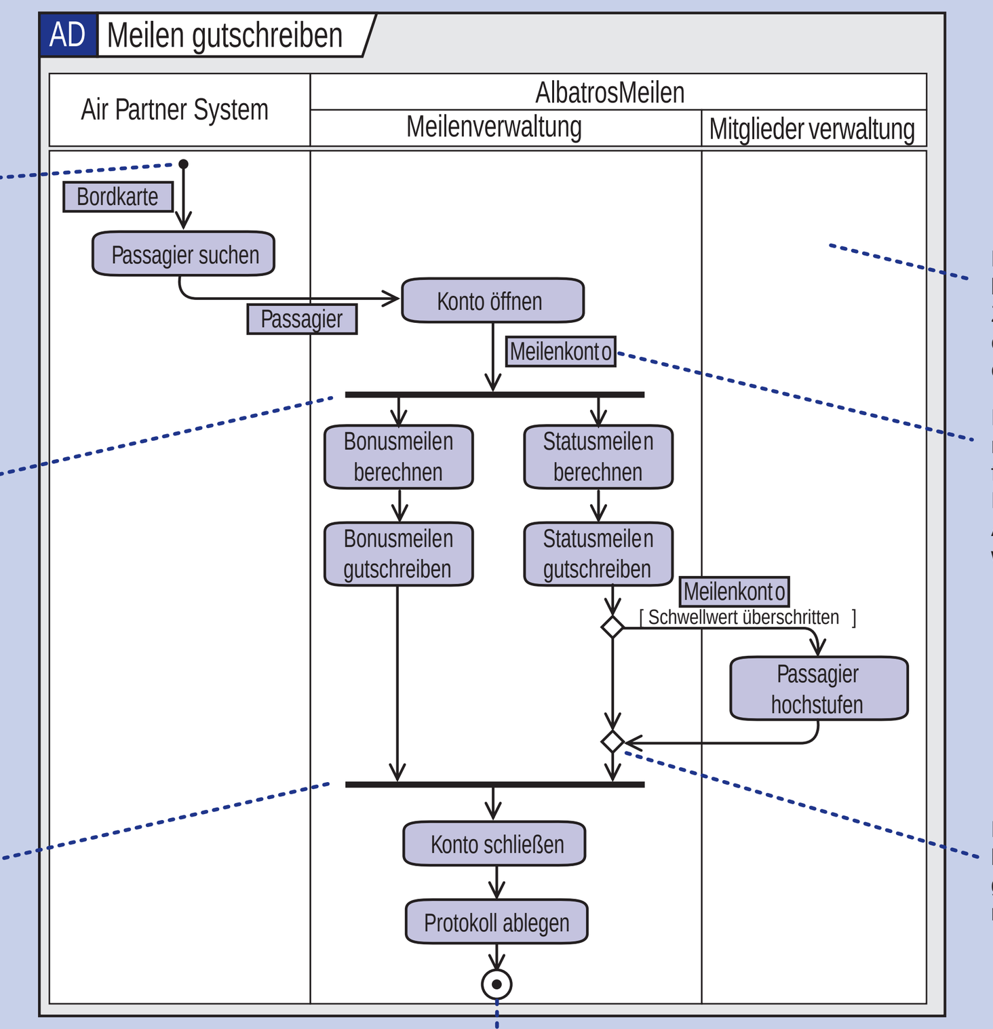
Eine Leitung repräsentiert eine Verbindung zwischen Knoten.

In Installationsdiagrammen werden oft sehr viele visuelle Stereotype wie die folgenden verwendet.



Aktivitätsdiagramme

Ein **Aktivitätsdiagramm** kann zur Modellierung von Abläufen jeder Art benutzt werden, z.B. für Geschäftsprozesse, Nutzfälle oder algorithmische Abläufe.



Ein **Anfangsknoten** markiert den Beginn eines Ablaufs.

Ein **Verzweigungsknoten** spaltet einen Kontrollfluss in mehrere auf. Die Verzweigung kann durch eine logische Bedingung gesteuert werden (z.B. or).

Ein **Vereinigungsknoten** führt mehrere Kontrollflüsse zu einem zusammen. Die Vereinigung kann durch eine logische Bedingung gesteuert werden (z.B. or).

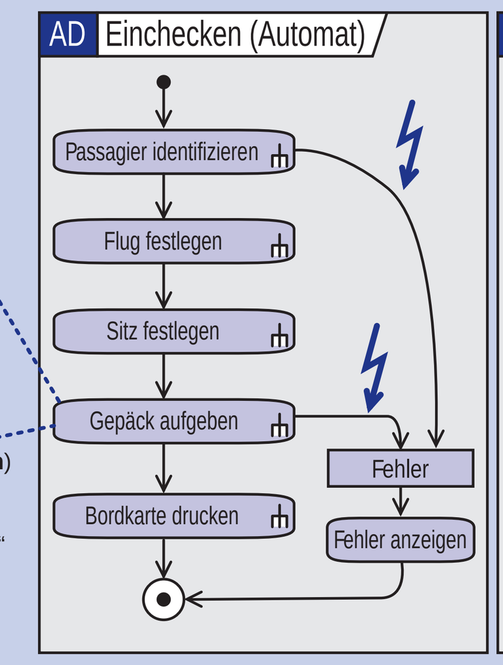
Eine **Partition** („Schwimmbahn" in UML 1.x) stellt einen Zuständigkeitsbereich dar, etwa ein Subsystem oder einen Akteur.

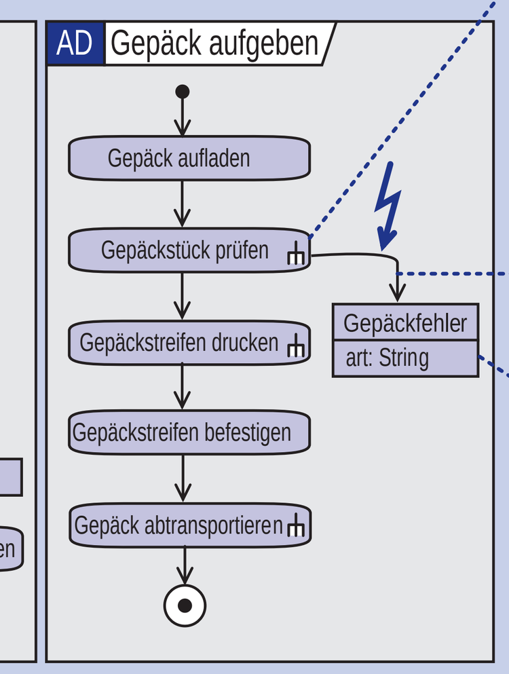
Ein **Objektflussknoten** repräsentiert Daten oder Artefakte (z.B. Dokumente, Briefe, Akten), die von einer Aktion zur nächsten geschickt werden.

Mit **Fallunterscheidungsknoten** werden Verzweigungen und Zusammenflüsse modelliert.

Ein **Endknoten** markiert das Ende einer Aktivität.

Aktivitätsdiagramme können auch dazu benutzt werden, um Abläufe z.B. von Geschäftsprozessen schrittweise zu verfeinern.





Eine **verfeinerte Aktion** wird durch ein Symbol in der rechten unteren Ecke angedeutet. In diesem Fall ist die Verfeinerung die Aktivität im Diagramm rechts außen.

Im Zustand „Gepäckstück aufgeben" (**geschützter Knoten**) kann eine **Ausnahme** des Typs „Fehler" ausgelöst werden, die von der Aktion „Fehler anzeigen" behandelt wird.

Im Zustand „Gepäckstück aufgeben" kann eine **Ausnahme** des Typs „Gepäckfehler" ausgelöst werden, die innerhalb der Aktivität „Gepäck aufgeben" nicht behandelt, und also zum Aufrufer der Aktivität propagiert wird (hier die gleich- namige Aktion in der Aktivität „Einchecken (Automat)").

Eine **Ausnahmekante** wird durch ein Blitzsymbol gekennzeichnet. Ein Knoten, von dem eine Ausnahmekante ausgeht, heißt „geschützter" Knoten.

Ein **Objektflussknoten** ist ein temporärer Speicher für Datenpakete, die zwischen Aktionen fließen.

Ein **Pin** ist ein notwendiger Parameter für den Aufruf einer Aktion, bzw. ein garantiertes Resultat.

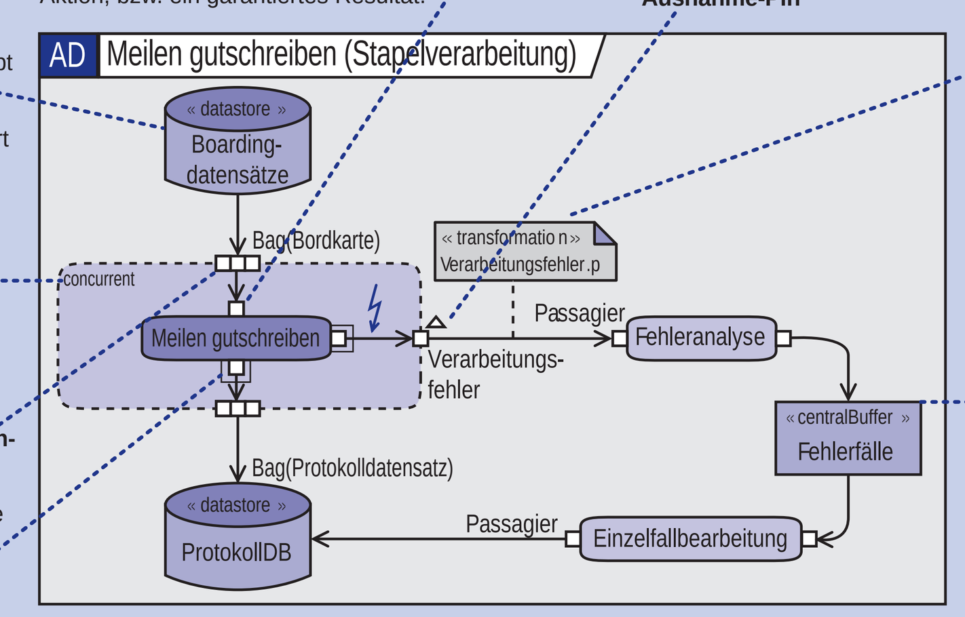
Ausnahme-Pin

Ein **Datenspeicher**(knoten) speichert alle in ihn abgelegten Datenpakete permanent, und gibt nur jeweils Kopien wieder ab.

Ein **Auffaltungsbereich** definiert einen Bereich einer Aktivität, in dem Massendaten nach verschiedenen Modi en gros verarbeitet werden:
- **concurrent**: gleichzeitig
- **iterative**: nacheinander
- **stream**: zeitlich überlappend nacheinander.

Die Parameter von Auffaltungsbereichen werden durch **Mengendaten-Pins** markiert.

Wenn nicht alle Pins notwendige Parameter sind, kann eine Teilmenge davon als **Parametermenge** ausgezeichnet werden.



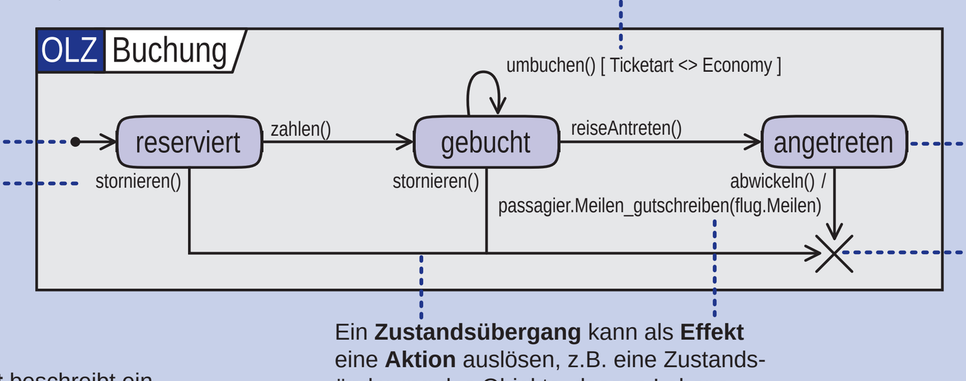
Transitionen können durch eine **Transformationsregel** ergänzt werden, um den Dateninhalt zu manipulieren (hier: Auswahl eines Attributs). Analog dazu gibt es **Selektionsregeln** an Transitionen, die von Datenknoten ausgehen. Sie erlauben es, zwischen mehreren verfügbaren Datenpaketen zu wählen.

Ein **Datenpuffer**(knoten) speichert Datenpakete temporär, und dient vor allem als Datenweiche.

Zustandsautomaten

Ein **Objektlebenszyklus** beschreibt das Verhalten aller Objekte einer Klasse, von der Instantierung zur Termination. Der Zustand eines Objekts besteht aus den Werten aller Attribute des Objekts.

Ein **Zustandsübergang** kann durch eine **Bedingung** (Guard) geschützt werden.



Anfangszustand

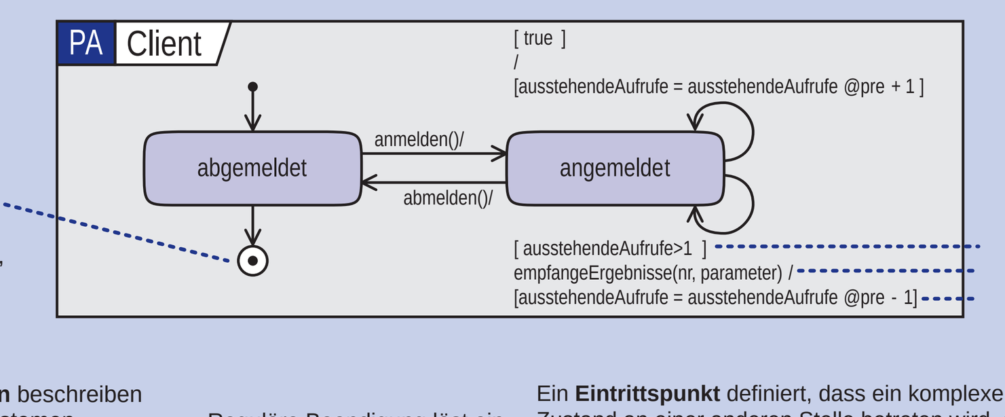
Ein **Ereignis** als **Auslöser** eines Zustandsübergangs

Zustand

Wenn der **Terminatorzustand** eines Zustandsautomaten erreicht wird, wird nicht nur der Zustandsautomat beendet, sondern das Objekt gelöscht, zu dem er gehört.

Ein **Zustandsübergang** kann als **Effekt** eine **Aktion** auslösen, z.B. eine Zustandsänderung des Objekts, dessen Lebenszyklus beschrieben wird.

Ein **Protokollzustandsautomat** beschreibt ein Verhalten ausschließlich über Zustände und Übergänge, aber ohne Seiteneffekte und Speicher.



Wenn der **Endzustand** eines Zustandsautomaten erreicht wird, wird die Region beendet, in der der Endzustand liegt.

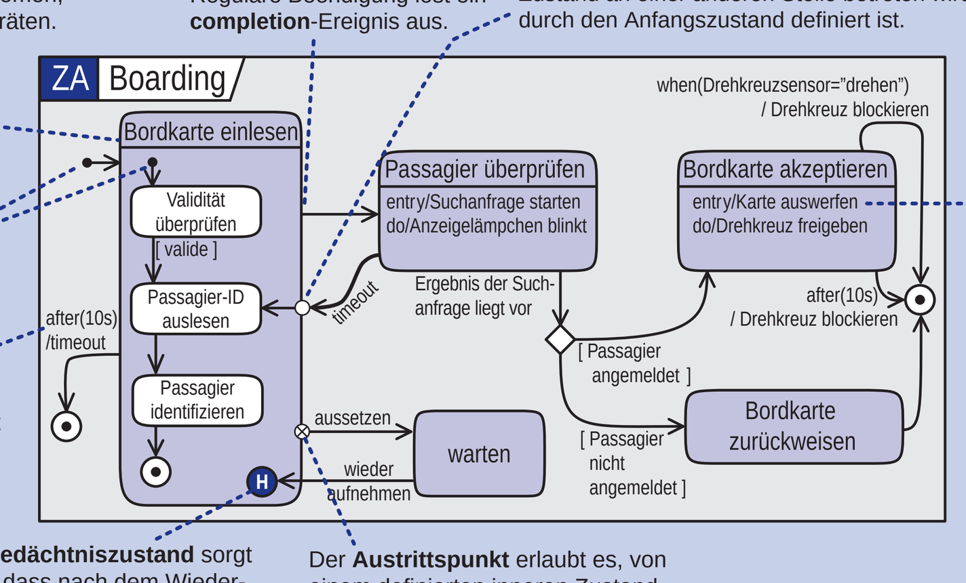
Die Zustandsübergänge von Protokoll-Zustandsautomaten verfügen über eine **Vorbedingung**, einen **Auslöser** und eine **Nachbedingung** (alle optional) – jedoch nicht über einen Effekt.

Protokollzustandsautomaten beschreiben das Verhalten von Softwaresystemen, Nutzfällen oder technischen Geräten.

Reguläre Beendigung löst ein **completion**-Ereignis aus.

Ein **Eintrittspunkt** definiert, dass ein komplexer Zustand an einer anderen Stelle betreten wird, als durch den Anfangszustand definiert ist.

Ein **komplexer Zustand** mit einer Region.



Der **Anfangszustand** markiert den voreingestellten Startpunkt von „Boarding" bzw. „Bordkarte einlesen".

Das **Zeitereignis** after(10s) löst einen Abbruch von „Bordkarte einlesen" aus.

Ein Zustand löst von sich aus bestimmte Ereignisse aus:
- **entry** beim Betreten;
- **do** während des Aufenthaltes;
- **completion** beim Erreichen des Endzustandes einer Unter-Zustandsmaschine
- **exit** beim Verlassen.

Diese und andere Ereignisse können als Auslöser für Aktivitäten herangezogen werden.

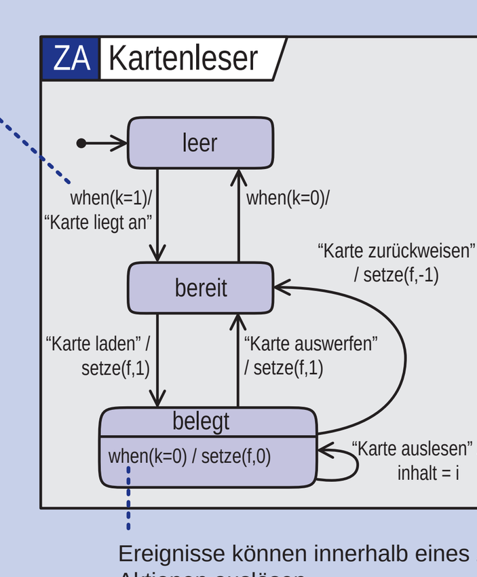
Der **Gedächtniszustand** sorgt dafür, dass nach dem Wiederaufnehmen der gleiche Zustand wie vor dem Aussetzen eingenommen wird.

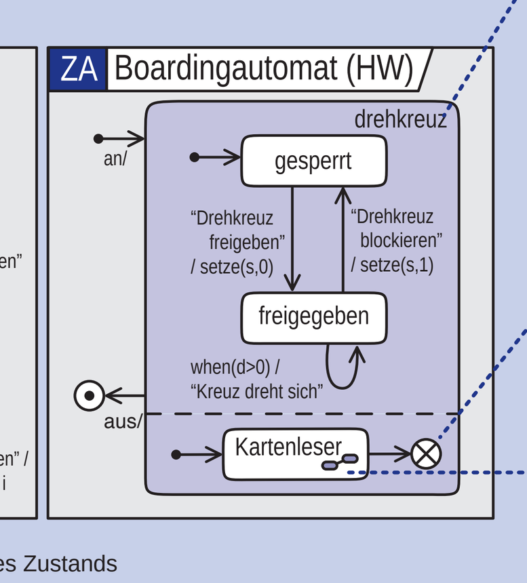
Der **Austrittspunkt** erlaubt es, von einem definierten inneren Zustand aus den Oberzustand zu verlassen.

Auch Zeit- und Änderungsereignisse können Zustandsübergänge auslösen:

- **after** definiert das Verstreichen eines Intervalls;
- **when** definiert einen Zustandswechsel.

Zustände und zeitlicher Bezugsrahmen werden über den umgebenden Classifier definiert, hier die Werte der Ports, siehe das Montagediagramm „Abfertigung" links oben.





Ein Zustand kann eine oder mehrere **Regionen** enthalten, die wiederum Zustandsautomaten enthalten können. Wenn ein Zustand mehrere Regionen enthält, werden diese in verschiedenen Abteilen angezeigt, die durch gestrichelte Linien voneinander getrennt sind. Regionen können benannt werden. Alle Regionen werden parallel zueinander abgearbeitet.

Wenn ein **Regionsendzustand** erreicht wird, wird der gesamte *komplexe* Zustand beendet, also auch alle parallelen Regionen.

Ein **verfeinerter Zustand** verweist auf einen Zustandsautomaten (angedeutet von dem Symbol unten links), der das Verhalten des Zustandes definiert.

Ereignisse können innerhalb eines Zustands Aktionen auslösen.

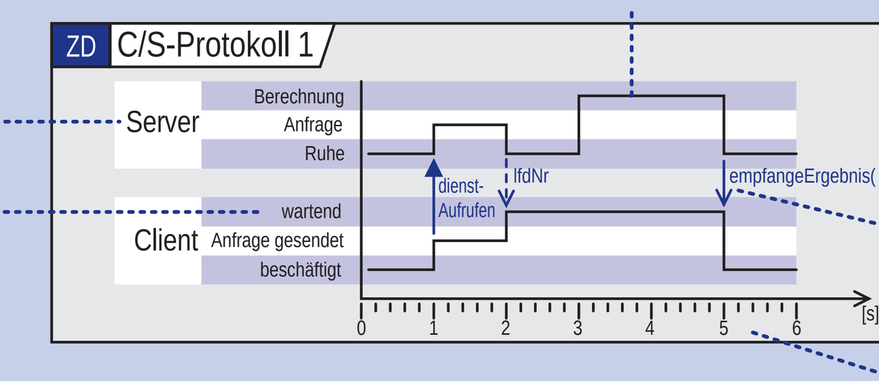
Zeitdiagramme

Zeitdiagramme sind eine Form von Interaktionsdiagrammen, die besonders zur Modellierung und Visualisierung komplexer zeitlicher Abläufe geeignet sind.

Ähnlich zu Sequenzdiagrammen besitzen auch Zeitdiagramme eine Art **Lebenslinie**, die hier aber auch den Zustand eines Interaktionspartners zu einem Zeitpunkt ausdrückt.

Interaktionspartner

Zustände eines Interaktionspartners



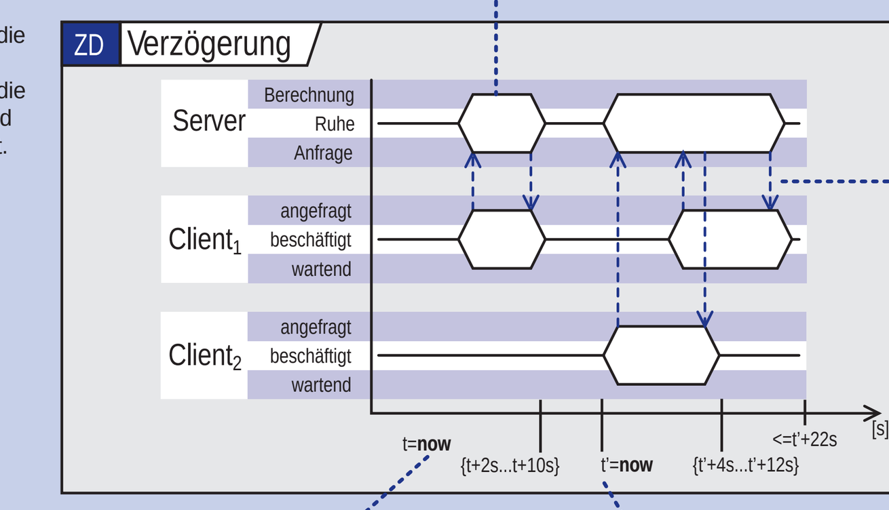
Nachrichtenaustausch zwischen Interaktionspartnern

Im Unterschied zu Sequenzdiagrammen besitzen Zeitdiagramme eine echte **Zeitskala**, hier eine absolute (metrische) Skala.

Um zeitliche Abläufe stärker abstrahiert darzustellen, können Abschnitte eines zeitlichen Ablaufes zu Phasen zusammengestellt werden.

Wenn während eines Zeitabschnitts verschiedene Zustände eingenommen werden können, ohne dass zwischen ihnen unterschieden werden soll, kann dies durch eine **Phase** ausgedrückt werden.

Mit Zeitdiagrammen lässt sich die **Koordinierung verteilter Zustände** modellieren, sofern die Zahl der Interaktionspartner und ihrer Zustände nicht zu groß ist.



Nachrichtenaustausch zwischen Interaktionspartnern

Zeitbedingungen können als Zeitpunkte oder als Intervalle, und absolut oder relativ angegeben werden. Um einen Zeitpunkt als Bezugspunkt auszuwählen, kann das Schlüsselwort **now** benutzt werden.

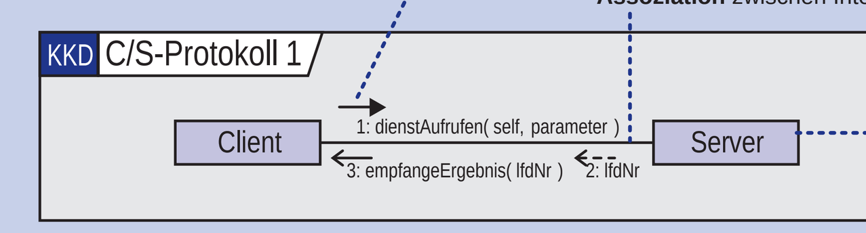
Statt einer absoluten, kann in einem Zeitdiagramm auch eine **relative Zeitskala** verwendet werden. Darauf werden nur einzelne Punkte definiert, ggf. benannt, und in einschränkenden Ausdrücken benutzt.

Kommunikationsdiagramme

Kommunikationsdiagramme („Kollaborationsdiagramme" in UML 1.x) sind eine Form von Interaktionsdiagrammen, die besonders gut dann geeignet sind, wenn wenige Nachrichten ausgetauscht werden, aber die Beziehungen zwischen den Interaktionspartnern hervorgehoben werden sollen.

Die **Reihenfolge** von Nachrichten in einem Kommunikationsdiagramm kann lediglich durch **Sequenznummern** dargestellt werden. Richtung und Typ der Nachricht werden durch einen **kleinen Pfeil** angezeigt

Assoziation zwischen Interaktionspartnern

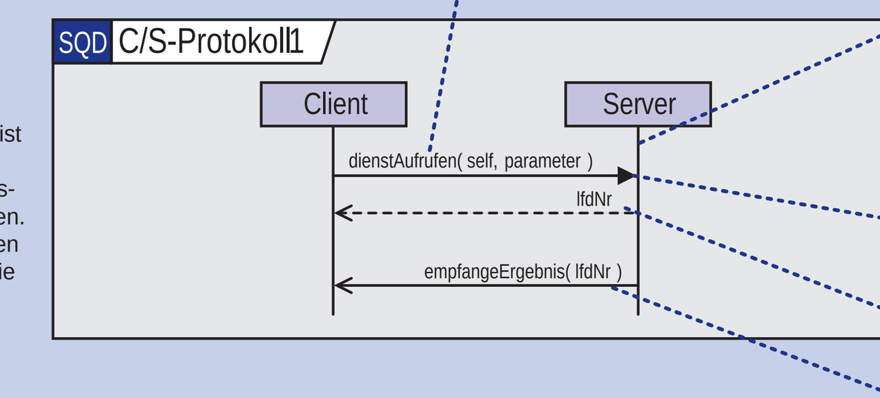


Als **Interaktionspartner** kann jeder Classifier dienen, z.B. eine Klasse, ein Akteur oder ein Knoten.

Sequenzdiagramme

Sequenzdiagramme sind eine Form von Interaktionsdiagrammen, die besonders gut dann geeignet sind, wenn wenige Interaktionspartner viele Nachrichten austauschen, und wenn komplexe Muster des Nachrichtenaustauschs zu modellieren sind.

Die Menge der **möglichen Nachrichten** an einen Interaktionspartner wird durch dessen Signatur bestimmt. In diesem Fall muss es z.B. in der Klasse „Server" eine für „Client" sichtbare Operation „dienstAufrufen" mit den entsprechenden Parametern geben.

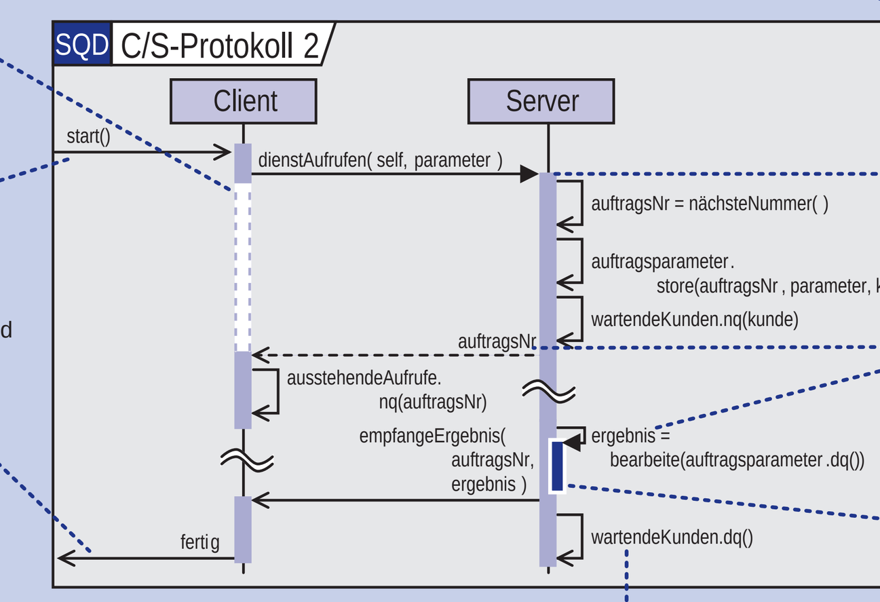


Das Sequenzdiagramm rechts ist **bedeutungsgleich** mit den gleichnamigen Kommunikations- und Zeitdiagrammen weiter oben. Die drei Diagramme beschreiben exakt die gleiche Interaktion. Die drei Diagrammtypen sind aber nicht austauschbar.

In einem Sequenzdiagramm wird für jeden Interaktionspartner eine **Lebenslinie** angezeigt. Nachrichten zwischen den Interaktionspartnern werden durch Pfeile zwischen den Lebenslinien dargestellt.

Der **Typ einer Nachricht** wird durch die Form des Pfeils angezeigt:
- **Aufrufe** sind synchron, d.h., die Aktivierung des sendenden Interaktionspartners wird ausgesetzt;
- **Antworten** reaktivieren die Aktivierung des empfangenden Interaktionspartners;
- **synchrone Nachrichten** lassen die Aktivierung unverändert.

ausgesetzte Aktivierung



Anfang und **Ende** einer Interaktion können durch Nachrichten von bzw. zum Rand des Interaktionsdiagramms dargestellt werden.

Ein **Aktivierungsbalken** zeigt an, dass die Kontrolle bei einem Interaktionspartner liegt, bzw. wie tief Aufrufe geschachtelt sind.

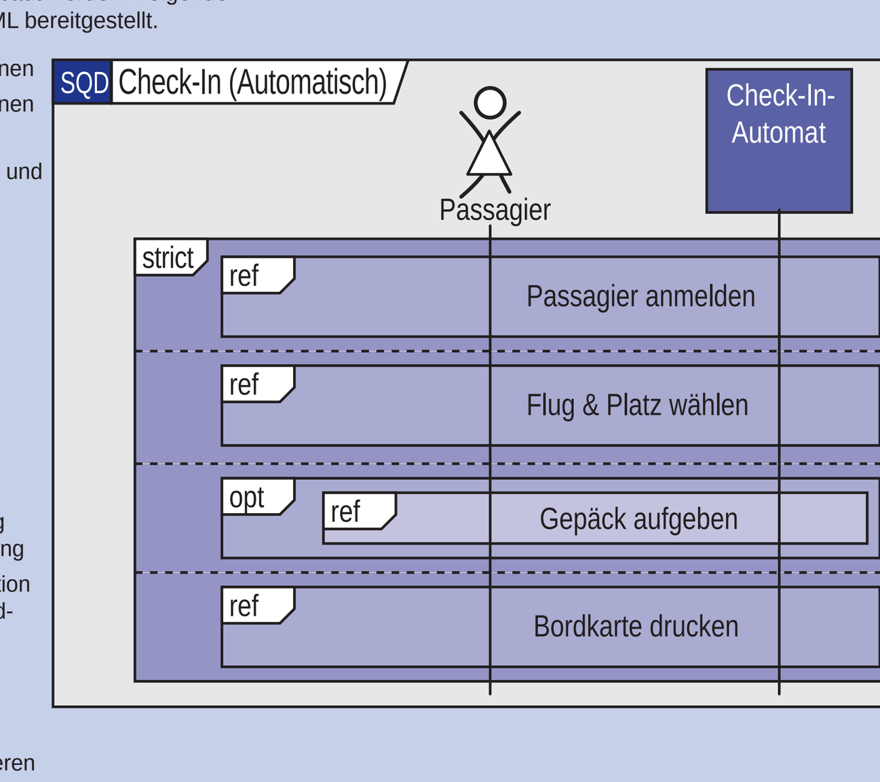
Das **Resultat** eines Aufrufs wird typischerweise in einem Attribut abgelegt. Dieses Attribut kann an der Antwort-Nachricht oder direkt im Aufruf angegeben werden.

Bei einem **Selbstaufruf** werden Aktivierungsbalken übereinander gelegt.

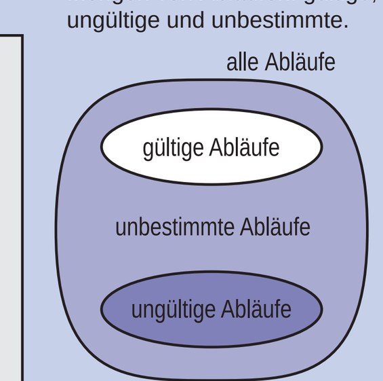
Bei einer synchronen Nachricht wird dagegen **kein Resultat** erwartet.

Komplexe Interaktionen können modular durch **Interaktionsoperatoren** aufgebaut werden. Folgende Operatoren werden von der UML bereitgestellt.

strict	Abfolge von Interaktionen
seq	Abfolge von Interaktionen je Interaktionspartner
loop	Schleife mit minimaler und maximaler Anzahl von Durchläufen
par	Parallele Ausführung (Interleaving)
critical	Ununterbrochene Ausführung („kritische Region")
ref	Referenz auf eine Interaktion
opt	optionale Ausführung
alt	alternative Ausführung nach Fallunterscheidung
brk	Abbruch einer Interaktion und Ausnahmebehandlung
ignore	Ausblenden von Nachrichten
consider	dual zu **ignore**, Ausblenden aller anderen Nachrichten



Eine Interaktion definiert drei Mengen von Abläufen: gültige, ungültige und unbestimmte.



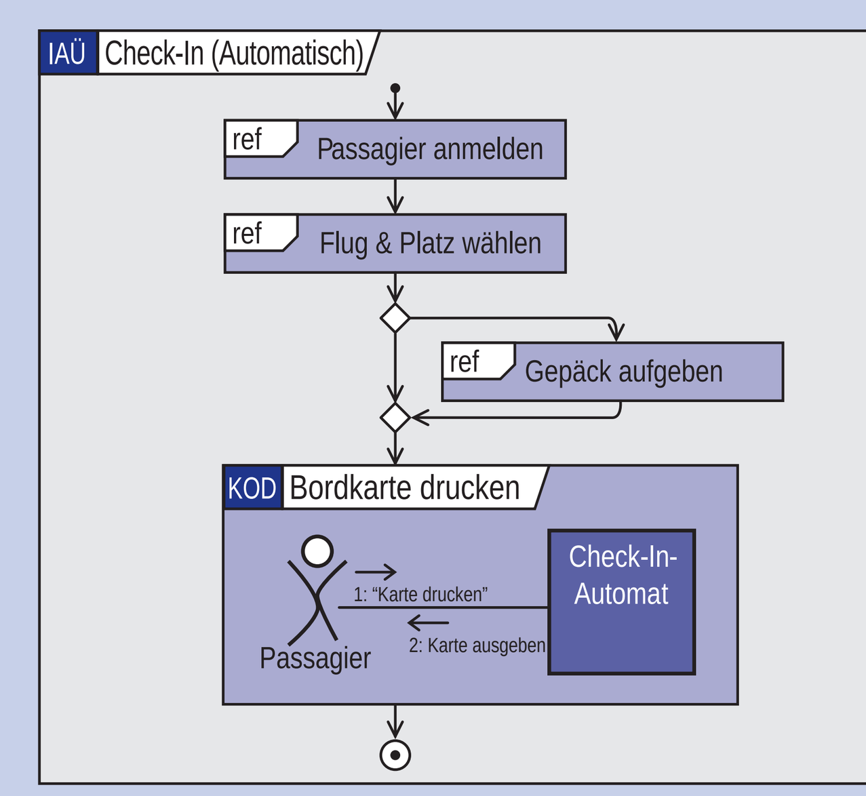
Unbestimmte Abläufe entsprechen nicht-durchgeführten Testfällen, ungültige Abläufe kann man als Gegenbeispiele verstehen.

assert	Ausschluss von Unbestimmtheit, d.h. unbestimmte Abläufe werden als ungültig definiert
neg	Definition von ungültigen Abläufen

Interaktionsübersichten

Wenn man den Zusammenhang zwischen vielen Interaktionen graphisch darstellen will, kann man **Interaktionsübersichten** nutzen. Hier können einige Ausdrucksmittel von Aktivitätsdiagrammen benutzt werden, die Semantik bleibt aber die gleiche wie bei Interaktionen.

Im einfachsten Fall enthält eine Interaktionsübersicht lediglich Verweise auf Interaktionen, die anderswo definiert sind. Es ist aber auch möglich, diese Interaktionen direkt innerhalb der Interaktionsübersicht, also „in-line"zu definieren. Hierbei kann natürlich jede Art von Interaktionsdiagramm benutzt werden.

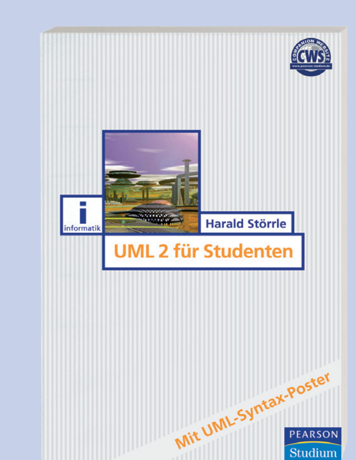


Folgende Ausdrucksmittel von Aktivitätsdiagrammen sind zulässig:
- **Aufruf** von Interaktionen
- **Einbettung** von Interaktionen
- **Fallunterscheidung** (entspricht **par**);
- **Parallelbearbeitung** (entspricht **alt**).

Andere Ausdrucksmittel sind nicht gestattet!

Die Diagramme dieses Posters stammen aus dem Buch

UML 2 für Studenten
Harald Störrle
Pearson Studium 2005
ISBN 3-8273-7143-0
€ 19,95 [D], € 20,60 [A]



www.pearson-studium.de

PEARSON Studium