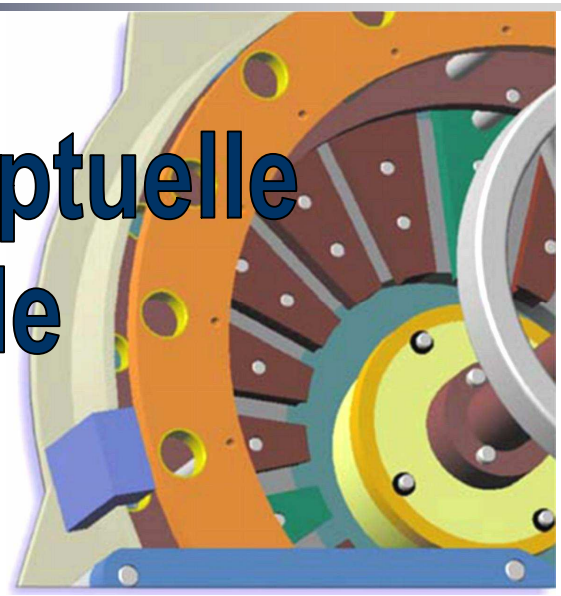
Vorlesung Softwaretechnik
(SS 2005)

Prof. Dr. O. Drobnik Michael Berschin, Michael Lauer

Professur für Telematik / ABVS
J. W. Goethe-Universität / Frankfurt am Main

Kapitel 4

Konzeptuelle Modelle



Modellierung

Modellierung ist die Technik zur Abstraktion der Realität mit der Konzentrierung auf wesentliche Aspekte.

Modellierung

Modellierung ist die Technik zur Abstraktion der Realität mit der Konzentrierung auf wesentliche Aspekte.

- Modellierung:
  - Ist unabdingbar für die Entwicklung (komplexer) Systeme,
  - hilft die gewünschte Struktur und das Verhalten eines Systems zu beschreiben und zu diskutieren,
  - hilft ein System besser zu verstehen, indem man es ausschnittsweise betrachtet und vertieft,
  - hilft frühzeitig Probleme eines Systems zu erkennen und zu beheben.

Modellierung

Modellierung ist die Technik zur Abstraktion der Realität mit der Konzentrierung auf wesentliche Aspekte.

- Modellierung ermöglicht die
  - Visualisierung,
  - Spezifizierung,
  - Konstruktion und
  - Dokumentation von Systemen.
- Durch die Technik der Modellierung eines Aspekts eines Systems kommt man zu einem *konzeptuellen Modell*.

Konzeptuelle Modelle

- Konzeptuelle Modelle im Analyseprozess:
  - Präsentieren abstrakte Beschreibungen des zu entwickelten Systems,
  - entstehen durch methodenbasierte Analyseansätze,
  - orientieren sich an Begriffen aus der Informatik (Objekte, Funktionen) anstelle von Begriffen aus dem Anwendungsgebiet.
  - Ermöglichen damit den Brückenschlag zwischen Analyse und Entwurf.
- Konzeptuelle Modelle sind prinzipiell unvollständig:
  - Gefragt ist hier keine alternative **Darstellung**, sondern eine wirkliche **Abstraktion** des zu untersuchenden Systems.
  - Eine Darstellung enthält alle Informationen des dargestellten Systems.
  - Eine Abstraktion vereinfacht und enthält nur wesentliche Eigenschaften.
- Verschiedene Modelle abstrahieren in verschiedene Richtungen.
- Den Abstraktionsfokus eines Modells nennt man *Modellierungs-priorität*.

Konzeptuelle Modelle

- Viele Analysemethoden enthalten eine Anzahl von konzeptuellen Modellen als Teil der Methode. Es wird behauptet (normalerweise ohne Begründung), dass diese Modelle zur Darstellung des Systems ausreichen.
- Bei der Analyse sollte man sich nicht auf die Modelle einer bestimmten Methode beschränken.
- Zum Beispiel enthalten objektorientierte Methoden selten ein Datenflussmodell. Es zeigt sich aber, dass Datenflussmodelle auch in einer objektorientierten Analyse nützlich sein können, da sie für Endbenutzer leichter zu verstehen sind und die Datenflüsse dabei helfen, Objekte und Operationen darauf zu identifizieren.
- Man unterscheidet prinzipiell zwischen **formalen Modellen** und **informalen Modellen**.

4.1 Informale Modelle

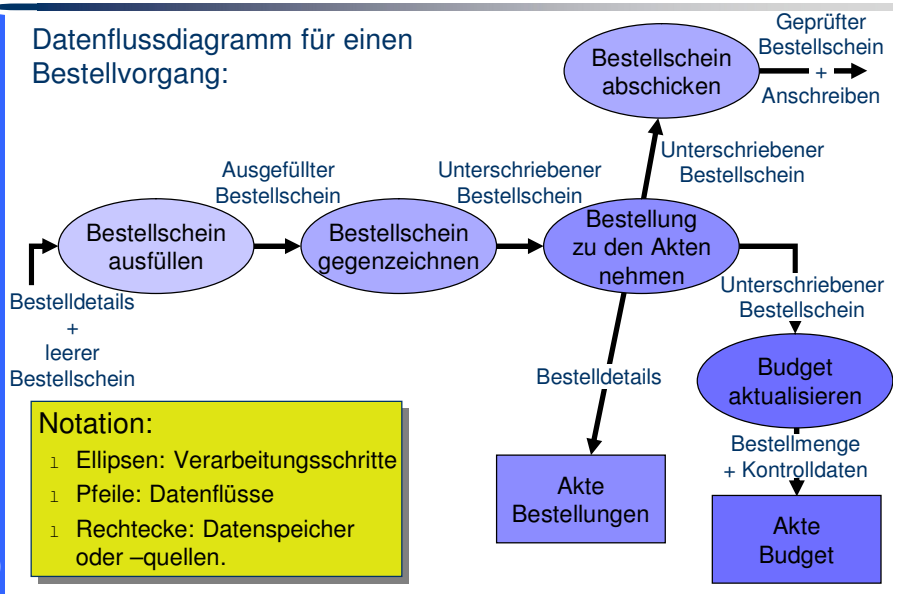
- Einige Typen informaler Modellen sind:
  - **Datenflussmodelle:** Datenflussdiagramme (*data flow diagrams*) können zeigen, wie Daten in den verschiedenen Stufen des Systems verarbeitet werden.
  - **Zusammensetzungsmodelle:** Zusammensetzungsdiagramme (*Entity-Relationship diagrams*) können zeigen, dass (und wie) Teile des Systems aus anderen Teilen zusammengesetzt sind.
  - **Klassifizierungsmodelle:** Objektklassen- und Vererbungsdiagramme können zeigen, dass Teile des Systems gemeinsame Eigenschaften haben.
  - **Reiz-Antwort-Modelle:** Zustandsübergangsdiagramme (*stimulus-response diagrams*) können zeigen, wie das System auf interne und externe Ereignisse reagiert.
  - **Prozessmodelle:** Prozessmodelle können die Hauptaktivitäten und Resultate bei der Ausführung eines Vorgangs zeigen.

Datenflussmodelle

- **Datenflussmodelle:**
  - Sind beliebt seit DeMarco (1978).
  - Ermöglichen auf einfache und intuitive Art, die Datenverarbeitung in einem System zu zeigen.
  - Werden bei der Analyse verwendet, um den Datenfluss in einem existierenden System zu untersuchen.
  - Sind wertvoll, da sie den Analysten helfen, die Bewegung von Daten zu verfolgen und zu dokumentieren.
  - Können potentiellen Benutzern erklärt werden, die dadurch an der Analyse teilnehmen können.
- Grundlegende Vorgehensweise:
  - Der Datenfluss wird durch eine Folge von Verarbeitungsschritten modelliert.
  - In jedem Schritt werden die Daten transformiert.
  - Beim Entwurf entspricht jede Transformation einer Programmfunktion.
  - Bei der Analyse werden Transformationen von Menschen oder Computern vorgenommen.

Datenflussmodelle

Datenflussdiagramm für einen Bestellvorgang:



Abstraktionsstufen

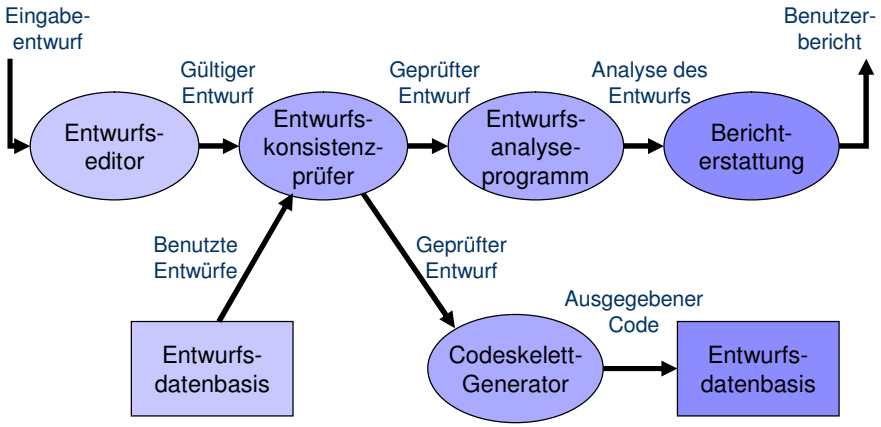
- Datenflussdiagramme können auf verschiedenen Abstraktionsstufen angeordnet werden (auch hierarchisch).
- In der Theorie werden Datenflussmodelle „von oben nach unten“ aufgestellt.
- In der Praxis ist das unmöglich, da man auf mehreren Ebenen zur selben Zeit arbeitet.
- Allgemeine Modelle können später aus speziellen abstrahiert werden.

Perspektiven

- Datenflussmodelle können statt einer funktionalen Perspektive (jede Transformation entspricht einer einzelnen Funktion) auch eine architekturorientierte Perspektive zeigen – in diesem Falle werden verschiedene Subsysteme identifiziert, die Informationen austauschen.
- Das Datenflussmodell zeigt dann den Informationsaustausch zwischen den Subsystemen.
- Datenflussdiagramme sind nicht dafür geeignet, Subsysteme mit komplexen Schnittstellen genau zu beschreiben, da für jede Art von Eingabedaten ein eigenes Diagramm benötigt wird. Hier sind beispielsweise Objektmodelle besser zu gebrauchen.

Datenflussmodelle

Architekturorientiertes Datenflussmodell für ein CASE-Tool:



Semantische Datenmodelle

- Die meisten großen Softwaresysteme verwenden große Datenbasen. Ein wichtiger Bestandteil der Entwicklung solcher Systeme ist die Definition der logischen Struktur der verarbeiteten Daten.
- Während Datenflussmodelle die Modellierungspriorität auf die beteiligten Quellen und Senken legen, enthalten **semantische Datenmodelle** Informationen über die Semantik (Bedeutung) der ausgetauschten Daten.
- Semantische Datenmodelle werden normalerweise graphisch dargestellt, da eine graphische Notation im Allgemeinen auch für Manager und Benutzer verständlicher ist.

Semantische Datenmodelle

- Ein aus dem Datenbankbereich bekannter möglicher Ansatz ist die Verwendung eines *relationalen Datenmodells* (Codd 1970):
  - Daten sind eine Menge von Tabellen.
  - Manche dieser Tabellen haben gemeinsame Schlüssel.

Angestellter	Personal-nummer	Abteilung
Meier	14093-2	EDV
Siebert	21145-9	Verwaltung

Personal-nummer	Gehalt
14093-2	3700 €
21145-9	3200 €

- So können die Zusammenhänge zwischen Daten beschrieben werden, ohne die physikalische Auslegung der Datenbasis zu beachten.

Entity-Relationship-Modell

- Nachteile bei der Verwendung von relationalen Datenmodellen:
  - Daten haben nur implizite Typen, d.h. Typangaben können nur aus den Relationsnamen hergeleitet werden.
  - Logische Zusammenhänge zwischen Daten werden nur implizit dargestellt (durch gemeinsame Einträge in Tabellen), nicht explizit.
  - Die Relationen können keine Namen oder Attribute haben.
- Ein besserer Ansatz ist das *Entity-Relationship-Modell* (Chen 1976, hier mit Erweiterungen von Hull und King 1987).
- ER-Modelle werden gerne zum Entwurf von Datenbasen verwendet, da sie sich leicht auf relationale Datenbasen abbilden lassen. Durch ihre expliziten Typen und Vererbungsmöglichkeiten eignen sie sich auch für objektorientierte Datenbasen.

Entity-Relationship-Modell

Die Grundideen des ER-Modells:

Entity-Relationship-Modell

Die Grundideen des ER-Modells:

- Entities sind Dateneinträge – vergleichbar mit Verbundtypen (**record** oder **struct**) in Programmiersprachen.

Entity-Relationship-Modell

Die Grundideen des ER-Modells:

- Entities sind Dateneinträge – vergleichbar mit Verbundtypen (**record** oder **struct**) in Programmiersprachen.
- Es gibt es Relationen zwischen Entities.

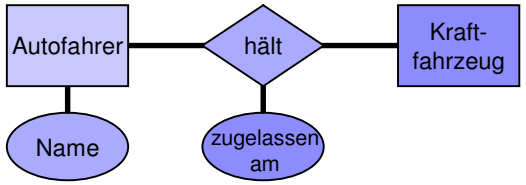
Entity-Relationship-Modell

Die Grundideen des ER-Modells:

- Entities sind Dateneinträge – vergleichbar mit Verbundtypen (**record** oder **struct**) in Programmiersprachen.

Entity-Relationship-Modell

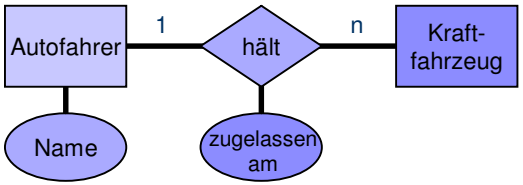
Die Grundideen des ER-Modells:



- Entities und Relationen verfügen über Attribute.
- Attribute sind normalerweise atomar, d.h. sie werden nicht weiter unterteilt.
- Attribute müssen nicht notwendigerweise den Basistypen von Programmiersprachen entsprechen

Entity-Relationship-Modell

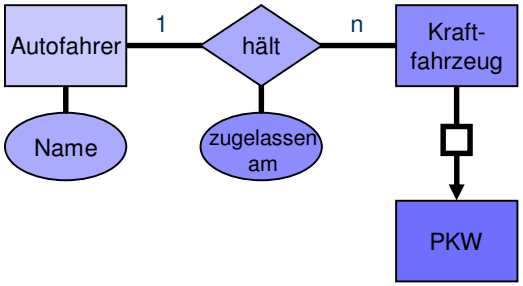
Die Grundideen des ER-Modells:



- Kardinalität von Relationen zwischen Entities:
  - **1:1** Eine Entity steht in Relation mit genau einer anderen.
  - **1:n** Eine Entity steht in Relation mit (möglicherweise) mehreren anderen.
  - **m:n** Mehrere Entities stehen in Relation mit mehreren anderen.

Entity-Relationship-Modell

Die Grundideen des ER-Modells:



- Zu den ER-Diagrammen gibt es eine Typenhierarchie:
  - Es gibt eine spezielle Vererbungsrelation.
  - Subtypen erben die Attribute der übergeordneten Typen.
  - Es können private Attribute hinzugefügt werden.

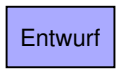
Entity-Relationship-Modell

Beispiel eines *Entwurfs* in einem CASE-Tool:

- Das CASE-Tool soll Entwürfe und deren logische Zusammenhänge bei Bedarf auf dem Bildschirm abbilden.
- Die Details der Bildschirmdarstellung sind dabei für die Zwecke anderer Systemkomponenten unerheblich.

Entity-Relationship-Modell

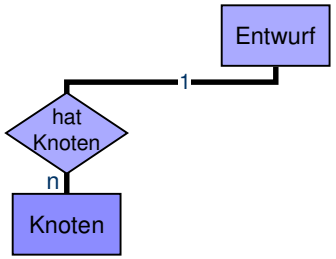
Beispiel eines *Entwurfs* in einem CASE-Tool:



Entwürfe sind gerichtete Graphen:

Entity-Relationship-Modell

Beispiel eines *Entwurfs* in einem CASE-Tool:

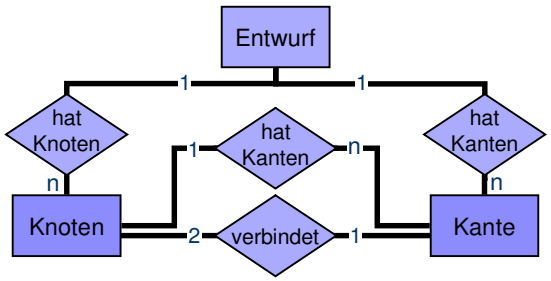


Entwürfe sind gerichtete Graphen:

- Knoten entsprechen unterschiedlichen Typen.

Entity-Relationship-Modell

Beispiel eines *Entwurfs* in einem CASE-Tool:

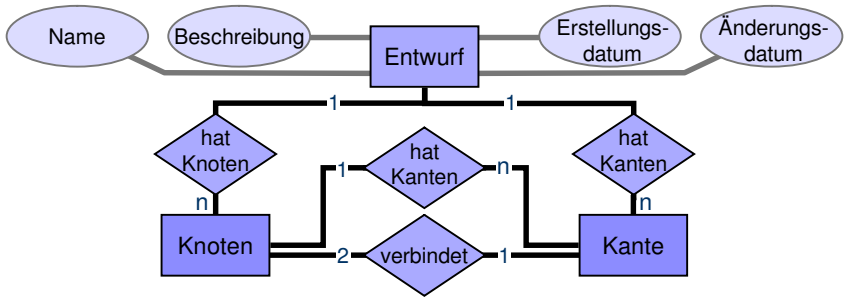


Entwürfe sind gerichtete Graphen:

- Knoten entsprechen unterschiedlichen Typen.
- Kanten entsprechen Zusammenhängen zwischen Knoten.

Entity-Relationship-Modell

Beispiel eines *Entwurfs* in einem CASE-Tool:

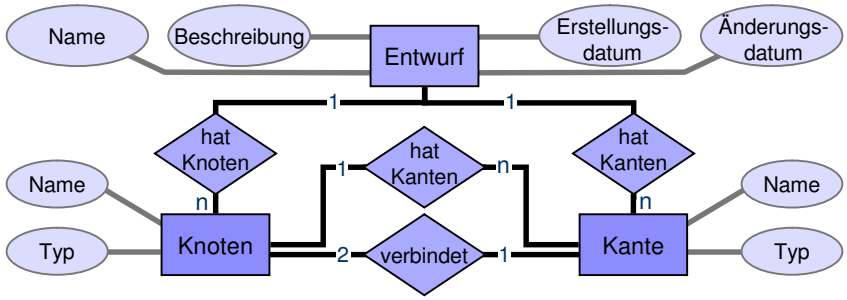


Entwürfe verfügen über Attribute:

- Entwurfsname,
- Entwurfsbeschreibung,
- Erstellungsdatum,
- und Änderungsdatum.

Entity-Relationship-Modell

Beispiel eines *Entwurfs* in einem CASE-Tool:



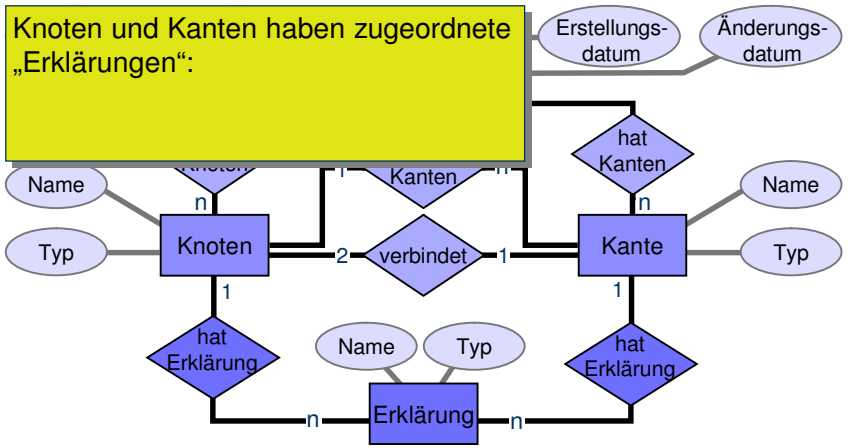
Knoten und Kanten haben Attribute:

- Namen,
- Typangabe.

Entity-Relationship-Modell

Beispiel eines *Entwurfs* in einem CASE-Tool:

Knoten und Kanten haben zugeordnete „Erklärungen“:

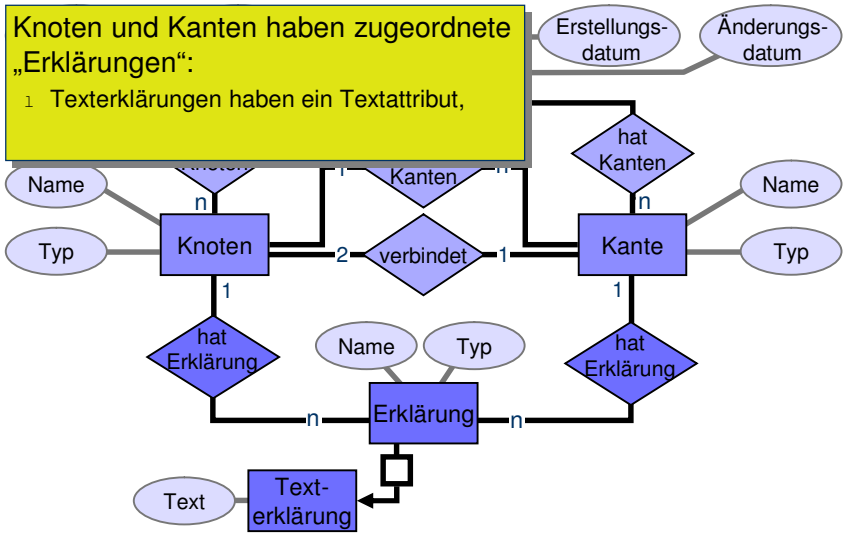


Entity-Relationship-Modell

Beispiel eines *Entwurfs* in einem CASE-Tool:

Knoten und Kanten haben zugeordnete „Erklärungen“:

- Texterklärungen haben ein Textattribut,

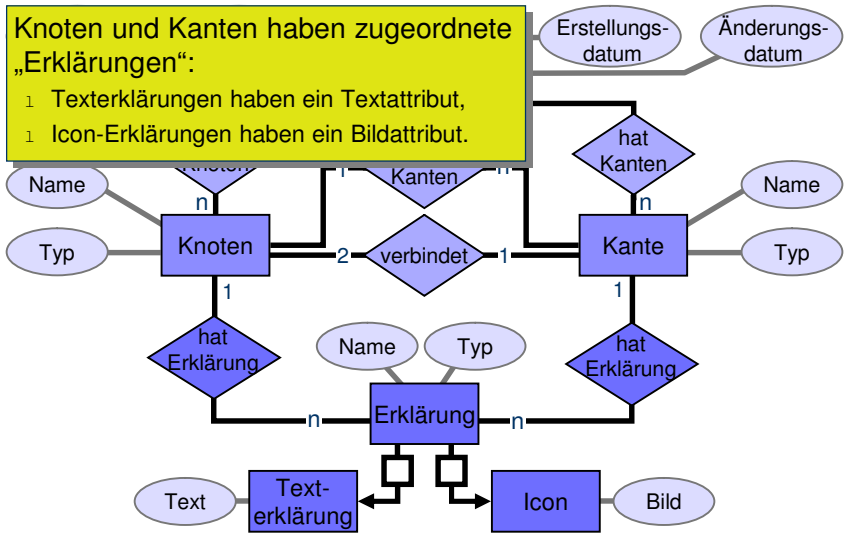


Entity-Relationship-Modell

Beispiel eines *Entwurfs* in einem CASE-Tool:

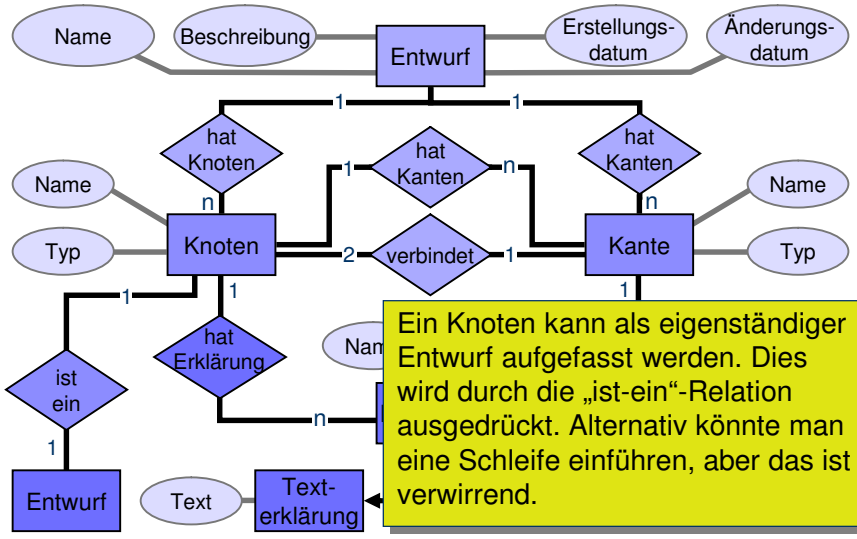
Knoten und Kanten haben zugeordnete „Erklärungen“:

- Texterklärungen haben ein Textattribut,
- Icon-Erklärungen haben ein Bildattribut.



Entity-Relationship-Modell

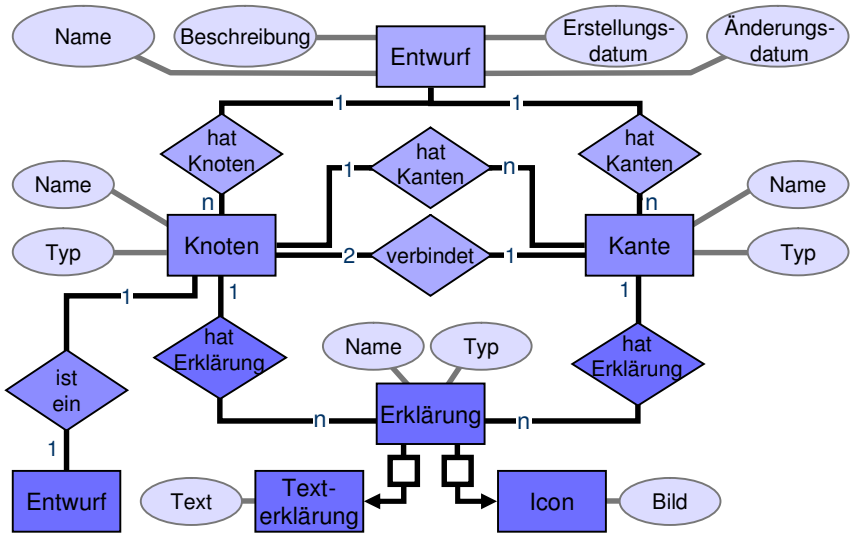
Beispiel eines *Entwurfs* in einem CASE-Tool:



Ein Knoten kann als eigenständiger Entwurf aufgefasst werden. Dies wird durch die „ist-ein“-Relation ausgedrückt. Alternativ könnte man eine Schleife einführen, aber das ist verwirrend.

Entity-Relationship-Modell

Beispiel eines *Entwurfs* in einem CASE-Tool:

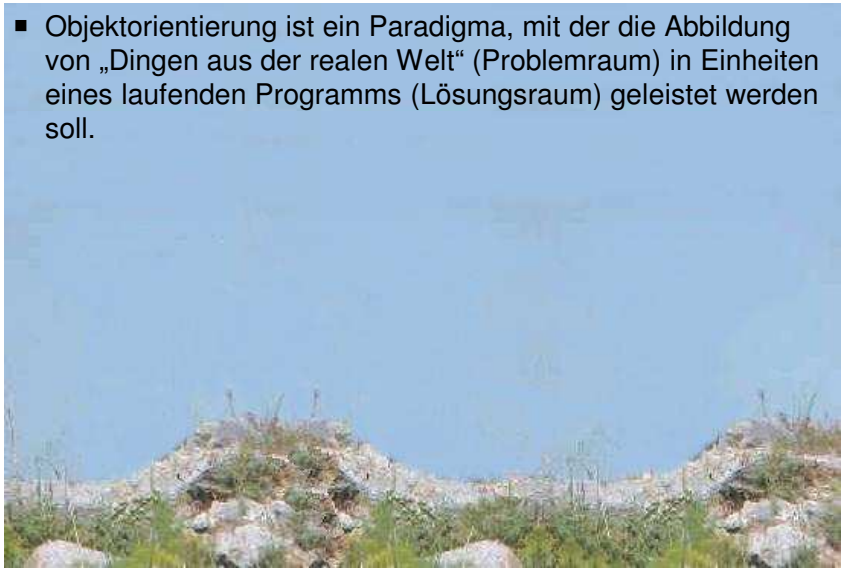


Objektmodelle

- Objektorientierte (OO) Ansätze zur Programmierung wurden zuerst 1967 vorgeschlagen (Simula-67), sind aber erst seit Ende der 80er Jahre verbreitet.
- Zur Vereinfachung der OO-Programmierung kann man OO-Methoden einsetzen. Dabei werden die Anforderungen in einem OO-Modell ausgedrückt, das System objektorientiert entworfen und anschließend in einer objektorientierten Programmiersprache implementiert.
- Objektmodelle sind daher sowohl im Analyseprozess, im Entwurfsprozess und auch im Implementierungsprozess anwendbar.
- Im Gegensatz zu den oft geäußerten Ansichten ihrer Anhänger finden Endbenutzer Objektmodelle oft schwierig zu verstehen und bevorzugen evtl. eine eher funktionale Sichtweise.

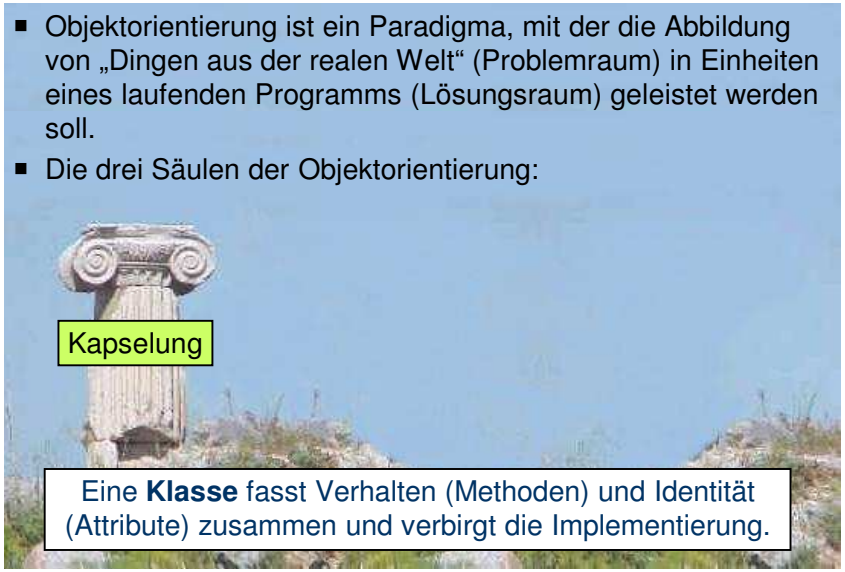
Objektorientierung

- Objektorientierung ist ein Paradigma, mit der die Abbildung von „Dingen aus der realen Welt“ (Problemraum) in Einheiten eines laufenden Programms (Lösungsraum) geleistet werden soll.



Objektorientierung

- Objektorientierung ist ein Paradigma, mit der die Abbildung von „Dingen aus der realen Welt“ (Problemraum) in Einheiten eines laufenden Programms (Lösungsraum) geleistet werden soll.
- Die drei Säulen der Objektorientierung:

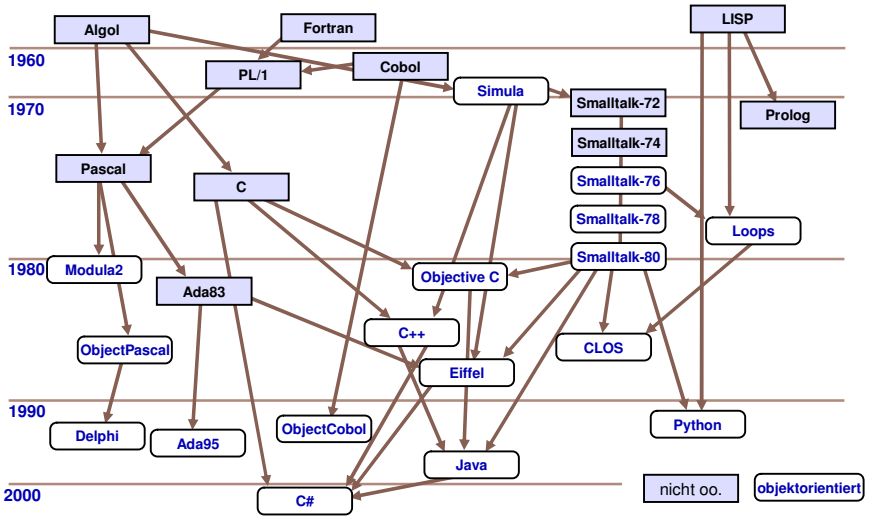


Objektorientierung

- Objektorientierung ist ein Paradigma, mit der die Abbildung von „Dingen aus der realen Welt“ (Problemraum) in Einheiten eines laufenden Programms (Lösungsraum) geleistet werden soll.
- Die drei Säulen der Objektorientierung:

Unterklassen erben Verhalten und Identität von **Oberklassen** und können diese weiter verfeinern oder verändern.

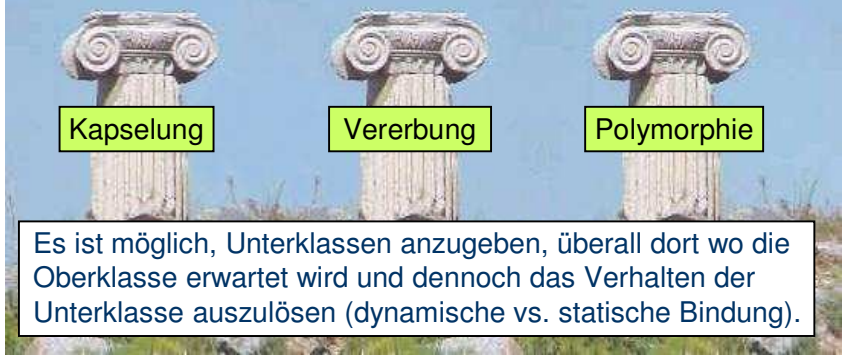
Historie der Objektorientierung



Angelehnt an: http://www.oose.de/uml

Objektorientierung

- Objektorientierung ist ein Paradigma, mit der die Abbildung von „Dingen aus der realen Welt“ (Problemraum) in Einheiten eines laufenden Programms (Lösungsraum) geleistet werden soll.
- Die drei Säulen der Objektorientierung:



Es ist möglich, Unterklassen anzugeben, überall dort wo die Oberklasse erwartet wird und dennoch das Verhalten der Unterklasse auszulösen (dynamische vs. statische Bindung).

Objektmodelle

- OO-Modelle bei der Anforderungsanalyse:
  - Zeigen Systemdaten *und* ihre Verarbeitung – und haben daher eine Kombination der Eigenschaften von Datenfluss- und semantischen Modellen,
  - zeigen, wie Teile des Systems aus anderen Teilen zusammengesetzt sind,
  - sollten keine Details der einzelnen Objekte im System zeigen,
  - sollten Gemeinsamkeiten von Objekten aus dem Problembereich modellieren.
- Es gibt unterschiedliche Modellierungsprioritäten bei Objektmodellen:
  - Zusammenhänge zwischen verschiedenen Klassen.
  - Aggregation oder Komposition von Objekten zu anderen Objekten.
  - Welches Objekt benutzt Dienste von welchem Objekt?

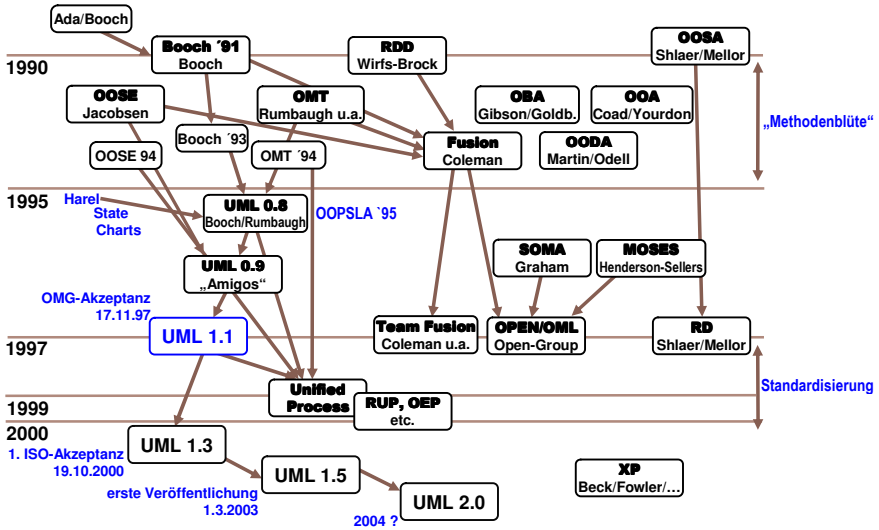
Objektmodelle

- Problembereiche:
  - In manchen Problembereichen sind Objekte leicht zu finden, z.B. „Autos“, „Flugzeuge“ oder „Bücher“.
  - Abstraktere Konzepte wie „Bücherei“ oder „Textverarbeitungsprogramm“ sind schwerer als Objektklassen zu modellieren, da sie keine Schnittstelle mit einfachen, unabhängigen Operationen haben.
  - Die Identifikation von Objekten, Klassen und vor allem deren Beziehungen untereinander gelten als einer der schwierigsten Bereiche der OO-Entwicklung.
- Ein möglicher Ansatz zur Identifikation von Objekten, Klassen und deren Beziehungen kann sich aus der Analyse von Systembeschreibungen ergeben.
- Hierbei können möglicherweise Substantive als Klassen bzw. Objekte und Verben als Operation bzw. Beziehungen angesehen werden.

Unified Modelling Language (UML)

- In der **Unified Modelling Language** kommen die OO-Methoden der selbsternannten OO-Päpste zusammen:
  - Grady Booch: Object Oriented Design (OOD), vornehmlich anwendungsorientiert,
  - James Rumbaugh: Object Modelling Technique (OMT), nahe an „strukturierten“ Methoden und
  - Ivar Jacobson: Object Oriented Software Engineering (OOSE), use cases.
- Als Gruppe nennen sich Booch, Rumbaugh und Jacobson die „drei Amigos“.
- Da diese drei Ansätze mit die populärsten auf dem Markt waren, definierte die UML von Anfang an einen Quasi-Standard.
- Als solcher wurde die UML der OMG (Object Management Group) zur Standardisierung eingereicht.
- Zur Zeit ist UML in der Version 1.5 aktuell – die Spezifikation der Version 2.0 ist für 2004 erwartet.

Historie der UML



Angelehnt an: http://www.oose.de/uml

Einordnung und Abgrenzung

- Die UML hat sich mittlerweile durchgesetzt und ist die Standardsprache zur Modellierung, da sie in nahezu allen Projektphasen eines Entwicklungsmodell zur
  - Visualisierung,
  - Spezifikation,
  - Konstruktion und
  - Dokumentation einsetzbar ist.
- UML gehört nicht zur formalen Spezifikation und gestattet daher auch keine Verifikation.
- Unterstützung für Spezialanwendungen ist in UML nicht enthalten, die UML ist jedoch zur Erweiterung vorgesehen.
- Die Erweiterungsmöglichkeiten der UML sind ebenfalls in UML spezifiziert. Dies macht die UML zu einem Metamodell.

Einordung und Abgrenzung

- UML ist eine Modellierungssprache und *keine* Methode.
- Mit UML-Diagrammen kann *unabhängig* von einer Methode oder dem Prozess kommuniziert werden.
- UML impliziert keinen bestimmten Entwicklungsprozess, eignet sich jedoch eher für iterative und objektorientierte Prozesse.
- Die UML ist weniger ein „Kochrezept“ zur OO-Entwicklung als eher eine einheitliche Notation.
- Als bevorzugten Entwicklungsprozess empfehlen die „Amigos“ (natürlich) den *Rational Unified Process*.

Rational Unified Process

- Das Prozessmodell **Rational Unified Process** (RUP) bietet eine Vielzahl praxiserprobter Details zur Entwicklung und zum Aufbau einer Software-Architektur sowie zum richtigen Einsatz der UML.
- RUP enthält eine vorgeschriebene Vorgehensweise, um Aufgaben und Verantwortungen in einer Organisation zu vergeben und versucht, Garantien für die Entwicklung von qualitativ hochwertiger Software mit prognostizierbaren Zeit- und Budgetplan zu geben.
- RUP wird durch eine webbasierte, durchsuchbare Knowledge-Base abgebildet.
- RUP ist ein kostenpflichtiges Produkt der Firma Rational, die damit ihr Software-Engineering Know-How sowie ihre Werkzeuge zur Verfügung stellt.

Einordung und Abgrenzung

- Die UML enthält eine Sammlung von Modellen, innerhalb deren man unterschiedliche Modellierungstechniken mit verschiedenen Schwerpunkten anwenden kann.
- Die Bearbeitung und Nutzung ist daher durch verschiedene Interessensgruppen möglich:
  - Analytiker und Entwickler,
  - System-Administratoren und -Integratoren,
  - Endbenutzer,
  - Tester,
  - Dokumentationsschreiber,
  - Projektleiter,
  - Manager.

Sichten und Blickwinkel

- Die umfassende Visualisierung, Spezifikation, Konstruktion und Dokumentation eines Systems erfordert verschiedene **Sichten** (*views*).
- Anforderungssicht
  - Beschreibt Anwendungsfälle.
- Strukturelle Klassifikation (statischer bzw. logischer Aufbau)
  - Beschreibt Objekte eines Systems und deren Beziehungen untereinander.
- Dynamisches Verhalten
  - Beschreibt zeitliche Abläufe zur Laufzeit des Systems.
- Alle Sichten und deren Diagrammtypen sind dank des Metamodells konsistent erweiterbar.

Anforderungssicht und strukturelle Klassifikation

- Die Anforderungssicht wird durch Anwendungsfalldiagramme unterstützt:
  - Ein **Anwendungsfalldiagramm** zeigt Akteure, Anwendungsfälle (*use cases*) und deren Beziehungen untereinander.
- Die strukturelle Klassifikation wird durch *statische* bzw. *logische* Diagramme unterstützt:
  - Ein **Klassendiagramm** zeigt Klassen mit Attributen und Operationen sowie Assoziationen zu anderen Klassen.
  - Ein **Komponentendiagramm** zeigt Komponenten (Programm-, Quell- oder Objektcode) und ihre Beziehungen (z.B. Compilierungsreihenfolge).
  - Mit Hilfe von **Paketdiagrammen** zeigt man die funktionelle Dekomposition (Zerlegung) von Systemen, Subsystemen und Modulen.
  - Ein **Einsatzdiagramm** bzw**. Verteilungsdiagramm** zeigt Komponenten und Knoten, auf denen die Komponenten laufen, sowie ihre Beziehungen.

Dynamisches Verhalten

- Die Sicht auf dynamisches Verhalten wird durch dynamische Diagramme unterstützt:
  - Ein **Zustandsdiagramm** zeigt Zustände, Zustandsübergänge und Ereignisse für ein Objekt (ähnlich einem endlichen Automat).
  - Ein **Aktivitätsdiagramm** zeigt Zustände, Zustandsübergänge, Ereignisse und Objektzustände für eine Aktivität (einen Schritt in einem „Ablauf“).
  - Ein **Sequenzdiagramm** zeigt Objekte und ihre Beziehungen beim zeitlich geordneten Nachrichtenaustausch.
  - Ein **Kollaborationsdiagramm** zeigt den Ablauf der Kommunikation zwischen voneinander abhängigen Objekten, um einen Vorgang zu erfüllen.

Stereotypen

- Die UML spezifiziert eine Anzahl von flexiblen Elementen, die sicht- und diagrammunabhängig eingesetzt werden können.
- Ein **Stereotyp** (*stereotype*) ist eine projekt-, unternehmens- oder methodenspezifische Erweiterung des UML Metamodells. Entsprechend der mit der Erweiterung definierten Semantik wird das Modellierungselement, auf das es angewendet wird, semantisch beeinflusst.
- In der Praxis geben Stereotypen vor allem die möglichen Verwendungszusammenhänge einer Klasse, einer Beziehung oder eines Paketes an.
- Graphische Notation:

```
<<Stereotyp>>
```


Eigenschaften

- Eine **Eigenschaft** (*property*) ist ein benutzerdefiniertes, sprach- und werkzeugspezifisches Schlüsselwort-Wert Paar (*tagged value*), das die Semantik eines Modellelements um eine spezielle charakteristische Eigenschaft erweitert.
- Der Unterschied zum Stereotyp besteht darin, dass durch ein Stereotyp das Metamodell um ein neues Element erweitert wird – mit Eigenschaftswerten hingegen werden einzelne Ausprägungen bestehender Modellelemente um bestimmte Eigenschaften erweitert.
- Graphische Notation:

```
{Schlüsselwort = Wert}
```

Zusicherungen und Einschränkungen

- Eine **Zusicherung** (*assertion*) ist ein Ausdruck für mögliche Inhalte und Zustände eines Modellelements, der stets erfüllt (`true`) sein muss.
- Eine **Einschränkung** (*constraint*) ist ein Ausdruck der die möglichen Inhalte, Zustände oder die Semantik eines Modellelements einschränkt.
- Zusicherungen und Einschränkungen können Stereotypen, Eigenschaftswerte oder beispielsweise auch Vor- und Nachbedingungen sein.
- Graphische Notation:

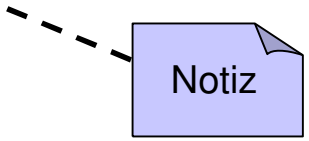
```
context Klassenname
   inv:  Invariante
   pre:  Vorbedingung
   post: Nachbedingung
```

Beispiel (vereinfacht):

```
Rechteck
     {länge > 0}
     {breite > 0}
```


Notizen

- **Notizen** bzw. **Annotationen** sind Kommentare zu einem Diagramm oder zu einem Bestandteil eines Diagramms.
- Graphische Notation:



- Anmerkung: Notizen und Kommentare werden oft bei der Erstellung von Diagrammen vernachlässigt. Obwohl die UML-Notation oft selbsterklärend ist, sind Kommentare und Annotationen *die* Hilfsmittel, um seine Entwürfe auch für andere Personen leicht verständlich zu halten!

Anwendungsfälle

- Anwendungsfälle sind meistens der Ausgangspunkt bei der Modellierung eines Systems – oft werden sie schon während der Anforderungsanalyse erstellt.
- Ein Anwendungsfall ist eine komplette, unteilbare Beschreibung – in der Praxis erstellt man zuerst ein Formular, das den einzelnen Anwendungsfall näher beschreibt.

Schablone für einen Anwendungsfall

Anwendungsfall	Bezeichnung
Kurzbeschreibung	
Beteiligte Akteuere	Kürzel, Bezeichnung
Auslöser, Vorbedingungen	Aktion oder Ereignis, das den Anwendungsfall auslöst.
Ergebnis, Nachbedingungen	Ergebnis der Interaktion

Ablauf, Interaktionen	
Nr. Akteur / System	Kurzbeschreibung der einzelnen Schritte eines erfolgreichen Ablaufs von Beginn der auslösenden Aktion oder Vorbedingung bis hin zum gewünschten Ergebnis oder der erwarteten Nachbedingung; dabei sollte immer ein Tätigkeitswort verwendet werden.

Erweiterungen, alternative Interaktionen	
Nr. Akteur / System	Bedingung, die zur Erweiterung des Hauptablaufs führt. Angabe der entsprechenden Aktion oder der Bezeichnung eines untergeordneten Anwendungsfalls.

Schablone für einen Anwendungsfall

Anwendungsfall	Beratung und Information
Kurzbeschreibung	Ein Interessent wird über das Fahrzeugangebot, die Verfügbarkeit und den Mietpreise für ein gewähltes Mietdatum informiert.
Beteiligte Akteure	Kb Kundenberaterin
Auslöser, Vorbedingungen	Telefonische oder persönliche Anfrage eines Interessenten
Ergebnis, Nachbedingungen	Dem Interessenten wird die Preisinformation für die gewählte Fahrzeugkategorie und Mietdauer übermittelt.

Ablauf, Interaktionen		
1		**Erfassung der Interessentenwünsche**
1.1	Kb	Die gewünschte Fahrzeugkategorie und der Abholort werden per Auswahlmenü festgelegt, das Alter des Fahrers und die Mietdaten werden in der Form TT.MM.JJ für das Datum der Abholung und das Datum der Rückgabe eingegeben.
1.2	System	Das System prüft die Plausibilität der Daten; wenn Daten in Ordnung, listet das System den Mietpreis sowie die Aufschläge für Verlängerungstage und -stunden auf.
1.3	System	**Ausnahme:** Datum der Rückgabe liegt später als Datum der Abholung.
1.4	Kb	Gibt neue Mietdaten ein

Schablone für einen Anwendungsfall

		listet das System den Mietpreis sowie die Aufschläge für Verlängerungstage und -stunden auf.
1.3	System	**Ausnahme:** Datum der Rückgabe liegt später als Datum der Abholung.
1.4	Kb	Gibt neue Mietdaten ein
1.5	System	wie 1.2
1.6	System	**Ausnahme:** Kein Fahrzeug am gewählten Abholort zum am gewünschten Zeitpunkt verfügbar .
1.7	System	Gibt neue Daten ein oder bricht Beratung auf Wunsch des Kunden ab.

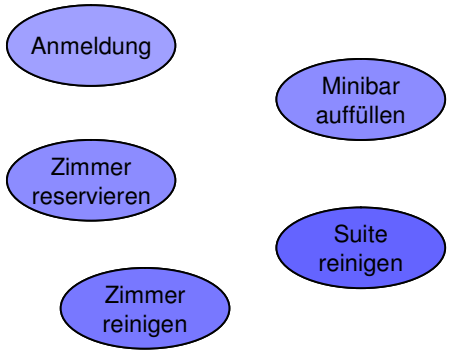
Erweiterungen, alternative Interaktionen		
2		**Interessent weist sich per Kundennummer als Firmenkunde aus**
2.1	Kb	Die Kundennummer wird eingegeben.
2.2	System	System überprüft Gültigkeit der Kundennummer und bestätigt oder verwirft die eingegebene Kundennummer.
2.3	Kb	Falls Kundennummer vom System bestätigt wurde, gibt Beraterin die Daten wie in 1.1 ein.
2.4	System	Das System listet den Mietpreis sowie die Aufschläge für Verlängerungstage und -stunden auf, falls Datenüberprüfung bestätigt; die angegebenen Preise enthalten 20% Rabatt für Firmenkunden.
2.5	System	weiter wie bei 1.3

Anwendungsfalldiagramm

Anwendungsfalldiagramm einer Hotelbuchung:

Anwendungsfalldiagramm

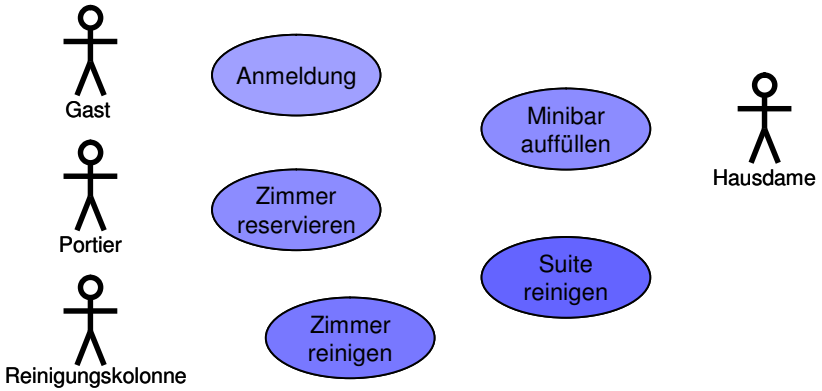
Anwendungsfalldiagramm einer Hotelbuchung:



- Ein **Anwendungsfall** beschreibt eine Menge von Aktivitäten eines Systems und damit ein bestimmtes Verhalten des zu erstellenden Systems.
- Ein Anwendungsfall modelliert nur die extern sichtbare Funktionalität.

Anwendungsfalldiagramm

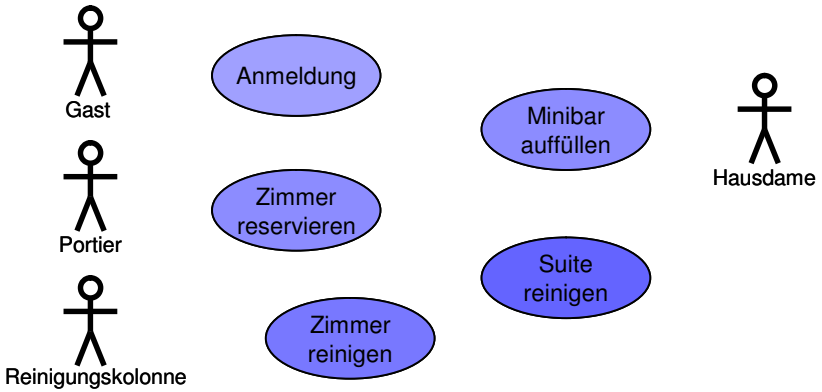
Anwendungsfalldiagramm einer Hotelbuchung:



Akteure sind außerhalb des zu realisierenden Systems liegende Einheiten. Sie interagieren mit dem System, z.B. durch die Benutzung von Diensten des Systems, oder werden vom System „benutzt“. Als Akteur kann auch ein anderes System auftreten, z.B. SAP oder ein Betriebssystem.

Anwendungsfalldiagramm

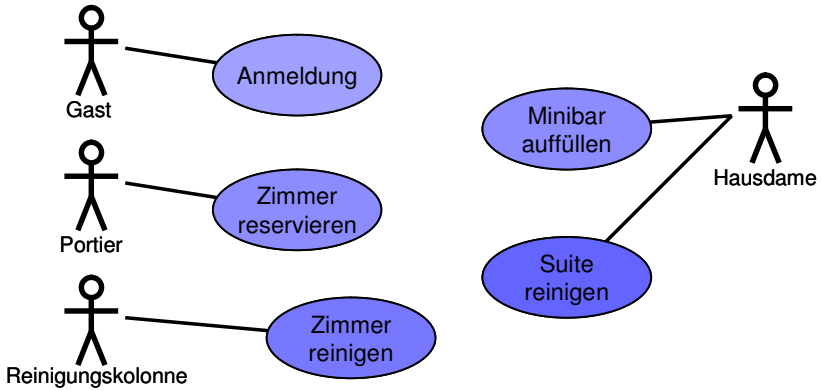
Anwendungsfalldiagramm einer Hotelbuchung:



Ein **Anwendungsfalldiagramm** beschreibt die Zusammenhänge zwischen verschiedenen Anwendungsfällen untereinander sowie zwischen Anwendungsfällen und den beteiligten Akteuren.

Anwendungsfalldiagramm

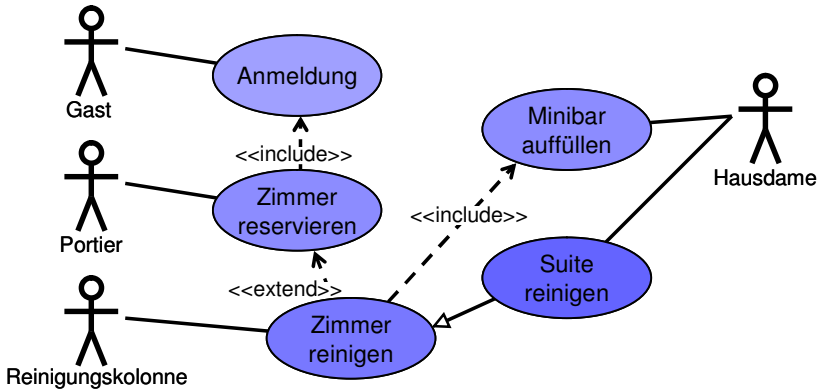
Anwendungsfalldiagramm einer Hotelbuchung:



Jeder Akteur hat mindestens eine Beziehung zu einem Anwendungsfall – ein Anwendungsfall wird stets durch einen Akteur initiiert.

Anwendungsfalldiagramm

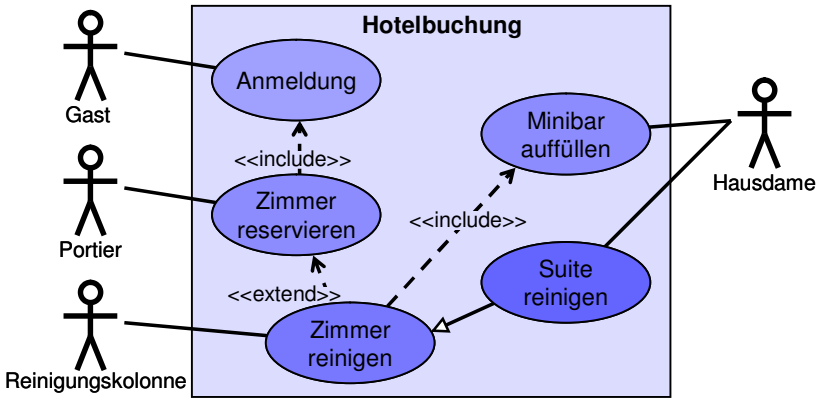
Anwendungsfalldiagramm einer Hotelbuchung:



Anwendungsfälle können nicht nur zu Akteuren, sondern auch untereinander in Beziehung stehen, z.B. durch Enthaltensein (**`include`**) oder Erweiterung (**`extend`**).

Anwendungsfalldiagramm

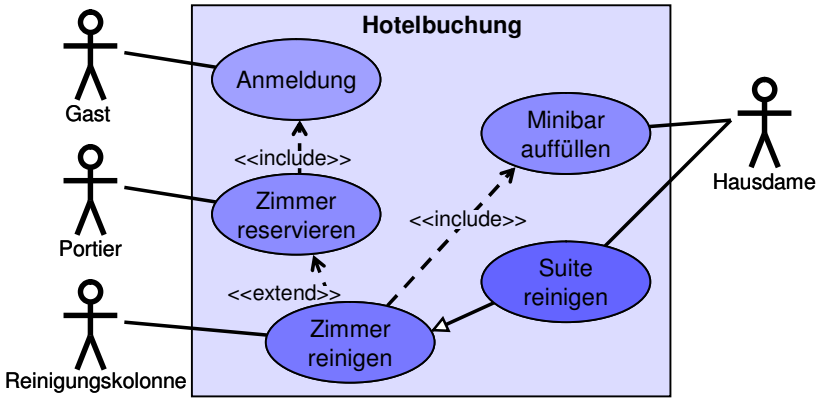
Anwendungsfalldiagramm einer Hotelbuchung:



Zusammenhängende Anwendungsfälle können durch eine benannte Systemgrenze visualisiert werden.

Anwendungsfalldiagramm

Anwendungsfalldiagramm einer Hotelbuchung:



Klassendiagramme

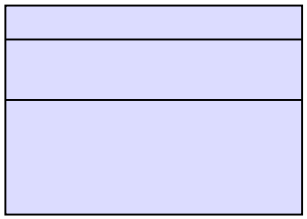
- Das **Klassendiagramm** ist *das* zentrale Modellierungsdiagramm beim objektorientiertem Design und der UML.
- Mit dem Klassendiagramm können Konzepte des Anwendungsbereiches sowie interne Konzepte visualisiert werden.
- Abhängig von dem gewählten Blickwinkel und der Modellierungspriorität gibt es mehrere mögliche Sichtweisen beim Entwurf eines Klassendiagramms, z.B.
  - Konzeptionell, d.h. ohne eine direkte Abbildungen zur Implementierung.
  - Spezifizierend, d.h. mit der Angabe verpflichtender Schnittstellen.
  - Implementierend, d.h. sehr detailliert und mit der Angabe von Konsequenzen für die Implementierung.

Objekte, Klassen und Nachrichten

- **Objekte** sind die agierenden Grundelemente einer Anwendung.
- Eine **Klasse** ist eine Zusammenfassung gleichartiger Objekte.
- Die Gleichartigkeit bezieht sich auf die Eigenschaften und das Verhalten der Objekte einer Klasse.
- Eine Klasse enthält gewissermaßen die Konstruktions-beschreibung für die Objekte, die mit ihr erzeugt werden.
- Klassen sind Elemente zur Entwicklungszeit, während Objekte Elemente zur Laufzeit eines Programms sind (**Instanzen**).
- Das Verhalten der Objekte wird durch die Möglichkeit eines Objektes **Nachrichten** zu empfangen und zu verstehen beschrieben.
- Eine Nachricht besteht aus dem Selektor (ein Name) und einer Liste von Parametern. Sie wird an genau einen Empfänger gesendet.

UML Klassendiagramm

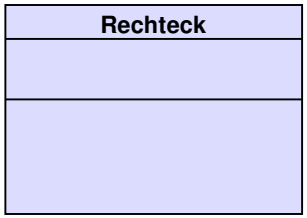
Beispiel: Die Klasse **Rechteck** in UML:



- In UML wird eine Klasse durch mindestens ein und bis zu drei Beschreibungsteile dargestellt.

UML Klassendiagramm

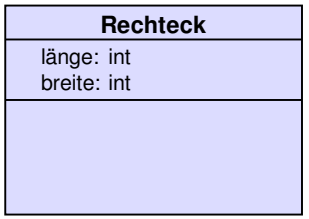
Beispiel: Die Klasse **Rechteck** in UML:



- In UML wird eine Klasse durch mindestens ein und bis zu drei Beschreibungsteile dargestellt.
- Der **Name** beschreibt eine Klasse in ihrer grundsätzlichen Funktion bzw. als Abbildung einer Entität aus der „wirklichen Welt“.

UML Klassendiagramm

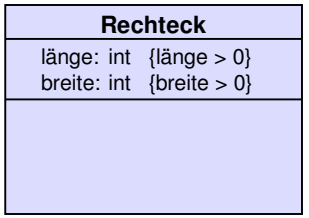
Beispiel: Die Klasse **Rechteck** in UML:



Attribute sind Informationen bzw. Daten, die ein Element einer Klasse näher beschreiben. Sie werden mindestens durch ihren Namen beschrieben, können aber zusätzlich einen Typen, einen Initialwert sowie Zusicherungen und Merkmale besitzen.

UML Klassendiagramm

Beispiel: Die Klasse **Rechteck** in UML:



Attribute sind Informationen bzw. Daten, die ein Element einer Klasse näher beschreiben. Sie werden mindestens durch ihren Namen beschrieben, können aber zusätzlich einen Typen, einen Initialwert sowie Zusicherungen und Merkmale besitzen.

- Durch **Zusicherungen** kann der Wertebereich bzw. die Wertemenge eines Attributes eingeschränkt werden.
- Mit Hilfe von **Merkmalen** können weitere besondere Eigenschaften von Attributen beschrieben werden, z.B. dass ein Attribut nur gelesen werden darf.

UML Klassendiagramm

Beispiel: Die Klasse **`Rechteck`** in UML:

Rechteck
länge: int {länge > 0} breite: int {breite > 0}
setzeLänge(x): int setzeBreite(y): int gibLänge(): int gibBreite(): int

- Damit ein Objekt Nachrichten empfangen kann, benötigt es entsprechende **Operationen**.
- Eine Operation setzt sich zusammen aus dem Namen der Operation, den Parametern (falls vorhanden) und eventuell einem Rückgabewert. Die Parameter einer Operation entsprechen in ihrer Definition den Attributen.

UML Klassendiagramm

Beispiel: Die Klasse **`Rechteck`** in UML:

Rechteck
länge: int {länge > 0} breite: int {breite > 0}
setzeLänge(x): int {x > 0} setzeBreite(y): int {y > 0} gibLänge(): int gibBreite(): int

- Eine Operation ist innerhalb einer Klassendefinition eindeutig identifizierbar. Operationen können mit Zusicherungen versehen werden, um bestimmte Bedingungen einer Operation beim Aufruf zu sichern.
- Mit Hilfe von Merkmalen können bestimmte Eigenschaften einer Operation beschrieben werden.

UML Klassendiagramm

Beispiel: Die Klasse **`Rechteck`** in UML:

Rechteck
länge: int {länge > 0} breite: int {breite > 0}
setzeLänge(x): int {x > 0} setzeBreite(y): int {y > 0} gibLänge(): int gibBreite(): int

Die Begriffe Operation und Nachricht werden oft synonym verwendet, was allerdings nicht richtig ist. Objekte kommunizieren untereinander mit Hilfe von Nachrichten. Ein Objekt kann aber nur eine solche Nachricht verstehen, zu der es eine entsprechende Operation gibt.

UML Klassendiagramm

Beispiel: Die Klasse **`Rechteck`** in UML:

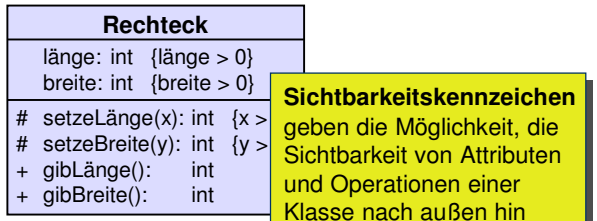
Rechteck
länge: int {länge > 0} breite: int {breite > 0}
setzeLänge(x): int {x > # setzeBreite(y): int {y > + gibLänge(): int + gibBreite(): int

Sichtbarkeitskennzeichen geben die Möglichkeit, die Sichtbarkeit von Attributen und Operationen einer Klasse nach außen hin einzuschränken.

Symbol	Name	Beschreibung
+	(**`public`**):	verfügbar für alle Klassen des Systems.
~	(**`package`**):	verfügbar für alle Klassen eines Paketes des Systems.
#	(**`protected`**):	verfügbar für Objekte der eigenen und aller abgeleiteten Klassen.
-	(**`private`**):	verfügbar nur für Objekte genau dieser Klasse.

UML Klassendiagramm

Beispiel: Die Klasse **`Rechteck`** in UML:



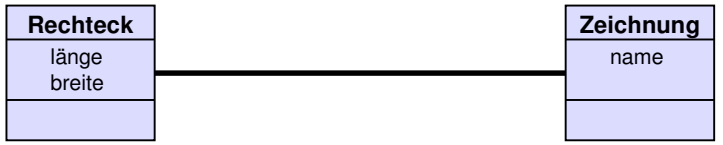
Sichtbarkeitskennzeichen geben die Möglichkeit, die Sichtbarkeit von Attributen und Operationen einer Klasse nach außen hin einzuschränken.

Symbol	Name	Beschreibung
+	(**`public`**):	verfügbar für alle Klassen des Systems.
~	(**`package`**):	verfügbar für alle Klassen eines Paketes des
#		
-	(**`private`**):	verfügbar nur für Objekte genau dieser Klasse.

Bemerkung: Auch Sichtbarkeitskennzeichen sind zunächst nur konzeptionelle Kennzeichnen – je nach Implementierungssprache werden sie u.U. unterschiedlich umgesetzt bzw. garantiert.

Assoziationen

Assoziation zwischen einem Rechteck und einer Zeichnung:

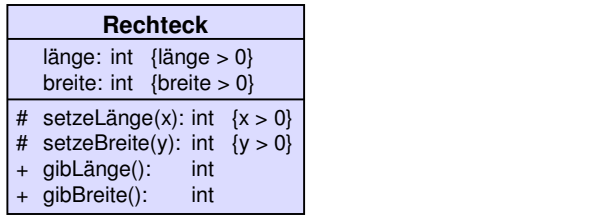


- In einem Klassendiagramm werden ebenfalls **Assoziationen** zwischen Klassen dargestellt.
- Eine Assoziation entspricht dabei im einfachsten Fall nur der Aussage, dass zwei Klassen „etwas miteinander zu tun haben“, d.h. also in einer gewissen Abhängigkeit zueinander stehen.

Vorlesung Softwaretechnik (SS 2003)

UML Klassendiagramm

Beispiel: Die Klasse **`Rechteck`** in UML:

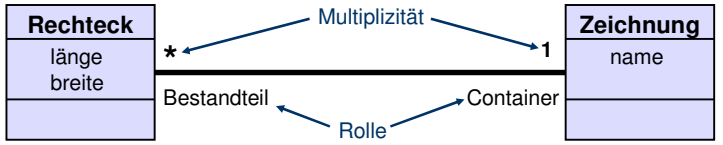


Symbol	Name	Beschreibung
+	(**`public`**):	verfügbar für alle Klassen des Systems.
~	(**`package`**):	verfügbar für alle Klassen eines Paketes des Systems.
#	(**`protected`**):	verfügbar für Objekte der eigenen und aller abgeleiteten Klassen.
-	(**`private`**):	verfügbar nur für Objekte genau dieser Klasse.

Vorlesung Softwaretechnik (SS 2003)

Assoziationen

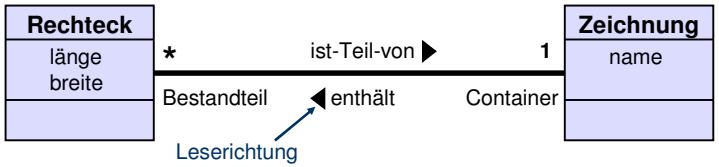
Assoziation zwischen einem Rechteck und einer Zeichnung:



- Assoziationsenden haben eine **Rolle** und eine **Multiplizität**: Vielfachheiten können aufgezählt werden (**`0,1`**) oder als Bereiche angegeben (**`0…5`**). Ein Stern (**`*`**) steht dabei für eine beliebige Vielfachheit (auch Null).

Assoziationen

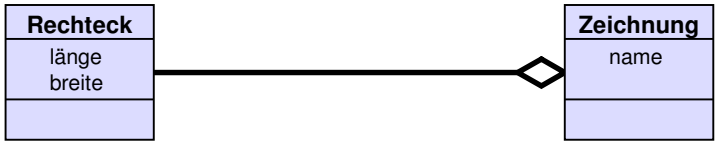
Assoziation zwischen einem Rechteck und einer Zeichnung:



- Assoziationsenden haben eine **Rolle** und eine **Multiplizität**: Vielfachheiten können aufgezählt werden (`0,1`) oder als Bereiche angegeben (`0…5`). Ein Stern (`*`) steht dabei für eine beliebige Vielfachheit (auch Null).
- Assoziationen können eine beschriftete Navigierbarkeit enthalten.
- Assoziationen können ebenfalls Eigenschaften sowie Einschränkungen und Zusicherungen enthalten.

Aggregation

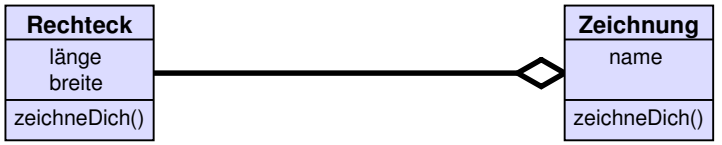
Aggregation von Rechtecken zu einer Zeichnung:



- Für die wichtigsten Assoziationen gibt es spezielle Notationen und eine weitergehende Semantik.
- Eine **Aggregation** ist die Zusammensetzung eines Objektes aus einer Menge von Einzelteilen.

Aggregation

Aggregation von Rechtecken zu einer Zeichnung:



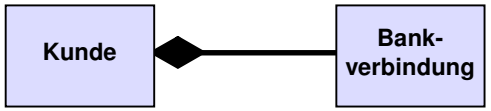
- In der Aggregation nimmt das Ganze stellvertretend für seine Teile Aufgaben wahr.
- Die Aggregatklasse kann Operationen enthalten, die keine unmittelbare Wirkung im Aggregat selbst erzeugen, sondern die entsprechenden Nachrichten an seine Teile weiterleiten (Delegation).

Aggregation

- Die beteiligten Klassen führen keine gleichberechtigte Beziehung. Statt dessen bekommt die Aggregatklasse eine verantwortliche und führende Rolle.
- In einer Aggregationsbeziehung muss an einem Ende das Aggregat stehen und am anderen Ende die zugehörigen Teile. Würde auf keiner Seite ein Aggregat stehen, wäre dies (nur) eine Assoziation. Steht auf beiden Seiten ein Aggregat, wäre das ein Widerspruch zur obigen Definition.
- Ein Teil kann dabei durchaus auch zu mehreren Aggregationen gehören.

Aggregation und Komposition

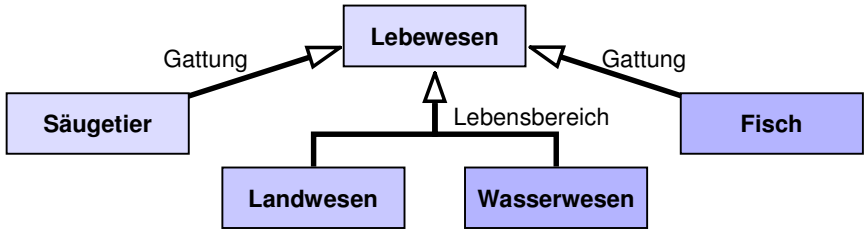
- Es gibt auch Fälle in denen die Teile existenzabhängig vom Aggregat sind. Wenn also das Aggregat nicht mehr existiert, dann werden auch die Teile gelöscht. Wird aber ein Einzelteil gelöscht, so bleibt das Aggregat erhalten.
- Diese Form der Aggregation nennt man **Komposition**:



- Ein Kunde ohne Bankverbindung ist denkbar, eine Bankverbindung ohne zugehörigen Eigentümer jedoch nicht!

Generalisierung und Spezialisierung

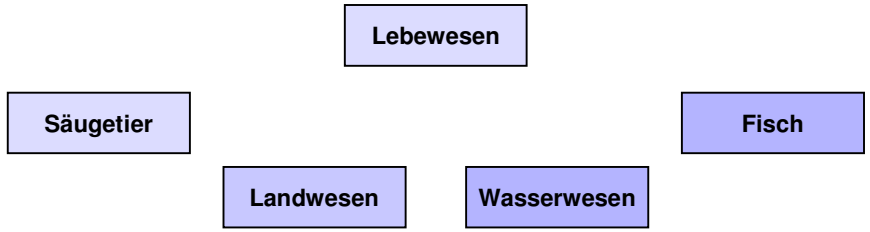
Klassendiagramm von Lebewesen:



Bei der **Generalisierung** bzw. **Spezialisierung** werden Eigenschaften hierarchisch gegliedert, d.h. Eigenschaften mit allgemeinerer Bedeutung werden **Oberklassen** zugeordnet und speziellere Eigenschaften werden **Unterklassen** zugeordnet.

Generalisierung und Spezialisierung

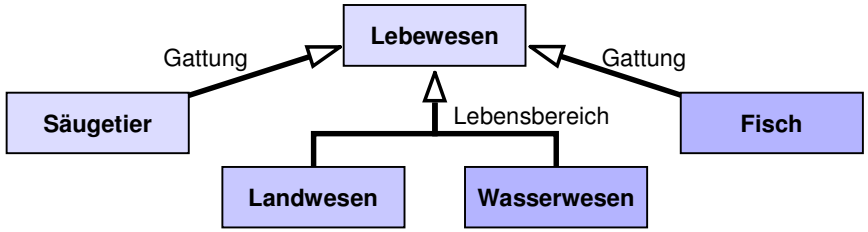
Klassendiagramm von Lebewesen:



- Die Unterscheidung in Ober- und Unterklasse erfolgt mit Hilfe eines Unterscheidungsmerkmals, dem sogenannten **Diskriminator**. Dieser definiert den für die Strukturierung maßgeblichen Aspekt.
- Er ist nicht von selbst gegeben, sondern das Ergebnis einer Modellierungsentscheidung.
- Beispielsweise könnte man Lebewesen aufgrund des Diskriminators *Gattung* gliedern oder aber aufgrund des bevorzugten *Lebensbereichs*.

Generalisierung und Spezialisierung

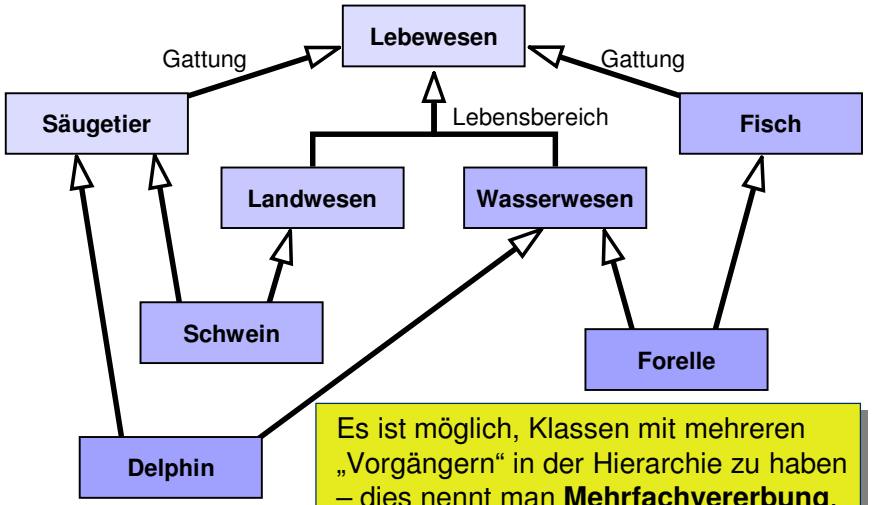
Klassendiagramm von Lebewesen:



- Eigenschaften der Oberklasse werden dabei über den Mechanismus der **Vererbung** an die zugehörige Unterklasse weitergegeben.
- Eine Unterklasse verfügt folglich über ihre speziellen Eigenschaften und über die Eigenschaften ihrer Oberklasse.

Generalisierung und Spezialisierung

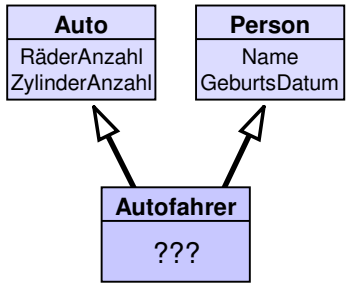
Klassendiagramm von Lebewesen:



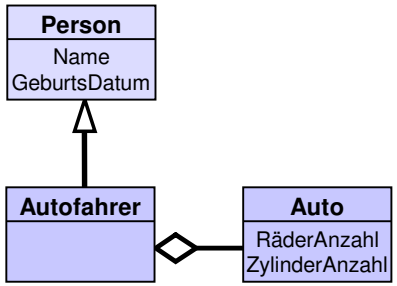
Es ist möglich, Klassen mit mehreren „Vorgängern“ in der Hierarchie zu haben – dies nennt man **Mehrfachvererbung**.

Mehrfachvererbung

- Mehrfachvererbung ist ein sehr mächtiges Strukturierungselement, dessen Benutzung jedoch nicht immer angebracht ist.
- **Häufiger Entwurfsfehler:** Aggregation („hat-ein“) wird als Vererbung modelliert („ist-ein“):



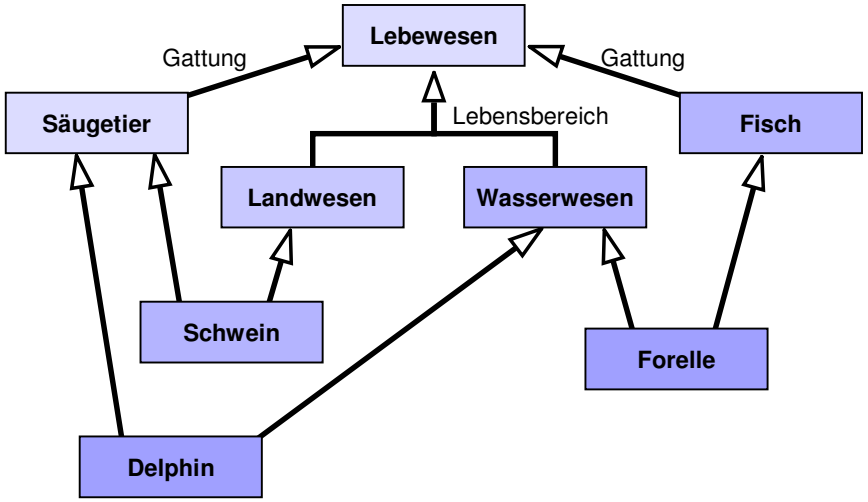
Schlecht: Abstraktion



Besser: Aggregation

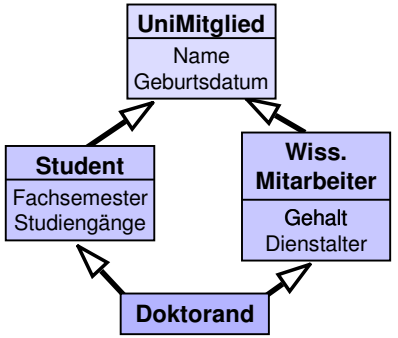
Generalisierung und Spezialisierung

Klassendiagramm von Lebewesen:

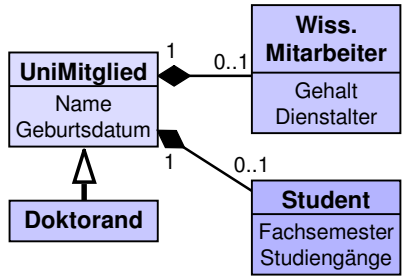


Mehrfachvererbung

- **Häufiger Entwurfsfehler:** Verschiedene Rollen werden durch Vererbung modelliert.
- Dies führt unter anderem dazu, dass Attribute oder Implementierungen mehrfach von der gleichen Oberklasse ererbt werden können (Diamantgraph):



Schlecht: Abstraktion



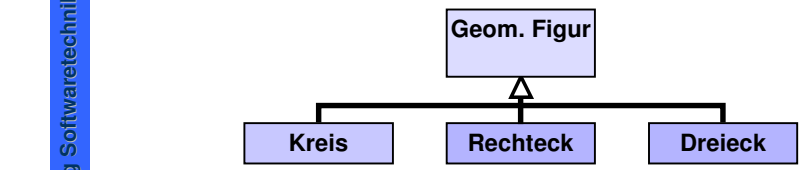
Besser: Komposition mit Implementierten Rollen

Mehrfachvererbung

- Weitere Probleme, die entstehen können:
  - Namenskonflikte zwischen Vorgängerklassen, d.h. verschiedene ererbte Komponenten können denselben Namen tragen.
  - Es können gleiche Methodenimplementierungen mit unterschiedlicher Semantik ererbt werden.
- Auch seitens der Übersetzer oder Interpreter ist die Unterstützung von Mehrfachvererbung schwierig. Dies ist einer der Gründe, warum in vielen Sprachen auf dieses Merkmal verzichtet wird.

Abstrakte Klassen

- Eine **abstrakte Klasse** bildet ausschließlich die Grundlage für Unterklassen. Sie wird bewusst unvollständig gehalten und von ihr selbst können zur Laufzeit keine konkreten Objekte erzeugt werden.
- Eine abstrakte Klasse stellt häufig einen Allgemeinbegriff oder Oberbegriff dar, von dem konkrete Begriffe abgeleitet werden.
- **Beispiel:** Der Oberbegriff der *geometrische Figur:*



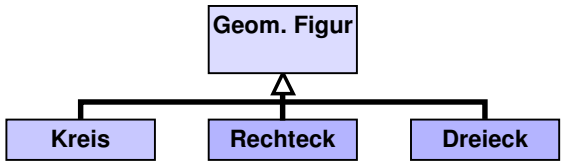
Abstrakte Klassen

- Eine **abstrakte Klasse** bildet ausschließlich die Grundlage für Unterklassen. Sie wird bewusst unvollständig gehalten und von ihr selbst können zur Laufzeit keine konkreten Objekte erzeugt werden.
- Eine abstrakte Klasse stellt häufig einen Allgemeinbegriff oder Oberbegriff dar, von dem konkrete Begriffe abgeleitet werden.
- **Beispiel:** Der Oberbegriff der *geometrische Figur:*

Von der geometrischen Figur kann man die konkreten Begriffe *Kreis*, *Rechteck*, *Dreieck*, etc. ableiten.

Abstrakte Klassen

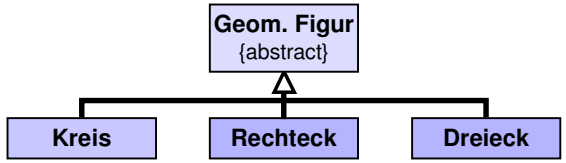
- Eine **abstrakte Klasse** bildet ausschließlich die Grundlage für Unterklassen. Sie wird bewusst unvollständig gehalten und von ihr selbst können zur Laufzeit keine konkreten Objekte erzeugt werden.
- Eine abstrakte Klasse stellt häufig einen Allgemeinbegriff oder Oberbegriff dar, von dem konkrete Begriffe abgeleitet werden.
- **Beispiel:** Der Oberbegriff der *geometrische Figur:*



- Von jedem dieser konkreten Begriffe können Exemplare erzeugt werden, z.B. ein Kreis `Kreis1` mit dem Durchmesser von 12 cm.
- Von einer geometrischen Figur selbst können jedoch keine konkreten Exemplare erstellt werden.

Abstrakte Klassen

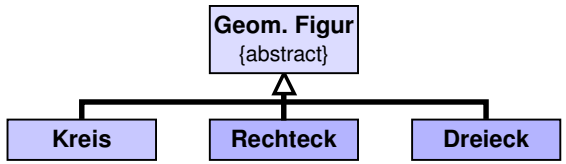
- Eine **abstrakte Klasse** bildet ausschließlich die Grundlage für Unterklassen. Sie wird bewusst unvollständig gehalten und von ihr selbst können zur Laufzeit keine konkreten Objekte erzeugt werden.
- Eine abstrakte Klasse stellt häufig einen Allgemeinbegriff oder Oberbegriff dar, von dem konkrete Begriffe abgeleitet werden.
- **Beispiel:** Der Oberbegriff der *geometrische Figur:*



Eine abstrakte Klasse wird wie eine normale Klasse dargestellt. Unter dem Klassennamen steht das Merkmal **`abstract`**. Sie kann wie eine normale Klasse Attribute, Operationen und Zusicherungen enthalten.

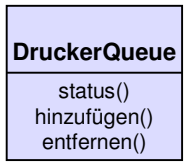
Abstrakte Klassen

- Eine **abstrakte Klasse** bildet ausschließlich die Grundlage für Unterklassen. Sie wird bewusst unvollständig gehalten und von ihr selbst können zur Laufzeit keine konkreten Objekte erzeugt werden.
- Eine abstrakte Klasse stellt häufig einen Allgemeinbegriff oder Oberbegriff dar, von dem konkrete Begriffe abgeleitet werden.
- **Beispiel:** Der Oberbegriff der *geometrische Figur:*



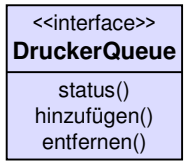
Schnittstellen

- **Schnittstellen** spezifizieren das externe Verhalten von Klassen und enthalten in abstrakter Form Signaturen und Beschreibungen von Operationen.



Schnittstellen

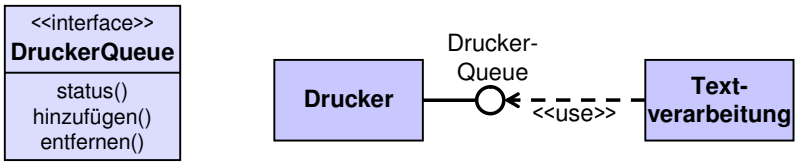
- **Schnittstellen** spezifizieren das externe Verhalten von Klassen und enthalten in abstrakter Form Signaturen und Beschreibungen von Operationen.



- Schnittstellen werden als abstrakte Klassen mit dem Stereotyp **`interface`** modelliert**.**

Schnittstellen

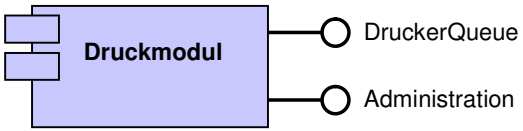
- **Schnittstellen** spezifizieren das externe Verhalten von Klassen und enthalten in abstrakter Form Signaturen und Beschreibungen von Operationen.



- Schnittstellen werden als abstrakte Klassen mit dem Stereotyp **`interface`** modelliert**.**
- Klassen, die alle von einer Schnittstellenklasse geforderten Operationen bereitstellen können, sind eine Umsetzung (Implementierung) dieser Schnittstelle.
- Eine Klasse kann Schnittstellen anbieten, die von anderen Klassen benutzt (**`use`**) werden können.

Komponentendiagramm

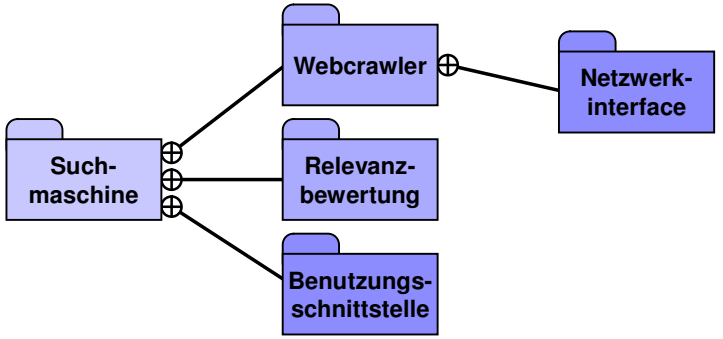
- Damit bei späterer Implementierung der Softwarelösung Compiler- und Laufzeitabhängigkeiten klar sind, können die Zusammenhänge der einzelnen Bestandteile in einem **Komponentendiagramm** dargestellt.



- Eine **Komponente** stellt ein physisches Stück Programmcode dar, welches entweder ein Stück Quellcode, Binärcode oder ein ausführbares Teilprogramm der gesamten Softwarelösung sein kann.
- Eine Komponente repräsentiert ein funktional in sich geschlossenes Modul, welches wiederum weitere Elemente wie Objekte, Komponenten oder Knoten enthalten und Schnittstellen besitzen kann.

Paketdiagramm

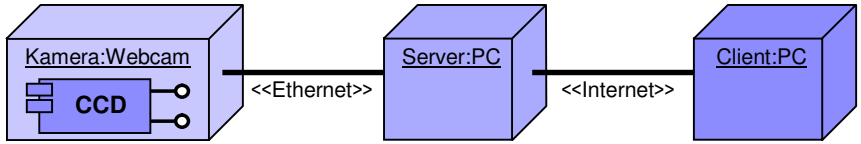
- Ein **Paketdiagrammen** zeigt die funktionelle Dekomposition (Zerlegung) von Systemen in Subsysteme und Module:



- Weiterhin können in Paketdiagrammen Modellelemente aus Komponenten- und Klassendiagrammen verwendet werden.

Einsatz- bzw. Verteilungsdiagramm

- Ein **Einsatzdiagramm** (oft auch **Verteilungsdiagramm** genannt) beschreibt, welche Komponenten auf welchen Knoten ablaufen, wie diese konfiguriert sind und welche Abhängigkeiten bestehen.



- Ein **Knoten** ist ein physisch vorhandenes Objekt, das über Ressourcen wie z.B. Rechenleistung oder Speicher verfügt – es kann sich aber auch um einen Sensor oder ein Messgerät handeln.
- In den Knoten können die dort ablaufenden Komponenten dargestellt werden, wobei auch Schnittstellen und Abhängigkeitsbeziehungen zwischen den Elementen erlaubt sind.
- Kommunizierende Knoten werden durch Assoziationen visualisiert – diese entsprechen in der Praxis meist physikalischen Verbindungen.

Zustandsdiagramm

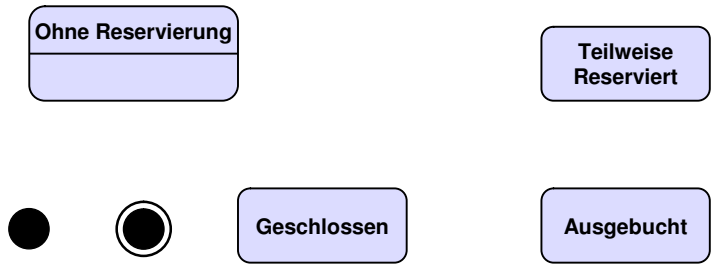
- Ein Objekt kann in seinem Leben verschiedenartige Zustände annehmen. Ein **Zustand** ist die endliche nichtleere Menge von möglichen Attributwerten, die die Objekte einer Klasse annehmen können. Er gehört zu genau einer Klasse.
- Eine Klasse muss nicht notwendigerweise Zustände beinhalten, sie muss über entsprechendes signifikantes Verhalten verfügen. Die Modellierung von Zuständen erübrigt sich, sobald alle Operationen eines Objektes einer Klasse unabhängig von seinem inneren Zustand in beliebiger Reihenfolge durchgeführt werden können.

Zustandsdiagramm

- Ein Zustand wird als Zeitspanne zwischen zwei Ereignissen angesehen. Eine Änderung der Attributwerte eines Objektes, die das Verhalten des Objektes maßgeblich verändern, heißt **Zustandsänderung**.
- Zustände sind durch einen für sie eindeutigen Namen definiert. Gleichnamige Zustände innerhalb eines **Zustandsdiagramms** beschreiben den selben Zustand eines Objektes.
- Zusätzlich können anonyme Zustände definiert werden. Im Unterschied zu benannten Zuständen beschreiben mehrere anonyme Zustände grundsätzlich verschiedene Zustände eines Objektes.

Zustandsdiagramm

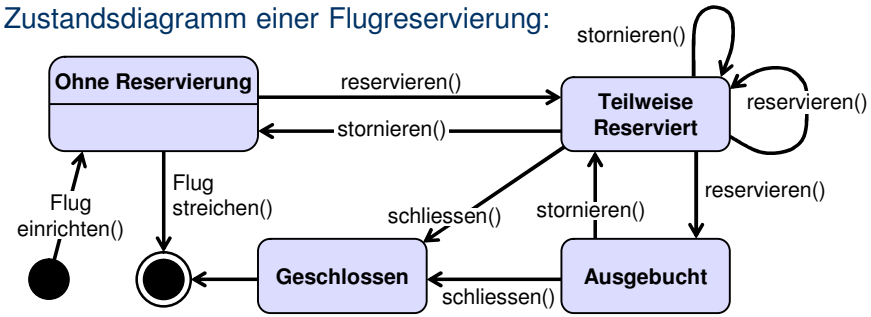
Zustandsdiagramm einer Flugreservierung:



Ein **Zustandsdiagramm** beschreibt eine hypothetische Maschine, die sich zu jedem Zeitpunkt in einem von einer endlichen Menge möglicher Zustände befindet.

Zustandsdiagramm

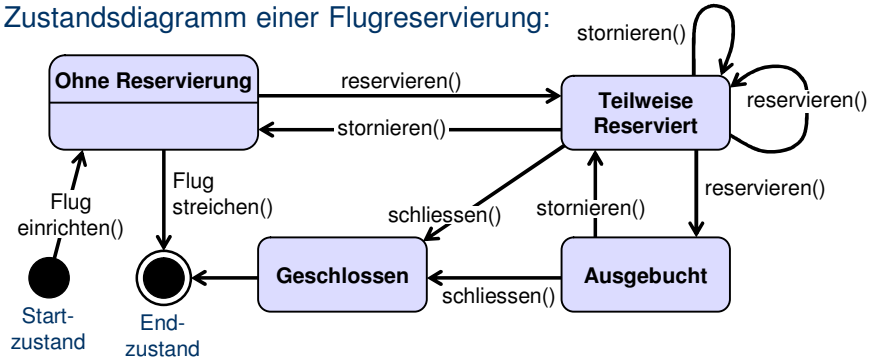
Zustandsdiagramm einer Flugreservierung:



Ein **Ereignis** löst den Übergang (Transition) von einem Zustand eines Objektes in den nächsten Zustand aus. Es ist definiert durch einen Namen und einer Liste möglicher Ereignisargumente.

Zustandsdiagramm

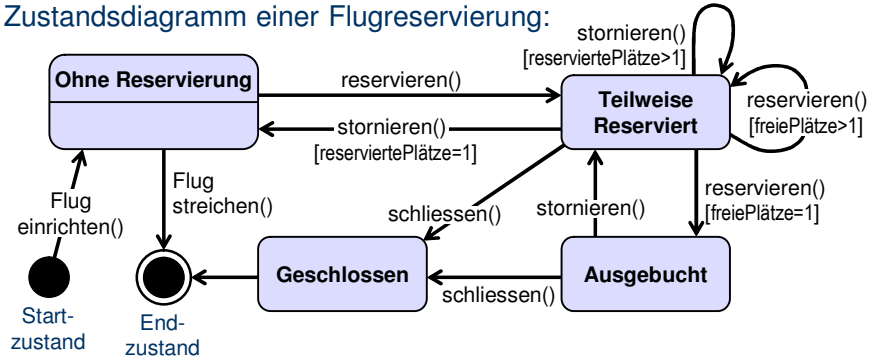
Zustandsdiagramm einer Flugreservierung:



Start- und Endzustand eines Objektes sind als besondere Zustandstypen anzusehen, da zu einem Startzustand kein Übergang stattfinden und dem Endzustand eines Objektes keine Zustandsänderung folgen kann.

Zustandsdiagramm

Zustandsdiagramm einer Flugreservierung:



Ein Zustand kann Bedingungen mit dem Ereignis verbinden, die erfüllt sein müssen, um den Folgezustand zu erreichen, bzw. um zu entscheiden, welchen Folgezustand das Objekt einnimmt.

Zustandsdiagramm

Zustandsdiagramm einer Flugreservierung:

Durch Ereignisse können Aktionen innerhalb eines Zustandes eines Objektes ausgelöst werden. Auslöser solcher Aktionen sind:

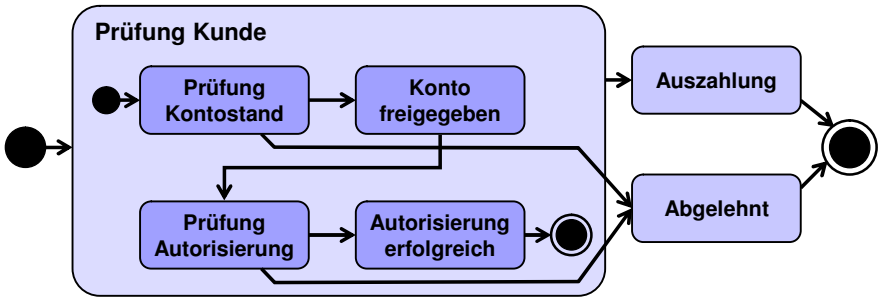
- entry - löst eine Aktion automatisch bei Erreichen eines Zustandes aus.
- exit - löst eine Aktion automatisch bei Verlassen eines Zustandes aus.
- do - löst eine Aktion automatisch aus, solange der Zustand aktiv ist, d.h. kein anderer Zustand durch das Objekt erreicht wird.

Zustandsdiagramm

Zustandsdiagramm einer Flugreservierung:

Unterzustände und Nebenläufigkeit

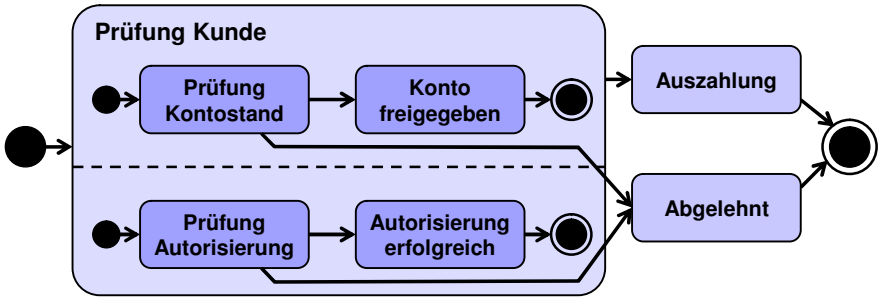
Zustandsdiagramm für einen Geldautomaten:



- Ist es nötig, Ereignisse innerhalb eines Zustandes eines Objektes näher zu untersuchen, lässt sich dieser durch die Definition von Unterzuständen aufgliedern.
- Die Unterzustände werden in die eigentlichen Zustände eingebettet. Die Notation von Unterzuständen ist gleich denen von Zuständen.

Unterzustände und Nebenläufigkeit

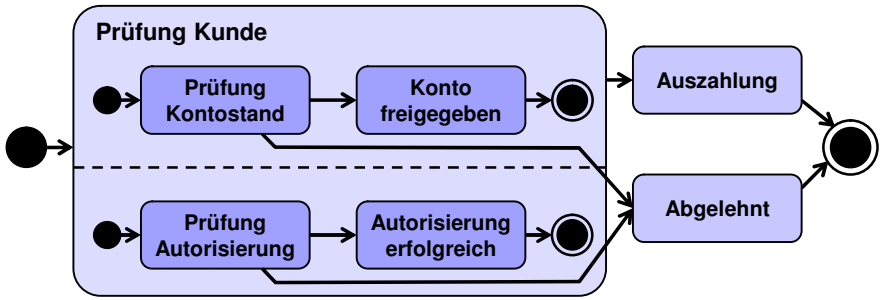
Zustandsdiagramm für einen Geldautomaten:



Nebenläufige Zustandsdiagramme erlauben die Darstellung von (zeitweilig) unabhängigem Verhalten, indem parallele bzw. konkurrierende Unterzustände durch gestrichelte Linien in verschiedene Abschnitte unterteilt werden.

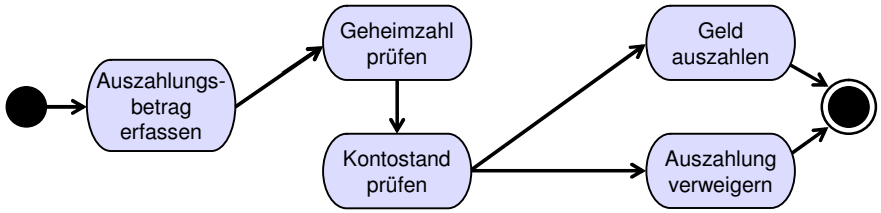
Unterzustände und Nebenläufigkeit

Zustandsdiagramm für einen Geldautomaten:



Aktivitätsdiagramm

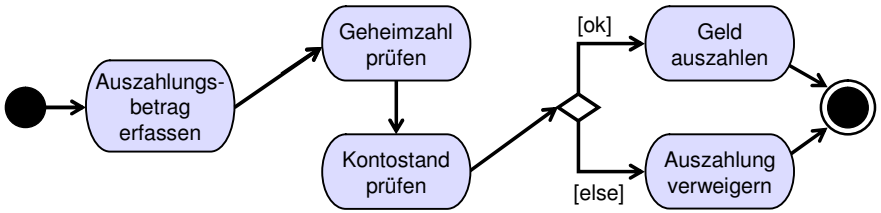
Aktivitätsdiagramm für einen Geldautomaten:



- In einem **Aktivitätsdiagramm** werden die Objekte eines Programms mittels der zeitlich geordneten Angabe der Aktivitäten, die sie während des Programmablaufs vollführen, beschrieben.
- Eine **Aktivität** ist ein einzelner Schritt innerhalb eines Programmablaufs, d.h. ein spezieller Zustand eines Modellelements, der eine interne Aktion sowie eine oder mehrere von ihm ausgehende Transitionen enthält.

Aktivitätsdiagramm

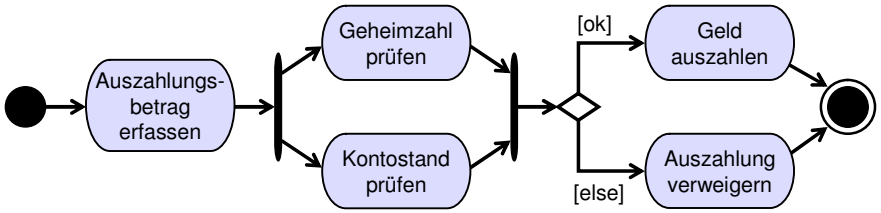
Aktivitätsdiagramm für einen Geldautomaten:



Gehen mehrere Transitionen von der Aktivität aus, so müssen diese mittels Bedingungen voneinander zu unterscheiden sein.

Aktivitätsdiagramm

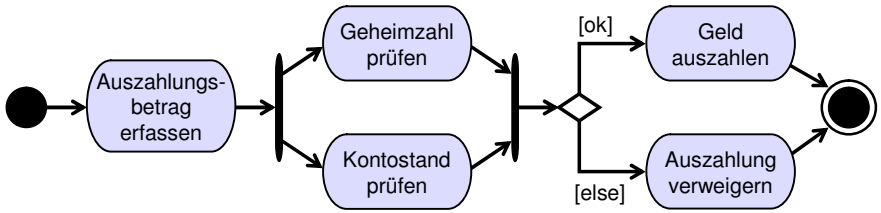
Aktivitätsdiagramm für einen Geldautomaten:



- Aktivitätsdiagramme sind ebenfalls dazu geeignet, um parallele Abläufe (d.h. konkurrierende bzw. synchrone Kontrollflüsse) darzustellen . In der Implementierung wird dies oft auf Multithreading abgebildet.
- Parallele Aktivitäten sind durch **Aufspaltungen** darstellbar und durch **Joins** synchronisierbar.

Aktivitätsdiagramm

Aktivitätsdiagramm für einen Geldautomaten:

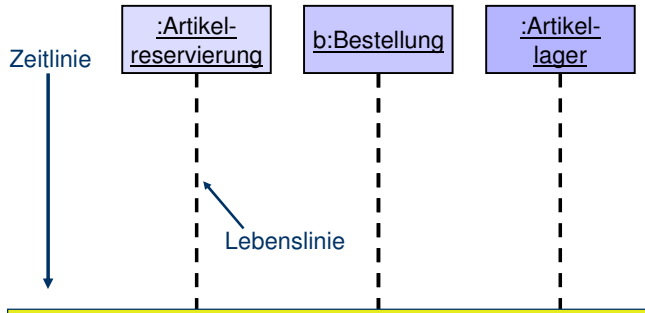


Aktivitätsdiagramm

- Ein Aktivitätsdiagramm kann als Sonderform eines Zustandsdiagramms angesehen werden, dessen Zustände der Modellelemente in der Mehrzahl als Aktivitäten definiert sind.
- Diese Aktivitäten könnten in ein Zustandsdiagramm integriert werden – besser aber ist die Visualisierung in einem eigenen Aktivitätsdiagramm.
- Ein Aktivitätsdiagramm ähnelt in gewisser Weise einem prozeduralem Flussdiagramm, jedoch sind hier alle Aktivitäten eindeutig Objekten zugeordnet, d.h. sie sind entweder einer Klasse, einer Operation oder einem Anwendungsfall eindeutig untergeordnet.

Sequenzdiagramm

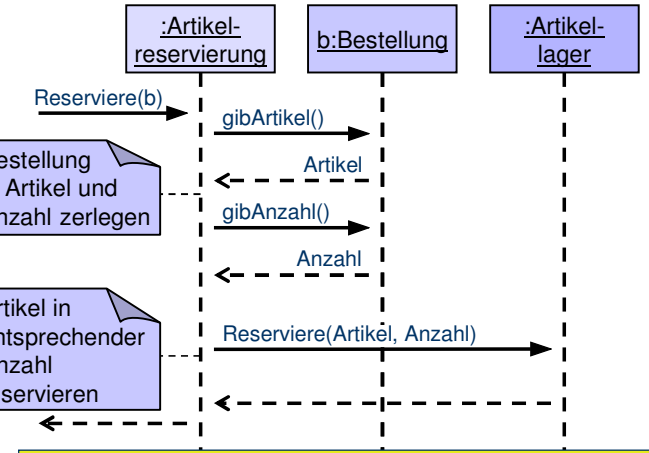
Sequenzdiagramm einer Artikelbestellung:



- Ein **Sequenzdiagramm** beschreibt die zeitliche Abfolge von Interaktionen zwischen einer Menge von Objekten innerhalb eines zeitlich begrenzten Kontexts.
- Die **Zeitlinie** verläuft senkrecht von oben nach unten – die Lebenszeit eines Objekts wird durch seine **Lebenslinie** beschrieben.

Sequenzdiagramm

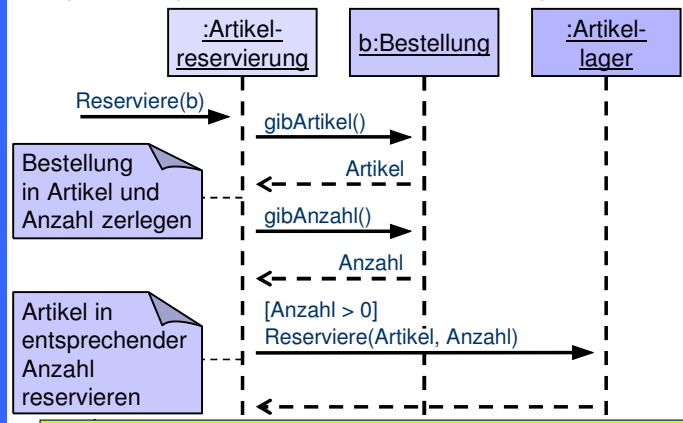
Sequenzdiagramm einer Artikelbestellung:



Von den Objekten gehen an bestimmten Punkten der Lebenslinie benannte **Nachrichten** an andere Objekte aus.

Sequenzdiagramm

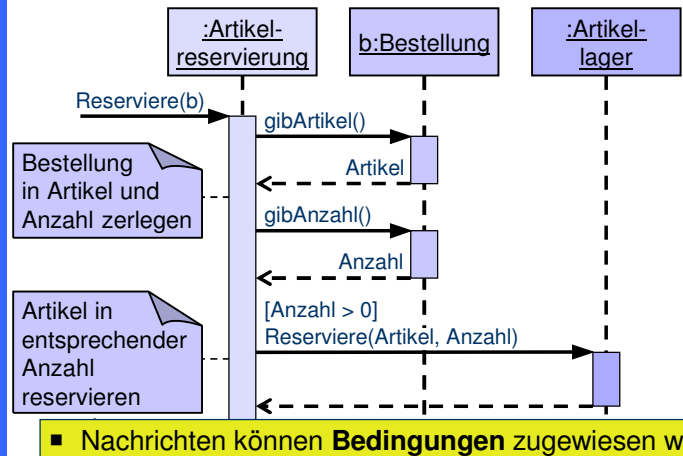
Sequenzdiagramm einer Artikelbestellung:



- Nachrichten können **Bedingungen** zugewiesen werden.

Sequenzdiagramm

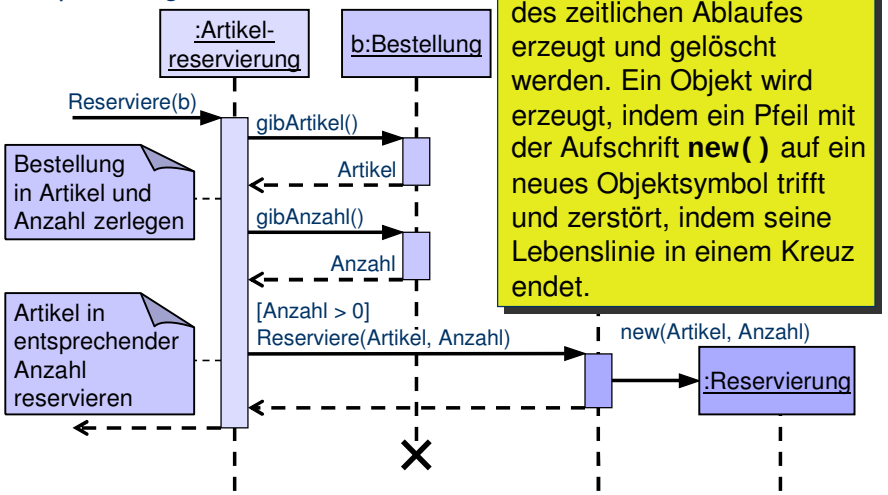
Sequenzdiagramm einer Artikelbestellung:



- Nachrichten können **Bedingungen** zugewiesen werden.
- Objekte, die gerade aktiv an Interaktionen beteiligt sind, werden durch einen Balken auf ihrer Lebenslinie gekennzeichnet.

Sequenzdiagramm

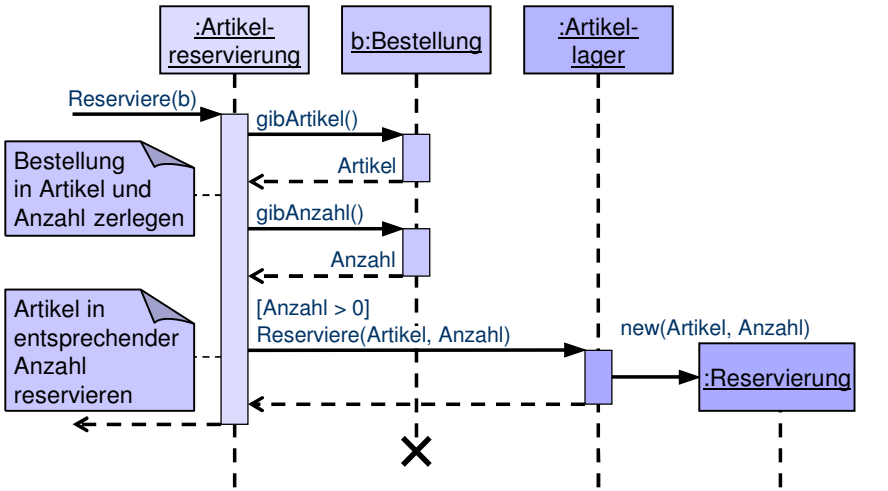
Sequenzdiagramm einer Artikelbestell



Objekte können während des zeitlichen Ablaufes erzeugt und gelöscht werden. Ein Objekt wird erzeugt, indem ein Pfeil mit der Aufschrift **new()** auf ein neues Objektsymbol trifft und zerstört, indem seine Lebenslinie in einem Kreuz endet.

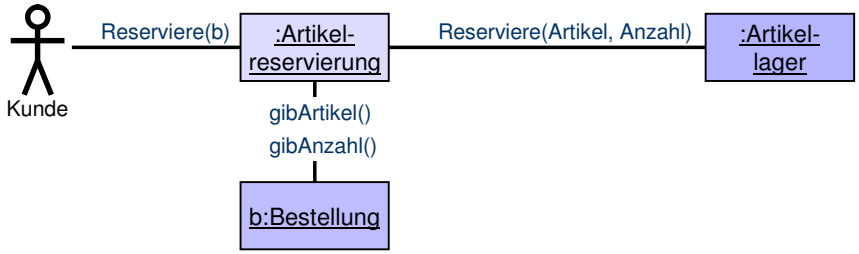
Sequenzdiagramm

Sequenzdiagramm einer Artikelbestellung:



Kollaborationsdiagramm

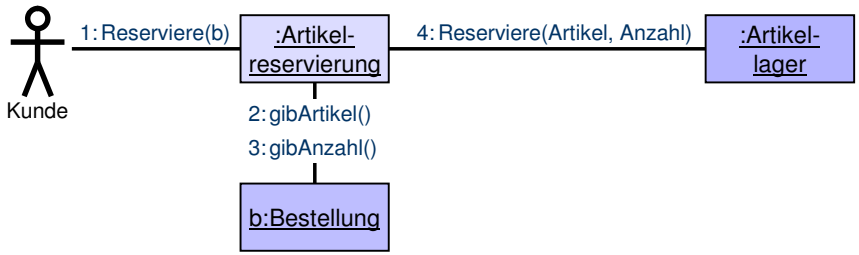
Kollaborationsdiagramm einer Artikelbestellung:



- Ein **Kollaborationsdiagramm** beschreibt wie eine Menge von zusammenhängenden Objekten einen gemeinsamen Vorgang (z.B. einen Anwendungsfall) ausführen.
- Man kann Aspekte anderer Modellierungsdiagramme (z.B. Akteure) mit einbeziehen.

Kollaborationsdiagramm

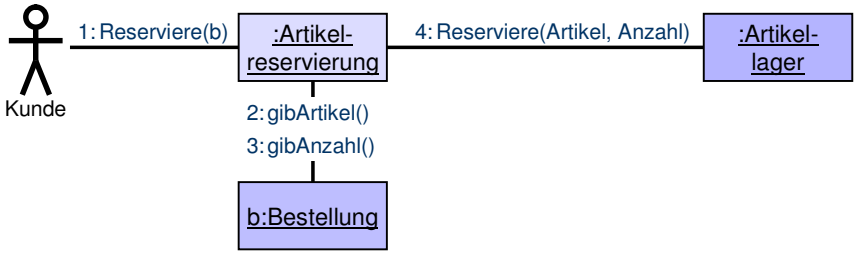
Kollaborationsdiagramm einer Artikelbestellung:



- Die Reihenfolge der ausgetauschten Nachrichten wird durch Nummerierung dargestellt.
- Einzelne Kollaborationen zeigen natürlich jeweils nur einen Teilaspekt der Verantwortlichkeiten eines Objekts.

Kollaborationsdiagramm

Kollaborationsdiagramm einer Artikelbestellung:



Fazit UML

- Die UML ist eine ausdrucksfähige und vielseitige Modellierungssprache, mit der viele Aspekte eines zu realisierenden Systems beschrieben werden können.
- Die UML lässt sich in allen Phasen eines Entwicklungsprozess gewinnbringend einsetzen.
- Die UML löste den Wildwuchs der Modellierungssprachen und Diagrammen ab und stellt zurzeit die lingua franca der Modellierungssprachen dar.
- In der Implementierungsphase kann die UML mit der Unterstützung durch CASE-Tools einen wesentlichen Bestandteil am Entwurf und der Wartung komplexer Systeme leisten.

Fazit UML

- Die UML ist standardisiert und wird weiterentwickelt – so ist im nächsten Jahr (wahrscheinlich) die Version 2.0 zu erwarten, die wesentliche Erweiterungen mit sich bringt.
- Leider bringt dies – wie es bei der Entwicklung von Sprachen üblich ist – damit auch die Markierung bestimmter Sprachelemente als veraltet bzw. vereinzelt auch die Korrektor der Semantik mit sich: Es ist also nötig, hier „am Ball“ zu bleiben.
- Da die UML sehr umfangreich und ausdrucksstark ist kann sie im Rahmen dieser Vorlesung nicht erschöpfend behandelt werden.
- Als zusätzliche Quellen werden empfohlen:
  - `http://www.omg.org/technology/documents/formal/uml.htm`
  - Hitz & Kappel: „UML @ Work“, dpunkt, 2002.
  - Martin Fowler: „UML konzentriert“, Addison-Wesley, 2000.
- Wir werden in späteren Kapiteln nochmals auf verschiedene Bereiche der UML zurückkommen.

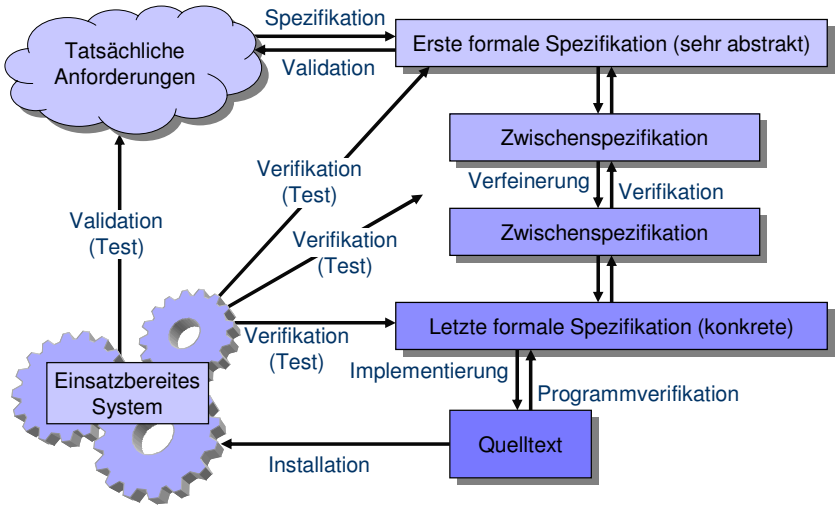
4.2 Formale Modelle

4.2.1 Formale Spezifikation

- Formale Spezifikation:
  - geschrieben in einer formalen Spezifikationssprache
  - definiert rigoros alle akzeptablen Verhalten des spezifischen Systems.
  - ermöglicht Argumentieren über die Spezifikation und Verfeinerungen.
- Formale Spezifikationstechnik:
  - Syntax: Zur Festlegung einer syntaktisch korrekt formulierten Spezifikation.
  - Semantik: Zur Beschreibung der mit einer Spezifikation assoziierten Modelle (Verhalten des Systems).
  - Inferenzsystem: Zur Definition der Deduktionen, die man aus einer formalen Spezifikation ableiten kann: automatisches Beweisen, funktionales Testen, Prototypen, Verifikation von Verfeinerungen.
- Verschiedene Klassen von formalen Spezifikationstechniken:
  - Eigenschaftsorientiert (deklarativ):
    → algebraische Spezifikationen
  - Operational:
    → Petri-Netze, Prozess-Algebren

Formale Modelle

Formale und informale Aspekte bei der Softwareentwicklung:



Formale Modelle

- Während die frühen Stufen der Anforderungsanalyse endbenutzerorientiert sind, sind die späten Stufen (das Aufstellen einer vollständigen, konsistenten und präzisen Spezifikation) auftraggeberorientiert, da die Grundlagen für die Implementierung gelegt werden.
- Formale Spezifikation kann als Teil der Softwarespezifikation angesehen werden (siehe Anforderungsanalyse).
- Während der Softwarespezifikation schaut man sich die Anforderungen genau an, um herauszufinden, was sie für das zu konstruierende System bedeuten. Hierbei werden normalerweise Fehler in der informalen Spezifikation aufgedeckt, die in den Spezifikationsprozess zurückfließen müssen.
- Falls die Anforderungsspezifikation schon als Vertragsbestandteil akzeptiert wurde, muss man diese Probleme mit dem Kunden zur Sprache bringen und auflösen.

Formale Spezifikation

- Eine formale Softwarespezifikation ist eine Spezifikation, die in einer Sprache ausgedrückt wird, deren Vokabular, Syntax und Semantik formal definiert ist. Dies schließt „natürliche Sprache“ aus und impliziert „Mathematik“.
- Bei den herkömmlichen Ingenieurdisziplinen verlief die Entwicklung der Disziplin selbst parallel mit der Entwicklung der Mathematik. Es gab normalerweise kein Problem damit, Fortschritte in der Mathematik in die Ingenieurdisziplin aufzunehmen. In der Softwareentwicklung ist das nicht so. Formale Methoden werden in der Praxis nur wenig eingesetzt.

Argumente gegen formale Spezifikation:

- Einige Gründe, warum formale Spezifikationsmethoden nicht weit verbreitet sind:
  - Das Management von Softwarefirmen ist konservativ und weigert sich, neue Techniken zu akzeptieren, bei denen nicht offensichtlich ist, welche Vorteile sie aufweisen.
  - Viele Programmierer haben nie gelernt, mit formalen Methoden umzugehen.
  - Die Kunden kennen sich damit nicht aus und wollen nicht für etwas bezahlen, das sie nicht bewerten können.
  - Einige Klassen von Software (Benutzungsschnittstellen, parallele Systeme und interruptgesteuerte Systeme) lassen sich mit heutigen Mitteln nicht gut formal spezifizieren.
  - Formale Spezifikationstechniken gelten als unpraktisch (obwohl sie in einigen nichttrivialen Projekten erfolgreich benutzt wurden).
  - Die Forschung befasst sich wesentlich mehr mit Sprachen und ihren theoretischen Grundlagen als mit Methoden und Werkzeugen.
- Die Gegenargumente haben meist mit der Einführung formaler Spezifikationen in die gängigen Entwicklungsprozesse zu tun.

Argumente für formale Spezifikation

- Die Befürworter formaler Methoden stellen ihre technischen Vorteile in den Vordergrund:
  - Die Entwicklung formaler Spezifikationen führt dazu, dass man die Anforderungen und den Softwareentwurf besser versteht. Dadurch werden Fehler und Auslassungen verringert, und es entsteht ein eleganterer Entwurf.
  - Da formale Spezifikationen Mathematik sind, können sie mit mathematischen Mitteln analysiert werden.
  - Es kann möglich sein, Konsistenz und Vollständigkeit der Spezifikation zu beweisen – anstatt nur zu vermuten. Eventuell kann man sogar Korrektheitsbeweise für die späteren Programme führen.
  - Formale Spezifikationen können mit geeigneten Werkzeugen automatisch verarbeitet werden.
  - Formale Spezifikationen können beim Testen einer Komponente helfen, geeignete Testfälle zu identifizieren.

Sieben Mythen der formalen Methoden (nach Hall)

- Hall (1990) führt die „sieben Mythen der formalen Methoden“ auf und erklärt, warum sie seiner Ansicht nach nicht stimmen:
  1. *„Formale Methoden führen zu perfekter Software.“*
     Dies ist offensichtlicher Unsinn, da auch formale Spezifikationen Fehler haben können. Ferner setzt dies voraus, daß auch Compiler, Betriebssystem und Hardware perfekt sind. Formale Methoden sind trotzdem nützlich, weil Spezifikationsfehler leichter aufgedeckt werden können.
  2. *„Formale Methoden bedeuten Korrektheitsbeweise für Programme.“*
     Man kann natürlich Korrektheitsbeweise führen, aber von einer Spezifikationsanalyse hat man normalerweise mehr.
  3. *„Formale Methoden sind so teuer, dass man sie nur in sicherheitskritischen Systemen anwenden sollte.“*
     Hall behauptet, dass in seine Firma die allgemeinen Entwicklungskosten durch formale Methoden gesunken sind.

Sieben Mythen der formalen Methoden (nach Hall)

4. *„Formale Methoden benötigen große mathematische Fähigkeiten.“*
   Das stimmt nicht. Die dahinter stehende Mathematik ist simpel.
5. *„Formale Methoden erhöhen die Entwicklungskosten.“*
   In Halls Erfahrung ist das nicht so, aber die Kosten werden zumindest umgeschichtet.
6. *„Kunden können formale Spezifikationen nicht verstehen.“*
   Hall schlägt vor, die Spezifikationen in natürlicher Sprache zu umschreiben oder mit geeigneten Werkzeugen zu animieren. Seiner Ansicht nach führt die Entwicklung einer formalen Spezifikation zu einer besseren, leichter verständlichen Anforderungsspezifikation.
7. *„Formale Methoden wurden nur für triviale Systeme benutzt.“*
   Dafür gibt es diverse Gegenbeispiele.

Bewertung (nach Sommerville)

- Es ist schwer, zu einem abschließenden Urteil zu kommen, da die Argumente und Gegenargumente von unterschiedlichen Voraussetzungen ausgehen.
- Das ganze ist auch ein emotionales Thema, da manche Akademiker, die sich mit formalen Methoden befassen, behaupten, dass „Software-Engineering“ die Verwendung formaler Methoden voraussetzt.
- Solcher Unsinn macht die Praktiker, die tatsächlich Software entwerfen, natürlich argwöhnisch.

Bewertung (nach Sommerville)

- Zwei Einflüsse haben die Einführung formaler Methoden in der letzten Zeit weiter behindert:
  1. Interaktive Systeme sind viel wichtiger geworden. Man hat herausgefunden, dass die Entwicklung von Prototypen wichtig ist, um Anforderungen an interaktive Systeme zu verstehen.

     Prototypentwicklung lässt sich nur schwer mit formalen Methoden unter einen Hut bringen. Dies gilt auch für die immer wichtiger werdenden graphischen Schnittstellen.

  2. „Software-Engineering klappt auch so." Die Anwendung (anderer) Software-Engineering-Methoden haben die Softwarequalität schon merkbar erhöht.

     Dadurch werden Behauptungen der frühen Befürworter widerlegt, die sagten, dass Korrektheitsbeweise für Qualitäts-verbesserungen unabdingbar seien.

     Wenn es auch ohne mathematische Methoden besser geht, warum sollte man dann jetzt damit anfangen?

Bewertung (nach Sommerville)

- Bei der Entwicklung interaktiver und graphischer Systeme ist es wohl effektiver, Ressourcen in andere Verbesserungen des Entwicklungsprozesses zu investieren als in formale Methoden.
- Bei sicherheitskritischen Systemen werden formale Methoden sich weit verbreiten, da sie das Vertrauen in die Sicherheit und Zuverlässigkeit steigern. Es kann sinnvoll sein, die kritischen Teile dieser Systeme mit formalen Methoden zu entwickeln und den Rest (die interaktiven Teile) wie üblich.
- Formale Methoden werden auch in der Definition von Normen verwendet werden, da diese unzweideutig und präzise sein müssen.
- Man kann also nicht eindeutig sagen, ob formale Spezifikationen und die dazugehörigen Methoden eine gute Idee sind oder nicht. Für viele Systeme sind sie einfach nicht kosteneffektiv. Für eine andere – u. U. kleine, aber sehr wichtige – Klasse von Systemen sind sie vermutlich lebenswichtig, wenn die Softwarequalität verbessert werden soll.

4.2.2 Spezifikation durch Funktionen

- Ein klassischer und bekannter Ansatz zur formalen Spezifikation beruht auf Funktionen, die Eingaben annehmen und ein Ergebnis liefern, aber keinen inneren Zustand erhalten.
- Solche Funktionen können über **Vor**- und **Nachbedingungen** spezifiziert werden; Vorbedingungen beschreiben Anforderungen, die die Eingaben der Funktion erfüllen – Nachbedingungen beschreiben Anforderungen an das Ergebnis der Funktion.
- Eine Funktion wird also durch den Unterschied zwischen Vor- und Nachbedingung definiert.

Vor- und Nachbedingung

- **Zusicherungskalkül** (Hoare 1969):

  Definition: *A* sei eine Anweisung in einer imperativen Programmiersprache und die Zusicherungen *P* bzw. *Q* beschreiben den Zustand der Programmausführung vor und nach der Anweisung *A*.

  Wenn die Ausführung von *A* in einem durch *P* beschriebenen Zustand nach endlich vielen Schritten zu dem durch *Q* beschriebenen Zustand führt, soll die Zusicherung $\{P\}\ A\ \{Q\}$ gelten.

$$\{P\} \wedge A \rightarrow \{Q\} \quad \rightarrow \quad \{P\}\ A\ \{Q\}$$

- *P* heißt **Vor-** und *Q* **Nachbedingung** einer solchen Anweisung *A*.
- Beispiel: Eine Zuweisung `i:=7` liefert:

$$\{P: i \text{ beliebig}\}\ \texttt{i:=7}\ \{Q: i = 7\}$$

Vor- und Nachbedingung

- P und Q sind Zusicherungen über einen Zustand; $\{P\}\ A\ \{Q\}$ ist eine Zusicherung über den Ablauf. Der Zusatz „nach endlich vielen Schritten“ in der Definition stellt sicher, dass Schleifen und rekursive Prozeduren terminieren.
- *Modus Ponens*: Aus der Gültigkeit von $\{P\} \wedge A$ und $\{P\} \wedge A \rightarrow \{Q\}$ lässt sich die Gültigkeit von $\{Q\}$ folgern.

$$\frac{\{P\} \wedge A \qquad \{P\} \wedge A \rightarrow \{Q\}}{\{Q\}}$$

- Der Zusicherungskalkül enthält Axiome / Inferenzregeln für Leeranweisungen, Zuweisungen, Sequenzen, Selektionen, Iterationen, Schleifen, Prozeduren und weitere Konstrukte von Programmiersprachen.

Vor- und Nachbedingung

- Vorbedingungen können verschärft und Nachbedingungen abgeschwächt werden:

$$P' \rightarrow P,\ Q \rightarrow Q' \text{ und } \{P\}\ A\ \{Q\} \quad \rightarrow \quad \{P'\}\ A\ \{Q'\}$$

- Die Anweisung A kann auch das ganze Programm sein. Das Paar (P,Q) spezifiziert dann das Problem, A löst das Problem wenn $\{P\}\ A\ \{Q\}$ gilt.
- Die Schreibweise

$$P\ \{A\}\ Q$$

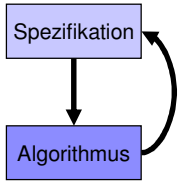
bezeichnet die partielle Korrektheit von A: Falls A im Zustand P ausgeführt wird und terminiert, so gilt anschließend Q. Bei der Notation $P\ \{A\}\ Q$ muss die Terminierung von A gesondert bewiesen werden.

Spezifikation durch Funktionen

- Der Begriff der Vor- und Nachbedingungen ist für die formale Spezifikation grundlegend.
- Es ist möglich, große Systeme als eine Anzahl von Funktionen zu beschreiben, aber es ist unnatürlich und führt zu komplizierten Spezifikationen.
- Es gibt Ansätze, die leichter zu benutzen sind:
  - Programmentwicklung durch Transformation,
  - ein algebraischer Ansatz, bei dem das System durch Operationen und ihre Zusammenhänge beschrieben wird,
  - ein modellbasierter Ansatz, bei dem aus gut verstandenen mathematischen Begriffen wie *Menge* und *Folge* ein Modell des Systems konstruiert wird.
- Die Spezifikation paralleler Systeme ist komplizierter und wird hier aus Zeitgründen nicht behandelt.

4.2.3 Programmentwicklung durch Transformation

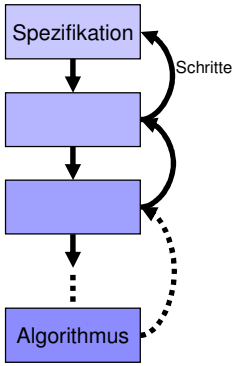
- Wenn man eine formale Spezifikation für ein Programm hat, kann man versuchen, das Programm durch Transformation daraus zu enthalten. Hierzu wird eine Folge von „korrektheitserhaltenden“ Transformationen auf das Programm angewendet:



4.2.3 Programmentwicklung durch Transformation

- Wenn man eine formale Spezifikation für ein Programm hat, kann man versuchen, das Programm durch Transformation daraus zu enthalten. Hierzu wird eine Folge von „korrektheitserhaltenden“ Transformationen auf das Programm angewendet:

Transformation:



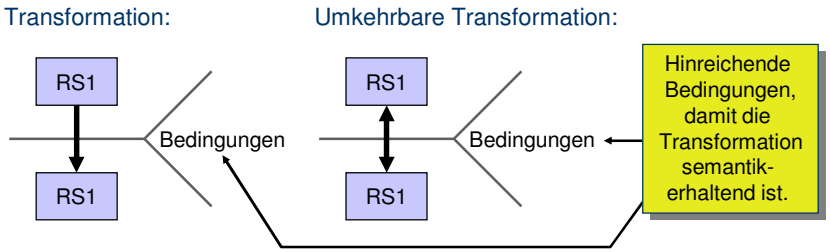
- Die Idee dahinter ist, dass der Schritt, den eine einzelne Transformation macht, nicht zu kompliziert ist, so dass man ihre Korrektheit „leicht“ nachvollziehen kann.
- Wenn es keine Transformationsfehler gibt, kann man garantieren, dass das Programm eine zutreffende Implementierung der Spezifikation ist.

Programmentwicklung durch Transformation

- Die Komplexität der einzelnen Transformationen ist geringer als die Entfernung zwischen Spezifikation und Programm. Die Korrektheit der einzelnen Transformationen lässt sich leichter beweisen, als dass das „wie auch immer erstellte“ Programm die Spezifikation einhält.
- Korrektheitsbeweise sind für große Systeme nicht praktikabel. Allerdings ist die Auswahl der Transformationen auch nicht trivial.
- Wenige große Systeme (wenn überhaupt) wurden auf diese Weise entwickelt. Es ist unwahrscheinlich, dass ein solcher Ansatz jemals ernsthaft eingesetzt wird.
- Allerdings kann es für die Entwicklung kritischer Systeme sinnvoll sein, diese Vorgehensweise in andere Prozessmodelle zu integrieren.

Programmentwicklung durch Transformation

- 2 Rechenschablonen RS1, RS2 heißen **äquivalent**, wenn sie nach Einsetzen beliebiger Anweisungsfolgen zu Programmstücken führen, die bei beliebiger Vorbesetzung der Variablen und beliebigen Prozeduraufrufen stets die gleichen Ergebnisse liefern.
- Die Ersetzung eines Programmstücks nach Schablone RS1 durch ein Programmstück nach einer äquivalenten Schablone RS2 heißt **semantikerhaltende Transformation**.



Programmentwicklung durch Transformation

Beispiel: Vertauschen von Abfragen

- Ist P die Vor- und Q die Nachbedingung der gesamten bedingten Anweisung, so muss

  $\{P \wedge B \wedge B'\}$ A1 $\{Q\}$,
  $\{P \wedge B \wedge \neg B'\}$ A2 $\{Q\}$,
  $\{P \wedge \neg B \wedge B'\}$ C1 $\{Q\}$ und
  $\{P \wedge \neg B \wedge \neg B'\}$ C2 $\{Q\}$

  gelten, damit die erste Form der geschachtelten bedingten Anweisung richtig ist.
- Setzen wir dies in die zweite Form ein, so ergibt sich deren Korrektheit.

Transformationsschema:

```
if B
then if B'
     then A1
     else A2
     end
else
     if B'
     then C1
     else C2
     end
end
```

Berechnung von B und B' vertauschbar.

```
if B'
then if B
     then A1
     else C1
     end
else
     if B
     then A2
     else C2
     end
end
```

4.2.4 Algebraische Spezifikation

- Große Systeme werden normalerweise in Teilsysteme zerlegt, die über definierte Schnittstellen miteinander kommunizieren.
- Diese Schnittstellen kann man als Menge von **abstrakten Datentypen** (*ADT*) oder Objekten definieren, die die Teilsysteme implementieren.
- Der Zugriff auf die Teilsysteme ist nur über diese Schnittstellen möglich.
- Die Schnittstellen müssen eindeutig spezifiziert werden, um Missverständnisse zwischen dem aufgerufenen (Dienstanbieter) und dem aufrufenden Subsystem (Dienstnutzer) zu vermeiden.

Algebraische Spezifikation

- Die Vorgehensweise ist üblicherweise so, dass die Designer der Subsysteme sich über die Schnittstellen einigen und eine informale Spezifikation herstellen, die eine Menge von abstrakten Datentypen oder Objekten einführt, deren Operationen ihrerseits formal definiert werden.
- Es kann nützlich sein, eine solche Spezifikation zu machen, selbst wenn man nicht vorhat, mit ihr mathematisch umzugehen.
- Die Aufstellung einer formalen Spezifikation erfordert, dass man sich sehr eingehend mit der informalen Spezifikation beschäftigt. Dabei können mögliche Inkonsistenzen und Mehrdeutigkeiten in der informalen Beschreibung aufgedeckt werden.
- Eine formale Spezifikation ergänzt die Informale und verringert Verständigungsprobleme zwischen den Entwicklern eines Subsystems und den Benutzern der Schnittstelle.

Algebraische Spezifikation

- Die Algebraische Spezifikation ist der theoretische Hintergrund für abstrakte Datentypen.
- Sie erlaubt eine elegante Spezifikation abstrakter Datentypen und ist daher sehr wichtig.
- Generelles Vorgehen:
  1. Angabe von Objektarten, über die man etwas sagen will (*Sorten*, *Typen*).
  1. Angabe von Operationen samt Argument- / Ergebnistypen (*Signatur*).
  1. Angabe der Semantik der Operationen in Form von (bedingten) Gleichungen.

Algebraische Spezifikation

Natürliche Zahlen

Algebraische Spezifikation

Natürliche Zahlen
algebra NatAlg **uses** BoolAlg

Algebraische Spezifikation

Natürliche Zahlen
algebra NatAlg **uses** BoolAlg **sorts** Nat, Bool

Algebraische Spezifikation

Natürliche Zahlen
Operations: 0: → Nat s: Nat → Nat +: Nat × Nat → Nat ==: Nat × Nat → Bool

Signatur (Syntax)

Algebraische Spezifikation

Natürliche Zahlen
algebra NatAlg **uses** BoolAlg **sorts** Nat, Bool
Operations: 0: → Nat s: Nat → Nat +: Nat × Nat → Nat ==: Nat × Nat → Bool
Equations: 0 == 0 := true 0 == s(x) := false s(x) == 0 := false s(x) == s(y) := x == y x + 0 := x x + s(y) := s(x + y)

Signatur (Syntax)

Axiome (Semantik)

Algebraische Spezifikation

Natürliche Zahlen

algebra NatAlg **uses** BoolAlg
sorts Nat, Bool

Operations:
0: → Nat
s: Nat → Nat
+: Nat × Nat → Nat
==: Nat × Nat → Bool

Equations:
0 == 0 := true
0 == s(x) := false
s(x) == 0 := false
s(x) == s(y) := x == y
x + 0 := x
x + s(y) := s(x + y)

Beispiel:

s(s(s(0)))+s(s(0))

Algebraische Spezifikation

Natürliche Zahlen

algebra NatAlg **uses** BoolAlg
sorts Nat, Bool

Operations:
0: → Nat
s: Nat → Nat
+: Nat × Nat → Nat
==: Nat × Nat → Bool

Equations:
0 == 0 := true
0 == s(x) := false
s(x) == 0 := false
s(x) == s(y) := x == y
x + 0 := x
x + s(y) := s(x + y)

Beispiel:

$\underbrace{s(s(s(0)))}_{x} + \underbrace{s(s(0))}_{s(y)} = s(x + y)$

Algebraische Spezifikation

Natürliche Zahlen

algebra NatAlg **uses** BoolAlg
sorts Nat, Bool

Operations:
0: → Nat
s: Nat → Nat
+: Nat × Nat → Nat
==: Nat × Nat → Bool

Equations:
0 == 0 := true
0 == s(x) := false
s(x) == 0 := false
s(x) == s(y) := x == y
x + 0 := x
x + s(y) := s(x + y)

Beispiel:

$\underbrace{s(s(s(0)))}_{x} + \underbrace{s(s(0))}_{s(y)} = s(x + y)$

$= s(s(s(s(0)))+s(0))$

Algebraische Spezifikation

Natürliche Zahlen

algebra NatAlg **uses** BoolAlg
sorts Nat, Bool

Operations:
0: → Nat
s: Nat → Nat
+: Nat × Nat → Nat
==: Nat × Nat → Bool

Equations:
0 == 0 := true
0 == s(x) := false
s(x) == 0 := false
s(x) == s(y) := x == y
x + 0 := x
x + s(y) := s(x + y)

Beispiel:

$\underbrace{s(s(s(0)))}_{x} + \underbrace{s(s(0))}_{s(y)} = s(x + y)$

$= \underbrace{s(s(s(0)))}_{x} + \underbrace{s(0)}_{s(y)} = s(x + y)$

Algebraische Spezifikation

Natürliche Zahlen

algebra NatAlg **uses** BoolAlg
sorts Nat, Bool

Operations:
0: → Nat
s: Nat → Nat
+: Nat × Nat → Nat
==: Nat × Nat → Bool

Equations:
0 == 0 := true
0 == s(x) := false
s(x) == 0 := false
s(x) == s(y) := x == y
x + 0 := x
x + s(y) := s(x + y)

Beispiel:

$\underbrace{s(s(s(0)))}_{x} + \underbrace{s(s(0))}_{s(y)} = s(x + y)$

$= \underbrace{s(s(s(0)))}_{x} + \underbrace{s(0)}_{s(y)} = s(x + y)$

$= s(s(s(s(s(0))))+0))$

Algebraische Spezifikation

Natürliche Zahlen

algebra NatAlg **uses** BoolAlg
sorts Nat, Bool

Operations:
0: → Nat
s: Nat → Nat
+: Nat × Nat → Nat
==: Nat × Nat → Bool

Equations:
0 == 0 := true
0 == s(x) := false
s(x) == 0 := false
s(x) == s(y) := x == y
x + 0 := x
x + s(y) := s(x + y)

Beispiel:

$\underbrace{s(s(s(0)))}_{x} + \underbrace{s(s(0))}_{s(y)} = s(x + y)$

$= \underbrace{s(s(s(0)))}_{x} + \underbrace{s(0)}_{s(y)} = s(x + y)$

$= \underbrace{s(s(s(s(0))))}_{x} + 0) \quad + 0 \quad = x$

$= s(s(s(s(s(0)))))$

Algebraische Spezifikation

Natürliche Zahlen

algebra NatAlg **uses** BoolAlg
sorts Nat, Bool

Operations:
0: → Nat
s: Nat → Nat
+: Nat × Nat → Nat
==: Nat × Nat → Bool

Equations:
0 == 0 := true
0 == s(x) := false
s(x) == 0 := false
s(x) == s(y) := x == y
x + 0 := x
x + s(y) := s(x + y)

Beispiel:

$\underbrace{s(s(s(0)))}_{x} + \underbrace{s(s(0))}_{s(y)} = s(x + y)$

$= \underbrace{s(s(s(0)))}_{x} + \underbrace{s(0)}_{s(y)} = s(x + y)$

$= \underbrace{s(s(s(s(0))))}_{x} + 0) \quad + 0 \quad = x$

$= s(s(s(s(s(0)))))$

Algebraische Spezifikation

- In der hier gezeigten Notation hat jede Spezifikation einen Namen und eine optionale Liste von generischen Parametern.
- Durch diese ist es möglich, abstrakte Typen zu definieren, die eine Sammlung (z.B. *Feld*, *Liste*) von anderen Typen sind, ohne sich um die enthaltenen Typen kümmern zu müssen.
- Die generischen Typen können durch tatsächliche Typen ersetzt werden, um eine spezifischere Spezifikation zu erhalten.

Algebraische Spezifikation

- Eine Spezifikation besteht aus vier Teilen:
  1. Eine **Einleitung**, die den Typnamen (*sort*) des zu definierenden Elements angibt, außerdem welche Spezifikationen (nicht Typnamen) benutzt werden.
     Durch diesen „Import“ können die darin enthaltenen Typnamen in der vorliegenden Spezifikation verwendet werden.
  2. Einen **Beschreibungsteil**, in dem die Operationen informal beschrieben werden. Dadurch wird die formale Spezifikation leichter verständlich.
  3. Einen **Signaturteil**, der die Syntax der Schnittstelle zur Klasse bzw. zum abstrakten Datentyp festlegt.
     Hierzu gehören die Namen der Operationen, Namen und Typnamen der Parameter und der Ergebnisse.
  4. Einen **axiomatischen Teil**, der die Semantik der Operationen durch eine Menge von Axiomen beschreibt, die das Verhalten des Typs beschreiben.
     Diese Axiome setzen Konstruktionsoperationen mit Inspektionsoperationen in Beziehung.

Algebraische Spezifikation

- **Beispiel:** Ein Array ist ein abstrakter generischer Datentyp, den die meisten Programmiersprachen anbieten.
- Der generische Parameter *Elem* soll für „jeden beliebigen Elementtyp“ stehen, und wir müssen die Operationen angeben, die für den generischen Datentyp definiert sein müssen, so dass er benutzt werden kann.
- So könnte der Typ eine Operation *Undefined* haben, deren Resultat (vom Typ *Elem*) anzeigt, dass irgendwo ein Fehler aufgetreten ist – etwa weil ein Element angesprochen wurde, das keinen Wert hat.

Algebraische Spezifikation

Array (Elem: [Undefined ⇒ Elem])

sort Array

imports INTEGER

Arrays sind Sammlungen von Elementen des generischen Typs **Elem**. Sie haben eine Ober- und Untergrenze (durch die Operationen **First** und **Last** abfragbar). Einzelne Elemente werden mit **Eval** durch ihren numerischen Index angesprochen. **Create** übernimmt dir Grenzen als Parameter und erzeugt das Array (alle Einträge sind **Undefined**). **Assign** erzeugt aus einem Array ein neues Array, bei dem das bezeichnete Element den angegebenen Wert hat. Zuweisungen jenseits der Arraygrenzen liefern **Undefined**.

Signatur:

Create: Integer × Integer → Array

Assign: Array × Integer × Elem → Array

Algebraische Spezifikation

liefern **Undefined**.

Signatur:

Create: Integer × Integer → Array

Assign: Array × Integer × Elem → Array

First: Array → Integer

Last: Array → Integer

Eval Array × Integer → Elem

Axiome:

Firste(Create(x, y)) := x

First(Assign(a, n, v)) := First(a)

...

Eval(Assign(a, n, v), m) :=

if m < First(a) or m > Last(a) **then** Undefined

else if m = n **then** v **else** Eval(a,m)

Systematisches Vorgehen

- Im folgenden Abschnitt zeigen wir eine systematische Methode zur Herleitung einer algebraischen Spezifikation für einen abstrakten Datentyp bzw. eine Objektklasse, hier einer einfachen verketteten Liste.
- Die Methode hat sechs Stufen, die aber nicht notwendigerweise streng in der angegebenen Reihenfolge durchlaufen werden – oft kommt man aufgrund neuer Erkenntnisse wieder zu früheren Stufen zurück.

Systematisches Vorgehen

1. **Identifikation:**
   Aus der Systemspezifikation heraus muss die Notwendigkeit einer konzeptuellen Abstraktion identifiziert werden, die als Datentyp umgesetzt werden kann.
2. **Strukturieren der Spezifikation:**
   Die informal gegebene Schnittstellenspezifikation muss in eine Menge von abstrakten Datentypen strukturiert und Operationen müssen identifiziert werden.
3. **Namensvergabe:**
   Der Spezifikation müssen eine Name und evtl. generische Parameter zugewiesen werden. Ein Typname ist ebenfalls erforderlich.

Systematisches Vorgehen

4. **Operationsauswahl:**
   Man wählt auf der Basis der gewünschten Funktionalität eine Menge von Operationen aus. Gegebenenfalls muss man zu den in der informalen Spezifikation identifizierten Operationen noch weitere hinzufügen.
5. **Syntaxdefinition:**
   Man bestimmt die Syntax der Operationen und die Art sowie die Typen der beteiligten Parameter. Dies führt zum Signaturteil der formalen Spezifikation.
6. **Axiomdefinition:**
   Man bestimmt die Semantik der Operationen.

Systematisches Vorgehen

- Für das Listenbeispiel gehen wir davon aus, dass der erste Schritt bereits durchgeführt wurde. (Wir haben festgestellt, dass wir eine Liste benötigen.)
- Der Spezifikations- und der Typname können gleich sein, aber es ist nützlich, zwischen den beiden durch eine Konvention zu unterscheiden. Die Spezifikation nennen wir **LIST** und den Typnamen **List**. Der generische Parameter soll **Elem** heißen.
- Im Allgemeinen benötigt man für jeden abstrakten Typen eine Operation zum Erzeugen entsprechender Objekte (*Create*) und zum Zusammensetzen des Typs aus den Elementen (*Cons*).

Systematisches Vorgehen

- Für eine Liste benötigen wir eine Operation, die das erste Element der Liste liefert (*Head*) und eine, die die Liste ohne das erste Element liefert (*Tail*).
- Ferner ist eine Operation, die die Anzahl der Elemente in der Liste bestimmt (*Length*) notwendig.
- Nun müssen Parameter und deren Typen festlegen werden. Parameter können den zu definierenden Typen (*List*), den generischen Typen (*Elem*) und weitere (z.B. *Integer* oder *Boolean*) enthalten.

Beispiel: Liste

LIST (Elem: [Undefined → Elem])
sort List **imports** INTEGER
Definiert eine Liste, bei der Elemente hinten angehängt und vorne weggenommen werden können. **Create** erzeugt eine neue (leere) Liste, **Cons** erzeugt eine Liste mit einem angehängten neuen Element, **Length** bestimmt die Länge der Liste, **Head** greift auf das vorderste Element der Liste zu und **Tail** gibt die Liste ohne ihr vorderstes Element zurück.
Signatur: Create: → List Cons: List × Elem → List Tail: List → List Head: List → Elem

Beispiel: Liste

Signatur:
Create: → List Cons: List × Elem → List Tail: List → List Head: List → Elem Length: List → Integer
Axiome: Head(Create) := Undefined // *Leere Liste: kein Wert* Head(Cons(L, v)) := **if** L = Create **then** v **else** Head(L) Length(Create) := 0 Length(Cons(L, v)) := Lenght (L) +1 Tail(Create) := Create Tail(Cons(L, v)) := **if** L = Create **then** Create **else** Cons(Tail(L), v)

Beispiel: Liste

Beispiel: Tail(Cons(L, v)) = Cons(Tail(L), v) mit L = $\{x_1, x_2, x_3\}$

Tail(Cons(L, v)) = Tail(Cons($\{x_1, x_2, x_3\}$, v))

= Tail($\{x_1, x_2, x_3, v\}$)

= $\{x_2, x_3, v\}$

= {Tail($\{x_1, x_2, x_3\}$), v}

= {Tail(L), v}

= Cons(Tail(L), v)

Axiome

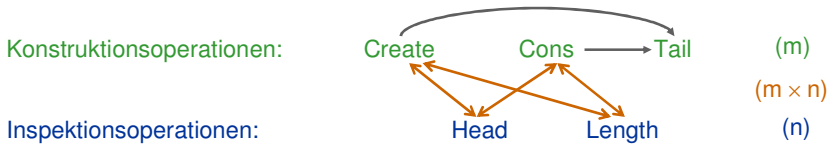
- Die **Axiome**, die die Semantik eines abstrakten Datentyps definieren, werden mit den Operationen im Signaturteil beschrieben.
- Sie geben die Semantik an, indem sie festlegen, welche Aussagen über Objekte des betreffenden Typs allgemeingültig sind.
- Abstrakte Datentypen haben normalerweise zwei Klassen von Operationen:
  1. **Konstruktionsoperationen:**
     Diese erzeugen oder verändern Objekte des betreffenden Typs. Sie haben meist Namen wie *Create*, *Update*, *Add*, usw.
  2. **Inspektionsoperationen:**
     Diese werten Attribute des betreffenden Typs aus. Ihre Namen entsprechen oft Attributnamen oder Namen wie z.B. *Eval* und *Get*.

Axiome

- Eine Faustregel zum Aufstellen von Axiomen besteht darin, für jedes Paar aus Konstruktions- und Inspektionsoperation ein Axiom zu formulieren.
- Wenn es m Konstruktions- und n Inspektionsoperationen gibt, sollte man also m*n Axiome definieren.
- Allerdings sind manche Konstruktions-Operationen aus einfacheren zusammengesetzt; in diesem Fall muss man nur die „primitiven“ Konstruktionsoperationen betrachten.

Axiome

- Eine Faustregel zum Aufstellen von Axiomen besteht darin, für jedes Paar aus Konstruktions- und Inspektionsoperation ein Axiom zu formulieren.
- Wenn es m Konstruktions- und n Inspektionsoperationen gibt, sollte man also m*n Axiome definieren.
- Allerdings sind manche Konstruktions-Operationen aus einfacheren zusammengesetzt; in diesem Fall muss man nur die „primitiven“ Konstruktionsoperationen betrachten.
- Im Beispiel kann man Tail durch Cons und Create ausdrücken, so dass es nicht nötig ist, für Tail Axiome für die Head- und Length-Operationen aufzustellen.



Wiederverwendung von Spezifikationen

- Das Schreiben formaler Spezifikationen ist zeitaufwendig und teuer.
- Eine Strategie dafür, den Aufwand zu verringern, besteht darin, existierende Spezifikationen wiederzubenutzen.
- Man muss dazu in der Lage sein, Spezifikationen inkrementell voneinander abzuleiten.
- Einige Methoden dafür sind:
  - Ausprägung generischer Spezifikationen,
  - Inkrementelle Entwicklung von Spezifikationen,
  - Anreicherung von Spezifikationen.

Ausprägung

- Die einfachste Form der Wiederverwendung ist, eine existierende Spezifikation zu nehmen, die einen generischen Parameter enthält, und diesen durch einen tatsächlichen Typ zu ersetzen.

CHAR_ARRAY: ARRAY
sort Char_Array **instantiates** Array (Elem := Char)
imports INTEGER

- Hier setzen wir voraus, dass der Typname *Char* anderswo definiert wurde und eine konstante Operation namens *Undefined* besitzt. Hierfür könnte man z.B. ein bestimmtes Bitmuster reservieren.
- Bei der Ausprägung ist die Menge der Operationen dieselbe wie die Menge der Operationen in der generischen Spezifikation.

Inkrementelle Entwicklung

- Bei der **inkrementellen Entwicklung** konstruiert man erst mehrere einfache Spezifikationen, die man dann zu Komplizierteren zusammensetzt.
- Die Einfachen werden dabei importiert, so dass ihre Operationen für den Gebrauch in den Komplizierteren zur Verfügung stehen.

Inkrementelle Entwicklung

Beispiel: Grundlegend für ein graphisches System ist ein abstrakter Datentyp, der kartesische Koordinaten repräsentiert:

COORD
sort Coord **imports** INTEGER, BOOLEAN
Definiert einen Typ für kartesische Koordinaten. Die Operationen sind X und Y, die die x- und y-Komponenten eines solchen Objekts liefern, und Eq, mit dem zwei solche Objekte auf Gleichheit geprüft werden.
Signatur: Create: Integer × Integer → Coord X: Coord → Integer Y: Coord → Integer Eq: Coord × Coord → Boolean
Axiome: X(Create(x, y)) := x Y(Create(x, y)) := y Eq(Create(x_1, y_1), Create(x_2, y_2)) := ($x_1 = x_2$) ∧ ($y_1 = y_2$)

Inkrementelle Entwicklung

Die Spezifikation COORD kann man in der Spezifikation eines Cursors für eine graphische Oberfläche benutzen:

CURSOR
sort Cursor **imports** INTEGER, COORD, BITMAP
Ein Cursor ist eine Darstellung einer Bildschirmposition. **Create** assoziiert ein Icon mit dem Cursor an einer Bildschirmposition. **Position** gibt die aktuellen Cursorkoordinaten zurück. **Translate** bewegt den Cursor um eine gegebene Strecke in x- und y-Richtung.
Signatur: Create: Coord × Bitmap → Cursor Translate: Cursor × Integer × Integer → Cursor Position: Cursor → Coord
Axiome: Translate(Create(c, i), dx, dy) := Create(COORD, Create(X(c) + dx, Y(c) + dy), i) Position(Create(c, i)) := c Position(Translate(c, dx, dy)) := COORD.Create(X(c) + dx, Y(c) + dy)

Inkrementelle Entwicklung

Die Spezifikation COORD kann man in der Spezifikation eines Cursors für eine graphische Oberfläche benutzen:

CURSOR

- Man beachte, dass die in COORD definierten Operationen direkt angesprochen werden können, wenn ihre Namen nicht mit denen in CURSOR kollidieren.
- Wenn dieselben Namen in beiden Spezifikationen vorkommen, dann muss beim importierten Namen der Spezifikationsname mit angegeben werden.

Create: Coord × Bitmap → Cursor
Translate: Cursor × Integer × Integer → Cursor
Position: Cursor → Coord

Axiome:

Translate(Create(c, i), dx, dy) := Create(COORD, Create(X(c) + dx, Y(c) + dy), i)
Position(Create(c, i)) := c
Position(Translate(c, dx, dy)) := COORD.Create(X(c) + dx, Y(c) + dy)

Anreicherung

- Die **Anreicherung** einer Spezifikation erinnert an die Vererbung in der objektorientierten Denkweise.
- Die Operationen und Axiome des Basistyps werden geerbt und somit Teil der Spezifikation.
- Neue Operationen in der Spezifikation können gleichnamige Operationen im Basistyp überschreiben und man kann Operationen hinzufügen oder entfernen.
- Anreicherung ist nicht dasselbe wie der Import einer Spezifikation. Beim Import werden die importierten Typen und Operationen nur zugänglich gemacht, jedoch nicht zum Bestandteil der neuen Spezifikation erklärt.

Anreicherung

- **Beispiel:** Wir definieren eine neue Art von Liste, bei der man Elemente an beiden Enden hinzufügen sowie prüfen kann, ob ein Element in der Liste enthalten ist.
- *New_List* erbt die Operationen und Axiome von *List*; im Prinzip könnte man sie direkt in *NEW_LIST* aufnehmen, wenn man überall, wo *List* steht, *New_List* einsetzt.
- Um die Spezifikation zu vervollständigen, müssen die Zugriffsoperationen *Head*, *Tail* und *Member* für die neue Konstruktionsoperation *Add* definiert werden.
- Des Weiteren muss *Member* für die anderen Konstruktionsoperationen definiert werden.
- Bei der Anreicherung werden auch die generischen Parameter des Basistyps vererbt. Diese müssen im angereicherten Typ die Operationen enthalten, die sie auch beim Basistyp hatten.
- Wir fordern zusätzlich noch eine Vergleichsoperation (==).

Anreicherung

NEW_LIST (Elem: [Undefined → Elem; . == . → Boolean])

sort New_List **enriches** List
imports INTEGER, BOOLEAN

Definiert eine erweiterte Liste, die die Operationen und Eigenschaften der einfacheren Liste erbt. Die Operationen **Add** und **Member** werden hinzugefügt: **Add** fügt ein Element an den Anfang der Liste an, **Member** prüft, ob ein Element der Liste gleich dem zweiten Parameter ist.

Signatur:
Add: New_List × Elem → New_List
Member: New_List × Elem → Boolean

Axiome:
Add(Create, v) := Cons(Create, v)
Member(Create, v) := FALSE
Member(Add(L, v), v´) := (v == v´) ∨ Member(L, v´)
Member(Cons(L. v), v´) := (v == v´) ∨ Member(L, v´)
Head(Add(L, v)) := v
Tail(Add (L, v)) := L
Length(Add(L, v)) := Length(L) + 1

Mehrere Rückgabewerte

- In den vorhergehenden Beispielen wurden alle Operationen so definiert, dass sie einen einzigen Wert als Ergebnis zurückgeben. Es gibt aber einige Arten von Operationen, die mehrere Werte verändern.
- Die bekannte Operation **Pop** für Stapel gibt z.B. den obersten Wert des Stapels zurück – modifiziert aber auch den Stapel selbst.
- Solche Operationen kann man durch mehrere einfachere Operationen ausdrücken. Es ist jedoch natürlicher Operationen zu definieren, die anstelle eines einzelnen Werts ein *Tupel* zurückliefern.
- Die Operation Pop könnte zum Beispiel als

  Pop: Stack → (Elem, Stack)

  spezifiziert werden.

Fehlerspezifikation

- Ein Problem, das sich beim Entwickeln von Spezifikationen stellt, ist, wie man Fehler und Ausnahmebedingungen anzeigt.
- Unter normalen Umständen ist das Ergebnis einer Operation von einem Typ X, aber unter gewissen Bedingungen sollte ein Fehler angezeigt werden. Der passende Fehleranzeiger ist aber vielleicht nicht vom Typ X, so dass ein Typfehler resultiert.
- Es gibt mehrere Möglichkeiten, dies anzugehen:
  - Man erweitert den Rückgabetyp um den ausgezeichneten Wert *Undefined*, und lässt die betreffende Operation im Fehlerfall diesen zurückliefern (siehe das Array-Beispiel).
  - Die Operation kann ein Tupel zurückliefern, von dem eine Komponente anzeigt, ob die Operation erfolgreich war oder nicht.
  - Die Spezifikation kann einen Ausnahmen-Abschnitt (*Exceptions*) enthalten, der angibt, wann die Spezifikationen (*Axiome*) nicht gelten.

Fehlerspezifikation

LIST(Elem)
sort List **imports** INTEGER
Signatur: Create: → List Cons: List × Elem → List Tail: List → List Head: List → Elem Length: List → Integer
Axiome: Head(Cons(L, v)) := **if** L = Create **then** v **else** Head(L) Length(Create) := 0 Length(Cons(L, v)) := Length(L) + 1 Tail(Create) := Create Tail(Cons(L, v)) := **if** L= Create **then** Create **else** Cons(Tail(L), v) **Exceptions:** Length (L) = 0 ⇒ **failure**(Head(L))

Fazit

- Vorteile der algebraischen Spezifikation:
  - Es ist ein formales Verfahren mit exakter mathematischer Semantik.
  - Viele Konsistenzprüfungen sind automatisierbar.
  - Oft sind diese Spezifikationen maschinell übersetzbar.
  - Testfälle können automatisch aus der Spezifikation erzeugt werden.
- Neben den Vorteilen existieren bei der algebraischen Spezifikation jedoch auch einige Nachteile:
  - Eine generelle Verwendbarkeit ist nicht gegeben (z.B. bei Realzeitsystemen oder Benutzerschnittstellen).
  - Die Anwendbarkeit auf wirklich große Probleme ist nach wie vor umstritten.
  - Die Lernkurve zur erfolgreichen Anwendung ist steiler als bei anderen Verfahren.