Dipl.-Kfm.
Friedrich Kiltz

Objektorientierte Analyse und Design

Celecta GmbH

für den Ausbildungsverbund

Inhalt

1. Grundlagen der Objektorientierung

1.1. Analyse und Design mit der UML

Inhalte

- Grundlagen der Objektorientierung
  - Objekte, Klassen
  - Abstraktion
  - Kapselung
  - Vererbung

- Überblick über Unified Modeling Language (UML)
  - Modelle, Sichten und Diagramme
  - Beschreibung von Systemanforderungen
    - Anwendungsfall
    - Aktivitätsdiagramm
  - Statische Sicht auf ein System
    - Klassen, Objekte und ihre Relationen
      Klassendiagramm
  - Dynamische Sicht auf ein System
    - Sequenzdiagramm

- Praxisbeispiel für die Umsetzung OOAD in Java
- Nutzung von UML Tools

2. Grundlagen

2.1. Objekte und Klassen

Dieses Kapitel beschreibt die Begriffe:

- Objekt, Klasse
- Eigenschaft, Methode
- Instanz

Ausgangssituation

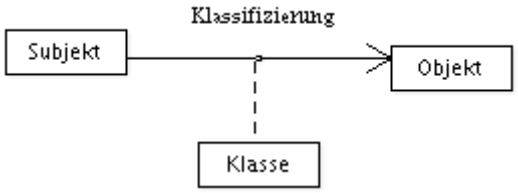
Im Mittelpunkt des objektorientierten Modells steht die Übereinkunft, dass

- gewisse Subjekte (Gegenstände, Vorgänge)
- in unserem Sprachgebrauch
- die gleichen Eigenschaften haben.

z. B.:

- Ein Fahrzeug mit 4 *Rädern*, einer *Karosserie* und *Motor* ist ein PKW.
- Ein Pilz mit *Poren* auf der Unterseite des Hutes ist ein Röhrling und somit meist essbar.

Durch diese Übereinkunft können die **Subjekte** *klassifiziert* werden. Ist ein Subjekt *klassifiziert* worden, so spricht man von einem **Objekt**.

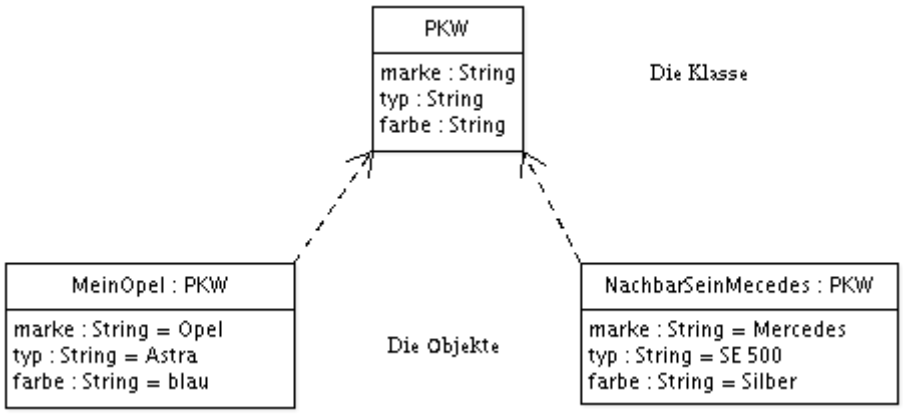


Eigenschaften können hierbei selbst komplexe Objekte (z. B. Motor) sein oder auch ganz primitive Werte wie die **Länge in cm** oder der **Name** einer Person. Man sollte sich auf die Eigenschaften beschränken, die eine eindeutige Klassifizierung zulassen. So wäre für die Klasse **Bank** nicht die Eigenschaft *Ort*, sondern die Eigenschaft *BLZ* geeignet, um zwischen einer Parkbank und einem Geldinstitut zu unterscheiden.

Ein Objekt ist eine Instanz einer Klasse

Die Klasse ist ein Bauplan gleichartiger Objekte, der lediglich die Eigenschaften beschreibt, ohne Ihnen Werte zuzuweisen. Das Objekt ist eine reale *Ausprägung* einer Klasse, man sagt auch: ein Objekt ist eine **Instanz** einer Klasse, die unsere Eigenschaften mit den entsprechenden Werten füllt.

Z. B. definiert die Klasse "PKW" eine Eigenschaft "Farbe", erst ein Objekt dieser Klasse (ein ganz spezieller PKW) weist dieser Eigenschaft "Farbe" einen Wert (z. B. "blau") zu.



UML-Anm.
Objekte können auch in der Notation **Objekt : Klasse** dargestellt werden. Die Klasse **PKW** definiert die Eigenschaften

- **marke** (als Zeichenkette)
- **typ** (als Zeichenkette)
- **farbe** (auch als Zeichenkette)

Die Objekte weisen diesen Eigenschaften Werte zu.

MeinOpel und *NachbarSeinMercedes* sind **Instanzen** der Klasse PKW

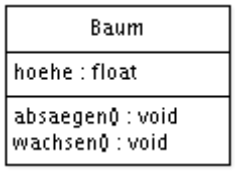
Methoden

Die Veränderung der Werte einer Eigenschaft werden

- von dem Objekt selbst oder
- durch eine Operation, die man auf ein Objekt ausführen kann

vorgenommen.

Diese Operationen nennt man Methoden.



So wird z. B. die Höhe eines Baumes durch die Operationen wachsen und absägen geändert.

UML-Anm.
Name, Eigenschaften und Methoden werden durch horizontale Linien voneinander getrennt.

Zusammenfassung

Klasse
Somit ist eine Klasse die Beschreibung einer Struktur (Eigenschaften) und des Verhaltens (Methoden) einer Menge gleichartiger Objekte.

Objekt
Ein Objekt ist eine Instanz (Ausprägung) einer Klasse. Das Objekt verhält sich entsprechend den Vorgaben der Klasse.

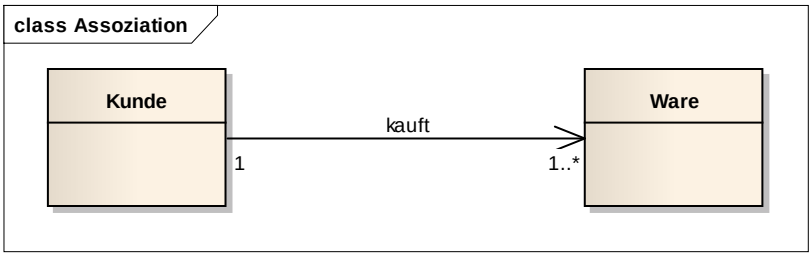
2.2. Beziehungen zwischen Klassen

Dieses Kapitel beschreibt die Begriffe

- Assoziation
- Aggregation
- Komposition

Assoziation

Ganz allgemein gesprochen ist eine Assoziation eine **Beziehung** zwischen zwei Objekten.
Der Mensch **pflanzt** einen Baum und der Kunde **kauft** Waren.



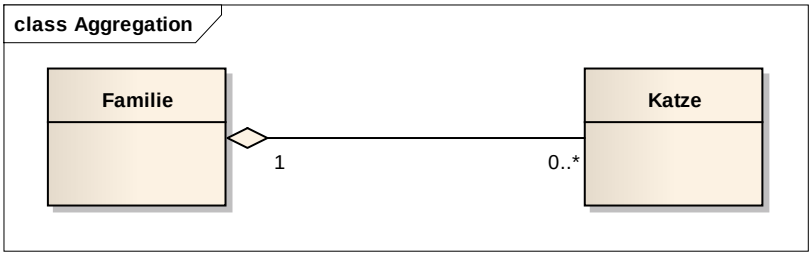
Ein Kunde kauft mindestens eine Ware.

UML-Anm.
Eine Assoziation wird durch eine Linie dargestellt. Optional kann ein Pfeil für die Leserichtung angegeben werden. Sofern bekannt oder für die Betrachtung relevant, kann man **Multiplizitäten** angeben.

In obiger Darstellung kennt der *Kunde* die Klasse *Ware*. Umgekehrt "kennt" die Ware den Kunden nicht.

Aggregation

Die Aggregation ist eine Sonderform der Assoziation, die darin besteht, dass die Einzelteile der Klasse andere Klassen darstellen. Man spricht hier von einer **Teile-Ganzes-Hierarchie**. Im Sprachgebrauch nutzt man für diese Art der Beziehung **"besteht aus"** oder **"hat"**.



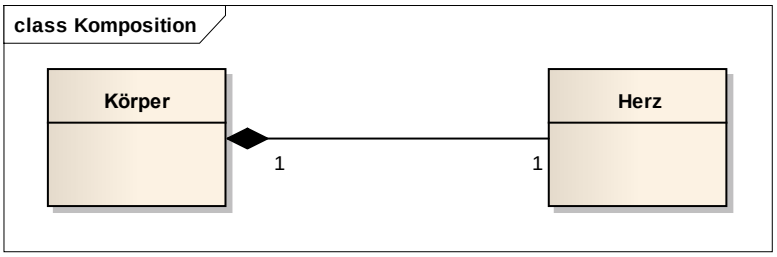
Eine Familie **hat** eine unbestimmte Anzahl Katzen.

UML-Anm.
Eine Aggregation wird durch eine Linie dargestellt, die bei dem *Ganzen* in einer nicht ausgefüllten Raute endet. Auch hier kann man Leserichtung und Multiplizitäten angeben.

Die Einzelteile übernehmen dabei Funktionen für das *Ganze* und tragen dazu bei, dass das *Ganze* funktioniert. vgl.: Oestereich S. 50 ff.

Komposition

Die Komposition ist eine Sonderform der Aggregation. Der Diskriminator besteht in der **existentiellen Abhängigkeit** der Teile.



Ein Körper **hat** ein Herz.

UML-Anm.
Eine Komposition wird durch eine Linie dargestellt, die bei dem *Ganzen* in einer ausgefüllten Raute endet. Auch hier kann man Leserichtung und Multiplizitäten angeben.

Ein Herz ohne seinen Körper wird recht zügig und nachhaltig seine Aufgabe nicht mehr wahrnehmen können (Transplantationen ausgenommen).

Die Unterscheidung zwischen der Aggregation und der Komposition basiert auf der Frage, ob das *Teil* von anderen Klassen genutzt wird oder alleine existieren kann. Dabei ist auch die Perspektive entscheidend, so ist ein Motor für einen PKW-Nutzer ein existensabhängiges Teil eines PKW, wohin gehend ein Automechaniker sich auch andere Verwendungen für den Motor denken könnte. Dennoch bleibt der Motor ein Teil eines Ganzen.

Unterscheidung Aggregation und Komposition

Die Entscheidung zwischen Aggregation u d Komposition kann auch Ihre Tücken haben und hängt zum Teil von der Betrachtungsweise und dem Kontext ab. Als Faustregel gilt: **Wenn ein Objekt dieser Klasse jemals alleine existieren soll, nutze man die Aggregation.** Zwei Eigenschaften von Objekten helfen auch diese Entscheidung zu treffen:

1. Die Lebenszeit eines Objektes:

Die Lebenszeit ist die Zeitspanne zwischen Erzeugung und Zerstörung des Objektes. Bei der **Aggregation** sind die Lebenszeiten **unabhängig**, d. h. ein Druckbleistift existiert weiter, auch wenn ein Miene aufgebraucht (also *zerstört*)ist. Bei der Komposition ist die Lebenszeit gleich, d. h. eine Bleistiftmiene hat ausgedient, wenn ein Bleistift zerstört ist.

2. Die Verwaltung eines Objektes:

Die Frage der Verwaltung von Objekten hängt mit den "Besitzverhältnissen" zusammen. Zum Beispiel kann eine Bibliothek einige Bücher verwalten, die Bibliothek ist damit "Besitzer" der Bücher.

Bei der **Aggregation** kann der Besitz eines Objektes **weitergegeben** werden. So kann aus obigem Beispiel ein Buch von einer Person ausgeliehen werden, also der Besitz an dem Buch von der Bibliothek zu der Person übergehen.

Bei der **Komposition** sind die Besitzverhältnisse **kein Thema**. Die Objekte werden zusammen erstellt und zerstört. Ein Besitzerwechsel findet nicht statt.

2.3. Abstraktion

Abstraktion ist einer der grundlegenden Eigenschaft der Objektorientierung.

Abstraktion durch den Blickwinkel

Stellen Sie sich vor, ein Waldarbeiter, ein Biologe und ein Künstler betrachten einen Baum.

Der Waldarbeiter betrachtet den Baum unter den Gesichtspunkten von Qualität und Menge des Holzes und den Besonderheiten, die beim Fällen des Baumes beachtet werden müssen.

Der Biologe betrachtet die Auswirkungen des Baumes für das umgebene Ökosystem, welche Tiere nisten auf dem Baum, wie gesund ist der Baum.

Der Künstler schaut sich den Baum unter den Gesichtspunkten Farbe, Form und Ausdruck an.

Alle Drei betrachten das gleiche Objekt und alle Drei arbeiten unterschiedliche Eigenschaften heraus und alle Drei beschreiben den Baum "richtig".

Abstraktion ist eine Methode, bei der unter einem bestimmten Gesichtspunkt die wesentlichen Merkmale eines Gegenstandes oder Begriffes heraus gesondert werden.
(http://www.oose.de/glossar)

Unterschiedliche Abteilungen eines Unternehmens betrachten z. B. den Kunden unter verschiedenen Perspektiven:

- Der Verkauf
  - Name, Vorname, Anrede
  - Einkommen
  - Kaufverhalten
- Die Retouren-Abteilung
  - Name, Vorname, Anrede
  - Rücksendungen
  - Häufigkeiten von Bestellungen und Rücksendungen
- Buchhaltung
  - Name, Vorname, Anrede
  - Offene Rechnungen
  - Kreditwürdigkeit
  - Zahlungsverhalten

Einige Eigenschaften und Methoden sind gleich und müssen auch für die unterschiedlichen Abteilungen die gleiche Bedeutung haben.

Abstraktion durch Veränderung der Detaillierung

Neben der Abstraktion durch Fokussierung auf den Blickwinkel kann man auch durch die Detaillierung eine Abstraktion vornehmen.

Hierbei ist zu beachten, in welcher Situation ich mich befinde. Möchte ich die Zusammenhänge von bestimmten Klassen darstellen, mag es reichen ein paar wesentliche Eigenschaften ohne Typ-Bezeichnung und Zusicherungen zu betrachten. Möchte ich die gleiche Klasse unter dem gleichen Blickwinkel für die Implementation darstellen benötige ich den Datentyp und die Zusicherungen, sowie die anderen notwendigen Eigenschaften und Methoden, die man zur Unterstützung der Übersichtlichkeit weggelassen hat.

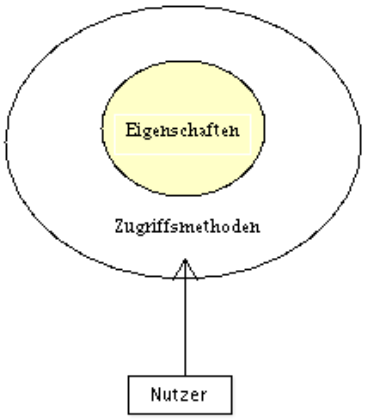
2.4. Kapselung

Definition

Datenkapselung (Programmierung) aus Wikipedia, der freien Enzyklopädie

Als Datenkapselung oder Einkapselung (englisch: encapsulation, nach David Parnas auch bekannt als information hiding) bezeichnet man in der Programmierung das Verbergen von Implementierungsdetails. Der direkte Zugriff auf die interne Datenstruktur wird unterbunden und erfolgt statt dessen über definierte Schnittstellen. Z.B. ist eine Deklaration von Daten nur innerhalb eines Programmmoduls eine Form von Kapselung.

Kapselung ist auch ein wichtiges Konzept der objektorientierten Programmierung. Als Kapselung bezeichnet man den kontrollierten Zugriff auf Methoden bzw. Attribute von Klassen. Klassen können den internen Zustand anderer Klassen nicht in unerwarteter Weise lesen oder ändern. Eine Klasse hat eine Schnittstelle, die darüber bestimmt, auf welche Weise mit der Klasse interagiert werden kann. Dies verhindert das Umgehen von Invarianten des Programms. Vom Innenleben einer Klasse soll der Verwender (gemeint sind sowohl die Algorithmen, die mit der Klasse arbeiten, als auch der Programmierer, der diese entwickelt) möglichst wenig wissen müssen (Geheimnisprinzip). Durch die Kapselung werden nur Informationen über das "Was" einer Klasse (was es leistet) nach außen sichtbar, nicht aber das "Wie" (die interne Repräsentation). Dadurch wird eine Schnittstelle nach außen definiert und zugleich dokumentiert.



Für die Kapselung verwendete Zugriffsarten

Die UML als De-facto-Standardnotation erlaubt die Modellierung folgender Zugriffsarten (in Klammern die Kurznotation der UML):

public (+) zugreifbar für alle Ausprägungen (auch die anderer Klassen),

private (-) Nur für Ausprägungen der eigenen Klasse zugreifbar,

protected (#) Nur für Ausprägungen der eigenen Klasse und von Spezialisierungen derselben zugreifbar,

package (~) erlaubt den Zugriff für alle Elemente innerhalb des eigenen Pakets.

Die Möglichkeiten zur Spezifizierung der Zugriffsmöglichkeiten sind je nach Programmiersprache unterschiedlich.

Vorteile der Kapselung

- Da die Implementierung einer Klasse anderen Klassen nicht bekannt ist, kann die Implementierung geändert werden, ohne die Zusammenarbeit mit anderen Klassen zu beeinträchtigen.
- Erhöhte Übersichtlichkeit, da nur die öffentliche Schnittstelle einer Klasse betrachtet werden muss.
- Es wird verhindert, dass innere Zusammenhänge, die möglicherweise später einmal verändert werden, Änderungen in anderen Programmteilen erfordern.
- Beim Zugriff über eine Zugriffsfunktion spielt es von außen keine Rolle, ob diese Funktion 1:1 im Inneren der Klasse existiert, das Ergebnis einer Berechnung ist, oder möglicherweise aus anderen Quellen (z. B. einer Datei oder Datenbank) stammt.
- Deutlich verbesserte Testbarkeit, Stabilität und Änderbarkeit der Software bzw. deren Teile (Module).

Nachteile der Kapselung

- In Abhängigkeit vom Anwendungsfall Geschwindigkeitseinbußen durch den Aufruf der Methode (direkter Zugriff auf die Datenelemente wäre schneller)

Beispiel

Gehen wir davon aus, dass eine Klasse Konto einen Kontostand und einen Dispolimit hat. Weiterhin gibt es die Methoden auszahlen und einzahlen.

Dadurch, dass die Eigenschaften *private* sind, kann der Nutzer der Klasse die Inhalte nicht direkt setzten und somit können wir verhindern, dass das das Konto schamlos überzogen wird, bzw jemand von außen den Kontostand auf 1.000.000 setzt.

Um den Dispolimit zu setzen bieten wir die Zugriffsmethoden

- `public void setDispolimit(float betrag)`
- `public float getDispolimit()`

Bei dem Kontostand bieten wir nur den *Getter*

- `public float getKontostand()`

2.5. Vererbung, Abstrakte Klassen, Schnittstellen

Dieses Kapitel beschreibt die Begriffe

- Vererbung und seine Sonderformen
  - Abstrakte Klasse
  - Schnittstelle

Vererbung

Im täglichen Leben helfen wir uns damit, dass wir Klassen über die Vererbung beschreiben.

Auf die Frage:

"Was ist ein Cabriolet?"

können wir folgende Antwort geben:

*"Ein Cabriolet **ist ein** Auto, bei dem man das Verdeck öffnen kann."*

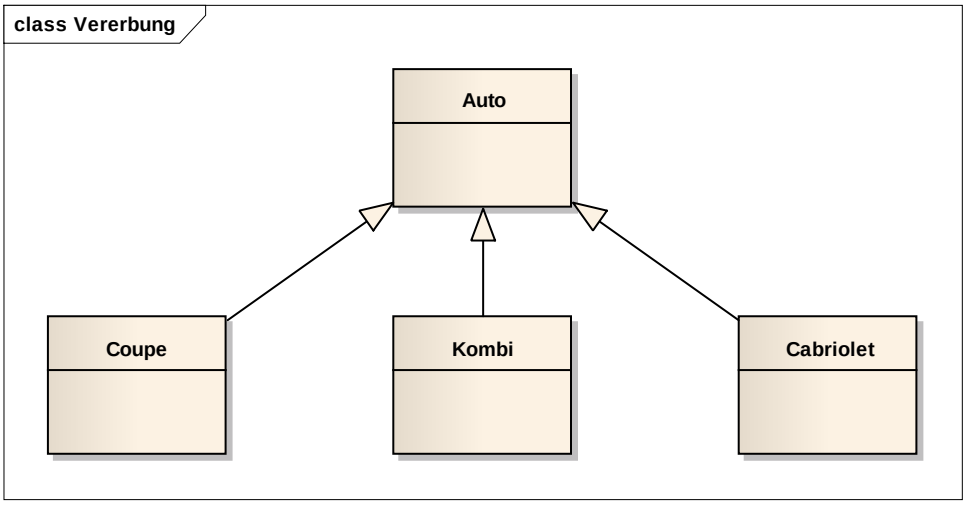
Eine Vererbung stellt immer eine *"ist ein"*-Beziehung dar. Folgende Begriffe sind hierbei Entscheidend:

- **Superklasse** ist die Klasse die vererbt. (hier: Auto)
- **Subklasse** ist die Klasse die erbt. (hier: Cabriolet)
- **Ableitung** ist der Vorgang der Vererbung . (hier: *Cabriolet* ist abgeleitet von *Auto*)
- **Diskriminator** ist das Merkmal, nach dem abgeleitet wird. (hier: *"bei dem man das Verdeck öffnen kann"*)

Wichtige Regeln für die Vererbung:

- Alle **Eigenschaften** der Superklasse müssen in der Subklasse enthalten sein.
- Eine Einschränkung des Wertebereiches ist möglich! (Zusicherung)
- Alle **Methoden** der Superklasse müssen in der Subklasse enthalten sein.
- Der Ablauf innerhalb der Methode kann sich verändern (dynamischer Polymorphismus)
- Subklassen können **zusätzliche** Eigenschaften und Methoden erhalten

Beispiel:

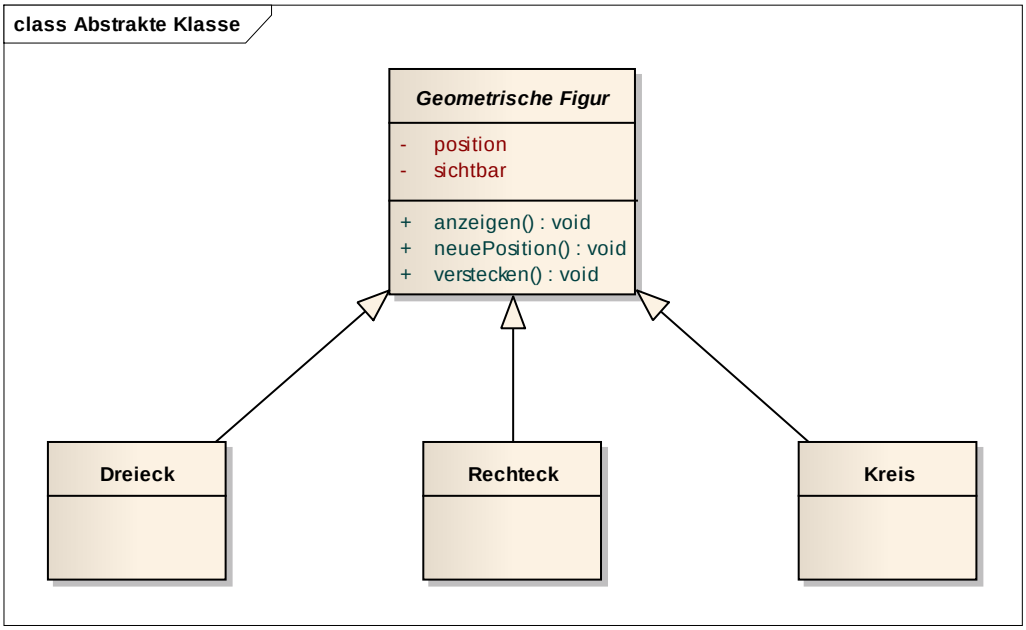


UML-Anm.
Die Vererbung wird mit einem Dreieck zur Superklasse dargestellt.

Abstrakte Klasse

Eine abstrakte Klasse ist eine Superklasse, aus der man kein Objekt erzeugen kann. Ihre Existenzberechtigung ergibt sich aus der Zusammenfassung gleichartiger Elemente für eine Gruppe von Subklassen.

Betrachten wir folgendes Beispiel:



Die geometrische Figur als solches können wir nicht in einer realen Ausprägung darstellen. Sie ist aber dahingehend hilfreich, dass mit ihr die Gemeinsamkeiten für die Subklassen behandelt werden können. Somit brauchen wir uns bei den Subklassen nur auf die "neuen" Anforderungen zu konzentrieren. (z. B. den Radius für den Kreis)

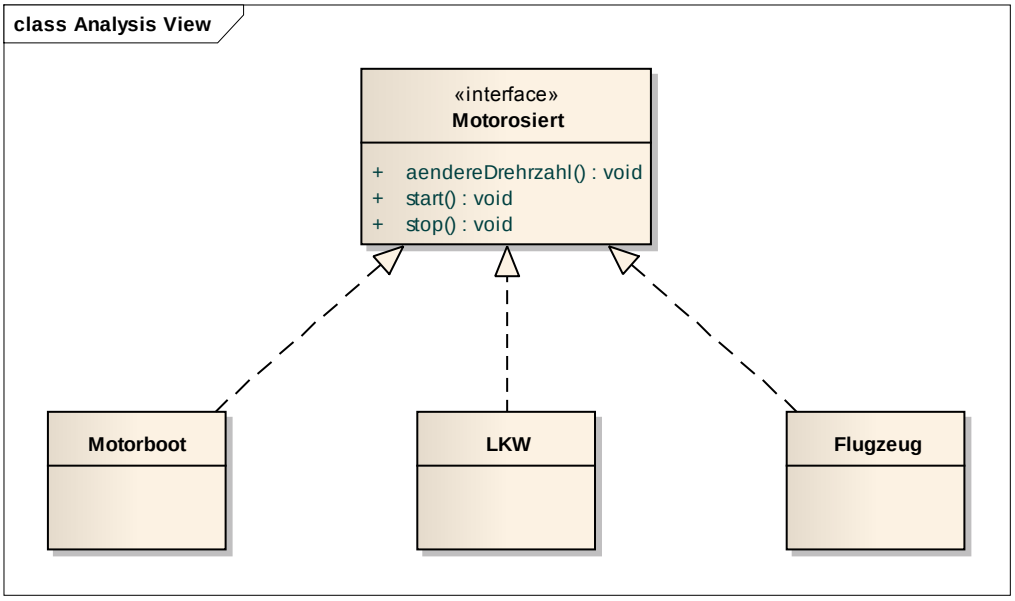
UML-Anm.
Bei der abstrakten Klasse wird der Name *kursiv* dargestellt.

Schnittstelle

Eine Schnittstelle (oder auch "Interface") ist eine Klasse, die nur Namen für bestimmte Methoden vorgibt, ohne diese in irgend einer Art auszuführen.

Im Gegensatz zur abstrakten Klasse werden in der Schnittstelle höchstens Konstanten als Eigenschaften definiert.

Aufgabe einer Schnittstelle ist es *"Verträge"* mit Klassen zu schließen. Ich kann bestimmte Methoden aufrufen und bekomme eine bestimmte Leistung. Die Art der Leistungserstellung ist der Klasse vorbehalten, die die Schnittstelle **implementiert** (oder auch realisiert), d. h. eingebunden hat.



Die Klassen **Motorboot**, **LKW** und **Flugzeug** *realisieren* alle die Schnittstelle **Motor**. Somit weiß jeder, dass man z. B. ein Flugzeug **starten** kann, auch ohne zu wissen, wie das wirklich funktioniert.

UML-Anm.
Schnittstellen werden als Kreis oder mit dem Stereotyp «Interface» dargestellt. Klassen, die eine Schnittstelle realisieren werden mit einer gestrichelten Linie verbunden, ein Dreieckspfeil zeigt auf die Schnittstelle.

3. UML

3.1. Welche Diagramme in welchen Disziplinen genutzt werden.

1. Aufgabenbeschreibung

Aufgabe der Disziplin:
Darstellung der Vorgaben des Auftraggebers

Genutzte Modelle:

- Freitext
- Klassendiagramme (konzeptionell) evtl. erweitert durch Verhaltensdiagramme

2. Anwendungsfallanalyse

Aufgabe der Disziplin:
Ermittlung der Sicht der künftigen Anwender

Genutzte Modelle:

- Anwendungsfalldiagramme incl. Beschreibung der Awf.

3. Systemdesign

Aufgabe der Disziplin:
Erste Designentscheidungen auf hohem Abstraktionsniveau

Genutzte Modelle:

- Erzeugung von Paketen
- evtl. Ergänzung der Awf. durch Aktivitäts- und Sequenzdiagramme

4. Designmodell

Aufgabe der Disziplin:
Weiterführung des Geschäftsmodells mit Hilfe der vorherigen Phasen

Genutzte Modelle:

- Klassendiagramme
- Sequenzdiagramme
- Kollaborationsdiagramme
- evtl. Zustandsdiagramme

5. Implementationsmodell

Aufgabe der Disziplin:
Erweiterung des Designmodells hinsichtlich seiner Anforderungen bezüglich der gewählten Programmiersprache

Genutzte Modelle:

- Komponentendiagramme
- Deploymentdiagramm
- Weiterentwicklung von
    - Klassendiagrammen
    - Sequenzdiagrammen
    - Kollaborationsdiagrammen
    - evtl. Zusatandsdiagrammen

Klassifizierung der Diagramme

- **Strukturdiagramme** dienen zur Visualisierung, Spezifizierung, Konstruktion und Dokumentation der *statischen Aspekte* eines Systems.
    - Klassendiagramm
    - Komponentendiagramm
    - Objektdiagramm
    - Verteilungsdiagramm

- **Verhaltendsiagramme** dienen zur Visualisierung, Spezifizierung, Konstruktion und Dokumentation der *dynamischen Aspekte* eines Systems.
    - Anwendungsfalldiagramm, zur Organisation des Verhaltens
    - Sequenzdiagramm, fokussiert auf die zeitliche Abfolge von Nachrichten
    - Aktivitätsdiagramm zeigt den Steuerungsfluss von Aktivitäten
    - Zustandsdiagramm fokussiert auf den veränderlichen Zustand eines Systems

3.2. Use-Case-Diagramme

Beschreibung

Ein Use-Case-Diagramm (auch Anwendungsfalldiagramm) beschreibt eine Interaktion eines Akteurs mit dem System um ein bestimmtes Ziel zu erreichen.

Ein "Akteur" muss dabei nicht unbedingt ein Benutzer sein, es kann auch ein anderes System sein.

Besonderheiten

Es gibt zwei Beziehungs-Typen zwischen den Anwendungsfällen

1. *«extend»* eine Erweiterung des Anwendungsfalles oder ein ähnlicher aber weiterführender Fall

2. *«include»* (oder auch "uses" genannt) zeigt an, dass der Anwendungsfall einen anderen Anwendungsfall beinhaltet oder auslöst.

Ablauf

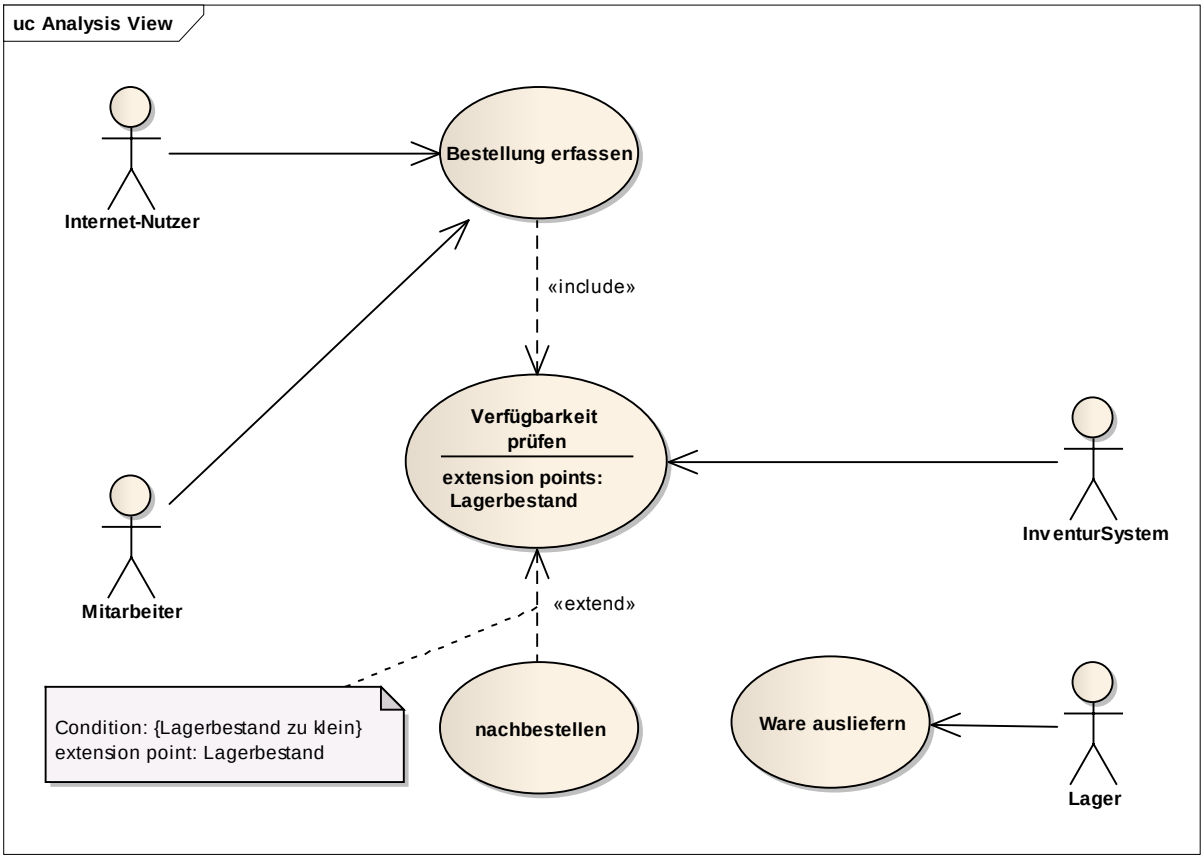
Um ein Use-Case-Diagramm zu erstellen sollte man in folgenden Schritten vorgehen:

1. Indentifiziere alle Anwendungsfälle der Applikation
2. zeichen und benenne alle Aktuere der Applikation
3. zeichen und benenne alle Anwendungsfälle der Applikation
4. Verbinde die entsprechenden Akteure mit den dazugehörigen Anwendungsfällen
5. Schreibe eine kurze Beschreibung zu dem Use-Case

Teilweise hilft es, zuerst die Akteure zu identifizieren und sich dann zu überlegen, was die Akteure tun.

Beispiel

Bestellung und Lieferung von Waren in UML. (stark vereinfacht)



Anm.: OOAD ist eine Kunst und keine Wissenschaft!

Anwendungsfälle

"Ein Anwendungsfall beschreibt anhand eines zusammenhängenden Arbeitsablaufes die Interaktionen mit einem (geschäftlichen oder technischen) System. Ein Anwendungsfall wird stets durch einen Akteur initiiert und führt gewöhnlich zu einem für die Akteure wahrnehmbaren Ereignis." (Quelle: www.oose.de)

Ein Anwendungsfall (Awf oder auch Use Case) beschreibt eine Leistung unserer Software aus der Sicht des Akteurs. Der Akteur ist der Auslöser des Awf. Dies können Personen sein (Interessent, Kunde, Auftraggeber) oder auch andere Systeme. Awf wachsen mit der Realisierung des Projektes. Man kann hierbei bestimmte Phasen unterscheiden:

1. Anwendungsfälle identifizieren
2. Anwendungsfälle essentiell beschreiben
3. Anwendungsfälle ausarbeiten

Bei der **Identifikation** eines Awf benötigen wir folgende Informationen:

Name des Awf	Kurze treffende Bezeichnung, meist bestehend aus einem Verb und einem Substantiv
Kurz-beschreibung	Ein kurzer Satz.
Akteur	Name des(der) Akteur(e), die den Awf auslösen

Die **essentielle** Beschreibung des Awf erweitert die Daten der Identifizierung durch:

Vorbedingung und Auslöser	Aus welcher Situation heraus (oder auch warum) beginnt der Awf?
Nachbedingung und Ergebnis	Welches Ziel (wahrnehmbar für den Akteur) soll erreicht werden?
Ablauf	Darstellung der Schritte im grundlegenden Ablauf ohne Ausnahmen und Fehlersituationen

Bei der **Ausarbeitung** des Awf wird dieser durch weitere Inhalte ergänzt. Die wichtigsten Inhalte dabei sind folgende:

Invarianten	Bedingungen, die stets erfüllt sein müssen
Ausnahmen, Fehlersituation	Beschreibung mit Bezug zu dem Schritt im Ablauf, in dem die Ausnahme auftritt
Regeln	Regelwerk, das im kompletten Awf zu erfüllen ist
Ansprechpartner	Fachbereich, Anwender
Änderungshistorie	Beschreibung der Version
Offene Fragen	Liste zu Abarbeitung
Dokumente, Referenzen, Dialogbeispiele	
Diagramme	Klassen und Ablaufdiagramme

Alle obigen Inhalte sind optional.

3.3. Aktivitäts-Diagramme

Beschreibung

Aktivitäts-Diagramme sind sinnvoll, um die Verbindungen der Objekte mit dem Workflow darzustellen. Sie zeigen weiterhin die parallele Verarbeitung von Methoden.

Pro Aktivitäts-Diagramm wird ein Use-Case zugrunde gelegt.

Besonderheiten

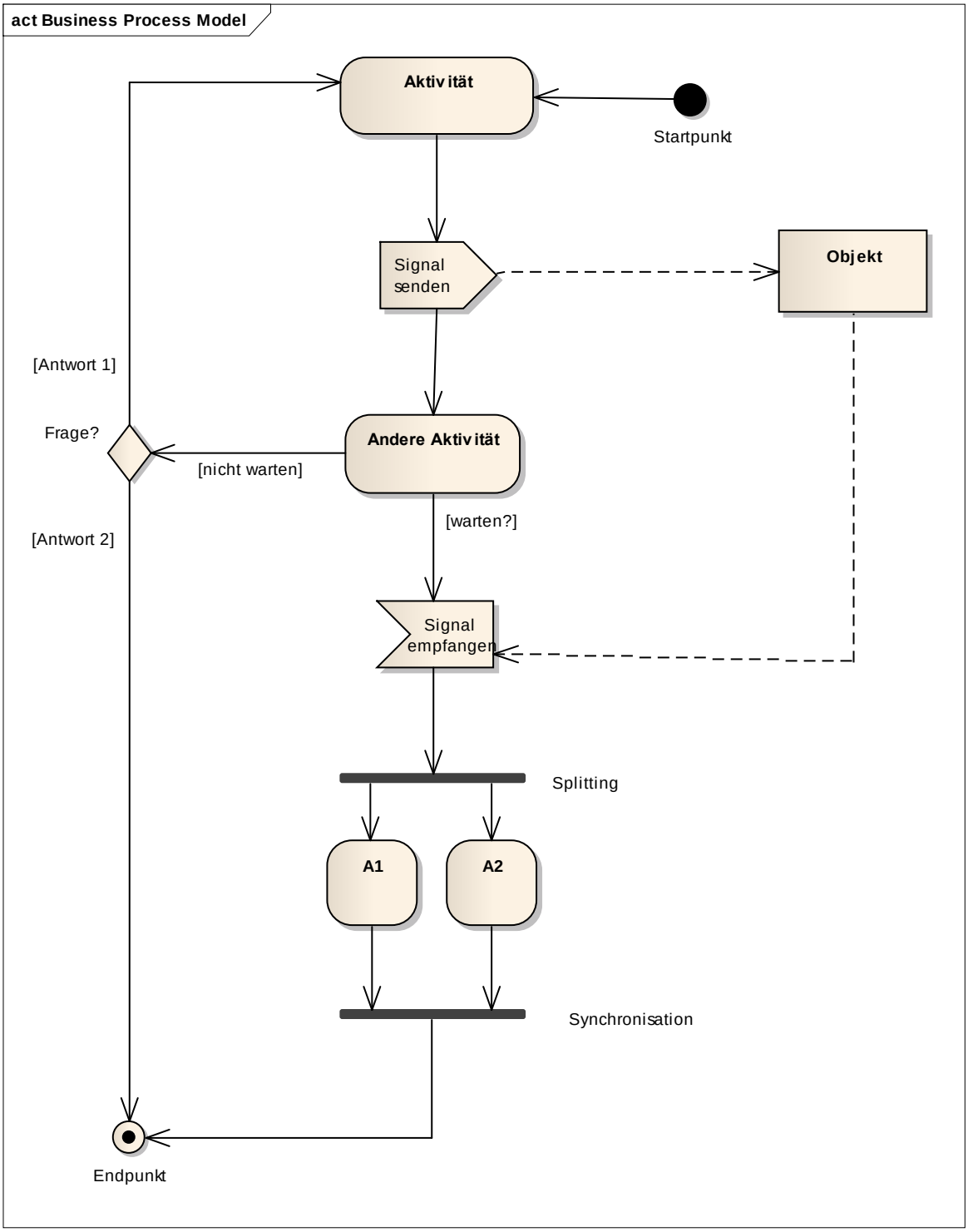
- Eine **Aktivität** repräsentiert hierbei eine Methode oder eine Gruppe von Methoden.
- Die Verbindungen der Aktivitäten, **Transition (** oder Trigger) zeigen welche Aktivität welch andere Aktivität **ansteuert**.
- **Synchronisationslinien** zeigen den gemeinsamen Level bei parallelen Aktivitäten

Ablauf

Um ein Aktivitäts-Diagramm zu erstellen sollte man in folgenden Schritten vorgehen:

1. Wähle **ein** Use-Case
2. Zeichen die Aktivitäten in das Diagramm
3. Füge die Trigger hinzu
4. Füge, sofern notwendig, Bemerkungen oder Bedingungen hinzu
5. Füge, sofern notwendig, Synchronisationslinien hinzu

Die wichtigsten Elemente für Aktivitätsdiagramme sind:



3.4. Klassen-Diagramme

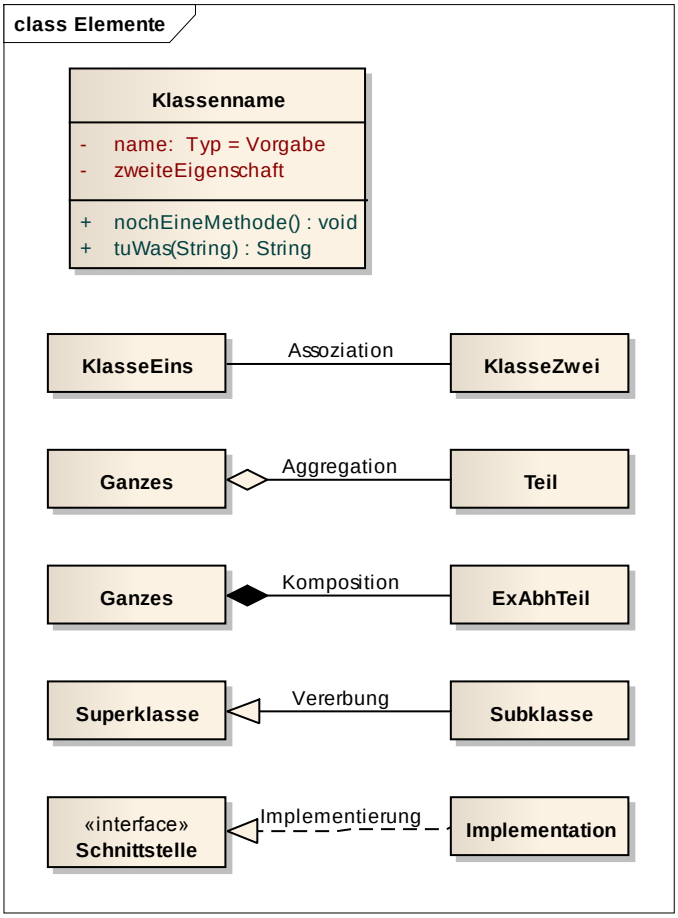
Beschreibung

Ein Klassendiagramm ist die statische Darstellung von Klassen mit deren Beziehungen. Die Ausführlichkeit des Diagrammes hängt von dem Kontext und dem Ziel des Diagrammes ab.

Grundsätzlich kann man zwei Typen von Klassendiagrammen unterscheiden:

- Klassendiagramme für die Konzeption - zur Identifikation und Darstellung der Geschäftsklassen
- Klassendiagramme für die Spezifikation - zur Darstellung der Informationen, die für die Implementierung benötigt werden.

Die wichtigsten Elemente



3.5. Klassen-Diagramme für Konzeption

Beschreibung

Ein Klassendiagramm beschreibt die Objekte einer Applikation und die Beziehungen der Objekte untereinander. Hierbei wird der Softwareimplementierung noch keine große Bedeutung beigemessen, vielmehr zielt dieser Diagrammtyp darauf hin, das Problem in konzeptioneller Weise anzugehen.

Besonderheiten

Die Beziehungen zwischen Objekten beschränken sich auf

- Subtypen (Vererbung) und
- Assoziationen

Dargestellt werden auch folgende Elemente:

- Multiplizitäten, d. h. Wie viele Objekte der einen Klasse mit wie vielen der anderen Klasse Zusammenarbeiten (für einen Use-Case!)
- Eigenschaften

Methoden werden in der konzeptionellen Phase **ausgelassen**.

Ablauf

Um ein Klassen-Diagramme für die Konzeption zu erstellen sollte man in folgenden Schritten vorgehen:

1. Wähle **ein** Use-Case
2. Füge die Objekte und die Variablen zu dem Diagramm
3. Füge die Beziehungen für die Assoziationen ein
4. Füge die Multiplizitäten ein
5. Füge die Beziehungen für die Subtypen ein
6. Wiederhole die Schritte 1 - 5 für alle Use-Case, die zu diesem Klassendiagramm gehören.

3.6. Klassen-Diagramme für Spezifikation

Beschreibung

Dieser Diagramm-Typ ist eine Weiterentwicklung des konzeptionellen Diagrammes. Hierbei werden die Informationen hinzugefügt, die für die Implementierung benötigt werden. Dies sind die **Verantwortlichkeiten** und die **Methoden**.

Besonderheiten

Auf der Grundlage der *Klassen-Diagramme für Konzeption* werden in diesem Diagramm die Veranwortlichkeiten und die Methoden ermittelt.

Die Methoden zur Veränderung der Eigenschaften (set... und get...) werden impliziert und nicht dargestellt.

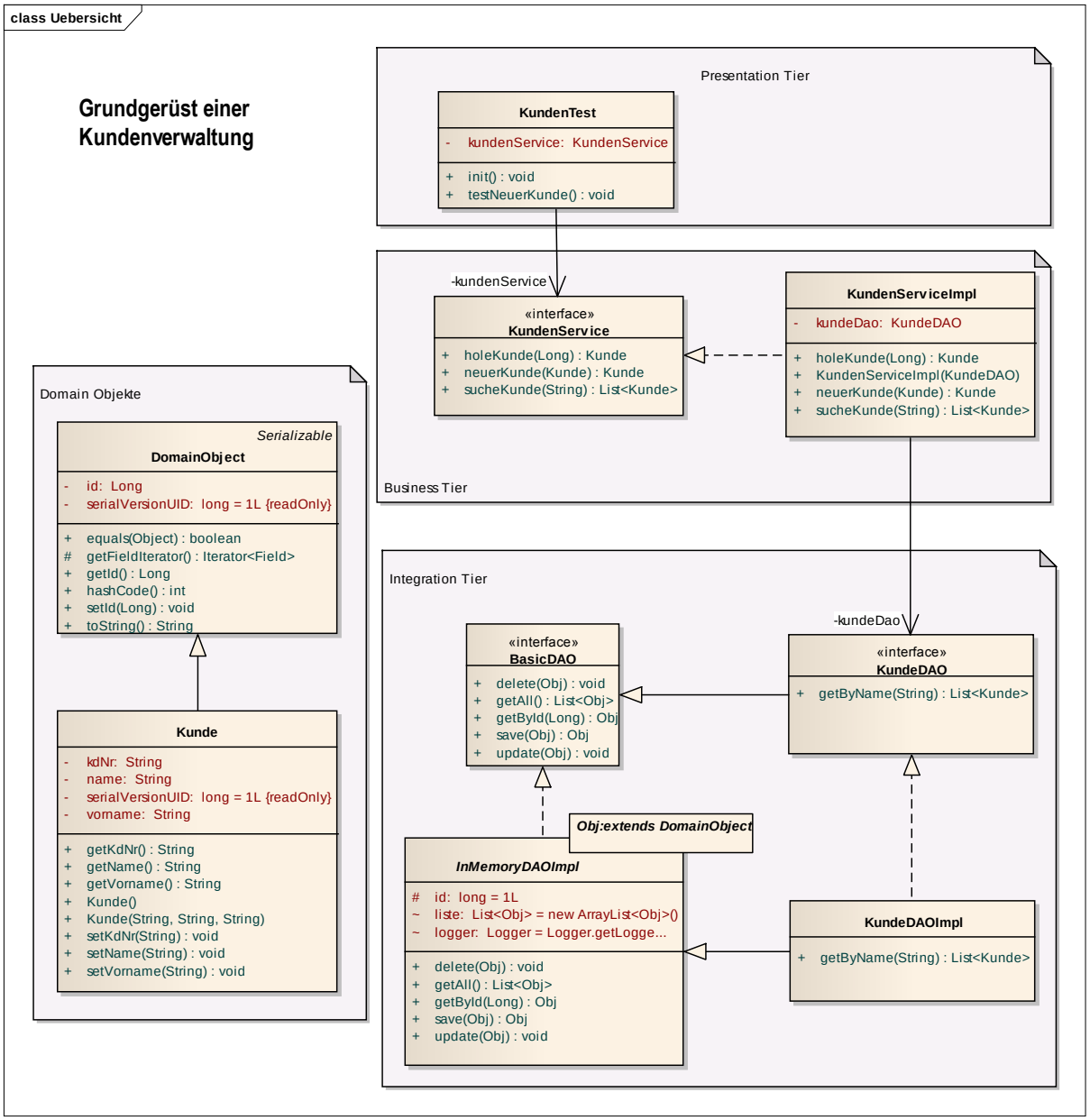
Für die Verantwortlichkeiten gilt die Frage, welches Objekt eine Referenz zu dem anderen Objekt enthält.

Ablauf

Um ein Klassen-Diagramme für die Spezifikation zu erstellen sollte man in folgenden Schritten vorgehen:

1. Wähle ein Klassen-Diagramm aus der Konzeptions-Phase
2. Füge die Pfeile für die Verantwortlichkeiten ein.
3. Füge die Methoden zu den Klassen hinzu (ohne get- und set-Methoden)

Beispiel



Beschreibung: Dieses Diagramm zeigt die Beziehungen und die Verantwortlichkeiten der einzelnen Klassen untereinander. Mit Notizen wurde hier eine Gruppierung vorgenommen. Diese hätte man auch mit unterschiedlichen Farben oder Paketen vornehmen können.

3.7. Sequenz-Diagramme

Beschreibung

Ein Sequenz-Diagram wird benutzt, um die Methoden eines einzelnen Use-Case zu erfassen. Man zeigt damit auf, in welcher Art und Weise eine Gruppe von Objekten miteinander Kommuniziert.

Besonderheiten

Die Lebenszeit eines Objektes (von der Erzeugung bis zur Zerstörung) ist dabei von Bedeutung.

Folgende **Elemente** sind für Sequenz-Diagramme u. a. möglich:

- Nachrichten()
- Antworten
- Zusicherungen { r > 0 }
- Bedingungen [r > 10]
- Konstruktion (Erzeugung)
- Destruktion (Zerstörung)

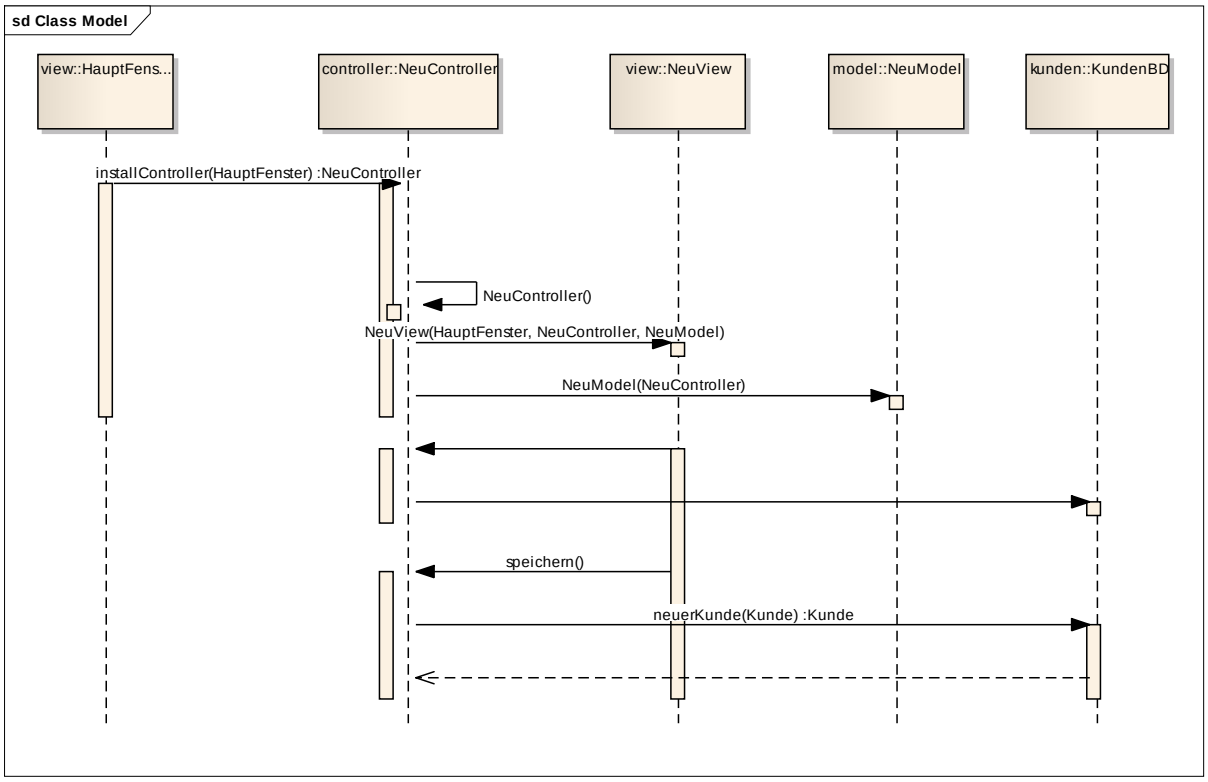
Ablauf

Um ein Sequenz-Diagramm zu erstellen sollte man in folgenden Schritten vorgehen:

1. Wähle **ein** Use-Case
2. Füge das erste Objekt in das Diagramm
3. füge die Methoden des Objektes hinzu und das Objekt das angesprochen wird (sofern es nicht das eigene Objekt ist)
4. Überprüfe ob das zweite Objekt Nachrichten an weitere Objekte schickt
5. Wiederhole die Schritte 3 und 4 solange notwendig
6. Füge die notwendigen Elemente (beschrieben in "Besonderheiten") hinzu

Beispiel

Teilaspekt einer Kundenverwaltung.



4. Beispiel: Realisierung OOAD in Java

Dieses Beispiel zeigt stark vereinfacht den Einsatz von OOP. Im Beispiel wird ein Bild erstellt, auf dem

- 2 Punkte,
- 2 Striche und
- 2 Kreise

vorhanden sind.

Die Klassen

Im folgenden Diagramm sehen Sie folgende Klassen:

1. Bild

Aufgabe der Klasse *Bild* ist es die "Leinwand" für das Bild bereitzustellen und das Bild darzustellen.

- Eigenschaften:
  - Breite und Höhe des Bildes
  - Eine Zeichenfläche. Wir können davon ausgehen, dass die Zeichenfläche von unserer Programmiersprache vorgegeben ist.
  - Das Bild **hat** eine unbestimmte Anzahl von *Elementen*

- Methoden
  - Zwei **Konstruktoren** zur Erzeugung eines Bildes. Konstruktoren haben immer den gleichen Namen wie die Klasse selbst. Der
  - Konstruktor ohne Parameter erstellt ein Bild mit der Größe 200x200 Pixel.
  - Die Methode **add(element)** wird genutzt, um dem Bild ein Bildelement hinzuzufügen.
  - Die Methode **draw()** hat die Aufgabe
    - 1. die "Leinwand" zu erstellen und
    - 2. für jedes Bildelement, welches das Interface *Drawable* implementiert die Methode draw(Bild) aufzurufen. Der Parameter *Bild* ist hierbei die Referenz des Bildes, damit das Element weiß, auf welcher Leinwand er zeichnen muss.
    - 3. Als letzten Schritt stellt *draw()* das Bild dar.

2. Element

Element ist eine abstrakte Klasse, aus der kein Objekt erzeugt wird. Ihre Aufgabe ist es alle Eigenschaften und Funktionalitäten auszuführen, die für die folgenden Klassen gleich sind.

3. Punkt

In der Methode *draw()* wird die sprachspezifische Realisierung des Zeichen-Vorganges implementiert.

4. Strich

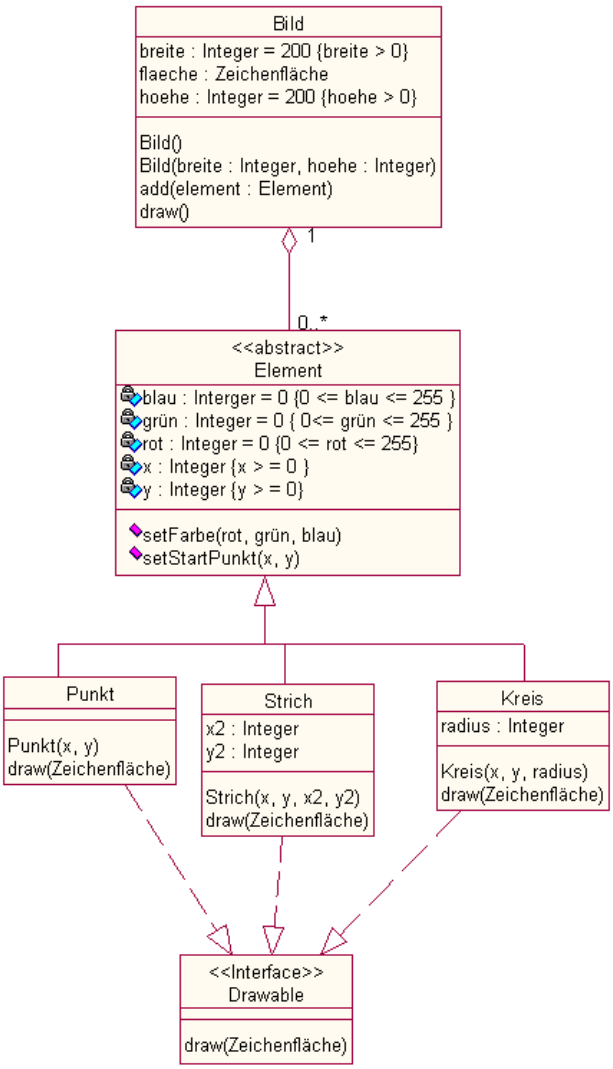
Die Klasse *Strich* benötigt zwei weitere Elemente für den Endpunkt.

5. Kreis

Die Klasse *Kreis* benötigt zusätzlich den Radius.

6. Interface Drawable

Das Interface stellt sicher, dass alle Klassen die es implementieren eine Methode **draw(Zeichenfläche)** bereitstellen. Aus diesem Grunde kann die Methode *draw()* der Klasse *Bild* sicher sein, dass ein Aufruf der Methode möglich ist.



Der Ablauf:

Mit folgendem Ablauf können wir ein Bild mit den zu Anfang besprochenen Elementen erzeugen:

1. Wir erzeugen mit *new* eine Instanz der Klasse Bild: bild = new Bild();
2. Wir erzeugen eine Instanz der Klasse Punkt: p1 = new Punkt(10,10);
3. Optional können wir die Farbe verändern: p1.setFarbe(255,0,0); // rot
4. Wir fügen diesen Punkt dem Bild hinzu: bild.add(p1);
5. Wir erzeugen die zweite Instanz der Klasse Punkt: p2 = new Punkt(100,100);
6. Wir fügen diesen Punkt dem Bild hinzu: bild.add(p2);
7. Wir erzeugen eine Instanz der Klasse Strich: s1 = new Strich(10,10,100,100);
8. Optional können wir die Farbe verändern: s1.setFarbe(255,0,0); // rot
9. Wir fügen diesen Strich dem Bild hinzu: bild.add(s1);
10. ...analog dazu den 2. Strich und die Kreise
11. Zum Abschluss fordern wir das Bild auf alles zu zeichnen: bild.draw();

Realisierung

eine mögliche Realisierung in Java sieht folgendermaßen aus:

Die Klasse Bild

```
 1    import javax.swing.*;
 2    import java.awt.*;
 3    import java.util.Vector;
 4
 5    /**
 6     * Aufgabe der Klasse Bild ist es, die Leinwand für die Elemente
 7     * bereitzustellen und das Bild darzustellen.
 8     */
 9
10    public class Bild extends JFrame
11    {
12      int breite = 200;
13      int hoehe = 200;
14      // Die "Leinwand"
15      Graphics zeichenflaeche;
16      // die Sammlung der Elemente
17      List<Element> elemente = new Hashmap<>();
18
19      // Erzeugung mit Standard-Werten
20     public Bild()
21     {
22       init();
23     }
24     // Erzeugung mit neuer Höhe und Breite
25     public Bild( int breite, int hoehe )
26     {
27       this.breite = breite;
28       this.hoehe = hoehe;
29       init();
30     }
31
32     // Element dem Bild zuordnen
33     public void add( Element e )
34     {
35       elemente.add(e);
36     }
37
38     // Für jedes Element "draw()" aufrufen
39     public void draw()
40     {
41       Drawable e;
42       for ( int i = 0; i < elemente.size(); ++i )
```

```
43        {
44          e = (Drawable) elemente.elementAt(i);
45          e.draw( zeichenflaeche );
46        }
47      }
48
49      // vom System
50      public void paint( Graphics g )
51      {
52        zeichenflaeche = g;
53        draw();
54      }
55
56      // interne Methode
57      private void init()
58      {
59        this.setSize(breite, hoehe);
60        this.setVisible(true);
61        this.setBackground(new Color(255,255,255));
62      }
63    }
```

Die Klasse Element

```
 1    /**
 2     * Abstrakte Klasse für die Gemeinsamkeiten der Bildelemente
 3     */
 4    public abstract class Element
 5    {
 6      // Farbe
 7      int rot = 0;
 8      int gruen = 0;
 9      int blau = 0;
10      // Position
11      int x = 0;
12      int y = 0;
13
14      void setFarbe( int rot, int gruen, int blau )
15      {
16        //Gültigkeitsbereich müßte noch geprüft werden...
17        this.rot = rot;
18        this.gruen = gruen;
19        this.blau = blau;
20      }
21
22      void setStartPunkt( int x, int y )
23      {
24        this.x = x;
25        this.y = y;
26      }
27    }
```

Die Klasse Punkt

```
 1    import Element;
 2    import java.awt.*;
 3    /**
 4     * Zeichen eines Punktes
 5     * Ein wenig dicker, damit er besser gesehen wird
 6     */
 7
 8    public class Punkt extends Element implements Drawable
 9    {
10
11      public Punkt( int x, int y )
12      {
13        this.setStartPunkt(x, y);
```

```
14      }
15
16      public void draw( Graphics zeichenflaeche )
17      {
18        // Farbe setzen
19        zeichenflaeche.setColor( new Color( this.rot, this.gruen,
20        // zeichnen
21        zeichenflaeche.fillRect(this.x, this.y, 4, 4 );
22      }
23    }
```

Die Klasse Strich

```
 1    import Element;
 2    import java.awt.*;
 3    /**
 4     * Erweiterung des Punktes durch die Endkoordinaten
 5     */
 6
 7    public class Strich extends Element implements Drawable
 8    {
 9      int x2 = 0;
10      int y2 = 0;
11
12      public Strich( int x, int y, int x2, int y2)
13      {
14        this.setStartPunkt(x,y);
15        this.x2 = x2;
16        this.y2 = y2;
17      }
18
19      public void draw( Graphics zeichenflaeche )
20      {
21        // Farbe setzen
22        zeichenflaeche.setColor( new Color( this.rot, this.gruen,
23        // zeichnen
24        zeichenflaeche.drawLine(this.x, this.y, this.x2, this.y2);
25      }
26    }
```

Die Klasse Kreis

```
 1    import Element;
 2    import java.awt.*;
 3    /**
 4     * Ausgabe eines Kreises
 5     */
 6    public class Kreis extends Element implements Drawable
 7    {
 8
 9      int radius = 0;
10
11      public Kreis( int x, int y, int radius)
12      {
13        this.setStartPunkt(x, y);
14        this.radius = radius;
15      }
16
17      public void draw( Graphics zeichenflaeche )
18      {
19        // Farbe setzen
20        zeichenflaeche.setColor( new Color( this.rot, this.gruen,
21        // zeichnen
22        zeichenflaeche.drawArc(this.x, this.y,
                        this.radius / 2 , this.radius / 2, 0, 360 );
23      }
24    }
```

Das Interface

```
1    import java.awt.Graphics;
2    /**
3     * Die Schnittstelle
4     */
5    public interface Drawable
6    {
7      public void draw( Graphics zeichenflaeche );
8    }
```

Die Klasse, die das Bild erzeugt

```
 1    /**
 2     * Die Anwendung für die Ablaufsteuerung
 3     */
 4    public class Anwendung
 5    {
 6      public static void main(String[] args)
 7      {
 8        // erzeuge das Bild
 9        Bild bild = new Bild(400,400);
10        // erzeuge 1. Punkt
11        Punkt p1 = new Punkt( 48, 48 );
12        // setze Farbe des Punktes
13        p1.setFarbe(255,0,0);
14        // füge Punkt dem Bild hinzu
15        bild.add( p1 );
16
17        // 2. Punkt
18        Punkt p2 = new Punkt( 248, 248 );
19        p2.setFarbe(0,255,0);
20        bild.add( p2 );
21
22        // 1. Strich
23        Strich s1 = new Strich(50,55,50,250);
24        // setze Farbe des Striches
```

```
25        s1.setFarbe(255,0,0);
26        // füge Strich dem Bild hinzu
27        bild.add( s1 );
28
29        // 2. Strich
30        Strich s2 = new Strich(50,250,245,250);
31        // setze Farbe des Striches
32        s2.setFarbe(0,255,0);
33        // füge Strich dem Bild hinzu
34        bild.add( s2 );
35
36        // 1. Kreis
37        Kreis k1 = new Kreis(200,200,50);
38        // setze Farbe des Striches
39        k1.setFarbe(255,0,0);
40        // füge Strich dem Bild hinzu1
41        bild.add( k1 )2;
42
43        // 2. Kreis
44        Kreis k2 = new Kreis(100,100,50);
45        // setze Farbe des Striches
46        k2.setFarbe(0,255,0);
47        // füge Strich dem Bild hinzu
48        bild.add( k2 );
49
50        // bild malen
51        bild.repaint();
52      }
53    }
```