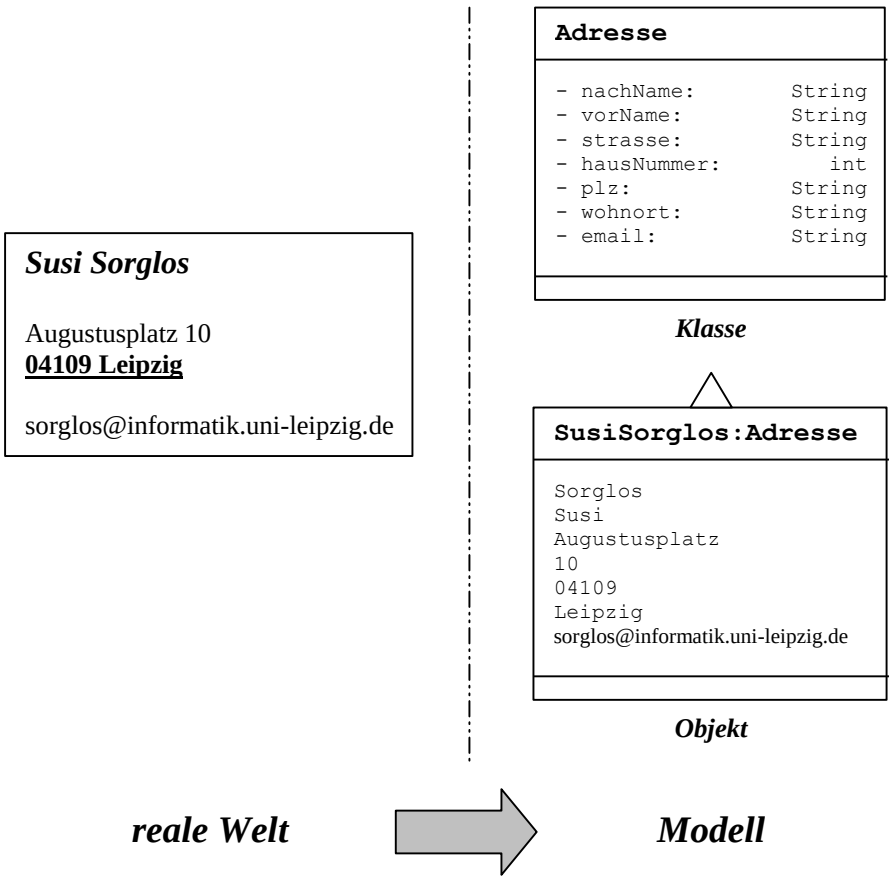
Inhalt

10Modellierung

Ziel der objektorientierten Programmierung ist es, einen Ausschnitt des Weltbildes auf einen Computer zu transferieren. Diesen Vorgang nennt man **Modellierung**. Man erhält ein **Modell**, bestehend aus mehreren miteinander kommunizierenden **Modulen**. Jedes solches *Modul* wird in Form von *Klassen* realisiert, aus denen man im Programmablauf *Objekte* bildet. Objekte führen ein Eigenleben. Sie stellen das *Äquivalent* zu unserem Weltbild dar, welches auf dem Rechner simuliert werden soll.



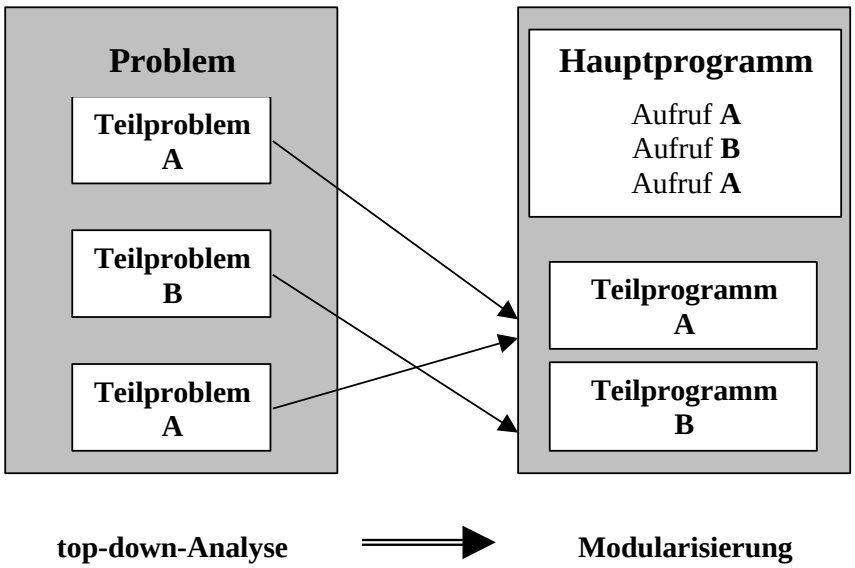
Aufgabe der Programmentwicklung ist **Modularisierung**, d.h. Entwickeln solcher Module, Herausarbeiten ihrer Beziehungen und Analysieren notwendiger Funktionalitäten.

10.1 Entwurfs- und Zerlegungstechniken

10.1.1 Zielgerichteter Entwurf

Der **zielgerichtete Entwurf** dient der *Bewältigung der Komplexität* eines zu lösenden Problems und erfolgt nach dem Prinzip der **schrittweisen Verfeinerung**:
Mittels **top-down-Analyse** wird ein *komplexes Problem* solange in *kleinere überschaubare Teilprobleme* zerlegt. Diese Teilprobleme werden zunächst als *Teilprogramme* realisiert und einzeln ausgetestet, ehe sie zu einem Ganzen zusammenfasst werden, um schließlich das komplexe Problem zu lösen.

Die Gesamtheit der Teilprogramme, zusammen mit einer Vorschrift ihres Zusammenwirkens im Hauptprogramm, ergibt ein Programm zum Problem.



Diese *zielgerichtete Entwurfsstrategie* findet man vor allem bei der *prozeduralen Programmierung* in *imperativen* Sprachen.

10.1.2 Kompositioneller Entwurf

Der **kompositionelle Entwurf** erfolgt nach dem Baukastenprinzip genau umgekehrt. Ausgehend von vorgegebenen *Elementarkonstruktionen* werden durch deren Kombination sukzessiv *komplexere Konstruktionen* gebildet, mit denen man schließlich das Problem lösen kann.

10.1.3 Objektorientierter Entwurf

Bei einem **objektorientierten Entwurf** findet zunächst eine *kompositionelle Entwurfstrategie* Anwendung. Man beginnt mit den zu verarbeitenden Daten und versucht mittels diesen *Objekte* und deren *Klassen* festzulegen, die zur Lösung eines Problems erforderlich sind. Die Klassen werden mit möglichst eng definierten Aufgaben versehen. Dabei können Teilaufgaben auftreten, für die bereits Lösungen bekannt sind. Es gibt hierfür umfangreiche

Klassenbibliotheken und eine Reihe dokumentierter **Entwurfsmuster** (*design patterns*). Ergebnis dieses Schrittes ist eine *Klassenhierarchie*, bestehend aus mehreren Klassen und ihren Beziehungen untereinander.

Bei der Aufbereitung der *Funktionalität* der Methoden als Schnittstelle einer Klasse, setzt die *zielgerichtete Entwurfsstrategie* ein. Durch top-down-Analyse der Aufgabenstellung einer, in der Regel *öffentlichen*, Methode entstehen zusätzlich *nichtöffentliche* Methoden.

10.2 Klassenbeziehungen

10.2.1 UML (Unified Modeling Language)[1]

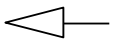
Ein wichtiges Hilfsmittel bei der Modularisierung ist **UML**, eine *Sammlung von Diagrammtypen zur grafischen Darstellung der Beziehungen der Objekte* untereinander. Zu ihnen gehören die schon verwendeten **Klassendiagramme**, in denen die Attribute und Methoden einer Klasse und die Beziehungen mehrerer Klassen untereinander beschrieben werden. UML kann aber noch mehr. Sogenannte **Use-Case-Diagramme** dienen der grafischen Darstellung der *Anwendungsfälle*. **Sequenzdiagramme** stellen *Abläufe* innerhalb der Klassen dar, welche Methode ruft wann welche andere Methode auf. Für **verteilte Systeme** wird in **Verteilungsdiagrammen** dargestellt, welche Komponenten auf welchem *Rechner* beheimatet sind.

Programmiertools

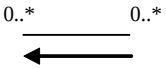
Inzwischen gibt es eine Vielzahl von **Entwicklungsumgebungen**, in denen man Klassen mit *UML* entwirft und anschließend *Java-Code* (oder ein Code einer anderen OOP-Sprache) automatisch generieren kann, auch umgekehrt.

10.2.2 Assoziation, Aggregation und Komposition

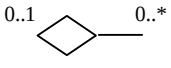
Klassen können in unterschiedlicher Art und Weise in Beziehung stehen.
Neben der **Vererbung** unterscheidet man noch **Assoziation**, **Aggregation** und **Komposition**. Diese werden dann verwendet, wenn sich eine Vererbungsbeziehung nicht eignet. Aggregation und Komposition sind *Spezialfälle* der Assoziation. Komposition ist eine strengere Form der Aggregation. Die Fachbegriffe werden in der Literatur nicht sehr scharf getrennt, es handelt sich bei den Begriffen nicht um dogmatisch festgelegte Terminologien.



Vererbung
Objekte, die durch Spezialisierung oder Generalisierung aus anderen Objekten hervorgehen. Alle Attribute und Methoden werden *vererbt*.
Beispiel: Boot und Paddelboot.



Assoziation
Objekte, die miteinander in Beziehung treten. Auf Grund der Assoziationen können dann Objekte dieser Klassen *miteinander kommunizieren*.
Beispiel: Ein Hund und sein Herrchen.

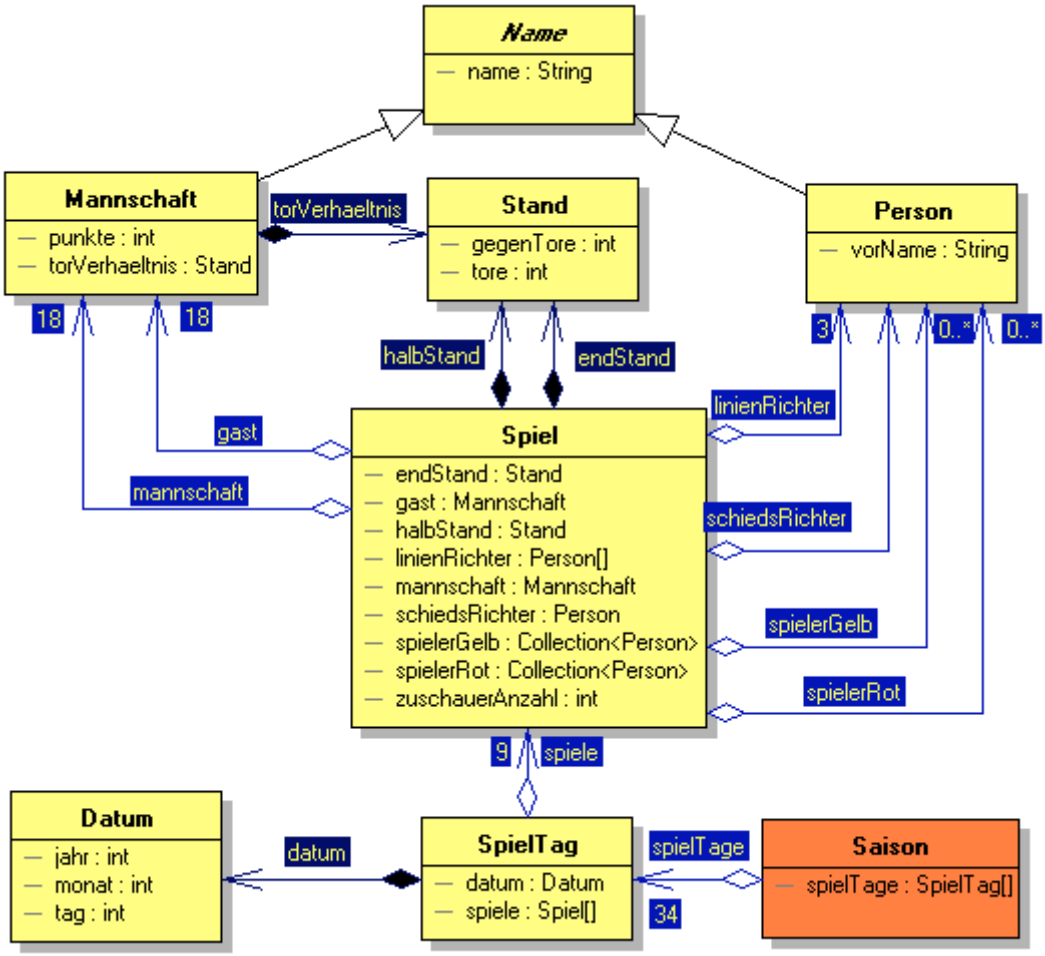


Aggregation
Objekte, die *lose* miteinander in *Verbindung* treten.
Beispiel: In einem Raum befinden sich mehrere Möbelstücke. Diese existieren unabhängig vom Raum. Sie können auch in einem anderen Raum verbracht werden (Umzug).

Komposition
Die Teilobjekte sind *existenzabhängig* vom aggregierten Objekt.
Beispiel: Ein Gebäude besteht aus mehreren Räumen. Ein Raum kann nicht ohne Gebäude existieren.

---

[1] UML-Klassendiagramm: http://timpt.de/topic95.html
Dia - Programm für Diagrammerstellung:
Windows http://dia-installer.de/index.html.de, Linux http://projects.gnome.org/dia/



Multiplizitäten
Die Multiplizität an den Beziehungen zwischen den Objekten gibt an, *wie viele* Objekte des einen Typs mit Objekten des anderen Typs verbunden sein können oder müssen.

Beispiele

1	*genau* 1 Objekt	Die Anfangsklasse einer *Komposition* hat stets diese Multiplizität. Sie kann deshalb weggelassen werden.
18	*genau* 18 Objekte	
0..1	*kein* oder 1 Objekt	Hat die Anfangsklasse einer *Aggregation* diese Multiplizität, so kann sie weggelassen werden.
1..4	1 bis *höchstes* 4 Objekte	
0..*	*beliebig viele* Objekte	

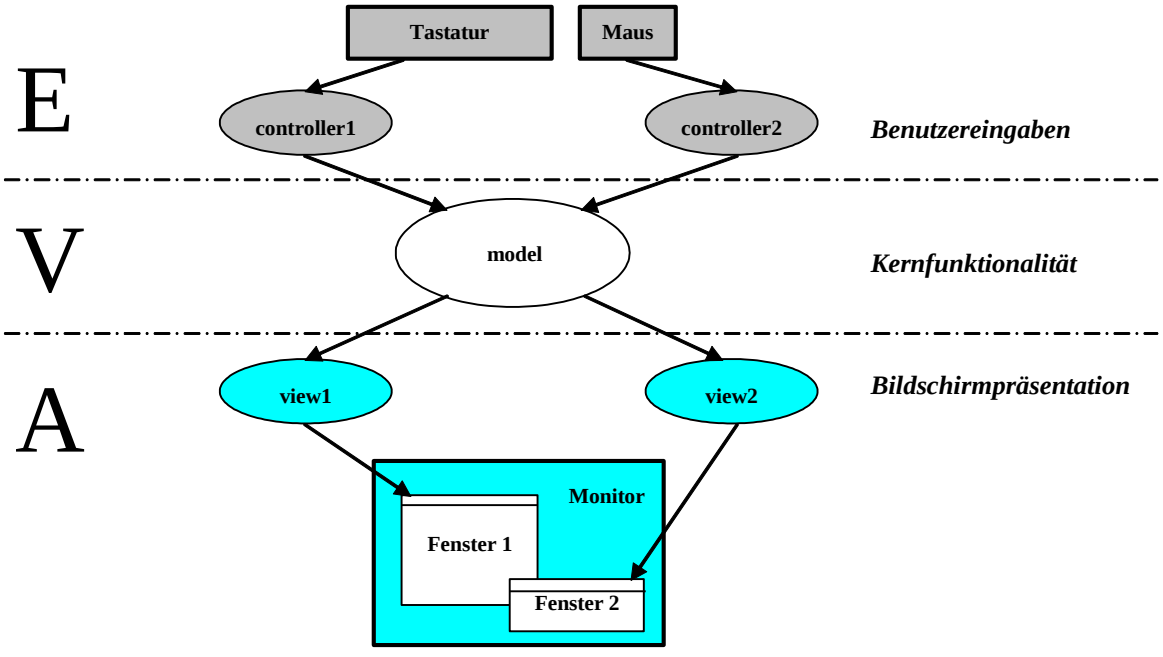
Beziehung	Begründung
Spezialisierung Mannschaft → Name	Jede Mannschaft hat einen Namen und noch weitere Eigenschaften.
Spezialisierung Person → Name	Jede Person hat einen Namen und noch weitere Eigenschaften.
Aggregation Saison → SpielTag	Eine Saison hat mehrere Spieltage (34). Ein Spieltag gehört zur Saison oder nicht.

Aggregation SpielTag → Spiel	An einem Spieltag finden mehrere Spiele statt (9). Ein Spiel findet an einem Spieltag statt oder nicht.
Komposition SpielTag → Datum	Ein Spieltag hat genau ein Datum. Ein Datum existiert nur in Verbindung mit einem Tag
Komposition Spiel → Stand	Ein Spiel hat einen Spielstand. Ein Spielstand existiert nur in Verbindung mit einem Spiel.
Aggregation Spiel → Mannschaft	In jedem Spiel spielen zwei Mannschaften. Es spielen 18 Mannschaften. Eine Mannschaft spielt oder spielt nicht.
Aggregation Spiel → Person	Ein Spiel hat Schiedsrichter (1) und Linienrichter (3). Eine Person ist Schiedsrichter bzw. Linienrichter oder nicht.
Aggregation Spiel → Person	In einem Spiel erhalten ggf. Spieler gelbe bzw. rote Karten (0..*). Eine Person erhält eine gelbe bzw. rote Karte oder nicht.

10.2.3 Entwurfsmuster

Es gibt eine Reihe vorgefertigter **Entwurfsmuster** (*Design Patterns*), die eine Kombination mehrerer Objekte verschiedener Klassen darstellen und ihre Beziehungen untereinander festlegen. Häufig auftretende Programmstrukturen werden in Sammlungen unter einem Namen zusammengefasst und zur Wiederverwendung zur Verfügung gestellt.

MVC-Architektur (M .. Model, V .. View, C .. Controller)



10.3 Modulbildung

Einfache Probleme (kompositioneller Entwurf)
- Unterstreichen der *Substantive* als potentielle Kandidaten für Klassen:

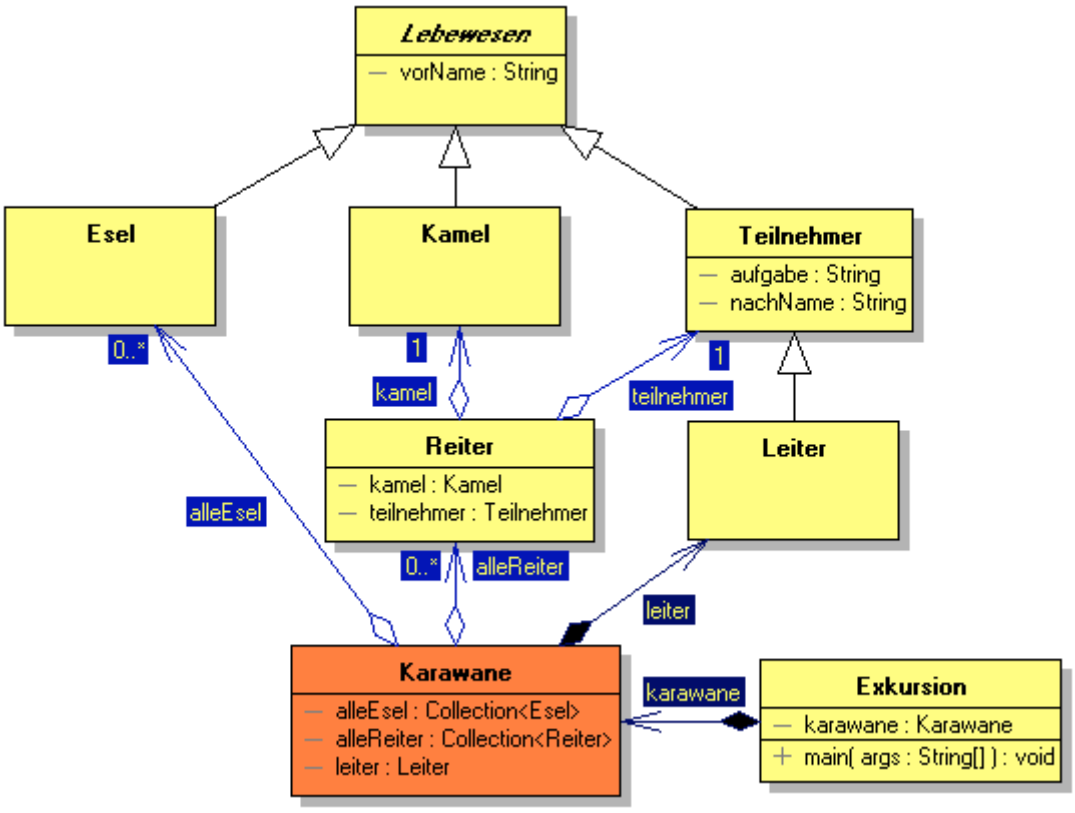
Ein sehr **einfaches Verfahren** besteht darin, aus einer verbalen Aufgabenstellung die **Substantive** herauszusuchen. Diese sind dann die potentiellen Klassenkandidaten. Für die hier betrachteten Beispiele reicht meist dieses Verfahren aus.

Eine *Wüstenexkursion* soll ausgerichtet werden. Dazu wird eine *Karawane* zusammengestellt. Der *Leiter* der Exkursion sucht sich 5 weitere *Teilnehmer*, für jeden Teilnehmer ein *Kamel* und zusätzlich 3 *Lastesel*. Von jedem Teilnehmer werden *Vorname*, *Nachname* und *Arbeitsgebiet* festgehalten. Die Tiere besitzen Vornamen.

Modellieren Sie die Karawane als UML – Klassendiagramm (ohne Funktionalität), beachten Sie Hierarchien zwischen den beteiligten Klassen.

Substantive	**Klasse**
Wüstenexkursion	Exkursion
Karawane	Karawane
Leiter	Leiter
Teilnehmer	Teilnehmer
Kamel	Kamel
Lastesel	Esel
Vorname, Nachname, Arbeitsgebiet	String

Klassenhierarchie:



Komplexe Probleme (objektorientierter Entwurf)
- Schrittweises Vorgehen in sogenannten *Modellierungsphasen*:

Je größer ein objektorientiertes System wird, desto leichter verliert ein Entwickler den Überblick. Insbesondere für den Einsatz objektorientierter Softwaretechnologien sind verschiedene **Software-Engineering-Methoden** entstanden. Allen gemeinsam ist das schrittweise Vorgehen in sogenannten **Modellierungsphasen**:

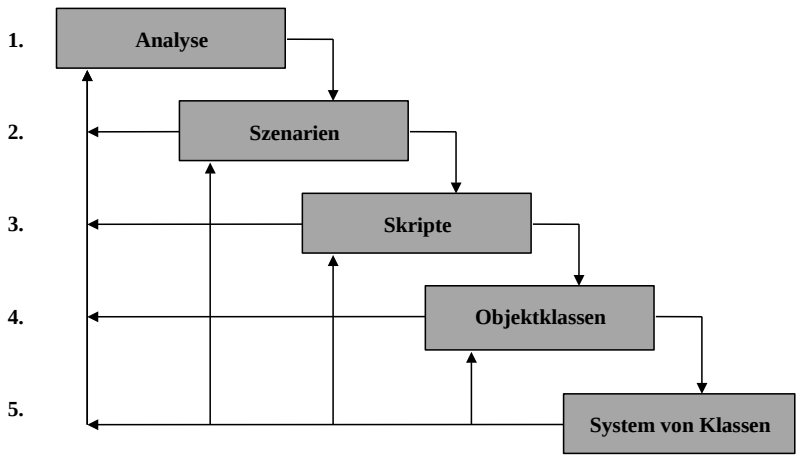
1. ***Analyse***
   Analyse der Situation zur Klarheit über *Aufgabe* und *Ziel* der Projektes, *detaillierte Formulierung der Problemstellung*, *Projektplanung*.

2. ***Szenarien***
   Beschreibung der Anforderungen (*Anwendungsfälle*) des Programms mit Hilfe von **Szenarien**, d.h. Formulieren der *Funktionalität* des Programms.

3. ***Skripte***
   Herausarbeiten der miteinander *kommunizierenden Einheiten*, den potentiellen Programmobjekten, mit Hilfe *tabellarischer* **Skripte** zu den einzelnen Szenarien (Wer kommuniziert mit wem? Wer fordert von wem welche Dienstleistung?).

4. ***Objektklassen***
   Bilden von **Objektklassen** einschließlich ihrer *Dienstleistungen*, die Objekte anderen Objekten zur Verfügung zu stellen haben:
   Alle **Anbieter** von Dienstleistungen sind potentielle Objektklassen eines Problems. **Nachfrager**, *die keine Anbieter sind*, werden nicht als Objektklassen aufgenommen. **Attribute** sind alle Eigenschaften, die ein Anbieter haben muss, um eine Dienstleistung eines Nachfragers erbringen zu können. Die angebotenen Dienstleistungen sind **Instanzmethoden** der Klassen. Hinzu kommen noch **Zugriffsmethoden** auf Attribute der Objekte.

5. ***System von Klassen***
   Erstellen eines **Systems von Klassen**, einschließlich ihrer *Klassenhierarchie*, welches den Anforderungen der Aufgabenstellung gerecht wird, unter Verwendung neuer und bereits vorhandener Klassen aus *Bibliotheken* und *Entwurfsmustern*.



Wasserfallmodell von Adele Goldberg

Die Ergebnisse werden in sogenannte **CRC-Karten** (Class Responsibility Card) dokumentiert. Für jede Klasse wird eine solche Karteikarte angelegt. Auf ihr werden Attribute und Methoden festgeschrieben, aber auch eine detaillierte Beschreibung der Aufgaben dieser Klasse, alle anderen Klassen, mit der diese interagiert u.v.m. Diese Karten sind Entwicklungsgrundlage für Programmierer, die in Gruppen an großen Projekten arbeiten.

10.4 Beispiel „Milchladen“

Schritt 1. Analyse
Ein Kunde möchte in einem Milchladen eine bestimmte Anzahl (*Soll*) von Litern Milch kaufen und bringt dazu eine Milchkanne (*K*) ohne Maßeinteilung mit, welche ein Fassungsvermögen von K_{max} Litern hat. Der Verkäufer besitzt seinerseits einen Eimer (*E*) mit einer Eichmarke bei E_{max} Litern.

Wie kann der Verkäufer den Wunsch des Kunden erfüllen?

Betrachte *Soll* = 6, E_{max} = 10, K_{max} = 7:

Handlungsfolge	**Liter in *K***	**Liter in *E***
Anfangszustand	0	0
fülle *E*,	0	10
schütte *E* nach *K*,	7	3
leere *K*,	0	3
schütte *E* nach *K*,	3	0
fülle *E*,	3	10
schütte *E* nach *K*,	7	6
leere *K*,	0	6
schütte *E* nach *K*,	6	0
fertig.		

Schritt 2. Szenarien
Szenario 1: Ist Problem lösbar?
Ein Kunde kommt in den Laden, hat eine Kanne mit einem festgelegten Fassungsvermögen und übergibt diese mit einer gewünschten Menge von Milch dem Verkäufer. Der Verkäufer hat einen Eimer mit einem festgelegten Fassungsvermögen im Laden und überprüft, ob er das Problem lösen kann.

- Algorithmus ***kannenProblem***:

```
'leere E,'; 'leere K,';
if( soll == 0) { 'fertig.';}
if( soll == Kmax ) { 'fülle K,'; 'fertig.';}
if( soll > Kmax ) {'K zu klein,'; 'fertig.';}
if( soll % ggT( Kmax, Emax) != 0)
{ 'nicht lösbar,'; 'fertig.';}
abmessen();
```

Szenario 2: Problem lösen
Ist das Problem zu lösen, so misst der Verkäufer durch geeignetes Umschütten, füllen und leeren von Kanne und Eimer die gewünschte Menge Milch ab.

- Algorithmus ***abmessen***:

```
while( K != soll)
{
  'fülle E,';
```

```
    while( E != 0)
    {
      'schütte E nach K,';
      if( K == Kmax) {'leere K,';}
    }
  }
  'fertig.';
```

Schritt 3. Skripte

Nachfrager	Aktion	Anbieter	Dienstleistung
Szenario 1			
Kunde	Wie kann man das Problem lösen?	Verkäufer	kannenProblem
Verkäufer	*Soll == 0*, Kanne leeren.	Kanne	leeren
Verkäufer	*Soll == K_{max}* , Kanne füllen.	Kanne	fuellen
Verkäufer	*Soll > K_{max}* , Problem nicht lösbar.	Verkäufer	nachricht
Verkäufer	*Soll % ggT(K_{max}, E_{max}) != 0*, nicht lösbar.	Verkäufer	nachricht
Verkäufer	Problem lösen.	Verkäufer	abmessen
Szenario 2			
Verkäufer	stellt fest, ob *Soll* in Kanne ist	Kanne	istInhSoll
Verkäufer	Eimer füllen	Eimer	fuellen
Verkäufer	stellt fest, ob Eimer leer ist	Eimer	istLeer
Verkäufer	Kanne nimm Milch aus dem Eimer auf	**Kanne**	**eingießenAus**
Kanne	Eimer schütte Milch in Kanne	**Eimer**	**ausgießenIn**
Kanne	prüft nach, wie viel Milch in ihr Platz hat	**Kanne**	**leerPlatz**
Verkäufer	stellt fest, ob Kanne voll ist	Kanne	istVoll
Verkäufer	Kanne leeren	Kanne	leeren

Wegen des *Grundprinzips* des objektorientierten Paradigmas **„Ein Objekt verwaltet sich selbst“** verlangt das Umschütten vom Eimer in die Kanne genau genommen nach zwei Methoden:
Der Eimer gießt Milch aus (**ausgießenIn**), die Kanne nimmt Milch auf (**eingießenAus**).

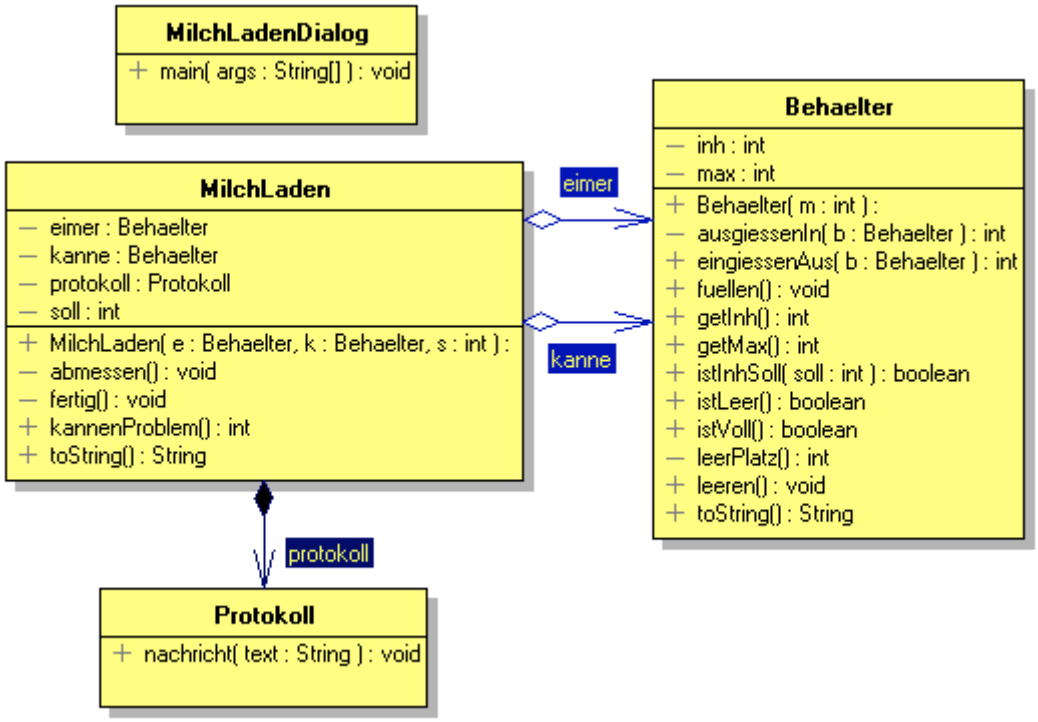
Schritt 4. Objektklassen
Objektklassen: Verkäufer, Eimer, Kanne.

Objekt	Attribute	Dienstleistungen	Zugriffsmethoden
MilchLaden (Verkäufer)	Eimer Kanne Soll	kannenProblem abmessen nachricht	-
Eimer	Inhalt Maximum	fuellen ausgiessenIn istLeer	getInh getMax
Kanne	Inhalt Maximum	fuellen leeren eingiessenAus leerPlatz istVoll istInhSoll	getInh getMax

Schritt 5: System von Klassen, Klassenhierarchie

- Eimer und Kannen sind Behälter, die sich ähnlich verhalten und austauschbar sind. Deshalb bietet sich an, diese Objekte in einer Klasse `Behaelter` zusammenzufassen.
- Für die Berechnung des `ggT` benötigen wir keine neue Methode. Wir verwenden die bereits in der Klasse `Euklid` implementierte.
- Eine Dienstleistung `nachricht` wird durch eine Klasse `Protokoll` für eine Konsolenausgabe als Bildschirmrepräsentation realisiert.

Damit ergibt sich folgendes System von Klassen für das Milchladenproblem:



Implementierung

Protokoll.java

```
// Protokoll.java                                        MM 2014
/**
 * Protokolliert Vorgänge im Milchladen.
 */
public class Protokoll
{
/* ------------------------------------------------------- */
// Service-Methode
/**
 * Gibt Nachricht auf der Konsole aus.
 * @param text Nachricht
 */
  public void nachricht( String text)
```

```
    {
      System.out.print( text);
    }
  }
```

Behaelter.java

```
  //Behaelter.java                                        MM 2014

  /**
   * Behaelter (Kanne oder Eimer).
   */
  public class Behaelter
  {
  /* ------------------------------------------------------ */
  // Attribute
  /**
   * Inhalt in Liter.
   */
    private int inh;

  /**
   * Fassungsvermoegen in Liter.
   */
    private int max;

  /* ------------------------------------------------------ */
  // Konstruktor
  /**
   * Konstruktor, erzeugt Behaelter.
   * @param m maximales Fassungsvermoegen in Liter
   */
    public Behaelter( int m)
    {
      max = m; inh = 0;
    }

  /* ------------------------------------------------------ */
  // Service-Methoden
  /**
   * Vergleicht Inhalt mit Soll.
   * @param soll gewuenschte Fuellmenge
   * @return true, falls Soll erreicht
   */
    public boolean istInhSoll( int soll)
    {
      return inh == soll;
    }

  /**
   * Behaelter ist voll.
   * @return true, falls voll
```

```
   */
    public boolean istVoll()
    {
      return inh == max;
    }

  /**
   * Behaelter ist leer.
   * @return true, falls leer
   */
    public boolean istLeer()
    {
      return inh == 0;
    }

  /**
   * Behaelter wird geleert.
   */
    public void leeren()
    {
      inh = 0;
    }

  /**
   * Behaelter wird gefuellt, bis er voll ist.
   */
    public void fuellen()
    {
      inh = max;
    }

  /**
   * Behaelter nimmt Inhalt eines anderen Behaelters auf.
   * @param b anderer Behaelter
   * @return eingegossene Liter
   */
    public int eingiessenAus( Behaelter b)
    {
      int liter = b.ausgiessenIn( this);
      inh += liter;
      return liter;
    }

  /**
   * Inhalt des Behaelters wird
   * in einen anderen Behaelter gegossen.
   * @param b anderer Behaelter
   * @return ausgegossene Liter
   */
    private int ausgiessenIn( Behaelter b)
    {
      int liter = b.leerPlatz();
```

```
      if (liter > inh) liter = inh;
      inh -= liter;
      return liter;
    }

  /**
   * Restlicher Platz im Behaelter.
   * @return Platz in Liter
   */
    private int leerPlatz()
    {
      return max - inh;
    }

  /* ------------------------------------------------- */
  // Zugriffsmethoden
  /**
   * Liest aktuelle Fuellmenge.
   * @return Inhalt in Liter
   */
    public int getInh()
    {
      return inh;
    }

  /**
   * Liest maximale Fuellmenge.
   * @return Maximum in Liter
   */
    public int getMax()
    {
      return max;
    }

  /**
   * Beschreibt Zustand des Behaelters.
   * @return Inhalt und Fassungsvermoegen in Liter
   */
    public String toString()
    {
      return inh + " von "+ max;
    }
  }
```

MilchLaden.java

```
  // MilchLaden.java                                    MM 2014

  import Tools.Euklid.*;                                  // ggT
  import Tools.IO.*;                                   // Eingaben

  /**
```

```
 * Bestimmen der Handlungsschritte beim Abfuellen
 * von 'Soll' Liter mittels einem leeren Eimer 'E'
 * und einer leeren Kanne 'K' mit festgelegtem
 * Fassungsvermoegen.
 */
public class MilchLaden
{
/* ------------------------------------------------- */
// Attribute
/**
 * Behaelter des Kunden.
 */
  private Behaelter kanne;

/**
 * Behaelter des Verkaeufers.
 */
  private Behaelter eimer;

/**
 * Abfuellmenge.
 */
  private int soll;

/**
 * Protokolliert die Aktionen im Milchladen.
 */
  private Protokoll protokoll;

/* ------------------------------------------------- */
// Konstruktor
/**
 * Konstruktor, erzeugt Milchladen mit Protokoll.
 * @param e Behaelter des Verkaeufers
 * @param k Behaelter des Kunden
 * @param s Abfuellmenge
 */
  public MilchLaden( Behaelter e, Behaelter k, int s)
  {
    eimer = e; kanne = k; soll = s;
    protokoll = new Protokoll();
  }

/* ------------------------------------------------- */
// Service-Methoden
/**
 * Behandelt Problem mittels Fallunterscheidung.
 * @return 0 Problem loesbar, 1 sonst.
 */
  public int kannenProblem()
    throws NPlusException
  {
```

```
      if( soll == 0)                          // Sonderfaelle
      {
        fertig(); return 0;
      }

      if( soll == kanne.getMax())
      {
        protokoll.nachricht( this.toString());
        kanne.fuellen();
        protokoll.nachricht("\tfuelle K,\n");
        fertig(); return 0;
      }
                                     // Nicht loesbare Faelle
      if( soll > kanne.getMax())
      {
        protokoll.nachricht
        ( this.toString() + "\tMilchkanne zu klein!");
        return 1;
      }

      if( soll %
        Euklid.ggT( eimer.getMax(), kanne.getMax()) != 0)
      {
        protokoll.nachricht
        ( this.toString() + "\tKeine Loesung moeglich!");
        return 1;
      }

      abmessen(); fertig();                    // Normalfall
      return 0;
    }

  /**
   * Loest Problem durch Umschuetten.
   */
    private void abmessen()
    {
      while( !kanne.istInhSoll( soll))            // fertig?
      {
        protokoll.nachricht( this.toString());
        eimer.fuellen();
        protokoll.nachricht( "\tfuelle E,\n");

        while( !eimer.istLeer())              // Eimer leer?
        {
          protokoll.nachricht( this.toString());
          kanne.eingiessenAus(eimer);
          protokoll.nachricht( "\tschuette von E nach K,\n");

          if( kanne.istVoll())                // Kanne voll?
          {
            protokoll.nachricht( this.toString());
```

```
            kanne.leeren();
            protokoll.nachricht( "\tleere K,\n");
          }
        }
      }
    }

  /**
   * Nachricht, dass Problem geloest wurde.
   */
    private void fertig()
    {
      protokoll.nachricht( this.toString() + "\t"
        + kanne.getInh() + " Liter abgefuellt, fertig.");
    }

  /* ------------------------------------------------------ */
  // Zugriffsmethode
  /**
   * Stellt Stand in Eimer und Kanne dar.
   * @return Inhalt und Fassungsvermoegen in Liter
   */
    public String toString()
    {
      return "E: " + eimer.toString()
         + "\tK: " + kanne.toString();
    }
  }
```

MilchLadenDialog.java

```
  // MilchLadenDialog.java                                MM 2014

  import Tools.IO.*;                                  // Eingaben

  /**
   * Loesen des Milchladenproblem.
   */
  public class MilchLadenDialog
  {
  /**
   * Hauptprogramm.
   */
    public static void main( String[] args)
    {
      int soll = IOTools.readInteger( "Soll = ");
      int eMax = IOTools.readInteger( "Emax = ");
      int kMax = IOTools.readInteger( "Kmax = ");

      System.out.println( "\nSoll = " + soll + " Emax = "
        + eMax + " Kmax = " + kMax + "\tMilchladen:");
```

```
      Behaelter e = new Behaelter( eMax);
      Behaelter k = new Behaelter( kMax);

      MilchLaden laden = new MilchLaden( e, k, soll);
      try
      {
        laden.kannenProblem();
      }
      catch( Exception ex){}
    }
  }
```

MilchLaden.out

```
Soll = 5, Emax = 10, Kmax = 7   Milchladen:
E: 0 von 10   K: 0 von 7   fuelle E,
E: 10 von 10  K: 0 von 7   schuette von E nach K,
E: 3 von 10   K: 7 von 7   leere K,
E: 3 von 10   K: 0 von 7   schuette von E nach K,
E: 0 von 10   K: 3 von 7   fuelle E,
E: 10 von 10  K: 3 von 7   schuette von E nach K,
E: 6 von 10   K: 7 von 7   leere K,
E: 6 von 10   K: 0 von 7   schuette von E nach K,
E: 0 von 10   K: 6 von 7   fuelle E,
E: 10 von 10  K: 6 von 7   schuette von E nach K,
E: 9 von 10   K: 7 von 7   leere K,
E: 9 von 10   K: 0 von 7   schuette von E nach K,
E: 2 von 10   K: 7 von 7   leere K,
E: 2 von 10   K: 0 von 7   schuette von E nach K,
E: 0 von 10   K: 2 von 7   fuelle E,
E: 10 von 10  K: 2 von 7   schuette von E nach K,
E: 5 von 10   K: 7 von 7   leere K,
E: 5 von 10   K: 0 von 7   schuette von E nach K,
E: 0 von 10   K: 5 von 7   5 Liter abgefuellt, fertig.
```

10.5 Vor- und Nachteile objektorientierter Programmierung

Die vier Grundpfeiler der objektorientierten Programmierung:

1. **Klassenbildung zur *Kapselung* der zusammengehörigen *Attribute* und ihrer *Methoden*,**
2. ***Vererbung* zur Entwicklung und Nutzung von vorgefertigten Klassen, innerhalb von Bibliotheken oder nutzerspezifisch, nach dem Bausteinprinzip,**
3. ***Generalisierung* bzw. *Spezialisierung* zur Verallgemeinerung bzw. Erweiterung bereits vorhandener Klassen und dem Aufbau einer Klassenhierarchie und**
4. ***Polymorphismus* zum Erhöhen des Abstraktionsniveaus bei der Programmierung.**

☺ **Vererbung** und **Klassenbibliotheken** fördern die Wiederverwendbarkeit.

⇨ Reduzierung des Entwicklungsaufwands und Verkürzung der Entwicklungszeit.

☺ **Wiederverwendung** vermeidet die Entwicklung von Duplikaten.

⇨ Beschränkung der Wartungsarbeiten auf das Original.

☺ **Polymorphie** und **dynamisches Binden** vermeiden zusätzliche Fallunterscheidungen.

⇨ Programmstrukturen sind einfach und leicht erweiterbar.

☺ Der objektorientierte Entwurf findet auf einer **hohen Abstraktionsebene** statt.

⇨ Entwurfsfehler können vermieden werden.

☺ Programmiertools ermöglichen **automatische Codegenerierung**.

⇨ Beschleunigung der Programmierungstätigkeit.

☹ Objektorientierte Programmierung erfordert „**objektorientiertes Denken**".

⇨ Der Umstieg fällt erfahrenen Entwicklern aus der konventionellen imperativen Programmierung besonders schwer.

☹ Die Beherrschung einer objektorientierten Sprache und Wissen über die Formulierung von Algorithmen reichen nicht aus. Die Benutzung von **Klassenbibliotheken** und **Design Patterns** setzt die Kenntnis über ihren Aufbau voraus.

⇨ Umfangreiche Einarbeitungszeit ist erforderlich.

☹ Objekte haben wegen ihrer dynamischen Speicherverwaltung einen höheren **Speicherbedarf** als konventionelle Daten und die dynamische Bindung kann die **Programmlaufzeit** verlängern.

⇨ Erhöhung der Rechenzeit und des Speicherbedarfs gegenüber herkömmlichen Sprachen.

☹ Die **Oberflächen** der Programmiertools sind noch unausgereift und der damit generierte Code ist teilweise sehr unübersichtlich.

⇨ Umfangreiche Einarbeitungszeit ist erforderlich.

Die Entscheidung für oder gegen ein Programmierparadigma und damit für oder gegen eine Programmiersprache ist immer von objektiven Kriterien abhängig, d.h. vom konkreten Anwendungsfall.