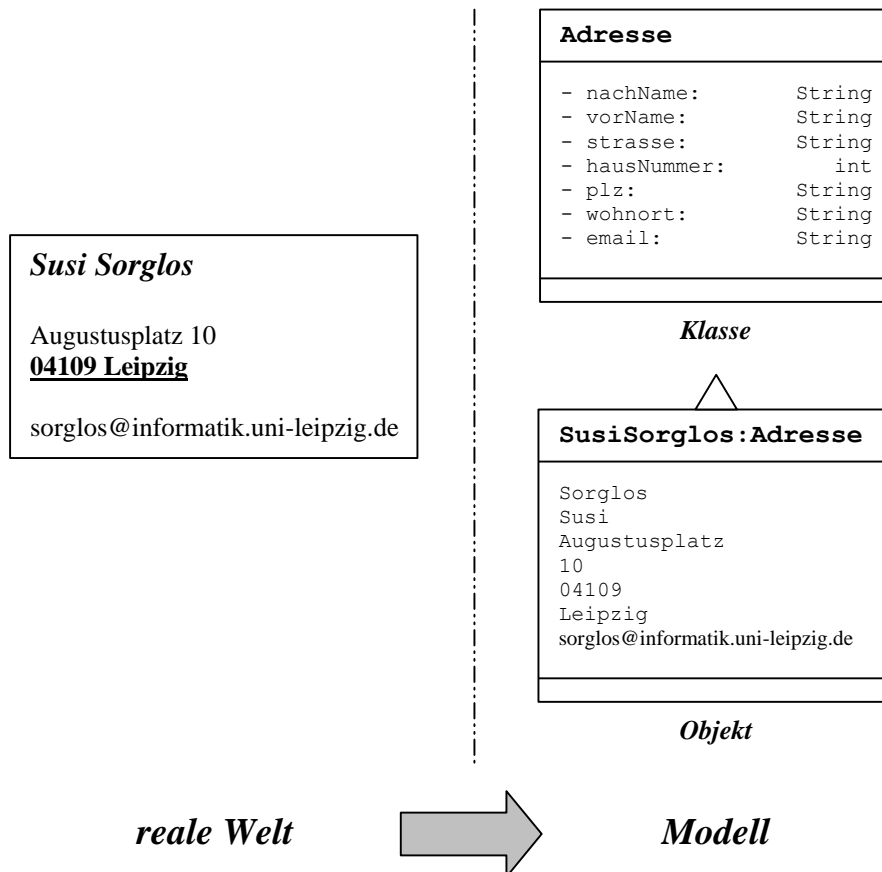


## Inhalt

10	Modellierung .....	10-2
10.1	Entwurfs- und Zerlegungstechniken .....	10-3
10.1.1	Zielgerichteter Entwurf .....	10-3
10.1.2	Kompositioneller Entwurf .....	10-3
10.1.3	Objektorientierter Entwurf .....	10-3
10.2	Klassenbeziehungen.....	10-5
10.2.1	UML (Unified Modeling Language).....	10-5
10.2.2	Assoziation, Aggregation und Komposition .....	10-5
10.2.3	Entwurfsmuster .....	10-6
10.3	Modulbildung.....	10-7
10.4	Beispiel „Milchladen“ .....	10-9
10.5	Vor- und Nachteile objektorientierter Programmierung.....	10-18

## 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 damit die **Modularisierung**, d.h. Entwickeln solcher Module, Herausarbeiten ihrer Beziehungen und Analysieren notwendiger Funktionalitäten.

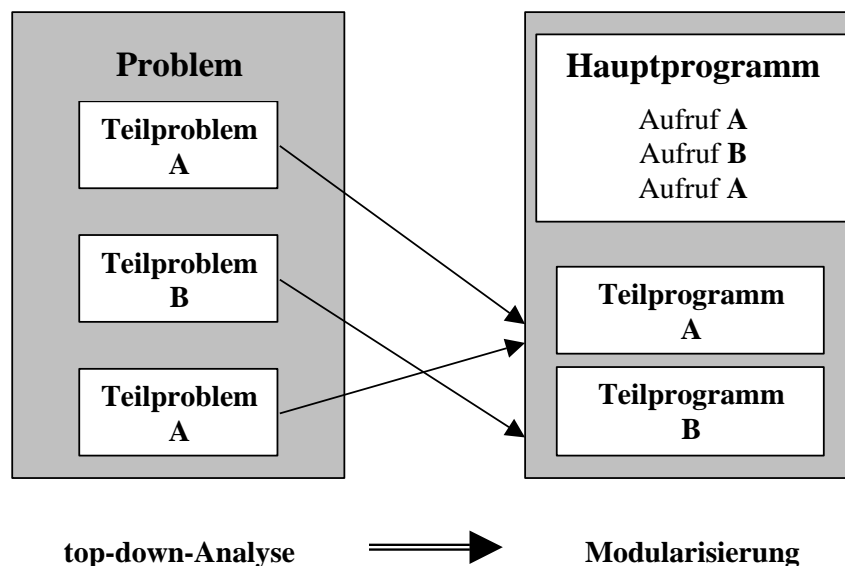
## 10.1 Entwurfs- und Zerlegungstechniken

### 10.1.1 Zielgerichteter Entwurf

Der **zielgerichtete Entwurf** dient der *Bewältigung der Komplexität* des zu lösenden Problems. Die Analyse erfolgt durch die **Zerlegung** des Problems in seine Teilprobleme nach dem Prinzip der **schrittweisen Verfeinerung**:

Mittels **top-down-Analyse** zerlegt man ein *komplexes Problem* solange in *kleinere Teilprobleme* bis diese überschaubar werden. Diese Teilprobleme kann man zunächst als *Teilprogramme* realisieren und auch einzeln testen, ehe man sie zu einem Ganzen zusammenfasst, um schließlich das komplexe Problem als Ganzes 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 *imperativen Programmierung*.

### 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 des 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 ihre Beziehungen untereinander.

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

## 10.2 Klassenbeziehungen

### 10.2.1 UML (Unified Modeling Language)<sup>1</sup>

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 mehrere 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

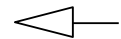
Inzwischen gibt es eine Vielzahl von **Entwicklungsumgebungen**, in denen man Klassen mit *UML* entwirft und anschließend *Java-Code* (oder ein Code einer anderen OOP-Sprache) automatisch generieren kann. *Umgekehrt* gibt es Tools, welche aus dem bereits vorhandenen Code entsprechende UML-Diagramme erzeugen.

### 10.2.2 Assoziation, Aggregation und Komposition

Klassen können in unterschiedlicher Art und Weise in Beziehung stehen.

Neben der **Vererbung** unterscheidet man noch die **Assoziation**, die **Aggregation** und die **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.

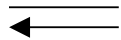
#### Vererbung:



Objekte, die durch Spezialisierung oder Generalisierung aus anderen Objekten hervorgehen. Alle Attribute und Methoden werden *vererbt*.

Beispiel: Boot und Paddelboot.

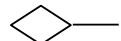
#### Assoziation:



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: Ein Schwarm von Vögeln, auf einen Vogel mehr oder weniger kommt es bzgl. des Ganzen nicht so drauf an.

#### Komposition:

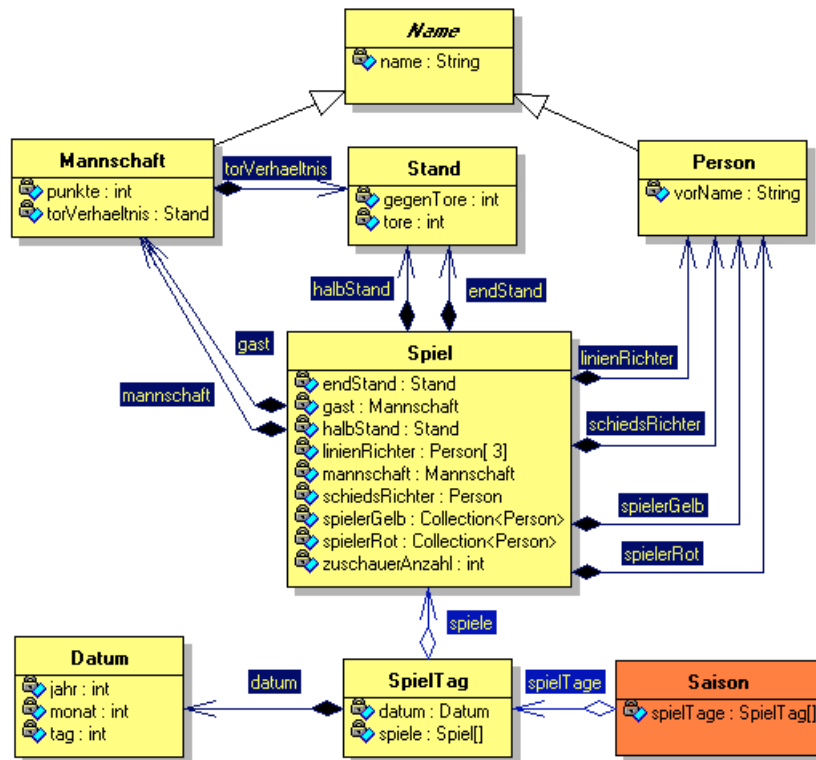


Objekt, welches aus verschiedenen Teilobjekten *zusammengesetzt* wird. Das Objekt ist von den einzelnen Teilobjekten *existenzabhängig*.

<sup>1</sup> Dia: Programm für Diagrammerstellung

Windows <http://dia-installer.de/index.html.de>, Linux <http://projects.gnome.org/dia/>

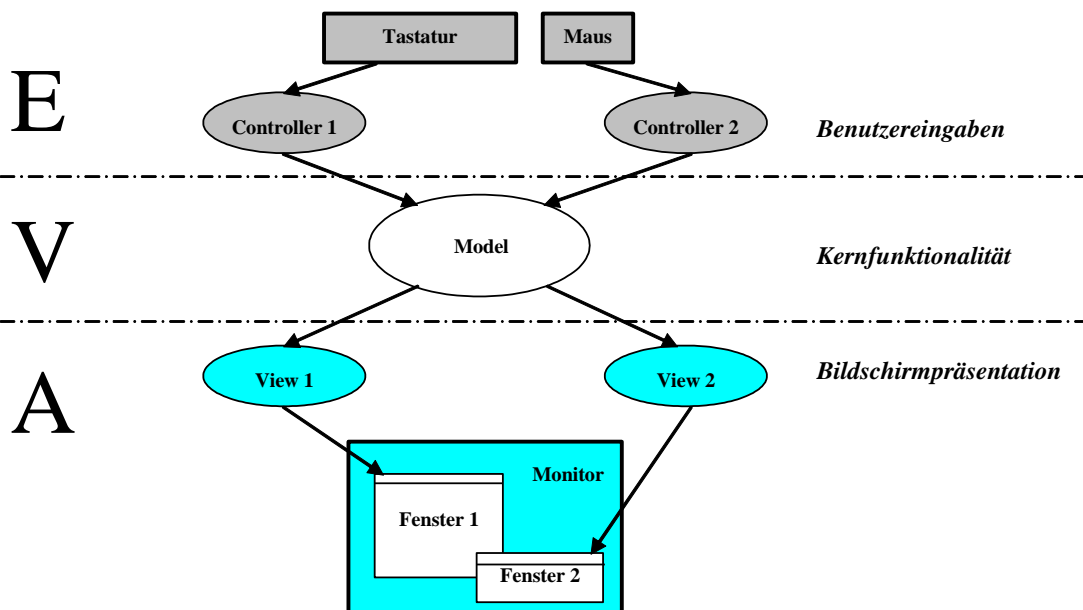
Beispiel: Ein Auto ist aus Motor, 4 Räder, Vergaser, ... zusammengesetzt. Jedes dieser einzelnen Teile ist für die Funktionsfähigkeit des Autos notwendig.



### 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.

Ein Beispiel für eine häufig vorkommende Kombination von Objekten ist die bereits mit Smalltalk eingeführte **MVC-Architektur** (*M* .. *Model*, *V* .. *View*, *C* .. *Controller*). Ein Objekt **Model** realisiert die *Kernfunktionalität* und ist das *Anwendungsobjekt*, ein Objekt **View** ist die Schnittstelle für die *Bildschirmrepräsentation* und ein Objekt **Controller** für die *Benutzereingaben*. Durch diese Architektur ist es möglich, zu einem Model *mehrere* Viewer und *mehrere* Controller zur Verfügung zu stellen.



### 10.3 Modulbildung

#### Einfache Probleme (kompositioneller Entwurf)

##### - Unterstreichen der *Substantive* als potentielle Kandidaten für Klassen:

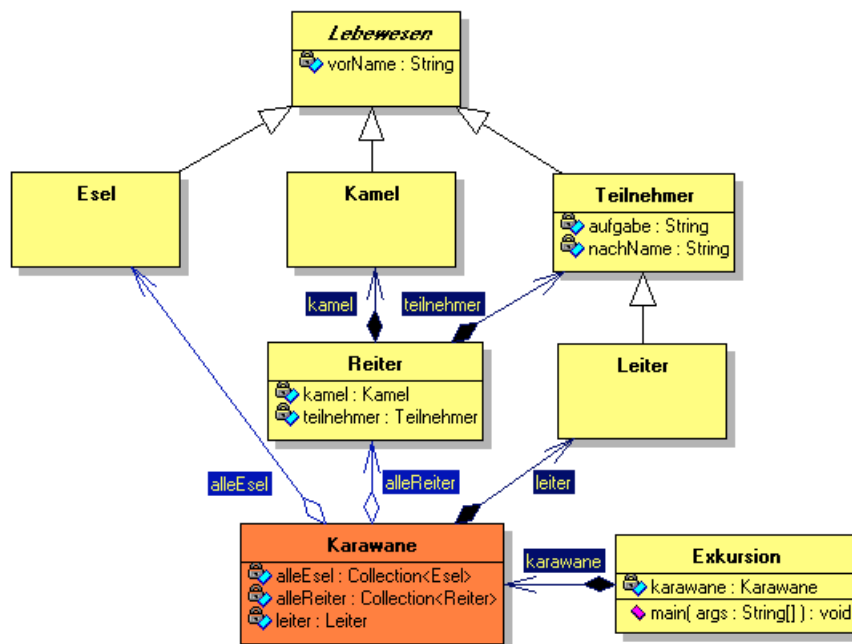
Ein sehr **einfaches Verfahren** besteht darin, aus einer verbalen Aufgabenstellung die **Substantive** herauszusuchen. Diese sind dann die potentiellen Klassenkandidaten. Für die hier betrachteten Beispiele reicht meist dieses Verfahren aus.

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 der Vor-, Nachname und sein Arbeitsgebiet festgehalten. Die Tiere haben Vornamen.

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

Vorname, Nachname, Arbeitsgebiet:

Klasse String



#### 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, mittels *tabellarischen Skripten* zu den einzelnen Szenarien (Wer kommuniziert mit wem? Wer fordert von wem welche Dienstleistung?).

4. **Objektklassen**

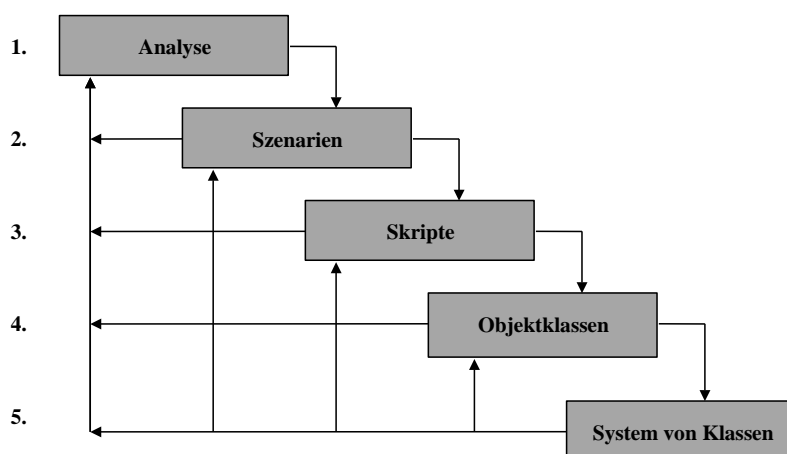
Bilden von **Objektklassen** einschließlich ihrer *Dienstleistungen*, die Objekte einer Klassen anderen Objekten zur Verfügung zu stellen haben:

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*.

Die Ergebnisse werden in sogenannte **CRC-Karten** (Class Responsibility Card) abgelegt. 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.



Wasserfallmodell von Adele Goldberg

## 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  $Soll = 6$ ,  $E_{max} = 10$ ,  $K_{max} = 7$ :

Lösung:

Handlungsfolge	Liter in <i>K</i>	Liter in <i>E</i>
Anfangszustand	0	0
fülle <i>E</i> ;	0	10
schütte <i>E</i> nach <i>K</i> ;	7	3
leere <i>K</i> ;	0	3
schütte <i>E</i> nach <i>K</i> ;	3	0
fülle <i>E</i> ;	3	10
schütte <i>E</i> nach <i>K</i> ;	7	6
leere <i>K</i> ;	0	6
schütte <i>E</i> nach <i>K</i> ;	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;',
wenn Soll == 0 dann 'fertig.',
wenn Soll == Kmax dann 'fülle K;', 'fertig.',
wenn Soll > Kmax dann 'K zu klein;', 'fertig.',
wenn Soll mod ggT(Kmax, Emax) != 0 dann '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*:

```
berechne solange K != Soll
'fülle E;',
```

*berechne solange  $E \neq 0$*   
*'schütte  $E$  nach  $K$ ;',*  
*wenn  $K == K_{max}$  dann 'leere  $K$ ;',*  
*'fertig.'*

### Schritt 3. Skripte

Nachfrager	Aktion	Anbieter	Dienstleistung
<b>Szenario 1</b>			
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	$Soll \bmod ggT(K_{max}, E_{max}) \neq 0$ , nicht lösbar.	Verkäufer	nachricht
Verkäufer	Problem lösen.	Verkäufer	abmessen
<b>Szenario 2</b>			
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	<b>Kanne</b>	<b>eingießenAus</b>
Kanne	Eimer schütte Milch in Kanne	<b>Eimer</b>	<b>ausgießenIn</b>
Kanne	prüft nach, wie viel Milch in ihr Platz hat	<b>Kanne</b>	<b>leerPlatz</b>
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

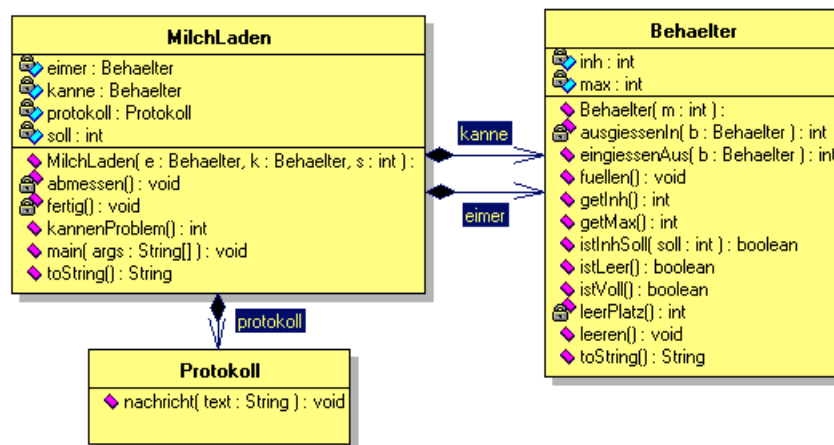
Objektklassen: Verkäufer, Eimer, Kanne.

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
/**
 * 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);
    }
}

```

MM 2007

#### Behaelter.java

```

//Behaelter.java
/**
 * Behaelter (Kanne oder Eimer).
 */

```

MM 2007

```
public class Behaelter
{
/* ----- */
// Attribute
/**
 * Inhalt.
 */
    private int inh;

/**
 * Volumen.
 */
    private int max;

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

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

/**
 * Behaelter ist voll.
 */
    public boolean istVoll()
    {
        return inh == max;
    }

/**
 * Behaelter ist 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
 */
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
 */
private int ausgiessenIn( Behaelter b)
{
    int liter = b.leerPlatz();
    if (liter > inh) liter = inh;
    inh -= liter;
    return liter;
}

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

/* ----- */
// Zugriffsmethoden
/**
 * Liest aktuelle Fuellmenge.
```

```

*/
public int getInh()
{
    return inh;
}

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

/**
 * Beschreibt Zustand des Behaelters.
 */
public String toString()
{
    return inh + " von " + max;
}
}

```

**MilchLaden.java**

```

// MilchLaden.java                                MM 2007
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
    {
        // Sonderfaelle
        if( soll == 0)
        {
            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;
    }

// Normalfall

    abmessen(); fertig();
    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 abgefüllt, fertig.");
}

/* ----- */
// Zugriffsmethode
/**
 * Stellt Stand in Eimer und Kanne dar.
 */
public String toString()
{

```

```

        return "E: " + eimer.toString()
            + "\tK: " + kanne.toString();
    }

    /* ----- */
    // Startmethode
    /**
     * Startet Problem.
     */
    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 abgefüllt, 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.
- ☺ Programmierertools 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 Pattern** 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 herkömmlicher Sprachen.
- ☹ Die **Oberflächen** der Programmierertools 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.**