© Hans-Georg Beckmann 2004

UML

1. UML im Unterricht 1
2. Klassendiagramme 1
2.1 Klassendiagramme und Vererbung 4
2.2 Klassendiagramme, abstrakte Klassen 5
3 Interfaces in Java 8
4. Weitere UML - Elemente 8
5 Ein Beispiel 10
5.1 Aufgaben 14
6 Darstellung zeitlicher Abläufe in UML 15
6.1 Sequenzdiagramme 16

© Hans-Georg Beckmann 2004

UML im Unterricht

Bei der Entwicklung von Programmen, die in Java oder einer anderen Programmiersprache geschrieben werden sollen, haben sich im Unterricht Struktogramme bewährt, die wir in den vorangegangenen Abschnitten auch schon oft benutzt haben. Dabei hat man jedoch keine übersichtliche schematische Darstellung der verwendeten Objekte bzw. Klassen sondern eher eine Planung des Programmablaufs vor sich. Für Schülerinnen und Schüler ist es aber wichtig, eine übersichtliche Darstellungsmethode für Klassen und deren Beziehungen zu haben, da dabei schon Festlegungen zu treffen sind, die für den späteren Programmaufbau entscheidend sind.
Daher ist es sinnvoll, schon früh im Unterricht mit einer passenden Darstellungsweise zu beginnen, die dann erweitert werden kann.
In der objektorientierten Programmierung wird oft zur Veranschaulichung und Planung die UML- Notation verwendet. Dabei steht UML für *unified modeling language* = vereinheitlichte Modellierungssprache.

Über UML kann man ganze Bücher schreiben [1](was natürlich auch schon geschehen ist).
Wir werden uns auf einige wichtige Dinge beschränken und in den vielen Kapiteln , die noch folgen sollen UML-Elemente dann genauer erläutern, wenn es nötig ist.

In verschiedenen Javalehrbüchern wird die UML-Notation auch in einer jeweils individuellen Weise benutzt, was der eigentlichen Idee der Vereinheitlichung widerspricht. Man sollte sich also nicht wundern, wenn man unterschiedliche Darstellungen vorfindet.

Wir werden uns bei UML zuerst mit *Klassendiagrammen* befassen, die dazu dienen , Objekte bzw. Klassen und deren Beziehungen untereinander darzustellen.

Klassendiagramme

Klassen werden in einem Rechteck mit drei Segmenten dargestellt.

Im obersten Segment steht der Name der Klasse. Er wird in vielen Büchern fett und zentriert gedruckt. In einigen Büchern ist dieses Segment aber auch grau unterlegt, um es hervorzuheben.

Im zweiten Segment stehen die Attribute der Klasse. Dabei wird immer zuerst der Attributname geschrieben, gefolgt von einem Doppelpunkt und dem Typ. Das entspräche einem Syntaxdiagramm, wie wir es schon kennen:

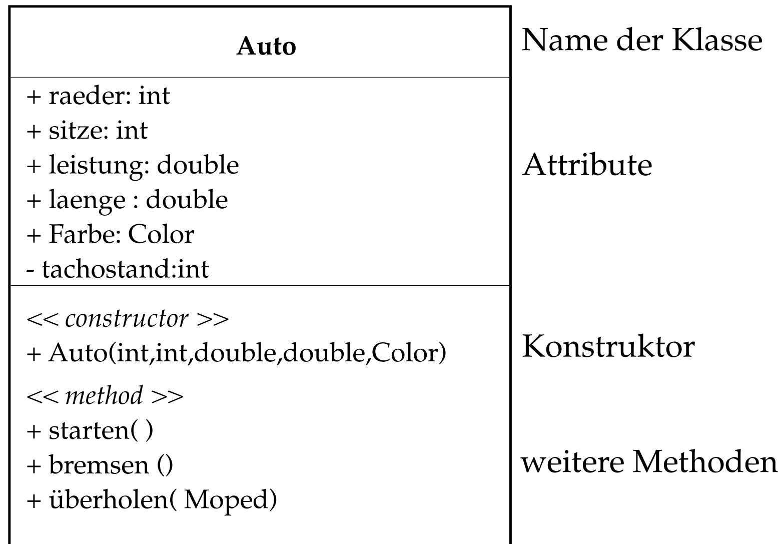
Vor dem Attribut wird aber noch eine Art "Vorzeichen" plaziert, das angibt, ob das Attribut öffentlich (*public*) zugänglich sein soll oder nur intern (*private*) oder ob es unveränderbar (*protected*) sein soll. *public* wird durch ein **+** - Zeichen dargestellt, *private* durch ein **-** - Zeichen und *protected* durch das # - Zeichen.
Hier muss man beachten, dass dann im eigentlichen Javaprogramm die Reihenfolge von Attributname und Typ genau anders herum festgelegt ist.

[1] z.B. Helmut Blazert : "Lehrbuch Grundlagen der Informatik", Spektrum Akademischer Verlag

© Hans-Georg Beckmann 2004

Im dritten Segment stehen dann alle Methoden der Klasse, die ebenfalls mit einem Vorzeichen versehen werden.
Da zur Erschaffung eines Objekts (einer Instanz einer Klasse) ein Konstruktor nötig ist, wird vor diese spezielle Methode das Wort << *constructor* >> *oder* << *Konstruktor* >> gesetzt und vor alle anderen Methoden << *method* >> *oder* << *Methoden* >>.
Diese Kennzeichnung ist nicht in allen UML-Darstellungen zu finden, aber recht hilfreich.

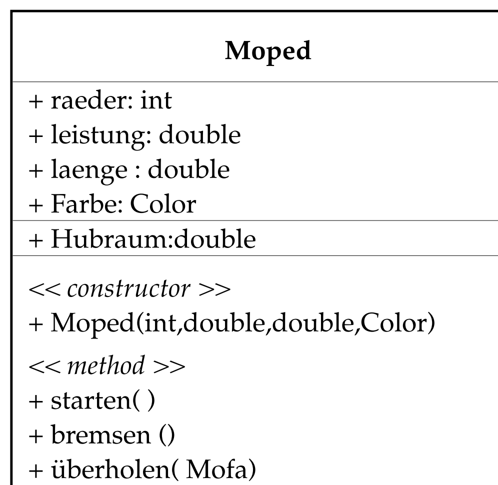
Nun aber ein Beispiel:



Mit dem Aufruf

```
meinAuto = new Auto(4,5,102,4.53,Color.red);
```

wird dann ein Exemplar der Klasse Auto erzeugt. Das Auto hat den Namen meinAuto, es hat zum Glück 4 Räder, 5 Sitze, eine Leistung von 102 kW und es ist schön rot.
Passend dazu gibt es dann auch eine Klasse Moped:



© Hans-Georg Beckmann 2004

Hier wird mit

```
deinMoped = new Moped(2,12.5,1.90,Color.blue,125)
```

ein Moped mit 2 Rädern,12,5 kW Leistung, 1,90 m Länge, blauer Farbe und 125 ccm Hubraum erzeugt, das auch noch Mofas überholen kann.
Man erkennt, dass einigen Methoden auch Parameter übergeben werden. Wenn

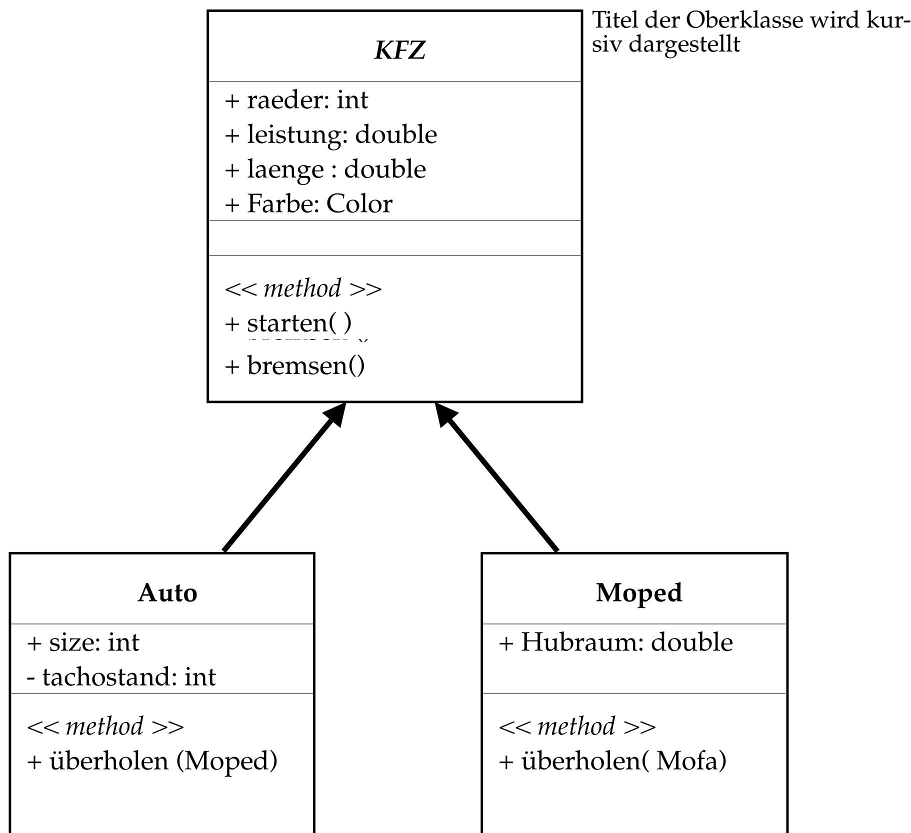
```
meinAuto.überholen(deinMoped)
```

aufgerufen wird. Dann wird die Instanz *meinMoped* der Klasse *Moped* an die Methode *überholen* der Instanz *meinAuto* der Klasse *Auto* übergeben. Wenn die Methode als Ergebnis die übergebene Instanz verändert, dann würde in der UML-Darstellung die entsprechende Zeile

```
überholen(Moped) : Moped
```

lauten.

Offensichtlich sind Auto und Moped zwei spezielle Klassen einer "Oberklasse" , die z.B. KFZ heißen könnte.
Will man das in einer UML-Darstellung zeigen, dann sieht das wie folgt aus:



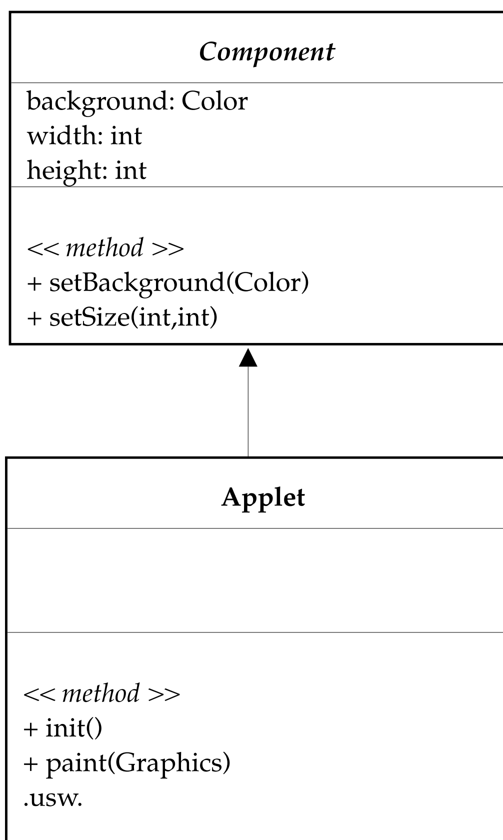
© Hans-Georg Beckmann 2004

Die Oberklasse KFZ **vererbt** nun die gemeinsamen Attribute an die beiden Unterklassen Auto und Moped. Wenn also die Klasse Auto erzeugt wird mit:

class **Auto** *extends* **KFZ**

dann hat die Klasse Auto ohne weitere Angaben schon alle Attribute und Methoden der Oberklasse KFZ geerbt und muss nun nur noch die Dinge festlegen, die über die Inhalte der Oberklasse hinausgehen.
Wenn von einer Oberklasse keine eigenen Instanzen abgeleitet werden können / sollen, dann heißt so eine Klasse *abstrakte* Klasse. Unser oben angegebenes Beispiel kann man so auffassen.

Vererbungen haben wir bereits kennengelernt.



Ein oft gesehenes Beispiel ist die Klasse *Applet*, deren Oberklasse *Component* ist:
Die Eigenschaften von Component übertragen sich automatisch auf ein Applet, auch wenn sie in dessen Bauanleitung gar nicht vorkommen.

Ein großes Problem für gequälte Programmierer besteht darin, nicht immer zu wissen, welche Oberklassen zur Verfügung stehen und welche Attribute und Methoden damit benutzt werden können

Nur wenn man alle Klassen (nach sorgfältiger Planung mit Hilfe von UML) selbst konstruiert hat, kann man sicher sein, alle Vererbungen unter Kontrolle zu haben.
In allem anderen Fällen hilft nur Nachlesen - leider.

Die bis hierher dargestellten Inhalte sollen nun noch an weiteren Beispielen verdeutlicht werden.
Wir erzeugen eine abstrakte Klasse **Spieler**, die Personen beim 17 und 4 - Spielen darstellen soll. Die Spieler haben eine bestimmte Augenzahl auf der Hand, wenn sie ihren Einsatz machen. Dabei können die Spielstrategien sehr unterschiedlich sein. Man stelle sich den Spielertyp "Angsthase" vor, der immer nur 50% seines Geldes einsetzt, wenn er als erste Karte eine 10 oder ein As bekommt, sonst setzt er immer nur 20% seines Kapitals. Dagegen spielt der Typ "Supermann" so, dass er bei 10 oder As 100% seines Einsatzes riskiert und sonst immer 50%.

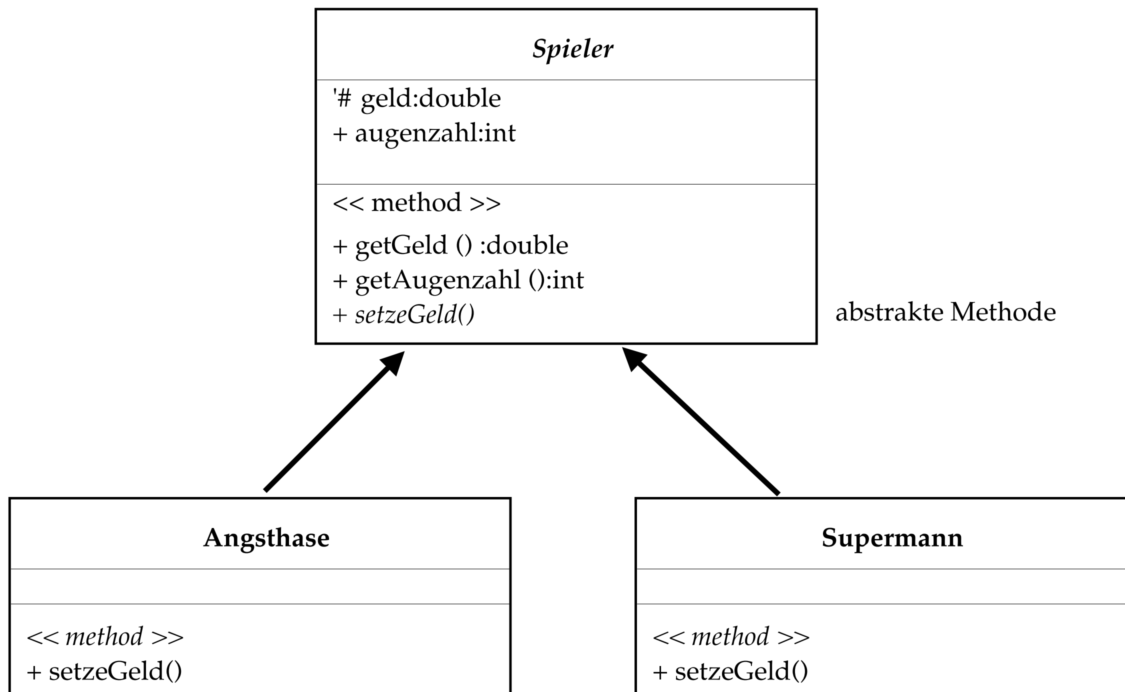
Die abstrakte Klasse Spieler würde eine abstrakte Methode *geldsetzen*(int) enthalten, die noch nicht weiter ausgeführt wird. Wenn dann zwei Unterklassen definiert werden, in denen das

© Hans-Georg Beckmann 2004

Spielverhalten genauer spezifiziert werden muss, dann wird diese *abstrakte* Methode überschrieben.

In der abstrakten Oberklassen werden wieder alle die Attribute festgehalten, die an alle Unterklassen vererbt werden sollen.

Im UML-Diagramm kann das dann, wie folgt aussehen:



Erst wenn wir durch einige Ausschnitte eines möglichen Javaprogramms hier Leben in die Klassen bekommen, wird klar, was gemeint ist.

```
public abstract class Spieler
{
   protected double geld = 500;       // am Anfang 500 Euro
   public int augenzahl;              // muss noch festgestellt werden

   public double  getGeld()
   {
     return(geld);// gibt nach aussen bekannt, wieviel Geld da ist
   }

   public int  getAugenzahl()
   {
     return(augenzahl);               // gibt die aktuelle Augenzahl bekannt
   }

   public abstract void setzeGeld(); // bleibt leer, weil abstract
} // Ende von class Spieler
```

© Hans-Georg Beckmann 2004

Das war die abstrakte Klasse, von der nun die beiden weiteren Klassen abgeleitet werden:

```
public class Angsthase extends Spieler
{
    public void setzeGeld()
    {
    if (augenzahl >=10)  { geld = geld-0.5*geld;}
    else                 { geld = geld- 0.2*geld;}
    }
}


public class Supermann extends Spieler
{
    public void setzeGeld()
    {
    if(augenzahl >=10) {geld=0;}
    else               {geld=geld-geld*0.5;}
    }
}
```

Nun betrachte man die folgenden Programmzeilen, die den Rest des Applets zeigen:

```
import java.awt.*;
import java.applet.Applet;

public class Vererbung extends Applet
{
   Angsthase  erwin= new Angsthase(); // neuer Angsthase
   Supermann  egon = new Supermann(); // neuer Supermann

public void init()
   {
   erwin.augenzahl=9;                // eriwn bekommt 9 Augen
   erwin.setzeGeld();                // erwin setzt Geld

   egon.augenzahl=10;                // egon bekommt 10 Augen
   egon.setzeGeld();                 // egon setzt Geld
   repaint();
   }

public void paint( Graphics g )
   {
      g.drawString("Restgeld ist :" + erwin.geld,30,30);
      g.drawString("Restgeld ist bei diesem Supermann :"+egon.geld,30,45);

   }
}  // Achtung !Ende von Applett. Die oben aufgeführten Klassen kommen hinterher.
```

Man beachte, dass die abstrakte Klasse nicht innerhalb des Applets definiert werden kann, sondern außerhalb. Ganz elegant wäre `es,` die Klasse in einer eigenen Datei zu lassen, was aber hier nicht nötig ist.

© Hans-Georg Beckmann 2004

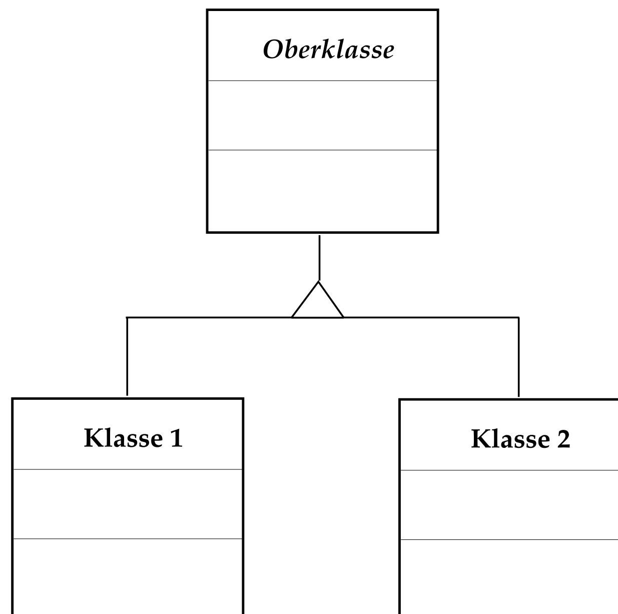
Hier ist z.B. noch nicht darüber nachgedacht worden, was passiert, wenn das Geld alle ist und wie man es wohl bewerkstelligt, wenn ein Gewinn eingefahren wird. Das alles ist aber nicht so wichtig, da es nur um das Prinzip der Vererbung an diesem einfachen Beispiel ging (siehe Aufgaben).

Interfaces in Java

Java erlaubt keine Mehrfachvererbung. Das bedeutet, es ist nicht möglich, dass eine Klasse aus mehr als einer Oberklasse erbt.
Man vermeidet damit eine ganze Reihe möglicher Probleme, die z.B. auftreten könnten, wenn in mehr als eine Oberklasse Methoden oder Attribute mit gleichem Namen zu finden wären.
Dafür gibt es in Java sehr spezielle Oberklassen, aus denen geerbt werden kann, die Schnittstellen (Interfaces).
Ein Interface ist eine Klasse, die ausschließlich Konstanten und abstrakte, öffentliche Methoden enthält jedoch keine Konstruktoren.
Das Schlüsselwort bei der Definition von Interfaces ist dann nicht *class*, sondern *interface.*
Bei der Benutzung von interfaces wird dann auch nicht das Schlüsselwort *extends* verwendet, sondern *implements*.
Beispiele habe wir schon mehrfach kennengelernt (... implements ActionListener ... oder implements Runnable....)

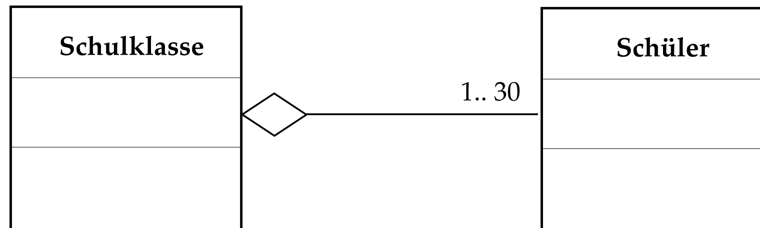
Weitere UML - Elemente

Alternativ zu der Darstellung der Vererbung, wie sie oben zu sehen ist, gibt es in einigen Lehrbüchern alternative Pfeildarstellungen:

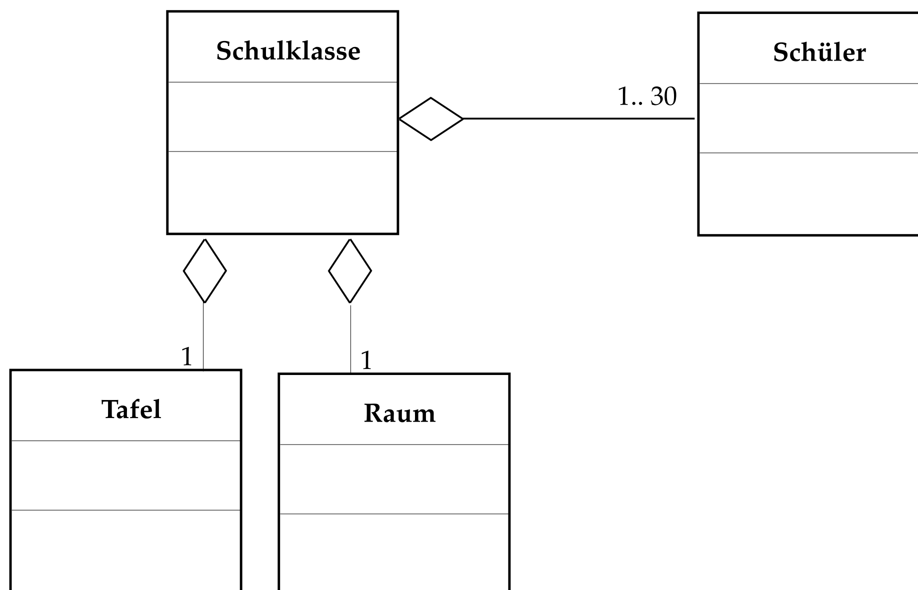


© Hans-Georg Beckmann 2004

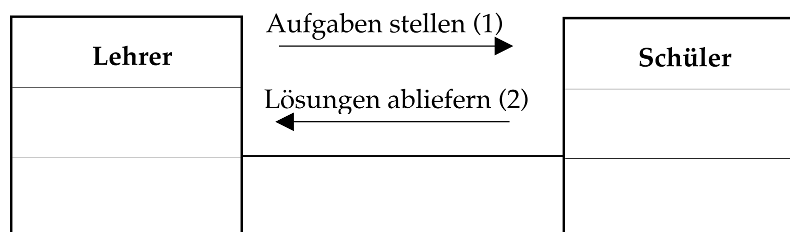
Ist eine Klasse komplett in einer anderen Klasse enthalten, dann wird die Pfeilspitze durch eine Raute ersetzt. Dabei ist es egal, ob die beiden Klassenrechtecke übereinander oder nebeneinander angeordnet sind.
Auf der Linie wird dann noch angegeben, wie viele Objekte der Unterklasse in der Oberklasse enthalten sein können.



Das kann auch für den Fall so dargestellt werden, dass weitere Unterklassen enthalten sind.



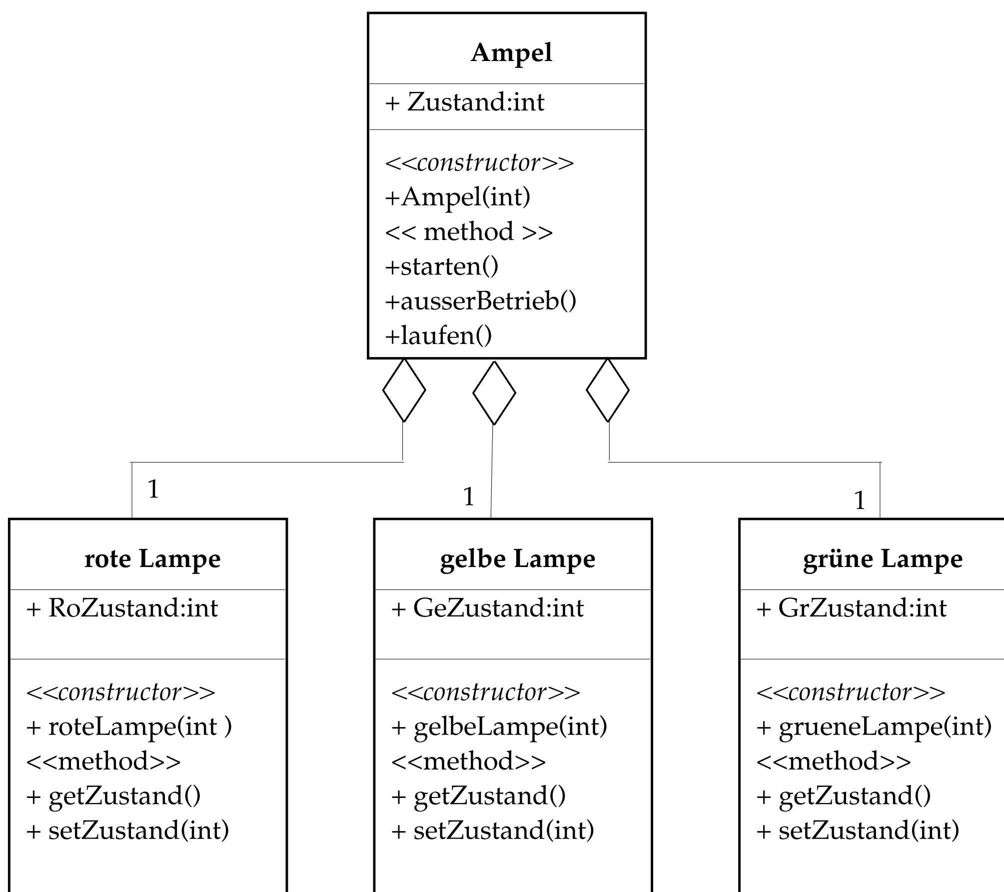
Bei Ablauf von Programmen "unterhalten" sich die Instanzen verschiedener Klassen möglicherweise miteinander. Diese Nachrichtenübermittlung ist der schwierigste Teil in der OOP. Im UML-Diagramm hat man mehrere Möglichkeiten, das darzustellen.
Gibt es einen Informationsfluss zwischen zwei Klassen, dann wird das durch einen Pfeil dargestellt, der vom Sender zum Empfänger zeigt und mit dem Nachrichteninhalt beschriftet wird. Gibt es mehrere Nachrichten, dann gibt es auch mehrere Pfeile. Sind die Nachrichten in einer bestimmten zeitlichen Reihenfolge abzuarbeiten, kann man die Pfeile einfach numerieren.
Bleibt man bei dem obigen Beispiel *Schüler* und es kommt nun noch die Klasse *Lehrer* hinzu, dann könnte die Beziehung wie nebenstehend dargestellt werden:



© Hans-Georg Beckmann 2004

Ein komplexeres Beispiel soll die verschiedenen Möglichkeiten der Darstellung von Klassen und deren Beziehungen noch einmal verdeutlichen. Aus der UML-Darstellung soll dann auch ein Javaprogramm entwickelt werden.

Das Beispiel ist nicht besonders aufregend, aber sehr übersichtlich: eine Verkehrsampel.
Sie hat drei Lampen (rot, gelb und grün). Sie soll im Normalbetrieb laufen und dann mit einem bestimmten Zeittakt schalten (rot -- rot,gelb-- grün -- gelb -- rot ---). Am Anfang soll sie gestartet werden können , wobei die rote Lampe leuchten soll. Bei Betriebsausfall soll die gelbe Lampe blinken und alle hoffen, dass die Verkehrsteilnehmer noch die Vorfahrtsregeln können.
Es ist sicher vernünftig, von den Lampen zu verlangen, dass man sie ein - und ausschalten kann, und dass es ist möglich ist, ihren Zustand abzufragen.
Eine Möglichkeit der Darstellung könnte sein:



Hier sieht man schon, dass es wohl möglich wäre , für die drei Lampen eine gemeinsame Oberklasse festzulegen oder die drei Lampen als unterschiedliche Instanzen einer Klasse zu nehmen, wenn als Attribut etwa noch "Farbe" hinzukommen würde, was aber nicht notwendig ist.. Aber dazu später.
Die Zustände, die für die Lampen und für die Ampel gelten sollen, sind hier willkürlich über eine int-Größe festgelegt:
Ampel:
Zustand=0 bedeutet Anfangszustand = rotes Dauerlicht;
Zustand =1 bedeutet Ampel läuft ;
Zustand = 2 bedeutet, Ampel ist außer Betrieb = gelbes Blinklicht;

© Hans-Georg Beckmann 2004

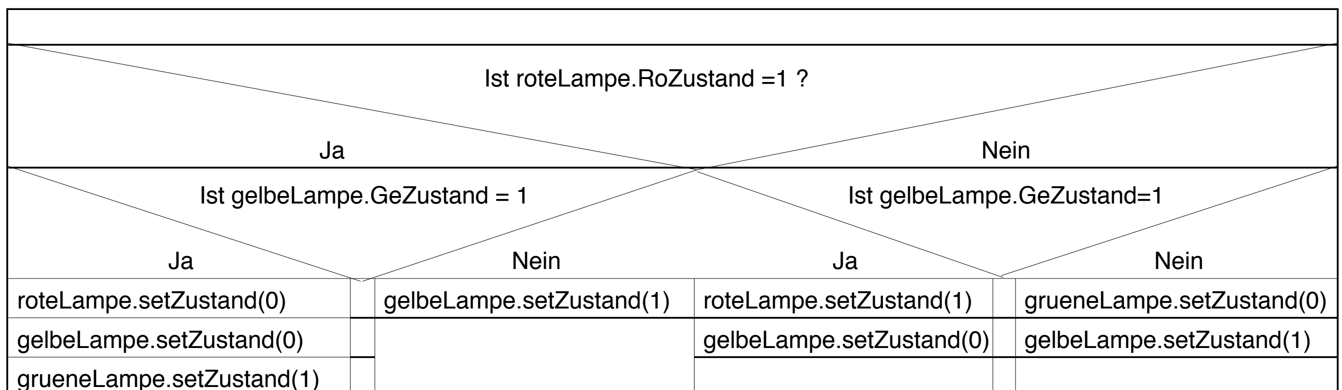
Lampen:
Zustand = 0 bedeutet, Lampe ist aus;
Zustand=1 bedeutet ,Lampe ist an;

In dieser Darstellung der Klassen steckt natürlich auch schon eine Ablaufidee, die in umgangssprachlicher Formulierung lautet:

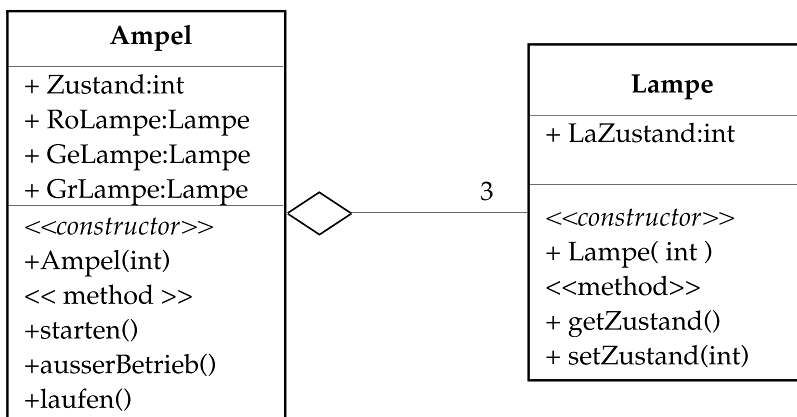
> Erzeuge die drei Lampen
> Erzeuge die Ampel mit dem Zustand "Start" (Lampe rot leuchtet)
> Rufe "starten" für die Ampel auf
> Wiederhole bis *ausserBetrieb* gesetzt.
> Hole Zustand von *roterLampe*, von *gelberLampe* und von *grünerLampe*
> Wenn nur *roteLampe* an, dann schalte auf rot und gelb
> Wenn *roteLampe* an und *gelbeLampe* an dann schalte auf grün
> Wenn nur *grüneLampe* an dann schalte auf gelb
> Wenn nur *gelbeLampe* an, dann schalte auf rot
> Wenn *ausserBetrieb*, dann blinke gelb

Dieser zeitliche Ablauf wird in der Methode *laufen*() der Ampel untergebracht.

Als Struktogramm:



Man kann hier sehen, dass die Abfrage nach dem Zustand der grünen Lampe eigentlich nicht sein muss, da sich dieser Zustand aus dem Zustand der anderen beiden Lampen ergibt. In diesem Struktogramm ist der Fall "gelbes Blinklicht" auch noch nicht erfasst.



Nun ist aber wohl doch sinnvoll, die drei Lampen aus **einer** Klasse zu gewinnen, womit die UML-Darstellung anders aussieht. Man sieht jetzt, dass die Klasse Ampel 3 Instanzen der Klasse Lampe enthält. Das oben erarbeitete Struktogramm bleibt aber im Prinzip erhalten.

© Hans-Georg Beckmann 2004

Im Programmtext könnte es nun wie folgt aussehen:

.....

```
Ampel     neueAmpel= new Ampel(0);
```

das erzeugt eine neue Ampel mit Startzustand 0.

....

und dann

....

```
public class Ampel
{
       int         Zustand;
       Lampe       RoLampe = new Lampe(1);
       Lampe       GeLampe = new Lampe(0);
       Lampe       GrLampe = new Lampe(0);

  public Ampel(int Z) // Konstruktor
       {
       Zustand = Z;
       }
  public void starten()
       {
       Zustand=0;
     }
  public void ausserBetrieb()
       {
       Zustand = 2;
       }

  public void laufen()
       {
       Zustand=1;
       // der Rest des Programmcodes kommt hier
       }

}      // Ende von Klasse Ampel

public class Lampe
{
        int LaZustand;
   public Lampe (int Z)      //Konstruktor
   {
   LaZustand=Z;
   }
   public int getZustand()
   {
   return (LaZustand);
   }
   public void setZustand(int Z)
   {
   LaZustand=Z;
   }

}// Ende von class Lampe
```

© Hans-Georg Beckmann 2004

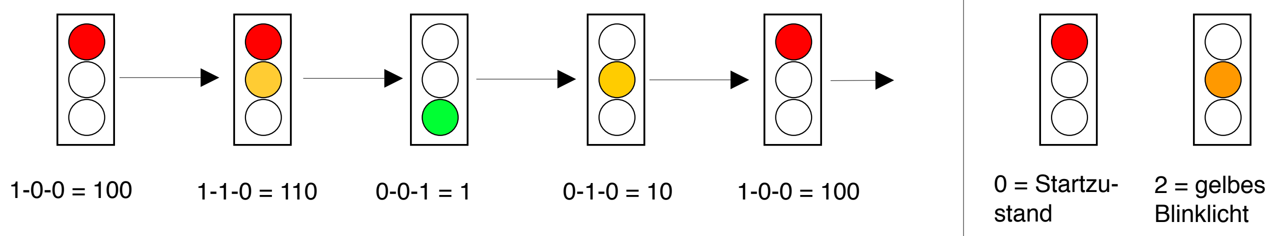
Nun muss man nur noch die Reste des Programms hinzufügen, was möglicherweise mehr Probleme macht , als das UML-Diagramm.
Es geht um eine Ausgabe am Bildschirm, um die Realiserung des Blinkes (gelbe Lampe) und um die Frage, wie man die Ampel von außen (z.B. über ein Key-Event oder über Buttons) beeinflussen kann.

Es geht auch anders

Es sieht bisher alles ganz nett aus. Dennoch merkt man schnell, dass diese Umsetzung eigentlich viel zu kompliziert ist. Man braucht eigentlich für die ganze Ampel nur einen Integerwert als Innenleben:
Wenn man nebenstehendes Schema sieht, wird klar, dass sich die Ampel durch drei Ziffern darstellen läßt, die - richtig interpretiert - den Zustand der Ampel und damit auch einen möglichen Folgezustand zeigen:

Ampel
+ Zustand:int
<<constructor>> + Ampel() <<method>> + getZustand() + laufen()

Die Methode *laufen* tut dann intern in der Ampel nichts anderes, als den Zustand weiterzustellen und unser schönes Struktogramm ist eigentlich wertlos:




```
public void laufen()         // schaltet die Ampel um einen Schritt weiter
     {
          switch (Zustand)
          {
          case(100):
                    Zustand=110;break;
          case(110):
                    Zustand=1;break;
          case(1):
                    Zustand=10;break;
          case(10):
                    Zustand=100;break;
          default: {}
          }         // Ende von switch

     }    // Ende von laufen()
```

Man sieht auch, dass in dieser Darstellung die Funktion der Ampel von ihrer Darstellung (z.B. am Bildschirm) vollkommen getrennt ist. Wenn man die graphisch darstellen möchte und die Arbeit der Ampel mit Mausklicks simulieren will, ist man gut beraten, die dazu notwendigen Klassen und Methoden nicht in der Klasse Ampel unterzubringen. Man spricht auch von der Trennung der sog. Fachklassen und der GUI-Klassen. (GUI = Graphic User Interface)

© Hans-Georg Beckmann 2004

Die Methode, in der die Ampelfunktion dann graphisch realisiert werden könnte ist z.B. die Methode *paint* , die wir allemal im Applet überschreiben. Die könnte wie folgt aufgebaut sein:

```
public void paint( Graphics g )
    {
         Ampelzustand=neueAmpel.getZustand();// neueAmpel wurde vorher erzeugt
         switch(Ampelzustand)
         {
         case 0:
         { // male rotes Dauerlicht } break;
         case 2:
         { // male gelbes Dauerlicht } break;
         case 100:
         { // male rot - nicht gelb - nicht grün} break;
         case 110:
         { // male rot - gelb - nicht grün} break;
         case 1:
         { // das entspricht 001 also nicht rot- nicht gelb - grün} break;
         case 10:
         { // das entspricht nicht rot - gelb - nicht grün } break;
         default: {}
    }    // Ende von switch
    }
```

Aufgabe 1:

Bringen sie die Ampel zum Laufen.

Aufgabe 2:

Kehren sie zum 17 und 4 Programm zurück. Schreiben sie ein Programm, das Folgendes leistet: Es kommt eine Klasse "Bank" hinzu, die des Bankhalter darstellen soll. Auch der Bankhalter sollte nach einem festen Prinzip spielen.
Ändern sie die Spielstrategie von "Supermann" um. Er würde ja in jedem Falle alles verlieren, sobald er auch nur ein einzelnes Spiel verliert. Es ist z.B. denkbar, dass er auf Kredit weiter spielt, oder aber dass er seine Setz-strategie ändert, wenn er das erste Mal gewonnen hat.
Das Programm soll dann mehrere Spiele nacheinander ausführen können.
Erzeugen sie nun noch ein Kartenspiel (z.B. als Array), das die 32 Karten eines Skatblattes simuliert. Aus die-sem Kartenspiel werden die Karten für Spieler und Bank zufällig gezogen. Beim zweiten und den folgenden Spielen dürfen schon benutzte Karten natürlich nicht wieder verteilt werden. Es muss aber möglich seit durch eine Art "Reset" wieder alle 32 Karten für eine neue Serie von Spielen bereit zu stellen.

Das Programm kann dann noch beliebig verschönert werden. (Buttons, Grafiken .....)

© Hans-Georg Beckmann 2004

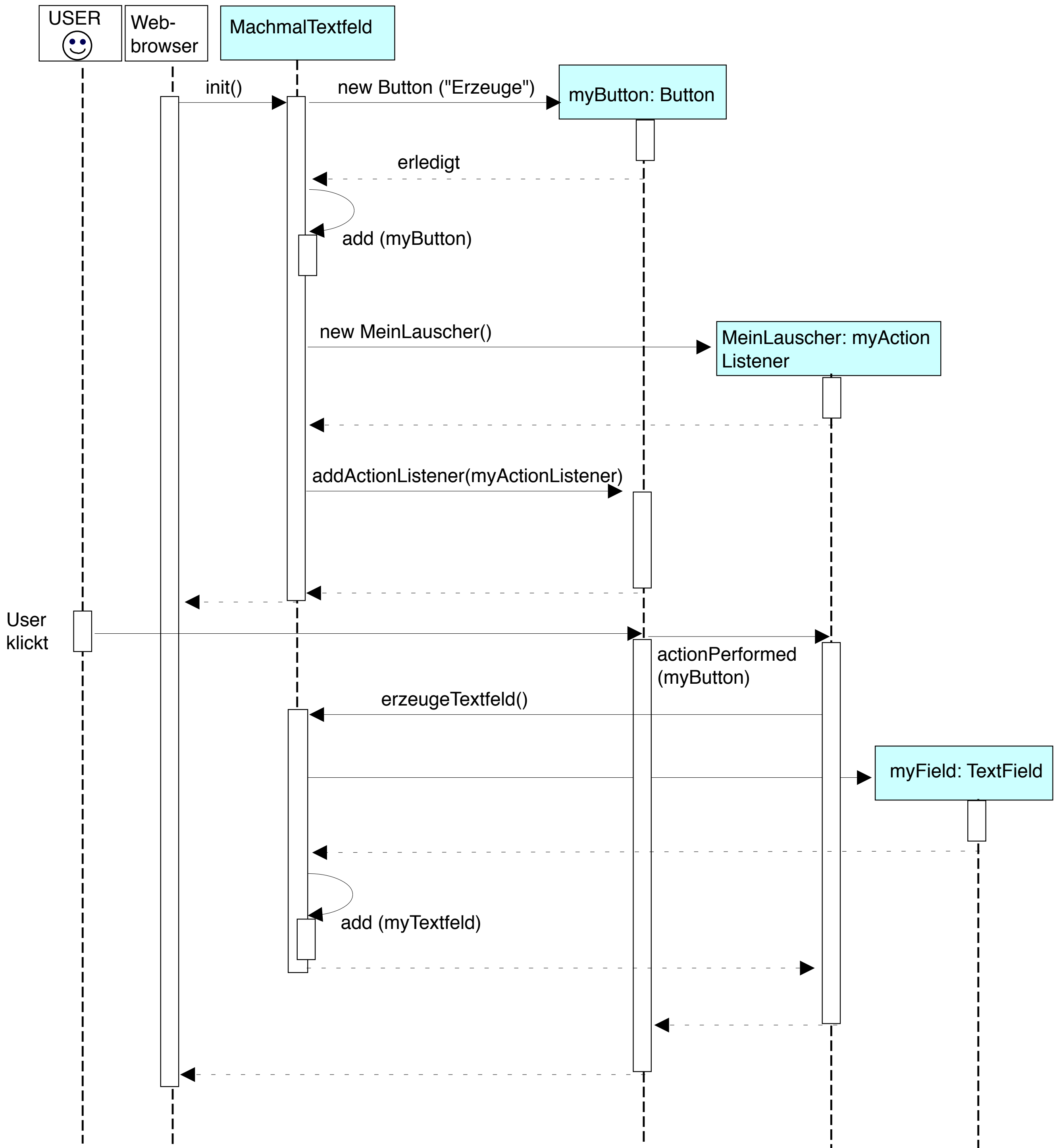
Darstellung zeitlicher Abläufe in UML

Bei beiden oben dargestellten Beispielen hatte man es mit zeitlichen Abläufen zu tun. Diese zeitlichen Abläufe sind nicht in den Klassendiagrammen zu erkennen. Sie befinden sich oft in einzelnen Methoden. Das weit oben dargestellte Struktogramm könnte so einen zeitlichen Ablauf zeigen.
Schwierig wird es aber bei der Planung eines GUI-Programms in dem bestimmte Objekte erst erzeugt werden (z.B. Buttons) die dann betätigt werden können (oder auch nicht) und dann bestimmte Dinge auslösen.

In UML gibt es zur Darstellung des zeitlichen Ablaufs **Sequenzdiagramme**.
In so einem Diagramm gibt es immer eine Zeitachse, die von oben nach unten zeigt (für diese Festlegung gibt es aber keinen einsichtigen Grund - man muss es so hinnehmen).
Objekte, die Botschaften mit andern Objekten austauschen, werden auf einer eigenen vertikalen gestrichelten Zeitlinie (Objektlinie) dargestellt. Tut das Objekt irgend etwas, dann wird die Linie für die entsprechende Zeit als schmales Rechteck gezeichnet. Wird ein Objekt gelöscht, dann endet seine "Lebenslinie" mit einem Kreuz.
Im Diagramm wird jede Nachricht als Pfeil vom Sender zum Empfänger gezeichnet. Am Ende einer Botschaft wird manchmal auch ein Pfeil vom Empfänger zum Sender zurückgezeichnet , um zu zeigen , dass die Aufgabe erledigt ist (das muss aber nicht sein).
Im Diagramm ganz links wird in vielen Fällen der Benutzer , der hoffentlich die ganze Zeit aktiv ist, als "oberstes Objekt" dargestellt.
Ein Beispiel[2] macht das alles sehr viel klarer.
Man nehme an , es läuft in einem Webbrowser ein Applet ab, das einen Button zeigt, der den Titel "Erzeuge" hat. Wenn der Benutzer diesen Knopf mit der Maus klickt, dann soll,ein Textfeld erzeugt werden, dass die Beschriftung "Hallo" (was auch sonst ?) trägt.
Das ganze Applet soll "*MachmalTextfeld*" heissen und es hat die Methode *init*(), die den Bildschirm mit den nötigen Elementen erzeugt. Der Button bekommt einen ActionListener und damit ist auch klar, wo hier Botschaften zu erwarten sind:

[2] Blazert: Lehrbuch Grundlagen der Informatik Seite 183 f

© Hans-Georg Beckmann 2004



Man muss dieses Diagramm in Ruhe studieren, und wird dann aber den zeitlichen Ablauf nachvollziehen können. Diese Art Diagramme hat aber auch Grenzen. Es fällt schwer, einen zeitlichen Ablauf zu planen, der erst bei Benutzung des Programms entsteht.. Schon wenn ein zweiter Button hinzukommt, wird es problematisch. Es könnte sein, dass der User den Knopf 1 zuerst drückt und damit eine Veränderung an Knopf 2 vornimmt . Diese Veränderung tritt aber möglicherweise nicht ein, wenn Knopf 2 zuerst gedrückt wird. Wie schon einmal in einem anderen Kapitel er-

© Hans-Georg Beckmann 2004

wähnt, ist es der große Vorteil aktueller Programme, dass Benutzerinnen und Benutzer nicht gezwungen werden, Dinge in einer bestimmten zeitlichen Reihenfolge zu tun, sondern die freie Entscheidung haben, was als Nächstes passieren soll (in gewissen Grenzen).

Der Einsatz solcher Sequenzdiagramme im Unterricht ist daher mit Vorsicht zu handhaben.

Man muss für den Unterricht auch bedenken, dass der formale Balast bei der Programmentwicklung nicht zu groß werden darf, weil sonst die Problemslösungen dahinter verschwinden. Auch ist mancher formale Aspekt von UML in der Praxis schon deswegen schwer zu bewältigen, weil ganz triviale Probleme dem entgegenstehen. Stellen sie sich bitte vor, ein Klassendiagramm wird an der Tafel (ja , ja so was hat man noch) entwickelt und den Schülerinnen und Schülern wird mitgeteilt, welche Kreideworte nun "*kursiv*" oder **"fett"** zu lesen sind.
Wenn der "Powerpointterror" im Unterricht weiter voranschreitet , ist das ja in Zukunft vielleicht dann kein Problem mehr.
In vielen Büchern hat es sich auch wieder eingebürgert, bei der Planung von Programmen (Methoden) Flussdiagramme einzusetzen,von denen wir gehofft hatten, dass sie nicht mehr auftauchen. Wir werden darauf hier verzichten und mit Struktogrammen arbeiten, die für den Unterricht besser geeignet sind.

Aufgabe:

Versuchen sie ein Sequenzdiagramm für die Ampel zu erstellen