Harro von Lavergne

Über Züge, Ampeln und Objekte
oder
Das A und O der OOA ist die Analyse

- Eine Lokomotive ist eine Lokomotive ist eine Lokomotive
- Eine Ampel ist eine Ampel ist eine Ampel
- Eine sémaphore ist (k)eine Ampel
- Ein Objekt ist eine Instanz einer Klasse
- Die Klassenanalyse ist in vielen wissenschaftlichen Disziplinen ein Problem

La trahison des images

Eine reale Lokomotive ist zweifellos zu schwer für einen Computer und einen Schüler. Wie dagegen die mehr oder minder gelungene Abstraktion (Strichzeichnung oder was immer) eines Lokomotivobjekts am Bildschirm aussieht, ist strukturell unerheblich. Das gilt natürlich ebenso für Ampeln.

Eine Ampel kommt selten allein. Wenigstens zwei korrelierte Signalgeber sind eine Ampelanlage - eine singuläre Ampel erinnert an Gesslers Hut.

Sowohl eine Lokomotive als auch eine Ampel haben Funktionalität, sie bewegen materielle Güter, sie steuern Prozessabläufe. Diese Stichworte heben selbstverständlich nur jeweils einen Aspekt aus der Vielfalt möglicher Betrachtungsweisen heraus, es gibt Dutzende andere.

Damit ist man bei der Frage der (objektorientierten) Analyse angekommen. Welche Aspekte des betrachteten Gegenstands oder Prozesses sollen analysiert werden? Welches sind adäquate Mittel der Repräsentation dieser Aspekte in einem Rechner? Wie sollen und wie können diese Aspekte modelliert werden?

Das bis vor kurzem - oder auch jetzt noch - verbreitetste Credo zur Analyse und Modellierung lautet: *Objektorientierung von Anfang an!* Nun kann Objektorientierung klarerweise kein Selbstzweck sein. Sie bringt in den Programmentwurf zusätzliche Komplikationen, die lediglich zu rechtfertigen sind, wenn das betrachtete Problem Objektorientierung nicht nur möglich macht (was fast immer der Fall ist), sondern dabei auch relevante Gewinne zu erwarten sind.

Objektorientierung bedeutet Abstraktion, der mit ihr verbundene Aufwand ist nur dann sinnvoll, wenn die jeweilige Datenstruktur in Varianten auftritt und diese Varianten gemeinsam verwaltet werden sollen. In allen anderen Fällen lohnen sich die für den Abstraktionsprozess erforderlichen Mehrinvestitionen nicht (s. Kasten). In diesem Sinn liest man in einigen neueren Publikationen die eine oder andere vorsichtige Kritik an dem Credo der universellen Objektorientierung; zu viele forsche Versuche seiner Umsetzung scheinen mißlungen, es wird eine Korrektur angemahnt - in Grenzen.

René Magritte
La trahison des images

Unberührt davon versuchen andere Autoren, die auftretenden Schwierigkeiten durch den Einsatz immer neuer “Hilfsmittel” zu überwinden. Waren die Hilfsmittel ursprünglich (und sind es teilweise auch heute noch) handgeschnitzte Systeme mit - scheinbar - schülergerecht einfachen Schnittstellen, werden inzwischen sowohl der professionelle als auch der schuldidaktische Markt mit immer mehr und immer neuen Wunderwerkzeugen überflutet, wobei es sich doch immer wieder nur um die eine oder andere Variante altbekannter CASE-Tools handelt.

Und merkwürdigerweise finden sich immer wieder Menschen, die endlich den Stein der Weisen gefunden zu haben glauben, ein Mittel, das ihnen genau die Arbeit abnimmt, die sie eigentlich erlernen oder lehren wollten. (Als ich anfing zu unterrichten, hieß das ultimate dieser Wunderwerkzeuge: *LaßMichInRuheMacheTollesProgrammUndVermarkteEsMitÜberweisungDerEinnahmenAufMeinKonto.EXE*).

Das Thema Werkzeuge und die daran geknüpften Hoffnungen scheinen ein ewiges Mantra zu sein, dessen Sinnhaftigkeit - oder Sinnlosigkeit - stets aufs Neue beschworen werden muss. Entsprechend findet man in nur vier der in jüngerer Zeit erschienenen Hefte der Zeitschrift LOG IN (127; 128/129; 130; 131/132; s. Literaturverzeichnis) eine respektable Reihe von Beiträgen, die sich mit der Problematik von Objektorientierter Analyse (Object Oriented Analysis, OOA) und Objektorientiertem Entwurf (Object Oriented Design, OOD) sowie - im Zusammenhang damit - dem Einsatz von Werkzeugen im Unterricht befassen und zu teilweise recht widersprüchlichen Ergebnissen kommen.

So liest man bei Balzert/Balzert [BaBa04] zu den didaktischen Problemen der Objektorientierung:

Beim Erlernen einer Programmiersprache (...) kann dagegen, um die Lernenden nicht zu überfordern, nicht sogleich mit der Einführung des Objekt- und Klassenbegriffs begonnen werden; es sollen vielmehr zunächst - im herkömmlich prozeduralen Stil - die Grundkonzepte von Algorithmen und elementaren Datenstrukturen erarbeitet werden.

Und Heubaum [Heu04] schreibt zu der Frage, wann Objektorientierung angebracht ist:

Welchen Nutzen bringt nun der Übergang zum objektorientierten Programmieren? Zunächst fallen gewaltige Nachteile (...) auf. Die Programme sind (...) erheblich aufgebläht und (...) wesentlich komplizierter geworden. (...) Der Zwang, die Operationen in Klassen zu lokalisieren, kann begriffliche und programmtechnische Schierigkeiten verursachen. (...) Die Schwierigkeiten (...) werden auch nicht dadurch behoben, daß man Werkzeuge wie BLUEJ oder ein “didaktisches System” einsetzt. (...) Die Frage, wofür ein so aufwendiges Begriffssystem überhaupt gut ist, wird damit nicht beantwortet.

Auf der anderen Seite finden sich Autoren wie Forbrig [For04], Leipholz-Schumacher [L-S04], Prätorius [Prä04], Spolwig [Spol04], Steinbrucker [Stein04], Thomas [Tho04], die den objektorientierten Ansatz verfolgen und sich bei seiner Umsetzung fest auf das eine oder andere Werkzeug stützen, mit dem sich das oben zitierte Credo angeblich realisieren läßt, sei es nun eine der öffentlich verfügbaren bzw. individuell maßgeschneiderten Graphik-Bibliotheken oder eins der vielen CASE-Tools.

Unter den Graphik-Bibliotheken findet man die den jeweiligen Programmiersprachen und Entwicklungsumgebungen eigenen Standardbibliotheken, wie sie Prätorius (für JAVA) und Steinbrucker (für DELPHI) verwenden. Bei Prätorius wird das angestrebte Ziel - Computergraphik - bereits im Titel genannt, ob das Thema tatsächlich für den Anfängerunterricht geeignet ist, müsste gesondert untersucht werden.

Die in dem Beitrag stehende Behauptung:

An Graphikproblemen lassen sich wesentliche Konzepte und Methoden der Informatik, insbesondere der Objektbegriff, erarbeiten.

mag zwar für Fortgeschrittene richtig sein, scheint mir jedoch im Hinblick auf den Anfängerunterricht irrelevant. Eine Rekursion auf die Aussage gibt es in dem anschließenden, vorwiegend handwerklich-technisch gehaltenen Teil nicht.

Neben den systemeigenen Graphik-Bibliotheken findet sich die wahrhaft ehrwürdige Sammlung *Stifte und Mäuse* [SuMa], die in ihren Anfängen aus einer Zeit stammt, in der es noch nötig schien und zum Ehrgeiz eines anständigen Informatiklehrers gehörte, Bildschirmgraphik selbst zu programmieren. Diese Bibliothek kritisiert Spolwig in seinem Beitrag zu recht als nicht mehr zeitgemäß, weil sie auf Grund ihrer Herkunftsgeschichte zu den von ihm dargestellten Fehlern bei der OOA - und unvermeidlich dem folgenden OOD - führt. Anschließend präsentiert er allerdings eine analoge Bibliothek und den Anspruch, mit dieser das zu leisten, was *Stifte und Mäuse* nicht leisten könne, nämlich den Schülern eine sachgerechte und erschöpfende Analyse einer Ampel(anlage) zu vermitteln.

Als deren Kern liest man:

Hier soll es (...) um die Art Ampeln gehen, die gern im Anfangsunterricht programmiert wird und auf dem Bildschirm zu sehen ist (...). Es ist allgemein bekannt, wie eine Ampel aussieht und im Prinzip aufgebaut ist: ein dunkelgrauer Kasten mit drei Lampen drin, die abwechselnd an- und ausgehen. Die objektorientierte Analyse ergibt das Klassendiagramm von Bild (...).

Bei aller Einsicht in die Notwendigkeit einer analytischen Reduktion im Anfängerunterricht muß die Frage erlaubt sein, was die explizit intendierte zeichnerische Erzeugung des simplen Abbilds einer speziellen Ampelart mit einer auch nur irgendwie ernst zu nehmenden Analyse einer Ampel zu tun hat. Spolwig ist dies sicherlich bewußt, wenn er schreibt:

Ein Elektroniker (...) würde zu einer Fachklasse kommen, die keinerlei Bezüge zur physischen Wirklichkeit enthält.

Ich setze hinzu: Dafür enthält sie die verkehrstechnische Wirklichkeit. Und ich glaube nicht, daß man ernsthaft darüber streiten kann, welche der beiden Interpretationen die richtigere und wichtigere ist. Außerdem bleibt festzuhalten, daß es sich bei der von Spolwig angegebenen “Klasse” nicht um eine solche, sondern vielmehr um einen Abstrakten Datentyp (ADT) handelt, die angegebene Struktur ist final, nirgendwo ist eine Einbindung in einen größeren Zusammenhang angedeutet, dies aber wäre die für eine Klassenbildung notwendige Voraussetzung (s. Kasten).

In ähnlicher Weise stellt Steinbrucker in seinem Beitrag zur Analyse und Modellierung einer Taschenlampe als wesentlich deren graphische Präsentation und die damit verbundene Verwendung eines GUI in den Vordergrund. Die lampenspezifische Fachklasse *TTaschenlampe* spielt im Verhältnis zur Erstellung der Graphik eine eher nebensächliche Rolle. Von einer seriösen Analyse des Themas Taschenlampe kann nicht die Rede sein. Die Verwendung einer Klasse für die zu modellierende Datenstruktur erscheint in der Einführungssequenz der vorgestellten Unterrichtsreihe alternativlos, ansatzweise verständlich wird sie erst in der Endphase der Darstellung, die eine in der Analyse nicht erwähnte Unterklasse mit einer spezialisierenden Eigenschaft (*TEdisonTaschenlampe*) einführt.

Auch dem Beitrag von Forbrig fehlt eine - den Namen verdienende - objektorientierte Analyse, die die von ihm angegebene Basisklasse *Geschirrspüler* rechtfertigen würde. Als “Analyse” gibt es statt dessen eine Variante des objektorientierten Standardtexts:

Bei der objektorientierten Analyse wird von Objekten ausgegangen, die in der realen Welt existieren. Durch geeignete (?) *Abstraktion wird aus einem realen Objekt ein Objekt eines Modells. (...) Dies* (Die Modellierung*) geschieht durch Klassen, die mit ihren Beziehungen in Klassendiagrammen dargestellt werden.*

Wiederum erscheint die Klassenbildung als Naturnotwendigkeit, Alternativen oder Begründungen für die Wahl einer Klasse als Datenstruktur sind nicht zu finden, allenfalls das am Ende stehende *Klassendiagramm mit Vererbungshierarchie*, das ähnlich wie bei Steinbrucker zwei formale Unterklassen aufweist, könnte als Indiz angesehen werden. (Unverständlich bleibt in dem Diagramm, warum die Erstellung zweier - zumindest nach außen - identischer Unterklassen erforderlich ist; beim Anblick kann der Verdacht aufkommen, daß eine Verwechslung von Klasse und Objekt vorliegt, obwohl der dem Diagramm vorausgehende Text dies eigentlich ausschließt.)

In den übrigen Beiträgen liest man Darstellungen der vielfältigen Versuche, in der Schule (semi-) professionelle, teils UML-basierte CASE-Tools einzusetzen (*UML = Uninterrupted Marketing Literature*, wie ein Informatiker der ETH Zürich bei einem Kongress mal spöttelte). Immerhin werden in einigen dieser Publikationen freundlicherweise gleichzeitig die sich bei der Verwendung der Tools im Unterricht ergebenden Schwierigkeiten erwähnt. Jeder der Versuche ist anscheinend über die mit Recht von den Autoren kritisierte Komplexität dieser Software-Systeme gestolpert und folgerichtig vermutlich an ihr gescheitert. Der didaktische Wert des Einsatzes solcher Tools wird dabei allerdings seltener reflektiert.

Als neuesten Versuch - der wievielte ist es eigentlich? - erleben wir die Propagierung des nun wirklich ultimaten Werkzeugs ([L-S04], [Tho04]). Sein Name: BLUEJ (Verfallsdatum unbekannt, aber wahrscheinlich nicht sehr weit entfernt). Den Umgang mit Hilfsmitteln dieser Art konnte man schon seit längerem an LEGO-Mindstorms üben, Kinderprogramme mit Drag 'n' Drop Ikonen. Wer einmal versucht hat, mit Schülern bei der Programmierung der LEGO-Roboter den Umstieg von Mindstorms auf eine Programmiersprache, z. B. NQC, durchzuführen, weiß aus eigener Erfahrung über die resultierende Frustration. Alle derartigen Systeme können schließlich nichts anderes tun, als die real existierenden Schwierigkeiten zu cachieren - werden diese anschließend aufgedeckt, ist der Verdruß um so größer.

Und selbst wenn das neueste System nun das ultimate System wäre stellt sich die Frage, was ein Schüler damit lernt. Rasche Anfangserfolge verpuffen ebenso rasch und der Wissenszuwachs ist zweifelhaft. Was ist gewonnen, wenn man mit einem - falschen oder richtigen, selbstgemachten oder von "Profis" entwickelten - Werkzeug arbeitet? Werden dadurch Analyse und Modellierung einer konkreten Situation, z. B. einer Ampel, wirklich leichter? Ich bezweifle das. Vielmehr kann die - wie auch immer motivierte - Fixierung auf ein solches Werkzeug ein Hindernis bei der Analyse sein, da sie zu unnötigen oder sachfremden Vorbedingungen führt, die dem gesuchten Modell nicht inhärent sind.

Es ist wahrscheinlich unbestritten, daß es für die Modellierung eines Eisenbahnzugs und seiner Funktion in einem Rechner unerheblich ist, ob dieser als einfache Strichzeichnung entsteht, ob er "Tuuut" machen kann und Rauchfahnen von sich gibt; ebenso ist es bei einer Ampel belanglos, ob sie ihren Zustand in Kreisen darstellt, die unterschiedliche Farben aufweisen und diese wechseln können. Programmieren mit *Colours, Shapes and Sounds* (*Programming in CSS*) ist zwar ein immer wieder beliebter aber eben noch nie wesentlicher Aspekt der Funktionsmodellierung gewesen: ***La trahison des images***.

Wann ist eine Ampel eine Ampel

Die Konsequenzen für die objektorientierte Analyse sollten klar sein; ich möchte sie am Beispiel der Ampelanlage erläutern. Für die Funktion einer Ampel ist die Art der Präsentation ihres Zustands unerheblich, im einfachsten Fall kann diese als rein textuelle Bildschirmausgabe ("*Rot*" etc) erfolgen. Daraus ist eindeutig ersichtlich, daß die Darstellung dieses Zustands - Rechteck mit bunten Kreisen oder auch anders - kein wesentlicher Bestandteil der Funktion einer Ampelanlage ist.

Es ist folglich keineswegs notwendig oder auch nur sinnvoll, Schüler mit der Verwendung vielleicht halb, wahrscheinlich aber gar nicht verstehbarer Graphikbibliotheken und Entwicklungswerkzeuge zu quälen. Die Verwendung solcher Hilfsmittel führt nicht zu einer besseren Durchdringung der Problematik, vielmehr besteht die Gefahr, daß der Umgang mit den Werkzeugen in den Mittelpunkt des Geschehens rückt und die ursprüngliche Fragestellung, die Analyse und anschließende Modellierung eines Systems (Ampel, Eisenbahnzug, Taschenlampe etc) nur unscharf oder - wegen der Fixierung auf das eingesetzte Werkzeug - sogar falsch beantwortet wird.

Wesentlich in der Analyse einer Ampelanlage ist dagegen die Tatsache, daß eine einzelne Ampel (ein einzelnes Signal) praktisch sinnlos ist. Im Straßenverkehr, bei einer Schiffsschleuse, an einem Eisenbahngleis signalisiert ein Signal jeweils die Sperrung bzw. Freigabe einer Bewegungsrichtung während die andere freigegeben bzw. gesperrt ist, für diese andere Richtung muß notwendigerweise ein zweites Signal zur Anzeige des jeweiligen Zustands vorhanden sein. Außerdem können Signale und Ampeln sehr verschiedene Formen und - wichtiger - Funktionsweisen haben, man denke nur an die früher weit verbreiteten Zeigerampeln (Heuer-Ampeln).

Was also sind die wesentlichen Merkmale einer Ampel

- Von einer Ampel gibt es **mehrere**, korrelierte **Exemplare**
- Ampeln können in unterschiedlichen **Varianten** (Funktionsweisen und Formen) auftreten

An dieser Stelle erhebt sich die Frage, welche Datenstruktur den Merkmalen angemessen ist. Zu ihrer Beantwortung empfiehlt Mössenböck [Möss98] die Verwendung eines Entscheidungsbaums (s. Kasten). Er zeigt, unter welchen Voraussetzungen es sich lohnt, eine Klasse einzuführen, also objektorientiert zu arbeiten. Sind diese Voraussetzungen nicht gegeben, sollte man keine Klasse, sondern eine andere Datenstruktur wählen.

Nur wenn man tatsächlich beide in den oben stehenden Punkten angeführten Eigenschaften einer Ampel als relevant ansieht, ist es sinnvoll, diese Eigenschaften in einer Klasse zusammenzufassen. Unterstellt man dagegen lediglich eine einzelne Ampel (Taschenlampe etc), ist es besser, sich für eine Abstrakte Datenstruktur (ADS) zu entscheiden. Geht man - was allerdings in der Analyse nicht implizit erfolgen sollte - von der Existenz mehrerer gleicher Exemplare der betrachteten Datenstruktur aus, ist ein Abstrakter Datentyp (ADT) die vernünftige Wahl, nicht eine Klasse.

Ist man bei der Klasse als angemessener Datenform angekommen, steht man vor der Frage ihrer Modellierung. Nach einem bekannten Verfahren (Methode von Abbot [Abb83]) bieten sich für die Modellierung einer Klasse die Substantive und Verben einer informellen textlichen Beschreibung der Eigenschaften der zu modellierenden Objekte an

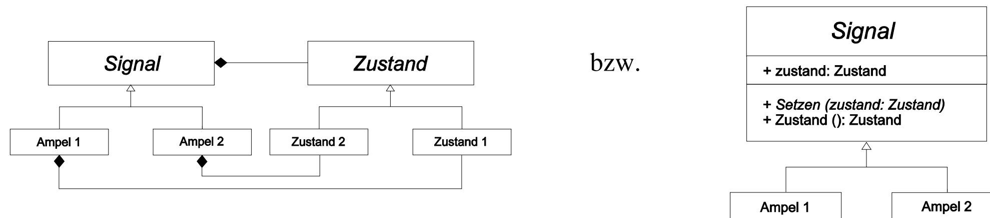
- Eine Ampel **ist** ein Signal(geber)
- Ein Signal **hat** einen **Zustand**
- Ein Signal **kann** seinen Zustand **ändern**
- Ein Signal **kann** seinen Zustand **bekanntgeben**

Eine **Ist**-Beziehung repräsentiert eine Vererbung. Da Signalanlagen in der Regel eine vernetzte Struktur bilden (Zentralrechner), korreliert sind und in sehr verschiedenen Formen mit unterschiedlicher Funktionalität existieren, ist es sinnvoll, die gemeinsamen Eigenschaften in einer universellen, abstrakten Basisklasse *Signal* zusammen zu fassen. Eine der möglichen Unterklassen ist dann die Klasse *Ampel*, die in mehreren gleichberechtigten Varianten auftritt oder zumindest auftreten kann. Beschränkt man sich in der ersten Entwurfsphase auf die häufig zu findende Lichtampel mit drei Signalleuchten, ist in der Implementierung ihrer Funktion immer noch zu beachten, daß der *Zustand* dieser Ampelvariante in verschiedenen Ländern verschieden sein kann, die Abfolge der Lichtsignale ist keineswegs überall die in der Bundesrepublik Deutschland gewohnte.

Eine **Hat**-Beziehung verweist auf eine eigenständige Datenstruktur, die von der betrachteten Struktur benutzt wird, es handelt sich um eine auch als *Aggregation* oder *Assoziation* bekannte Beziehung. Assoziationen treten in ähnlicher Form in sehr vielen Zusammenhängen auf, man denke beispielsweise an eine *Lineare Liste* mit den ihr assoziierten *Elementen* ([HvL04]).
Für den dem Signal assoziierten *Zustand* gilt, daß er veränderlich ist, das Signal wäre anderenfalls sinnlos. Der Zustand kann je nach konkret betrachtetem Signaltyp variieren. Somit kommen für seine Modellierung generell eine Abstrakte Datenstruktur (ADS), ein Abstrakter Datentyp (ADT) oder auch eine Klasse in Frage.

Verben der Beschreibung deuten auf eine Aktivität hin. Die durch sie ausgedrückten Eigenschaften sind Kandidaten für Prozeduren oder Funktionen (Methoden).

Die Ergebnisse der Analyse führen zu dem in den folgenden Diagrammen dargestellten Modell einer Signalanlage. *Signal* und *Zustand* sind abstrakte Klassen, *Ampel 1, Ampel 2* denkbare Konkretisierungen von *Signal* und *Zustand 1, Zustand 2* sind die assoziierten jeweiligen Konkretisierungen von *Zustand*



Die Methode *Setzen (zustand: Zustand)* muß abstrakt sein, denn die abstrakte Basisklasse *Signal* kann nicht wissen, welche Zustände ihre Unterklassen haben (die Klasse *Zustand* und ihre Unterklassen sind zur Vereinfachung im rechten Diagramm nicht dargestellt). *Setzen* muß in den Unterklassen überschrieben werden. Dagegen ist die Methode *Zustand(): Zustand*, die den aktuellen Zustand eines Signalobjekts zurückgibt, auf Grund der Polymorphie der Klasse *Signal* und ihrer Unterklassen konkret und kann bereits in der Basisklasse *Signal* ausprogrammiert werden.

Mit dieser Analyse und den Schlußfolgerungen könnte man zufrieden sein und die Implementierung beginnen. Es stellt sich allerdings die Frage, ob die Analyse wirklich vollständig ist. Bisher wurde der Aspekt der möglichen Zustände eines Signals nur pauschal behandelt, zu unpräzise für den Systementwurf. Untersucht man diesen Punkt im Detail, sieht man, daß der *Zustand* aller denkbaren Signale unabhängig von deren spezieller Ausformung eine endliche Abfolge periodisch sich wiederholender, strukturell identischer Einzelzustände sein muß. Jeder der Einzelzustände selbst hat zwei Einstellmöglichkeiten: *An* oder *Aus*. Mit diesen Feststellungen bietet sich für die Implementierung in kanonischer Weise eine Aneinanderreihung binärer Zustände an, wobei es sich wahlweise um eine Menge (Datentyp *SET*) oder eine Reihung Boole'scher Werte (Datentyp *ARRAY OF BOOLEAN*) handeln kann (hier und im weiteren sind Quelltexte und Schnittstellen in COMPONENT PASCAL Schreibweise wiedergegeben, UML Diagramme stellen demgegenüber keine Verbesserung dar, vielmehr gibt es für einige nachstehend verwendete wichtige Konstrukte keine UML-Äquivalente).

Bis auf die letzten Erläuterungen waren die bisherigen Darlegungen absichtlich programmiersprachenunabhängig gehalten, denn Analyse und Entwurf einer Datenstruktur sollen soweit irgend möglich allgemeingültig sein. Die sich anschließende Detailplanung der Implementierung hängt allerdings unvermeidlicherweise von den Möglichkeiten der jeweils verwendeten Programmiersprache ab. Mit Sprachen, die Reihungen als dynamische und damit

faktisch polymorphe Datenstrukturen zur Verfügung stellen, kann bei der Implementierung des Zustands der *Signal*-Klasse der Umweg über eine assoziierte abstrakte Klasse eingespart werden, der Typ *Zustand* lässt sich als quasi abstrakter Datentyp - beispielsweise als *Zustand = POINTER TO ARRAY OF BOOLEAN* - realisieren.
Mit diesem Datentyp für *Zustand* bekommt die Klasse *Signal* folgende Schnittstelle

```
DEFINITION Signal;

    TYPE
        Signal = POINTER TO ABSTRACT RECORD
            (signal: Signal) Ermitteln- (zustand: Zustand), NEW, ABSTRACT;
            (signal: Signal) Setzen (zustand: Zustand), NEW;
            (signal: Signal) Zustand (): Zustand, NEW
        END;

        Zustand = POINTER TO ARRAY OF BOOLEAN;

END Signal.
```

Außer der bereits angesprochenen Deklaration des Typs *Zustand* als dynamisches Array anstelle einer Klasse findet sich eine weitere Änderung gegenüber der oben stehenden UML Version. Anders als dort angegeben ist die Methode *Setzen* konkret, sie ist im Modul *Signal* ausprogrammiert, dies anscheinend im Widerspruch zu der dargestellten Tatsache, daß die Klasse *Signal* den konkret zu setzenden Zustand nicht kennen kann. Die in der Schnittstelle gezeigte Implementierung ist deshalb möglich, weil die Methode *Setzen* ihrerseits die abstrakte Methode *Ermitteln* verwendet, mit der der konkrete Zustand des jeweiligen, die Methode *Setzen* aktivierenden Objekts ermittelt wird. Das sieht im ersten Moment wie eine unnötige Komplikation aus. Statt *Setzen* abstrakt zu lassen und der Unterklasse die Verantwortung für die Implementierung zu übergeben, gibt es eine zusätzliche Methode (und einen zusätzlichen Aufruf), die ihrerseits in der Unterklasse implementiert werden muß.

Der Sinn des Verfahrens erschließt sich erst, wenn man berücksichtigt, daß Programme, die in prinzipiell offenen Systemumgebungen ablaufen, anders als closed shop Programme eine große Flexibilität realisieren und gleichzeitig hohe Anforderungen an Sicherheit und Robustheit erfüllen müssen. Flexibilität erfordert die Veröffentlichung vieler potentiell sensibler Details und steht dadurch unvermeidlich im Gegensatz zu Sicherheit und Robustheit. Diese lassen sich am ehesten gewährleisten, wenn den unbekannten Klienten eines Basismoduls keine oder so wenig wie möglich Gelegenheiten gelassen werden, auf das Basismodul zurückzuwirken.

Für die Verwirklichung so widerstreitender Ziele wie Flexibilität und Sicherheit gibt es in COMPONENT PASCAL ein in nur wenigen Programmiersprachen zu findendes Konstrukt: Implement Only Export von Methoden, erkenntlich an der Minus-Marke hinter dem Methodennamen. Damit ist eine exportierte abstrakte Methode gemeint, die von einem Klienten implementiert werden muß, aber nicht von ihm aktiviert werden kann; die Aktivierung ist nur innerhalb des exportierenden Moduls möglich (*upcall* oder auch *callback*). Auf diese Weise hat der Klient einerseits die Möglichkeit, die Methode exakt auf seine (nur ihm bekannten) Bedürfnisse zuzuschneiden, andererseits hat der Implementierer des Basismoduls die vollständige Kontrolle über den Einsatz der Methode, er kann durch entsprechende Maßnahmen verhindern, daß potentiell gefährlicher Code ausgeführt wird. Insgesamt gewährleistet das beschriebene Verfahren also die gewünschte Flexibilität bei gleichzeitiger wesentlicher Erhöhung der Programmstabilität.

In der nachstehend zu findenden erweiterten Schnittstelle des Klientenmoduls *Ampel* (zur Vereinfachung ist in der Grundversion nur eine der möglichen Konkretisierungen - die deutsche "Standardampel" - implementiert) enthält die Klasse *Ampel* die Methode *Ermitteln* zweimal; einmal als eine der drei von der Basisklasse *Signal* geerbten Methoden in der abstrakten und ein zweites Mal in konkreter Form als Methode der Unterklasse *Ampel*, als Zeichen dafür, daß sie in der Unterklasse implementiert worden ist

```
DEFINITION Ampel;

   IMPORT Signal;

   CONST
      rot = 0; gelb = 1; grün = 2;

   TYPE
      Ampel = POINTER TO LIMITED RECORD (Signal.Signal)
         (signal: Signal.Signal) Ermitteln- (zustand: Signal.Zustand), NEW, ABSTRACT;
         (signal: Signal.Signal) Setzen (zustand: Signal.Zustand), NEW;
         (signal: Signal.Signal) Zustand (): Signal.Zustand, NEW;
         (ampel: Ampel) Ermitteln- (zustand: Signal.Zustand)
      END;

      Zustand = Signal.Zustand;

   PROCEDURE NeueAmpel (): Ampel;

END Ampel.
```

Die übrigen Teile der Schnittstelle sind praktisch selbsterklärend; die drei Konstanten sind Aliasbezeichner für die Einzelfelder der Reihung *Zustand*, die den unveränderten Typ des Basismoduls hat. Dieser muß im Klienten allerdings den Gegebenheiten der Unterklasse angepaßt werden, das geschieht in dem Konstruktor *NeueAmpel*

```
PROCEDURE NeueAmpel* (): Ampel;
   VAR
      ampel: Ampel;
      zustand: Zustand;
BEGIN
   NEW(ampel); NEW(zustand, 3);
   zustand[rot] := TRUE; zustand[gelb] := FALSE; zustand[grün] := FALSE;
   ampel.Setzen(zustand);
   zustand := ampel.Zustand();
   ASSERT(zustand[rot] & ~zustand[gelb] & ~zustand[grün], 99);
   RETURN ampel;
END NeueAmpel;
```

Es werden eine neue *ampel* und ein neuer *zustand* erzeugt, wobei dieser mit dem zweiten Parameter in der Anweisung *NEW(zustand, 3)* auf die für den konkreten Ampeltyp nötige Länge beschränkt wird. Die übrigen Zeilen belegen die *ampel* mit einem festen Anfangszustand, überprüfen ihn, nachdem er gesetzt worden ist und geben die *ampel* dann an die anfordernde Stelle zurück.

Die eigentliche Realisierung einer konkreten Ampelanlage erfolgt im Modul *Ampeltest*. Es werden zwei Ampelobjekte instantiiert, wobei eines von ihnen als Fußgängerampel, also ohne Gelblicht, realisiert ist (dies erfordert und rechtfertigt keine separate Klasse, da lediglich die Ampelschaltung entsprechend angepaßt werden muß). Die Schnittstelle und die “graphische” Benutzeroberfläche des Anwendermoduls sehen folgendermaßen aus

```
DEFINITION AmpelTest;

   VAR
      FußgängerAmpel: RECORD
         rot-, grün-: BOOLEAN
      END;
      KfzAmpel: RECORD
         rot-, gelb-, grün-: BOOLEAN
      END;

   PROCEDURE NeueAmpelanlage;
   PROCEDURE Schalten;

END AmpelTest.
```

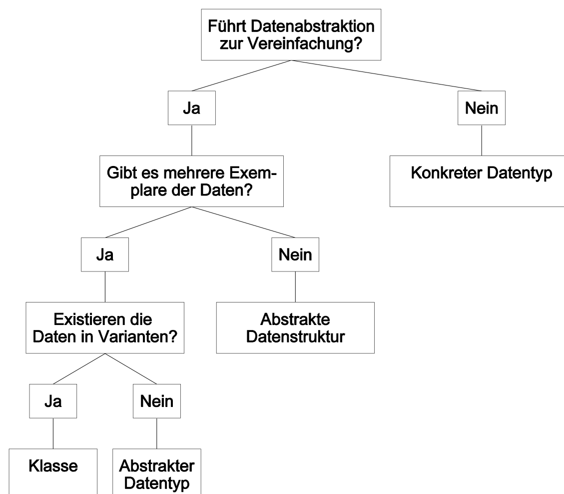
Die Dialogbox enthält eine sinnfällige Realisierung der beiden Ampeln der Anlage. Für die Präsentation der jeweiligen Zustände sind Kreise und Farben unnötig, es genügen Kontrollfelder (check boxes), die die Boole'schen Ampelwerte unmittelbar anzeigen; die Dialogbox selbst ist mit zwei Mausklicken erzeugt, Schüler können sich auf die Gestaltung des Layouts beschränken.

Erweiterungen der hier vorgestellten Basisampelanlage sind leicht vorstellbar und werden höchstwahrscheinlich von Schülern selbst vorgeschlagen, beispielsweise: Ausbau der Ampel zu einer vollen Straßenkreuzung; Hinzunahme anderer Ampeltypen für Bus oder Tram; überlappende Intervalle für Sperrung und Freigabe der Bewegungsrichtungen; Erweiterung um eine Zeitschaltautomatik etc.

Quintessenz

- Im Unterricht sind Werkzeuge keine sinnvollen Hilfen bei der Analyse und Modellierung eines Systems.
- Graphik ist kein wesentlicher Bestandteil der Analyse und Modellierung eines Systems, es sei denn, dies ist ein Graphik-System.
- Dem Prozess der objektorientierten Modellierung geht der Prozess der objektorientierten Analyse voraus.
- Die Analyse sollte ergebnisoffen sein, stellt sich in ihrem Verlauf heraus, daß keine Notwendigkeit für die Bildung von Klassen besteht, sollte man eine andere, einfachere Struktur für die zu modellierenden Daten wählen.
- Der objektorientierte Ansatz ist nur dann sinnvoll, wenn die beiden wesentlichen mit ihm verbundenen Eigenschaften - Vererbung und Polymorphie - tatsächlich benötigt werden.

Wann ist eine "Klasse" eine Klasse?



Nicht jede Datenstruktur, die den Typbezeichner *Klasse* trägt, wird dadurch automatisch eine Klasse. Der nebenstehende Entscheidungsbaum [Möss98] zeigt auf, wann es sich bei einer heterogenen Datenstruktur - unabhängig von ihrem deklarierten Typ - tatsächlich um eine Klasse handelt.
Formal besteht zwischen einem ADT (Verbund, Datentyp *RECORD*) und einer Klasse kein Unterschied, beide können sowohl Datenfelder (Attribute) als auch Prozeduren (Methoden) enthalten. Der Unterschied liegt vielmehr darin, daß eine Klasse erweiterbar ist, es lassen sich Unterklassen bilden, die zusätzliche Attribute und Methoden enthalten können. Im Gegensatz dazu gibt es bei einem ADT keine Vererbung, er ist nicht erweiterbar.
Man kann, wenn man will, einen ADT als eingeschränkte Klasse ansehen, als Klasse, die final ist. Allerdings besteht ein wesentlicher Unterschied zwischen einer Unterklasse und einem ADT. Eine Basisklasse [Beispiel: *Signal* = ***POINTER TO ABSTRACT RECORD***] "kennt" ihre Unterklasse [Beispiel: *Ampel* = ***POINTER TO LIMITED RECORD*** *(Signal)*] polymorphisch, ein Objekt der Unterklasse kann an jeder Stelle verwendet werden, an der ein Objekt der Basisklasse erwartet wird; dies ist mit Variablen eines von der Basisklasse unabhängigen ADT [Beispiel: *ADTAmpel* = ***POINTER TO RECORD***] nicht möglich, auch wenn die beiden Datentypen in allen Einzelheiten übereinstimmen.
Unabhängig davon, ob in einer Programmiersprache für heterogene Datenstrukturen nur Klassen zur Verfügung stehen oder ob es unterschiedliche Möglichkeiten für ihre Realisierung gibt, sollte sich ein Programmierer in jedem Einzelfall Klarheit darüber verschaffen, ob das angestrebte Ziel tatsächlich eine Klasse erfordert oder nur einen ADT oder noch etwas anderes (s. Entscheidungsbaum). Der Einsatz des speziellen Datentyps Klasse sollte auf die Fälle beschränkt werden, die ihn erfordern. Meint man, "grundsätzlich" oder auch nur "sicher-

heitshalber" eine Klasse verwenden zu sollen, weil man diese vielleicht bei einer späteren Änderung des Programmentwurfs benötigen könnte, ist die dem Entwurf zu Grunde liegende Analyse höchstwahrscheinlich nicht vollständig und sollte präzisiert werden.
Die Verwendung einer Klasse ist nur gerechtfertigt, wenn die von dieser Klasse abgeleiteten Objekte in Varianten existieren, die von einer gemeinsamen Oberklasse verwaltet werden, d. h. differierenden Unterklassen dieser Basisklasse entstammen, wenn also die wichtigste Eigenschaft des Vererbungskonzepts - die dynamische Polymorphie - tatsächlich benötigt wird. (Für eine detaillierte Darstellung s. [HvL04])

Typische Fehler der OOP

Erzwungene Objektorientierung - Alles und jedes wird einer Klassenbildung unterworfen.
Dies ist in vielen Fällen unnötig und führt zu aufgeblähten Programmstrukturen, die keineswegs besser zu verstehen sind als klassische Programme. Entweder sind die betrachteten Daten nicht heterogen oder ihnen fehlt ein gemeinsamer Aspekt für die Zusammenfassung in einer Klasse. Darüber hinaus erfordert Objektorientierung während der Analysephase die Untersuchung auf Verallgemeinerbarkeit. Nur wenn eine polymorphe Datenstruktur, d. h. die dynamische Verwendung verschiedener Varianten einer zu Grunde liegenden gemeinsamen Abstraktion erkennbar ist, besteht ein Anlaß objektorientiert zu arbeiten.
Mathematische Bibliotheken sind ein typisches Gegenbeispiel, sie stellen praktisch ausschließlich Funktionen zur Verfügung, jeder Versuch einer weiter gehenden Abstraktion und Klassenbildung zur "Methodisierung" dieser Funktionen sähe gekünstelt aus.

Falsche Klassenbildung - Die Basisklasse wird zu eng, d. h. zu konkret, evtl. sogar falsch gefasst.
Beispiel ***Ökonomie***: Niemand hat jemals eine Ware gesehen, allerdings zweifellos erstaunlich (oder eher erschreckend?) viele Objekte mit Warencharakter.
Gerade die Heterogenität der konkreten Warenformen hat dazu geführt, das ihnen allen Gemeinsame als das Wesentliche zu definieren und dieses Gemeinsame - unter dem Handelsaspekt einzig Wichtige - mit dem abstrakten Begriff *Ware* zu kennzeichnen.
Beispiel ***Computergraphik***: Die Basisklasse aller graphischen Objekte ist nicht etwa eine Klasse *Punkt*, diese ist bereits unnötig konkret und eingeschränkt, vielmehr ist es die vollständig abstrakte Klasse *Figur*. Nur diese ist so allgemein, daß wirklich jede noch so merkwürdige konkrete Figur von ihr abgeleitet und polymorphisch durch sie verwaltet werden kann.
Darüber hinaus steckt in der Annahme, *Punkt* sei als Basisklasse geeignet, ein logischer Fehler. Vererbung stellt eine *Ist (Is a)* - Beziehung dar; eine von *Punkt* erbende Klasse *Linie* ist aber kein Punkt sondern besteht aus einer Menge von Punkten, es handelt sich also um eine *Hat (Has a)* - Beziehung, eine Assoziation.
Beispiel ***Mathematik***: Die Basisklasse aller Zahlen ist nicht etwa eine Klasse *NatürlicheZahl*, vielmehr die vollständig abstrakte Klasse *Zahl*, nur diese ist so allgemein, daß wirklich jede noch so merkwürdige konkrete Zahl von ihr abgeleitet werden kann. Aus guten Gründen werden allerdings Standardzahltypen nur in wenigen Programmiersprachen (z. B. Smalltalk) als Klassen implementiert. Elementare statische Zahltypen sind einfacher zu handhaben und zudem effizienter.

Und schließlich gibt es, als absurdeste Konsequenz eines universellen Zwangs zur Klassenbildung, ein im wörtlichen Sinn fabelhaftes Phänomen. Programmiersprachen, die nur Klassen kennen, erfordern für die Erzeugung eines Objekts ein an Münchhausen-Tricks erinnerndes Verfahren. Die Objekterzeugung muß - und kann nur (sieht man von Objektfabriken einmal ab) - durch eine Klassen-Methode erfolgen. In derartigen Sprachen muß also das - noch nicht existierende - Objekt den Auftrag erhalten, sich selbst zu erschaffen; das ist wahrhaft an den eigenen Haaren aus dem Sumpf gezogen!

Literatur

[Abb83]	Abbott, R.:	Program Design by Informal English Descriptions. Communications of the ACM, vol. 26 (11), 1983, S. 882
[BaBa04]	Balzert, H., Balzert, H.:	Modellieren oder Programmieren oder beides? Plädoyer für einen schrittweisenAufbau mentaler Modelle im Unterricht. LOG IN 128/129 (2004), S. 20
[For04]	Forbrig, P.:	Objektorientierung. Stand und aktuelle Entwicklungen. LOG IN 128/129 (2004), S. 12
[Heu04]	Heubaum, A.:	Möglichkeiten und Grenzen maschineller Intelligenz. Unterrichtsvorschläge in JAVA. Teil 1, Suchbaum und Rückziehungsverfahren. LOG IN 128/129 (2004), S. 62
[L-S04]	Leipholz-Schumacher, B.:	Objektorientiertes Modellieren und Programmieren. Ein Unterrichtskonzept mit JAVA und BLUEJ in der Sekundarstufe II. LOG IN 128/129 (2004), S. 32
[HvL04]	von Lavergne, H.:	Objekte, Klassen, Module, Kontrakte und Komponenten. LOG IN 131/132 (2004), S. 46 (s. a. http://www.lahini.de/daten/publik/kompo.pdf)
[Möss98]	Mössenböck, H.:	Objektorientierte Programmierung in Oberon-2. Springer-Verlag, Heidelberg; 3. Auflage, 1998
[Prä04]	Prätorius, P.:	Grafik im Anfangsunterricht. Programmbeispiele in JAVA. LOG IN 127 (2004), S. 35
[Spol04]	Spolwig, S.:	Kritisches zu Stiften und Mäusen. Was ist objektorientierte Modellierung? LOG IN 130 (2004), S. 35
[Stein04]	Steinbrucker, Ch.:	Objektorientierung im Anfangsunterricht!!! Simulation einer Taschenlampe mit DELPHI. LOG IN 131/132 (2004), S. 56
[Tho04]	Thomas, M.:	Objektorientierung und informatische Bildung: Stellenwert und Konkretisierung im Unterricht mit BLUEJ. LOG IN 128/129 (2004), S. 26
[SuMa]		Stifte und Mäuse http://www.learn-line.nrw.de/angebote/oop/medio/sum/kern/uml/ klassenbibliothek.html

Erschienen in:
LOG IN 136/137, 2005