Modellieren und Programmieren

"Geld wechseln" als Algorithmikübung oder Programmieraufgabe - wann lohnt es sich ein Programm zu machen?

von
Siegfried Spolwig

Einleitung

> *Sagt der Prozedurale zum Objektorientierten:*
>
> *"Bei einem derart simplen Programm ist der Einsatz objektorientierter Programmierung für die Durchführung des Wechselvorgangs zweifellos absurd.*
> *Kleine Programme ... lassen sich immer noch einfacher ... in konventioneller Weise umsetzen." (Lavergne)*
>
> *Erwidert der Objektorientierte dem Prozeduralen:*
>
> *"Warum soll man für Geldwechseln überhaupt ein Programm machen? Das ist eine kleine Algorithmikübung, die besser nur mit Papier und Bleistift gelöst wird.*
> *Das Programm ist der Geldwechselautomat." (Spolwig).*

Während einem Prozeduralen bei dieser Fragestellung nach kurzem Nachdenken ein pfiffiger Algorithmus einfällt, der berechnet wie viele Stücke der Münzsorten gebraucht werden, sieht der Objektorientierte sofort eine Handvoll Münzen, Geldscheine, einen Kasten mit einer Geldausgabeklappe usw. Wechseln ist dabei eine Methode unter anderen, die irgend ein Objekt dann später anbietet.

Damit sind zwei grundsätzlich verschiedene Herangehens- und Sichtweisen im Informatikunterricht beschrieben - scheinbar ein endloses Thema, das jeder Lehrer wohl für sich entscheiden muss. Eines scheint sicher, mit Programmieraufgaben, die ein herausgelöstes Algorithmikproblem behandeln und vielleicht noch ein aufgepepptes Layout mit einem GUI-Builder darum herumstricken, wird den Schülern ein völlig falsches Bild von Informatik und der Komplexität ihrer Aufgabenstellungen vermittelt.

In den neueren Ansätzen zur Fachdidaktik lässt sich ablesen, dass Modellbildung und Modellierungstechniken als ein Kern der Schulinformatik gesehen werden, während die Rolle spezieller Programmiersprachen in den Hintergrund rückt. Eine erfreuliche Entwicklung, die die Sichtweise, wie wir sie auch an dieser Stelle seit langem vertreten und im Unterricht praktiziert haben, bestätigt.

Zu den oft beschworenen Kategorien der "neuen Informatikdidaktik" gehören Komplexität, Abstraktion, Modellbildung, Modularisierung, Klassenbildung und mehr. Leicht gesagt, aber scheinbar schwer umzusetzen. Das zeigt sich schon daran, dass relativ wenige Beispiele veröffentlicht sind. Im Anschluss an die Beiträge von LAVERGNE und SPOLWIG in LOG IN 19 (1999) Heft 5 sollen ein Thema und einige Fragestellungen, die sich aus einem OOM-zentrierten Unterricht ergeben, vorgestellt werden.

Es geht mir keineswegs um einen neuen Sprachenstreit "Meine ist besser als Deine", sondern darum, bei den Kollegen vielleicht die Neugier auf OOM zu wecken, weil diese Denkweise, unabhängig von der verwendeten Sprache, an vielen Stellen besser geeignet ist, Informatikprobleme und deren Lösung den Schülern verständnisbildend näher zu bringen. Sie eröffnet die Möglichkeit, Programmsysteme im Unterricht zu erstellen, die früher außerhalb der Erreichbarkeit lagen mangels effizienter Analysetechniken und wegen vergleichsweise primitiver Werkzeuge, die den Schulen zur Verfügung standen.

Das Schulfach Informatik hat hier neben dem allgemeinbildenden Beitrag, um den wir ja alle ringen, noch die nahezu einmalige Chance, den Grundstein für eine handfeste Berufsorientierung zu legen, wenn wir den Schülern ein realitätsnahes Bild von den Aufgaben der Informatik und ihren zeitgemäßen Verfahren zur Lösung geben. Nutzen wir also diese Möglichkeit richtig! Die Inder kommen sowieso nicht.

Damit ist aber nicht gesagt, dass im Unterricht dann keine Probleme mehr auftreten - man bekommt dafür andere.

Wann lohnt es sich, ein OO-Programm zu machen?

Wenn das Stundenthema vielleicht hieße "Heute programmieren wir die vorprüfende Schleife!", dann sicher nicht. Aber dafür lohnt auch kein noch so kleines ausführbares Hauptprogramm, weil man damit die Vorstellung vermittelt, so würden irgendwie Programme entstehen.

> *Wenn es nicht gelingt, aus einer Problemstellung ein Fachkonzept mit einigen Klassen zu finden, liegt offensichtlich keine Aufgabe vor, die es rechtfertigt, ein Programm zu machen. Man kann davon ausgehen, dass bei einer OO-Umsetzung immer einige Klassen, die das Anwendungsproblem beschreiben (GUI-Klassen zählen hier nicht), beteiligt sind. Nach einigen Jahren Unterrichtserfahrung mit OOM und OOP wissen wir, dass es ausgesprochen schwierig ist, vernünftige abgeschlossene Anwendungsbeispiele zu finden, die klein genug (!) für den Unterricht sind.*

Einer der ersten Versuche bestand darin, im Anfangsunterricht eine Rechnung als eine Klasse darzustellen, in der Artikelpositionen, Preise usw. als Attribute auftraten und Methoden, die Zwischensumme, Umsatzsteuer und Bruttobetrag errechneten. Es war alsbald klar, dass so etwas eine völlige Fehlentscheidung ist, weil eine gründliche Analyse im Handumdrehen bei genauer Betrachtung ein Warenwirtschaftssystem zu Tage fördert mit einer Vielzahl von Klassen, je nachdem wie tief man hinunter geht, und dass die Rechnung nur ein View ist, wie bei einer Datenbankmodellierung auch. Man steht immer vor der Frage, wie weit darf ich das Datenmodell reduzieren, ohne dass es völlig hirnrissig und weltfremd wird.

Sicher wird man im Unterricht immer Phasen haben, bei denen algorithmische Probleme zu behandeln sind. Ein separates Programm "BubbleSort" ist sicher in meinem Sinne die schlechteste Lösung. Sortieren und Suchen bekommen dann einen annehmbaren Rahmen, wenn sie in ein adäquates Anwendungsprogramm eingebettet sind. In OOP wäre ein angemessener Ort eine Listenklasse mit der Methode *Liste.Sortieren*. Und Sortieren wird nur dann als ernsthafte Aufgabe eines Prozessors erfahren, wenn man nicht 20 Elemente sortiert, sondern 100.000. Vermutlich hat jeder Lehrer irgendwann mal eine sequentielle Dateiverwaltung gemacht. Es ist eine lehrreiche Übung, daraus ein OO-Programm mit einer Listenverwaltung zu machen, und darin Algorithmik zu betreiben, ist der bessere Weg. Wenn man im Laufe

einer Kursfolge eine handvoll Programme und Klassen anlegt, die aufgrund der OO-Architektur ganz oder in Teilen wieder verwendbar und modifizierbar sind, ist man gut gerüstet. Die unten beschriebene Listenklasse wird in einer Vielzahl von Programmen, in denen Datenkollektionen auftauchen, von Schülern eingesetzt. Auch dadurch kann man im Unterricht demonstrieren, was wichtige Ziele der Informatik sind. Die Methode, den Schülern respektable und weitgehend fertige OO-Programme zu geben und sie dort Erweiterungen, Änderungen einbauen zu lassen, hat sich bewährt.

Modellieren statt Programmieren - Auswirkungen auf den Unterricht

Bei den letzten halbjährigen Softwareprojekten ergab sich, dass die OOM-Phasen (OOA, OOD einschließlich der vollständigen Spezifikation) bis zu 3/4 der verfügbaren Zeit in Anspruch nahmen, so dass in der Kodierungsphase nur noch ein Prototyp mit einem GUI-Builder erstellt werden konnte, um die Benutzerschnittstelle zu realisieren. Praktisch bedeutete es, dass die Schüler ca. 12-15 Wochen lang kaum eine Zeile kodiert haben und es ist darüber kein Unmut ausgebrochen, wie man vielleicht befürchten würde. Der PC wurde in dieser Zeit im Wesentlichen zur Dokumentation, für Protokolle und Referate benutzt. Der Verzicht auf die Programmierphase fiel den Schüler nicht schwer, denn es war allen klar geworden, dass die eigentliche Aufgabe, ein Fachkonzept zu erarbeiten, gelöst war und was jetzt kommen würde, nur noch eine Fleißarbeit ist.

Es ist eindeutig zu beobachten, dass die Verlagerung des Unterrichtsschwerpunktes auf OOM mit einem dramatischen Verlust an Programmierkompetenz (Kodierung und Kenntnisse der Programmiersprache) einhergeht, während die Analyse- und Modellierungskompetenzen deutlich besser ausgeprägt sind. Eigentlich kaum verwunderlich - wer auf den systematischen Pascalkurs verzichtet, kann nur punktuelle Kenntnisse erwarten. Die Situation wird möglicherweise dadurch verschärft, dass wir von der ersten Stunde an objektorientiert arbeiten und die Programmiersprache + Entwicklungssystem quasi nebenher gelernt werden. Insgesamt eine erhebliche Anforderung.
Das ist eine Situation, der man sich stellen muss und entscheiden, ob man sie akzeptieren will, die aber gleichzeitig unbefriedigend ist, weil letztlich nur mit hinreichenden Programmierkenntnissen überprüft werden kann, ob das Modell vernünftig konzipiert war. Ich bin skeptisch, ob Versuche, die Programmiersprache aus dieser Funktion herauszunehmen und durch andere Verifizierungsinstrumente zu ersetzten, gelingen werden. Neben dem Verifizierungsaspekt erzieht die Notwendigkeit des fehlerfreien Programmierens zu einer Reihe von Verhaltensweisen und "Tugenden", die sich in der Schule kaum besser mit anderen Mitteln erreichen lassen.

Meine Schule übernimmt regelmäßig Schüler mit teilweise guten Kenntnissen, die in anderen Schulen strukturiertes Programmieren in "Pascal-Kursen" kennen gelernt haben. Diese Schüler haben gegenüber Anfängern erhebliche Vorteile beim Implementieren von Klassen usw. Das scheint darauf hinzudeuten, dass es günstig sei, Programmieren quasi "genetisch" zu lernen, also erst Sprache und strukturiert und danach OOP als Weiterentwicklung. Leider funktioniert das nicht so gut. Diese Schüler haben signifikant große Schwierigkeiten auf das objektorientierte Denken umzusteigen. Offensichtlich ist die "Prägekraft des ersten Eindrucks" auch hier von bleibender Stärke. Das ist alles natürlich nicht wissenschaftlich fundiert, aber übereinstimmende Erfahrung mehrerer Kollegen über Jahre.

Ein Modellierungsbeispiel

Der Modellbegriff

Im Gegensatz zum mathematischen wird hier ein informatischer Modellbegriff verwendet, der einen Ausschnitt aus der Realität beschreibt. Letztlich ist es die Modellierung eines informatischen Modells einer willkürlich definierten Anwendung mit den Mitteln der Abstraktion (Weglassen von Eigenschaften, die unter dem Aspekt der Nützlichkeit für das Modell unwichtig sind und Herausarbeiten der wirklich notwendigen Verhaltensweisen), der Hierarchisierung und Modularisierung. Man darf dabei nie vergessen, dass bei aller Logik und Stringenz der Vorgehensweise hier *Entwurfentscheidungen* getroffen werden, auf die Kategorien wie falsch oder richtig nur bedingt zutreffen. Aus der Sicht der angestrebten Nutzung kann es immer durchaus verschiedene Lösungen geben. Eine Sichtweise übrigens, die vielen Informatiklehrern, die von Hause aus Mathematiker sind, Unbehagen verursacht, wie wir häufig in Lehrerfortbildungen gehört haben.

Der Geldwechselautomat - Aufgabenstellung und Modellannahmen

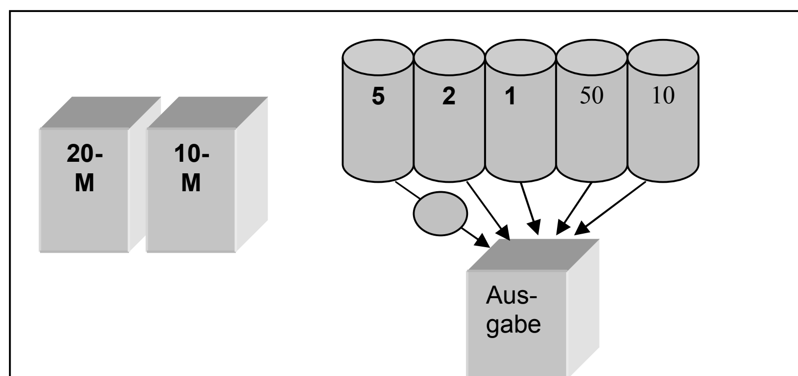
Die Idee ist nun wirklich nicht neu, aber eine Umsetzung als objektorientiertes Softwaresystem stellt durchaus eine neue Herausforderung dar, die weit über das hinausgeht, was man vielleicht früher daran probiert hat.

Es ist eine Simulation eines weitgehend realen Automaten zu bauen, der Banknoten in Münzen wechselt. 10-M, 20-M Noten werden angenommen, Falschgeld wird erkannt und abgewiesen. Als Münzen werden 5-M, 2-M, 1-M, 50-Pf und 10-Pf Stücke ausgegeben. Noten und Münzen werden als grafisch animierte Objekte auf dem Bildschirm dargestellt. Die Geldannahme wird durch einen Tastendruck angestoßen, Auswahl und Münzentnahme durch Mausklick.

Der Automat enthält

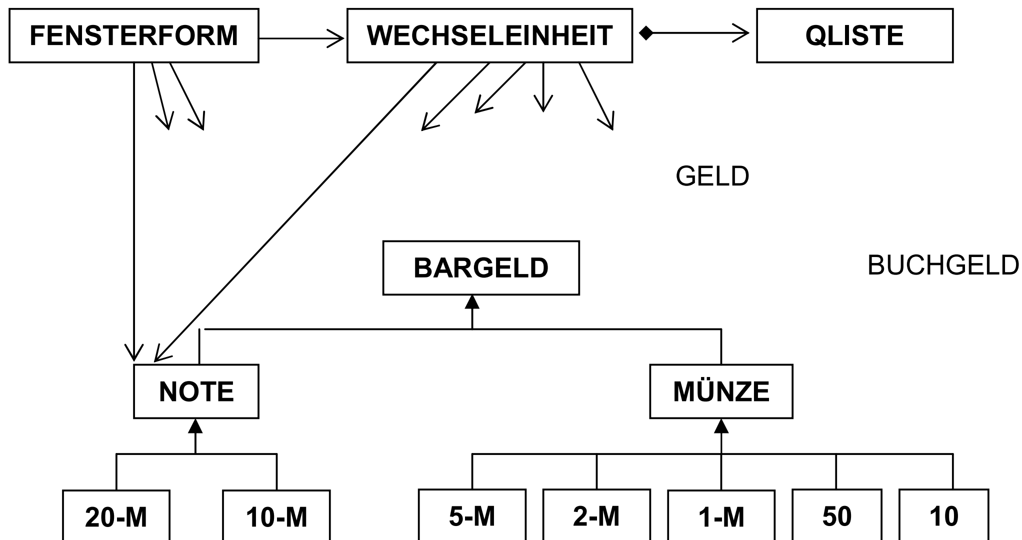
- 2 Annahmeschächte für 10-M und 20-M, die jeweils Exemplare (Instanzen) der Bank noten aufnehmen
- 5 Ausgabeschächte für die Münzsorten, die bei Start aufgefüllt werden mit diskreten Münzen, also nicht die Anzahl verwalten (25 St.), sondern 25 einzeln erzeugte Exemplare einer Münzsorte aufnehmen
- Bei Münzausgabe werden die benötigten Münzen in den Ausgabeschacht (Geldausgabe klappe) getan und können dann dort entnommen werden
- Benutzerhinweise und Störungsmeldungen sollen in einem Display angezeigt werden. Wenn ein Schacht leer ist, wird die Geldannahme mit einer Störungsmeldung abgebrochen.

Modellskizze Geldschächte

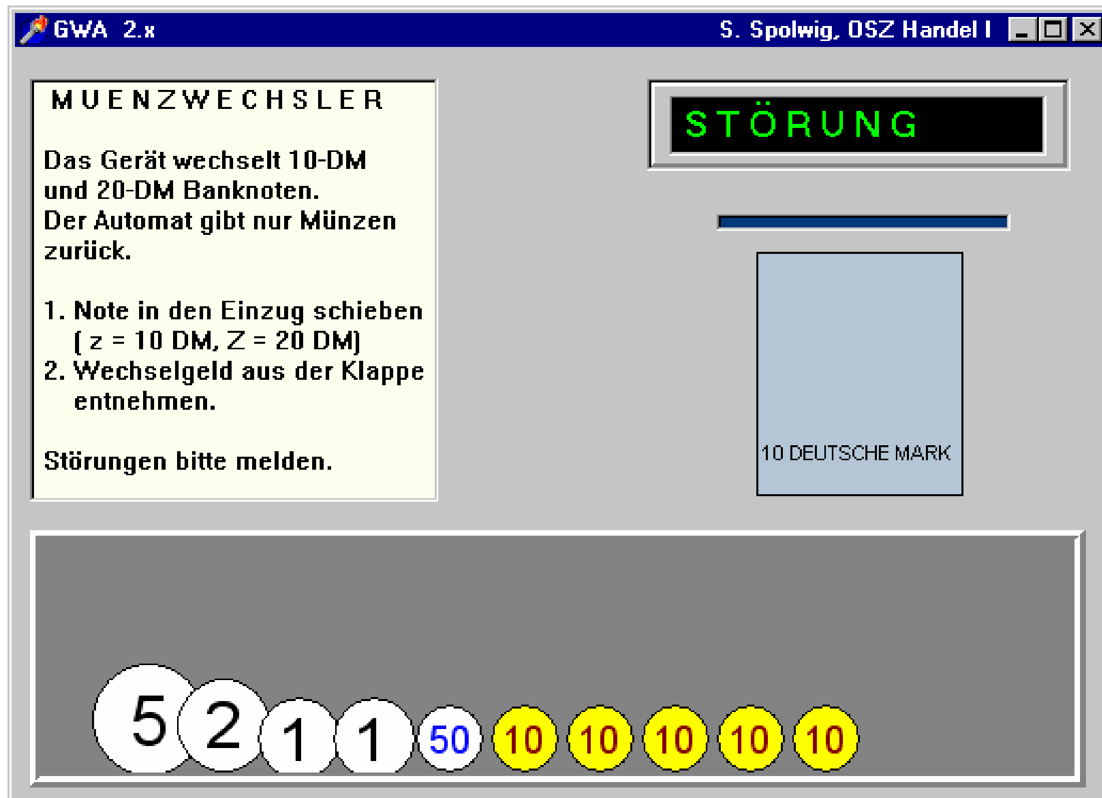


Informatische Inhalte

- **OOA** des Systems
- **OOM**
- **Klassenbildung**
- **Konkrete Klassen** für 20-M, 10-M, 5-M usw.
- **Abstrakten Klassen** : Münzen, Noten; üblicherweise nur zum besseren Strukturieren von Klassenhierarchien verwendet, hier auch eingesetzt, um Falschgeld, dass keine der eingeführten Klassen angehört, identifizieren zu können
- **Klassenattribute**: Wertattribut aller Münzen derselben Klasse hat denselben Wert
- **Vererbungsbeziehungen** mit **polymorphen** Methoden beim Fachkonzept
- **Aggregationen (**Komponenten**)** und **Assoziationen**
- **Polymorphie** bei Geld, Schächten, Listen
- Polymorphe **Liste (Queue)** zur Realisierung der Geldschächte (Die Münzen werden bei Programmstart von oben aufgefüllt und fallen unten raus in den Ausgabeschacht.)
- **Grafikprogrammierung** zur Darstellung und Animation von Noten und Münzen



Statisches Modell der Fachklassen und FensterFormular in UML-Notation. Auf das Eintragen von Attributen und Methoden wurde aus Platzgründen verzichtet. Geld und Buchgeld sind nicht im Modell.



Benutzungsoberfläche mit GUI-Builder erstellt.

Obwohl nach der objektorientierten Modellierung theoretisch noch offen ist, mit welcher Sprache (und Programmierstil) die Implementierung erfolgen soll, scheint mir - besonders in der Schule - die Umsetzung in einer OO-Sprache die natürlichste Wahl zu sein und zwar mit einer, die den Entwurf möglichst 1:1 übersetzt. Die nachfolgenden Quellcodes sind in Object-Pascal (DELPHI) geschrieben und zeigen ausschnittsweise die Realisierung der Klassen.

TBargeld ist eine abstrakte Klasse mit überwiegend leeren Methoden, die in *TMuenze* überschrieben werden. Die abgeleiteten Klassen *TMark* usw. haben dann nur zwei spezielle Methoden, der Rest wird geerbt.
In *TWechselEinheit* kann man die Realisierung der Münzschächte sehen, die auf *TQListe* basieren. *TwechselEinheit.SchaechteFuellen* und *TwechselEinheit.ZwanzigMarkWechseln* zeigt die polymorphen Methoden der Liste.

Quellcodes

```
UNIT uBargeld;
(* ****************************************************************** *)
(* K L A S S E : TBargeld                                             *)
(* ------------------------------------------------------------------ *)
(* Version     : 1.0                                                  *)
(* Autor       : (c) S. Spolwig (OSZ-Handel I)  10997 Berlin          *)
(*                                                                    *)
(* Aufgabe     : stellt die abstrakte Klasse Geld dar.                *)
(* Compiler    : Delphi 4.0                                           *)
(* Aenderung   : V.1.0  - 25-DEC-99                                   *)
(* ****************************************************************** *)

INTERFACE
(* ================================================================== *)
type
  TBargeld = class (TObject)
             protected
              Wert      : 0..5000;  (* in Pfennig *)

             public
              constructor Create;                            virtual;
              procedure   SetPos (ax1, ay1, ax2, ay2 : Word);virtual;
              procedure   PosVersetzenUm (dX, dY : integer);  virtual;
              procedure   BewegenNach (neuX, NeuY : integer); virtual;
              procedure   Zeigen;              virtual;
              procedure   Loeschen;            virtual;
              function    IstEcht : boolean;   virtual;
              function    GetWert : Word;      virtual;
             end;


IMPLEMENTATION
(* ================================================================== *)

constructor TBargeld.Create;
(* ------------------------------------------------------------------ *)
begin
  inherited Create;
  Wert := 0;
end;

procedure TBargeld.SetPos (aX1, aY1, aX2, aY2 : word);
(* ------------------------------------------------------------------ *)
begin
end;

...

END.
```

```
UNIT uMuenze;
(* ******************************************************************** *)
(* K L A S S E : TMuenze                                                *)
(* ******************************************************************** *)

INTERFACE
(* ========================= Export ================================== *)
USES
   Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
   Dialogs, StdCtrls, ExtCtrls, uZEIT, uGRAFIK, uBargeld;

type
  TMuenze = class (TBargeld)
             protected
              Form      : TKreis;
              Radius    : integer;
              Praegung  : TText;
              Farbe     : TColor;  (* clYellow, Clwhite = gueltige *)

             public
              constructor Create;                                override;
              procedure   SetPos (ax, ay : Word); reintroduce;   virtual;
              procedure   PosVersetzenUm (dX, dY : integer);     override;
              procedure   BewegenNach (neuX, NeuY : integer);    override;
              procedure   Zeigen;             override;
              procedure   Loeschen;           override;
              function    IstEcht : boolean;  override;
             end;

...
IMPLEMENTATION
(* ==================================================================== *)

constructor TMuenze.Create;
(* -------------------------------------------------------------------- *)
begin
  inherited Create;
  Farbe := clPurple;       // Falschgeld
  Radius:= 22;

  Form := TKreis.Create;
  Form.SetPos(0,0);
  Form.Setfarbe(clBlack);
  Form.SetFuellfarbe(Farbe);
  Form.Setradius(Radius);

  Praegung := TText.Create;
  Praegung.Setfont('Elefant',10);
  Praegung.SetFarbe(clBlack);
  Praegung.SetFuellFarbe(Form.GetFuellFarbe);
  Praegung.Settext('  ');
end;

procedure TMuenze.SetPos (aX, aY : word);
(* -------------------------------------------------------------------- *)
begin
  Form.SetPos(ax,ay);
end;

...
END.
```

```
UNIT uMark;
(* ******************************************************************** *)
(* K L A S S E : TMark                                                  *)
(* ******************************************************************** *)

INTERFACE
(* ========================== Export ================================== *)

USES Graphics, uGrafik, uMuenze;

type
  TMark = class (TMuenze)
            public
              constructor Create; override;
              procedure SetPos (aX, aY : word); override;
          end;
...

IMPLEMENTATION
(* ==================================================================== *)

constructor TMark.Create;
(* -------------------------------------------------------------------- *)
begin
  inherited Create;
  Wert  := 100;
  Farbe := clWhite;

  Form.setfuellfarbe(Farbe);
  Form.Setradius(22);
  Praegung.Setfarbe(clBlack);
  Praegung.SetFuellFarbe(Form.GetFuellfarbe);
  Praegung.Setfont('Elefant',24);
  Praegung.Settext('1');
end;


procedure TMark.SetPos (aX, aY : word);
(* -------------------------------------------------------------------- *)
begin
  inherited SetPos(ax,ay);
  Praegung.SetPos(Form.GetXPos - 11, Form.GetYpos - 17);
end;
...
END.
```

```
UNIT uQListe;
(* ******************************************************************* *)
(* K L A S S E   : TQListe - abstrakte Klasse                          *)
(* ******************************************************************* *)

INTERFACE
(* =================================================================== *)
const
   MAXLAENGE = 1000;

type
  TQListe  = class (TObject)
             protected
              Kollektion  : array [0..MAXLAENGE + 1] of Pointer;
              LiLaenge    : Word;     // Anzahl der belegten Elemente

             public
              constructor Create;                        virtual;
              procedure Init;                            virtual;

              procedure Append (Elem : pointer);         virtual;
              function  GetElement: pointer;             virtual;

              procedure RemoveAll;                       virtual;
              function  GetLen   : integer ;             virtual;
              function  IsEmpty : boolean;               virtual;
              function  IsFull   : boolean;              virtual;

              procedure Load (Dateiname : string);       virtual;
              procedure Store(Dateiname : string);       virtual;
            end;

...
END.
```

```
UNIT uWechselEinheit;
(* ******************************************************************* *)
(* K L A S S E : TWechselEinheit                                       *)
(* ------------------------------------------------------------------- *)
(* Aufgabe     : Stellt das Bauteil zur Annahme und Ausgabe von Noten   *)
(*               und Muenzen dar. Es steuert den internen Wechselvorgang*)
(*               Die Rueckgabe erfolgt nur in Muenzen. Falschgeld wird  *)
(*               wieder ausgeworfen.                                    *)
(*               Die auszugebenden Muenzen werden einem Ausgabeschacht  *)
(*               uebergeben. Keine kaufm. Buchfuehrung.                 *)
(* ******************************************************************* *)

INTERFACE
(* =================================================================== *)

USES
   SysUtils, uZeit, uGrafik, uQListe,  // fuer Schächte
   uMuenze, uNote, uGroschen, uFuenfziger, uMark, uZweimark, uFuenfmark,
   uZehnmark, uZwanzigmark;

type
   TWechselEinheit = class (TObject)
                 private
                  Schacht20M,
                  Schacht10M,
                  Schacht5M,
                  Schacht2M,
```

```
                  Schacht1M,
                  Schacht50,
                  Schacht10,
                  Ausgabeschacht : TQListe;
                 public
                  constructor Create;
                  procedure SchaechteFuellen;            virtual;
                  procedure GeldAnnehmen (Note : TNote); virtual;
                  procedure ZehnMarkWechseln;            virtual;
                  procedure ZwanzigMarkWechseln;         virtual;
                  procedure MuenzenAuswerfen;            virtual;
                  procedure NoteAuswerfen (Note : TNote);virtual;
                  function  HatZuwenigMuenzen : boolean; virtual;
               end;
...

IMPLEMENTATION
(* ==================================================================== *)
...

constructor TWechselEinheit.Create;
(* -------------------------------------------------------------------- *)
begin
  inherited Create;
  Ausgabeschacht := TQListe.Create;
  Schacht20M     := TQListe.Create;
  ....
end;


procedure TWechselEinheit.SchaechteFuellen;
(* -------------------------------------------------------------------- *)
begin
  while NOT Schacht5M.IsFull do
  begin
    Fuenfmark := TFuenfmark.Create;
    Schacht5M.Append(Fuenfmark);
   ...
  end;
end;

...

procedure TWechselEinheit.ZwanzigMarkWechseln;
--------------------------------------------------------------------
var
  i : integer;
begin
  Ausgabeschacht.Init;

  if NOT HatZuwenigMuenzen
  then
    begin
      for i :=  1 to 3 do
      begin
        Ausgabeschacht.Append(Schacht5M.GetElement);
      end;
      Ausgabeschacht.Append(Schacht2M.GetElement);
      Ausgabeschacht.Append(Schacht2M.GetElement);
      Ausgabeschacht.Append(Schacht1M.GetElement);
    end;
end;
```

```
procedure TWechselEinheit.MuenzenAuswerfen;
(* ------------------------------------------------------------------ *)
...
begin
  ...
  while NOT Ausgabeschacht.IsEmpty do
  begin
    Muenze := Ausgabeschacht.GetElement;  // man weiss nicht, was kommt !

    if muenze is TFuenfMark
    then
      begin
        Muenze.SetPos(XStart + abstand, YStart - 10);
      End
      ...

    Muenze.Zeigen;

 end; // while
end;
...
END.
```

Einsatz im Unterricht

Der Vorzug dieses Beispiels liegt in der Einfachheit der Fachklassen: Geld wie aus dem Wirtschaftslehrebuch; bzw. wie es jedes Kind schon einordnen kann - also keine langwierige Systemanalyse. Man kommt schnell zur Sache. Und ein Beispiel, das sich nicht nur schnell erschließt sondern auch in hervorragender Weise die Denkweise in Klassenhierarchien demonstriert, besser als z. B. kaufmännische Anwendungen, die von Gymnasialschüler kaum erarbeitet werden können, weil ihnen normalerweise der betriebswirtschaftliche Hintergrund und die Detailkenntnisse fehlen.

Das Programm wurde vom Lehrer gemacht, um als Einführung in die Programmierung (Berliner Rahmenplan - Analyse eines komplexen Systems mit anschließenden Modifikationen), wie schon mal in LOGIN früher mit "Fahrscheinautomat" beschrieben. Auch hier lassen sich algorithmische Teilprobleme bearbeiten, z. B. bei der Ausformung der Methoden innerhalb einer Fachklasse. Entscheidend ist aber der erste Eindruck, dass Programmsysteme nach den beteiligten Objekten strukturiert sind.

Die andere Möglichkeit ist die vollständige Erstellung als Programmierprojekt für ein Halbjahr. Die reine Kodierarbeit lag etwa bei 25 MannStunden, wobei die verwendete Grafikklasse vorlag. Hoch gerechnet dürfte das für eine Schülergruppe für ein halbes Jahr reichen. Wem diese Version noch zu dürftig ist, kann noch eine Menge Verschärfung einbauen, z.B. kaufmännische Buchführung, flexibles Verhalten des Automaten, automatischer Schachtwechsel, wenn z.B. der 2-M-Schacht leer ist usw.

Unter der Voraussetzung, dass ein Schwergewicht auf der auf der Analyse der Anwendung und der Umsetzung in einem Modellierungsprozess liegt, sollte immer eine strikte Trennung zwischen dem eigentlichen Fachkonzept (hier Geld und Operationen, für die die Geldklassen verantwortlich sind) und den Ein- und Ausgaben durch den Benutzer (GUI) herbeigeführt

werden. Deshalb sollten die Klick-Methoden als Teil der Ereignissteuerung nur Klassenmethoden des Fachkonzeptes aufrufen, aber keinesfalls Aufgaben und Verarbeitungsprozesse des Fachkonzeptes beinhalten.
Ein lupenreiner Einsatz des MVC-Konzepts hätte erfordert, dass die grafische Darstellung der Münzen und Noten aus den Geldklassen herausgenommen würde. Da dafür aber keine GUI-Komponenten in DELPHI vorhanden sind, hätte man sie selbst erstellen müssen. Das wäre sehr aufwändig.

Plädoyer für den GUI-Builder

Um mehr Zeit für OOM im Unterricht zu gewinnen, hat sich der GUI-Builder als ein unverzichtbares Hilfsmittel erwiesen. Früher unter Turbo-Pascal wurde bis zu 90% der Kodierungszeit im Unterricht für *gotoxy*, *writeln* und *readln* verbraucht, um ein Programm halbwegs ansehnlich zu machen. Jeder Lehrer, der einmal versucht hat damit ein komfortables INOUT-Modul mit formatierter Ein- und Ausgabe aller gängigen Typen und Editiermöglichkeit vor und nach dem Komma usw. zu programmieren, weiß, wovon ich rede. (Meines war am Ende 1.347 Zeilen lang - schon daran kann man sehen, dass es noch nicht der richtige Programmierstil war.)

Dieser Teil der Programmierung z. B. mit DELPHI ist meistens in einer viertel Stunde mit ein paar Mausklicks erledigt. Wenn Schüler in der aller ersten Programmierstunde eine Oberfläche mit einigen Buttons und Labels zusammengeschoben haben, sind sie jedes Mal erstaunt, wie leicht es geht, wie professionell es aussieht, dass dieses "Programm" sofort funktioniert, aber leider nun absolut gar nichts tut. Genau diese Stelle ist der Zeitpunkt klar zu machen, dass es neben den verfügbaren Objekten des GUI-Builders wohl auch noch solche geben muss, die er nicht kennt und die man selbst herstellten muss, nämlich die Objekte der Fachklassen, damit das Programm seine eigentliche Aufgabe löst. Bereits so früh ist damit der entscheidende Schritt getan, den Schülern die Erkenntnis einzupflanzen, dass Daten (Fachklassen) und ihre Repräsentation auf dem Bildschirm zwei verschiedene Ebenen sind, die man getrennt behandelt (MVC-Konzept).
Nachdem wir diesen didaktischen Schritt gegangen sind, ist auch der früher an dieser Stelle geäußerte Widerstand gegen den Einsatz von GUI-Buildern relativiert worden. Damit wird auch deutlich, dass der Kern des Unterrichts auf der Analyse und der Modellierung des Fachkonzeptes liegt.

Mit einiger Gelassenheit kann man OOP als Weiterentwicklung der strukturierten Programmierung und der abstrakten Datentypen ansehen. Aber OOP ist nichts ohne OOM. Das war der wirkliche Durchbruch. Keiner meiner näheren Kollegen würde darauf wieder verzichten wollen.

Literatur

Meyer, H.: Unterrichtsmethoden, I + II Theorie- und Praxisband, Frankfurt/Main, 1992.
Lavergne, H. von: "Visuelle Welt - Schön". IN: LOG IN 19 (1999) Heft 5
Spolwig, S.: Objektorientierung im Informatikunterricht. Bonn: Dümmlers Verlag, 1997.
Spolwig, S.: "Hello World in OOP." IN: LOG IN 19 (1999) Heft 5

Der komplette Quellcode des Programms kann unter http://www.bics.be.schule.de bei Informatik Sek. II / Programmiersprachen heruntergeladen werden.