

Harro von Lavergne

Visuelle Welt - Schön

The missing link

In Heft 5 '98 der LOG IN weisen Penon/Spolwig in ihrem Beitrag *Schöne visuelle Welt?* [PS] auf ein für den gegenwärtigen Informatikunterricht wesentliches Dilemma hin; Informatikunterricht - der heilige Satz sei hier noch einmal geschrieben - soll die wesentlichen Grundlagen des Faches vermitteln, nicht aber gewiefte Experten einer bestimmten Programmiersprache ausbilden. In einem davon unabhängigen Beitrag [La] desselben Heftes erwähnte ich, daß eines der wichtigen Konzepte im neuen Markt die Erstellung von Komponenten ist und stellte heraus, daß Komponenten der Tendenz zur Sprachunabhängigkeit entsprechen im Sinn des *programming by contract*, Programmieren für und mit Schnittstellen, bei denen es nicht darauf ankommt, in welcher Sprache die dahinter stehenden Implementierungen erstellt wurden.

Das erwähnte Dilemma besteht darin, daß Programmieren ganz ohne Programmiersprache, obwohl prinzipiell möglich, nicht praktikabel ist. Die Autoren schildern den verdienstvollen Versuch dreier engagierter Kollegen, einen gangbaren Weg aus diesem Dilemma zu finden. Sie stellen an einem Beispiel mit Objektorientierter Programmierung (OOP) einige aktuelle Programmiertechniken vor und ihren Versuch, "zwei unvereinte Welten" zu versöhnen. Zwei Welten, deren erste die Notwendigkeit aufweist, eine einfache, möglichst schülergerechte Programmiersprache zu benutzen, von der Gelegenheitsprogrammierer, Schüler wie Lehrer also, nicht überfordert sind, deren zweite den verständlichen Wunsch hat, moderne Programmierverfahren darzustellen und mit graphischen, dem heutigen Standard entsprechenden Benutzeroberflächen Erwartungen und Vorstellungen von Schülern entgegenzukommen.

Im Fazit des Beitrages wird aufgezeigt, daß bei objektorientierter Programmierung, deren Nützlichkeit heute nicht mehr bezweifelt werden kann, die Verwendung einer der aktuellen Entwicklungsumgebungen wie Delphi oder das Java Development Kit (JDK) der inzwischen veralteten, kommandozeilenorientierten Turbo Pascal Umgebung vorzuziehen sei. Gleichzeitig weisen die Autoren zu Recht darauf hin, daß die meisten der bekannteren Entwicklungsumgebungen à la Delphi es zwar erlauben, schnell und unkompliziert einfache Benutzeroberflächen zu erzeugen, aber jeder Versuch, eine über den vorgegebenen Standard hinausgehende Benutzeroberfläche selbst zu entwerfen oder gar zu programmieren, eine intensive, in der Schule nicht zu bewältigende Beschäftigung mit den weitverzweigten, tief ineinander geschachtelten Bibliotheken der jeweiligen Systeme erfordern (ich durfte einem Kollegen, der bei einem solchen, ein dreiviertel Jahr währenden Versuch bewundernswert gescheitert ist, ehrfurchtsvoll schauernd zusehen).

Offensichtlich ist also keine der drei verwendeten Entwicklungsumgebungen uneingeschränkt für den Unterricht zu empfehlen. Von den mit ihnen verbundenen Programmiersprachen erscheint im Tenor des Beitrages Pascal wegen der Einfachheit und Eleganz des Sprachkonzepts immer noch am ehesten geeignet, obwohl gerade der (nachträglich angefügte) Objektteil der Sprache mit ihrer falschen Terminologie und der für Schüler schwer zu begreifenden Verwendung von Konstruktoren und Destruktoren als wenig gelungen angesehen werden muß. Ein schwerwiegender (in [PS] nicht erwähnter) Grund, Turbo Pascal auszuschließen, liegt darin, daß der Objekt-Teil lediglich statische Polymorphie kennt, OO-Programmierung jedoch dynamische Polymorphie erfordert. Ein weiteres, ebenso gravierendes Manko ist das Fehlen einer dynamischen Speicherverwaltung (garbage collection).

Das Arbeiten mit Delphi als (terminologisch) nur wenig verbesserter Pascal Dialekt ist im Prinzip ebenso einzuschätzen. Eine automatische Speicherverwaltung fehlt wie bei Turbo Pascal, vorteilhaft ist die Existenz dynamischer Polymorphie; schulisch relevante Probleme ergeben sich hauptsächlich aus der Undurchschaubarkeit der Bibliotheken.

Java kennt dynamische Polymorphie und garbage collection, auf die Komplexität der Klassenbibliotheken des JDK und die damit verbundenen Schwierigkeiten bin ich in meinem oben erwähnten Beitrag eingegangen. Ein weiteres, im Schulunterricht wahrscheinlich relevantes Problem mit Java ist die stark gewöhnungsbedürftige, der Programmiersprache C angelehnte Syntax (ich habe bis heute nicht begriffen, warum ein Gleichheitszeichen ein Zuweisungsoperator sein soll, damit anschließend eine Gleichheit [im mathematischen Sinn] durch ein doppeltes Gleichheitszeichen dargestellt werden muß). Wichtiger als diese Geschmacksfragen aber scheint mir zu sein, daß Java eine rein objektorientierte, ausschließlich mit Klassen und Methoden arbeitende Sprache ist. Eine derartige Sprache macht es praktisch unmöglich, die keineswegs veralteten Verfahren eines strukturierten, prozedurorientierten Programmierstils zu vermitteln (ich werde am Ende dieses Beitrags darauf zurückkommen).

Der folgende Teil soll zeigen, daß es eine Versöhnung der "unvereinbaren Welten" geben kann, wenn man mit der Programmiersprache Component Pascal und der dazugehörigen Entwicklungsumgebung BlackBox arbeitet. Component Pascal ist die jüngste der von Nicklaus Wirth geschaffenen Sprachen der Erbschaftsreihe Pascal -> Modula -> Oberon -> Component Pascal und enthält die ganze Erfahrung, die bei der Entwicklung der Vorgängersprachen gewonnen wurde. Component Pascal besitzt alle wesentlichen Eigenschaften der Vorgängersprachen, wurde aber, wie schon die anderen Sprachen dieser Ahnenreihe, konsequent auf Grund von im Lauf der Zeit hinzugekommenen Erkenntnissen verbessert und neu aufgetauchten Notwendigkeiten angepaßt. Anders aber als viele "alte" Sprachen, denen im Lauf der Zeit lediglich immer mehr "Fähigkeiten" zugefügt wurden, sind Oberon und Component Pascal im Zuge des Entwicklungsprozesses "verschlankt" worden nach dem Einstein-Motto (von Wirth A. E. zugesprochen): *Make it as simple as possible, but not simpler.*

Obwohl die BlackBox Umgebung nicht speziell für den Einsatz im Unterricht geschaffen worden ist, sondern als professionelles Instrument zur schnellen Erstellung von Anwendungen (RAD, Rapid Application Development) und für die Programmierung von Komponenten (COP, Component Oriented Programming oder CBD, Component Based Development genannt, manchmal auch referiert als *beyond objects*), eignet sie sich auf Grund ihrer Entwicklungsgeschichte sehr gut für den Einsatz in der Ausbildung.

Component Pascal ist eine Hybridsprache, die mit den aus Pascal bekannten skalaren und strukturierten Typen alle Möglichkeiten der prozeduralen und modularen Programmierparadigmen bietet, gleichzeitig enthält sie ein simples und effizientes Konzept der Typerweiterung von RECORD-Typen, mit dem die wesentlichen Möglichkeiten der OOP, Vererbung, Polymorphie etc. realisiert werden können, ohne dafür ein gesondertes Klassenkonstrukt einzuführen, wie das in den meisten anderen OO-Hybridsprachen der Fall ist. In Component Pascal können daher OOP-Techniken an den Stellen eingesetzt werden, an denen sie sinnvoll sind und an den Stellen weggelassen werden, an denen ihre Verwendung sinnlos oder sogar nachteilig ist.

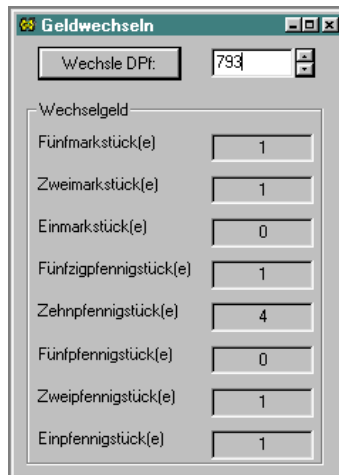


Abb. 1a
Dialogbox im Benutzer-Modus

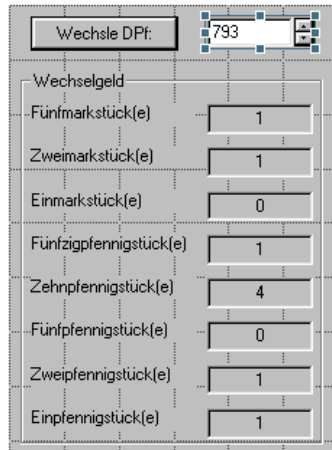


Abb. 1b
Dialogbox im Layout-Modus

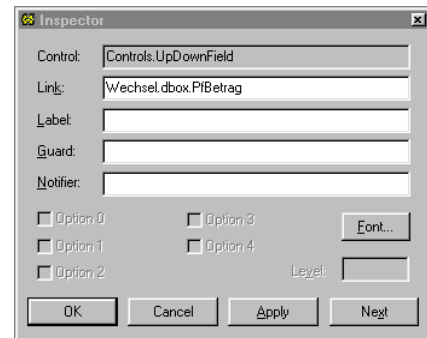


Abb. 2
Inspektor-Dialog

Das BlackBox System stellt mit einer relativ kleinen, gut gegliederten und optimal auf die Sprache abgestimmten Modulbibliothek eine Entwicklungsumgebung dar, die die Bezeichnung Rapid Application Development verdient. Insbesondere sind die Verfahren zur Erstellung graphischer Anwenderschnittstellen (GUI) bemerkenswert einfach, wie im folgenden gezeigt werden soll. In BlackBox stehen alle üblichen Kontrollelemente wie Schaltflächen, Eingabefelder usw. zur Verfügung, anders aber als in vielen Entwicklungsumgebungen gibt es hinter den damit erstellten Eingabemasken keinen versteckten Programmcode, der oft die eigenständige Weiterentwicklung dieser Masken behindert oder unmöglich macht, es wird lediglich ein Layout erzeugt, dessen Merkmale in einer normalen Textdatei gespeichert und von dort wieder geladen werden können. Die einzelnen Kontrollelemente werden jeweils beim Laden des gespeicherten Layouts dynamisch mit den zugeordneten Daten bzw. Prozeduren der entsprechenden Code-Dateien verbunden (late linking oder dynamic linking), eine Änderung des ausführbaren, dem Maskeninhalte zugrundeliegenden Codes führt deshalb unmittelbar zu einer entsprechenden Verhaltensänderung, sofern die in der Maske verwendeten (exportierten) Bezeichner der den Maskenkontrollelementen zu Grunde liegenden Module nicht geändert wurden.

An einem betont einfachen Beispiel, das ich bereits im ersten Semester des Oberstufen-Informatikunterrichts eingesetzt habe, möchte ich die Konstruktion graphischer Benutzeroberflächen in Component Pascal erläutern. Das folgende Modul *Wechsel* bestimmt mittels eines einfachen Algorithmus' zu einem beliebigen, vom Benutzer einzugebenden Pfennigbetrag die Anzahlen der bei einem "optimalen" Wechselvorgang auszugebenden Münzen.

```

MODULE Wechsel;

IMPORT
  Dialog;

VAR
  dbox*: RECORD
    PfBetrag*: INTEGER; (* Zu wechselnder Betrag *)
    fünfhundert-, zweihundert-, einhundert-, (* Pf-Beträge der existie- *)
    fünfzig-, zehn-, fünf-, zwei-, ein-: INTEGER; (* renden Münzen *)
  END;

PROCEDURE Wechseln*;
VAR
  Betrag: INTEGER; (* Jeweils aktueller Rest *)
BEGIN
  Betrag := dbox.PfBetrag;
  dbox.fünfhundert := Betrag DIV 500; Betrag := Betrag MOD 500;
  dbox.zweihundert := Betrag DIV 200; Betrag := Betrag MOD 200;
  dbox.einhundert := Betrag DIV 100; Betrag := Betrag MOD 100;

```

```

dbox.fünfzig := Betrag DIV 50; Betrag := Betrag MOD 50;
dbox.zehn := Betrag DIV 10; Betrag := Betrag MOD 10;
dbox.fünf := Betrag DIV 5; Betrag := Betrag MOD 5;
dbox.zwei := Betrag DIV 2;
dbox.ein := Betrag MOD 2;
Dialog.Update(dbox); (* Aktualisieren des Dialogs *)
END Wechseln;

BEGIN (* Dateninitialisierung *)
dbox.PfBetrag := 0;
dbox.fünfhundert := 0; dbox.zweihundert := 0; dbox.einhundert := 0;
dbox.fünfzig := 0; dbox.zehn := 0; dbox.fünf := 0; dbox.zwei := 0;
dbox.ein := 0;
END Wechseln.

```

Die zugehörige Datenmaske beruht auf den vom Modul *Wechsel* exportierten Bezeichnern, der Prozedur *Wechseln* und dem Verbund *dbox*, dessen Felder ebenfalls exportiert werden, wobei die Exporte, bis auf das Feld *PfBetrag*, schreibgeschützt erfolgen (Exporte werden entweder mit einer "*" Marke [read/write] oder mit einer "-" Marke [read only] gekennzeichnet). Alle exportierten Bezeichner finden sich in den ersten beiden abgebildeten Dialogboxen wieder, die Prozedur *Wechseln* als Schaltfläche, das Verbundfeld *dbox.PfBetrag* als (nicht schreibgeschütztes) up/down Feld, die übrigen Verbundfelder stehen innerhalb des Rahmens *Wechselgeld*, wobei die andersfarbige Unterlegung der Felder deren schreibgeschützten Status signalisiert.

Solche Dialogboxen können automatisch vom BlackBox System anhand der Exporte erzeugt werden, lassen sich aber auch individuell erstellen. Wesentlich ist dabei die im Vergleich zu Systemen wie Delphi ungewöhnliche Art, in der eine Dialogbox mit dem zugrunde liegenden Modul (hier *Wechsel*) verbunden ist. Die gesamte Box ist in einer gewöhnlichen Textdatei gespeichert, in der lediglich das Layout und die Verknüpfungen stehen, aber keinerlei Programmcode existiert. Gespeichert werden außer den Layout-Daten nur die Angaben über die den einzelnen Kontrollelementen zugeordneten Bezeichner des Basismoduls. Das Öffnen einer gespeicherten Dialog-Layout-Datei führt zu einem dazu, daß die Box auf dem Bildschirm gezeichnet wird, zum anderen werden die Kontrollelemente dynamisch mit den Basismodulbezeichnern verbunden, wobei das (übersetzte) Modul in den Arbeitsspeicher geladen wird, sofern noch nicht vorhanden (dynamic linking).

Während die Abbildung 1a den Dialog in der Version Benutzer-Modus zeigt, reproduziert die Abbildung 1b den Layout-Modus, in dem die einzelnen Kontrollelemente bearbeitet werden können, das up/down Feld ist markiert. Die Abbildung 2 zeigt den Objektinspektor, mit dem die Eigenschaften der einzelnen Kontrollelemente der Dialogbox festgelegt werden. Das wichtigste Feld des Inspektors ist das *Link*-Feld, in dem die speicherbare Verbindung des Dialogfeldes zu einem (exportierten) Bezeichner eines Moduls hergestellt wird, hier also die Verbindung zwischen dem Kontrollelement *Controls.UpDownField* und der Variablen *Wechsel.dbox.PfBetrag*, entsprechendes gilt für die übrigen Felder des Dialogs.

Als einziger Hinweis darauf, daß es einen im Hintergrund ablaufenden Prozess der Datenübermittlung zwischen dem Modul *Wechsel* und der Dialogbox geben muß, kann die letzte Zeile *Dialog.Update(dbox)* der Prozedur *Wechseln* angesehen werden, in der die Prozedur *Dialog.Update* aufgefordert wird, den Verbund *dbox* zu aktualisieren, womit nicht die Datenfelder innerhalb des Moduls *Wechsel* gemeint sind, sondern alle Sichten auf diese Daten innerhalb der BlackBox Systemumgebung (genau genommen ist BlackBox nach heutiger Terminologie nicht ein *System*, sondern ein *framework*; zur Klassifizierung der Unterschiede s. beispielsweise das Buch von C. Szyperski *Component Software* [Sz], die Bezeichnung *System* behalte ich aus Bequemlichkeitsgründen bei). Dem anwendungsorientierten Programmierer von Applikationen und dem Anfänger im Informatikunterricht wird an dieser Stelle die Arbeit mit Systembibliotheken bzw. Schnittstellendokumentationen weitgehend abgenommen, das System erledigt (fast) alles im Hintergrund.

Dieser Bequemlichkeit liegt eine konsequente Anwendung moderner Programmwurfsmuster zugrunde, wie sie in dem sehr empfehlenswerten Buch *Design Patterns* von Gamma et. al. [GHJV] beschrieben werden. Bei diesen Entwurfsmustern handelt es sich um eine Sammlung immer wiederkehrender Verfahren zur Behandlung von Standardprogrammierproblemen, die praktisch in allen gegenwärtigen Programmiersprachen verwendbar sind und deshalb in [GHJV] weitgehend sprachunabhängig dargestellt werden. Viele der beschriebenen Entwurfsmuster haben im wesentlichen dasselbe Ziel, die Entkopplung der Daten eines Programms von der Verarbeitung durch die jeweilige Systemumgebung.

Der in [PS] auftauchende Hinweis auf solche Entwurfsmuster und das MVC-Konzept, das als eines der ersten Verfahren zur Separation der Daten von ihrer Verarbeitung vielen der in [GHJV] beschriebenen Muster zugrunde liegt, scheint mir der wichtigste Aspekt des Beitrages von [PS] zu sein. Wie am Beispiel des Moduls *Wechsel* zu sehen ist, lassen sich auf der Grundlage dieses Konzeptes und der Anwendung der jeweils geeigneten Entwurfsmuster die von *Wechsel* exportierten Daten (Model) und die Bildschirmdarstellung in der Dialogbox (View) sowie die Dateneingabe durch Tastatur oder Maus (Controller) fast völlig entkoppeln.

Ein gut konzipiertes framework kann aber nicht nur wie bei dem Modul *Wechsel* sein Künste hinter den Kulissen zur Verfügung stellen, es läßt sich auch dazu verwenden, die benutzten Verfahrensweisen durchsichtig zu machen, ohne den (End-) Anwender zu überfordern. Das zweite der beiden nachstehenden Module, *ScanForm1*, zeigt an dem simplen Beispiel der Summierung einiger Zahlen wesentliche Muster des MVC-Konzeptes in einer Form, die für Schüler im zweiten oder dritten Semester verständlich ist.

Zuerst sei aber die Anfängerversion vorgestellt, die sich bereits zu Beginn des ersten Ausbildungssemesters von Schülern realisieren läßt (ohne Verwendung von Masken, für den Anfängerunterricht sind die beiden Module *In* und *Out* ausreichend)

```
MODULE ScanForm0;
  IMPORT
    In, Out;
  (* ----- Summieren *)
  PROCEDURE Summieren*;
    VAR
      Summe, Zahl: REAL;
  BEGIN
    Summe := 0.0;
    In.Open;
    In.Real(Zahl);
    WHILE In.Done DO
      Summe := Summe + Zahl;
      In.Real(Zahl);
    END;
    Out.String("Die Summe der Zahlen ist: ");
    Out.Real(Summe, 6);
    Out.Ln;
  END Summieren;
END ScanForm0.
```

Die Prozedur *Summieren* liest mit Hilfe der Routinen des Moduls *In* Zahlen ab einer markierten ersten Zahl ein und zeigt die Summe mit Hilfe von Modul *Out* in der Standardausgabe an.

Das folgende Modul *ScanForm1* führt im Wesentlichen den gleichen Prozess aus, allerdings ist die Eingabe exakt auf die markierten Zahlen beschränkt und die Ausgabe erfolgt einmal am Ende des aktuellen Fensters, aus dem die Zahlen eingelesen wurden und ein zweites Mal in einem neu erstellten, separaten Fenster.

```
MODULE ScanForm1;
  (* Das MVC - Muster *)
  IMPORT
    TextModels, TextControllers, TextMappers, TextViews, Views;
  (* ----- Summieren *)
  PROCEDURE Summieren*;
  (*
    VAR
      m0: TextModels.Model;
      m1: TextModels.Model;
  *)
```

```

(* TextModels.Model ist eine Unterklasse von Models.Model und stellt den auf
Texte spezialisierten Modellteil des Model-View-Controller Musters dar *)
    v1: TextViews.View;
(* TextViews.View ist eine Unterklasse von Views.View und stellt den auf
Texte spezialisierten Sichtteil des Model-View-Controller Musters dar *)
    c: TextControllers.Controller;
(* TextControllers.Controller ist eine Unterklasse von Controllers.Controller
und stellt den auf Texte spezialisierten Kontrollteil
des Model-View-Controller Musters dar *)
    s: TextMappers.Scanner;
    f0, f1: TextMappers.Formatter;
(* TextMappers.Scanner und TextMappers.Formatter sind Iterator-Muster [GHJV],
die aus einem Textmodell lesen bzw. in ein Textmodell schreiben *)
    beg, end: INTEGER;
    Summe: REAL;
BEGIN
    Summe := 0.0;
    c := TextControllers.Focus();
(* TextControllers.Focus() gibt den Kontrollteil des aktiven Fensters zurück,
sofern dies ein Textfenster ist *)
    IF (c # NIL) & c.HasSelection() THEN
(* c.HasSelection() gibt TRUE zurück, sofern das aktive (Text-) Fenster
eine Markierung enthält *)
        m0 := c.text;
        s.ConnectTo(m0);
        s.ConnectTo(c.text);
(* Iterator s und Modell (m0) des Kontrollteils c werden verbunden; die
beiden auskommentierten Zeilen zeigen das Prinzip ausführlich *)
        c.GetSelection(beg, end);
(* Anfang und Ende des markierten Intervalls [beg, end) werden erfasst *)
        s.SetPos(beg);
        s.Scan;
        WHILE (s.type = TextMappers.real) & (s.Pos() <= end + 1) DO
            Summe := Summe + s.real;
            s.Scan;
        END;
(* Addition der vom Scanner-Objekt gelesenen Werte der Markierung
zur Summe *)
        f0.ConnectTo(m0);
        f0.SetPos(m0.Length());
        f0.ConnectTo(c.text);
        f0.SetPos(c.text.Length());
(* Ein Formatter-Objekt wird mit dem Textmodell des aktiven Fensters
verbunden und am Ende des Textes positioniert *)
        f0.WriteRealForm(Summe, 5, 7, 0, 8FX);
(* Schreiben der Summe in das Modell des Formatter-Objekts; das Modell
veranlasst alle [geöffneten] Sichten, die Darstellung zu aktualisieren *)
        m1 := TextModels.dir.New();
(* Ein neues, leeres Textmodell wird alloziert; TextModels.dir.New() ist
eine Fabrik-Methode (factory method) [GHJV] *)
        f1.ConnectTo(m1);
(* Ein Formatter-Objekt wird mit dem neuen Textmodell verbunden *)
        f1.WriteString("Die Summe der markierten Zahlen ist: ");
        f1.WriteRealForm(Summe, 5, 7, 0, 8FX);
(* Schreiben der Summe in das Modell *)
        v1 := TextViews.dir.New(m1);
        Views.OpenView(v1);
        Views.OpenView(TextViews.dir.New(m1));
(* Allokierung einer neuen Sicht für das Textmodell m1 und Öffnen eines
neuen (Text-) Fensters als Sicht auf das Modell.
Da TextViews.View eine Unterklasse von Views.View ist, kann das Objekt v1
als Objekt der Oberklasse geöffnet werden (dynamische Polymorphie) *)
    END;
END Summieren;
END ScanForm1.

```

Die im Modul *ScanForm1* gezeigte Anwendung des *Model-View-Controller-Paradigmas* verdeutlicht, daß es möglich ist, Schülern im zweiten oder dritten Ausbildungssemester eine vereinfachte Version dieses grundlegenden Musters verständlich zu machen, wenn ein component framework wie *BlackBox* die entsprechenden Grundlagen zur Verfügung stellt. Selbstverständlich wäre es hypertroph zu glauben, man könne derartig komplexe Datenstrukturen im Anfängerunterricht individuell erstellen (lassen), ein framework zu konstruieren, erfordert jahrelange Erfahrung und ist auch für durchschnittlich professionelle Programmierer nicht machbar. Es ist aber möglich, in der dargestellten Form einen Einblick in die zugrundeliegenden Datenstrukturen und die dahinter stehenden Ideen zu vermitteln.

Die in das Modul *ScanForm1* eingefügten Kommentare sollten für das Verständnis der Einzelheiten ausreichend sein, ich beschränke mich im folgenden auf wenige grundlegende Erläuterungen.

Abbildung 3 zeigt das Prinzip des *Model-View-Controller Schemas*, wie es im Modul *ScanForm1* angewendet wird

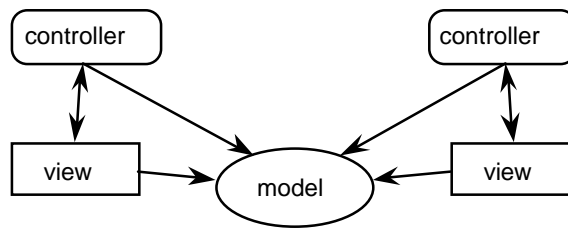


Abb. 3
Das MVC-Paradigma

Abbildung 4 gibt die konkreten Vererbungs- und Importbeziehungen innerhalb der BlackBox Systembibliothek wieder (in der *Unified Modeling Language* stellt ein Dreieck eine Vererbungsbeziehung dar [class inheritance], ein Pfeil mit Raute eine Importbeziehung [class composition]).

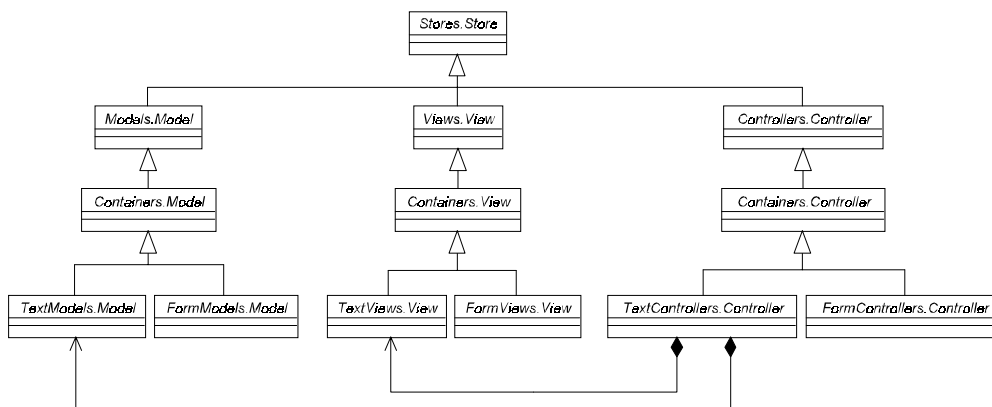


Abb. 4
UML-Diagramm der Klassenhierarchien des MVC-Schemas im BlackBox-framework
Die Pfeile mit Raute zeigen Importbeziehungen (class composition)

Ein Modell (model) ist der Prototyp einer Datenstruktur, es repräsentiert die Datenstruktur, ohne selbst zu wissen, wie die Daten dargestellt werden. Die Darstellung der Daten erfolgt durch eine dem Modell zugeordnete Sicht (view), wobei es mehr als eine Sicht des Modells geben kann (beispielsweise zwei verschiedene Ausschnitte eines Textes, statistische Daten als Zahlen oder Diagramme usw; die beiden Versionen der Dialoge zum Modul *Wechsel* [Abb. 1a und 1b] stellen zwei Sichten desselben Modells dar). Zu jeder Sicht gehört ein Kontrollteil (controller), der die aus Veränderungen der Daten resultierenden Aktionen veranlasst und zwischen Modell und Sicht vermittelt, Sicht und Kontrollteil sind deshalb in vielen Implementationen zusammengefasst. Alle Sichten zeigen dasselbe Modell, wenn sich das Modell ändert, müssen die Sichten nachgeführt werden. Damit dies möglich ist, sind im MVC-Schema die verschiedenen Ebenen separiert, eine Sicht bzw. ein Kontrollteil sorgen niemals selbst für die Veränderung (z. B. Auffrischen der Bildschirmdarstellung), dies geschieht ausschließlich auf dem Weg über das Modell, da nur auf diese Weise garantiert werden kann, daß alle Sichten konsistent bleiben.

In Kategorien von Entwurfsmustern ist das MVC-Schema eine Realisierung von im wesentlichen drei Mustern, *observer*, *composite* und *strategy*, die in [GHJV] folgendermaßen beschrieben werden:

1. **Observer:** Definiert eine 1:∞ Beziehung zwischen Objekten, so daß bei Änderung des "1"-Objekts alle anderen

- Objekte benachrichtigt und geändert werden (Modell-Sicht-Beziehung).
2. **Composite:** Zusammenfassung mehrerer Einzelobjekte zu einem Gesamtobjekt. Dies meint die Tatsache, daß Sichten hierarchisch gestaffelt sein können, eine Textsicht also beispielsweise eine Graphik enthalten kann, die ihrerseits weitere Sichten mit möglicherweise wiederum darin enthaltenen Sichten präsentiert.
 3. **Strategy:** Hüllobjekt für eine Gruppe ähnlicher Algorithmen, das die Wahl des passenden Algorithmus' generalisiert und vom jeweiligen Klienten entkoppelt. In diesem Sinn ist ein Kontrollobjekt ein Strategie-Objekt, es entkoppelt Benutzer-Aktionen (Tastatur, Maus usw) vom darunter liegenden Modell.

Ein weiteres, in den Kommentaren zu *ScanForm1* explizit erwähntes Entwurfsmuster ist das *Iterator*-Muster. Ein Iterator ist ein Objekt, das eine Datenstruktur durchlaufen kann, ohne daß diese im Klienten explizit bekannt sein muß. Typische Iteratoren sind Traversierungsalgorithmen für ARRAYS, lineare Listen oder binäre Bäume. Ein Iterator ermöglicht die Anwendung von klientenspezifischen Operationen auf die Datenstruktur, ohne daß der Klient die Implementierung des Traversierungsalgorithmus' kennt. Zwei derartige Iteratorklassen, die vom BlackBox System zur Verfügung gestellt werden, sind *TextMappers.Formatter* und *TextMappers.Scanner*.

Ein interessantes Detail der Verwendung des MVC-Schemas in BlackBox möchte ich noch erwähnen, das aus der Entkoppelung von Modell und Sicht resultiert. Der letzte Teil des Moduls *ScanForm1* erzeugt für die zweite Darstellung der berechneten *Summe* in einem separaten Fenster ein neues Textmodell und führt das *Formatter*-Objekt über die Datenstruktur, ohne daß die Daten sichtbar gemacht werden. Die Erzeugung und das Öffnen der neuen, zweiten Sicht auf die Daten erfolgt explizit erst nach Abschluß der Modellerstellung, dadurch ist einerseits eine schnelle Bearbeitung des Modells gesichert, andererseits wird die Darstellung der Daten auf dem Bildschirm nicht durch unnötige und komplizierte Aktualisierungen der Sicht verlangsamt, abgesehen davon, daß den Benutzer das damit verbundene unvermeidliche Bildschirmflackern vermutlich ärgert.

Im Rückblick möchte ich noch einmal die besondere Einfachheit der Erstellung von Datenmasken in Component Pascal hervorheben. Das Modul *Wechsel* importiert ein einziges BlackBox Modul, *Dialog*, und enthält eine einzige Zeile zur Aktualisierung der Datenmaske, *Dialog.Update(dbox)*. Die in dem programmtechnisch vergleichbaren Ratespiel bei [PS] sowohl für Delphi als auch für Java nötigen separaten Methoden *DatenAktualisieren* und *MaskeAktualisieren* erscheinen dagegen umständlich und wegen der benötigten Importe (und dem dazu gehörenden Erwerb des nötigen Verständnisses der Importhierarchien) Schülern wohl nur schwer begreiflich.

Zwei allgemeine Hinweise möchte ich an dieser Stelle noch anfügen. Bei meinen Gesprächen mit Fachkollegen wurde immer wieder als Hindernis für den Unterrichtseinsatz der Sprache Component Pascal (bzw. der quasikompatiblen Programmiersprache Oberon) erwähnt, daß hierfür keinerlei Lehrmaterial zur Verfügung stünde. Ich möchte deshalb zum einen darauf verweisen, daß es eine sehr ausführliche und verständlich geschriebene (englische) Dokumentation zum BlackBox System gibt [Pf], außerdem existiert seit ca. einem halben Jahr auf der home page der BlackBox/Component Pascal Programmierer (www.oberon.ch, dort ist auch das BlackBox System erhältlich) ein Verweis auf ein amerikanischsprachiges Tutorium von S. Warford [Wf]. Ich selbst schreibe zur Zeit an einem deutschsprachigen Component Pascal Tutorium, das unter <http://www.lahini.de> im Internet zu finden ist.

Weiter möchte ich alle, die sich für die Realisierung eines frameworks mit Text- und Graphikeditoren und einem GUI-Fenstersystem interessieren, auf das auch für Schüler in der Projektphase (5./6. Semester) verständliche Buch von H. Mössenböck *Objektorientierte Programmierung in Oberon 2* [Mö] hinweisen, in dem das MVC-Paradigma und die

Verwendung moderner Entwurfsmuster an einer exemplarisch vorgeführten Fallstudie dargestellt werden.

Eine grundsätzliche Anmerkung zum Einsatz objektorientierter Programmierung erscheint mir zum Schluß angebracht. Die Verwendung von Klassen, Methoden und Objekten ist bei der Benutzung der Systembibliotheken einer Entwicklungsumgebung zur Erstellung der graphischen Datenmasken zweifellos notwendig und sinnvoll, der Umgang mit ihnen sollte schon deshalb zu den Lernzielen des Informatikunterrichts gehören. Das Modul *Wechsel* macht deutlich, daß ein entsprechend konzipiertes framework den Anwendungsprogrammierer dabei weitgehend entlasten kann. Andererseits wäre bei einem simplen Programm wie dem Modul *Wechsel* (oder analog in [PS] bei dem Ratespiel) der Einsatz objektorientierter Programmierung für die Durchführung des Geldwechselforganges zweifellos absurd. Im Anhang möchte ich an einem ähnlichen Beispiel zeigen, daß sich zwar auch einfachste Programme mit Klassen und Objekten erstellen lassen, daß aber solche "Modernisierungen" sowohl den Quellcode als auch die ausführbaren (Objekt-) Dateien unnötig aufblähen. Klassen und Objekte sind, richtig eingesetzt, äußerst brauchbare Datenstrukturen zur Modellierung komplexer Applikationen und Halbfabrikate wie frameworks, sie sind prädestiniert für das Programmieren "im Großen", kleine Programme, wie die angesprochenen, lassen sich immer noch einfacher und effizienter in konventioneller Weise umsetzen, sofern nicht, wie in Java, zwangsweise alles mit Klassen und Methoden erstellt werden muß. Wer trotzdem meint, die "konsequente Anwendung der OOP-Konstrukte" auch im Anfängerunterricht vermitteln zu müssen, sei daran erinnert, daß mit der von B. Meyer geschaffenen Sprache Eiffel (<http://www.eiffel.com>) eine rein objektorientierte Sprache incl. entsprechende Entwicklungsumgebung existiert, die in ihrer Klarheit an Pascal erinnert, zu der es mit [Me] auch eine lesenswerte Einführung in die OOP-Grundlagen und -Techniken gibt.

Anhang:

Die beiden nachstehenden Module geben ein altbekanntes englisch/australisches Trinklied wieder; die erste Version, *BottlesOfBeers* ist in geradlinigem prozeduralen Code geschrieben; die zweite, *BeerSong* stellt dagegen eine formal korrekte Umsetzung im objektorientierten Stil dar. Die Anmerkungen am Anfang der zweiten Version erläutern, warum diese Version zwar möglich, aber unnötig aufgeblasen ist.

(Eine syntaktische Erklärung ist eventuell für diejenigen angebracht, die Component Pascal/Oberon nicht kennen, ein Prozedurkopf wie *PROCEDURE (VAR wall: ClassWall) LineEmUp* definiert eine Methode der Klasse *ClassWall*. Component Pascal/Oberon kennen keine gesonderten Klassenkonstrukte, jeder RECORD-Typ ist eine [potentielle] Klasse, Methoden sind spezielle, typgebundene Prozeduren, die an dem vor dem Prozedurnamen stehenden Empfängerparameter [receiver parameter] erkennbar sind).

```
MODULE BottlesOfBeers;
  IMPORT Out;
  PROCEDURE Start*;
  VAR
    bottles: INTEGER;
  BEGIN
    bottles := 99;
    REPEAT
      Out.Int(bottles, 2);
      Out.String(" bottles of beer on the wall, ");
      Out.Int(bottles, 2);
      Out.String(" bottles of beer.");
      Out.Ln;
      Out.String("Take one down, pass it around, ");
      DEC(bottles);
      Out.Int(bottles, 2);
      Out.String(" bottles of beer on the wall.");
      Out.Ln
    UNTIL bottles = 1;
    Out.String("1 bottle of beer on the wall, one bottle of beer.");
    Out.Ln;
    Out.String("Take one down, pass it around,");
    Out.String(" no more bottles of beer on the wall");
    Out.Ln
  END Start;
END BottlesOfBeers.

MODULE BeerSong;
(*
This module is intended as an example of the object-oriented programming
style where any communication should happen via method calls only.
Thus all procedures are methods (Type bound procedures in Component Pascal)
an exception being of course the exported command procedure Sing.
Remark:
Although this module is an example of how object-oriented programming
looks like, it first of all is an example of what not to do. On the one
hand none of the essential features and advantages of OOP can be shown in
a tiny thing like this beersong. And on the other hand it certainly shows
that OOP should be used only where appropriate, as all of this module's
output can be achieved with less effort both for the programmer and the
compiler by straightforward code-writing (see Module BottlesOfBeers). *)
  IMPORT
    Out;
  TYPE
    ClassWall = RECORD
      beer: BYTE;
    END;
  (* ----- AreThere *)
  PROCEDURE (VAR wall: ClassWall) AreThere (): BOOLEAN, NEW;
  BEGIN
    RETURN wall.beer > 0
  END AreThere;
  (* ----- LineEmUp *)
  PROCEDURE (VAR wall: ClassWall) LineEmUp, NEW;
  BEGIN
    wall.beer := 99
  END LineEmUp;
  (* ----- Paper *)
  PROCEDURE (VAR wall: ClassWall) Paper, NEW;
  BEGIN
    Out.String(" on the wall ");
    Out.Ln
  END Paper;
  (* ----- PassOneAround *)
  PROCEDURE (VAR wall: ClassWall) PassOneAround, NEW;
  BEGIN
```

```

        Out.String("Take one down and pass it around,");
        Out.Ln;
        DEC(wall.beer)
    END PassOneAround;
(* ----- SingOfBeer *)
PROCEDURE (VAR wall: ClassWall) SingOfBeer, NEW;
BEGIN
    IF wall.beer = 0 THEN
        Out.String("No more bottles of beer");
        wall.Paper;
        Out.Ln;
        Out.String("Go to the store buy some more");
        Out.Ln;
        Out.String("99")
    ELSE
        Out.Int(wall.beer, 1)
    END;
    Out.String(" bottle");
    IF wall.beer # 1 THEN
        Out.Char("s")
    END;
    Out.String(" of beer")
END SingOfBeer;
(* ----- SingVerse *)
PROCEDURE (VAR wall: ClassWall) SingVerse, NEW;
BEGIN
    wall.SingOfBeer;
    wall.Paper;
    wall.SingOfBeer;
    Out.Ln;
    wall.PassOneAround;
    wall.SingOfBeer;
    wall.Paper;
    Out.Ln
END SingVerse;
(* ----- Sing *)
PROCEDURE Sing*;
VAR
    WallBeers: ClassWall;
BEGIN
    WallBeers.LineEmUp;
    WHILE WallBeers.AreThere() DO
        WallBeers.SingVerse
    END
END Sing;
END BeerSong.

```

Literatur:

- [GHJV] Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J.: Design Patterns - Elements of Reusable Object-Oriented Software. Reading, Addison-Wesley, 1998 (auch auf Deutsch erschienen)
- [La] Lavergne, H. von: Thesen zur aktuellen Orientierungslosigkeit. In: LOG IN, 18 (1998), Heft 5, S. 19-23
- [Me] Meyer, B.: Object-Oriented Software Construction, Second Edition; Santa Barbara, Prentice Hall Professional Technical Reference, 1997
- [Mö] Mössenböck, H.: Objektorientierte Programmierung in Oberon 2; Berlin, Springer, 1998 (auch auf Englisch erschienen)
- [Pf] Pfister, C.: Component Software: A Case Study using BlackBox Components; Zürich, Oberon Microsystems, 1997
- [PS] Penon, J.; Spolwig, S.: Schöne visuelle Welt? In: LOG IN, 18 (1998), Heft 5, S. 40-46.
- [Sz] Szyperski, C.: Component Software - Beyond Object-Oriented Programming. Harlow, Addison-Wesley, 1997
- [Wf] Warford, J. S.: Programming in BlackBox; Pepperdine University, 1998 (Vorveröffentlichung)

(Erschienen in LOG IN 19 (1999) Heft 5, S. 43-50)

Der Autor ist per e-post zu erreichen unter:

havlav@lahini.de