Computer & Literatur

Herausgegeben von der Redaktion Toolbox

C++ Builder in Team

Software-Entwicklung und GUI-Design mit dem Borland-Compiler

- Ein- und Umstieg, AnsiStrings
- Undokumentiertes
- Datenbank-Entwicklung
- Internet-Programmierung
- Komponentenbau
- Grafik mit API, VCL und OpenGL
- Datenaustausch mit DDE und OLE
- Multitasking und Multithreading

- Borland Resource Workshop
- Über 500 C++Builder – getestete lizenzfreie Komponenten
- FixPacks für Windows 95,

KAPITEL 3	**OOP UND C++BUILDER**	**103**
	Warum OOP?	103
	Merkmale von OOP	107
KAPITEL 4	**ANSISTRINGS**	**143**
	Reingefallen... – oder doch nicht?	157
	Out of Delphi – es lebe C++!	159
	AnsiStrings und Windows-Ressourcen	164
	Was Borland vergessen hat – Versionsinformationen	171
	Wir erweitern die vom Projekt angelegte RES-Datei	171
KAPITEL 5	**STRUCTURED EXCEPTION HANDLING**	**175**
	Try und Catch	176
	Throw	179
	Exception-Spezifikation	180
	Konstruktoren und Exceptions	181
	Unbehandelte Exceptions	183
	Eigene Exception-Klassen	183
	Vordefinierte Exception-Klassen	185
	Exceptions der C++ Laufzeitbibliothek	185
	Exceptions der VCL	185
	Zusammenfassung	189
KAPITEL 6	**EVENT HANDLING MIT DER VCL**	**191**
	Von Menschen und Mäusen...	192
	MouseMove	192
	Durchgeschleift	196
	Spiel mir das Lied vom Event...	204
	Feuerzangenbowle – oder: Jetzt stellen wir uns mal ganz dumm...	205
	Message hui, Doku pfui...	205
	Alles hört auf mein Kommando...	216
	Menü-Gimmick	222
	Ohne Berührungsängste – die Tastatur	226
	OnKeyDown und OnKeyUp	227
	OnKeyPress	230
	Verschiebebahnhof mit Drag&Drop	232

	Control-Arrays selbstgemacht...	243
	Ein dynamisch erzeugter Label	250
KAPITEL 8	**AUSGESUCHT**	**267**
	Allgemeines zum System	268
	Wo ist noch Platz?	273
	Pixel über Pixel ...	276
	Wissenswertes zur Installation	280
	Schnittstellen	281
KAPITEL 10	**DLLS ERSTELLEN UND VERWENDEN**	**337**
	Einsatz von DLLs	337
	Entwickeln von DLLs mit dem C++Builder	339
	Einbinden von DLLs in C++Builder	349
	DLL-Interna	357

KAPITEL 3

OOP und C++ Builder

von Marian Heddesheimer

Warum OOP?

Warum diese Frage? Schließlich ist OOP »cool« und als »state of the Art«-Methode völlig »abgefahren« und außerdem verwendet doch wohl jeder objektorientierte Programmierung, wenn er nicht als Hacker-Dinosaurier abgestempelt werden will. Außerdem weiß ja schließlich auch schon jeder C++ Programmierer, was OOP ist, wie es sinnvoll verwendet wird und wie toll die Ergebnisse von objektorientierten Programmen sind. Damit wäre dann wohl alles gesagt und Sie könnten den Rest des Kapitels überblättern.

Leider ist die Sache nicht ganz so einfach, wie es in dieser etwas ironischen Form zusammengefaßt ist. Wer sich hin und wieder auf einschlägigen Seminaren oder Konferenzen mit Gleichgesinnten getroffen hat, wird vielleicht die Erfahrung gemacht haben, daß der Begriff »objektorientiert« oft nur ein Modewort ist. Man kann durchaus feststellen, daß verschiedene Experten unterschiedliche Auffassungen von dem Begriff haben. Wenn Sie einen Mitarbeiter von Microsoft fragen, wird der Ihnen sicher erzählen, daß OLE objektorientiert ist. Wenn Sie diese Informationen an einen eingefleischten C++-Spezialisten weitergeben, wird dieser möglicherweise mit den Augen rollen, weil OLE-Objekte nur Schnittstellen vererben können, aber keine Implementierung.

So ist auch in der Gruppe der Software-Architekten, die normalerweise sehr große Projekte mit objektorientierten Methoden entwickeln, die Meinung verbreitet, daß der Aspekt der Vererbung zu stark überbewertet wird. Diese Leute empfehlen ganz dringend, den Mechanismus Vererbung sehr sparsam einzusetzen. Der typische C++-Hacker wird möglicherweise die Meinung vertreten, ohne Vererbung, wenn möglich gleich Mehrfachvererbung wie in C++, ist eine Programmiersprache überhaupt nicht objektorientiert.

Wenn Sie nun hinreichend verwirrt sind, sollten Sie vielleicht doch das Kapitel zu Ende lesen. Es mag durchaus dazu beitragen, die Verwirrung aufzulösen. Das Kapitel soll das Thema OOP von einem etwas anderen Ansatz näherbringen.

Wenn Sie Kapitel in anderen Bücher zu diesem Thema gelesen haben, wird dort meist das typische Beispiel von den Wundern der Vererbung dargelegt. Oft wird eine Klasse »Tier« erzeugt von der sich dann »Vierbeiner« und »Zweibeiner« ableiten die dann zu konkreten Klassen wie »Hund«, »Katze« und »Wellensittich« führen. An diesen Beispielen läßt sich natürlich sehr einleuchtend erläutern, wie der Mechanismus der Vererbung funktioniert, doch für die praktische Programmierarbeit nützt es recht wenig, wenn Sie nicht gerade ein Zoo Ihr Auftraggeber ist.

Der folgende Ansatz sollen dennoch einige Grundbegriffe von OOP prinzipiell erläutern, für all diejenigen, denen das Tierbeispiel noch nicht so geläufig ist. Dabei werden die Vor- und Nachteile von OOP allgemein sowie die einzelnem Merkmale wie Kapselung, Vererbung und Wiederverwendung näher erläutert. Rein technische Grundlagen wie die Begriffe »Eigenschaften«, »Methoden« und »Konstruktor« werden weitgehend vorausgesetzt, das sie meist aus der Beschäftigung mit der VCL bereits kennengelernt wurden.

Nachteile von OOP

Objektorientierte Programmierung sowie alle objektorientierten Techniken sind keineswegs der ultimative Problemlöser, wie es in den ersten Jahren dargestellt wurde. Mit steigender Zahl der verwendeten Klassen nimmt die Zufriedenheit mit der Methode ab. Das liegt natürlich nicht an OOP an sich, ebensowenig wie es an der strukturierten Methode liegt, wenn große Programme mehr Programmfehler aufweisen als kleine. Im Gegenteil, erst mit OOP werden manche großen Projekte erst realisierbar.

Was also sind die typischen Fallstricke von OOP?

OOP ist auch Arbeit

Auf einen einfachen Nenner gebracht könnte man sagen, daß objektorientierte Methoden als Erleichterung der Programmierung angesehen werden. Das sind sie leider ganz und gar nicht. Einfache Lösungen, bei denen sich alle Probleme durch automatische Vererbung der gerade benötigten Eigenschaften und Methoden von selbst erledigen, gibt es nicht. Die Entwicklung eines Programms ist immer noch recht harte Arbeit. Es muß konzipiert, entworfen, erstellt, überarbeitet und getestet werden, wie es auch schon bei der strukturierten Programmierung notwendig war.

Außerdem kommt noch dazu, daß zusätzlich viele Besonderheiten der Objektorientierung mit berücksichtigt werden müssen.

Planung wird komplexer

Um die Möglichkeit eines groben Designfehlers auszuschließen, sollten objektorientierte Projekte besonders gründlich geplant werden. Fehler im Basisdesign schlagen nämlich bei OOP genauso brutal zurück, wie man es vielleicht bereits in der strukturierten Programmierung kennt. Leider kennt OOP noch eine zusätzliche Grausamkeit: Wurde der Fehler nämlich irgendwo tief unten in der Basisklasse einer hochkomplexen Klassenhierarchie gemacht, kann er nicht einfach »gepatched« werden, ohne das ganze Objektmodell in Frage zu stellen.

Designfehler in Klassen, die relativ tief in der Hierarchie liegen, müssen also möglichst an der Basis korrigiert werden, so daß alle darauf aufbauenden Klassen diese Korrektur ebenfalls mitbekommen. Das ist die schöne Welt der Vererbung, in der jeder Nachkomme einer Basisklasse jede Veränderung hautnah miterlebt. Muß jedoch eine Methode grundlegend geändert werden, muß natürlich der Programmcode für alle Klassen angepaßt werden, die diese Methode auch verwenden. Beispielsweise, wenn ein Parameter geändert werden muß.

Hierarchie wird größer

Wie schon eingangs erwähnt, wird die Technik der Vererbung oft überbeansprucht, was zu übermäßig tiefen Klassenhierarchien führt. Richtig schlimm kann es werden, wenn eine Klassenbibliothek dazugekauft wird, die bereits Klassen mit vier oder fünf Vorfahren verwendet. Wenn Sie als Entwickler nun diese Klassen Ihrerseits zu eigenen Klassen ableiten, fügen Sie automatisch eine weitere Hierarchiestufe hinzu.

Ein weiterer Nebeneffekt bei großen Klassenhierarchien ist, daß manchmal eine Funktionalität einer Klasse gar nicht bekannt ist und auch nur schwer über die Dokumentation herauszufinden ist. Durch den bequemen Mechanismus der Vererbung werden manchmal bestimmte Methoden überschrieben, um die Funktion in einer neuen Klasse abzuschalten. Wenn diese Tatsache nicht bekannt ist, wird vielleicht der nächste Entwickler (oder Sie selbst nach einigen Wochen) eine selbstgeschriebene Methode in die Klasse einfügen, die weiter unten in der Hierarchie bereits vorhanden war.

Wiederverwendung ist teuer

Oft wird behauptet, OOP sei deshalb so praktisch, weil sich einmal programmierte Objekte immer wieder verwenden lassen. Leider werden Sie in der Praxis feststellen müssen, daß Sie eben nicht die Wiederverwendung als kostenlosen Bonus dazubekommen, wenn Sie OOP verwenden. Klassen und Objekte, die tatsächlich wiederverwendet werden sollen, müssen auch für die Wiederverwendung programmiert werden.

Ein einleuchtender Grund für diese Tatsache ist, daß eine wiederverwendbare Klasse möglichst allgemein gehalten sein soll. Die Schnittstellen dieser Klasse sollen sich nur auf das Wesentliche beschränken. Die Klasse darf keinen direkten Bezug zu einer anderen Klasse haben, sonst kann sie nicht allgemein eingesetzt werden.

Vorteile von OOP

Wenn Sie die bis hierhin aufgeführten Nachteile nicht völlig abgeschreckt haben, dann wird es jetzt für Sie interessant. Werden objektorientierte Methoden nämlich sinnvoll und diszipliniert eingesetzt, ergeben sich eine ganze Reihe von Vorteilen, die OOP als gute Wahl erscheinen lassen.

Robusterer Code durch Kapselung

Objekte verstecken Ihre Daten vor der Außenwelt, wenn sie sonst niemand sehen soll. Dieses Verstecken von Informationen dient in erster Linie dazu, unberechtigte Zugriffe auf Daten durch andere Objekte zu verhindern. Jeder Programmierer kennt das Problem der Seiteneffekte, wenn er eine globale Variable in einer Funktion ändert, während eine andere Funktion noch den unveränderten Inhalt in dieser Variable erwartet. Das Ergebnis sind schwer zu lokalisierende Fehler, die aufwendig gesucht werden müssen.

Ein Objekt hingegen definiert für sich lokale Variablen, die über Zugriffsmethoden verändert oder ausgelesen werden können. Da alle Zugriffe auf die Variablen über diese Methoden erfolgen, kann das Objekt selbst kontrollieren, wann eine Variable verändert wurde. Damit sind Seiteneffekte natürlich nicht aus der Welt geschafft. Selbstverständlich ist es auch hier problematisch, wenn ein Objekt eine bestimmte Eigenschaft verändert, während ein anderes Objekt nichts davon weiß. Allerdings läßt sich hier der Fehler schneller einkreisen, beispielsweise durch einen Haltepunkt in der Zugriffsmethode.

Klarere Strukturen durch Vererbung

Unbestritten ist die Vererbung, sinnvoll und sparsam eingesetzt, eine der hilfreichsten Eigenschaften von OOP. Wird sie verwendet, um aus allgemeineren Klassen spezialisiertere Klassen abzuleiten, ist die Vererbung eine hilfreiche und nützliche Angelegenheit. Auf diese Weise können Methoden von Basisklassen ohne zusätzlichen Programmieraufwand auch von den Nachfolgern dieser Klasse verwendet werden. C++Builder macht in der VCL regen Gebrauch davon und ein eingehendes Studium der VCL-Klassenhierarchie liefert wertvolle Einsichten, wie Vererbung nutzbringend eingesetzt werden kann.

Sichere Standardmodule durch Wiederverwendung

Auch wenn Wiederverwendung zusätzliche Arbeit bedeutet, kann sie durchaus im weiteren Verlauf eines Projekts Arbeit einsparen. Auch hier ist die VCL ein gutes Beispiel. Jedes Control in der VCL ist auf Wiederverwertung ausgelegt, sonst könnten Sie die Controls nicht beliebig auf Ihren Forms plazieren.

Merkmale von OOP

Kapselung

Das Prinzip der Kapselung von Objekten wird auch manchmal als »Information Hiding«, also Verstecken von Informationen, bezeichnet. Was steckt hinter diesem Konzept und warum sollte eine Anwendung seine Informationen verstekken wollen?

Bei den versteckten Informationen handelt es sich keinesfalls um Dinge, die mit Datenschutz oder Verschlüsselung zu tun haben. Das Konzept der Kapselung sieht lediglich vor, daß bestimmte Eigenschaften und Methoden einer Klasse vor einem Zugriff von Außen geschützt werden.

Wie dieses Konzept dabei helfen kann, ein Programm robuster und weniger fehleranfällig zu machen, soll als anschauliches Beispiel eine kleine Bank-Anwendung dienen. Grundlage hierfür ist eine Konto-Klasse, die alle Informationen zu einem Kundenkonto kapselt. Im Verzeichnis »Oop\Konto1« wird eine Klasse *KontoClass* deklariert:

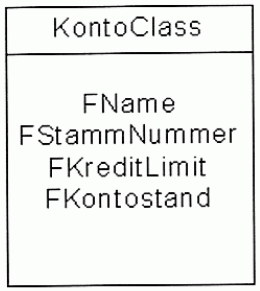

Abb. 1: Die Klasse KontoClass

```
class KontoClass
{
private:
    String FName;
    long FStammNummer;
    double FKreditLimit;
    double FKontostand;
protected:
public:
    KontoClass(long Nummer);
    void ShowInfo(TMemo *Memo);
    int SetKreditLimit(double newLimit);
    double GetKreditLimit(void);
};
```

Das dazugehörige Programm erstellt eine Instanz dieser Klasse. Der Konstruktor bekommt dabei die Stammnummer des Kontos übergeben, aus der sich dann über einen Datenbankzugriff der Kunde mit den Daten wie Kreditlimit und Kontostand auslesen läßt. Für dieses einfache Beispiel wird einfach ein Dummy-Kunde angelegt.

```
KontoClass::KontoClass(long Nummer)
{
    FStammNummer = Nummer;
    FName = "KundenName";
    FKreditLimit = 10000;
    FKontostand = 0;
}
```


Das Programm verfügt über zwei Schaltflächen, eine EditBox und ein Memo-Control. Im MemoControl werden die aktuellen Informationen über das Konto angezeigt. Dies erledigt die Methode *ShowInfo()*:

```
void KontoClass::ShowInfo(TMemo *Memo)
{
    String Zeile;
    Memo->Clear();
    Zeile = "Meine Stammnummer lautet: " + FloatToStr(FStammNummer);
    Memo->Lines->Add(Zeile);
    Zeile = "Name: " + FName;
    Memo->Lines->Add(Zeile);
    Zeile = "Kreditlimit: " + FloatToStr(FKreditLimit);
    Memo->Lines->Add(Zeile);
    Zeile = "Kontostand: " + FloatToStr(FKontostand);
    Memo->Lines->Add(Zeile);
}
```

Das sieht dann etwa folgendermaßen aus:

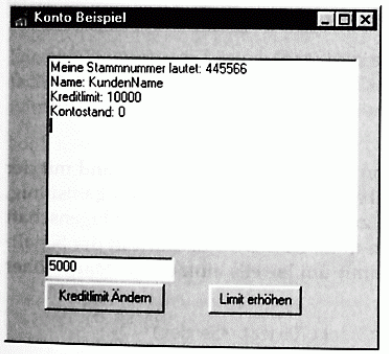

Abb. 2: Das erste Beispiel zur Konto-Klasse

Zur Änderung des Kreditlimits wäre es nun ein leichtes, die entsprechende Eigenschaft von *KontoClass* zu ändern, etwa folgendermaßen:

```
void __fastcall TfrmMain::btnChgLimitClick(TObject *Sender)
{
  Konto->FKreditLimit = StrToFloat(Edit1->Text);
}
```

Wenn Sie dies versuchen (die Zeile wurde im Beispiel nachträglich auskommentiert), werden Sie vom Compiler die Meldung erhalten: »'KontoClass::FKreditLimit' is not accessible«. Recht hat er! Sonst könnte ja schließlich jeder kommen und sein Kreditlimit erhöhen.

Damit wird deutlich, wie restriktiv gekapselte Daten sein können, war doch noch innerhalb der Klasse der Zugriff auf die einzelnen Eigenschaften problemlos möglich. Sobald jedoch eine Instanz der Klasse erzeugt worden ist, sind alle in der *private*-Sektion deklarierten Eigenschaften nicht mehr direkt lesbar oder gar veränderbar.

Um das Kreditlimit des aktuellen Kontos neu setzen zu können, wurde die entsprechende Funktion der Schaltfläche folgendermaßen umgeschrieben:

```
void __fastcall TfrmMain::btnChgLimitClick(TObject *Sender)
{
  Konto->SetKreditLimit(StrToFloat(Edit1->Text));
  Konto->ShowInfo(Memo1);
}
```

Dies entspricht in diesem Fall der direkten Zuweisung eines neuen Wertes an *FKreditLimit*, denn die Methode *SetKreditLimit* sieht folgendermaßen aus:

```
void KontoClass::SetKreditLimit(double newLimit)
{
  FKreditLimit = newLimit;
}
```

Man kann sich nun fragen: Was soll denn eigentlich der ganze Umstand mit der Kapselung? In diesem einfachen Beispiel erscheint es tatsächlich völlig unsinnig, die Variablen nicht direkt zu verändern, zumal selbst zum Lesen der Eigenschaft eine eigene Methode aufgerufen werden muß. Dies wird deutlich in der Schaltfläche »Limit erhöhen«, die das Kreditlimit um jeweils eintausend Mark höher setzen soll:

```
void __fastcall TfrmMain::btnRiseLimitClick(TObject *Sender)
{
  Konto->SetKreditLimit(Konto->GetKreditLimit() + 1000);
  Konto->ShowInfo(Memo1);
}
```

Hier werden in einer einzigen Zeile gleich zwei Methoden aufgerufen, lediglich um eine Variable um einen bestimmten Wert zu erhöhen. Tatsächlich ist genau das der Punkt, an dem viele Programmierer die Kapselung ihrer Klassen wieder aufbrechen, indem sie die Eigenschaften einfach als *public* deklarieren, so daß sie jederzeit darauf zugreifen können.

Die Stärke der Kapselung wird allerdings deutlich, wenn an bestehenden Klassen Änderungen vorgenommen werden sollen. So hat Ihr Chef vielleicht herausgefunden, daß mit der bestehenden Software jeder Sachbearbeiter ein Kreditlimit nach Belieben herauf- und heruntersetzen kann. Er entscheidet nun, dies dürfen fortan nur berechtigte Mitarbeiter tun. Die Berechtigung muß über ein Paßwort geprüft werden.

Stellen Sie sich weiterhin vor, Ihre Software verwendet die Änderung des Kreditlimits nicht nur in einem Programmteil, sondern diese Funktion ist im gesamten System mehrfach verteilt. Weiterhin angenommen, die Funktion »Limit erhöhen« ist die Funktion, die Sie zuletzt eingerichtet haben. Die Funktion »Kreditlimit ändern« wurde vor Jahren von einem Kollegen eingerichtet und Ihnen ist gar nicht klar, daß es sie im System überhaupt gibt.

Gehen Sie einmal von der üblichen Arbeitsweise aus. Sie müßten nun an allen Stellen im System, an denen irgend eine Person das Kreditlimit ändern kann, eine entsprechende Routine einfügen. Nehmen wir weiter an, Sie finden alle Stellen im System, an denen die Funktion »Limit erhöhen« aufgerufen wird, dann haben Sie nicht nur viel Arbeit mit der Änderung aller dieser Stellen, sondern Sie haben auch eine Sicherheitslücke durch die nicht entdeckte Funktion »Kreditlimit ändern«.

Die objektorientierte Lösung bietet über die Kapselung des Kreditlimits eine einfache und zugleich sichere Lösung. Die Funktion der Sicherheitsabfrage wird nämlich in »Oop\Konto2« in die Methode *SetKreditLimit()* eingefügt:

```
bool KontoClass::SetKreditLimit(double newLimit)
{
  if (GetPassword("Kreditlimit ändern"))
  {
    FKreditLimit = newLimit;
    return true;
  } else {
    return false;
    }
}
```

Hierbei wurden zwei Änderungen vorgenommen. Zunächst wird die Zuweisung des neuen Kreditlimits von der erfolgreichen Durchführung der Methode *GetPassword()* abhängig gemacht. Dann wird auch noch ein Rückgabewert an die aufrufende Funktion zurückgegeben, der erkennbar macht, ob das Kreditlimit verändert wurde oder nicht.

Gleichzeitig wurde die Methode der Schaltfläche »Limit erhöhen« so verändert, daß dieser Rückgabewert auch gleich mit ausgewertet wird:

```
void __fastcall TfrmMain::btnRiseLimitClick(TObject *Sender)
{
  if (!Konto->SetKreditLimit(Konto->GetKreditLimit() + 1000))
  {
    Application->MessageBox(
      "Kreditlimit nicht geändert",
      "Hinweis", MB_OK);
  }
    Konto->ShowInfo(Memo1);
}
```

An der eigentlichen Zeile zur Erhöhung des Kreditlimits hat sich nichts geändert. Wenn nun der Sachbearbeiter auf die Schaltfläche klickt, wird er erst nach dem Paßwort gefragt. Ist dies korrekt, wird das Limit kommentarlos um eintausend erhöht, wenn nicht, erhält er eine entsprechende Meldung.

Was passiert nun aber mit der Sicherheitslücke »Kreditlimit ändern«, die sich irgendwo im System versteckt? Versuchen Sie es einmal. In der Methode für diese Schaltfläche wurde der Code überhaupt nicht verändert, schließlich wußten Sie ja gar nicht mehr, daß er existiert. Dennoch werden Sie fortan nach dem Paßwort gefragt. Hier erscheint dann zwar keine Meldung, wenn das Paßwort falsch war, aber das Kreditlimit wird nicht mehr verändert.

Wenn nun nächste Woche Ihr Chef verlangt, das System sollte bei Änderung eines Kreditlimits auch noch Datum, Uhrzeit und den Namen des Sachbearbeiters für spätere Kontrollen aufzeichnen, können Sie sich gelassen zurücklehnen. Schließlich sind die Eigenschaften Ihrer Klassen ja perfekt gekapselt und Sie können diese Funktionalität genauso einfach einfügen wie die Paßwortabfrage.

Damit sind folgende Vorteile erreicht:

- Der Entwickler muß nicht in verschiedenen Programmen nach einem Ansatzpunkt für die Erweiterung suchen, sondern ergänzt lediglich die entsprechende Methode in einer Klasse
- Alle Anwendungen profitieren unmittelbar davon, denn die Funktionalität steht sofort allen Anwendungen zur Verfügung, die diese Klasse ebenfalls verwenden.
- Die Fehlersuche wird wesentlich erleichtert. Dadurch, daß eine bestimmte Funktionalität immer über die gleiche Methode einer Klasse realisiert wird, ist die Gefahr geringer, daß etwas übersehen wird.
- Weil keine Anwendung direkt auf die Daten zugreifen kann, ist der Entwickler gezwungen, die entsprechenden Methoden der Klasse zu verwenden.

Das Prinzip der Kapselung benötigt im Grund gar keine objektorientierte Programmiersprache. Auch in prozeduralen Sprachen ist dies möglich, wenn gewisse Regeln eingehalten werden. So waren globale Variablen immer eine beliebte Fehlerquelle, weil einige Programmteile die Variable verändert haben, während der Rest des Programms davon nichts mitbekam.

Eine gut realisierbare Lösung dieses Problems war es, globale Variablen immer über Funktionen anzusprechen. Dadurch wurden die Variablen nicht »wild« von überall geändert und die entsprechenden Zugriffsfunktionen konnten gegebenenfalls regulierend eingreifen. Dieses Vorgehen erforderte allerdings eine gewisse Disziplin vom Entwickler, denn er wurde ja nicht daran gehindert, doch hin und wieder eine globale Variable direkt zu lesen oder zu ändern.

Durch den Mechanismus der Kapselung kann der Entwickler selbst Klassen entwerfen, die es ihm unmöglich machen, auf bestimmte Daten direkt zuzugreifen. Wie eingangs schon erwähnt, muß er das allerdings nicht tun, aber er hat die Möglichkeit, es zu tun.

Vererbung

Die Vererbung genießt bei »echten« OOP-lern den Ruf, zu einer richtigen objektorientierten Sprache dazuzugehören, sonst könne man sie nicht als objektorientiert bezeichnen. Tatsächlich verbinden wohl die meisten Programmierer das Stichwort »objektorientiert« mit der Vererbung von Eigenschaften und Methoden einer Klasse. Und tatsächlich ist dies ja auch die herausragendste Eigenschaft einer objektorientierten Sprache.

Auch in den prozeduralen Sprachen konnte ein ähnlicher Mechanismus wie die Vererbung eingerichtet werden, nur eben mit größerem Aufwand und sehr viel fehleranfälliger. Dort gab es keine Objekte, sondern man mußte die Funktionalität in eine Funktion packen. Wurde eben diese Funktion an anderer Stelle mit kleinen Änderungen benötigt, so konnte man eine weitere Funktion um die vorhandene herum schreiben. Die entsprechenden Änderungen wurden dann vor oder nach dem eigentlichen Funktionsaufruf durchgeführt. Hatte man mehrere Funktionen, mußte man dies für jede dieser Funktionen durchführen. Ein ausgesprochen mühsames Verfahren.

Glücklicherweise enthalten Klassen ihre Eigenschaften und Methoden innerhalb der Klasse beziehungsweise im konkreten Objekt gekapselt, wie bereits erläutert wurde. Wird von einer bestehenden Klasse eine neue Klasse abgeleitet, erbt die neue Klasse alle öffentlichen Eigenschaften und Methoden von der Basisklasse. Auf diese Weise hat man alle Methoden der Klasse, was früher die Funktionen waren, auf einen Schlag mit einer neuen Methode umgeben. Die heißt auch noch genauso wie die ursprüngliche, bezieht sich aber nun auf die abgeleitete Klasse.

Auch hier kann durch Überschreiben der Methode entweder die Funktionalität ganz neu programmiert oder ergänzt werden.

Wer gute Beispiele für Vererbung und Ableitung sucht, sollte sich unbedingt die Quellcodes zur VCL von C++Builder ansehen. Die Hilfefunktion der IDE liefert zu jedem Control auch die dazu gehörige Hierarchie. Bei Borland wird jedes Control von der Basisklasse *TControl* abgeleitet und entsprechend erweitert, bis es seine endgültige Funktionalität erreicht hat. Der Anwender kann dann seinerseits ein Control ableiten und seinem neu erzeugen Control zusätzliche Eigenschaften geben.

Dies alles ist wahrscheinlich bereits bekannt und wurde auch schon in zahlreichen Büchern beschrieben. Doch die Vererbung läßt sich nicht nur für technische Komponenten sinnvoll einsetzen. Leider gibt es viel zu wenig Beispiele für den Ansatz bei Geschäftsobjekten, die eine objektorientierte Ansatzweise praktisch »hinter den Kulissen« der technischen Details durchführen.

Ein einfaches Beispiel in »Oop\Konto3« soll dies erläutern. Nehmen Sie an, Sie sollen für eine Bank ein EDV-System entwickeln, bei dem die Bankkonten der Kunden automatisch geführt werden sollen. Hier bietet sich die Klasse *Konto* dazu an, durch Vererbung spezialisiert zu werden.

Die Basisklasse *Konto* kennt den Namen des Kunden, seine Stammkontonummer und sein Kreditlimit. Die Bank bietet Ihren Kunden sowohl Gehalts- als auch Kreditkartenkonten an. Warum sollten Sie also für jedes dieser Konten eine eigene Klasse erzeugen? Die Klasse *Konto* bildet als Basisklasse alle Informationen, die in jedem Fall bekannt sind.

Das Gehaltskonto ist nun eine besondere Art von Konto, daher wird es von der Klasse *Konto* abgeleitet. Damit »weiß« es automatisch den Namen des Kunden, die Stammkontonummer und das Kreditlimit. Beim Gehaltskonto wird zur Stammkontonummer eine dreistellige Zahl vorangestellt, so daß es als Gehaltskonto zu erkennen ist. Das Gehaltskonto kennt außerdem eine Postengebühr für jeden Buchungsvorgang und eine Jahresgebühr für die Kontoführung.

Ein Kreditkartenkonto muß die gleichen Informationen im Zugriff haben wie jedes andere Konto, also Name, Nummer und Kreditlimit. Damit wird es auch von der Klasse *Konto* abgeleitet und hat ebenfalls eine dreistellige Zusatzzahl, das es als Kreditkartenkonto ausweist. Allerdings berechnet die Bank Ihren Kunden bei diesem Konto keine Postengebühr, sondern nur eine Jahresgebühr.

Hier die Deklaration dieser Klassen:

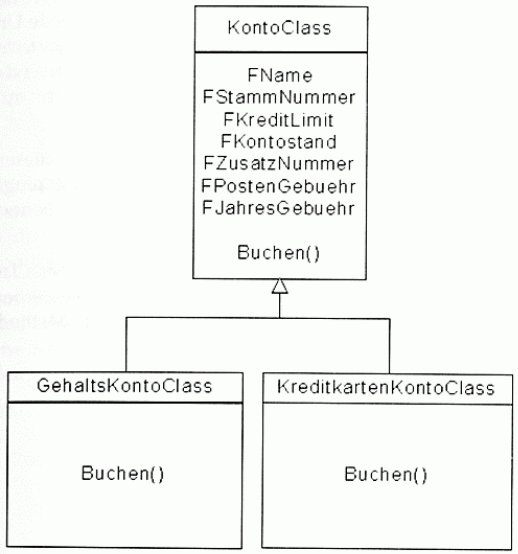

Abb. 3: Grafische Darstellung der Unterklassen von KontoClass

```
class GehaltsKontoClass : public KontoClass
{
private:
protected:
public:
  GehaltsKontoClass(long Nummer);
  void Buchen(double Betrag);
};

class KreditkartenKontoClass : public KontoClass
{
private:
protected:
public:
  KreditkartenKontoClass(long Nummer);
  void Buchen(double Betrag);
};
```

Die Felder *FZusatzNummer*, *FPostenGebuehr* und *FJahresGebuehr* wurden in der Basisklasse *KontoClass* in der Sektion *protected* hinzugefügt, statt sie für jede Unterklasse einzeln zu deklarieren. Eigenschaften und Methoden, die als *protected* deklariert sind, können von ihren abgeleiteten Unterklassen direkt verwendet werden. Die strenge Kapselung wird damit unter »direkten Verwandten« aufgehoben.

Die Kontoklasse benötigt für den Konstruktor eine Kontonummer, in diesem Fall die Stammnummer des Kontos. Aus diesen Daten kann sich das erzeugte Konto-Objekt dann beispielsweise aus einer Datenbank den zugehörigen Namen des Kontoinhabers und sein Kreditlimit auslesen.

Die Klassen *KreditkartenKontoClass* und *GehaltsKontoClass* werden von *KontoClass* abgeleitet, so daß sie alle Eigenschaften und Methoden dieser Basisklasse erben. Der Vorteil dieser Vorgehensweise wird deutlich, wenn man sich die Methode *ShowInfo* für die einzelnen Klassen näher ansieht:

```
void KontoClass::ShowInfo(TMemo *Memo)
{
  String Zeile;
  Memo->Clear();
  Zeile = "Kontonummer: " +
    IntToStr(FZusatzNummer) + " " +
    FloatToStr(FStammNummer);
  Memo->Lines->Add(Zeile);
  Zeile = "Name: " + FName;
  Memo->Lines->Add(Zeile);
  Zeile = "Kreditlimit: " + FloatToStr(FKreditLimit);
  Memo->Lines->Add(Zeile);
  Zeile = "Kontostand: " + FloatToStr(FKontostand);
  Memo->Lines->Add(Zeile);
  Zeile = "Jahresgebühr: " + FloatToStr(FJahresGebuehr);
  Memo->Lines->Add(Zeile);
}
```

Die Methode *ShowInfo* für die Basisklasse zeigt hier nur die Eigenschaften an, die für alle Konten relevant sind. Die feste Postengebühr für jeden Buchungsvorgang wird nur bei Gehaltskonten erhoben und braucht also bei anderen Konten nicht angezeigt zu werden. Daher wird diese Zeile durch Überschreiben der Methode *ShowInfo* in *GehaltsKontoClass* zusätzlich angezeigt:


```
void GehaltsKontoClass::ShowInfo(TMemo *Memo)
{
  String Zeile;
  KontoClass::ShowInfo(Memo);
  Zeile = "Postengebühr: " + FloatToStr(FPostenGebuehr);
  Memo->Lines->Add(Zeile);
}
```

Beim Gehaltskonto sollen zunächst die Werte für das Konto allgemein angezeigt werden. Hierfür wird einfach die Methode der Basisklasse *KontoClass::ShowInfo()* explizit aufgerufen. Hat diese Ihre Aufgabe erledigt, wird die Zeile im Memo-Control um die Zusatzinformationen für die Postengebühr ergänzt.

Um überhaupt ein Objekt der Klasse *GehaltsKontoClass* erzeugen zu können, muß dessen Konstruktor natürlich wissen, daß auch die Basisklasse *KontoClass* eine Kontonummer benötigt. Aus diesem Grund muß der Konstruktor so implementiert werden:

```
GehaltsKontoClass::GehaltsKontoClass(long Nummer): KontoClass(Nummer)
{
  FZusatzNummer = 170;
  FPostenGebuehr = 0.5;
  FJahresGebuehr = 60;
}
```

Damit weiß der Compiler, daß ein Objekt der Klasse *GehaltsKontoClass* als Konstruktor die Klasse *KontoClass* mit dem Parameter *Nummer* aufrufen muß. Wird ein solches Objekt in dieser Weise erzeugt, sind alle Werte, die zur Basisklasse gehören, bereits initialisiert. Ein Objekt der Klasse *GehaltsKontoClass* weiß also automatisch den Namen des Kontoinhabers und dessen Kreditlimit. Der Konstruktor muß lediglich die Eigenschaften setzen, die ausschließlich zum Gehaltskonto gehören.

In der Praxis würde man hier natürlich die Werte für die einzelnen Gebühren aus einer Datenbank einlesen, da sich diese Werte im Laufe der Zeit ändern können. Sie werden hier als konstante Werte eingetragen, um das Beispiel einfach zu halten. Ebenso verhält es sich mit dem Konstruktor der Basisklasse *KontoClass*:

```
KontoClass::KontoClass(long Nummer)
{
  FStammNummer = Nummer;
  FName = "KundenName";
  FKreditLimit = 10000;
}
```

Auch hier wäre selbstverständlich eine Routine nötig, die aus der übergebenen Kontonummer den passenden Kunden heraussucht und die Daten für Name und Kreditlimit entsprechend einträgt.

Vererbung wird üblicherweise dort verwendet, wo die meisten Eigenschaften und Methoden der Basisklasse nicht verändert werden müssen. Lediglich die Methoden, die ein abweichendes Verhalten zeigen sollen, müssen in der abgeleiteten Klasse überschrieben werden, wie dies im Beispiel bei der Methode *ShowInfo* geschehen ist.

Ein weiteres Beispiel bietet sich bei der Behandlung der Umsätze an. Im Bankengewerbe ist es ja üblich, die Umsätze von Kreditkarten über ein Bonitätsprüfung laufen zu lassen. Dies geschieht üblicherweise beim Händler, der dies entweder über ein Online-Gerät durchführt, oder bei der Kreditkartengesellschaft anruft und sich eine Genehmigungsnummer geben läßt. Üblicherweise wird diese nur erteilt, wenn die Karte nicht als gestohlen gemeldet wurde und das Kreditlimit noch nicht überschritten ist.

Eine solche Situation bildet die Methode *Buchen()* nach, die bei einem Kreditkarten-Umsatz zunächst die Methode *Bonitaet()* aufruft. Erhält die Buchungsmethode eine positive Rückmeldung, wird der Umsatz abgebucht, andernfalls bleibt der Kontostand erhalten.

```
void KreditkartenKontoClass::Buchen(double Betrag)
{
  if (Bonitaet(Betrag))
    KontoClass::Buchen(Betrag);
}

bool KreditkartenKontoClass::Bonitaet(double Betrag)
{
  if (-(FKontostand + Betrag) <= FKreditLimit)
  {
    Application->MessageBox(
      "Danke für Ihren Auftrag. Umsatz ist genehmigt",
      "Info", MB_OK);
    return true;
  } else {
    Application->MessageBox(
      "Leider kann der Umsatz nicht genehmigt werden",
      "Info", MB_OK);
    return false;
  }
}
```


Im Gegensatz hierzu werden Buchungen auf dem Gehaltskonto auch ohne eine solche Prüfung durchgeführt. Wie es bei den Banken üblich ist, kann auch ein vereinbarter Überziehungskredit problemlos überschritten werden, da die Banken dann besonders hohe Zinsen kassieren können.

Aggregation

Wie Sie nun wissen, können Klassen wiederum andere Klassen erzeugen und verwenden, wie im Beispiel mit der Paßwortabfrage, wo der Paßwort-Dialog eine Ableitung von *TForm* darstellt:

```
bool KontoClass::GetPassword(String Label)
{
  TfrmPassword *pass;
  pass = new TfrmPassword(NULL);
  return pass->GetPassword(Label);
}
```

Andererseits können Klassen wieder Unterklassen erzeugen, die Ihre Eigenschaften und Methoden von der Basisklasse erben.

Im ersten Fall findet keine Kopplung zwischen den einzelnen Klassen statt, denn das Paßwort-Formular wird bei Bedarf erzeugt und anschließend wieder verworfen. Der zweite Fall stellt ein direktes Verwandschaftsverhältnis zwischen Klassen dar, die sich in einem Eltern-Kind-Verhältnis darstellt. Dennoch gibt es auch hier keine direkte Verbindung zwischen Klassen, denn eine Instanz von *GehaltsKontoClass* erzeugt zwar die geerben Eigenschaften und Methoden von *KontoClass*, aber *KontoClass* tritt nicht als eigenständige Klasse in Erscheinung. Das Gehaltskonto ist eben ein eigenes Objekt für sich.

Die objektorientierte Programmierung lebt allerdings hauptsächlich von der Interaktion zwischen eigenständigen Objekten. Wer Anwendungen in C++Builder entwickelt, hat ständig damit zu tun. Jedes Control, das auf eine Form plaziert wird, ist ein eigenständiges Objekt, das mit der Form in Beziehung steht. Eine solche Beziehung wird auch »Aggregation« genannt.

Sie bedeutet, daß ein Objekt mindestens ein oder mehrere andere Objekte »besitzt«, wie im Fall der Form, die verschiedene Controls beherbergt. Wird die Form gelöscht, verschwinden auch die darauf vorhandenen Controls. Von der anderen Seite betrachtet sind die auf der Form plazierten Controls »Teil von« der Form, auf die sie plaziert wurden. Sie gehören zur Form und sind von deren Lebensdauer abhängig.

In der klassischen Literatur über OOP wird Aggregation meist mit Beispielen von zusammengesetzten Objekten aus dem täglichen Bereich beschrieben. So

hat ein Objekt wie ein Auto die Objekte Motor, Räder, Bremsen und so weiter. In dem Beispiel »Oop\Beweg1« wird wieder das vorhandene Bank-Modell verwendet, um die abhängige Klasse Kontobewegung zu definieren:

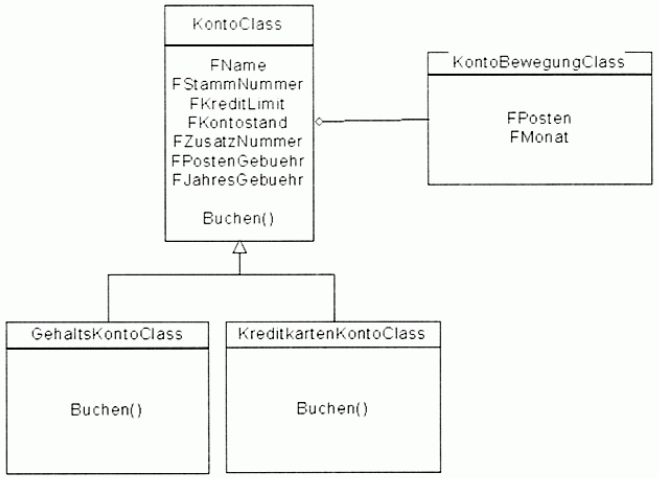

Abb. 4: Einführung einer Klasse für Kontobewegungen

Die Klasse *KontoBewegungClass* soll dafür sorgen, daß jede Buchung des Kontos als Posten gezählt wird, um am Monatsende die Gebühren für das Konto abrechnen zu können. Eine Trennung in gesonderte Klassen ist in der objektorientierten Ansatzweise durchaus üblich und sinnvoll, um eine Klasse wie die Kontoklasse nicht mit zu vielen Aufgaben zu belasten. Dies würde bestenfalls die Übersicht erschweren, schlimmstenfalls könnte es zu Mamutklassen führen, die praktisch nicht mehr vernünftig zu ändern sind.

Die Verbindung zwischen den beiden Klassen erfolgt über einen Zeiger. Da die Kontoklasse zuerst in Erscheinung tritt, ist es ihre Aufgabe, ein Objekt der Klasse *Kontobewegung* zu erzeugen. Dies geschieht im Konstruktor der Kontoklasse:

```
FBewegung = new KontoBewegungClass(this);
```

Die Eigenschaft *FBewegung* wird in der Sektion *protected* als Zeiger auf *KontoBewegungClass* deklariert. Der Konstruktor der Kontobewegung erhält außerdem einen Zeiger auf das Objekt, das ihn aufgerufen hat. Dieser Zeiger heißt *this* und

stellt in diesem Fall einen Zeiger auf das Konto-Objekt dar. Im Konstruktor der Klasse *Kontobewegung* wird dieser Zeiger wiederum einer Eigenschaft *FKonto* zugewiesen. So kann *KontoBewegungClass* auch Methoden von *KontoClass* aufrufen, so daß in diesem Fall eine Kommunikation der beiden Klassen in zwei Richtungen möglich wird.

```
KontoBewegungClass::KontoBewegungClass(KontoClass *Konto)
{
  FKonto = Konto;
  FPosten = 0;
  FMonat = 1;
}
```

Ein weiteres Merkmal der objektorientierten Vorgehensweise ist es, daß jedes Objekt soviel Verantwortung für seine Tätigkeiten übernimmt wie möglich, und nur so wenig über seine verbundenen Objekte weiß, wie nötig. Daher wird die Verantwortung für die Anzeige der Kontobewegungen an die neue Klasse delegiert. Dies geschieht, indem die folgende Zeile in *KontoClass::ShowInfo()* hinzugefügt wird:

```
FBewegung->ShowInfo(Memo);
```

Statt also die Angaben über den laufenden Monat und die Anzahl der Posten direkt in der Kontoklasse anzuzeigen, wird eine Methode *ShowInfo()* der Klasse *KontoBewegungClass* aufgerufen. Diese Methode muß übrigens nicht ShowInfo heißen, sie kann auch einen beliebigen anderen Namen haben. Ein ähnliches Verfahren haben Sie bereits beim Überschreiben von geerbten Methoden kennengelernt. Auch hier war es unter anderem die Methode *ShowInfo()*, die bei der Gehaltskonto-Klasse überschrieben wurde.

Damit ergibt sich an dieser Stelle auch eine gute Gelegenheit, den Begriff der Nachrichten zu erläutern. In den Lehrbüchern über OOP ist häufig davon die Rede, daß sich Objekte gegenseitig Nachrichten schicken. Der Aufruf einer Methode wie *ShowInfo()* entspricht in dem Beispiel einer Nachricht an das entsprechende Objekt. Das Konto-Objekt fordert damit sein zugehöriges Kontobewegung-Objekt auf, die ihn betreffenden Informationen auszugeben. Als zusätzliche Information dieser Nachricht wird ein Zeiger auf das Memo-Control mitgegeben, in das die Informationen eingetragen werden sollen.

Und so sieht das Formular dann aus:

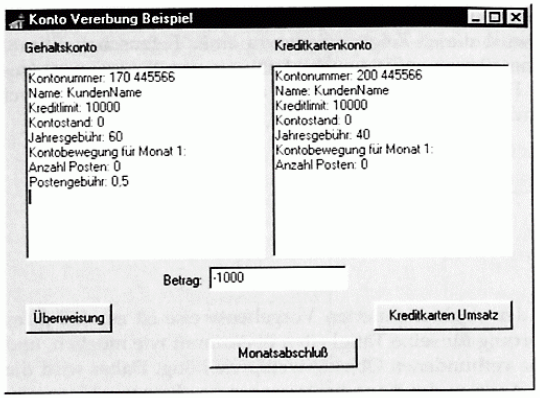

Abb. 5: Das Programm nach Zufügen einer Kontobewegung-Klasse

An dieser Stelle fällt auch schon eine Schwäche dieser Konstruktion auf. Die Daten werden nicht so angezeigt, wie man es eigentlich erwarten würde. Die Postengebühr, die ja durch Überschreiben von *ShowInfo()* in der Klasse *GehaltsKontoClass* ausgegeben wird, erscheint hier als letzter Eintrag in der Liste. Die Angaben über die Kontobewegung erscheinen zwischen den allgemeinen Kontoangaben aus *KontoClass* und der speziellen Zeile aus *GehaltsKontoClass*.

Diese Reihenfolge wäre ungünstig, wenn Sie die Angaben über Kontobewegung immer am Schluß stehen haben wollten. Von der Programmlogik ist diese Reihenfolge allerdings leicht zu erklären:

Zunächst wird ja vom Formular selbst die Kontoklasse aufgefordert, Ihre Informationen anzuzeigen. Beim Gehaltskonto wird so die Methode *GehaltsKontoClass::ShowInfo()* aufgerufen. Diese wiederum fordert mit *KontoClass::ShowInfo()* Ihre Basisklasse auf, erst einmal deren Informationen anzuzeigen.

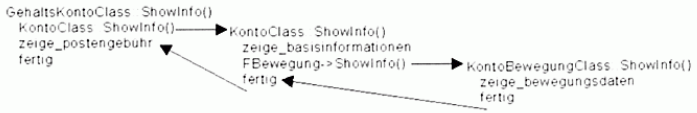

Abb. 6: Aufrufreihenfolge bei ShowInfo

Wäre die Reihenfolge der Anzeige in diesem Fall besonders wichtig, könnte man beispielsweise den Aufruf von *FBewegung->ShowInfo()* aus der Basisklasse generell in die abgeleiteten Klassen verschieben. Das hätte dann wieder den Nachteil, daß man bei jeder abgeleiteten Klasse daran denken muß, die Bewegungsdaten explizit in *ShowInfo()* mit anzuzeigen.

Als neue Aufgabe für den Buchungsvorgang des Kontos kommt nun hinzu, das Kontobewegung-Objekt von jeder Buchung in Kenntnis zu setzen. Dies geschieht in der Methode *Buchen()*, nachdem der neue Kontostand ermittelt wurde:

```
void KontoClass::Buchen(double Betrag)
{
  FKontostand = FKontostand + Betrag;
  FBewegung->Buchen(Betrag);
}
```

Auch hier könnte die Methode einen beliebigen Namen haben. Der Aufruf *FBewegug->Buchen()* ruft auch hier wieder, wie bei *ShowInfo()*, die Buchen-Methode von der Klasse auf, auf die der Zeiger *FBewegung* zeigt. In diesem Fall ist es *KontoBewegungClass*.

```
void KontoBewegungClass::Buchen(double Betrag)
{
  FPosten++;
}
```

Die Übergabe des zu buchenden Betrags ist hier übrigens völlig unnötig, da er nicht benötigt wird. Im nächsten Schritt wird die Klasse um zusätzliche Funktionalität erweitert, so daß hier der Parameter schon mal gleich mit deklariert wurde. Im Augenblick wird einfach nur die Anzahl der Posten um Eins erhöht.

Eine weitere Neuerung in diesem Beispiel ist nun die Einführung einer Schaltfläche »Monatsabschluß«. Durch Anklicken werden die einzelnen Konten aufgefordert, den aktuellen Monat abzuschließen. Was dies genau bedeutet, geht aus dem Aufruf nicht hervor. Statt dessen wird wieder die Verantwortung direkt an das Objekt abgegeben:

```
void __fastcall TfrmMain::btnMonatsAbschlClick(TObject *Sender)
{
  GehaltsKonto->Monatsabschluss();
  KreditkartenKonto->Monatsabschluss();
  Refresh();
}
```

Die Kontoklasse selbst regelt den Monatsabschluß dann folgendermaßen:

```
void KontoClass::Monatsabschluss(void)
{
  double Summe;
  String Zeile;
  Summe = FBewegung->GetPosten() * FPostenGebuehr;
  frmInfo->Memo1->Clear();
  frmInfo->Memo1->Lines->Add("Gebühren für Monat " +
    IntToStr(FBewegung->GetMonat()) + " bei " +
    IntToStr(FBewegung->GetPosten()) + " Posten = " +
    FloatToStr(Summe));
  frmInfo->ShowModal();
  Buchen(-Summe);
  FBewegung->ResetMonat();
}
```

Die Daten werden entsprechend aus der Kontobewegung-Klasse ausgelesen und es wird daraus eine Summe der Gebühren errechnet. Diese Rechnung wird in einem Infofenster angezeigt und anschließend vom aktuellen Kontostand abgebucht. Um das Objekt *KontoBewegungClass* zurückzusetzen, wurde eine Methode *ResetMonat()* erstellt:

```
void KontoBewegungClass::ResetMonat(void)
{
  FPosten = 0;
  FMonat++ ;
  if (FMonat == 13)
  {
    FMonat = 1;
  }
}
```

ResetMonat() hat die Aufgabe, die Posten der Kontobewegung auf Null zu setzen, denn sie sind ja bereits abgerechnet. Weiterhin wird der aktuelle Abrechnungsmonat um Eins erhöht. Am Ende des Jahres wird aus dem Monat 12 wieder der Monat 1.

Wiederverwendung

Um Klassen möglichst überall wiederverwenden zu können, sollten sie mit möglichst wenig anderen Klassen in direkter Beziehung stehen. Der Grund dafür ist prinzipiell einleuchtend: Je weniger eine Klasse von Ihrer Umgebung weiß, desto universeller ist sie verwendbar. Klassen, die eine enge Kopplung enthalten, wie im obigen Beispiel die Klassen *KontoClass* und *KontoBewegung-*

Class, sind nur zusammen einsetzbar, denn jede Klasse ist auf das Vorhandensein der jeweils anderen Klasse angewiesen.

Eine Klasse wie *frmInfo* hat hingegen keine enge Kopplung zu der Klasse, von der sie aufgerufen wird. *FrmInfo* stellt in diesem Fall eine visuelle Klasse, nämlich eine Ableitung von *TForm* dar. Ihr Zweck ist es, mehrere Zeilen von Informationen darzustellen und anzuzeigen.

Bei der Entwicklung von wiederverwertbaren Klassen kann man prinzipiell zwei Ansätze verfolgen:

- Klassen, die möglichst viel Funktionalität aufweisen und die mit wenigen komplexen Methoden zu verwenden sind. Ein Beispiel für eine solche Klasse ist die Passworteingabe im Beispiel.
- Klassen, die möglichst allgemeingültig zu verwenden sind, daher wenig spezifische Funktionalität aufweisen und mit mehreren einfachen Methoden zu verwenden sind. Ein Beispiel für eine solche Klasse ist das Info-Fenster zur Anzeige von Textinformationen.

Während die komplexen Klassen möglichst viel wissen müssen, sollen allgemein verwendbare Klassen nur sehr wenig Grundinformationen enthalten. Bei der Paßworteingabe muß die Klasse wissen, wie ein Paßwort abzufragen ist, wie das Paßwort lautet und wie sich die Abfrage auf dem Bildschirm darstellen soll. All diese Informationen sind in der Klasse *frmPassword* enthalten. Sie hat einen vorgegebenen Anzeigetext, der noch um ein Label ergänzt werden kann. Als Rückgabewert wird nur *true* oder *false* geliefert. Der Aufruf erfolgt denkbar einfach mit der Anweisung:

```
frmPassword->GetPassword(Label);
```

Dies ist dann auch die einzige Methode, die für diese Klasse verwendet werden kann.

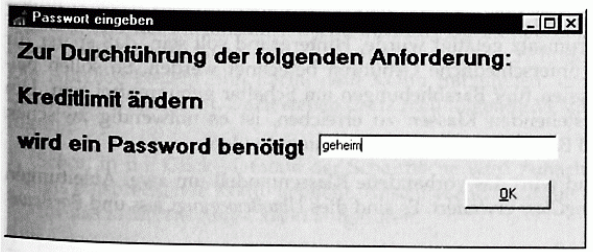

Abb. 7: Passwort-Formular als komplexe wiederverwendbare Klasse

Bei dem Info-Formular hingegen sind die Schnittstellen völlig offen. Es handelt sich ja eigentlich um ein ganz gewöhnliches Formular mit einem Memo-Control. Soll diese Klasse wiederverwendet werden, kann sie ganz universell eingesetzt werden. Der Nachteil beim universellen Einsatz ist, daß der Anwender zur Verwendung dieser Klasse umso mehr über die Klasse wissen muß, je flexibler er sie einsetzen will. So kann bei *frmInfo* der ausgegebene Text beliebig gewählt werden.

Nachdem die gewünschten Zeilen über *Memo1->Add()* angefügt wurden, wird das Formular mit *Show()* oder *ShowModal()* angezeigt. Da die Klasse in diesem Beispiel keine Rückmeldung liefern muß, braucht sie durch Anklicken der Schaltfläche OK lediglich vom Bildschirm entfernt zu werden.

Damit läßt sich folgender Grundsatz für die Wiederverwendung von Klassen ableiten:

Je allgemeingültiger eine Klasse für die Wiederverwendung gestaltet wird, desto mehr Dokumentation muß zu dieser Klasse geliefert werden.

Beispiel Paßwortabfrage: »Erzeugen Sie ein Objekt der Klasse *frmPassword* und rufen Sie die Methode *GetPassword(Label)* auf. *Label* enthält den Grund, wozu das Paßwort gebraucht wird. Gibt die Methode *true* zurück, war das Paßwort korrekt«.

Das ist ein Beispiel für eine recht knappe Dokumentation, die aber alles Notwendige enthält. Im Vergleich dazu würde eine Beschreibung der Klasse *frmInfo* entsprechend umfangreicher ausfallen.

Auswahl über Klassen

Das Beispiel der Bankanwendung soll um eine weitere Anforderung ergänzt werden. Das Beispiel selbst finden Sie in »Oop\Beweg2«.

Wir wollen bei unserer Musterbank unterscheiden, ob ein Umsatz per Überweisung oder als Barumsatz getätigt wurde. Hintergrund soll sein, daß später für die Abrechnung unterschiedliche Gebühren berechnet werden. So sollen beispielsweise die ersten fünf Barabhebungen am Schalter gebührenfrei sein. Um dies über die bestehenden Klassen zu erreichen, ist es notwendig zwischen Überweisung und Barumsatz prinzipiell zu unterscheiden.

Aus diesem Grund wird das vorhandene Klassenmodell um zwei Ableitungen von *KontoBewegungClass* erweitert. Es sind dies *UbwBewegungClass* und *BarBewegungClass*:

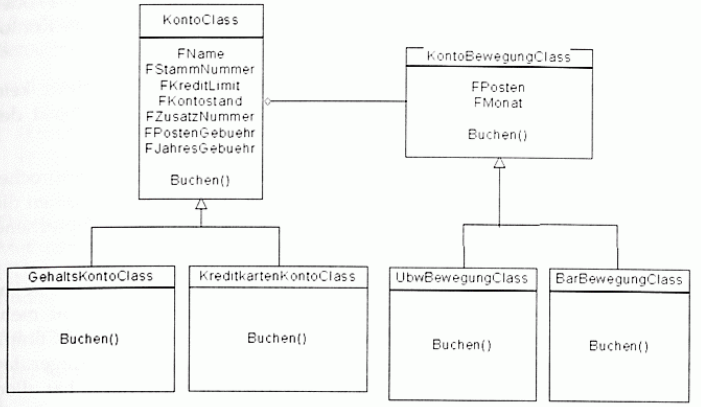

Abb. 8: Klassenmodell erweitert um zwei spezialisierte Kontobewegung-Klassen

Auch das Formular des Beispielsprogramms wurde um zwei Schaltflächen erweitert. Der Anwender kann nun zwischen vier Möglichkeiten wählen:

- Überweisung auf/von Gehaltskonto
- Ein/Auszahlung von Bargeld auf/von Gehaltskonto
- Überweisung auf Kreditkartenkonto oder Einkauf mit Kreditkarte
- Bareinzahlung oder Auszahlung von Kreditkartenkonto am Schalter

Hinweis: Barauszahlung am Geldautomaten wird später gesondert behandelt.

Damit ergeben sich für jedes Kontoobjekt zwei Situationen, nämlich die Überweisung und der Barumsatz. Da jede Buchung direkt über das jeweilige Objekt erfolgt, wird die Buchung damit automatisch auf dem richtigen Konto gebucht. Die Aufgabe besteht nun darin, dem Konto mitzuteilen, um welche Art von Umsatz es sich handelt. Dazu stehen prinzipiell folgende Möglichkeiten zur Verfügung:

1. Schon in der Click-Methode der Schaltfläche wird zunächst der Betrag auf das entsprechende Konto gebucht. Anschließend wird die Methode *Buchen()* für das *Kontobewegung*-Objekt aufgerufen.
2. Beim Aufruf der Methode *Buchen()* wird als zusätzlicher Parameter ein Kennzeichen mit übergeben, so daß die Kontoklasse weiß, um welche Art

Umsatz es sich handelt. Diese prüft dann das Kennzeichen über *if-else-* oder *switch*-Anweisung ab und ruft die Methode *Buchen()* des jeweiligen Kontobewegung-Objekts auf.

3. Beim Aufruf der Methode *Buchen()* wird als zusätzlicher Parameter kein Kennzeichen für die Umsatzart übergeben, sondern direkt ein Objekt der Klasse *Kontobewegung*.

Fall 1 entspricht der prozeduralen Vorgehensweise mit allen schon besprochenen Nachteilen. Wird ein Teil des Systems auf diese Weise erweitert, gehen die Änderungen an anderen Programmteilen spurlos vorüber, was entsprechend schwerwiegende Fehler in der Kontoabrechnung zur Folge hätte.

Fall 2 wäre eine wesentlich bessere Lösung. Durch Verwendung eines neuen Parameters für die Methode *Buchen()* können ältere Programmteile nicht mehr übersehen werden. Bei der Kompilierung des Gesamtsystems würde der Compiler nämlich erkennen, wenn die Methode mit zuwenig Parameter aufgerufen wird. Die Lösung, diesen Parameter als Kennzeichen zu verwenden, hat allerdings den Nachteil, daß die Methode *Buchen()* dieses Kennzeichen über eine *if*-Abfrage entsprechend zuordnen muß. Fehler können hier auftreten, wenn ein Kennzeichen verwendet wird, das von der Methode *Buchen()* nicht erkannt wird.

Fall 3 entspricht einem sogenannten Strategiemuster, das direkt beim Aufruf der Methode *Buchen()* auch die zugehörige Klasse *KontoBewegungClass* direkt als Objekt mit übergibt. Im Gegensatz zum Fall 2 kann hier kein ungültiges Kennzeichen übergeben werden, da man ein nicht existierendes Objekt nicht übergeben kann. Die Methode *Buchen()* der Kontoklasse braucht keine *if*-Abfragen durchzuführen, sondern kann direkt die passende Methode des übergebenen Objekts aufrufen.

Beispiel:

Der folgende Code wird ausgeführt, wenn die Schaltfläche »Überweisung« bei Gehaltskonto angeklickt wird:

```
void __fastcall TfrmMain::btnGehaltClick(TObject *Sender)
{
  double Betrag = StrToFloat(Edit1->Text);
  GehaltsKonto->Buchen(
    Betrag, GehaltsKonto->GetUbwBewegung());
  Refresh();
}
```


Der zweite Parameter von *GehaltsKonto->Buchen()* übergibt ein Objekt der Klasse *UbwBewegungClass*. Die Methode *Buchen()* der Kontoklasse wurde folgendermaßen erweitert:

```
void KontoClass::Buchen(double Betrag, KontoBewegungClass *beweg)
{
  FKontostand = FKontostand + Betrag;
  if (beweg != NULL) beweg->Buchen(Betrag);
}
```

Der Kontostand wird auf den neuesten Stand gebracht und anschließend die Methode *Buchen(Betrag)* der Klasse Kontobewegung ausgeführt. Die Abfrage auf NULL ermöglicht übrigens die Buchung, ohne daß eine Kontobewegung stattfindet.

An der Methode *Buchen()* der Kontobewegung hat sich nichts geändert:

```
void KontoBewegungClass::Buchen(double Betrag)
{
  FPosten++;
}
```

Anbindung und Anpassung von weiteren Klassen

Um zu zeigen, wie sich weitere Klassen an das bestehende Modell anbinden lassen, soll nun auch die Auszahlung am Geldautomaten ermöglicht werden. Hierzu wird eine eigene Klasse in Form eines Formulars *TfrmGeldautomat* erstellt, mit deren Hilfe die Geheimnummer abgefragt wird.

Um das Programm für die Zukunft leicht an neue Änderungen anpassen zu können, sollten vorhandene Klassen, wie zum Beispiel zugekaufte Klassenbibliotheken, möglichst nicht direkt eingebunden werden. Statt dessen ist es häufig sinnvoll, eine zusätzliche Klasse dazwischenzuschalten. Eine solche Zwischenstufe wird je nach Aufgabe als Brücke oder als Adapter bezeichnet. Sie hat lediglich die Aufgabe, die beiden Programmteile voneinander zu trennen. Dies hat folgende Vorteile:

Durch eine Trennung von Klienten-Programm und Programmbibliothek kann die konkrete Implementierung der Bibliothek von den Schnittstellen des Programms strikt getrennt werden. Wird die Programmbibliothek später einmal getrennt verändert, braucht das Klienten-Programm nicht noch einmal compiliert zu werden. Es muß lediglich über den Linker mit der neuen Bibliothek gebunden werden.

Weiterhin ist es möglich, Anpassungen an den Schnittstellen vorzunehmen, ohne diese im eigentlichen Programm durchführen zu müssen. Im vorliegenden Beispiel wird die Auszahlung am Geldautomaten über die Methode

```
Adapter->Auszahlen(KreditkartenKonto, Betrag)
```

ausgeführt. Der hier verwendete Geldautomat soll aber ein internationaler Geldautomat sein. Zu seiner Verwendung muß also noch eine Währung mit angegeben werden. Die Methode des Geldautomats sieht damit so aus:

```
bool Auszahlen(KontoClass *Konto, double Betrag, String Waehrung);
```

Um diesen Unterschied der beiden Schnittstellen auszugleichen, könnte man direkt beim Aufruf die Währung »DM« mit übergeben. Damit müßten alle entsprechenden Stellen im System mit diesem Parameter versehen werden. Die Klasse AdapterClass erspart Ihnen diese Mühe, indem Sie die Schnittstelle des Klienten an die Schnittstelle der Bibliothek, hier des Bankautomaten, anpaßt:

```
bool AdapterClass::Auszahlen(KontoClass *Konto, double Betrag)
{
  return frmGeldautomat->Auszahlen(Konto, Betrag, "DM");
}
```

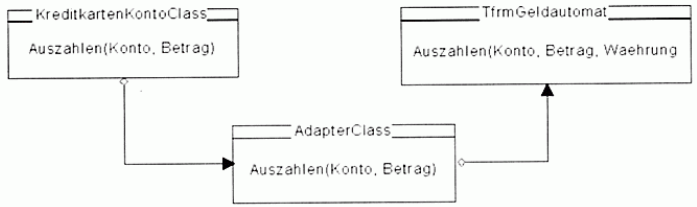

Abb. 9: Anpassung von Schnittstellen zwischen zwei Klassen

Hatte die Bank also vorher immer mit deutschen Geldautomaten gearbeitet und den Zugriff über *AdapterClass* durchgeführt, sind mit der Einführung der internationalen Automaten keine großen Änderungen im Code mehr nötig. Dies gilt natürlich nur, solange der Kunde keine Fremdwährungen abheben will.

Mehrstufige Interaktionen

Das Formular des Geldautomaten liefert selbst ein gutes Beispiel, wie Eingabesequenzen in objektorientierten Formularen durchgeführt werden sollten. Das Programm des Geldautomaten sieht folgende Vorgänge vor:

Hinweis: Es wird hier davon ausgegangen, daß bei Aufruf der Methode *Auszahlen()* die rein mechanischen Prozeduren wie das Einfügen der Keditkarte oder ec-Karte bereits erfolgt sind.

1. Geheimnummer (PIN-Nummer) abfragen.
2. Wenn Geheimnummer korrekt, dann Geld auszahlen. Ein Hinweis erscheint, daß das Geld ausgezahlt wird.
3. Wenn Geheimnummer falsch, dann kein Geld auszahlen. Ein Hinweis erscheint, daß die Auszahlung nicht möglich ist.
4. Die Karte entnehmen. Wird hier durch eine Schaltfläche »Beenden« simuliert.

Das Formular muß also diese Schritte in der Reihenfolge durchführen. Um die Sache einfacher zu machen, wird angenommen, daß der Vorgang durch Eingabe der falschen Geheimzahl abgebrochen wird. Die Schaltfläche auf dem Formular soll gleichzeitig zum Bestätigen der Geheimzahl und anschließend zum Schließen des Formulars dienen.

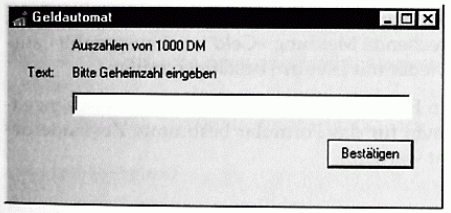

Abb. 10: Bankautomat vor der Eingabe der Geheimnummer

Nachdem die Geheimzahl eingegeben wurde, muß diese durch Anklicken von »Bestätigen« bestätigt werden. Das Formular entscheidet dann, ob sie korrekt war oder nicht. War die Nummer richtig angegeben, wird das Geld ausgezahlt:

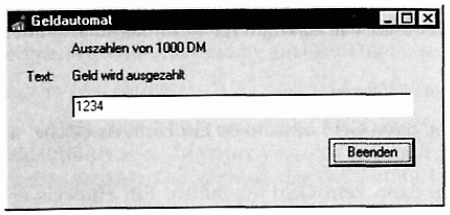

Abb. 11: Bankautomat nach der Eingabe der Geheimnummer

Die Beschriftung der Schaltfläche hat sich von »Bestätigen« auf »Beenden« verändert. Durch Anklicken der gleichen Schaltfläche wird das Formular geschlossen.

Da es sich hier um ein objektorientiertes Formular handelt, ist ein Vorgehen nach der üblichen Prozedurmethode nicht möglich. In der üblichen linearen Programmierung würde man eine Dialogbox zeigen mit der Aufforderung: »Bitte Geheimzahl eingeben«. Nach der Eingabe der Zahl und Bestätigung mit der [Return]-Taste würde die Dialogbox verschwinden. Eine neue Dialogbox würde erscheinen und die entsprechende Meldung »Geld wird ausgezahlt« ausgeben, die der Anwender dann wieder mit [Return] bestätigen müßte.

Um den gleichen Effekt mit einem Formular zu erreichen, ohne es zwischenzeitlich schließen zu müssen, kann man für das Formular bestimmte Zustände definieren, die nacheinander erreicht werden.

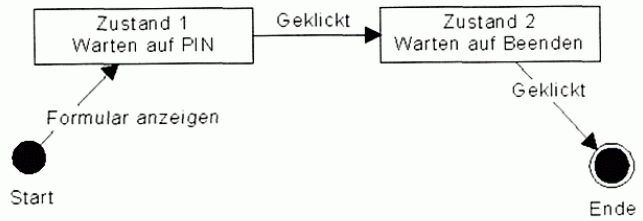

Abb. 12: Zustandsdiagramm der Schaltfläche

Das Bild zeigt die möglichen Zustände der Schaltfläche auf dem Formular. Normalerweise wird für eine Schaltfläche nur ein Zustand verwendet, sie wartet nämlich immer darauf, daß sie angeklickt wird. Soll die gleiche Schaltfläche

mehrere Funktionen übernehmen, wie in diesem Fall, so ist dies nur über unterschiedliche Zustände möglich. Wird das Formular zum ersten mal über die Methode *Auszahlen()* aktiviert, wird Zustand 1 eingerichtet:

```
bool TfrmGeldautomat::Auszahlen(
  KontoClass *Konto, double Betrag, String Waehrung)
{
  FStatus = 1;
  Label2->Caption = "Bitte Geheimzahl eingeben";
  Label3->Caption = "Auszahlen von " +
    FloatToStr(-Betrag) + " " +
      Waehrung;
  btnOK->Caption = "Bestätigen";
  ShowModal();
  return FFreigabe;
}
```

Vor der eigentlichen Anzeige werden so die Texte eingerichtet, die angezeigt werden sollen. Die Beschriftung der Schaltfläche wird auf »Bestätigen« eingestellt. Der aktuelle Status wird hier als Eigenschaft der Form in *FStatus* als Eins gespeichert.

Wird die Schaltfläche dann angeklickt, reagiert sie entsprechend dem eingestellten Status:

```
void __fastcall TfrmGeldautomat::btnOKClick(TObject *Sender)
{
  switch (FStatus)
  {
    case 1:
      if (Edit1->Text == "1234")
        FFreigabe = true;
      else
        FFreigabe = false;
      Freigabe();
      break;
    case 2:
      Close();
      break;
  }
}
```

Über die switch-Anweisung wird festgestellt, welcher Code ausgeführt werden soll. Bei Status 1 wird also die Überprüfung der Geheimzahl vorgenommen. Anschließend wird die Methode *Freigabe()* aufgerufen, um das angeforderte Geld

freizugeben und die entsprechende Meldung auszugeben. Anschließend wird die *Click*-Methode der Schaltfläche wieder verlassen.

```
void TfrmGeldautomat::Freigabe()
{
  if (FFreigabe)
    Label2->Caption = "Geld wird ausgezahlt";
  else
    Label2->Caption = "Auszahlung nicht möglich";
  FStatus = 2;
  btnOK->Caption = "Beenden";
}
```

Nach der Zuweisung der neuen Texte für die Anzeige und die Schaltfläche wird der aktuelle Status auf Zwei eingestellt. Das Formular bleibt solange mit den neuen Texten auf dem Bildschirm, bis die Schaltfläche erneut angeklickt wurde. Dieses Mal erkennt sie den aktuellen Status 2 und führt die Methode *Close()* aus, die das Fenster schließt.

Verändern von Klassen ohne Vererbung

Wie nützlich die Vererbung zur Anpassung von bestehenden Klassen ist, wurde bereits beschrieben. Aber auch ohne den Mechanismus der Vererbung lassen sich bestehende Klassen in ihren Eigenschaften verändern. Die Nachteile der Vererbung sind folgende:

- Mit jeder Hierarchiestufe wird das System komplexer. Klassen, die über mehr als vier Stufen abgeleitet werden, sind irgendwann nicht mehr zu pflegen.
- Mit jeder Hierarchiestufe wird das System langsamer. Jede Methode einer abgeleiteten Klasse muß den Methodenaufruf an die übergeordnete Klasse weitergeben, sofern die Methode nicht komplett überschrieben wurde.

Eine andere Methode, eine Klasse in Ihren Eigenschaften zu ändern, ist die Verwendung eines sogenannten Dekorierers. Für das vorliegende Beispiel soll die Klasse des Geldautomaten durch eine Klasse für einen Fahrkartenautomaten ergänzt werden. Die Eigenschaften dieses Automaten sollen ähnlich dem Geldautomaten sein. Lediglich die Eingabe der Geheimnummer soll entfallen.

Es würde sich hier anbieten, die Klasse *TfrmGeldautomat* durch Vererbung anzupassen. Statt dessen soll die neue Klasse aber kein Nachfolger, sondern ein Dekorierer von TfrmGeldautomat sein. Dazu wird eine neue Klasse *BahnkartenClass* erzeugt:

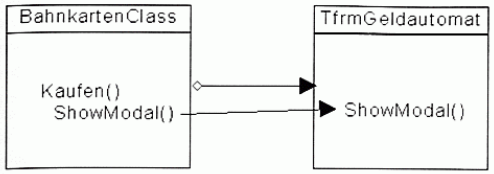

Abb. 13: Verwendung eines Dekorierers zur Erweiterung einer Klasse

```
BahnkartenClass::BahnkartenClass()
{
  Form = new TfrmGeldautomat(NULL);
}
```

Die neue Klasse enthält eine Instanz auf *TfrmGeldautomat*. Der Parameter NULL gibt lediglich an, daß die Form zu keinem anderen Fenster gehört. Im Destruktor wird die Klasse wieder zerstört.

Als neue Methode wurde die Methode *Kaufen()* eingeführt, um eine Bahnfahrkarte mit der Kreditkarte am Automaten zu erwerben:

```
BahnkartenClass::Kaufen(KontoClass *Konto)
{
  Form->SetStatus(2);
  Form->Edit1->Visible = false;
  Form->Label2->Caption =
    "Vielen Dank für Ihren Auftrag";
  Form->Label3->Caption =
    "Bitte entnehmen Sie die Fahrkarte";
  Form->Caption =
     "Fahrkarten-Automat";
  Form->ShowModal();
  return true;
}
```

Anders als bei der Ableitung einer Klasse aus einer Basisklasse wird hier der Aufruf der Methode an die erzeugte Instanz weitergeleitet. Dabei werden die Eigenschaften entsprechend verändert. In diesem Fall wird der Status der Form auf 2 gesetzt, da ja die Abfrage der Geheimnummer entfällt. Gleichzeitig wird das Eingabefenster unsichtbar gemacht und die Titel der Labels entsprechend vorgegeben. Anschließend wird *ShowModal()* aufgerufen.

Damit wird die Methode *ShowModal()* des Geldautomaten entsprechend »dekoriert«, was dieser Technik auch den Namen gegeben hat. Dekorierer können immer dort sehr praktisch eingesetzt werden, wo nur wenige Methoden von Klassen verändert werden sollen. Im Gegensatz zur Vererbung, wo jeder Methodenaufruf automatisch an die Basisklasse weitergegeben wird, muß beim Dekorierer jeder Methodenaufruf selbst weitergeleitet werden.

Beispielsweise kann die Methode *BahnkartenClass::Auszahlen()* nicht aufgerufen werden, weil sie nicht als Methode der Klasse deklariert wurde. Würde sie tatsächlich benötigt werden, müßte man die Methode im Dekorierer einrichten und den Aufruf direkt an die zu dekorierende Klasse weiterleiten.

Aufgerufen wird die Bahnkarten-Klasse wiederum über den Adapter, der bereits für den Aufruf des Geldautomaten zuständig ist:

```
bool AdapterClass::Bahnkarte(KontoClass *Konto, double Betrag)
{
  BahnkartenClass *Karten = new BahnkartenClass();
  Karten->Kaufen(Konto);
  delete(Karten);
  return true;
}
```

Da die Bahnkarten-Klasse nur zum Anzeigen, und gegebenenfalls zum Auswerfen der Fahrkarte benötigt wird, wird hier eine Instanz einfach neu erzeugt und nach dem Kaufvorgang wieder zerstört. Die Verwendung erfolgt ebenso wie die Auszahlen-Methode von *AdapterClass*, nur daß die Methode hier *Bahnkarte()* heißt.

Zuständigkeitskette einrichten

Ein weiteres interessantes Konzept in der objektorientierten Programmierung wird als »Zuständigkeitskette« bezeichnet. Es ist besonders auch deshalb interessant, weil es in MS-Windows ständig intern verwendet wird. Wer sich schon einmal näher mit den inneren Zusammenhängen von Windows befaßt hat, wird wissen, daß dort eine Nachrichtenschlange für die Weiterleitung von Systemnachrichten von einer Anwendung zur nächsten zuständig ist.

Es ist allerdings gar nicht nötig, dieses Konzept allein in technischen Verfahren zu suchen. Die objektorientierte Programmierung versucht ja, ein möglichst realistisches Abbild der Wirklichkeit mit ihren Objekten zu erzeugen. Und tatsächlich ist gerade die Zuständigkeitskette ein gutes Beispiel dafür. Das weiß sicher jeder, der schon einmal versucht hat, bei einer Behörde oder größeren Firma einen zuständigen Ansprechpartner zu erreichen.

Die Erklärung des Prinzips ist recht einfach: Ein Objekt erhält eine Nachricht oder Anforderung vom Programm und prüft zunächst, ob es dafür zuständig ist. Anschließend leitet das Objekt eben diese Nachricht an das nächste Objekt weiter. Das geht solange, bis kein Nachfolger mehr vorhanden ist.

Für das Telefonbeispiel heißt das: Die Zentrale verbindet an Herrn Meier, dieser ist nicht zuständig und verweist an Frau Krause, diese gibt an Herrn Müller und schließlich bricht die Verbindung ab, weil man Sie irgendwohin ins Leere verbunden hat.

Bezogen auf ein praktisches Beispiel im Zusammenhang mit der oben entwickelten Banksoftware könnte eine Anforderung an das System lauten: Entwickeln Sie ein E-Mail-System, mit dem Mitteilungen sowohl an einzelne Mitarbeiter als auch an ganze Abteilungen verschickt werden können. Ein Ansatz zur Lösung dieser Aufgabe könnte dann so aussehen:

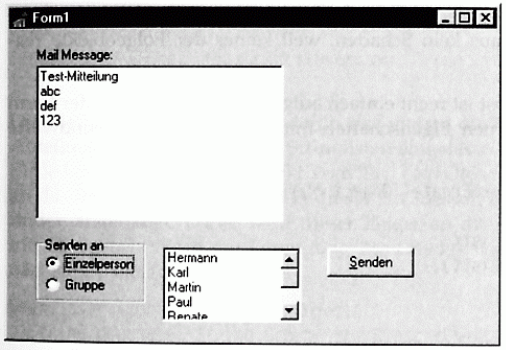

Abb. 14: Einfaches Modell für ein E-Mail System

Als Basis für das Modell dient eine Kette von User-Objekten. Jedes dieser Objekte besitzt einen Namen und eine zugeordnete Abteilung. Wird eine Mail vom System verschickt, wird die Methode *ProcessMail()* des User-Objekts verwendet. Jedes User-Objekt prüft nun, ob es für die Mail zuständig ist. Danach wird *ProcessMail()* für das Nachfolgeobjekt aufgerufen.

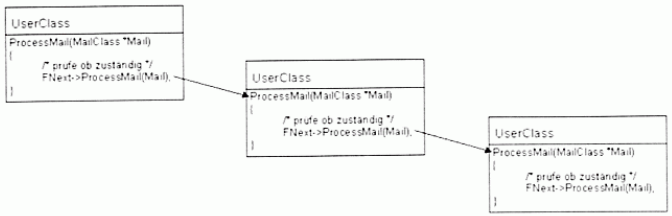

Abb. 15: Behandlung einer E-Mail in den User-Objekten

Anders als im wirklichen Leben wird die Nachricht immer an den Nachfolger weitergeleitet. Je nach Anforderung kann eine Zuständigkeitskette die Weiterleitung auch unterbinden, wenn die Nachricht von einem zuständigen Objekt abgearbeitet wurde. In der Regel sollten die Nachrichten immer bis an das letzte Glied der Kette versandt werden. Sind die nachfolgenden Objekte nicht dafür zuständig, so entsteht daraus kein Schaden, weil keines der Folgeobjekte reagiert.

Das Beispielprogramm selbst ist recht einfach aufgebaut. Bei Erzeugen der Form werden zunächst die internen Eigenschaften initialisiert und auf Grundwerte gesetzt:

```
void __fastcall TForm1::FormCreate(TObject *Sender)
{
  FUser     = new TStringList();
  FGroups   = new TStringList();
  FNextUser = NULL;
  ReadAllUser();
}
```

Die Eigenschaften *FUser* und *FGroups* sind Zeiger auf Listen vom Typ *TStringList*. Diese müssen zunächst über einen Konstruktor erzeugt werden. *FNextUser* stellt den Zeiger auf das nächste User-Objekt dar. Zunächst wird er mit NULL belegt, da noch keine User erzeugt wurden. Die Methode *ReadAllUser()* liest schließlich alle Mitarbeiter in die entsprechenden Listen ein und erzeugt die Liste der User-Objekte:

```
void TForm1::ReadAllUser(void)
{
  FUser->Clear();
  FUser->Sorted = true;
  FGroups->Clear();
  FGroups->Sorted = true;
  ReadUser("Martin", "Kasse");
```



```
  ReadUser("Renate", "Kasse");
  ReadUser("Wolfgang", "Immobilien");
  ReadUser("Ute", "Immobilien");
  ReadUser("Hermann", "Filialleiter");
  ReadUser("Karl", "Kredit");
  ReadUser("Paul", "Kredit");
}
```

In diesem Fall werden die Mitarbeiter über konstante Werte eingelesen. Normalerweise würden diese Werte aus einer Mitarbeiter-Datenbank stammen. Die String-Listen *FUser* und *FGroups* werden so eingestellt, daß sie eine sortierte Liste enthalten. Dadurch werden auch Doppeleinträge bei *FGroups* vermieden, wenn der Name einer Abteilung mehr als einmal vorkommt.

```
void TForm1::ReadUser(String Username, String Group)
{
  FUser->Add(Username);
  FGroups->Add(Group);
  FNextUser = CreateUserObject(Username, Group, FNextUser);
}
```

Die Methode *ReadUser()* sorgt dafür, daß einmal die Namen der Mitarbeiter und Abteilungen in die jeweiligen Stringlisten eingelesen werden. Andererseits wird für jeden Mitarbeiter ein Objekt vom Typ *UserObject* angelegt. Das Objekt wird an das letzte vorhandene Objekt dieser Art, dessen Zeiger sich in *FNextUser* befindet, angehängt. Daher wird dieser Zeiger an die Methode *CreateUserObject()* mit übergeben. Nach der Erzeugung des Objekts wird der neue Zeiger auf das nächste Objekt zurückgeliefert:

```
UserClass* TForm1::CreateUserObject(
  String Username, String Group, UserClass *FNextUser)
{
  static offset;
  offset += 10;
  UserClass *retval = new UserClass(Username, Group);
  retval->SetNext(FNextUser);
  retval->SetOffset(offset);
  return retval;
}
```

Hier wird nun das Objekt vom Typ *UserClass* erzeugt. Der alte Zeiger auf das nächste Objekt wird dem Objekt übergeben. In der Methode *UserClass::SetNext()* wird der übergebene Zeiger dann dort der Eigenschaft *FNext* zugewiesen. Auf diese Weise erhält jedes Objekt einen Zeiger auf sein Nachfolgeobjekt, solange bis *FNext* NULL ist, was bedeutet, daß kein Nachfolger mehr vorhanden ist. Die

Variable *offset* wird übrigens nur aus optischen Gründen benötigt, damit die angezeigten Nachrichtenfenster nicht alle übereinander ausgegeben werden.

Die Abarbeitung der Mail erfolgt über die Methode *ProcessMail()*, die beim Anklicken der Schaltfläche »Send« aufgerufen wird:

```
void __fastcall TForm1::btnSendClick(TObject *Sender)
{
  String the_name;
  the_name = ListBox1->Items->Strings[ListBox1->ItemIndex];
  FAdress->SetName(the_name);
  FMail = new MailClass(Memo1->Lines, FAdress);
  FNextUser->ProcessMail(FMail);
}
```

Um die Mail zu versenden, wird zunächst der Name aus der Listbox ermittelt. Je nachdem, ob an eine Person oder Abteilung versandt werden soll, steht hier entweder der Name des Mitarbeiters oder der Name der Abteilung. Die Eigenschaft *FAdress* ist ein Objekt der Klasse *IndivAdrClass* oder *GroupAdrClass*. Die Zuweisung der entsprechenden Klasse erfolgt je nach Einstellung von *RadioGroup1*, was anschließend noch näher erläutert wird.

Die Eigenschaft *FMail* ist ein Zeiger vom Typ *MailClass*. Um eine Mail zu versenden, wird zunächst ein Objekt vom Typ *MailClass* erzeugt, in dem dann die Nachricht und die Adresse des Empfängers gespeichert sind. Vorteil dieser Methode ist, daß für den Aufruf von *ProcessMail()* nur ein Objektzeiger als Parameter übergeben wird. Damit wird der Aufruf übersichtlicher und alle relevanten Daten sind in dem Objekt gespeichert.

Die Erzeugung eines Adressen-Objekts erfolgt jedesmal neu, wenn in der Optionsgruppe »Einzelperson« oder »Gruppe« ausgewählt wird. Dann wird die Methode *RefreshList()* aufgerufen:

```
void TForm1::RefreshList(void)
{
  ListBox1->Clear();
  if (FAdress != NULL)
  {
    delete(FAdress);
  }
  switch (RadioGroup1->ItemIndex)
  {
    case 0:
      ListBox1->Items->AddStrings(FUser);
      FAdress = new IndivAdrClass();
     break;
```



```
    case 1:
      ListBox1->Items->AddStrings(FGroups);
      FAdress = new GroupAdrClass();
     break;
  }
}
```

Ein eventuell vorhandenes Objekt in *FAdress* wird zunächst gelöscht. Anschließend wird ein Objekt vom Typ *IndivAdrClass* erzeugt, wenn die Nachricht an eine Einzelperson gehen soll. Im anderen Fall wird der Typ *GroupAdrClass* verwendet. Damit wird es später möglich, ohne umständliche *if*-Abfragen festzustellen, ob sich die Nachricht auf eine Person oder eine Abteilung bezieht.

Das User-Objekt ist nun dafür zuständig, die Nachricht dem zugehörigen Mitarbeiter anzuzeigen, und sie an das nächste Objekt in der Kette weiterzureichen:

```
void UserClass::ProcessMail(MailClass *Mail)
{
  if (Mail->GetAdress()->CanRead(this))
  {
    TfrmMessage *msg = new TfrmMessage(NULL);
    msg->Memo1->Lines = Mail->GetMessage();
    msg->Caption = FName;
    msg->Left -= FOffset;
    msg->Top -= FOffset;
    msg->Show();
  }
    if (FNext != NULL)
  {
    FNext->ProcessMail(Mail);
  }
}
```

Dazu wird zunächst aus dem übergebenen Mail-Objekt die Methode *CanRead()* des eingebundenen Adreßobjekts aufgerufen. *CanRead()* stellt fest, ob die Nachricht an das Userobjekt gerichtet ist. Dazu wird der Parameter *this* mit übergeben, der einen Zeiger auf sich selbst, also auf das gerade aktive Userobjekt, darstellt. Ist das Objekt für die Nachricht zuständig, wird ein Fenster vom Typ *TfrmMessage* erzeugt, in dem die Nachricht angezeigt wird. Anschließend wird die Mail mit

```
FNext->ProcessMail(Mail)
```

an das nächste Objekt weitergeleitet,

Die Methode *CanRead()* ist für jeden Adreßtyp jeweils anders deklariert, so daß zwischen Einzelnachricht und Gruppennachricht unterschieden wird:

```
bool IndivAdrClass::CanRead(UserClass *user)
{
  String name:
  name = user->GetName():
  return (name == FUsername):
}

bool GroupAdrClass::CanRead(UserClass *user)
{
  String name:
  name = user->GetGroup():
  return (name == FGroup):
}
```

Der übergebene Parameter *user* entspricht dem oben beschriebenen *this*-Zeiger, der das User-Objekt darstellt, das die Nachricht gerade bearbeitet. Im ersten Fall wird der Name des Mitarbeiters, im zweiten Fall der Name der Abteilung ausgelesen und mit dem entsprechenden Namen (*FUsername* oder *FGroup*) verglichen. Das spart unnötige *if*-Abfragen und kann später problemlos um weitere Parameter, beispielsweise für den Versand an ganze Filialen, erweitert werden.

Wenn Sie das Programm einmal ausprobieren, können Sie sowohl eine Nachricht an Einzelpersonen als auch an die Abteilungen verschicken. Beim Anklikken der Schaltfläche »Send« erscheint für jede Person, die diese E-Mail erhalten soll, ein Nachrichtenfenster.

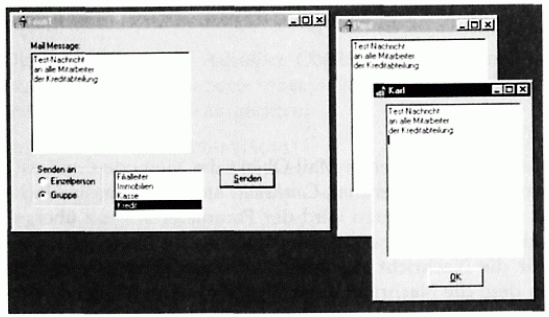

Abb. 16: Das Mail-System in Aktion

In einem praktischen Programm würde man hier natürlich eine Prozedur durchführen, mit dem die Nachricht über das hausinterne Mail-System an die richtige Adresse verschickt wird.