Von der Subroutinentechnik zur objektorientierten Programmierung : Teil 3 : das Prinzip der objektorientierten Programmierung

Autor(en): **Marty, Rudolf**

Objekttyp: **Article**

Zeitschrift: **Bulletin des Schweizerischen Elektrotechnischen Vereins, des Verbandes Schweizerischer Elektrizitätsunternehmen = Bulletin de l'Association Suisse des Electriciens, de l'Association des Entreprises électriques suisses**

Band (Jahr): **81 (1990)**

Heft 21

PDF erstellt am: **08.08.2024**

Persistenter Link: https://doi.org/10.5169/seals-903178

Von der Subroutinentechnik zur objektorientierten Programmierung

Teil 3 Das Prinzip der objektorientierten Programmierung

Rudolf Marty

In den beiden ersten Teilen dieser Reihe wurde die Softwareentwicklung von der Subroutinentechnik bis zu den abstrakten Datentypen dargestellt. Der dritte und letzte Teil befasst sich mit der Erweiterung dieser Konzepte zum Prinzip der objektorientierten Programmierung. Wichtigste Merkmale dieser modernen Softwaretechnik sind das Vererbungsprinzip sowie die Definition von Objekten als einheitliche Beschreibung von Daten und (auf diese anwendbaren) Operationen.

Dans les deux premiers articles de cette série, on a présenté le développement des logiciels en partant de la technique des subroutines précoces jusqu'au concept des types de données abstraites. Le troisième et dernier article s'occupe de l'évolution de ce concept vers le principe de la programmation orientée objet dont les principales caractéristiques sont le principe génétique et la définition d'objets en tant que description unitaire des données et opérations (applicables sur ces données).

Adresse des Autors

Prof. Dr. *Rudolf Marty*, Schweiz. Bankgesellschaft, UBILAB (UBS Informatics Laboratory), 8021 Zürich

6. Ein fünfter Abstraktionsschritt: Objektorientierte Programmierung

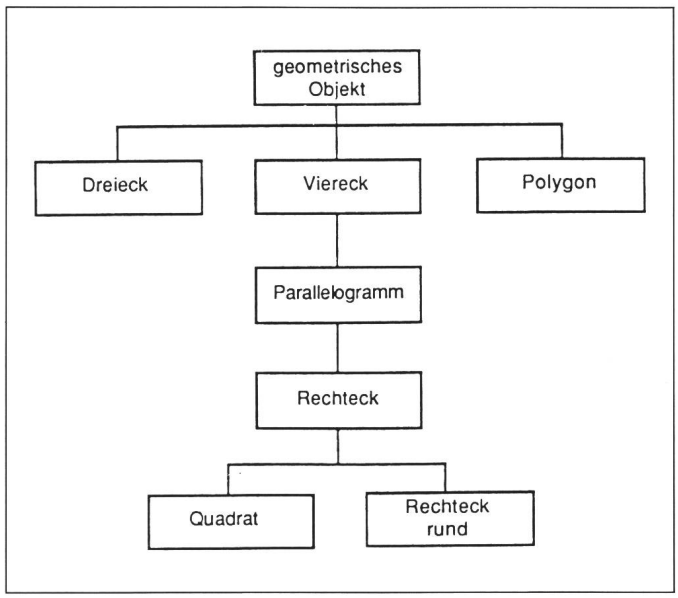
Durch Angabe der Ahnenklasse in einer Klassendefinition entstehen explizite Vererbungshierarchien. Bild 11 zeigt einen Ausschnitt aus einer Klassenhierarchie für geometrische Objekte, wie sie z.B. in einem Grafik-Softwarepaket verwendet werden könnten. Explizit in eine solche Klassenhierarchie eingebunden, wird RechteckRund durch die Deklaration in Bild 12 vollständig definiert. Es fallen im Vergleich mit der Definition eines abgeleiteten ADT (Bild 10, dort jedoch nur auszugsweise wiedergegeben) einige Änderungen auf:

- Im Kopf der Klassendeklaration erscheint die Angabe der *Oberklasse* Rechteck. Konkret bedeutet der C++ Text

```
class RechteckRund : public Rechteck
```

folgendes: Es wird eine Klasse RechteckRund definiert, die von der Klasse Rechteck abgeleitet ist, wobei public an dieser Stelle bedeutet, dass alle öffentlichen Namen von Rechteck auch öffentliche Namen von RechteckRund sind.
- Die Klasse RechteckRund enthält nur eine Instanzvariable Eckradius. Die beiden anderen zur Definition eines Rechtecks benötigten Instanzvariablen (die Eckpunkte) sind ja bereits in der Oberklasse Rechteck enthalten. Da RechteckRund von Rechteck abgeleitet ist, erbt sie automatisch all deren Instanzvariablen.
- Neben den Instanzvariablen erbt eine Klasse auch alle Funktionen ihrer Oberklasse. Zur Definition der Klasse RechteckRund müssen demzufolge nur noch diejenigen Funktionen ausprogrammiert werden, die in der Oberklasse fehlen oder in einer ungeeigneten Form definiert sind. Das sind lediglich die Funktion Setze, die neu zusätzlich den Eckradius als Parameter

Bild 11
Klassenhierarchie für geometrische Objekte



```
class RechteckRund : public Rechteck {
public:
    int     Eckradius;
    void    Setze   (Punkt *Pa, Punkt *Pb, int Radius);
    float   Flaeche ( );
};

void RechteckRund::Setze (Punkt *Pa, Punkt *Pb, int Radius)
{
    Rechteck::Setze(Pa,Pb);
    Eckradius = Radius;
}

float RechteckRund::Flaeche ( )
{
    return Rechteck::Flaeche() -
           Eckradius * Eckradius * (4 - pi);
}
```

Bild 12 Abgeleitete Klasse

RechteckRund ist durch diese Deklaration vollständig definiert.

übernimmt, und die abgeänderte Funktion Flaeche. Alle anderen zur Klasse Rechteck gehörenden Funktionen (cf. Bild 7 und 8) gelten unverändert auch für die Klasse RechteckRund.

Basierend auf den Definitionen der Klassen Rechteck und RechteckRund sind beispielsweise die Deklarationen und Anweisungen in Bild 13 denkbar.

```
Rechteck          r1, r2;
RechteckRund      ru1, ru2;
Punkt             p1, p2, p3, p4;
float             a,b;

...
r1.Setze(p1,p2);
ru2.Setze(p3,p4,5);
r1.Verschiebe(-12,10);
ru2.Verschiebe(42,0);
a = r1.Flaeche() + ru2.Flaeche();
b = ru2.Eckradius;
```

Bild 13 Objektdeklarationen und -verwendungen

Basierend auf den Definitionen der Klassen Rechteck und RechteckRund sind beispielsweise die obigen Deklarationen und Anweisungen denkbar, wobei vier Instanzen definiert sind.

Grundkonzepte objektorientierter Programmierung

Bevor wir weitergehen, müssen einige Begriffe der objektorientierten Programmierung eingeführt und gewisse Konzepte gefestigt werden:

1. Eine Instantiierung einer Klasse wird als *Objekt* bezeichnet. Wie eine Variable von einem bestimmten Typ ist, ist ein Objekt von einer bestimmten Klasse.

2. Die Funktionen einer Klasse werden *Methoden* genannt. Es wird also eine Methode eines Objektes aufgerufen. Noch präziser gesprochen: Es wird ein Objekt aufgerufen, wobei im Aufruf der Name der auf dieses Objekt auszuführenden Methode genannt wird. In Bild 13 bedeutet z.B.

r1.Verschiebe(−12,10)

«Rufe das Objekt r1 auf (das von der Klasse Rechteck ist) und führe dessen Methode Verschiebe mit den Argumenten −12,10 aus.»

3. Die Klasse, von der eine bestimmte Unterklasse (Subclass) abgeleitet wird, wird als deren Oberklasse (Superclass) bezeichnet. Die Unterklasse erbt (Inherits, Inheritance) alle Instanzvariablen und Methoden der Oberklasse. Die Vererbungshierarchie ist in ihrer Tiefe nicht begrenzt. Von einer Klasse können beliebig viele Unterklassen abgeleitet werden.

4. Eine Unterklasse kann geerbte Instanzvariablen durch eigene Instanzvariablen ergänzen. In der Unterklasse RechteckRund haben wir z.B. die Instanzvariable Eckradius zugefügt.

5. Eine Unterklasse kann geerbte Methoden durch Deklaration gleichnamiger Methoden ersetzen (z.B. Flaeche in RechteckRund) und auch neue Methoden einführen (z.B. Eckradius in RechteckRund).

6. Wird ein Objekt zusammen mit einer Methode aufgerufen, die es nicht kennt, dann wird der Aufruf an die Oberklasse weitergereicht. In Bild 13 wird mit

ru2.Verschiebe(42,0)

die Methode Verschiebe des Objektes ru2 (das von der Klasse RechteckRund ist) aufgerufen. Da in der Klasse RechteckRund keine solche Methode definiert wurde, wird der Aufruf automatisch an die Oberklasse Rechteck weitergereicht, die nun ihrerseits die Methode Verschiebe kennt. Hätte auch sie die Methode nicht gekannt, so wäre der Aufruf Stufe um Stufe weitergereicht worden (die Klassenhierarchie hinauf). Ein Objekt reagiert auf einen Methodenaufruf demnach wie in Bild 14 dargestellt.

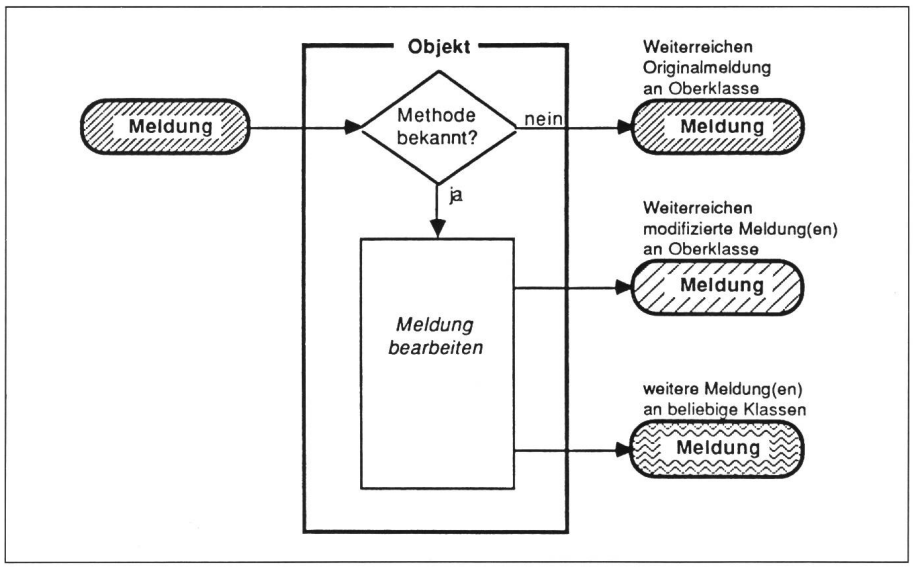


Bild 14 Objektverhalten bei Empfang einer Meldung

Systeme aus wiederverwendbaren Klassenhierarchien

Betrachten wir in Bild 15 ein weiteres Beispiel einer Klassenhierarchie: Die Klasse Matrix implementiert eine einfache, zweidimensionale Tabelle ohne Formatierinformationen und Rechenformeln für die einzelnen Matrixfelder. Die Klasse Spreadsheet baut auf der Klasse Matrix auf und führt die aus Tabellenkalkulationsprogrammen bekannten Möglichkeiten der automatischen Berechnung von Feldern, der Formatierung der Tabelle sowie der Verknüpfung mit anderen Tabellen ein. Das Kontoblatt schliesslich ist im Grunde genommen nichts anderes als eine spezielle Art von Tabelle; wir bilden die Klasse Kontoblatt also als Subklasse von Spreadsheet (Bild 16).

Bereits an den kleinen Beispielen von geometrischen Objekten und von Tabellenstrukturen erkennen wir eine zentrale Eigenschaft objektorientierter Systeme, eine Eigenschaft von Systemen also, die aus Objekten als Instanzen von in Hierarchien eingebundenen Klassen bestehen: Eine Klassendefinition wird sehr klein und übersichtlich, da sie auf einer bereits bestehenden Klasse aufbaut und diese lediglich leicht verändert oder ergänzt. Diese Situation ist uns bestens aus der industriellen Fertigung bekannt, wo man auch auf Halbfabrikaten aufbaut, diese u.U. leicht abändert (z.B. in einem Apparat einen Gleichstrommotor durch einen Wechselstrommotor ersetzt) oder ergänzt (etwa durch eine Halterung oder ein Anschluss-Zwischenstück).

Die Vorteile dieses Vorgehens sind weitreichender, als man dies auf den ersten Blick erahnt. In der Modultechnik und bei Verwendung von abstrakten Datentypen haben wir entweder bestehende Module bzw. Klassen übernommen und deren Quelltext manuell modifiziert oder aber Funktionen eines allgemeineren Moduls bzw. ADT explizit, also auch unter Kenntnis dessen Parameterstruktur, aufgerufen. Damit wird nicht nur die Programmierung ungebührlich belastet, wichtiger ist der Verlust an Flexibilität. Weil wir im objektorientierten Ansatz auf jeder Stufe der Klassenhierarchie nur genau die *Differenz zu der Oberklasse* ausformulieren und alles andere unbesehen übernehmen, schlagen Änderungen an einer Klasse automatisch auf alle Unterklassen durch.

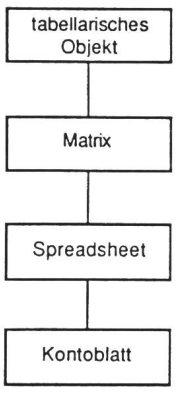


Bild 15 Klassenhierarchie für tabellarische Objekte

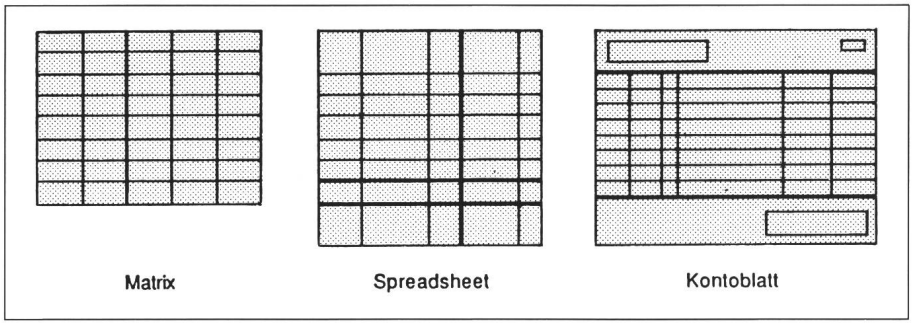


Bild 16 Matrizen, Spreadsheets und Kontoblätter

Stellen wir uns hierzu vor, man hätte die Klasse Matrix so implementiert, dass alle Matrixfelder als zweidimensionales Feld im Hauptspeicher abgespeichert sind. Folglich werden also auch alle Felder des Spreadsheets und des Kontoblatts hauptspeicherintern gehalten, da diese beiden Klassen bei korrektem objektorientiertem Aufbau nicht selbst einen Code für die Verwaltung der Einzelfelder und den Zugriff hierauf enthalten. Tritt nun das Bedürfnis nach grösseren Tabellen auf, was gerade für Konten mit möglicherweise Tausenden von Einzelbuchungen typisch ist, so wird in der Klasse Matrix die Organisation der Felder und der Zugriff hierauf verändert: Es wird nicht mehr die ganze Matrix im Hauptspeicher gehalten, sondern nur gerade derjenige Teil, der benötigt wird; der Rest wird auf Sekundärspeicher ausgelagert, wie wir das von Seitenaustauschverfahren in Betriebssystemen kennen. Diese Änderung der Klasse Matrix schlägt auf alle Unterklassen durch. Beliebig grosse Spreadsheets und Konten mit beliebig vielen Einzelbuchungen sind ohne auch nur die kleinste Korrektur an deren Klassendefinition Wirklichkeit geworden.

Bei kluger objektorientierter Programmierung ist auch das Anzeigen von Tabellenteilen in der Klasse Matrix (oder sogar noch höher in der Klassenhierarchie) auscodiert worden. Spreadsheet und Kontoblatt basieren auf den Anzeigemethoden der Oberklasse Matrix. Werden nun, wie vorgängig umschrieben, beliebig grosse Matrizen eingeführt, so wird innerhalb der Klasse Matrix auch die Anzeigemethode so umprogrammiert, dass Tabellenteile, die nicht gesamtheitlich auf einem Bildschirm Platz finden, horizontal und vertikal «gerollt» werden können. Auch diese Neuerung steht allen Unterklassen zur Verfügung; ein Kontoblatt wird plötzlich mit einem Rollbalken für den Buchungsteil angezeigt, ohne dass auch nur das Geringste an der Klasse Kontoblatt geändert worden wäre.

Damit haben wir auf der Ebene der Programmstruktur einen Grad an Wiederverwendbarkeit erreicht, wie er in der Softwareentwicklung mit allen bisher bekannten Methoden klassischer Programmierung nicht erreichbar war. Softwaresysteme entstehen als Hierarchie von Klassen, wobei von Hierarchiestufe zu Hierarchiestufe typischerweise nur sehr kleine Änderungen und Erweiterungen an den Klassen vorgenommen werden. Eine einzelne Klassendefinition und insbesondere die Definition einer Methode wird recht klein. Methoden mit zwei bis fünf Zeilen Code sind keine Seltenheit. Die Kunst der objektorientierten Programmierung besteht darin, kluge Klassenhierarchien aufzubauen, das heisst insbesondere, in Unterklassen entstehende Gemeinsamkeiten und Doppelspurigkeiten zu erkennen, aus diesen Gemeinsamkeiten ein allgemeines, höheres Schema abzuleiten und dieses sodann in der richtigen Oberklasse zu implementieren. Damit entsteht für alle Unterklassen dieser Oberklasse (nicht nur für diejenigen, aus der die Gemeinsamkeiten herausfaktorisiert wurden) eine zusätzliche Funktionalität.

Dynamische Bindung

Es wurde aus den vorangehenden Ausführungen zu der objektorientierten Programmierung klar, dass diesel-

```
Dreieck        *Dreieck1, *Dreieck2, *Dreieck3;
Viereck        *Viereck1, *Viereck2, *Viereck3;
Quadrat        *Quadrat1, *Quadrat2;
Polygon        *Polygon1, *Polygon2;
RechteckRund   *RRund1, *RRund2, *RRund3, *RRund4;

GeomObjekt     *ObjTab[20];
```

Bild 17
Objektreferenzen und Objekttabelle

In C++ wird durch einen Stern * vor einer Variablendeklaration der angegebene Name als Zeiger (Pointer) vereinbart. Mit der untersten Deklaration wird beispielsweise eine Tabelle mit 20 Zeigern auf Objekte des Typs GeomObjekt angelegt.

be Methode sehr oft auf verschiedenen Stufen der Klassenhierarchie verwendet wird (beispielsweise die Methode Flaeche für geometrische Objekte). In den bisherigen Beispielen war aus der syntaktischen Form des Methodenaufrufs stets die angesprochene Klasse ersichtlich: In Bild 13 resultiert z.B. die Ausführung des Ausdrucks

r1.Flaeche() + ru1.Flaeche()

im Aufruf der Methode Flaeche der Klasse Rechteck für r1 (das ja eine Instanz dieser Klasse ist) und im Aufruf der Methode Flaeche der Klasse RechteckRund für das Objekt ru1. Die Klasse dieser Objekte ist aus deren Deklaration ersichtlich; die Methodenaufrufe können also zur Übersetzungszeit an eine Klasse gebunden werden.

Begnügten wir uns mit der Bindung zur Übersetzungszeit, auch *statische Bindung* genannt, so würden wir enorm viel an Flexibilität und Wiederverwendbarkeit von Klassen verlieren. Betrachten wir zur Begründung der *dynamischen Bindung* ein paar Programmfragmente, die auf der Klassenhierarchie von geometrischen Objekten basieren, wie sie in Bild 11 dargestellt ist. In C++ wird durch einen Stern * vor einer Variablendeklaration der angegebene Name als Zeiger (Pointer) vereinbart. In Bild 17 werden also ausschliesslich Zeiger auf Objekte der angegebenen Klasse deklariert. Mit der untersten Deklaration wird beispielsweise eine Tabelle mit 20 Zeigern auf Objekte des Typs GeomObjekt angelegt. GeomObjekt ist die Wurzel der Klassenhierarchie gemäss Bild 11. ObjTab kann somit Zeiger auf Objekte aller GeomObjekt untergeordneten Klassen aufnehmen.

Betrachten wir die vier Abschnitte aus Bild 18:

1. Im ersten Abschnitt werden (dynamisch) drei Objekte angelegt. In C++ kann, wie bereits früher bemerkt, für jede Klasse eine Initialisierungsfunktion angegeben werden. Diese wird bei dynamischen Objekten unmittelbar nach Anlegen des Objektes, das heisst als Teil der durch new ausgelösten Aktionen ausgeführt.

2. Im zweiten Abschnitt erscheinen einige Zuweisungen von Objektzeigern. Die Zuweisung eines Objektzeigers an einen Zeiger, der auf ein in der Klassenhierarchie höher gelegenes Objekt verweist, ist zulässig. Diese Regel ist durchaus einsichtig: Eine Klasse umfasst wie besprochen alle Instanzvariablen ihrer Oberklasse. Wird ein Objektzeiger einem klassenhierarchisch übergeordneten Objektzeiger zugewiesen, so verweist dieser nach der Zuweisung vermeintlich auf ein Objekt seiner Klasse und damit auch auf die in seiner Klasse bekannten Instanzvariablen. In der Tat besteht die Referenz jedoch auf ein Objekt tieferer Klasse, das meist weitere Instanzvariablen kennt. Dies schadet jedoch nichts, denn die zusätzlichen Instanzvariablen werden strukturell an die bestehenden angehängt, so dass auf höherer Stufe automatisch die dieser Klassenstufe zugehörenden Instanzvariablen referenziert werden, ob das referenzierte Objekt von dieser oder einer beliebigen Unterklasse ist.

Ungültig ist die zweitletzte Zuweisung, da Viereck nicht eine Oberklasse von Dreieck ist. Auch Zuweisungen hierarchieabwärts sind nicht gestattet, da damit beispielsweise ein Zeiger auf ein Objekt der Klasse RechteckRund auf eines der Klasse Rechteck zeigen könnte, in dem die in der Klasse RechteckRund definierte Instanzvariable Eckradius fehlen würde (cf. Bild 12).

3. Aus dem dritten Abschnitt in Bild 18 wird ersichtlich, dass Referenzen auf Instanzvariablen und Methoden dynamisch angelegter Objekte mit dem Zeiger-Dereferenzierungssymbol «->» codiert werden. RRund2->Eckradius bezeichnet die Instanzvariable Eckradius des Objektes, auf welches der Zeiger RRund2 zeigt; Viereck2->Verschiebe(10,-5) ruft die Methode Verschiebe des mit Viereck2 referenzierten Objektes auf.

Die Frage entsteht nun sofort, welche Methode Verschiebe in der Konstruktion ObjTab[j]->Verschiebe(dx,dy) angesprochen wird. Offensichtlich diejenige des Objektes, auf welches der an Stelle j im Array ObjTab liegende Zeiger verweist. Je nach vorangegangener Zuweisung an ObjTab[j] kann dies ein Objekt einer beliebigen Unterklasse von GeomObjekt sein. Folglich kann der Methodenaufruf Verschiebe nicht zur Übersetzungszeit an eine Klasse gebunden werden; erst zur Laufzeit wird aufgrund des referenzierten Objektes entschieden, welche Klasse angesprochen werden muss. Dies nennt man *dynamische Bindung* (Dynamic Binding).

4. Im letzten Abschnitt von Bild 18 wird mit einer Schleife die Gesamtfläche aller durch ObjTab referenzierten geometrischen Objekte errechnet. Wiederum gelangt dynamische Bindung zum Einsatz, da zur Übersetzungszeit nicht entschieden werden kann, an welche Klasse der Methodenaufruf Flaeche gebunden werden muss.

Die dynamische Bindung bringt ausserordentlich viel an Zusatzflexibilität und Wiederverwendbarkeit. Es können auf hierarchisch höherer Stufe allgemeingültige Algorithmen ausco-

Bild 18
Objektverwendungen mit dynamischer Bindung

```
Dreieck1 = new Dreieck;
Polygon3 = new Polygon;
Quadrat2 = new Quadrat;
...
Viereck1 = Quadrat1;
Viereck2 = RRund4;
Viereck3 = Dreieck2;    ungültig!
ObjTab[12] = Polygon2;
...
r = RRund2->Eckradius;
Viereck2->Verschiebe(10,-5);
ObjTab[j]->Verschiebe(dx,dy);
...
Gesamtflaeche = 0;
for (i = 0; i < AnzahlObjekte; ++i)
     Gesamtflaeche += ObjTab[i]->Flaeche();
```

```
class GeomObjekt : public Object {
public:
    virtual float   Flaeche ( );
};

float GeomObjekt::Flaeche ( )
{
    Fehler("Flaeche nicht definiert");
}
```

Bild 19
Dynamisch gebundene Methode in C++

In C++ wird durch den Zusatz des Schlüsselwortes virtual zu einer Methodendefinition angeben, dass deren Aufrufe in dieser und allen untergeordneten Klassen dynamisch gebunden werden sollen.

diert werden, ohne die Struktur der durch sie behandelten Objekte zu kennen. Die Gesamtflächen-Berechnung, wie sie in Bild 18 dargestellt ist, bleibt unverändert, auch bei Definition von neuen Unterklassen von GeomObjekt, solange jede Klasse eine Methode zur Berechnung ihrer Fläche kennt. Bei Anwendung statischer Bindung müsste zwecks Aufruf der korrekten Methode eine Fallunterscheidung mit einer Switch-Anweisung (entspricht case in Modula-2) codiert werden. Ausserdem wären die Instanzvariablen in einen Variantenrecord einzugliedern. Damit wird nicht nur der Code erheblich verlängert und unlesbarer gemacht, gravierender ist die Tatsache, dass bei Einführung einer neuen Klasse an allen betroffenen Stellen die Fallunterscheidungen und der Variantenrecord nachgetragen werden müssen.

In C++ wird durch den Zusatz des Schlüsselwortes virtual zu einer Methodendefinition angegeben, dass deren Aufrufe in dieser und allen untergeordneten Klassen dynamisch gebunden werden sollen. Die Definition der Klasse GeomObjekt könnte beispielsweise, wie in Bild 19 gezeigt, codiert werden.

Die Funktion Flaeche der Klasse GeomObjekt kann selbstverständlich keine Flächenberechnung vornehmen, da die konkrete geometrische Gestalt der Figur nicht bekannt ist. Erst in Unterklassen kann die Flächenberechnung auscodiert werden. Die in Bild 19 deklarierte Funktion Flaeche ist gewissermassen ein «Fangnetz»: Wird in einer Unterklasse von GeomObjekt die Methode Flaeche nicht redefiniert, so wird bei deren Aufruf die geerbte Methode Flaeche von GeomObjekt ausgeführt, die lediglich eine Fehlermeldung produziert. Dies wäre beispielsweise dann der Fall, wenn der Klassenhierarchie von Bild 11 eine Unterklasse Kreis zugefügt würde, die die Methode Flaeche nicht explizit beinhaltet.

In der Praxis der objektorientierten Programmierung führt eine kluge Anwendung dynamischer Bindung dazu, dass algorithmische Gemeinsamkeiten möglichst hoch in die Klassenhierarchie «herausfaktorisiert» werden. Es entstehen algorithmische Grundmuster, die völlig unabhängig von den behandelten Objekten funktionieren. Diese Unabhängigkeit ist dabei nicht nur so zu verstehen, dass ein generischer Algorithmus beschrieben wird, der für verschiedenste Objekttypen instantiiert werden kann (wie wir das z.B. von generischen Modulen oder generischen ADT kennen), sondern dass der Algorithmus überhaupt unabhängig von den Objekttypen (bzw. Objektklassen) wird. Erst zur Laufzeit wird unmittelbar bei Anwendung einer Methode auf ein Objekt entschieden, welche konkrete Methode auszuführen ist.

Natürlich kostet die dynamische Bindung etwas: In C++ wird für jede Klasse eine Referenztabelle ihrer dynamisch gebundenen Methoden angelegt; bei einem dynamisch gebundenen Methodenaufruf wird dann über einen Verweis aus dem Objektdeskriptor die der Methode zugeordnete Prozeduradresse aus der Referenztabelle geladen und auf diese verzweigt. Die Mehrkosten bestehen also hauptsächlich aus einem gesteigerten Hauptspeicherbedarf. Die Laufzeitmehrkosten sind fast vernachlässigbar und werden durch eine üblicherweise gesteigerte Qualität der herausfaktorisierten Algorithmen mehr als wettgemacht.

7. Schlussbemerkungen

Zweifellos ist es ausserordentlich schwierig, aus dieser kurzen Einführung soviel an Einsicht und Verständnis zu gewinnen, um die technischen Möglichkeiten objektorientierter Programmierung umfassend erkennen und deren Auswirkungen auf Struktur, Flexibilität, Wartbarkeit und Wiederverwendbarkeit ganzer Softwaresysteme beurteilen zu können. Nur durch Auseinandersetzung mit objektorientierter Programmierung wird diese Erfahrung geschaffen und gefestigt. Wichtig ist dabei, dass man nicht auf dem absoluten Nullpunkt beginnt. Es sollte wenn möglich bereits eine vernünftige Klassenhierarchie bestehen. Andernfalls beschränkt sich die Erfahrung auf die «Mikro»-Ebene objektorientierter Programmierung: auf die Konstrukte der Programmiersprachen. Der Einsatz einer Klassenhierarchie in Entwurf und Implementation von Software ist aber ebenso wichtig wie das Beherrschen einer objektorientierten Programmiersprache allein. Eine umfassende und konzeptionell reine, jedoch recht umfangreiche Umgebung für einen Einstieg in die Welt der objektorientierten Softwareentwicklung bietet z.B. ein Smalltalk-System, wie es heute auf vielen PC verfügbar ist.

Nach einer Einführung in die Welt objektorientierter Programmierung hört man oft den Kommentar: «Das kann man ja alles auch mit Modula-2, Pascal, C, PL/I...». Selbstverständlich kann eine objektorientierte Softwarestruktur auch in anderen Sprachen nachgebildet werden, selbst in Assembler. Letztlich sind es ja auch nur «gewöhnliche Maschineninstruktionen», die ausgeführt werden. Man kann auch Module mit C, strukturierte Programmierung mit Assembler, Klassen mit Modula-2 nachbilden. Das Problem dabei ist nur, dass der Softwareentwickler in zwei Welten denkt: in der Entwurfswelt und in der Implementierungswelt. Seit geraumer Zeit hat sich die Erkenntnis durchgesetzt, dass die Programmstruktur möglichst kongruent mit dem abstrakten Denkmodell sein soll. Diese Suche nach Übereinstimmung von Denkmodell mit Beschreibungsmodell hat denn auch zur Entwicklung von Sprachen wie Pascal (das auf sauber strukturierte Programmierung ausgerichtet ist) und Modula-2 (wo das Modulkonzept als Sprachbestandteil übernommen wurde) geführt. Genauso verhält es sich in der objektorientierten Systementwicklung. Erst durch eine geeignete Programmiersprache wird es gelingen, die herausragenden Vorteile objektorientierter Programmierung zu nutzen. Bei Verwendung eines nichtobjektorientierten Entwicklungswerkzeuges erscheint die Systemstruktur auf Ebene des Quellcodes als unübersichtliches «Flickwerk».

Geographisches Informationssystem der Zukunft

GRADIS-UX, Software-Lösung der Firma Strässle, ist ein modulares, **Workstation-basierendes, Geographisches Informations-System** mit integrierter relationaler Datenbank ORACLE.

Die konzeptionelle Neuentwicklung GRADIS-UX vereint **modernste Hardwaretechnologie** auf der Basis der Computersysteme HP 9000/xxx unter dem **Betriebssystem UNIX.**

Mit GRADIS-UX steht Ihnen eine leistungsfähige Software für folgende Bereiche zur Verfügung:

- **Energiewirtschaft**
- **Ver- und Entsorgung**
- **Vermessungswesen**
- **Planung und Umwelt**

GIS-Seminar

Zusammen mit der Firma Strässle führen wir zu diesem Thema ein Seminar durch:

Datum/Ort 11. Dezember 1990
im Hotel Hilton,
Glattbrugg/Zürich

Zeit 09.00 Uhr zum Thema:
Vermessung und Energieversorgung
14.00 Uhr zum Thema:
Planung und Umwelt

Kosten Die Teilnahme ist kostenlos

Weitere Auskunft und Anmeldung bei:
Hewlett-Packard (Schweiz) AG, Zürich,
Frau Lucia Frei, Telefon 01-315 81 81

strässle
Technische Informationssysteme

HEWLETT PACKARD

Elektro - Zeichnungs - Service

Haben Sie einen Engpass -
oder suchen Sie eine langfristige Entlastung?

Wir erstellen Fabrikations-, Installations- und Service-Unterlagen nach **Ihren** Entwürfen und Angaben.

Auch Ändern und Nachführen von bestehenden Zeichnungen

Technisches Büro Ulrich Bircher
5000 Aarau Tel. 064 24 60 06

01/207 86 32

Direktwahl zu Ihrem Zielpublikum.

Elektroingenieure ETH/HTL
Leser des Bulletin SEV/VSE
mit Einkaufsentscheiden

NEU GRATIS NEU

Senden Sie uns gratis die angekreuzten Kataloge

- ☐ Gesamtkatalog
- ☐ Katalog Nr. 1 Installationskabel, Telefonkabel und Zubehör
- ☐ Katalog Nr. 2 Netzzuleitungen, Verlängerungen, Spiralkabel, Konfektionen
- ☐ Katalog Nr. 3 Steuerleitungen- und Datenübertragungs-Kabel
- ☐ Katalog Nr. 4 Computerkabel und Zubehör BNC,TNC,N, Twinax
- ☐ **Katalog Nr. 5 ICS Verkabelungssystem und Zubehör BNC, Twinax**
- ☐ Katalog Nr. 6 Ethernet
- ☐ Katalog Nr. 7 LWL

Absender _____

Einsenden an: Heiniger & Co AG Blankweg 4 3072 Ostermundigen