Zeitschrift: Bulletin des Schweizerischen Elektrotechnischen Vereins, des Verbandes Schweizerischer Elektrizitätsunternehmen = Bulletin de l'Association suisse des électriciens, de l'Association des entreprises électriques suisses

Herausgeber: Schweizerischer Elektrotechnischer Verein ; Verband Schweizerischer Elektrizitätsunternehmen

Band: 81 (1990)

Heft: 17

Artikel: Von der Subroutinentechnik zur objektorientierten Programmierung : Teil 2 : module und abstrakte Datentypen

Autor: Marty, Rudolf

DOI: https://doi.org/10.5169/seals-903153

Nutzungsbedingungen

Die ETH-Bibliothek ist die Anbieterin der digitalisierten Zeitschriften auf E-Periodica. Sie besitzt keine Urheberrechte an den Zeitschriften und ist nicht verantwortlich für deren Inhalte. Die Rechte liegen in der Regel bei den Herausgebern beziehungsweise den externen Rechteinhabern. Das Veröffentlichen von Bildern in Print- und Online-Publikationen sowie auf Social Media-Kanälen oder Webseiten ist nur mit vorheriger Genehmigung der Rechteinhaber erlaubt. Mehr erfahren

Conditions d'utilisation

L'ETH Library est le fournisseur des revues numérisées. Elle ne détient aucun droit d'auteur sur les revues et n'est pas responsable de leur contenu. En règle générale, les droits sont détenus par les éditeurs ou les détenteurs de droits externes. La reproduction d'images dans des publications imprimées ou en ligne ainsi que sur des canaux de médias sociaux ou des sites web n'est autorisée qu'avec l'accord préalable des détenteurs des droits. En savoir plus

Terms of use

The ETH Library is the provider of the digitised journals. It does not own any copyrights to the journals and is not responsible for their content. The rights usually lie with the publishers or the external rights holders. Publishing images in print and online publications, as well as on social media channels or websites, is only permitted with the prior consent of the rights holders. Find out more

Download PDF: 13.07.2025

ETH-Bibliothek Zürich, E-Periodica, https://www.e-periodica.ch

Von der Subroutinentechnik zur objektorientierten Programmierung

Teil 2 Module und Abstrakte Datentypen

Rudolf Marty

Nachdem der Autor im ersten Teil den geschichtlichen Weg der Software-Entwicklung von der frühen Subroutinentechnik bis zu den Prozeduren- und Funktionskonzepten, wie sie beispielsweise in C und Modula-2 realisiert sind, nachgezeichnet hat, behandelt er im zweiten Teil die Technik der Module und Abstrakten Datentypen. Ein dritter und letzter Teil wird den vorläufigen Endpunkt dieser Entwicklung, die objektorientierte Programmierung zum Inhalt haben.

Après avoir tracé l'historique du développement des logiciels en partant de la technique des subroutines précoces jusqu'au concepts des fonctions et des procédures, comme ils sont réalisés par exemple dans C et Modula-2, l'auteur traite dans la deuxième partie la technique des modules et des types de données abstraites. Un point final provisoire sera constitué par la programmation orientée objet qui fera l'objet d'un troisième et dernier article.

Adresse des Autors

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

4. Ein dritter Abstraktionsschritt: Module

Eines sei zu Modulen gleich vorweg gesagt: Der Begriff *Modul* ist in der Praxis der Softwareentwicklung für sehr viele unterschiedliche Konzepte verwendet worden. Leider führt dies immer wieder zu erheblichen Begriffsverwirrungen bei der Besprechung der Module in unserem Sinn, wo wir diesen Fachbegriff recht eng fassen und für ein ganz bestimmtes Gliederungskonzept verwenden. Wir befinden uns damit aber im Einklang mit der modernen Software-Engineering-Terminologie.

Und noch eine kleine Vorbemerkung: Nach neuestem Duden wird für ein Modul im technischen Sinn, für eine Art Bauteil in Materie oder Software, das sächliche Substantiv *das* Modul (die Module) verwendet; *der* Modul (die Modulen) steht für eine Verhältniszahl.

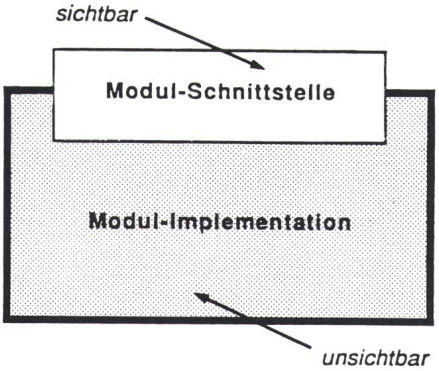


Bild 3 Ein Modul

Ein Modul besteht aus einer Schnittstelle und der Implementation. Die Schnittstelle ist vergleichbar mit der Steckerleiste der PC-Steckkarte, über die die Steuerung und der Datentransfer zwischen Prozessor und Steckkarte abgewickelt wird. Die Implementation entspricht der auf der Steckkarte installierten elektronischen Schaltung.

Ein Software-Modul kann strukturell mit einem elektronischen Bauteil wie etwa einer Steckkarte für einen PC verglichen werden (Bild 3). Ein Modul besteht aus einer Schnittstelle und der Implementation. Die Schnittstelle ist vergleichbar mit der Steckerleiste der PC-Steckkarte, über die die Steuerung und der Datentransfer zwischen Prozessor und Steckkarte abgewickelt wird. Die Implementation entspricht der auf der Steckkarte installierten elektronischen Schaltung. Für den Gebrauch der PC-Steckkarte genügt es vollauf, wenn die Definition der Schnittstelle bekannt ist. Ihr innerer Aufbau bleibt verborgen und kann im Normalbetrieb auch in keiner Art und Weise verändert werden.

Genauso verhält es sich mit Software-Modulen. Auch hier kann auf die (von aussen unsichtbare) Implementation nicht zugegriffen werden. Alle Deklarationen und Definitionen von Daten und Aktionen innerhalb eines Moduls sind demnach von der «Aussenwelt» vollständig abgekapselt, solange sie nicht in der Schnittstellenbeschreibung auftreten. Dadurch wird ein grosser konstruktiver Schritt gemacht: Durch eine saubere Modularstruktur im besprochenen Sinn entstehen Softwaresysteme, die weit besser in relativ autonome Komponenten gegliedert sind, als dies durch eine prozedurale Zergliederung möglich wäre. Diese relativ autonomen Komponenten in Form von Modulen kommunizieren miteinander ausschliesslich über definierte Schnittstellen. Es ist also stets aus dem Programmtext selbst ersichtlich, welche Kommunikationsstrukturen zwischen den einzelnen Komponenten bestehen. Implizite und versteckte Datenflüsse und Steuerungsstrukturen über Globalvariablen gibt es nicht mehr.

Fassen wir die zentralen Eigenschaften eines Moduls zusammen, bevor

```
DEFINITION MODULE Verzeichnis;
    TYPE Verz;
    PROCEDURE NewVerz ( VAR v          : Verz);
    PROCEDURE PutVerz ( v              : Verz;
                        Schluessel     : ARRAY OF CHAR;
                        Index          : INTEGER);
    PROCEDURE GetVerz ( v              : Verz;
                        Schluessel     : ARRAY OF CHAR)
                  : INTEGER;
END Verzeichnis;
```

Bild 4 Schnittstellenteil eines Moduls

wir im nächsten Abschnitt konkrete Module in der Sprache Modula-2 betrachten:

- Jedes Modul besteht aus einer von aussen «sichtbaren», das heisst verwendbaren Schnittstelle (Interface) sowie aus einer ausserhalb des Moduls selbst «unsichtbaren», das heisst unantastbaren Implementation.
- Die Modul-Schnittstelle enthält Deklarationen von Typen, Konstanten, Variablen und von Prozeduren. Im Sinne des Information Hiding wird nur gerade soviel explizit deklariert, wie für die Verwendung des Moduls nötig ist. Beispielsweise wird nur ein Typenname in der Schnittstelle deklariert, wobei die interne Struktur des Datentyps im Implementationsteil erscheint und damit gegen aussen versteckt bleibt. Entsprechend erscheint von einer Prozedur im Schnittstellenteil nur der Name, die Parametertypen und der Typ eines allfälligen Rückgabewertes.
- Die Modul-Implementation besteht im wesentlichen aus Deklarationen von Typen, Konstanten und Variablen sowie aus Prozeduren und Initialisierungscode für das Modul.
- In der Schnittstelle werden sämtliche Aussenbeziehungen eines Moduls deklariert. Die Importe und Exporte eines Moduls sind folglich im Programmcode selbst beschrieben und können somit vom Compiler auch auf korrekte Verwendung geprüft werden. Dies ist selbst dann möglich, wenn das Modul getrennt vom *Klienten* übersetzt wird. (Wir bezeichnen die Programmeinheiten, welche ein bestimmtes Modul verwenden, als Klienten dieses Moduls; damit vermeiden wir Verwirrungen mit dem Begriff Benutzer, der für ein menschliches Wesen Anwendung findet.)
- Alle in der Schnittstelle eines Moduls vorkommenden Variablen haben permanenten Charakter, sie behalten den aktuellen Wert auch nach Verlassen des Moduls. Dasselbe gilt für die innerhalb des Moduls global, das heisst nicht innerhalb Modulprozeduren deklarierten Variablen. Ein Modul ist also in seiner Struktur einem Hauptprogramm mit globalen Deklarationen und verschachtelten Prozeduren gleichgestellt.

Module in Modula-2

Das in Bild 4 und 5 dargestellte Modul implementiert einfache Verzeichnisse, die maximal 100 Paare, bestehend aus Schlüsselwert (eine Zeichenkette) und zugehörigem Indexwert, enthält. Ein solches Verzeichnis könnte beispielsweise zum assoziativen Zugriff auf eine Tabelle verwendet werden.

Man sieht, dass im Implementationsteil, versteckt von der Aussenwelt, «Buchhaltung» über die maximal 20 allozierten Verzeichnistabellen

```
IMPLEMENTATION MODULE Verzeichnis;
    TYPE    Verz        = POINTER TO VerzTabelle;
            VerzTabelle =
                    RECORD
                        Besetzt : INTEGER;
                        Tab     : ARRAY [1..100] OF RECORD;
                           Schl : ARRAY [1..20] OF CHAR;
                           Ind  : INTEGER
                        END
                    END;

    VAR     AnzTabellen : INTEGER;
            Tabelle     : ARRAY [1..20] OF Verz;


    PROCEDURE NewVerz ( VAR v : Verz);
    BEGIN
        IF AnzTabellen >= 20 THEN
            Fehler("Mehr als 20 Verzeichnisse")
        END
        Allokation einer Verzeichnistabelle
        v := Adresse der Tabelle ;
        v^.Besetzt:= 0;
        AnzTabellen := AnzTabellen + 1;
        Tabelle[AnzTabellen] := v;
    END NewVerz;

    ...

    PROCEDURE GetVerz ( v           : Verz;
                        Schluessel  : ARRAY OF CHAR)
                  : INTEGER;
    BEGIN
        suchen Schluessel in v^.Tab
        IF gefunden THEN RETURN v^.Index
        ELSE RETURN -1
        END
    END GetVerz;

    BEGIN
    (* Initialisierung des Moduls *)
    AnzTabellen := 0
END Verzeichnis;
```

Bild 5 Implementationsteil eines Moduls

geführt wird. Der Array Tabelle, der auf die Verzeichnistabellen zeigt, könnte beispielsweise dazu verwendet werden, auf Wunsch des Klienten alle Verzeichnisse wieder zu deallozieren. Hierzu müsste natürlich eine neue Modulfunktion eingeführt werden. Von dieser Änderung blieben aber, und dies ist im Sinne des Information Hiding ausserordentlich wichtig, alle Klienten unbetroffen, die die neue Funktion nicht benötigen.

Das Programmfragment (Bild 6) zeigt die Eröffnung von zwei Verzeichnissen Vz1 und Vz2, das Zufügen von Einträgen sowie die Suche nach Einträgen und die Verwendung des erhaltenen Index.

Bild 6
Verwendung eines Moduls

```
IMPORT Verzeichnis;

VAR Vz1              : Verz;      (* Name -> Index *)
    Vz2              : Verz;      (* RaumNr -> Index *)
    TelefonTab       : ARRAY [1..500] OF TelefonNr;
    i                : INTEGER;
...

NewVerz(Vz1); NewVerz(Vz2);
...
TelefonTab[53] := 752839;
PutVerz(Vz1,"Kunz",53);
PutVerz(Vz2,"34-F-12",53);
TelefonTab[32] := 8263450;
PutVerz(Vz1,"Müller",32);
PutVerz(Vz2,"05-G-18",32);
...
i := GetVerz(Vz1,"Suter");
IF i = -1 THEN
    kein Eintrag für "Suter"
ELSE
    TelNr := TelefonTab[i]
END
...
```

Ausprägungen von Modulen

Ein Modul kann sehr verschiedenartige Ausprägungen annehmen. Die folgenden fünf typischen Modularten werden in der Praxis moderner Softwareentwicklung besonders häufig angetroffen:

- Ein typisches *Bibliotheksmodul* enthält eine Sammlung von Prozeduren, die oft benötigte Funktionen implementieren und thematisch zusammengehören. Beispiele für Bibliotheksmodule sind ein Datum-Modul (Operationen auf Kalenderdaten), ein Trigonometrie-Modul (Winkelfunktionen), ein Systemschnittstellen-Modul (vereinfacht die Kommunikation mit Systemsoftware) usw.
- Ein *abstrakter Datentyp* wird durch ein Modul implementiert, das dem Klienten einen neuen Datentyp und die darauf definierten Operationen zur Verfügung stellt. Das Beispiel in Bild 4 und 5 ist ein solches Modul. Weitere Beispiele für oft benötigte abstrakte Datentypen sind unter anderem Stacks (LIFO-Struktur, Keller), Queues (FIFO-Struktur, Warteschlange), dünn besetzte Matrizen, Bäume, mehrdimensionale Tabellen, komplexe Zahlen und Koordinaten.
- Module werden oft zur *Kapselung* applikatorischer Objekte wie etwa eines Personalstamms, einer Stückliste oder eines Debitorenkontokorrentes verwendet. Damit wird das applikatorische Objekt, das physisch als ein oder mehrere Files organisiert ist, gegen aussen abgekapselt und alle Operationen durch ein einziges Modul geschleust.
- Der Modellierung *physischer Systeme* durch ein Modul kommt insbesondere in technischen Anwendungen der Informatik grosse Bedeutung zu. Beispiele für solche Module, die eine Mittlerrolle zwischen Hardwarekomponenten und dem Rest des Softwaresystems übernehmen, sind: Sensorsystem-Module, Gerätetreiber-Module, Kommunikations-Module, Anzeigetafel-Module usw.
- Neben physischen Systemen werden durch Module selbstverständlich auch *logisch-konzeptionelle Systeme* modelliert, das heisst für andere Softwarekomponenten auf hoher Abstraktionsstufe nutzbar gemacht: Grafik-Module, Datenbank-Module, Meldungsvermittlungs-Module und Dialog-Module sind ein paar wenige Beispiele dafür.

Modularer Entwurf

Dem modulorientierten Entwurf liegt nicht mehr die bei rein prozeduralen Softwaresystemen weitverbreitete Methode der schrittweisen Verfeinerung, des Top-Down-Entwurfes, zugrunde. Es wird vielmehr versucht, durch geeignete Strukturierungsformen zu einer Familie von Modulen zu kommen, die Gewähr für möglichst hohe strukturelle Einfachheit, Ausbaubarkeit und Wiederverwendbarkeit der Module bietet. Dies wird durch Mischformen von Top-down- und Bottom-up-Entwürfen erreicht.

Fragen, die mithelfen, eine vernünftige Modulstruktur (und ansatzweise später auch eine gute Objektstruktur) zu finden, sind unter anderem:

- Welche Funktionsbereiche und welche Datenobjekte gehören zusammen, und welche Dienste leisten sie dem Rest des Systems? Beispiel: Alle Dialogfunktionen werden in ein Modul zusammengefasst.
- Welches sind die physischen und logischen Systeme, die durch ein Softwaresystem betroffen werden? Wie sieht deren Funktionsschnittstelle aus? Beispiel: Ein zentrales, durch ein Modul gekapseltes Adressregister ist einer verteilten Verwaltung von Adressbeständen vorzuziehen.
- Bei welchen Softwarekomponenten sind in Zukunft Änderungen wahrscheinlich? Was muss der Rest der Software von deren interner Struktur kennen? Wie könnte eine änderungsunempfindliche Schnittstelle gestaltet sein? Beispiel: Über die Datenbank-Schnittstelle, welche die Datenbanksoftware zu Verfügung stellt, wird ein Zugriffsmodul gelegt, das die Anwendungssoftware unabhängig von der unterliegenden Datenbank macht.
- Welche Basisdienste werden in gleicher oder ähnlicher Art immer wieder benötigt? Beispiel: Ein flexibles Tabel-

lenmodul enthebt den Programmierer von der Codierung immer wieder in ähnlicher Form auftretender Tabellenstrukturen.
- Welche bekannten Schnittstellen zu Modul- und Funktionssammlungen im betrachteten Anwendungsbereich existieren zurzeit und könnten als Grundlage dienen? Beispiel: Wird ein Dialogmodul entworfen, so sollten zunächst andere Dialogschnittstellen (z.B. X-Windows [11], News [12], Macintosh Toolbox [10] o.a.) betrachtet werden, um Anhaltspunkte für Stärken und Schwächen bestimmter modularer Gliederungen zu erhalten.

Schwachstellen des Modulkonzeptes

Trotz der enormen Fortschritte einer gut modularisierten Software im Vergleich zu einem bloss in prozeduraler Hinsicht klug gegliederten Programmsystem verbleiben bei der Verwendung von Modulen immer noch zwei gravierende Mängel:
- Module sind in vielen Programmiersprachen (so auch in Modula-2) als statisches Objekt und nicht als Objekttyp definiert. Ein Modul kann deshalb nicht einmal definiert und mehrmals instantiiert, das heisst ins Leben gerufen werden. Dies wirkt sich insbesondere bei der Modellierung von abstrakten Datentypen sehr hinderlich aus, da sie mit Modulen nicht völlig transparent nachgebildet werden können. Das Beispiel in Bild 4 bis 6 zeigt den üblichen «Trick» bei der Nachbildung abstrakter Datentypen mit Modulen: Ein Datenobjekt muss explizit durch Aufruf einer Modulprozedur alloziert werden, das Datenobjekt wird durch einen sogenannten *Opaque Pointer*, einen «undurchsichtigen Zeiger» referenziert. Somit können auf diese Art nachgebildete abstrakte Datentypen nicht direkt in anderen Datenstrukturen erscheinen, nicht direkt in andere Prozesse transferiert und auch nicht direkt auf Dateien ausgeschrieben werden (man besitzt ja lediglich einen undurchsichtigen Zeiger als Referenz auf einen abstrakten Datentyp). Für solche Operationen müssten für jedes Datenobjekt spezielle Prozeduren mitdefiniert werden, die vom Klienten zum richtigen Zeitpunkt aufzurufen wären. Der Klient hat folglich abstrakte Datentypen unterschiedlich zu realen Datentypen zu behandeln. Der abstrakte Datentyp ist also nicht völlig transparent.
- Das Prinzip des Information Hiding bzw. der Kapselung von Informationen gilt nur, solange ein Modul unverändert benutzt wird. In der Praxis der Softwareentwicklung tritt jedoch oft die Situation auf, dass ein bestehendes Modul (bzw. ein vorhandener Programmteil) für eine neue Verwendung zwar beinahe, aber eben nicht perfekt passt. Man ist also zu einer Abänderung gezwungen. Diese Abänderung kann aber nur dadurch vorgenommen werden, dass man das ursprüngliche Modul kopiert und in dessen Kopie den Implementations- und eventuell den Schnittstellenteil verändert. Dies führt dazu, dass mit der Zeit eine ganze Familie von separaten, jedoch in ihrer Grundstruktur stark verwandten Modulen erzeugt wird. Ändert sich eine «Grundfeste» dieser Modulfamilie, eine Komponente, die in allen abgeleiteten Modulen identisch blieb, so muss die ganze Modulfamilie, die meist über sehr viele Programme verstreut ist, geändert werden.

Durch die beiden verbleibenden Schritte, die abstrakten Datentypen und die objektorientierte Programmierung, werden diese beiden Mängel überwunden. Insbesondere wird es möglich, saubere und transparente abstrakte Datentypen zu modellieren und, um vorerst in der Modulterminologie zu bleiben, Implementationsteile von Modulen abzuändern, ohne das Prinzip der Kapselung zu verletzen.

5. Ein vierter Abstraktionsschritt: Abstrakte Datentypen

Ein abstrakter Datentyp (im folgenden ADT genannt) umfasst die Definition eines Datentyps zusammen mit den auf diesen Datentyp zugelassenen Operationen. Dabei wird es, falls vom Entwerfer eines ADT so gewünscht und spezifiziert, unmöglich, auf die Elemente eines Datentyps zuzugreifen, es sei denn durch Anwendung der zusammen mit dem Datentyp definierten Prozeduren. Es entsteht somit ein reiner ADT, ein ADT also, dem nicht mehr die vorgängig bei der Besprechung der Module aufgeführten Nachteile anhaften.

Betrachten wir zur Illustration der grundsätzlichen Wesenszüge eines ADT das Beispiel in Bild 7, codiert in C^{++} [8], wo ein (abstrakter) Datentyp *Rechteck* definiert wird, der z.B. in einer graphischen Applikation verwendet werden kann. Was besagt in diesem Beispiel die sogenannte Klassenbeschreibung oder kurz *Klasse*? Sie besagt:
- Es soll ein ADT Rechteck gebildet werden. Ein Rechteck wird dabei beschrieben durch die zwei sogenannten *Instanzvariablen* p1 und p2, die zwei diagonal gegenüberliegende Eckpunkte des Rechtecks definieren. Wir nehmen an, der Punkt sei bereits als Koordinatenpaar (x,y) definiert worden.
- Die Klassenbeschreibung bzw. die Definition des ADT ist zergliedert in einen *privaten* und einen *öffentlichen* Teil, voneinander getrennt durch das Schlüsselwort public. In der Definition von Rechteck sind deshalb die Instanzvariablen privat, sie können also nur durch die zum ADT gehörenden Prozeduren verändert werden, nicht direkt «von aussen».
- Im öffentlichen Teil der Klasse werden fünf Prozeduren bzw. Funktionen definiert. Es erscheint jedoch nur der Funktionskopf, genau wie im Definitionsteil eines Moduls. Die Definitionen spezifizieren z.B., dass die Funktion Setze zwei Punkte als Argumente übernimmt und kein Funktionsresultat liefert (Void) oder dass die Funktion Flaeche keine Argumente hat und einen Float-Wert als Resultat zurückgibt.

Bevor ein ADT verwendet werden kann, müssen natürlich die Funktionen des ADT ausprogrammiert werden, ähnlich wie zum Gebrauch eines Moduls ja auch nicht dessen Schnittstellenbeschreibung allein ausreicht. In C^{++} geschieht die Beschreibung der

Bild 7
Eine Klassenbeschreibung in C^{++}
Es wird ein (abstrakter) Datentyp «Rechteck» definiert, der z.B.in einer graphischen Applikation verwendet werden kann.

```
class Rechteck {
    Punkt   p1,p2;
public:
    void    Setze       (Punkt Pa, Punkt Pb);
    void    Eckpunkte   (Punkt *Pa, Punkt *Pb);
    void    Verschiebe  (int DeltaX, int DeltaY);
    void    Schrumpfe   (int DeltaX, int DeltaY);
    float   Flaeche     ( );
};
```

```
void Rechteck::Setze (Punkt Pa, Punkt Pb)
{
    p1 = Pa; p2 = Pb;
}

void Rechteck::Verschiebe (int DeltaX, int DeltaY)
{
    p1.x += DeltaX; p1.y += DeltaY;
    p2.x += DeltaX; p2.y += DeltaY;
}

float Rechteck::Flaeche ( )
{
    return abs(p1.x - p2.x) * abs(p1.y - p2.y);
}
```

Bild 8
Klassenfunktionen
In C++ geschieht die Beschreibung der ADT-Funktionen in Form einer üblichen Funktionsdefinition, wobei die Verbindung zum ADT durch Voranstellen des ADT-Namens geschaffen wird.

ADT-Funktionen in Form einer üblichen Funktionsdefinition, wobei die Verbindung zum ADT durch Voranstellen des ADT-Namens geschaffen wird. In unserem Fall wird also vor jeder ADT-Funktion Rechteck das Zeichen :: stehen. Betrachten wir in Bild 8 drei der fünf Funktionsdefinitionen.

Es ist wichtig zu erkennen, dass die Klassendefinition allein noch keinerlei Datenobjekte erzeugt. Sie kann mit einer normalen Typendefinition von Modula-2, Pascal oder C verglichen werden, wo ja auch erst eine Variablendeklaration in der Erzeugung eines Datenobjektes resultiert. Wie eine Typendefinition zur Erzeugung einer beliebigen Anzahl von Variablen verwendet werden kann, ist es möglich, aus einer Klassendefinition beliebig viele *ADT-Instanzen* zu erzeugen, auch auf dynamische Art mit new. Eine ADT-Instanz ist also gewissermassen eine Variable, an die untrennbar Funktionen zur Manipulation dieser Variablen gebunden sind. Im Programmsegment von Bild 9 erzeugen wir durch die Deklaration «Rechteck r1,r2» zwei Instanzen (d.h. Vorkommen) des ADT Rechteck. Jede Instanz enthält ihre eigenen Instanzvariablen p1 und p2 (Bild 7). Nun wird auch klar, welche Instanzvariablen p1 bzw. p2 bei Aufruf einer ADT-Funktion angesprochen werden: Es sind diejenigen der ADT-Instanz, deren Name dem Funktionsaufruf vorangesetzt wird: Beim Aufruf r1.Verschiebe beispielsweise also diejenigen, die der Instanz r1 gehören.

Verglichen mit durch Module nachgebildeten abstrakten Datentypen bieten ADT doch einiges mehr an Klarheit, Sicherheit und erhöhtem Abstraktionsgrad in Beschreibung und Verwendung. Die Anwendung von ADT unterscheidet sich nicht von der Anwendung der durch die Programmiersprache vorgegebenen Typen. Damit wird ein hoher Grad an Transparenz erreicht und eine der grossen Schwachstellen von Modulen eliminiert, ohne auf der syntaktischen Ebene die Programmiersprache allzu stark zu belasten.

C++ bietet über die besprochene Class-Definition hinaus einige nützliche weitere Möglichkeiten, auf die hier nicht näher eingegangen wird: Für eine Class können gesondert Initialisierungs- und Terminierungsfunktionen angegeben werden. Diese werden automatisch für jede Class-Instanz dann aufgerufen, wenn diese ins Leben gerufen bzw. eliminiert wird: für automatische Klasseninstanzen bei Eintritt bzw. Verlassen des Gültigkeitsbereiches der Klasseninstanz, für globale Klasseninstanzen bei Programmstart bzw. Programmende. Ausserdem bietet C++ die Möglichkeit, Operatoren wie +, −, *, <, >, =, ==, ja selbst Indexklammern [] auch als Funktionsbezeichner für ADT zu verwenden. Damit wird die Verwendung von ADT vollständig transparent, beispielsweise etwa bei der Anwendung eines ADT für rationale oder komplexe Zahlen, die auf diese Art und Weise wie die vordefinierten ganzen oder reellen Zahlen verwendet werden können.

```
Rechteck        r1, r2;
Punkt           pt1, pt2, pt3, pt4;

...
r1.Setze(pt1,pt2);
r1.Verschiebe(-12,10);
r2.Setze(pt3,pt4);
printf("Fläche von r1: %8.3f", r1.Flaeche() );
```

Bild 9
Instantiierung und Verwendung von ADT

Schwachstellen abstrakter Datentypen

ADT können wohl zur Definition beliebiger Datenstrukturen angewendet werden. Es ist jedoch nicht möglich, einen bestehenden ADT auf einfache und flexible Art und Weise zu ergänzen, ohne das Prinzip des Information Hiding zu verletzen.

Verdeutlichen wir uns dies an einem Beispiel: Nehmen wir an, wir hätten einen ADT Rechteck wie er in den Bildern 7 ff. dargestellt wurde. Nun ergibt sich der Bedarf nach einem neuen ADT, der Rechtecke mit abgerundeten Ecken implementieren soll.

Zu den zwei Eckpunkten, die ein Rechteck definieren, kommt der Radius der Eckbogen als dritte Instanzvariable hinzu. Wie würden wir diesen neuen ADT realisieren? Wir definieren einen ADT RechteckRund, der als Instanzvariable ein (privates) Rechteck und eine (öffentliche) int Variable zur Definition des Eckradius hat. Die Funktionen von RechteckRund bestehen in den meisten Fällen lediglich aus einem Aufruf der entsprechenden Funktion des ADT Rechteck. Nur die Funktionen Setze und Flaeche enthalten veränderten Code. Trotzdem muss jede Funktion des ADT Rechteck in der Definition des ADT RechteckRund wiederholt werden, selbst wenn sie nur die gleichnamige Funktion des ADT Rechteck aufruft (siehe z.B. Verschiebe in Bild 10).

Strukturell ist der ADT RechteckRund nichts anderes als eine kleine Erweiterung seines «Ahnen-ADT» Rechteck. In der Praxis der Softwareentwicklung entstehen ganze Ahnenhierarchien von ADT oder ADT-ähnlichen Softwarekomponenten. Sehr oft

```
class RechteckRund {
    Rechteck    Huellrechteck;
public:
    int     Eckradius;
    void    Setze          (Punkt Pa, Punkt Pb, int Radius);
    void    Eckpunkte      (Punkt *Pa, Punkt *Pb);
    void    Verschiebe     (int DeltaX, int DeltaY);
    void    Schrumpfe      (int DeltaX, int DeltaY);
    float   Flaeche        ( );
}

...

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

void RechteckRund::Verschiebe (int DeltaX, int DeltaY)
{
    Huellrechteck.Verschiebe(DeltaX,DeltaY);
}

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

Bild 10 Abgeleiteter ADT

Strukturell ist der ADT RechteckRund nichts anderes als eine kleine Erweiterung seines «Ahnen-ADT» Rechteck.

geschieht dies sogar in weit unhomogenerer Form, indem von einem Ahnen-ADT der ganze Quellcode übernommen und die Kopie sodann direkt modifiziert wird. Gemeinsamkeiten von ADT werden dadurch repliziert und diffundieren allmählich; man erkennt die zugrundeliegenden Basisstrukturen nicht mehr. Dies führt zu Problemen im Zuge der Software erweiterung und -wartung.

Betrachten wir hierzu eine zwar hypothetische, in dieser Art im Praxisleben jedoch recht oft anzutreffende Situation: Für ein neues Software-Projekt sei unser ADT Rechteck samt aller davon abgeleiteten ADT, also auch der ADT RechteckRund, zu verwenden. Statt kartesischer Koordinaten sollen jedoch Polarkoordinaten benutzt werden. Die Funktion Verschiebe übernimmt natürlich neu nicht mehr eine Verschiebungsdistanz in kartesischen Grössen (δx und δy), sondern in polaren Grössen (Verschiebungsrichtung und -distanz). Nehmen wir weiter an, der Verschiebungswinkel sei ein reeller (Float-)Wert.

Das neue Koordinatensystem induziert Modifikationen nicht nur am ADT Rechteck, sondern auch an allen hievon abgeleiteten ADT, obschon in den abgeleiteten ADT nicht mehr direkt auf Koordinaten zugegriffen wird. Wegen der expliziten Weiterverwendung von ADT-Funktionen in abgeleiteten ADT wird eine ganze Lawine von Änderungen nötig. In jeder abgeleiteten Klasse muss z.B. die Funktion Verschiebe nachgetragen werden, da sich ein Parameter ändert. Eine ähnliche Situation ergibt sich, wenn ein ADT eine neue Funktion zugeteilt erhält, beispielsweise für unsere Rechtecke eine Funktion Skaliere, die ein Rechteck um einen gegebenen Faktor vergrössert bzw. verkleinert. Obschon diese Funktion nur einen ADT betrifft, nämlich Rechteck, muss in allen abgeleiteten ADT eine entsprechende Funktion nachgetragen werden, lediglich um jeweils die Funktion Skaliere des Ahnen-ADT aufzurufen.

Die Notwendigkeit all dieser Änderungen ist dadurch gegeben, dass die Weiterverwendungs- bzw. Vererbungshierarchie von ADT nirgends explizit festgehalten wurde. Statt einer «Vererbungsdeklaration» im Programm, der ADT RechteckRund sei abgeleitet vom ADT Rechteck, rufen wir aus den Funktionen von RechteckRund explizit die Funktionen von Rechteck auf. Anders betrachtet übernimmt es der Programmierer, durch explizite Aufrufe von Funktionen eines Ahnen-ADT die Vererbungshierarchie im Programmcode auszuformulieren, statt diese Aufgabe dem Compiler zu übertragen. Dieser könnte, explizite Vererbungsdeklarationen vorausgesetzt, sehr wohl selbst den Aufruf der gleichnamigen Funktion des Ahnen-ADT erzeugen, falls ein ADT die Funktion selbst nicht explizit deklariert. Diese Überlegung führt uns direkt zum letzten Abstraktionsschritt, der objektorientierten Programmierung.

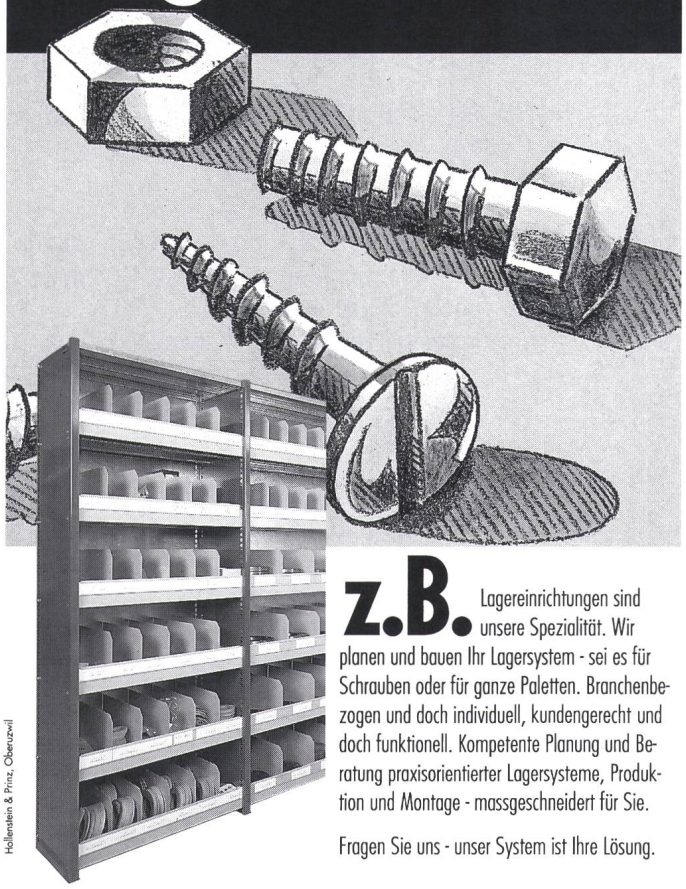
(Teil 3 folgt im Heft 21/90)

Literatur

[11] *R.W. Scheiffer, J. Gettys:* The X Window System, ACM Transactions on Graphics 5 (1986) 2.

[12] *Sun Microsystems Inc.:* News Technical Overview, 1987.

Lagern: Sie sagen was, wir sagen wie!



z.B. Lagereinrichtungen sind unsere Spezialität. Wir planen und bauen Ihr Lagersystem - sei es für Schrauben oder für ganze Paletten. Branchenbezogen und doch individuell, kundengerecht und doch funktionell. Kompetente Planung und Beratung praxisorientierter Lagersysteme, Produktion und Montage - massgeschneidert für Sie.

Fragen Sie uns - unser System ist Ihre Lösung.

Hollenstein & Prinz, Oberuzwil

COUPON

☐ Hilfe, schaffen Sie Ordnung in das Lager - beraten Sie uns über Ihre Lagersysteme

☐ Senden Sie uns Prospekt-Unterlagen

☐ Senden Sie uns gratis und unverbindlich das Büchlein "Lager-Tips"

Name: Firma: SEV

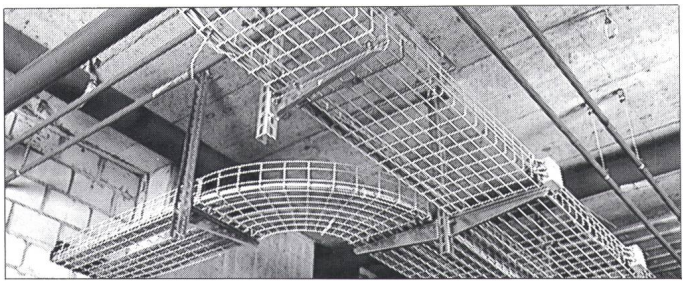
Adresse:

PLZ: Ort:

WEHRLE SYSTEM

Wilerstrasse
CH-9230 Flawil 2
Telefon 071 83 31 11
Telefax 071 83 30 04

Alles in Ordnung!



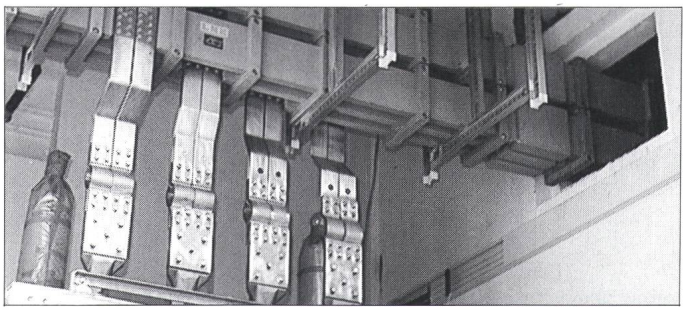
Chemins de câbles LANZ

Chemins de câbles Echelles à câbles
Chemins de câbles à grilles Canaux G

Pour la pose de câbles dans des bureaux et bâtiments administratifs, halles de dépôts, fabriques, installations de protection civile, etc.

- 3 exécutions avec éléments de montage légers, standard et lourds pour un rendement maximum
- zingué, avec en plus une projection de poudre époxi ou entièrement en polyester offrant une protection maximale contre la corrosion
- production propre avec un service optimal

Téléphonez à **lanz oensingen sa 062/78 21 21** ou à votre grossiste en électricité pour tout conseil, offre, livraison rapide à prix avantageux



Canalisations électriques LANZ BETOBAR

Pour la distribution de courant de 380 à 6000 A dans les bureaux, locaux artisanaux et industriels. Degré de protection IP 68.7

- compacts p.ex. 1940 A seulement 100×160 mm mesures extérieures
- montage exact au centimètre près dans les armoires de commande, zones montantes, aux parois et plafonds permettant une meilleure utilisation de place
- protection maximum des personnes, haute résistance aux courts-circuits — ne nécessitant pas d'entretien

LANZ planifie, livre et installe les canalisations électriques BETOBAR.

Les produits LANZ m'intéressent! Prière d'envoyer la documentation pour:

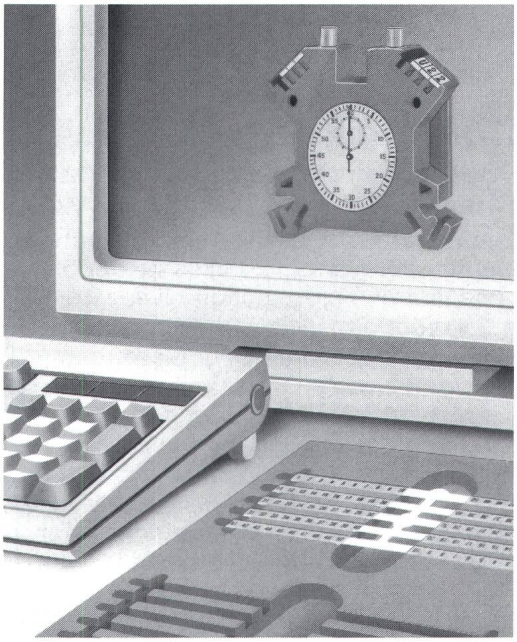
☐ Canalisations électriques d'éclairage
☐ Caniveaux à lampes LANZ
☐ Chemins de câbles LANZ
☐ LANZ Canaux G
☐ LANZ MULTIFIX
☐ Faux-planchers LANZ pour bureaux
☐ Faux-planchers LANZ pour charges lourdes
☐ Canaux d'allèges LANZ
☐ Câble plat LANZ pour courant, données et téléphone
☐ Pourriez-vous me/nous rendre visite? Avec préavis!

Nom, adresse:

lanz oensingen sa
CH-4702 Oensingen · téléphone 062 78 21 21

EEI-7

Neuer Klemmenbeschriftungs – Service



System RB–Script:
Markierung von kompletten Bezeichnungskarten zu Woertz–Klemmen: Neu mit unserem eigens entwickelten Computer–Beschriftungssystem RB–Script. Die Beschriftungen können nach Ihren Wünschen an jede Installation individuell angepasst werden.
Lieferung der nach Kundenwunsch beschrifteten Karten innert 48 Stunden: Der Woertz–Schnellservice.
So sparen Sie noch mehr Zeit bei Ihren elektrischen Installationen!

System RB–Script auch bei Ihnen!
Benötigen Sie häufig individuell beschriftete Woertz–Klemmen?
Wir bieten Ihnen die komplette, anwenderfreundliche Software inklusive Plotter und Kartenhalterungsplatten an. Das Programm ist auf jedem Personal–Computer (AT oder kompatibel) lauffähig.
Fordern Sie noch heute detaillierte Unterlagen an!

woertz
Elektrotechnische Artikel
Installationssysteme
Hofackerstrasse 47, 4132 Muttenz 1, Schweiz, Tel. 061 / 61 36 36

Service stark!

IHR PROBLEM:
Netzbelastung
erfassen und bearbeiten

DIE LOESUNG:
MED Gerät
ERSTELLUNG UND AUSKUNFT:
PANENSA
SOCIETE ANONYME
LES VERNETS - CH 2035 CORCELLES / NE
TEL (038) 31 34 34 - FAX (038) 31 69 62

Gitter-Kabelbahnen

519

ab Lager lieferbar

Swisstech 90
Stand 751 · Halle 311

Feuerverzinkt,
tauchfeuerverzinkt,
oder plastifiziert

Bruno Winterhalter AG
Industrieprodukte Tel. 01-830 50 30
Birgistr. 10, 8304 Wallisellen, Fax 01-830 79 52