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: 13

Artikel: Von der Subroutinentechnik zur objektorientierten Programmierung : Teil 1 : Subroutinen und Prozeduren

Autor: Marty, Rudolf

DOI: https://doi.org/10.5169/seals-903135

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: 11.07.2025

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

Von der Subroutinentechnik zur objektorientierten Programmierung

Teil 1 Subroutinen und Prozeduren

Rudolf Marty

Objektorientierte Programmierung ist eines der Schlagworte, auf das selbst der Software-Laie immer wieder stösst. Ausgehend vom guten alten Subroutinenkonzept zeigt der Autor in drei Folgeaufsätzen, wie die Softwareentwicklung durch Überwindung von Unvollkommenheiten der jeweils vorhergehenden Stufe fast zwangsläufig zu einer objektorientierten Softwarekonstruktion führt, wobei mit Unvollkommenheit vor allem eine schlechte Wiederverwendbarkeit und ungenügende Datensicherheit gemeint sind.

La programmation orientée objet est un de ces slogans auquel se heurte même le profane en logiciel fréquemment. Se basant sur le bon vieux concept des sous-programmes, l'auteur montre dans une succession de trois articles comment le développement de logiciels – du fait que l'on surmonte les imperfections de chaque étape précédente – mène presque inévitablement à une construction logicielle orientée objet; par imperfections on entend surtout une mauvaise aptitude à la réutilisation et une sécurité de données insuffisante.

Adresse des Autors

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

Das Gebiet des Software Engineering hat sich einhergehend mit der Aufgliederung und methodischen Verbreiterung der Informatik enorm stark ausgeweitet. Nach Pomberger verstehen wir heute unter Software Engineering «... die praktische Anwendung wissenschaftlicher Erkenntnisse für die wirtschaftliche Herstellung und den wirtschaftlichen Einsatz zuverlässiger und effizienter Software» [1]. Im weiten Feld der Interessenströmungen, Forschungsarbeiten und Entwicklungsvorhaben innerhalb des Software Engineering nimmt die Frage der konstruktiven Gliederung von Programmen eine zentrale Rolle ein: Seit den Anfängen der Softwareentwicklung beschäftigt uns die Frage, durch welche Entwurfsmethoden und Gliederungsstrukturen wir das Ziel hochwertiger, kostengünstiger Software am besten erreichen können. Stichworte zu Entwicklungen in diesem Bereich sind unter vielen anderen: Makro-Konstrukte, Prozeduren/Funktionen, Blockstruktur, strukturierte Programmierung, Modularität, abstrakte Datentypen und Information Hiding.

Objektorientierte Software ist ein in jüngerer Zeit auf sehr viel Interesse stossendes Konstruktionsprinzip für Computerprogramme. Auf wenige Wesenszüge reduziert geht es dabei darum, Software durch Reproduktionen von Objektbeschreibungen zu erzeugen. Eine Objektbeschreibung enthält Definitionen von Daten zusammen mit einer Spezifikation der auf diese Daten anwendbaren Aktionen. Im Unterschied zu der modularen Programmierung, wie sie zum Beispiel Modula-2 [2] zugrunde liegt, sind Objektbeschreibungen lediglich eine Art Typenbeschreibung. Sie bilden keine real existierenden Konstrukte wie etwa ein Modul im Sinne von Modula-2. Erst durch Instantiierung einer Objektbeschreibung wird ein Objekt erzeugt, ähnlich wie erst durch die Deklaration einer Variablen in Modula-2 oder Pascal [3] ein Datenobjekt vom angegebenen Typ erzeugt wird, nicht bereits durch die Typendefinition.

Ausserordentlich wichtig, ja geradezu zentral für den Erfolg der objektorientierten Softwareentwicklung ist die Tatsache, dass diese Beschreibungen nicht jeweils eine vollständige Definition des Objektverhaltens, also all seiner Daten und Aktionen beinhaltet. Die Objektbeschreibungen sind in einer Hierarchie angeordnet, so dass auf jeder Hierarchiestufe nur jeweils diejenigen Daten und Aktionen spezifiziert werden, die in übergeordneten Objektbeschreibungen nicht bereits definiert wurden. Dabei werden die Modifikationen an Daten und Aktionen beschrieben, ohne die übergeordneten Objektbeschreibungen zu verändern. Damit steht die objektorientierte Softwareentwicklung im krassen Gegensatz zu der modularen Programmierung, bei der eine Wiederverwendung eines Moduls nur dann ohne Korrekturen an dessen Implementationsteil möglich ist, wenn im Rahmen der Wiederverwendung das Modulverhalten, das heisst die semantische und die syntaktische Schnittstelle zum Modul, unverändert passt.

Durch die Möglichkeit der inkrementellen Ergänzung und Anpassung von Objektbeschreibungen, ohne dabei deren Code anzutasten, eröffnen sich der Wiederverwendbarkeit von Softwarekomponenten neue Dimensionen. Wie zu zeigen sein wird, hat dies profunde Auswirkungen auf die Organisation von Softwaresystemen als Ganzem. Anstelle der heute weitverbreiteten Bibliotheken von wiederverwendbaren Funktionsmodulen, de-

ren Komponenten bei der Konstruktion von Software verwendet werden, entstehen Bibliotheken von Objektbeschreibungshierarchien, die eigentliche Applikationsrahmen bilden. Beispiele für solche Applikationsrahmen sind MacApp [4] und ET++ [5].

Statt das Gerüst einer Applikation immer wieder von neuem zu bauen und sodann Bibliotheksmodule einzubinden, wird in der objektorientierten Softwareentwicklung von vorgefertigten Applikationen und Applikationsteilen ausgegangen, die an die konkreten Bedürfnisse angepasst werden, ohne diese übernommenen Teile jedoch zu verändern. Damit können später Änderungen an den vorgefertigten Applikationsteilen vorgenommen werden, die völlig transparent und ohne weiteres Zutun auch auf alle hievon abgeleiteten Applikationen durchdringen. Eine Situation, die im klassischen Schema der Wiederverwendung von Programmteilen undenkbar ist, werden doch hierbei die Programmteile typischerweise mit einem Editor auf Quelltextebene physisch kopiert und abgeändert.

Objektorientierte Softwareentwicklung ist also mehr als nur ein neues Programmierparadigma, mehr als lediglich eine Erweiterung prozeduraler und modularer Programmiersprachen. Durch die Wiederverwendung von bestehenden Komponenten unter Vornahme von Abänderungen, ohne dabei das Prinzip des *Information Hiding* [6] zu verletzen, wird die Softwareentwicklung auf neue Fundamente gebaut. Möglich wird dies zunächst durch die Verwendung objektorientierter Programmiersprachen. Software mit einer solchen zu implementieren, führt jedoch noch nicht automatisch zu einem objektorientierten Softwaresystem. Erst durch Befolgung von objektorientierten Konstruktionsprinzipien wird dieses Ziel erreicht. Wie auf der ganzen Breite des Software Engineering, so müssen auch in der objektorientierten Softwareentwicklung Methodik und Werkzeuge aufeinander abgestimmt sein: Ohne Verwendung einer objektorientierten Programmiersprache ist keine sinnvolle objektorientierte Konstruktion möglich, ohne objektorientierte Konstruktionslehre enstehen anderseits selbst bei Verwendung einer objektorientierten Programmiersprache keine befriedigenden objektorientierten Systeme.

Die folgenden Ausführungen sind als Einstiegslektüre in die objektorientierte Softwareentwicklung gedacht. Sie führen über fünf markante Gliederungsstufen prozeduraler Programmiersprachen zu einem Verständnis der Grundfesten objektorientierter Programmierung. Die Diskussion der einzelnen Gliederungsstufen ist bewusst relativ ausführlich gehalten, um auch einem im Umgang mit prozeduralen und modularen Programmiertechniken nicht ausgebildeten Praktiker eine methodisch saubere, schrittweise Hinführung zum Konzept objektorientierter Software zu bieten. Informatiker mit entsprechenden Vorkenntnissen können ohne weiteres direkt zu den Ausführungen über die Schwachstellen der Prozeduren in Kapitel 3 (Seite 9) oder die Schwachstellen der Module in Kapitel 4 (im 2. Teil) springen.

1. Am Anfang war die Maschine

Die Computer der Frühzeit elektronischer Informationsverarbeitung waren durch sehr knapp bemessene Speicher- und Prozessorressourcen gekennzeichnet. Es galt, der «Rechenmaschine» die zu bearbeitenden Vorgänge möglichst kompakt und auf deren Maschinenstruktur angepasst zu beschreiben. Eine wichtige Konsequenz dieser Art der Programmierung war die Modellierung der Denkvorgänge im Kopf des Software-Ingenieurs: Er dachte in der Begriffswelt und in Konzepten der Maschine, er gab durch sein Programm der Maschine Schritt für Schritt vor, was sie zu tun habe. Der gute Programmierer formulierte nicht nur, nein, er dachte auch in der «Maschinenwelt». Seine Qualifikation war insbesondere davon geprägt, wie gut er die Eigenheiten der Maschine kannte, wie füllig seine Programmier-Trickkiste war und wie gut er Speicherauszüge lesen konnte.

Die Abstraktionslücke zwischen Problem und Programm war damals enorm gross, und die Abbildung eines Problems in ein einigermassen effizientes Programm erforderte profunde Maschinenkenntnisse. Eine kleinere Linderung dieser Situation erfolgte durch das Konzept der sogenannten *Subroutinen* oder *Unterprogramme*, die wir nun als ersten Abstraktionsschritt auf dem Wege von der unstrukturierten, maschinenbezogenen Programmierung zu objektorientierter Software betrachten.

2. Ein erster Abstraktionsschritt: Subroutinen

Die Wurzeln der Subroutinen- oder Unterprogrammtechnik gehen auf das Bestreben zurück, in einem Programm mehrmals vorkommende Codesequenzen auszugliedern, an einem Platz zusammenzufassen und sodann durch eine geeignete Instruktion (meist *call subroutine* oder ähnlich genannt) aufzurufen.

Informationskapselung, Abstraktion

Neben der blossen Platzersparnis, die im Rahmen unserer Überlegungen an dieser Stelle nicht interessiert, wurde durch das Subroutinenkonzept auch eine Verbesserung der Programmstruktur und eine Anhebung des Abstraktionsniveaus der Programmierung erreicht: Die konkrete Instruktionsfolge zur Ausführung einer bestimmten Operation auf bestimmte Daten bleibt durch den Subroutinenaufruf verborgen. Die Subroutine ist also eine Art abstrakte, «höhere» Operation. Sie kapselt eine Instruktionsfolge gegen aussen ab, der Programmierer braucht sich bei deren Aufruf nicht mehr um Instruktionsdetails zu kümmern.

Ein Problem tritt auf, wenn die gewünschte abstrakte Operation nicht stets dieselbe ist, sondern in einem gewissen Grad parametrisiert werden muss. Ein Beispiel hierfür ist eine Subroutine zur Berechnung des Barwertes einer Rente, die mit Zinssatz, Rentenbetrag und Laufzeit der Rente parametrisiert werden muss. In Ermangelung eines Parameterkonzeptes, das uns die Subroutinen ja nicht zur Verfügung stellen, werden globale Variablen zu Parametern «missbraucht», wobei diese Variablenverwendung zur Steuerung der Subroutine im besten Fall in einem ihr vorangestellen Kommentar erklärt wird. Im schlechteren und oft anzutreffenden Fall wird eine solche «Pseudo-Parametrisierung» nur durch Analyse des Subroutinencodes deutlich.

Neben der Steuerung einer abstrakten Operation über Globalvariablen verbleibt natürlich noch das Problem des Datenaustausches zwischen dem aufrufenden Programmteil und der Subroutine. Die Subroutinentechnik bietet uns auch in diesem Bereich keine spezielle Unterstützung durch zweckmässige Konstruktionen. Aus der Beschreibung der Subroutine

(Kommentar innerhalb des Programmcodes oder separate Dokumentation) geht hervor, auf welchen globalen Datenobjekten eine Subroutine ihre Funktionen ausführt, welche sie modifiziert und in welchen sie Resultate ablegt. Globale Datenobjekte übernehmen eine Art Transferrolle, treten also in eine Art Import-Export-Schnittstellenrolle, ohne dass sie syntaktisch irgendwie besonders bezeichnet würden. Ihre spezielle Rolle tritt direkt nur durch die Analyse des Programmcodes oder indirekt durch Kommentare und Dokumentation hervor.

Zur Erfüllung ihrer Aufgabe benötigt eine Subroutine in praktisch jedem Fall eine Anzahl von sogenannten *Arbeitsvariablen.* Beispiele hierfür sind Schleifenvariablen, Indizes, Temporärvariablen zur Speicherung von Zwischenresultaten usw. Da die Subroutinentechnik keine Möglichkeit zur lokalen, das heisst an die Subroutine gebundenen Deklaration von Variablen bietet, werden die Arbeitsvariablen aller Subroutinen im globalen Namensbereich gehalten. Sie sind folglich allen Programmteilen bekannt und können überall verwendet werden. Eine sehr unliebsame Folge davon ist die Verunmöglichung einer konsequenten Abkapselung der subroutineninternen Datenobjekte gegen missbräuchliche Verwendung von aussen.

Subroutinen in höheren Programmiersprachen

Das Subroutinenkonzept, das in seiner Grundstruktur aus der Programmierung in Maschinensprache bzw. Assembler stammt, wurde in mehreren höheren Programmiersprachen zwar syntaktisch etwas angehoben, in seiner methodischen Grundstruktur jedoch unverändert übernommen. Es seien hier nur zwei repräsentative Beispiele genannt: Cobol und Basic. Eine Cobol-Subroutine wird gebildet durch eine Sequenz von Paragraphen, die durch PERFORM-Anweisungen aufgerufen werden, in Basic werden Sequenzen von Anweisungen durch GOSUB als Subroutine ausgeführt.

Auch in diesen sogenannten «höheren» Programmiersprachen verbleiben natürlich die Kernprobleme der konventionellen Subroutinenstrukturen wie mangelhafte Abstraktion, Interdependenzen über externe Variablen und das Fehlen von lokalen Arbeitsvariablen. Lediglich bei Verwendung von externen Subroutinen (in Cobol mit einer CALL-Anweisung aufgerufen) und bei Basic-Funktionen werden die Probleme ein wenig in Richtung eines Prozedurkonzeptes gelindert.

Schwachstellen des Subroutinenkonzeptes

Den Zielen einer möglichst in sich abgeschlossenen Abkapselung von Programmfunktionen und den syntaktisch klar formulierten Beziehungen zwischen einer solcherart abgekapselten Funktion und ihrer Umwelt wird bei Subroutinen in keiner Art und Weise Rechnung getragen. Konsequenzen sind unter anderem:

- mangelhafte funktionale Abstraktion, insbesondere wegen des Fehlens eines sauberen Parametermechanismus und der Möglichkeit zur Deklaration von lokalen Arbeitsvariablen
- schwer lesbare und modifizierbare Programme sowie unerwünschte Nebeneffekte und Fehlersituationen durch versteckte funktionale und datenmässige Beziehungen zwischen Programmteilen
- geringe Förderung der Wiederverwendbarkeit von Programmkomponenten als Folge der «Verwebung» von Subroutinen über die Verwendung von Globalvariablen.

3. Ein zweiter Abstraktionsschritt: Prozeduren, Funktionen

In den sechziger Jahren entstand, hauptsächlich durch die Entwicklung von Algol geprägt, eine prägnante Umorientierung der Denkweise über algorithmische Prozesse. Die herkömmliche Sicht, ein Programm weise den Computer «Schritt für Schritt» an, was er zu tun habe, wurde abgelöst durch eine mathematisch-abstrakte Denkweise: Ein Programm beschreibt ein Problem in einer höheren, mathematischen Notation; die Übersetzersoftware und die Hardware sorgen sodann für die Ausführung der algorithmischen Problembeschreibung, des Programmes also.

Die Mathematisierung der Softwareentwicklung hatte selbstverständlich einen grossen Einfluss auf die Evolution der Programmiersprachen und insbesondere auch auf die syntaktische und semantische Ausgestaltung ihrer Konstrukte zur Strukturierung und Modularisierung von Softwarekomponenten. Ein Resultat war die Verfeinerung des Konzeptes parametrisierter Subroutinen, wie sie von Fortran bekannt waren, in ein eigentliches Prozedurkonzept.

Eine *Funktion* ist eine Prozedur, die einen Rückgabewert erzeugt. Leider werden die Begriffe Prozedur und Funktion nicht in allen Programmiersprachen konsistent verwendet. Manche kennen nur Prozeduren (z.B. Modula-2) oder ausschliesslich Funktionen (z.B. C [7]); in diesen Sprachen wird dann zwischen Prozeduren bzw. Funktionen mit Rückgabewert und solchen ohne unterschieden. Wo dies nicht besonders vermerkt ist, meinen wir mit Prozedur jeweils auch solche, die einen Wert zurückgeben.

Das Prozedurkonzept ist Ausgangspunkt und Fundament für die Beschreitung des Weges hin zu einer entwicklungstechnisch adäquaten Problemformulierung auf abstraktem Niveau. Alle weiteren, im Laufe der schrittweisen Hinführung zur objektorientierten Softwaregliederung besprochenen Gliederungsstrukturen von Software bauen auf dem Prozedurkonzept auf. Um so erstaunlicher ist die Tatsache, dass in der Praxis solche Programmiersprachen die weiteste Verbreitung haben (und wohl auch noch einige Zeit haben werden), die noch nicht einmal den Schritt von den Subroutinenstrukturen zum Prozedurkonzept vollzogen haben.

Welches sind die wichtigsten methodisch-konzeptionellen Fortschritte der Prozedurtechnik gegenüber der Subroutinentechnik?

- Eine Prozedur implementiert eine bestimmte (abstrakte) Funktion. Ihre Aktivierung (bzw. ihr Aufruf) nimmt im Programm die Form einer mathematischen Funktionsbezeichnung an (z.B. Lies(KdNr) oder arctan(y)). Ein Prozeduraufruf kann deshalb im Gegensatz zu Subroutinen überall da vorkommen, wo ein Wert benötigt wird, also beispielsweise auch direkt in Ausdrücken wie sqrt(sin(x)/cos(x)). Eine Prozedur ist ein sauberes mathematisches Konzept (mindestens solange sie keine Nebeneffekte hat), wohingegen die Subroutine ein Maschinenkonzept ist.
- Eine Prozedur kann parametrisiert werden. Dadurch entsteht eine klare, syntaktisch und beschränkt auch semantisch überprüfbare Kommunikationsschnittstelle zwischen aufrufendem Programmteil und der Prozedur. Dasselbe gilt sinngemäss für Rückgabewerte einer Prozedur bzw. einer Funktion. Die bei den Subroutinen besprochene, fehleranfällige und obsku-

```
...
var a,b,c    : real;
    i        : integer;
    Top      : integer;
...

procedure LadeElement (x,y: real; n: integer);
begin
    if n > MaxElt then
        Stop("Überlauf von <Tab>")
    else begin
        Tab[n] := sqrt(sqr(x) + sqr(y));
        if n > Top then Top := n
    end
end;

function Annuität (Kapital,ZSatz: real;
                   Jahre: integer): real;
    var r,rn: real;
begin
    r := 1 + ZSatz/100;
    rn := exp(r,Jahre);  /* errechnet r hoch Jahre */
    Annuität := Kapital * rn * (r-1) / (rn - 1)
end;

...
Top := 0;
...
LadeElement(a,b,39);
...
writeln("jährliche Abzahlung = ", Annuität(a,c,24):8:4);
...
```

Bild 1 Prozeduren und Funktionen in Pascal

re Kommunikation über globale Variablen entfällt.

- Eine Prozedur hat einen eigenen Namensbereich, es können also innerhalb ihrer Grenzen Objekte deklariert werden, die ausserhalb der Prozedur unbekannt und nicht zugreifbar sind. Im Unterschied zu Subroutinen, die keinen eigenen Namensbereich aufweisen, werden Arbeitsvariablen innerhalb der Prozeduren abgekapselt.

Prozeduren und Funktionen in Pascal und C

Betrachten wir zur Illustration die syntaktische Struktur von Prozeduren bzw. Funktionen in Pascal und in C. Es handelt sich um dasselbe Programmgerüst mit je einer Prozedur mit und einer ohne Rückgabewert (Bild 1 und 2).

Da in der Folge oft C^{++} [8]-Programmfragmente erscheinen werden, sei für den C-Neuling gleich hier auf ein paar von manchen anderen Sprachen abweichende syntaktische Eigenheiten der Sprache C hingewiesen:

- Den begin-end-Paaren, wie sie in chen in C die geschweiften Klammern {}.
- In Deklarationen steht der Typ stets vor den Namen der deklarierten Objekte:

```
float a,b
```

deklariert also a und b als reelle (Gleitkomma-)Variablen,

```
int F(...)
```

definiert den Rückgabewert der Funktion F als integer.

- Das Symbol «=» ist in C das Zuweisungssymbol, der relationale Operator «gleich» nimmt die Form «==» an.

Die prozedurale Grundstruktur gleicht sich in den meisten Programmiersprachen, die ein Prozedurkonzept unterstützen. Immerhin gibt es zwischen Pascal und C zwei bemerkenswerte Unterschiede:

- Pascal kennt keine sogenannten statischen Variablen. Deshalb nehmen alle in einer Prozedur oder Funktion deklarierten Variablen bei jedem Eintritt undefinierte Werte an. Muss sich eine Pascal-Prozedur oder -Funktion gewisse Daten merken (also einen Zustand konservieren), so ist sie gezwungen, eine globale Variable dazu zu verwenden (siehe Top in Bild 1). Dies ist ein grosser Nachteil, da dies zu einer Durchlöcherung des Prinzips der Informationskapselung führt und potentielle Fehlersituationen schafft. In C behalten die als static deklarierten Variablen ihren Wert über Aufrufe hinweg (siehe Top in Bild 2).
- In C, das immer von Funktionen spricht, selbst bei Prozeduren ohne Rückgabewert wie LadeElement in Bild 2, können Funktionen lexikalisch nicht ineinander verschachtelt werden. Es gibt daher lediglich zwei hierarchische Namensbereiche: den globalen Namensbereich und den Namensbereich je Funktion.

Schrittweise Verfeinerung

Das Prinzip der schrittweisen Verfeinerung (Stepwise Refinement) wurde von Wirth als algorithmische Konstruktionslehre propagiert [9]. Sie ist eine im Rahmen prozeduralen Aufbaus von Software sehr typische und immer noch aktuelle Methode: Man beginnt bei der Konzeption eines Programmes oder eines Programmteils auf einer hohen Abstraktionsstufe und definiert dieser Stufe entsprechende, hypothetische Funktionen. Sodann zergliedert man diese hypothetischen (abstrakten) Funktionen in eine Reihe weniger abstrakte Funktionen, die man im nächsten Schritt einzeln betrachtet. So fährt man weiter, bis man in Top-Down-Manier auf der Stufe des Programmcodes angelangt ist. Durch hierarchische Komposition der Einzelfunktionen integriert man die Software schliesslich bottom-up.

Die schrittweise Verfeinerung ist eine ausserordentlich stark *funktionsorientierte* Entwurfsmethode, die nur sehr beschränkt zu Lokalität von Daten führt. Gewiss, die Arbeitsvariablen und die Parameterstrukturen der dabei entworfenen Funktionen genügen dem Prinzip der Informationskapselung, der Lokalität von Daten also. Was wir jedoch durch schrittweise Verfeinerung nicht erhalten, sind Softwarestrukturen höherer Ordnung, wie die nachfolgend besprochenen Module oder gar Klassenhierarchien. Es ist für das Verständnis der Prinzipien objektorientierter Softwareentwicklung wichtig, das Prinzip der schrittweisen Verfeinerung als Entwurfsmethode in

```
...
float       a,b,c;
int         i;
...

LadeElement (x,y,n)
    float       x,y;
    int         n;
{
    static int  Top = 0;

    if (n > MaxElt)
        Stop("Überlauf von <Tab>");
    else {
        Tab[n] = sqrt(sqr(x) + sqr(y));
        if (n > Top) Top = n;
    }
}

float Annuität (Kapital,ZSatz,Jahre)
    float       Kapital,ZSatz;
    int         Jahre;
{
    float       r,rn;

    r = 1 + ZSatz/100;
    rn = exp(r,Jahre);  /* errechnet r hoch Jahre */
    return (Kapital * rn * (r-1) / (rn - 1));
}

...
LadeElement(a,b,39);
...
printf("jährliche Abzahlung = %8.2f", Annuität(a,c,24));
...
```

Bild 2 Prozeduren und Funktionen in C

denjenigen Bereich einzuordnen, in den es auch gehört: Entwurf von Prozeduren und Funktionen auf unterster Stufe, gewissermassen auf der «Mikrostufe» der Programmierung.

Schwachstellen des Prozedurkonzeptes

Auch bei der Verwendung von Prozeduren verbleiben eine Reihe von gravierenden Schwachstellen, die uns die Anwendung höherer Konzepte zur Strukturierung von Software als wünschenswert erscheinen lassen:

- Die «Aussenbeziehungen» einer Prozedur sind lediglich was den Verkehr über Parameter und Rückgabewerte betrifft aus der Prozedurdeklaration ersichtlich. Die Verwendung von externen Variablen wird nirgends besonders vermerkt; in dieser Beziehung bieten die Prozeduren keine Fortschritte gegenüber Subroutinen.
- Irrtümlichen Zugriffen auf externe Variablen werden folglich keine syntaktischen oder semantischen Schranken gesetzt. Wird in einer Prozedur beispielsweise eine Arbeitsvariable i verwendet, und wird deren Deklaration innerhalb der Prozedur vergessen, so wird eine global deklarierte Variable i angesprochen. Dies kann zu Fehlern führen, wenn dadurch Invarianzannahmen in anderen Programmteilen verletzt werden.
- Eine Kapselung von höheren Datenobjekten umfasst stets Datenkomponenten und zugehörige Funktionen. Die Kapselung eines Koordinatensystems schliesst beispielsweise die Repräsentation der Koordinaten (z.B. polare und kartesische) sowie die Funktionen zur Durchführung von Operationen auf Koordinaten ein. Durch Prozeduren allein ist eine solche Modellierung nicht sauber möglich, da damit keine syntaktisch einfache und semantisch genügende Konstruktion zur Zusammenfassung von Daten und Funktionen zu einem gegen aussen abgekapselten Ganzen zur Verfügung steht. Insbesondere störend ist die Ausgliederung aller permanenten Statusvariablen aus Prozeduren in den globalen Bereich, was in allen Sprachen ohne statische Variablen zur Notwendigkeit wird.
- Weitgehend gleiche Gründe führen zur Einsicht, dass durch ein Prozedurkonzept allein eine anwendungsorientierte Modellierung von Subsystemen wie Grafiksysteme, Datenbanksysteme, Kommunikationsdienste, Betriebssystemservices usw. kaum sauber möglich ist. Wo dies trotzdem getan wird, verwaltet die Klientensoftware meist direkt oder indirekt auch Datenbestände, die eigentlich vollständig zum Subsystem selbst gehörten (siehe beispielsweise in der Macintosh Toolbox [10]). Auch stellt das Subsystem häufig auf ein «wohlwollendes» Verhalten der Klientensoftware ab und verspricht bei allfälligem Fehlverhalten (z.B. durch falsche Parameter oder unzulässige Folgen von Funktionsaufrufen) undefinierte Resultate.

Diese Schwachstellen des Prozedurkonzeptes traten natürlich im Umfeld professioneller Softwareentwicklung sehr bald an die Oberfläche. Dies war ein Grund, warum beispielsweise derart viele Pascal-Erweiterungen definiert wurden. Die meisten davon enthielten Konstrukte, die genau die aufgezeigten Schwachstellen zu lindern versuchten und damit in Richtung der als nächstes zu besprechenden *Module* gingen. Im Rahmen dieser Einführung in objektorientierte Software beleuchten wir einzelne typische Entwicklungsstufen. Es ist dem Kenner der Programmiersprachen-Evolution aber klar, dass in der Realität diese Schritte nicht so klar und für sich isoliert dastehen. Sie sind vielmehr Ausprägungen einer kontinuierlichen Entwicklung.

(Teil 2 folgt im Heft 17/90)

Literatur

[1] *G. Pomberger:* Softwaretechnik und Modula-2, Carl-Hanser-Verlag, 1984.

[2] *N. Wirth:* Programming in Modula-2, Springer-Verlag, 1982.

[3] *R. Marty:* Methodik der Programmierung in Pascal, Springer-Verlag, 1986.

[4] *K. Schmucker:* Object Oriented Programming for the Macintosh, Hayden Book Company, 1986.

[5] *E. Gamma, A. Weinand, R. Marty:* ET++ – An Object Oriented Application Framework in C++, Proceedings of OOPSLA '88, ACH, 1988.

[6] *D. Parnas:* On the Criteria to be Used in Decomposing Systems into Modules, Communications of the ACM 15 (1972) 12.

[7] *B.W. Kernighan, D.M. Ritchie:* The C Programming Language, Prentice-Hall, 1978.

[8] *B. Stroustrup:* The C++ Programming Language, Addison-Wesley, 1986.

[9] *N. Wirth:* Program Development by Stepwise Refinement, Communications of the ACM 14 (1971) 4.

[10] *Apple Computer Inc.:* Inside Macintosh, Vol. I–V, Addison-Wesley, 1988.

Elektromagnetische Verträglichkeit (EMV) ein entscheidendes Qualitätskriterium für elektronische Apparate und Anlagen

Unser Entstörungslabor

- prüft die Störempfindlichkeit und das Störvermögen,
- bestimmt Störschutz- und Schirmmassnahmen,
- kontrolliert Apparate und Anlagen auf Einhaltung der gesetzlichen Störschutzbestimmungen,
- führt Prototyp- und serienmässige Entstörungen aus,
- steht Fabrikations- und Importfirmen für fachmännische Beratung in EMV-Problemen zur Verfügung.

PRO RADIO-TELEVISION, Entstörungslabor, 3084 Wabern, Telefon 031 / 54 22 44

In modernen Unternehmen ist es üblich, Schutzbefehle optisch zu übertragen.

Aus sicherheitstechnischen, wirtschaftlichen und anwenderspezifischen Gründen wird vielfach FOX 20 bevorzugt. Denn unser modulares 20-Kanal-Lichtleitersystem für die Übermittlung von Daten, Sprache und speziell Schutzbefehlen ist ganz auf Sicherheit ausgelegt. Durch fehlerhafte Schutzsignalübertragung verursachte Fehlschaltungen von Energieversorgungsanlagen und die damit verbundenen hohen Folgekosten fallen dahin! Dann haben wir für den Einsatz in der Elektrizitätswirtschaft ein spezielles Sortiment von Schnittstellen für den Anschluss an Signalquellen ohne Modems, Fernauslösgeräte und andere kostspielige Komponenten entwickelt. Zur Schonung der Investitions- und Unterhaltsbudgets! Weiter sind dank der digitalen Durchschalt- und Abzweigtechnik nebst Punkt-Punkt-Verbindungen auch komplette Telekommunikationsnetze für die diversen EW-Applikationen realisierbar. Einfacher und anwenderfreundlicher geht's nicht! Natürlich spricht noch einiges mehr für FOX 20. Was genau, hören Sie, nachdem Sie uns mit Ihrer Visitenkarte ein Zeichen gegeben haben.

ABB Infocom, Abt. ENF, 5300 Turgi
Tel. 056 29 27 47, Fax 056 29 94 61

ABB
ASEA BROWN BOVERI