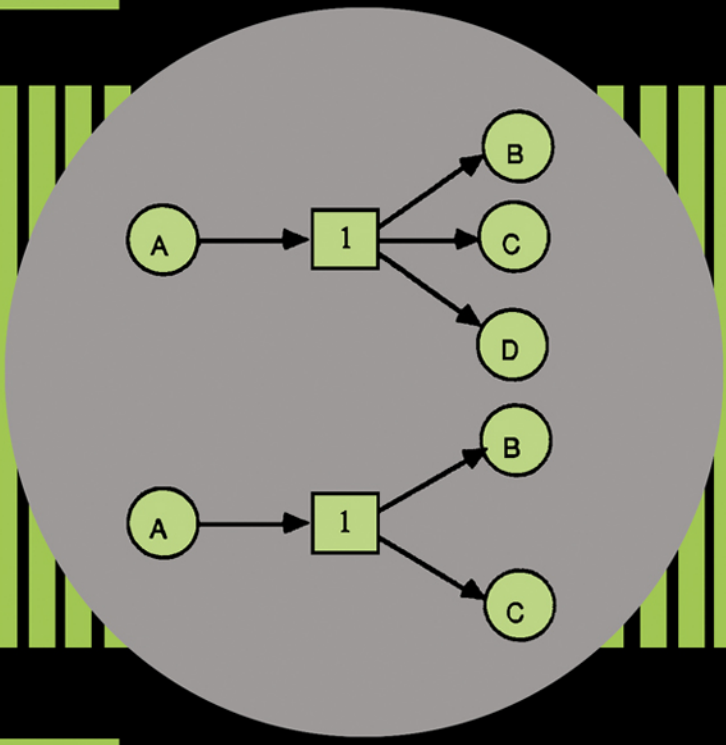
THE APIC SERIES ·30·

ADVANCED PROGRAMMING METHODOLOGIES



EDITED BY
GIANNA CIONI AND
ANDRZEJ SALWICKI

Advanced Programming Methodologies

This is volume 30 in A.P.I.C. Studies in Data Processing
General Editors: M. J. R. Shave *and* I. C. Wand
A complete list of titles in this series appears at the end of this volume

A.P.I.C. Studies in Data Processing
No. 30

Advanced Programming Methodologies

Edited by

GIANNA CIONI
Istituto di Analisi dei Sistemi ed Informatica, Consiglio Nazionale delle Ricerche, Rome, Italy

and

ANDRZEJ SALWICKI
Institute of Informatics, University of Warsaw, Warsaw, Poland

ACADEMIC PRESS
Harcourt Brace Jovanovich, Publishers
London San Diego New York Berkeley
Boston Sydney Tokyo Toronto

ACADEMIC PRESS LIMITED
24/28 Oval Road
London NW1 7DX

United States Edition published by
ACADEMIC PRESS, INC.
San Diego, CA92101

ISBN 0-12-174690-9

Printed in Great Britain by St Edmundsbury Press Ltd, Bury St Edmunds, Suffolk

Preface

The present volume is the result of a *Summer School on Advanced Progran ming Methodologies* which took place in Rome, 17–24 September 1987. Th work of the school concentrated on modern tools of software production. I motto was "*practice and theory should go together*". Therefore, new pr gramming tools, as well as new theoretical foundations for the production software, have been presented.

The inspiration for the school came from the Institute of Informatic University of Warsaw and Istituto di Analisi dei Sistemi ed Informatic (IASI), CNR of Rome. The school was organized jointly by these Institut and by Centro Interdipartimentale di Calcolo Scientifico, University Rome "La Sapienza".

One of the aims of that School was to attract the participants' attention the new, not well-known tools of advanced programming, in order to he the diffusion of new ideas. One of the subjects, and in our opinion the mo relevant, was object-oriented programming which is slowly gaining th attention of programmers. Its efficiency, its power in describing systems an its intrinsic modularity should be appreciated by all programmers. Neve theless, twenty years after the first definition of the ideas of class and objec these notions are not in wide use. This approach deserves more attentio Until now, only a few research papers, devoted to the properties of class and their objects, have been published. (Note that the number of pape devoted to the semantics of procedures are in thousands.) And this researc is most definitely non-trivial! The eventual results will be appreciated b those who know objects and would like to apply them in accordance to thei yet to be completely discovered, laws.

The Advanced Programming Methodologies School consisted of lecture demos and practical experiments. The participants had opportunities to ga experience in using the environments and languages presented during th lectures. This fact, we believe, made the statements of lectures mo convincing.

The production of software is slow, the products are to be debugged wi pain and costs. This is a well known fact. One can state that the softwa production is like manufacturing, a question of skills rather than of scienc

On the other hand we are aware of new techniques which can essential

change the work of programmers. These techniques, both of theoretical and software engineering aspects, are poorly known to the public. Moreover, the new tools of theoretical character are not introduced yet in the process of software production.

The new programming tools are still awaiting appropriate theoretical research.

From the above remarks it follows that we can profit from the methods offered by new programming languages, new environments etc. We should take also into account the challenge of new theoretical questions inspired by the new programming tools.

This book contains, as its part one, the collected papers prepared by the lecturers of the School. The second part, prepared by G. Cioni and A. Kreczmar, presents more detailed information on problems connected with implementation and application of high level programming languages. As one can see from the contents of the book the authors discuss mostly environments, modularity and methodology. We hope that readers will find the presented ideas and tools useful and inspiring. We are sure that the effort of learning new methods will be repaid by the results in the practice of programming. We would like to call the reader's attention to the LOGLAN'82 programming language. It offers all the possibilities already known and surpasses them by providing the programmers with many new tools. It seems worth mentioning that modules of programs (especially of LOGLAN programs) can be derived from algorithmic specification together with the proofs of their correctness.

The book is aimed at a broad circle of readers. It can be used during various courses on Methodology of programming. It can also be used by advanced students of Computer Science. The editors hope that the reader will appreciate and take up an invitation to study and research theoretical and software problems mentioned in the book.

We wish to express our sincere thanks to the IASI for the excellent organization of the School and for the computer facilities made available during the school. The school itself would never have taken place without the work of Mirella Schaerf whom all the lecturers and participants of the school wish to thank warmly. We thank the publisher for the encouragement to write the book and the patience with which they accepted our delays.

The Editors

Contributors

P. ATZENI
Dipartimento di Informatica e Sistemistica, Universita degli Studi di Napoli, Napoli, Italy

G. CIONI
Istituto di Analisi dei Sistemi ed Informatica, Consiglio Nazionale delle Ricerche, Viale Manzoni 30, 00185 Roma, Italy

A. CORRADI
Dipartimento di Elettronica, Informatica e Sistemistica, Viale Risorgimento 2, 40136 Bologna, Italy

A. FUGGETTA
Dipartimento di Elettronica Politecnico di Milano, Piazza L. da Vinci 32, 20133 Milano, Italy

C. GHEZZI
Dipartimento di Elettronica Politecnico di Milano, Piazza L. da Vinci 32, 20133 Milano, Italy

A. KRECZMAR
Institute of Informatics, University of Warsaw, 00901 Warszawa, Poland

D. MANDRIOLI
Dipartimento di Elettronica Politecnico di Milano, Piazza L. da Vinci 32, 20133 Milano, Italy

A. MIOLA
Dipartimento di Informatica e Sistemistica, Universita di Roma "La Sapienza", via Buonarroti 12, 00185 Roma, Italy

A. MORZENTI
Dipartimento di Elettronica Politecnico di Milano, Piazza L. da Vinci 32, 20133 Milano, Italy

A. NATALI
Dipartimento di Elettronica, Informatica e Sistemistica, Viale Risorgimento 2, 40136 Bologna, Italy

A. PETTOROSSI
Electronics Institute, Rome University, Via Orazio Raimondo, 00173 Roma, Italy

D. SACCA
Dipartimento di Sistemi, Universita della Calabria, 87030 Rende, Italy

A. SALWICKI
Institute of Informatics, University of Warsaw, 00901 Warsaw, Poland.

R. VITALE
Dipartimento di Informatica e Sistemistica, Universita degli Studi di Roma "La Sapienza", Via Eudossiana 18, 00184 Roma, Italy

M. SHERMAN
Information Technology Center, Carnegie-Mellon University, Pittsburgh, PA 15213, USA

C. ZANIOLO
Microelectronics and Computer Technology Corporation, Austin, TX 78759, USA

Contents

Development of Software from Algorithmic Specifications

Andrzej Salwicki
Institute of Informatics
University of Warsaw
PKiN room 850
00901 Warsaw POLAND

1. Introduction

Loglan is a name of a software project which contains as its kernel a universal programming language Loglan'82. The main objectives of the project were the tools for quick production of software and the application of scientific methods thus making software production a real technological process.

The speed in offering new software products, the possibility of introducing quickly improvements, are of importance. The eventual profits are of economical, technological and structural character. It is characteristic, for the present state of software "manufacturer's" production, that most of the big systems have been delivered with essential delays and that they are generally unreliable. This phenomenon is the best evidence of our thesis that the era of industrial production of software is before us yet. In our opinion one will recognize this era when at least two conditions will be satisfied:

1. when the production of software will be based on fundamental sciences, like civil engineering which is based on mathematics and physics,
2. when software systems will be assembled from subsystems, like cars are assembled from parts coming from different factories.

ADVANCED PROGRAMMING METHODOLOGIES
ISBN 0-12-174690-9

Is there a hope to satisfy these conditions in a future? Have we to wait long for this era? Our answer is: no, it is quite easy to meet the two criteria. The community of programmers and computer scientists knows enough many facts and has enough skill to arrive at the desired solution. Below, we shall present a point of view elaborated at Institute of Informatics, University of Warsaw. The opinions presented here are based on two projects which have been conducted in our Institute for many years. The first one was a theoretical project named Algorithmic Logic (AL). The goals of AL are to learn basic laws of computing which are independent of specific computer, programming language, data etc. The results of the research allow to use them as a methodology of software production. A similar research has been conducted with certain delay in West Europe and US, but the aims of Algorithmic Logic were wider than just Logic. There is enough evidence for the thesis that AL can serve as a tool for the formulation of the specification of software, as a deductive system for analysis of modules of programs, etc. Making use of the language of Algorithmic Logic we are able to provide complete axiomatic descriptions of data types, either "real" primitive data types of a programming language, or abstract ones. It turned out that such axiomatizations makes the analysis of correctness and of other semantical properties easier. Moreover, we found a formal counterpart of implementation notion. If one algorithmic theory is interpretable within another, then the corresponding data structure (its model) is implemented in the second structure.

Project Loglan brought a second factor: the possibility to compose, extend and apply modules of software which come from various producers.

The possibility of storing algorithms in libraries of procedures is well known. What the community needs is the possibility of storing, handling, composing etc. of modules which implement systems. Such a

possibility is offered by packages of ADA programming language. But we are sorry to say that much more general tool has been overlooked. It is the prefixing invented years ago by the designers of Simula. The virtues of prefixing are numerous, making programming in Simula highly efficient, but also totally different from programming in other languages. On the other hand, the Simula's implementation of prefixing, has many limitations which seem to contradict its potential profit.

2. Methodology of Programming

2.1 Abstract data types

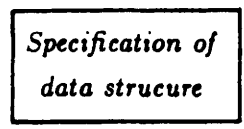
In the majority of the cases we have to develop a piece of software which performs certain operations not available in a moment. In other words, our future program is to be executed in a data structure other than supplied by hardware and system software. In 1972 C.A.R.Hoare [2] remarked that in such case one should factorize the goal onto two subgoals:

i) to specify and implement a data structure,

ii) to design, analyse and use a "abstract" program.

According to this advice we should develop two modules

The only link between these two pieces of software should consists of



Two teams of programmers can be created for the work on two modules. A team developing the abstract program should base only on the specification. That is, the semantical properties of program should be deduced only from the axioms contained in the specification. An implementing team uses the specification as a criterion of correctness of the implementation. The virtues of this method are manyfold. The principle of factorization makes possible to execute the abstract program in the presence of different implementing modules. However a correct program doesn't need to be adjusted. It will be the same for all implementing modules. We can gain or loose, on efficiency of computations depending on our choice of implementation for the data structure. Another advantage of the method consists in the possibility of multiple applications of once created implementing module.
The module can be conceived as an implementation of a new language.

The work should have at least three visible stages:

a) formulation of a specification, i.e. an axiomatization of the data structure,
b) design of abstract program and its verification basing on the specification,
c) realization of the data structure and verification of its correctness (also basing on specification we verify the validity of its axioms in a given implementation).

2.2 Systems

It is of importance to be able to handle systems, very much like we are able to handle algorithms today. In this place many readers can protest: well, we have built many systems already. That's correct. But are these systems decomposable? Is it easy to exchange certain part of it? etc. What we really need are modules of software which can be taken from shelves like one takes now modules of hardware

and assembles them. We need also encapsulated systems.

What we understand by a system? Any collection consisting of a set of elements, the universe, and of a set of operations and relations. Therefore a system is an algebraic structure, the fundamental notion of the mathematics. The practice imposes additional requirements, and it may be difficult to express them in the language of mathematics. Below, we shall list a few of them. The universe doesn't need to be homogeneous. It is frequently the case that the universe is partitioned onto disjoint subsets called sorts. In an example of the system of stacks we consider two sorts: E of elements, and S of stacks. The operations can require that the arguments should be of definite sorts, e.g. the first argument of sort E and the second of sort S, the result of the operation being of sort, say, S. Moreover it is important to create systems which have a better degree of dynamicity and then that the objects of the systems can perform their own actions. This option can be demanded on three ascending levels:

a) In the case objects are passive, it has however to be possible to perform an action on demand of certain active agent. Why it is desirable? One good reason is that it enables to write clear expressions. Another justification comes from the observation that this way of work with objects allows to save on the time and space of parameter's passing. But the most important outcome of such system is that it is a "system". More seriously, it is of importance to be able to collect into one module the definitions of data and of operations on them. Compare the PASCAL approach and the Loglan one.

EXAMPLE

{pascal}	*{loglan}*
type comp = record	*unit comp: class(re,im:real)*
re,im : real	

```
end record;

function add(z,t:comp):comp;
      var addtemp: comp;
begin
      addtemp:= new comp;
      addtemp.re:= z.re+t.re;
      addtemp.im:= z.im+t.im;
            add:= addtemp
end add;
function mult(z,t:comp):comp;
...{ details omitted}
end mult;
```

```
      unit add:function(z:comp):comp;
begin
      result:=newcomp(re+z.re,im+z.im)
end add;

      unit mult:function(z:comp):comp;
begin
      result:=newcomp(re*z.re
            -im*z.im,re*z.im+im*z.re)
      end mult;
end comp;
```

The similarities are visible, the differences require a word of comment. On the left side we find a collection of three modules which are supposed to work together. But what will happen when an inadvertent programmer will move two of them into certain place of his program leaving the third module alone? We are to keep in mind that the three form an entity. On the right side we have an encapsulated module. We don't need to worry about its structure. When we are going to use it, we use its full text. We can gain on execution time since the operations add and mult defined in the class comp require only half of memory access operations in comparison with those executed in the left side. The functions add and mult, being local in the class comp, can utilize the local attributes re and im of comp object. Finally, it is interesting to compare two expressions which use different implementations. Suppose we have the declaration

```
var z,t,u,v: comp
```

then the expressions are

{pascal}	*{loglan}*
mult(add(z,t), add(u,v))	*z.add(t).mult(u.add(v))*

Remark the differences in syntax. In the second case one can write expressions in an infix notation. One can also economize the number of parameters passed.

b) In programming of games, in simulation packages etc. we often wish to create objects which can be activated from time to time. Just like players in a game, certain objects are called to resume their actions at the latest reactivation point and when they perform actions which correspond to one step in a game, they renounce their activity till they are awakened again. Here one can differentiate among the schemes which demand that a name of the activated object is given explicitly (this is the case of coroutines), and another case in which an active object returns the processing ability to the object which activated it without knowing its name (this is the case of semicoroutines).

c) The third level is encountered when a system to be created should be able to deal with situations in which many objects execute their actions simultaneously. This demand causes the need for objects being concurrent processes.

In all three cases objects are not only manipulated from outside. They are not only objects but they are also sovereign subjects on their own.

2.3 Hierarchies

It is well known that big, complicated systems can be designed, realized and maintained if and only if a hierarchy is imposed. The hierarchy may concern various aspects of the systems. Sometimes it

is enough to consider a hierarchy of subsets of a certain universe of objects. Consider for example a general notion of bill. Every bill contains certain common attributes like:

amount_to_be_paid: currency

paid: boolean

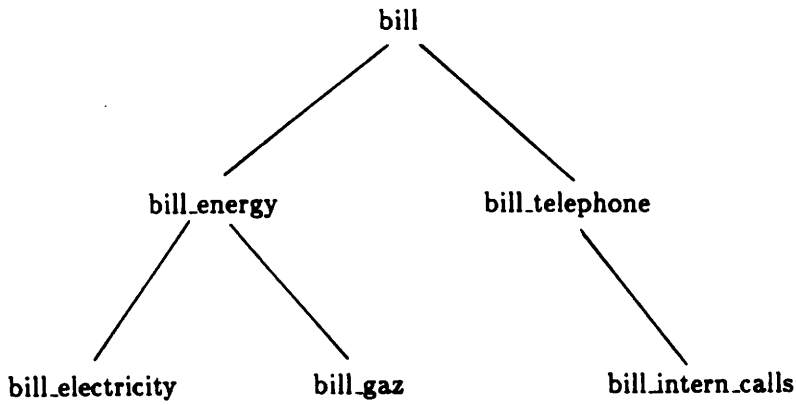
year_month_day: date

.....

and other attributes corresponding to a specific case. One can define various subsets of the set of the set of bills e.g.

bills_for_energy, bills_for_telephone, ...

The structure of the subsets can be further developed into a tree-like structure, e.g.



It seems important to have the ability to treat common features of bills by common algorithms. In order to do so we require that the rules of compatibility of types will allow to assign an object of type, say, bill_gaz to a variable of type bill. But not conversely. It would be disastrous if we allow to perform an operation proper for the type bill_gaz on an object which is enable to interpret its data in accordance

with the structure of bill_gaz.

Obviously one can consider also hierarchies of subsystems not only of subsets.

2.4 Protocols, axioms, behaviours

It is of importance to have the possibility to enforce certain axioms, protocols or behaviours on the systems and their elements. As an example we would like to quote: the ability to create entry procedures of monitors in a way guaranteeing that the protocol of mutual exclusion will be observed. Other examples of synchronization tools are easy to imagine.

A quite different demand may appear when we expect that all objects of certain system will satisfy specific axioms throughout its lifetime cycle, e.g. one can demand that all objects of certain type T are "normalized". This property can be inadvertently destroyed by a user. For example, how to make sure that when working with lists we shall never turn a list into a ring? How to ensure the integrity constraints of a data base? How to enforce objects that represent players in a game that they behave according to rules of game?

In all these cases we would like to have predefined frames of behaviour, which one can develop according to his need but preserving some axioms, or, if you wish, invariants.

2.5 Signaling and exceptional situations

It is frequently so that elements of systems communicate by sending and receiving signals. An arriving signal can interrupt the normal flow of calculations. The signals are either binary, just presence or absence of a signal, or they convey a complicated structured message.