Gustav Pomberger

Software Engineering and Modula-2

PRENTICE-HALL INTERNATIONAL SERIES IN COMPUTER SCIENCE

C.A.R. HOARE SERIES EDITOR

TP31
P784

SOFTWARE ENGINEERING AND MODULA-2

GUSTAV POMBERGER

University of Zürich

Prentice/Hall PHI International

Englewood Cliffs, NJ London Mexico New Delhi Rio de Janeiro
Singapore Sydney Tokyo Toronto Wellington

Library of Congress Cataloging in Publication Data

Pomberger, Gustav, 1949 –
Software engineering and modula-2.

Bibliography: p.
Includes index.
1. Modula-2 (Computer program language) 2. Computer software – Development.
I. Title. II. Title: Software engineering and Modula two.
QA76.73.M63P6613 1986 005.13'3 85-28168
ISBN 0-13-821794-7
ISBN 0-13-821737-8 (Pbk)

British Library Cataloguing in Publication Data

Pomberger, Gustav
Software engineering and Modula-2.
1. Programming (Electronic computers)
2. Computer engineering
I. Title II. Softwaretechnik und Modula-2,
English
005.1'2 QA76.6

ISBN 0-13-821794-7
ISBN 0-13-821737-8 (Pbk)

Prentice-Hall Inc., Englewood Cliffs, New Jersey
Prentice-Hall International (UK) Ltd, London
Prentice-Hall of Australia Pty Ltd, Sydney
Prentice-Hall Canada Inc., Toronto
Prentice-Hall Hispanoamericana S.A., Mexico
Prentice-Hall of India Private Ltd, New Delhi
Prentice-Hall of Japan Inc., Tokyo
Prentice-Hall of Southeast Asia Pte Ltd, Singapore
Editora Prentice-Hall do Brasil Ltda, Rio de Janeiro
Whitehall Books Ltd, Wellington, New Zealand

Printed and bound in Great Britain for
Prentice-Hall International (UK) Ltd,
66 Wood Lane End, Hemel Hempstead, Hertfordshire, HP2 4RG
by A. Wheaton & Co. Ltd, Exeter.

1 2 3 4 5 90 89 88 87 86

ISBN 0-13-821794-7
ISBN 0-13-821737-8 PBK

Foreword

"Software engineering" is a concept that was coined towards the end of the 60's. The intention is to imply that the technique of creating complex, programmed systems is subject to laws similar to those for the manufacture of machines (engines). In both cases a critical requirements analysis and a careful choice of solution methodology is necessary. On the other hand fundamental differences must not be ignored. The most obvious is certainly that the products of the software engineer, as opposed to those of the mechanical engineer, are immaterial, as we say, "soft." They are represented as programs, as formal texts which are first brought to life by the computer.

The fact that texts--especially with the aid of a computer--are easily changeable, has the advantage that any shortcomings which may arise can be eliminated (or hidden) quickly, and at the same time the disadvantage of seducing the engineer to abuse this advantage. Experience demonstrates that it is precisely here that a sound initial concept is essential for success, because numerous after-the-fact adaptations rapidly lead to a system complexity which--and here is another difference to mechanical engineering--is cheap to introduce, but which later, in use, turns out to be expensive.

The merit of this book is that it handles the field of software engineering comprehensively. First a modern foundation is laid for the technical tools, and then they are applied. All phases of a project are carefully discussed, and not just programming; requirements analysis, specification of the requirements, design of a solution, documentation, and finally testing and maintenance receive their deserved attention. In conclusion, a chapter is dedicated to project management, whereby its scope in comparison with that of the technical aspect is, happily, kept within bounds. Thus clear priorities are set, and the trend of placing the blame on management for failures due to a lack of technical competence has been checked.

The book thus distinguishes itself fundamentally from the usual introductions to programming. In spite of this, an important chapter is dedicated to the technique of programming, the pillar of all software engineering. I am pleased that the author based it on the language Modula-2, which was developed especially for projects of the nature treated herein, and which represents a natural evolution of the language Pascal, which, in the meantime, has itself become globally accepted. This book is a welcome guide to its competent application.

N. Wirth

Preface

This book is intended for everyone who plans, designs and implements software. The writer has occupied himself for years with the production of software, at first in industry, and in the past years, in teaching and research at the university level. This book is a result of the experience collected in the process.

The term "software crisis," applicable even today, indicates that software is often incorrect, that immense difficulties appear in mastering complexity in the production of software, and that programs are often only understood by their own authors. One question also often discussed in this connection is that of the generality and acceptability of a programming language. Most of the programming languages in practical use today are conceptually outdated by the present standards of software engineering, and lend themselves little or not at all to the support of the new concepts in software engineering (such as data capsuling).

This book is intended to guide the reader to a new area, which is in no way supported by a secure basis of facts, but which, rather, is still in the beginning stages. It represents an attempt to depict personal and outside experiences in the field of software engineering systematically. It describes the phases of a software project (software life cycle), the problems which occur and how they are solved by formal, workable methods of software engineering. Because the choice of programming language is of great importance to the implementation, efficiency, quality and portability of software, a major part of the book is dedicated to the description of a modern programming language which corresponds to today's technological level. For this the writer, after serious consideration, has chosen the programming language Modula-2, developed by N. Wirth. Modula-2 is a simple, widely machine independent language and, because of its descendency from Pascal, easily learned by many programmers. The volume of the language is modest in comparison to other new programming languages (as, for example, Ada). The language lends itself, therefore, also to implementation on microcomputers and supports many of the software engineering concepts known today. In addition, Modula-2 lends itself, in the view of the writer, equally well to systems and applications programming.

The author is primarily interested in handling the fundamental principles and methods of software engineering in detail and in depth. From this the reader should be left in a position to apply these techniques on his own. To this end each section contains carefully commented examples. Detailed descriptions of methods which were developed for extremely specialized tasks of software engineering are intentionally omitted. These methods are discussed briefly, and the reader is referred to the appropriate literature. The author has also taken pains to ease access to advanced literature for the more intensively interested reader, by referencing it in many places, especially where it was important to keep the book within bounds.

It is expected of the reader that he have an elementary knowledge of algorithms, data structures and programming languages. He should be acquainted with the fundamentals of mathematics and logic, and above all have an interest in the subject.

March 1984, Linz G. Pomberger

Acknowledgments

For me it is an honor and a pleasure, at this time, to express my thanks for the help rendered and the many suggestions received.

I am most especially indebted to *Prof. Peter Rechenberg*, my teacher and mentor. During the past years he has exerted a great influence on my thinking. Much of what I have recorded in this book is particularly influenced by his work. Whenever a question arose during the writing of this book, it was my habit to note in the margin of the manuscript, "Ask Rechenberg." *Prof. Rechenberg* was always supportive in word and deed. His priceless suggestions place me in his debt.

I wish to express my special gratitude also to *Prof. Niklaus Wirth* from Zürich. His work on algorithms and data structures, systematic programming, compiler construction and, above all, the definition of the programming languages Pascal and Modula-2 are of great importance for software engineering and have strongly influenced much of my work. I also owe to *Prof. Wirth* the possibility of a three month research stay at the ETH in Zürich. In this time it was possible for me to complete my understanding of the language Modula-2 and to discourse with him, the inventor of the language.

Still further I wish to express my thanks to the many other people who helpfully stood by my side during the long task. It is impossible for me to list each by name. However, I am most especially obligated to thank:
Günther Blascheck for the continuous exchange of thoughts, which, for me, were a great source of inspiration, and for his inspection of the manuscript,
Peter Mössenböck for helpful discussions, and because he read the entire manuscript and accorded me invaluable advice,
Prof. Jörg Mühlbacher, to whom I owe the contact to the publisher,
Mrs. Ingrid Kirchmaier for her excellent typing and because she was always at my side in good spirits,
the *students* who attended my software work sessions, and who provided the stimulating environment for my work through many discussions,
the original publisher, *Carl-Hanser-Verlag* , for their understanding and patience concerning the late delivery of the manuscript.

SOFTWARE ENGINEERING AND MODULA-2

Table of Contents

1. Introduction to the Problem

1.1 The Development of Software Engineering

The first computing machines were primarily used in the field of pure and applied science. The task of the programmer was not so much to discover complex algorithms as to formulate already known algorithms in a programming language in order to execute them on a computer. To this end no special knowledge was required except the command of a programming language. The programmer was ordinarily also the user. The programs were only occasionally used, and the tasks were seldom respecified. The only difficulties were to guarantee correctness and efficiency. The problems to be solved were comparatively simple, relative to those of today, and the programs were therefore relatively small. As a result the number of programming errors was small, and first became a major problem only when program systems came to be used in the solution of complex scientific and commercial applications. After this, whole teams of programmers worked on the production of program systems which were to be employed by various users. The specification of the problem and the demands placed on the system changed often during the design phase, and also long after the program system had been in use. Besides correctness and efficiency, the mastering of complexity through the decomposition of a problem into problem pieces, the specification of interfaces, security and reliability, flexibility, documentation, maintenance and project organization became major problems in the production of large program systems. This led to difficulties in the design and production of software to such an extent that in 1965 the term "*software crisis*" was coined. There is no comparison to this in the development of hardware. This is not to say that hardware is error-free, but in practice the reasons for difficulties which occur in large computer systems can usually be found in software errors.

Dijkstra (1972a) described this situation as follows:

"To put it quite bluntly: as long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become an equally gigantic problem. In this sense the electronic industry has not solved a single problem, it has only created them--it has created the problem of using its products."

The possibilities which were created by new computer generations vastly exceeded the programming techniques which had been developed up to that time. But the growing economical importance of software production (see *Boehm* 1973), and the enormous expansion of the data processing industry, which forced the development of numerous large program systems, pressed the demand for an improved *programming technology* more and more into the focus of research in the field of computer science.

The attempts at researching acceptable programming technology resulted in two software engineering conferences organized by NATO, in Garmisch in 1968 (*Naur* and *Randell*, 1969) and in Rome in 1969 (*Buxton* and *Randell*,1969). It was indicated at these two conferences that programs are industrial products, and that there was therefore the requirement for: Renunciation of the "art" of tricky, egotistical programming by the individual and adoption of planned, co-operative team programming (see also *Kimm* et al., 1979).

Since then an attempt has been made to analyse *software production* scientifically, considering it as a coherent process, and above all to place the questions of specification, methodical program design, requirements on a programming language, project organization, quality control, documentation and the automation of software production at the center of research interests.

1.2 The Concept of Software Engineering

What lies behind the term "*software engineering*"? Is this just an alternative phrase for programming, or is it a new technology? The term *software engineering* is obviously intended as provocation and indicates that the economical production of programs is an *engineering discipline*. There is to date, however, no generally acceptable, fixed definition of the concept.

Boehm (1979) defines software engineering as follows:

> "The practical application of scientific knowledge in the design and construction of computer programs and the associated documentation required to develop, operate, and maintain them."

In *Dennis* (1975) we find the definition:

> "Software Engineering is the application of principles, skills and art to the design and construction of programs and systems of programs."

D.L. Parnas (1974) writes:

> " . . . software engineering is programming under at least one of the following two conditions:
>
> (1) More than one person is involved in the construction and/or use of the program and
>
> (2) more than one version of the program will be produced."

Finally, *F.L. Bauer* (1975) writes:

> [The aim of software engineering is:] "To obtain economically software that is reliable and works efficiently on real machines."

The definitions given above show that the production of *large programs* involves new problems of a different nature to those of the production of *small programs*, and exhibits many similarities with the production of other technical products. The main problems here are:

* the mastering of complexity,
* the decomposition of a problem into pieces, which are then solved by various groups,
* project organization,
* the specification of interfaces between pieces,

* efficiency,
* the documentation and maintenance of the systems,
* portability and adaptability.

Common sense alone is not sufficient for solving all of these problems. It is necessary to examine the entire complex scientifically in order to create the prerequisites for the development of methods and tools which support software development and production. Therefore, it is expected of *good software engineering* that it provide methods, tools, norms and aids which make it possible to handle technical problems (such as *specification, design, construction, testing, efficiency, documentation and maintenance*) and organizational problems (such as *project organization* and *interface specification*) which occur in the production of software, and, in the process, to produce and apply software economically.

Based on this, *software engineering* can be defined as follows:

"*Software engineering* is the practical application of scientific understanding to the economical production and use of reliable and efficient software."

1.3 The Programming Language Modula-2

The development of computing machines was accompanied by the development and implementation of programming languages, in conjunction with compilers for these languages. *Rechenberg* (1983) writes in relation to this: "There already exist many hundreds of programming languages, and new ones are constantly being invented. The (computer) public takes absolutely no notice of most of these and they lead an insignificant life in the surroundings of their inventors." This is not surprising, if one considers which conditions must be satisfied for the general acceptance of a programming language (see *Goos*, 1982):

* The language must satisfy a factual need, because otherwise the effort spent on re-schooling programmers cannot be justified.
* The existence of the language and its compiler must be secured for long time spans (20 years and more, but at least for the life of the program system).
* Reliable compilers must exist for all important target machines, and maintenance and development of these compilers must be secured. In particular, the use of a language cannot be dependent on any given computer manufacturer.
* The implementation should be compatible with programming languages already in use, so that frictionless transition and rational co-existence of the languages is possible.
* The language should be easy to learn.

Until now, only Fortran, Cobol, PL/I, Basic and, increasingly, Pascal have been able to meet these primarily economic demands. Fortran and PL/I owe their success to the fact that they were offered and supported by the leading hardware producer, IBM. Cobol succeeded because the US Department of Defense, one of the larger computer and software customers, stipulated its use. Pascal and Basic are purely university products. Basic, although widespread, does not even begin to satisfy present day requirements on a programming language. Pascal is the only language which has become universally known on the basis only of its own qualities.

However, in the meantime software technology has continued to develop, and ever more stringent requirements have been placed on programming languages. Because none of the conventional programming languages could meet these requirements, new languages have been developed, the most important of which are *Ada* (*Ichbiah* et al., 1979) and *Modula-2* (*Wirth*, 1982).

Because this book handles the entire complex of software engineering, and thus the problem of the implementation and choice of a programming language, the author must consider which language he should recommend to the reader in order to emphasize correct software engineering techniques. Since only Ada and Modula-2 in some way meet the requirements of modern day software engineering the choice lies between them.

Striking similarities are apparent in a comparison of these languages, although they differ completely in volume, intended application and historical development. In Modula-2, as in Ada, the specification and implementation sections of programs are separated, both languages allow separate compilation and the program structure "module" in Modula-2 corresponds almost exactly to the "package" of Ada. Modula-2 allows the formulation of parallel processes, as does Ada, although at a lower level, but then again with more flexibility. The handling of exceptions and generic programs is missing from Modula-2, but these were purposely omitted in order to limit the size of the language.

In this connection *Rechenberg* (1983) writes: "The result of these similarities is that one may use Modula-2 instead of Ada in the vast majority of applications. The clarity of programming in Modula-2, the documentational value of Modula-2 programs and the extent to which the principles of modern software engineering may be applied by Modula-2 programs is just as great as for the corresponding Ada programs, if not in some respects even greater." The small size of Modula-2 is just as appealing; the language definition is only 25 pages long, that of Ada a few hundred. Modula-2 was also developed with an eye to its implementation on microcomputers, and there already exist effective compilers (for example for the workstation Lilith (*Wirth*, 1981) from *N. Wirth*, for Apple computers, Motorola 68000 processors and Intel 8080/8086 processors), which permit practical examination. Not only were the high expectations placed on Modula-2 completely satisfied in an intensive practical test by the author, they were surpassed. Modula-2 is easier to learn than was originally assumed.

Modula-2 was originally developed as a systems programming language for tasks for which Pascal was not sufficient. The language is largely machine independent and supports many software engineering concepts known today. This new language is thus suited to systems as well as to applications programming. The most important *advantages* which become apparent with the use of Modula-2, and which fundamentally add to increased programmer productivity are:

- The capability of separate compilation. This saves much unnecessary compilation time during the test phase.

* Complete type and interface checking, even if modules are compiled separately. This exposes a number of programming errors at compilation time and thus shortens the test phase.
* Modules, which constitute a program structure encompassing procedures, and which admit the implementation of data capsules. They not only increase the documentational value of programs, but also guarantee that large program systems be implemented faster than usual, through the minimization of error sources and the adaptability of modules independently from each other.
* The capability of formulating parallel processes, which extends the range of application markedly.

All these advantages, especially the simplicity of Modula-2 and the ability in Modula-2 to utilize all known software engineering concepts have led the author to present this language to those interested in software design.