



SOFTWARE ENGINEERING ECONOMICS

BARRY W. BOEHM

Barry W. Boehm

*Director, Software Research and Technology
TRW, Inc.*



SOFTWARE ENGINEERING ECONOMICS

Prentice-Hall, Inc., Englewood Cliffs, New Jersey 07632

Library of Congress Cataloging in Publication Data

BOEHM, BARRY W. (date)
Software engineering economics.

(Prentice-Hall advances in computing science and
technology series)

Bibliography: p.
Includes index.

1. Electronic digital computers—Programming—
Economic aspects. 2. Electronic digital computers—
Programming—Economic aspects—Case studies. I. Title.
II. Series.

QA76.6.B618 001.64'25'0681 81-13889
ISBN 0-13-822122-7 AACR2

© 1981 by Prentice-Hall, Inc., Englewood Cliffs, N.J. 07632

All rights reserved. No part of this book
may be reproduced in any form or by any means
without permission in writing from the publisher.

Editorial/Production Supervision
and Interior Design: *Lynn S. Frankel*
Cover Design: *Carol Zawislak*
Manufacturing Buyer: *Gordon Osbourne*

**Prentice-Hall Advances
in Computing Science and Technology Series**
Raymond T. Yeh, editor

Printed in the United States of America

10 9 8 .

ISBN 0-13-822122-7

Prentice-Hall International, Inc., *London*
Prentice-Hall of Australia Pty. Limited, *Sydney*
Prentice-Hall of Canada, Ltd., *Toronto*
Prentice-Hall of India Private Limited, *New Delhi*
Prentice-Hall of Japan, Inc., *Tokyo*
Prentice-Hall of Southeast Asia Pte. Ltd., *Singapore*
Whitehall Books Limited, *Wellington, New Zealand*

considerations are covered in Section 8.8, and Section 31.2 covers conversion cost estimation.

33.3 NONPROGRAMMING OPTIONS: PROGRAM GENERATORS

Another class of nonprogramming options for improving software productivity are the program generators, which are systems for constructing (or aiding in the construction of) programs out of pre-existing pieces.

Various types of program generators exist, such as program libraries, applications generators, and Very High Level Languages (VHLLs). In fact, relative to the initial usage of absolute binary code for programming, one can consider assemblers, macro-assemblers, and current higher order languages (HOLs) as initial successful steps in developing program generators.

We can use current HOL technology as a means of illustrating the four main elements of a program generator.

1. *A set of components.* For HOLs, these include assembly language instructions, call and return macros, and standard functions such as square root, successor, end-of-file, etc.
2. *A set of conventions for joining the components.* For HOLs, these include conventions on use of registers for call and return sequences, and handling of exception conditions (negative square root, divide by zero, etc.).
3. *A language for users to specify desired applications.* For HOLs, these include FORTRAN, COBOL, Pascal, etc.
4. *A set of capabilities for interpreting user specifications, configuring the appropriate set of components, and executing the resulting program.* For HOLs, these include compilers, linkers, loaders, and execution monitors.

The main classes of program generator capabilities are summarized below, in terms of their distinguishing characteristic sets of components, sets of conventions for joining components, user languages, and sets of interpretation, configuration and execution capabilities.

1. *Software Piece Parts.* These are largely extensions of the standard functions provided within HOLs. They perform elementary functions in such areas as text processing, matrix-vector manipulations, input editing, etc., usually in the form of in-line code generated by an extension of an assembler or HOL compiler.
2. *Program Libraries.* These generally refer to collections of more extensive data processing capabilities, such as statistical analysis, financial calculations, graphics display generation, etc. These again are typically called as subroutines or external procedures from an HOL program.
3. *Application Generators.* These typically consist of a program library which operates in a prestructured context, based on knowledge of a particular applica-

tion area (display management, inventory control, aircraft flight mechanics, etc.). The structure imposed by the particular knowledge domain allows users to generate application programs simply by specifying options, sequences, and parameters in a special application-oriented language. Some particularly powerful application generators are becoming available for business applications, such as Mathematica's RAMIS, National CSS's NOMAD [McCracken, 1980], Information Builders' FOCUS, and IBM's Application Development Facility.

4. *Very High Level Languages*. These typically combine the special knowledge-domain structure and program library of the application generator with the power of a HOL and its compiler. Examples are simulation languages, automatic test equipment languages, query languages, etc.
5. *Automatic Programming*. This refers to the ultimate in program generation capability, in which a user begins to specify his desired information processing activity to an automatic programming system, which then asks the user questions to resolve ambiguities, clarify relationships, etc., and to converge on a particular program specification. The system then automatically generates a program which implements the specification. Outside of some very restricted-domain systems which are more accurately termed application generators or very high level languages, automatic programming systems are still somewhat beyond the current frontier of the state of the art.

Productivity Advantages of Program Generators

The major productivity advantage of program generators is that sizable computer applications can be generated using a very small number of user-language directives. Developing software via program generators can certainly be a far more cost-effective pursuit than developing software one instruction at a time.*

However, in doing so, it is essential to consider whether the generated program is actually going to solve your problem. If it does, you reap the benefit of a large labor savings. If it does not, though, you may simply reap larger problems. As an example, suppose that the initial program described in the *Scientific American* case study in Chapter 1 could have been developed with a program generator. Although this would have reduced the amount of software development effort, the end result would still have been to increase operating costs, decrease reliability and quality of service, and decrease staff morale. Thus, the existence of a program generator for a given application area does not imply that all of our software problems in that area are solved. We still need a good deal of front-end effort in applying such techniques as cost-benefit analysis and the GOALS approach, to ensure that our program generator will generate the right program.

One clear advantage of the program generator in this regard is its value in developing quick prototypes of a desired software capability, on which our assumptions about system usage can be tested by actual use. If the resulting software is somewhat

* Another attractive option in this direction is the implementation of software componentry in hardware chips. See [Boehm, 1980] for a discussion of the potential capabilities and limitations of this option.

off the mark, but can be corrected within the domain of the program generator, we still achieve a significant labor savings. And even if the use of the prototype convinces us that we need to build a new product, we have learned that lesson much more cheaply than we would by building the software from scratch based on faulty assumptions.

In sum, although it is inadvisable to apply a program generator without some initial analysis of its applicability, it is equally inadvisable to reject the program generator option (or the software package option) with a wave of the hand and an assertion that "our problems are so special that we need custom software." More and more frequently, this will not be the case. Often, just a few inessential compromises in our mode of operation will make a program generator product or software package an adequate match for our needs.

On Rapid Prototyping and Evolutionary Design

The emergence of application generators oriented around a data base management system and report-generation capability has created an attractive approach for software development; the rapid-prototyping/evolutionary-design (RP/ED) approach. The basic RP/ED approach is as follows [McCracken, 1980a; 1981]:

1. Use the application generator to develop a rapid prototype of key portions of the user's desired capability.
2. Have the user try the prototype and determine where it needs improvement.
3. Use the application generator to iterate and evolve the prototype until the user is satisfied with the results.
4. If the performance of the resulting system is adequate, establish it as the user's system and continue to update and maintain it via the application generator. If higher performance is required, either tune the prototype or use the prototype as the *de facto* specification for developing a high-performance system.

Advantages of the RP/ED Approach

The RP/ED approach provides an alternative to the generation and validation of written requirements specifications and draft users' manuals as a way of ensuring that a software product will be responsive to the user's needs. The main advantages of the RP/ED approach are:

- Exercising the prototype provides a much more realistic validation of user requirements than does the review of a set of specifications and manuals.
- Using the prototype surfaces a number of second-order impacts that the system will have on the user's mode of operation.
- The rapid development minimizes the problems of accommodating the inevitable stream of requirements changes which surface during a long development period.
- The rapid-prototyping capability makes it possible to generate several alternative systems for comparative trials, and to provide quick-response solutions for user difficulties.