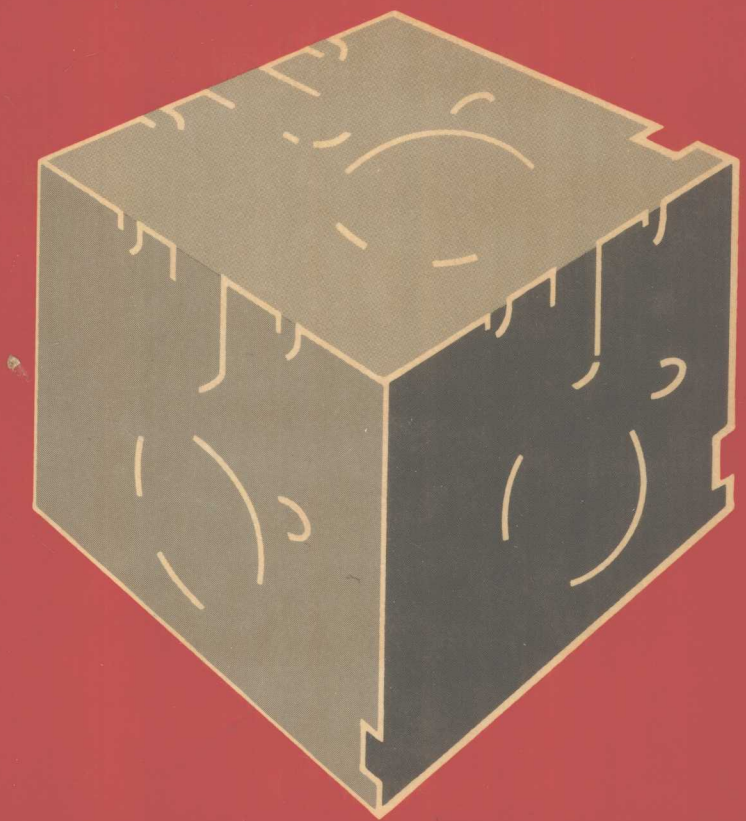
SOFTWARE LIFECYCLE MANAGEMENT

THE INCREMENTAL METHOD

William C. Cave
Gilbert W. Maymon



The Macmillan Database/Data Communications Series
Shaku Atre, Consulting Editor

Software Lifecycle Management:
The Incremental Method

William C. Cave
Prediction Systems, Inc.

Gilbert W. Maymon
Electronic Associates, Inc.

Macmillan Publishing Company
A Division of Macmillan, Inc.
New York
Collier Macmillan Publishers
London

Macmillan Publishing Company
866 Third Avenue, New York, NY 10022

Collier Macmillan Canada, Inc.

Printed in the United States of America

printing number
1 2 3 4 5 6 7 8 9 10

Library of Congress Cataloging in Publication Data

Cave, William C.
Software lifecycle management.

Bibliography: p.
Includes index.
1. Computer programming management. 2. Software maintenance. I. Maymon, Gilbert W. II. Title.
III. Title: Incremental method.
QA76.6.C39 1984 001.64'2'0685 84-11264
ISBN 0-02-949210-6

Software Lifecycle Management

The Macmillan Database/Data Communications Series

Shaku Atre, Consulting Editor

Available

Cave/Maymon *Software Lifecycle Management*
Fadok *Effective Design of a Codasyl Database*

Forthcoming

Brown/Geelen *Data Modeling in a Business Environment*
Bubley *Data Communications Systems*
Callender *An Introduction to PSL/PSA*
Cohen *A Guide to DBMs for Microcomputers*
Crout *Designing Better Systems for Endusers*
Fields *The DP/User Interface Conflict*
Ha *Digital Satellite Communications*
Musteata *How to use CICS Languages to Create On-Line Applications*
Myer *Global Communications: A Computer-Based Message System Approach*
Potter *Local Area Networks: Applications and Design*
Ranade/Ranade *VSAM: Multiprogramming with Virtual Storage*
St. Ammand *A Guide to Packet Switched Value Added Networks*
Towner *The ADS/On-Line Cookbook*

To our wives
Margaret Cave
Lois Maymon

Preface

There was a time when computer programs were used primarily by the programmers who wrote them. A 16K word machine was considered very large, transistors did not exist, and neither did the term "software." Today, with the advent of timesharing and the personal computer, millions of people can use the same computer programs (software). Machines with millions of words of memory can sit on a deck like a typewriter, and their users easily can access trillions of additional words of data over standard telephone lines. The transistor as a separate circuit element has come and gone, replaced by whole computer processors. One wonders if the field of software will move as fast as that of computer hardware.

The driving force behind the rapidly moving field of computer hardware today is simply economics. It is a highly competitive environment, one in which the survivors give the customer the best price for a desired amount of computing power, in a form that is "user friendly." In fact, user friendliness is fast becoming the most important requirement in the computer field—and this is where software can take on the growth pattern of hardware.

Just as semiconductor companies make greater profits in producing products that are directly useful in a broad market, so can software companies. However, this requires the realization that the end-user environment is significantly different from the development environment. Judgments on what the end user really wants must be made in close concert with those who represent the market. Users will expect to receive a product they can easily use directly, without help or a significant learning process.

To cut the costs of supporting such a market, developers must be able to deliver systems from a distance, through standard distribution channels, without much (if any) direct contact with the customer. This requires that the software product be of high quality, both from a user and support standpoint. The software features must be easily understandable and available—on a user-friendly basis. The software must be designed to be readily supported and, in particular, easily enhanced with new features and functions. A good piece of software has a life expectancy of 15 to 20 years. It never dies. It just grows and grows!

The *incremental method* represents a pragmatic approach to managing the development and support of user-oriented software products over their lifecycles. It represents an attempt to organize and integrate a number of concepts and techniques that have evolved in the successful development of a number of large software products. We are presenting no new material. To the contrary, we have tried to sift out and use only those methods that have proved successful as gauged within a competitive software vendor environment. Unproven approaches do not fit this context. They typically carry a great risk of failure and should be avoided in a production environment. And that is what this book is about. It is aimed at helping software product managers to maximize their return on investment in software assets, while keeping risk at an acceptable level.

Contents

CHAPTER 1

Gaining Management Perspective

1.1 THE INCREMENTAL METHOD

When people invest money in stocks, mutual funds, and money markets, return on investment (ROI) is scrutinized carefully. Managers of such funds try to ensure a high probability for returning good profits along with capital. This same measure of success underlies large construction projects, including investments in new plants and new equipment. Because most people are willing to take greater risks with other people's money, income for contractors and managers of such projects must be based on performance, thereby reducing the purchaser's risk. *When tomorrow's budgets depend on today's project successes or failures, great heed must be paid to the size of risks incurred.*

Software companies that build and sell packaged products must carefully scrutinize the probability that profits will be returned on their investments. Management must carefully assess the risk of converting capital assets (which are usually scarce) into software assets that do not generate a profit. Herein lies the problem. The complexity of software, its intangible nature, and the fact that it helps implement user policy makes measuring and controlling the risk very difficult.

The *incremental method* of risk management provides a framework for partitioning software projects into "risk increments," with resources committed on a correspondingly incremental basis. The size of each increment is determined by two considerations:

- The size of each resource commitment must be commensurate with the degree of risk associated with the work to be done.
- When resources are expended and the work increment completed, the risk of project failure must have been lowered accordingly.

Risk can be quantified by defining it as the probability of *not* reaching a point where revenues (or savings) will exceed expenditures; it can be defined as the probability of exceeding budgets to the extent that the project is considered a failure; or it can be defined as the probability of having management consider the project a failure.

The incremental method responds to the observation that certain software developers have been consistently more successful than others. As used here, "success" is the delivery of a quality product, on time and within budget, that results in a high degree of *user satisfaction*. The key parameter for measuring project success is obviously user satisfaction, but when user requirements force cost estimates to exceed budget constraints, it may be necessary to relax either user requirements or budget constraints, or to halt the project. In any case, *the decision to continue a project under uncertain terms places the responsibility for failure on management.* It is essential that management provide step-by-step decision points at times when halting the project causes minimal loss of resources and continuity (especially if continuation can be justified at a later time).

The incremental method provides specific procedures and standards for reducing risk of project failure as resources are invested across the software lifecycle. The method applies to the development of automated systems in which software is a major component along the critical path. It is particularly applicable to systems involving a high degree of human interaction. The objective of the incremental method is to prepare managers for the opportunity to develop major software systems with an approach that will markedly increase the probability of success.

1.2 TYPES OF SOFTWARE PROJECTS

Software projects are generally spawned from external product-requesting sources. The request typically specifies one of two types of systems: custom systems or product systems.

Custom Systems

A custom system is generally a one-of-a-kind software system specifically requested via a request for proposal (RFP) from a source outside the software development group. Typically, a contract to develop the system is agreed on after a competitive bidding process in which the prospective developers present proposals for solving the problem. For custom systems the developer must carefully analyze all of the requestor's requirements to ensure a reasonable profit margin and provide a clear understanding of all continuation or support obligations. A sophisticated requester usually provides such expectations in the RFP.

Product Systems

A product system is one that will be sold many times. The requesting source for such systems is generally the developer's product research department. The company's internal-marketing or product-planning organization issues a product proposal that describes, in general terms, the new product's characteristics, its place in the company's overall product line, and its relationship to, or advantages over, competitive products. The basic purposes of the proposal are to demonstrate a need or opportunity to general management and to secure commitment of the resources necessary to develop the product. The proposal specifically contains the following information.

1. A brief functional description of the proposed product in terms of customer needs satisfied
2. A general statement of the target market, the company's current position in that market, and the benefits to be derived by developing the proposed product.
3. A summary analysis of major competitors and their products currently offered, or anticipated to be offered, to the target market
4. A development strategy including resource needs and cost requirements, timing requirements for introducing the product to the market, effects on existing company product lines, pricing requirements for amortization goals, and market introduction plans

The incremental method, in this case, applies to the process of justifying and obtaining the initial funding from general management so that the developing organization can produce a problem definition and project a development plan. The company is not committing itself to a full-blown, long-range development program at this time, with all of the risks and uncertainties such a commitment entails; rather it is making a modest

investment to determine the feasibility of pursuing the development. If this initial phase reveals the product to be ill advised or impractical, little has been lost, and an expensive disaster has been avoided.

Although this book generally discusses the software product lifecycle (the second type of system described), the incremental method applies equally well to custom systems.

1.3 A REFERENCE FRAME FOR SCOPING PROJECTS

The economics of any product lifecycle depends both on the development environment and the operational (user) environment. In the case of software, the scope of problems encountered in each environment varies considerably. Software systems vary according to numbers of field installations, types of users, previous experience of both users and developers, complexity of the system due to size (e.g., number of programs, files, and so forth), and the research and development efforts required to solve technical problems.

We need a frame of reference for scoping software projects because we want to be able to estimate as accurately as possible the size of planned projects. Furthermore, we would like to look back at past project successes or failures and be able to draw accurate conclusions regarding the decisions made and value of methods used.

The reference frame used within the incremental method classifies a project according to the following dimensions:

- Quality of the software product: requirements on system functional availability, reliability, and ability to respond to problems
- Support requirements: number of geographically separate installations to be supported
- User orientation: numbers and different types of users expected to interact with the system
- Prior history: history of prior automation experience with the particular application, and difficulties encountered
- Degree of complexity: number of programs, individual program complexity, and hardware complexity

Clearly, developing a single program for use by one person is far different from developing a multifaceted system for many users expecting a high-quality product at many different installations. In addition, the software developer often must contend with the parallel problem of educating inexperienced users while the software is being specified and introduced. Well-prepared, experienced management is essential in dealing with such situations.

The quality of a software product depends on the availability of system functions required by the user and the cost of maintaining that

availability. Naturally, the developer constantly tries to "predict" and "control" quality from the inception of the system lifecycle; however, *in practice the final measure of quality cannot be determined until the system is in the hands of users in operational installations.*

In a multiple-installation environment, requirements for system availability and supportability are of major importance. As the number of installations and corresponding support costs rise, reliability and support requirements multiply. The nature of the support environment amplifies the need for design quality. Add to this a competitive requirement of satisfying user needs under warranty, and the result is an environment that places the highest demands on product quality and formal management control. In the commercial software vendor environment, it is difficult to sell software products without long-term warranties. Furthermore, Japanese producers are gearing up to compete. The Ministry of International Trade and Industry (MITI) has formed a review board that issues 25-year warranties on software. This competitive arena is the principal environment addressed by this book.

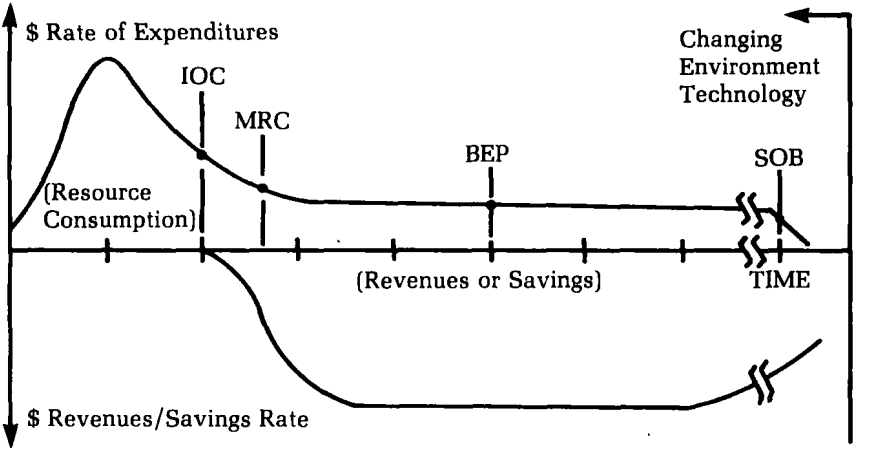
1.4 DEFINING THE SOFTWARE LIFECYCLE CONTROL PROBLEM

To start, we consider a graphical description of the lifecycle control problem. The top curve in Figure 1.1 shows rate of expenditures (dollars per month or per year) and represents resource consumption for a software product (see Norden, 1970). Assume that the rate of revenues or savings (bottom curve) can be measured in the same way as are resources consumed (dollars per month or per year). From this one can derive a clear measure (dollars and cents) of the return on investment (ROI) that can be achieved.

Such a measure can be precisely determined for a software vendor marketing a package product. This measure is also valid for products in which savings can be clearly determined. In cases where these measures are not so clear, there appears to be little doubt that the incremental method is directly applicable, although its evaluation may be more subjective.

The curves shown in Figure 1.1 provide a graphical representation of the software lifecycle control problem. This graphical representation is a general one because, for a given software problem, there can be an infinite set of possible curves. The curve's actual shape depends on judgments of the project manager as well as on many other factors across the lifecycle. The technical problem objective is to shape these curves in such a way as to maximize ROI.

Managers of large software projects must face many external con-



IOC — Initial Operational Capability
MRC — Maximum Resource Consumption
BEP — Break Even Point
SOB — System Obsolescense

Figure 1.1. The Software System Lifecycle.

straints. First, managers must determine the time period during which a market for the product is expected. In other words, they must determine the period during which revenues for the system will flow or savings will accrue. They must consider the changing environment, changing technology, competition, and the fact that at some time in the future the technology or market requirement will become obsolete.

If a market already exists for the system, the start of revenues, or initial operational capability (IOC), will depend only on development time to achieve the goal. The period for revenues is bound by IOC and system obsolescence (SOB). Obviously, if more money is spent up front to shorten the time to IOC, revenues will begin to come in earlier. However, the experienced project manager is aware that there is no even dollar-for-dollar tradeoff, and, in fact, the cost to do things faster can rise exponentially.

Other external financial constraints that must be considered are maximum resource consumption (MRC) and break-even point (BEP). MRC occurs when current (for example, monthly) revenues or savings exceed current expenses. Enough financial resources must be available at the rate required to cover those utilized to MRC. If this point is not estimated properly and, consequently, never reached, the project can terminate because of lack of funds. BEP occurs when total expenses have been recovered via revenues or savings. Vendor profits cannot be realized until this point, except through long-term capitalization of the up-front peak costs. In any event, a vendor facing many years to BEP, will be considering other places to invest resources.