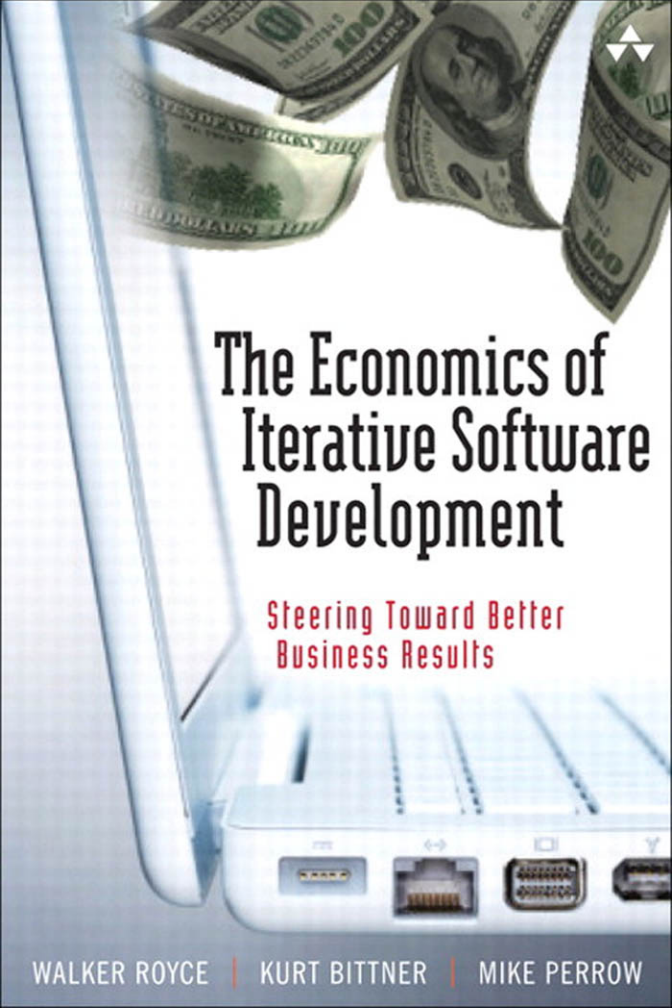
The Economics of Iterative Software Development
Steering Toward Better Business Results
WALKER ROYCE | KURT BITTNER | MIKE PERROW

Library of Congress Cataloging-in-Publication Data

Royce, Walker
The economics of iterative software development : steering toward better business results / Walker Royce, Kurt Bittner, Mike Perrow.
p. cm.
Includes bibliographical references and index.
ISBN 978-0-321-50935-2 (hardcover : alk. paper)
1. Computer software—Development—Management. 2. Software engineering. I. Bittner, Kurt. II. Perrow, Mike. III. Title.

QA76.76.D47R685 2009
005.1—dc22

2009003221

ISBN-13: 978-0-321-50935-2
ISBN-10: 0-321-50935-8
Text printed in the United States on recycled paper at R.R. Donnelley in Crawfordsville, Indiana.
First printing, March 2009

PREFACE

Imagine you're a wealthy, seasoned traveler planning a month-long, multi-country vacation. Sound nice? Go ahead, pick a continent, some part of the world you always wanted to explore. Where do you begin? If you had the luxury to actually consider such a notion, you might quickly determine your starting point and where you'd eventually end. You'd imagine a sequence of smaller journeys to famous cities, mountains, seaside resorts. But as you began scoping out the general profile of your trip, would you begin planning every meal, every evening stroll, every purchase you'd be making? Of course not. No one can plan in advance exactly what to do in all those unknown places, or exactly how to use the time and resources available along the way. Besides, you know that the quality of a journey is bound to suffer if you try sticking to rigid plans designed before you set out.

This book is about managing software development projects, which are seldom confused with long vacations. But from a management perspective, they have many things in common, all having to do with the unfamiliar—unfamiliar territory, languages, personal behavior, practices, costs, and infrastructure. This book is based on experiences that have involved thousands of miles of travel and thousands of hours of hard work alongside businesses whose software development teams encounter these uncertainties with every

project. The most successful of these teams plan their projects at a high level first, then, like seasoned travelers, they plan in smaller steps called *iterations* as their journey progresses.

We attempt to explain those successes in terms most managers interested in improving business results will understand. With results in mind, we don't assume a great deal of technical experience on the part of the reader, but we do assume a commitment to successful leadership. This book targets readers who find themselves in leadership positions at various levels in a business organization, especially organizations that acquire, manage, or develop software as a component of business strategy. Our objective is to describe the benefits of frequent course correction during the iterative project, how to measure the interim results, and how the overall approach contributes meaningfully to the bottom line.

This last point has everything to do with the underlying theme of this book: economics. In the broadest sense, good economics means efficient management of finite resources toward an optimal result. Software economics is based on these same principles. We spend some time exploring poor economics based on old-fashioned management styles, including the inefficiencies that occur when software projects are managed as if they were traditional engineering projects, such as the construction of a bridge. When it comes to software construction, these inefficiencies are costly in terms of time, budget, and missed opportunities in the competitive marketplace. By contrast, modern iterative development methods will improve results based on practical governance of your team's finite resources; hence, the title of this book.

The order of our parts and chapters is straightforward. Part I, "The Software-Driven Economy," presents the context for software development and management in today's business climate, the difficulties of success, as well as the consequences of failure. Part II, "Improving Software Development Economics," focuses on a modern approach to software engineering based on the principles and practices of iterative development. We certainly don't know everything.

But through decades of observation we know what *doesn't* work, and we have learned quite a bit about what does. Part III, "Practical Measurement for Software Engineering," offers a more detailed look at how you can be sure these techniques work—through measurement. As an update on the tenet that "you can't manage what you can't measure," this final portion of the book focuses on the purposes of metrics, including the means by which variance can be reduced throughout the project lifecycle.

Whether you're a seasoned software project manager looking for a relatively brief review of iterative development principles, or a novice looking for a digestible introduction to these concepts, I think you'll find this book valuable.

Mike Perrow
Medford, Massachusetts

3

TRENDS IN SOFTWARE ECONOMICS

Over the past two decades, the software industry has moved progressively toward new methods for managing the ever-increasing complexity of software projects. We have seen evolutions and revolutions, with varying degrees of success and failure. Although software technologies, processes, and methods have advanced rapidly, software engineering remains a people-intensive process. Consequently, techniques for managing people, technology, resources, and risks have profound leverage.

The early software approaches of the 1960s and 1970s can best be described as craftsmanship, with each project using custom or ad-hoc processes and custom tools that were quite simple in their scope. By the 1980s and 1990s, the software industry had matured and was starting to exhibit signs of becoming more of an engineering discipline. However, most software projects in this era were still primarily exploring new technologies and approaches that were largely unpredictable in their results and marked by diseconomies of scale. In recent years, however, new techniques that aggressively attack project risk, leverage automation to a greater degree, and exhibit much-improved economies of scale have begun to grow in acceptance. Much-improved software economics

are already being achieved by leading software organizations who use these approaches.

Let's take a look at one successful model for describing software economics.

A SIMPLIFIED MODEL OF SOFTWARE ECONOMICS

There are several software cost models in use today. The most popular, open, and well-documented model is the COnstructive COst MOdel (COCOMO), which has been widely used by the industry for 20 years. The latest version, COCOMO II, is the result of a collaborative effort led by the University of Southern California (USC) Center for Software Engineering, with the financial and technical support of numerous industry affiliates. The objectives of this team are threefold:

- To develop a software cost and schedule estimation model for the lifecycle practices of the post-2000 era
- To develop a software project database and tool support for improvement of the cost model
- To provide a quantitative analytic framework for evaluating software technologies and their economic impacts

The accuracy of COCOMO II allows its users to estimate cost within 30% of actuals, 74% of the time. This level of unpredictability in the outcome of a software development process should be truly frightening to any software project investor, especially in view of the fact that few projects ever perform better than expected.

The COCOMO II cost model includes numerous parameters and techniques for estimating a wide variety of software development projects. For the purposes of this discussion, we will abstract COCOMO II into a function of four basic parameters:

- **Complexity.** The complexity of the software solution is typically quantified in terms of the size of human-generated components (the number of source instructions or the number of function points) needed to develop the features in a usable product.
- **Process.** This refers to the process used to produce the end product, and in particular its effectiveness in helping developers avoid "overhead" activities.
- **Team.** This refers to the capabilities of the software engineering team, and particularly their experience with both the computer science issues and the application domain issues for the project at hand.
- **Tools.** This refers to the software tools a team uses for development—that is, the extent of process automation.

The relationships among these parameters in modeling the estimated effort can be expressed as follows:

$$\text{Effort} = (\text{Team}) \times (\text{Tools}) \times (\text{Complexity})^{(\text{Process})}$$

Schedule estimates are computed directly from the effort estimate and process parameters. Reductions in effort generally result in reductions in schedule estimates. To simplify this discussion, we can assume that the "cost" includes both effort and time. The complete COCOMO II model includes several modes, numerous parameters, and several equations. This simplified model enables us to focus the discussion on the more discriminating dimensions of improvement.

What constitutes a good software cost estimate is a very tough question. In our experience, a good estimate can be defined as one that has the following attributes:

- It is conceived and supported by a team accountable for performing the work, consisting of the project manager, the architecture team, the development team, and the test team.

- It is accepted by all stakeholders as ambitious but realizable.
- It is based on a well-defined software cost model with a credible basis and a database of relevant project experience that includes similar processes, similar technologies, similar environments, similar quality requirements, and similar people.
- It is defined in enough detail for both developers and managers to objectively assess the probability of success and to understand key risk areas.

Although several parametric models have been developed to estimate software costs, they can all be generally abstracted into the form given above. One very important aspect of software economics (as represented within today's software cost models) is that the relationship between effort and size exhibits a diseconomy of scale. The software development diseconomy of scale is a result of the "process" exponent in the equation being greater than 1.0. In contrast to the economics for most manufacturing processes, the more software you build, the greater the cost per unit item. It is desirable, therefore, to reduce the size and complexity of a project whenever possible.

SOFTWARE ENGINEERING: A 40-YEAR HISTORY

Software engineering is dominated by intellectual activities focused on solving problems with immense complexity and numerous unknowns in competing perspectives. We can characterize three generations of software development as follows:

1. *1960s and 1970s: Craftsmanship.* Organizations used virtually all custom tools, custom processes, and custom components built in primitive languages. Project performance

was highly predictable but poor: Cost, schedule, and quality objectives were almost never met.

2. ***1980s and 1990s: Early Software Engineering.*** Organizations used more repeatable processes, off-the-shelf tools, and about 70% of their components were built in higher level languages. About 30% of these components were available as commercial products, including the operating system, database management system, networking, and graphical user interface. During the 1980s, some organizations began achieving economies of scale, but with the growth in applications' complexity (primarily in the move to distributed systems), the existing languages, techniques, and technologies were simply insufficient.
3. ***2000 and later: Modern Software Engineering.*** Modern practice is rooted in the use of managed and measured processes, integrated automation environments, and mostly (70%) off-the-shelf components. Typically, only about 30% of components need to be custom built.

Figure 3.1 illustrates the economics associated with these three generations of software development. The ordinate of the graph refers to software unit costs (per source line of code [SLOC], per function point, per component—take your pick) realized by an organization. The abscissa represents the lifecycle growth in the complexity of software applications developed by the organization.

Technologies for achieving reductions in complexity/size, process improvements, improvements in team effectiveness, and tool automation are not independent of one another. In each new generation, the key is complementary growth in all technologies. For example, in modern approaches, process advances cannot not be used successfully without component technologies and tool automation.

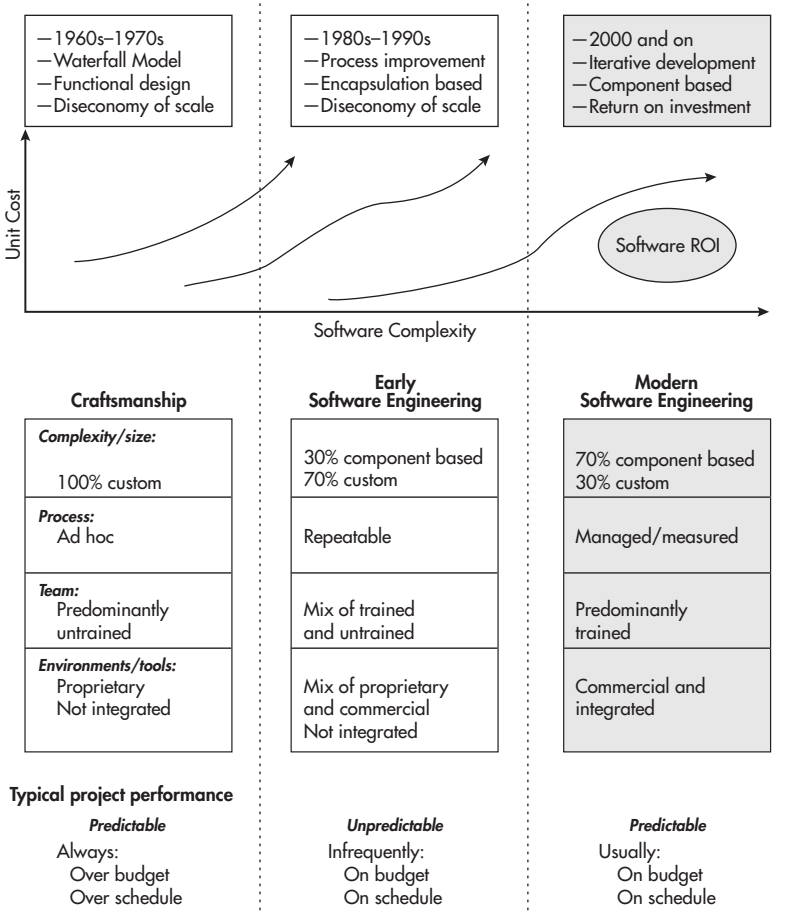

FIGURE 3.1 Trends in software economics

KEYS TO IMPROVEMENT: A BALANCED APPROACH

Improvements in the economics of software development have been not only difficult to achieve, but also difficult to measure and substantiate. In software textbooks, trade journals, and market literature, the topic of software economics is plagued by inconsistent jargon, inconsistent units of measure, disagreement among experts, and

unending hyperbole. If we examine only one aspect of improving software economics, we are able to draw only narrow conclusions. Likewise, if an organization focuses on improving only one aspect of its software development process, it will not realize any significant economic improvement—even though it may make spectacular improvements in this single aspect of the process.

The key to substantial improvement in business performance is a balanced attack across the four basic parameters of the simplified software cost model: complexity, process, team, and tools. These parameters are in priority order for most software domains. In our experience, the following discriminating approaches have made a difference in improving the economics of software development and integration:

1. Reduce the size or complexity of what needs to be developed.
    - Reduce the amount of developed code by understanding business needs and delivering only that which is absolutely essential to satisfying those needs.
    - Reduce the amount of human-generated code through component-based technology and use of higher levels of abstraction.
    - Reuse existing functionality, whether through direct code reuse or use of service-oriented architectures.
    - Reduce the amount of functionality delivered in a single release to shorten the release cycle and reduce complexity; deliver increments of functionality in a series of releases.
2. Improve the development process.
    - Reduce scrap and rework by transitioning from a waterfall process to a modern, iterative development process.
    - Attack significant risks early through an architecture-first focus.
    - Evaluate areas of inefficiency and ineffectiveness and improve practices in response.

3. Create more proficient teams.
    - Improve individual skills.
    - Improve team interactions.
    - Improve organizational capability.
4. Use integrated tools that exploit more automation.
    - Improve human productivity through advanced levels of automation.
    - Eliminate sources of human error.
    - Support improvements in areas of process weakness.

Most software experts would also stress the significant dependencies among these trends. For example, new tools enable complexity reduction and process improvements; size-reduction approaches lead to process changes; and process improvements drive tool advances.

In addition, IT executives need to consider other trends in software economics whose importance is increasing. These include the lifecycle effects of commercial components-based solutions and rapid development (often a source of maintenance headaches); the effects of service-oriented architectures; the effects of user priorities and value propositions (often keys to business case analysis and to the management of scope and expectations); and the effects of stakeholder/team collaboration and shared vision achievement (often keys to rapid adaptation to changes in the IT marketplace).

SUMMARY

The evolution of software project management since the 1960s has moved through the stages of individual craftsmanship, through the application of engineering principles, to the beginnings of repeatable, somewhat predictable processes based on a better understanding of project risk coupled with the use of automation in the process.

This rise in predictability has allowed the emergence of cost estimation techniques, the most popular of which is COCOMO II.

The cost of a software project can best be estimated in terms of the four essential COCOMO II parameters: complexity, process, teams, and tools. Cost improvements result when the following occur:

1. Complexity can be reduced, either in the finished product or in the iterations produced during the project lifecycle.
2. The process can be improved by addressing risks first and reducing human error through automation.
3. Teams can become more efficient through skill enhancement and improved communication.
4. Automated tools can be used to heighten productivity and strengthen areas of the process.

In the next sections, we will elaborate on the approaches listed above for achieving improvements in each of the four dimensions. These approaches represent patterns of success we have observed among successful software development organizations that have made dramatic leaps in improving the economics of their software development efforts.

INDEX

D

E

F

G

H

N

O

P

Q

R

S

T

U

V

W