1

Return on Software: Maximizing the Return on Your Software Investment

Almost every software organization on the planet is in the unenviable position of having to do the best it can with limited resources. We could always do more, and we could probably do it better, if we just had more people, more time, or more money. How do we get the most out of the resources we do have? How do we maximize our "bang for the buck"? That's what this book is about—helping you, the practicing software professional (or, the software professional-in-training), make purposeful, appropriate, business-conscious technical decisions so that you and your employer can get the most out of the limited resources you do have. This chapter explains why software professionals need the concepts and techniques in this book and gives a survey of the rest of the book.

Software on Purpose

There are hundreds, if not thousands, of books on how to develop software. Books on C, C++, Java, CORBA, XML, databases, and the like abound. However, most software organizations don't have a very good track record with the software they develop. After studying thousands of software projects, the Standish Group observed that about 23% of software projects fail to deliver any working software at all [Standish01a]. Unfortunately, these projects aren't being cancelled until well after their original schedule and budget have been exceeded.

The Standish study also showed that for projects that do deliver software, the average one is 45% over budget, 63% over schedule, and delivers only 67% of the originally planned features and functions. Based on our industry's track record, a software project that's estimated to take 12 months and cost $1 million can be reasonably expected to take closer to 20 months and cost about $1.5 million, while meeting only two thirds of its requirements.

Tracy Kidder [Kidder81] reports that about 40% of the commercial applications of computers have proven uneconomical. These applications don't show a positive return on investment in the sense that the job being automated ended up costing more to do after the system was installed than it did before. Return on investment is defined in Chapter 8, but, simply, those organizations paid more to develop the software than the software ever earned back for them.

Assuming the Standish and Kidder data can be combined, the resulting statistics are rather grim. If 23% of all software projects are cancelled without delivering anything, and 40% of the projects that do deliver software are net money losers, then about 54% of all software projects are counterproductive in the business sense. Over half the time, the organizations that paid for software projects would actually have been better off financially had they never even started those projects.

The total amount of money spent on software development in the United States has been estimated to be more than $275 billion annually [Standish01b]. This means a sizeable amount of money is being wasted every year—around $63 billion in cancelled software projects alone. The money wasted annually could be as much as $149 billion if projects not showing a positive return on their investment are included. These numbers may even be conservative when you consider that larger projects are much more likely to fail than smaller projects [Standish01b], [DeMarco99]. Be aware that this cost data is for the United States only; there's a lot of software development going on outside the United States. There's not necessarily any reason to believe that software organizations outside the United States are any more—or any less—successful, so the worldwide annual results could be staggering.

There might be a million and one different reasons for the poor software project performance observed by the Standish Group. Maybe

- The customer's requirements and specifications were incomplete, vague, or ambiguous.
- Those requirements kept changing throughout the project.
- Bad design decisions were made.
- The staff didn't have enough expertise in new technologies used on the project.
- The projects weren't given enough resources to be successful.
- The projects weren't sufficiently planned and managed.
- The project's externally imposed deadlines were unrealistic to begin with.
- . . .

Underlying all of these reasons is the more fundamental reason of bad business decisions being made. Either consciously or unconsciously *someone* decided to

- Not provide the project team with complete, precise requirements
- Allow the requirements to change throughout the project without considering—or maybe even being aware of—the effect of requirements change on project success
- Use an inappropriate design
- Not properly address—or even consider—the risks and uncertainties new technologies impose on software projects
- Not provide enough resources for the project to be successful
- Not sufficiently plan or manage the project
- Impose unrealistic deadlines on the project
- . . .

In spite of there being so many books on how to develop software, there aren't many books on why that software is being developed in the first place. Knowing why the software is being developed will help decision makers make better business decisions. This book doesn't say anything about how to develop software. It's all about why, and why not.

Waste Not, Want Not

Even in the best of financial times, a software organization shouldn't be sloppy or wasteful with its resources: people, money, and time. There will always be more functions that could be added to the existing software if there were just a few more people around to do the work. There will always be more new software that could be developed if we just had a bit more money. There will always be a few more defects in existing systems that could be fixed if we just had a bit more time to fix them.

When financial times get tough, there are even fewer people around to do the work. There is also less money. But getting the work done quickly is even more critical than before. In tough financial times, it's even more important for the organization to use its resources wisely. A wasted person-day, a wasted dollar, or a wasted calendar-day will always be just that: wasted. As resources get scarcer, it becomes that much more important to get the best return out of your software investment.

This book is about getting the most out of your software investment. A lot of time and money has been spent on software since the first programs were written. Some of it was spent wisely, but some of it was not. Regardless of whether it will be spent wisely or not in the future, people will continue to spend time and money on software. So how will you know if *your* organization's time and money are being well spent? How can you find out if you'd get more return from investing your limited resources in some other way? When your boss asks you, "Is this the best way for us to be spending our limited time and money?" how can you answer in a way that gives your boss confidence you really know what you're talking about?

The Primary Message

This book is about engineering economy, it's about aligning software technical decisions with the business goals of the organization. Many software professionals not only don't know how to look at the business aspects of their technical decisions, they don't even know that it's important to do so. Decisions such as "Should we use eXtreme Programming or should we use the Rational Unified Process on this project?" may be easy from a purely technical perspective, but those decisions can have serious implications on the business viability of the software project and the resulting software product. From my own experience teaching object-oriented development, I've asked more than 1,500 students why they were learning object-orientation. Reasons such as "It will help me develop higher quality software, quicker" or "It will be good for the company's bottom line" were extremely rare—fewer than 100 students total ever gave this kind of answer. The vast majority of the students answered, essentially, "It'll look good on my resumé."

In another case I was developing software to monitor radiation at nuclear power plants. Part of that software needed a sorting routine. I wrote a simple insertion sort routine and had that part of the system running in a matter of hours. A coworker insisted on developing a QuickSort routine because "everybody knows that QuickSort is better than insertion sort." At that time (early 1980s) reusable QuickSort routines weren't available; if you wanted one, you had to write it yourself. Unfortunately, QuickSort is a recursive algorithm, and the programming language we were using, Fortran-IV, didn't support recursion. My coworker spent more than a week developing QuickSort in Fortran-IV. Only later did he realize that the list that needed sorting averaged only about 30 entries and was predominantly sorted to begin with. Small lists that are already mostly sorted cause QuickSort to have extremely poor performance, typically worse than simpler algorithms such as insertion sort. Moreover, sorting happened in this system fewer than 50 times a day. Even if QuickSort did perform better than insertion sort, it would take decades for the company to recover its investment. My coworker's effort turned out to be a pretty big waste of the company's money and time.

The object-orientation and QuickSort examples are just two simple examples. Over the years I've seen technical decisions be inconsistent with the organization's business goals far more often than I've seen them be consistent. The software industry is hardly unique, however. This isn't the only time in history when the business impact of technical decisions was questionable. Eugene Grant [Grant90] wrote the following, referring to Arthur Wellington (a pioneer in the field of engineering economy).

> Railway location obviously is a field in which many alternatives are likely to be available. Nevertheless, Wellington observed what seemed to him to be an almost complete disregard by many locating engineers of the influence of their decisions on the prospective costs and revenues of the railways. In his first edition (1877) he said of railway location, "And yet there is no field of professional labor in which a limited amount of modest incompetency at $150 per month can set so many picks and shovels and locomotives at work to no purpose whatsoever."

The average salary of software professionals today is well over $150 per month, but are our decisions really that much better than the railway-locating engineers' of the late 1800s? As software professionals, we had better be concerned with the impacts of our technical decisions on our employer—I'd say that Wellington's "*a limited amount of modest incompetency*" describes the contemporary software industry quite well.

This book bridges the gap between software technical decisions and business goals. The concepts and techniques in this book will allow you—the practicing software professional—to align your technical decisions with the business goals of your organization. This will help you waste less of your employer's limited time and money. The fundamental question that software professionals should always ask is, "Is it in the best interest of the organization to invest its limited resources in this technical endeavor, or would the same investment produce a higher return elsewhere?"

A Secondary Message: Software Engineering Versus Computer Science

Many software professionals like to refer to themselves as "software engineers." Unfortunately, simply wanting our work to be considered engineering and continually saying that it is doesn't make it so. In several U.S. states, including New York and Texas [TBPE98], the term "engineer" is actually a legally reserved word. Those who inappropriately, or even inadvertently, misuse the term—such as calling themselves a software engineer—without meeting legally defined criteria can be subject to civil or criminal penalties. Similarly, under the law in Canada no one can call himself or herself an engineer unless licensed as such by the provincial engineering societies.

Another message in this book is the relationship between software engineering and computer science. There has been a fair amount of debate over the similarities and differences between the two. Instead of endlessly discussing opinions, we can look at "first principles"—what do scientists believe it means to be a scientist and what do engineers believe it means to be an engineer?

Science is defined as [Webster94]

> a department of systematized knowledge as an object of study; knowledge or a system of knowledge covering general truths or the operation of general laws esp. as obtained and tested through scientific method.

The Accreditation Board of Engineering and Technology (ABET) is the recognized authority for accrediting engineering and technology degree programs at colleges and universities in the United States. ABET defines engineering as [ABET00]

> the profession in which a knowledge of the mathematical and natural sciences gained by study, experience, and practice is applied with judgment to develop ways to utilize, economically, the materials and forces of nature for the benefit of mankind.

DeGarmo et al. [DeGarmo93] paraphrase the definition of engineering as

> finding the balance between what is technically feasible and what is economically acceptable.

Arthur Wellington offers a somewhat more lighthearted description [Wellington1887]:

> It would be well if engineering were less generally thought of, and even defined, as the art of constructing. In a certain sense it is rather the art of not constructing; or, to define it rudely but not inaptly, it is the art of doing that well with one dollar which any bungler can do with two.

Comparing and contrasting these definitions shows that science is the pursuit of knowledge and engineering is the application of that knowledge for the benefit of people. As an example, chemistry as a science is concerned with expanding our knowledge of chemical processes so we can better understand and explain phenomena observed in the universe. Chemical engineering, on the other hand, applies the knowledge derived from this "chemical science" to filling human needs. At the core of chemical engineering is an understanding of the body of chemical theory. In addition, chemical engineering calls upon the practical aspects of chemical processes, such as the design of pressure vessels and waste-heat removal mechanisms, together with the use of engineering economy as the basis for decisions.

The science branch and the engineering branch of a technical discipline are related but distinct. The science branch is concerned with expanding the body of theoretical knowledge about that discipline, whereas the engineering branch is concerned with the practical and economical application of that theoretical knowledge. The following equation is a simplified description of the general relationship between science and engineering:

$$\text{Engineering} = \text{Scientific theory} + \text{Practice} + \text{Engineering economy}$$

People who are recognized engineers (for instance, civil, mechanical, chemical, aeronautical) are usually required to take a course in engineering economy as part of their undergraduate education.

Based on the dictionary definition of science, above, computer science can be defined as

> a department of systematized knowledge about computing as an object of study; a system of knowledge covering general truths or the operation of general laws of computing esp. as obtained and tested through scientific method.

Based on the ABET definition of engineering, software engineering can be defined as

> the profession in which a knowledge of the mathematical and computing sciences gained by study, experience, and practice is applied with judgment to develop ways to utilize, economically, computing systems for the benefit of mankind.

So, from the equation above we can derive

Software engineering = Computer science + Practice + Engineering economy

Both computer science and software engineering deal with computers, computing, and software. The science of computing, as a body of knowledge, is at the core of both. Computer science is concerned with computers, computing, and software as a system of knowledge, together with expanding that knowledge. Software engineering, on the other hand, should be concerned with the application of computers, computing, and software to practical purposes, specifically the design, construction, and operation of efficient and economical computing systems.

The software industry as a whole has shown movement, albeit slow, toward becoming a legitimate engineering discipline: true software engineering. For that transition to fully take place, software professionals will need to learn about—and use—engineering economy as the basis for their technical decisions. This book is an engineering economy reference book for software professionals. It covers the same topics as would be found in its typical industrial engineering counterpart. Largely, only the examples have been translated into a software context.

Lack of engineering economy isn't the only issue preventing software from being generally recognized as a legitimate engineering discipline. For other perspectives on this topic see, for example, [Hooten90], [McConnell03], [Shaw90], and [SWEBOK01].

There will always be a need for qualified computer scientists to continue the advancement of computing theory. To be sure, every recognized engineering discipline has a corresponding science that is populated by dedicated researchers. The computer science curriculum is appropriate for meeting this need. But the software industry also has a distinct need for [Ford91]

> a practitioner who will be able to rapidly assume a position of substantial responsibility in an organization.

Providing those qualified practitioners should be the primary goal of software engineering education.

An Overview of the Book

Here is a quick look at the topics that are covered in this book. The book is divided into eight parts:

- **Part I: Introduction and Foundations**. This part introduces the subject and provides the background needed to understand the rest of the book. Topics include the fundamental concepts of business decisions, the business decision-making process, the time value of money (interest), financial equivalence, and ways to characterize proposed solutions including present worth, internal rate of return, and discounted payback period.

- **Part II: Making For-Profit Business Decisions.** In this part, the basic mechanics of making business decisions in for-profit organizations are presented. Specific topics are for-profit decision analysis, the concept of economic life and its impact on planning horizons, and two special cases in for-profit decision analysis: replacement decisions and asset-retirement decisions.
- **Part III: Advanced For-Profit Decision Techniques**. This part presents additional concepts and techniques that may be included in a for-profit decision analysis. These techniques don't always need to be applied; you would only use them when you need more precision in the decision analysis. The topics in this part are inflation and deflation, depreciation, general accounting, income taxes, and the consequences of income taxes on business decisions.
- **Part IV: Making Decisions in Government and Nonprofit Organizations**. This part explains the concepts and techniques for decision making in government agencies and in not-for-profit organizations. Specific topics are benefit-cost analysis and cost-effectiveness analysis.
- **Part V: Present Economy.** In this part, the concepts and techniques of break-even analysis and optimization analysis are discussed.
- **Part VI: Estimation, Risk, and Uncertainty**. Estimation is an essential part of business decision analysis. This part goes into detail about the concepts and techniques of estimation and explains risk and uncertainty and how they can influence, and be addressed in, decisions.
- **Part VII: Multiple-Attribute Decisions**. Parts I through VI explain how to make decisions when there is one decision criterion, money. Money will usually be the most important decision criterion, but it is often only one of several important decision criteria. This part presents several different techniques for addressing more than just one decision criterion, or attribute, in a decision analysis.
- **Part VIII: Summary**. This part summarizes the book.

Summary

Almost every software organization that has ever existed has had to deal with limited resources. However, these same software organizations have tended to not be very efficient or effective with the resources they do have. About 23% of all software projects are cancelled without delivering any usable software at all. Of the software projects that do deliver, they tend to run about 45% over budget, 63% over schedule, and satisfy only 67% of the original requirements.

There may be many specific reasons for this level of performance, however they almost all boil down to one underlying reality: Inappropriate decisions are being made somewhere in the organization. Maybe the inappropriate decision was to do the project at all. Maybe the inappropriate decision was to provide insufficient funding, inadequate staff, poor requirements, or overconstrain the project schedule (or all of these combined). Maybe the inappropriate decision was about how to plan or manage the project. Maybe the project team members themselves made inappropriate decisions. By being more careful about aligning software technical decisions with business goals, software organizations can better maximize the return on their software investment.

The alignment of technical decisions with business realities is also at the core of the difference between software engineering and computer science. Science is about expanding knowledge, and engineering is about applying that knowledge to build, operate, and maintain efficient and economical systems. Making technical decisions that align with the business realities is at the heart of software engineering.

This book is about getting the most out of your software investment. It's about helping you, the practicing software professional (or, the software professional-in-training), make purposeful, appropriate, business-conscious technical decisions so that you can get the most from the limited resources you do have. After learning the concepts and techniques in this book, if your boss were to ask you, "Is this the best way for us to be spending our limited time and money?" you could answer that question in a way that gives them confidence that you really know what you are talking about.

The next chapter explains why businesses exist and how they "work" in a financial sense.

Self-Study Questions

1. A software project has been estimated to cost $850,000 and take 10 months. Given the project outcomes from the Standish Group report mentioned at the beginning of the chapter, if the project completes at all what will its cost and schedule more likely turn out to be?
2. Name at least one software-intensive company that was in business in the year 2000 that isn't in business today. When did they go out of business? Why did they go out of business?
3. Name at least one software-intensive company that was in business in the year 2000 that is much smaller today (fewer employees, smaller market share, etc.) than they were then. What happened to the company between then and now? Why are they so much smaller now than before?
4. Can you describe a software project that was a net money loser for an organization? Who was that organization? What was the project? When did it happen? How much money do you think was spent on the project? How much, if any, do you think the software project recovered? Justify your answers.