PRENTICE HALL PTR

AGILE MANAGEMENT FOR SOFTWARE ENGINEERING

Applying the Theory of Constraints for Business Results

DAVID J. ANDERSON

Foreword by Eli Schragenheim

THE COAD SERIES

Library of Congress Cataloging-in-Publication Data

Anderson, David J. (David James)
Agile management for software engineering / David Anderson
p. cm
Includes index
ISBN 0-13-142460-2.
1. Software engineering. 2. Computer software--Development--Management. I. Title

QA76.75 .A48 2003
005.1--dc22

2003017798

Editorial/production supervision: *Carlisle Publishers Services*
Cover art: *Jan Voss*
Cover design director: *Jerry Votta*
Art director: *Gail Cocker-Bogusz*
Interior design: *Meg Van Arsdale*
Manufacturing manager: *Alexis R. Heydt-Long*
Manufacturing buyer: *Maura Zaldivar*
Executive editor: *Paul Petralia*
Editorial assistant: *Michelle Vincenti*
Marketing manager: *Chris Guzikowski*
Full-service production manager: *Anne R. Garcia*

Publishing as Prentice Hall Professional Technical Reference
Upper Saddle River, New Jersey 07458

Prentice Hall PTR offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales. For more information, please contact: U.S. Corporate and Government Sales, 1-800-382-3419, corpsales@pearsontechgroup.com. For sales outside of the U.S., please contact: International Sales, 1-317-581-3793, international@pearsontechgroup.com.

ISBN 0-13-142460-2

Text printed in the United States on recycled paper at Hamilton in Castleton, New York.

10th Printing September 2009

Pearson Education LTD.
Pearson Education Australia PTY, Limited
Pearson Education Singapore, Pte. Ltd.
Pearson Education North Asia Ltd.
Pearson Education Canada, Ltd.
Pearson Educación de Mexico, S.A. de C.V.
Pearson Education–Japan
Pearson Education Malaysia, Pte. Ltd.

Foreword

It is so good to finally have a book targeted at the software industry that challenges some of the basic business assumptions behind software engineering, and particularly those behind managing software organizations. At the time these words are written, the software business is facing huge difficulties worldwide. I hope that these difficulties also generate a willingness to look afresh at the business and to have the courage to contemplate changes. Other industries, particularly in manufacturing, went through such conceptual changes in their business processes during the 1980s and the 1990s. It is certainly not easy, but as I've personally experienced, it is highly desirable.

In 1985 I managed my own software company in Israel and was quite proud with my new package for certified public accountants. But, even though my package competed very nicely in the market, I noticed an on-going business problem: More and more development was needed to keep the package alive. In such a case, how do I justify the on-going investment? Eventually, I was not sure that a small software company, focused on a specific market niche, could be a good business—even when the product itself was enthusiastically accepted by the market. I felt that even though I already had my MBA, I needed a better perspective to understand the business.

Then I met Dr. Eli Goldratt.

I had heard a lot about Dr. Goldratt's international software company, Creative Output, Inc., which was seen as much more than just an excellent and innovative software company. It challenged some of the most sacred norms of business, such as the concept of product cost. I could not understand how anyone could challenge the basic concept that a unit of a product has a certain cost associated with it. I was intrigued enough to be open to an offer: Join Creative Output in order to develop a video game for managers that would deliver some new managerial ideas. At the time, computerized games were focused on fast fingers and perfect coordination. They were certainly not something of interest to adults. How could a computer game be readily accepted by grown-up managers and deliver new managerial ideas?

This was the start of a mental voyage into a new management philosophy that does not lose its grip on reality. I turned myself into a management consultant with a focus on improving whatever is the particular goal of the organization. Software became an important supporting tool, but not the focus of the change efforts.

The relevance of the Theory of Constraints (TOC) to the software industry is twofold:

1. Vastly improving the flow of new products to the market.
2. Determining the real value of a proposed project, or even just a feature, to the final user. The underlying assumption is that if we know the real value to the user, it is possible to develop the right marketing and sales approach to materialize the value to the user and to the software organization.

David Anderson focuses mainly on the first aspect in this book, which includes looking at the business case and ensuring the ability to make it happen. Software organizations can definitely be improved with the help of the new generic managerial insights that have already changed traditional western industries. David does a great job in bringing together the generic managerial ideas and rationale and combining them with software-focused approaches to come up with a coherent approach on how to improve the business.

Read this book carefully with the following objective: **Learn how to make more with less.** Don't accept every claim David raises just because he says it is so. If you truly want to make more with less, you need to be able to internalize the claim. All I ask is you give it a chance. Dedicate time in order to rethink your environment, and then see for yourself what to do. Overcoming inertia is the biggest challenge of any really good manager in any organization. Of course, rushing to implement new fads can be even worse. Keeping an open mind and confronting new ideas that invalidate basic assumptions are what I suggest you strive for. This book is for you to struggle with. It is not trivial, and it is not a fad. If you like what you do now, it should be your responsibility to check out new ideas that might yield huge improvements.

Here are some brief insights regarding the assessment of the value to a potential customer of a new feature, particularly to a new software package.

A new Feature can bring value to the user only if it eliminates, or vastly reduces, an existing limitation. The amount of the value depends on the limitation removed—not on the sophistication of the feature itself. Let us take a simple example. At a certain time in the history of word processors, somebody had an idea: Why not add a spell checker to the package?

What is the value of the spell check we now have as a routine feature? What limitation does it eliminate or reduce? For people with a very good knowledge of the language, spelling mistakes are caused by writing too fast. So, without a spell checker, those people need to read carefully what they just wrote. People who are not in full command of the language (for example, me, as an Israeli) need to look at the dictionary very often, which is quite time consuming.

This need leads us to recognize two additional insights.

People developed some rules to help them overcome the limitation. People who used word processors had to go over whatever they just wrote before sending the document to others. People without good command of the language needed to be supported by a dictionary.

Once the limitation is vastly reduced, people should replace the old rules with new ones that take full advantage of the removal of the limitation. If this does not happen, then there is no added value to the Feature.

Now we can see whether adding a spell checker to an existing word processor brings value. Suppose you have perfect command in the language, would you now refrain from carefully reading your recent document before sending it away? Spelling mistakes are hardly the main reason to go over any document that I want other people to read. So, for people with perfect knowledge, the spell checker offers no real value. But, for me as a person in good command of Hebrew, but not good enough in English, spelling mistakes in English are a nuisance. But, could I really avoid them just by the use of a spell checker? As long as the spell checker does not suggest how to write the word correctly—the limitation is only marginally reduced and thus not much value is produced. This means that if we want to generate significant value for the specific user group that has not mastered the language, we need to add good suggestions of what should be written instead.

In this simplified example, we already see the need to check the behavior rules both before the limitation is eliminated and after. Is it always clear what the new behavior rules should be? Assuming the user is well aware of what the new rules should be is a very common trap for too many software features.

Suppose that a new Feature is added to a sales-graph display module in which the trends shown by the graph are analyzed for statistical significance. The limitation is lack of knowledge on whether market demand is really up or down or just part of the normal statistical fluctuations. The current behavior of the management is: If sales are up, the sales agents are complimented and get appropriate bonus; if sales are down, there are no bonuses and some hard talk from management.

What should the new management rules be once management knows whether the rise in sales is significant? I'm afraid that in the vast majority of the cases the behavior will be exactly the same. Hence, the newly added Feature will not add value to the customer, even though some customers might ask for it and even exert a lot of pressure to have the Feature developed. Eventually, the value to the software company of developing the Feature will be negative.

Of course, for a good managerial consultant assisting in the formation of better decision processes, a specific Feature can bring immense value both to the consultant and the client. In this case, a strategic partnership between the consultant and the software company can be a win-win for all, including the client.

Improving the flow of the Features that truly bring value to the customer and also have a good chance of generating revenues for the software organization is what this unique book is all about. The Agile Manifesto principle of "Our highest priority is to satisfy the customer through early and continuous delivery of valuable software" is fully in line with the Theory of Constraint's objectives. To be more precise, TOC strives to generate more of the organization's goal. But, in order to do so, the organization has to generate more value

to its customers. The means of early and continuous delivery of software that truly generates value should assist both the software organization and its clients in achieving more of their respective goals.

Please bear in mind that improvement has only one criterion: Achieving more of the goal. The path to truly improving the performance of your software organization may well start with this book.

Eli Schragenheim

Introduction

"Poor management can increase software costs more rapidly than any other factor."

Barry Boehm[i]

Why Agile Management?

As Barry Boehm points out, bad management costs money [1981]. These days it also costs jobs. Senior executives, perplexed by the spiraling costs of software development and depressed by poor results, poor quality, poor service, and lack of transparency are simply shrugging their shoulders and saying, "if the only way this can be done is badly, then let me do it badly at a fraction of the cost." The result is a switch to offshore development and layoffs. With the state of the economy in 2003, what was a trickle has become a positive trend. If the trend isn't to become a flood, management in the information technology business needs to get better. Software development has to cost less and produce better results—more reliably, with better customer service, and more transparency. This book will teach ***the Agile manager*** how to achieve that.

The Economic Imperative

Building software costs a lot of money because it is labor intensive knowledge work.

Software engineers and their colleagues in project management and related functions are very well paid. It's a basic supply versus demand problem. Throughout most of my life, demand for IT workers has exceeded supply. The rates of pay have risen accordingly. Most software engineers under the age of 40 earn more than their peers who entered traditional professions such as medicine, law or accountancy. In many American firms, software engineering pays better than marketing.

Recently, with the global economic downturn, large corporations and many smaller ones are focused on trimming costs to improve profits or reduce losses. The high dollar line item for IT is under pressure. CIOs are having their budgets cut. The result is that jobs are moving offshore to outsource firms in Asia, Australia, and Eastern Europe. Knowledge work is moving out of rich countries and into poorer countries. Typically, an Indian outsource supplier can offer a labor rate of 25% of the rate for an equivalent U.S. software developer.

[i] [Boehm 1981] Software Engineering Economics

If software knowledge work is to remain in the rich, developed countries of the world and software engineers in America, Europe, and Japan are to maintain the high standard of living to which they have become accustomed, they must improve their competitiveness. There is a global market for software development, and the rise of communications systems such as the Internet, have made it all too easy to shrink the time and distance between a customer in North America and a vendor in India or China.

Jobs are at stake! Just as western manufacturing was threatened by the rise of Asia in the latter half of the 20th century, so too is the knowledge worker industry threatened by the rise of a well-educated, eager workforce who can do the same work for between one tenth and one quarter of the cost.

The answer isn't that software developers must work harder if they want to keep their jobs. Software engineers aren't the problem. The answer is that management techniques must improve and working practices must change in order to deliver more value, more often, in order to improve competitiveness.

The Thesis for Agile Management

This is a book about software engineering management. It is also a book about business. It is a book about managing software engineering for the purpose of being successful at business. It will offer proof that Agile software development methods are better for business.

The information technology industry hasn't been good at managing software engineering and hasn't shown an aptitude for management and process control. As a result, information technology businesses are often run by seat-of-the-pants intuition and rough approximations. It is common, to the point of being accepted as industry standard practice, for information technology projects rarely to follow the plan, to be late and over budget and fail to deliver what was promised.

Software engineering management is traditionally a poorly practiced profession. This may be because it is poorly (or rarely) taught. Only recently has my local college, the University of Washington, begun offering an MBA program in high technology management. Such programs are rare. As a result, there is little management expertise in the industry.

However, many techniques do exist that can improve the competitiveness of software development businesses. These techniques have been proven in other industries. The challenge has been figuring out how to apply them to software development. Techniques such as the Theory of Constraints [Goldratt 1990a], Lean Production [Womack 1991], Systems Thinking [Senge 1990], and new ideas evolving out of the recent science of Complex Adaptive Systems are providing insights that unleash the latent ability of knowledge worker talent.

The secret to economically viable software engineering is new working practices based on new management science. The Agile manager must construct an Agile learning organization of empowered knowledge workers. When this is achieved the results will be dramatic. Improvements of 4 times are easily achieved. 10 times is definitely possible. Imagine if your software engineering organization could do 5 times as much work in half the time it currently takes. What would it mean for you, your job, and your organization?

Accept Uncertainty, Manage with Transparency

Knowledge work isn't like manufacturing. Stamping out car bodies can be performed with a high degree of certainty. It is dependable to within a very low tolerance. Failures and errors are rare. The time to stamp two car bodies is almost precisely twice the time to stamp a single car body. The time to stamp 100 bodies is probably precisely derived from the time to stamp a single car body multiplied by 100. Manufacturing is in many ways predictable, linear, and, in the case of chemical processes, defined by scientific rules.

Knowledge work is neither linear nor defined. So it isn't like manufacturing. The assumption has been that because it isn't like manufacturing and isn't predictable and linear, it just can't be managed the same way. In fact, attempts to bring traditional management to software engineering processes have tended to fail. Software projects rarely if ever run to plan, and estimating is generally a black art with the resultant estimates often a complete fiction. Software development, from the perspective of the boardroom, has been out of control.

This book will show that dismissing software engineering as an uncontrollable process is wrong. It can be managed like other parts of a business. The secret is to manage the right things and to do so with transparency. Just because software engineering has greater uncertainty associated with it than manufacturing production does not mean that management methods are invalid. It simply means that those methods must accommodate greater uncertainty. The Theory of Constraints teaches managers how to buffer for uncertainty and this book will explain how to apply that technique to software development. It is important that value chain partners, management, and shareholders understand the correct model for managing software development, a model that accommodates uncertainty, and learn to trust the techniques of the Agile manager.

The Agile manager's new work becomes a study in setting the governing rules for controlling the system of software production. The Agile manager needs to learn what to track, how to track it, how to interpret the results, and what to report to senior management. This book explains what, why, and how to do this.

A Trend of Frustration

Some high technology workers on the west coast of the United States are giving up the profession and changing careers. All around the world, high tech workers are disillusioned. They are beginning to realize that a job in high technology is not worth sacrificing family life, social life, or health. They are realizing that their hourly rate doesn't look so good, considering all the unpaid overtime they are expected to work. They are realizing that there must be more to life.

One former colleague, from my time in Singapore, recently trained as an artist and photographer. Another, with whom I worked in Kansas City, quit the business and moved to Paris, France, where he works in the non-profit sector. Another colleague recently resigned in order to start an auto-tuning business. Yet other colleagues, who work as contractors, are only prepared to work part time. One prefers to work in a shoe store, and another does flower arranging. I hear similar anecdotes from people I meet all over the industry. What is happening?

IT workers turn up for work for four reasons: the cause (the vision and leadership of the organization), the love of technology (usually a specific choice in which an almost religious fervor is aroused), the money (and it is usually pretty good), and the boss (people really do work for people). Let's consider these in turn.

The cause and the technology can often be grouped together. They include the mission of the business, the vision of the future, the technology being used, and the industry into which all of this is being deployed. There are IT workers who will simply never work in the defense business, for example. Creating a great cause that will draw people to it is a matter for great leadership. There has been much written about leadership in recent years. Perhaps there is a yet-to-be-written great book about IT industry leadership but teaching leadership is not within the scope of this book.

The money is important. IT workers are in demand. Demand exceeds supply. Even in hard times, demand for IT workers remains strong. Often a recession strengthens demand because automated systems can replace other workers and reduce cost. Consider the recent trend in automated machines for airline check-in, for example.

The boss is very much the scope of this book. If the boss doesn't get it, the staff will get disillusioned and leave. High staff turnover in IT businesses is usually an indication that the management "doesn't get it." Management is important. People like to work in well managed, properly organized environments. They like to have clear objectives and an environment in which to do great work.

This book will give IT industry bosses a new set of tools for managing. It will show them how to assess the IT parts of their businesses, as they would any other part of the business. It will show how to demonstrate whether or not IT delivers true value-add and produces a suitable return on investment.

Running software engineering as a proper business actually produces effects that result in more optimal use of resources, more efficient production of code, and a better creative and professional environment for the staff. When the boss really "gets it," the staff knows it and like it. The key to low staff turnover and high performance from a software development organization is better management.

The Agile Manifesto

Recently, there has been a rebellion in the industry against the growing tide of poor performances, long lead times, poor quality, disappointed customers, and frustrated developers. It is a rebellion against poor management. A passionate body of software developers has declared that there must be a better way—delivering software should be more predictable. These passionate people espouse a number of new software development methods, which they claim will enable faster, cheaper, better software development with on-time, on-budget delivery of the agreed scope. These new methods are known collectively as Agile methods.

The word "agile" implies that something is flexible and responsive and in a Darwinian sense has an innate ability to cope with change. An agile species is said to be "genetically fit." By implication, Agile software development methods should be able to survive in an atmosphere of constant change and emerge with success.

The accepted definition of Agile methods was outlined in February 2001 at a summit meeting of software process methodologists which resulted in the Manifesto for Agile Software Development.[ii] It was created by a group of 17 professionals who were noted for what, at the time, were referred to as "lightweight" methods. Lightweight methods started with Rapid Application Development (RAD). The RAD approach sought to time-box software releases to strict delivery dates, subordinating everything else in the project, including budget, scope, and staffing levels to achieve the delivery date. The term "rapid" came from the suggested nature of the time-boxes—much more frequent than traditional software development, that is, 2 weeks to 3 months.

Agile methods are mostly derived from the lightweight approach of RAD. They add extra dimensions, primarily the recognition that software development is a human activity and must be managed as such.

Manifesto for Agile Software Development

We are uncovering better ways of developing
software by doing it and helping others do it.
Through this work we have come to value:

Individuals and interactions *over processes and tools*
Working software *over comprehensive documentation*
Customer collaboration *over contract negotiation*
Responding to change *over following a plan*

That is, while there is value in the items on
the right, we value the items on the left more.

Kent Beck, James Grenning, Robert C. Martin, Mike Beedle, Jim Highsmith, Steve Mellor, Arie Van Bennekum, Andrew Hunt, Ken Schwaber, Alistair Cockburn, Ron Jeffries, Jeff Sutherland, Ward Cunningham, John Kern, Dave Thomas, Martin Fowler and Brian Marick

The Agile Manifesto, as it has become known, is a very simple and concise declaration that seeks to turn the traditional view of software development on its head. The manifesto is based on 12 principles[iii]:

> *Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.*

[ii] http://www.agilemanifesto.org/.

[iii] http://www.agilemanifesto.org/principles.html. Kent Beck, James Grenning, Robert C. Martin, Mike Beedle, Jim Highsmith, Steve Mellor, Arie Van Bennekum, Andrew Hunt, Ken Schwaber, Alistair Cockburn, Ron Jeffries, Jeff Sutherland, Ward Cunningham, John Kern, Dave Thomas, Martin Fowler, and Brian Marick.

Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.

Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.

Business people and developers must work together daily throughout the project.

Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.

The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.

Working software is the primary measure of progress.

Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.

Continuous attention to technical excellence and good design enhances agility.

Simplicity—the art of maximizing the amount of work not done—is essential.

The best architectures, requirements, and designs emerge from self-organizing teams.

At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

There are a number of Agile methods. In Section 2, this book looks closely at three of them—Extreme Programming (XP), Feature Driven Development (FDD), and Scrum. Though other Agile methods are not explored, the book will provide the basic guidelines and metrics for making an appropriate assessment of each in comparison to more traditional software development methods.

The Problem with Agile Methods

Agile methods propose some unusual working practices. Extreme Programming, as its name suggests, has some of the more radical. They often go by names that sound as if they belong in the skateboard park or amongst the off-piste snowboarding community. The strange language and the strange practices scare management in large companies. Are they ready to stake their careers, reputations and fat bonuses on pair programming and stacks of filing cards?

Agile methods introduce scary counter intuitive working practices. If managerial fears are to be overcome, it is necessary to provide management methods that allay those fears. This requires methods that report believable and familiar statistics and have meaning to the business. It is necessary to demonstrate the economic advantages and focus on real business benefits. Software development is about making more profit, not about making great code. By leading with the financial arguments, senior managers in large companies can gain confidence that the expensive knowledge workers understand the true goal. This book will show how to mature a software engineering organization to the point where it can report believable financial metrics. It will also show what those metrics should be and how to calculate them.

Agile Methods—A Fad or a Trend?

Agile methods promise a lot, but where is the proof? Agile methodologists will reply that, "the proof is in the pudding." In other words, give it a try and find out for yourself. These claims have been heard before. Who can recall 4GL (so-called "fourth generation languages") that promised to eliminate developers and allow ordinary workers to create labor saving tools for themselves? Or perhaps you were sucked into the world of visual software assembly from components? Did the arrival of Visual Basic really eliminate developers? The IT world has been full of promises. Why should Agile methods be any different?

Are Agile methods a genuine trend in changing working practices or are they just another fad that lets software people "goof off" at work? This book will show that Agile methods echo the techniques of Lean Production and the Theory of Constraints which revolutionized manufacturing industry.

Agile software development is really about a change in working practices and a change in management style. Agile methods understand what management truly is. They understand that management is more than economics and engineering, that it is very much about people. "Rightly understood, management is a liberal art, drawing freely from the disciplines that help us make sense of ourselves and our world. That's ultimately why it is worth doing" [Magretta 2002, p.3]. Because of this basis in existing experience, I firmly believe that the Agile approach is a genuine trend, a change in working practices and paradigm shift in how software is produced. It is not a fad.

The Business Benefit

In order to adopt Agile methods in a large corporation, it is not enough to go before the board and say, "Gee, people say this works. Why don't we give it a try?" The CIO is likely to be a pragmatist, not prone to early adoption or risk taking. It will be necessary to argue a business case based on hard numbers indicating better profitability and higher return on investment. Doing so is the only way to make Agile methods look attractive and to fight against the short-term thinking that is driving decisions to outsource software engineering offshore.

This book will arm the Agile manager with the material to make a business case for agility. Agile methods can be justified on improved value-added and ROI. This book will teach the Agile manager to manage up and lead with a financial argument.

A framework for scientifically measuring and assessing Agile methods is presented. The metrics involved are used to determine the level of added value and the received return on investment. Much of the work that made this possible was developed by Eli Goldratt and his colleagues. It is a body of knowledge known as Throughput Accounting [Corbett 1997]. Throughput Accounting, based on the applications of the Theory of Constraints to manufacturing production, is used as the basis for the financial arguments presented.

Toward a Software Economic Miracle

While the West during the 1970s and 1980s was focused on increased automation through the use of robots on the assembly line, the Japanese produced far better results through management techniques that changed working practices. These working practices originated at Toyota and are known as the Toyota Production System or Kanban Approach.

The technology industry has for the last 30 years, like western manufacturing, also been focused on technology solutions. There have been third and fourth generation languages, modeling and abstraction tools, automated integrated development environments, and automated testing tools. To some extent, the Agile community rejects this ever increasing technology approach and instead embraces new management techniques and changes to working practices. In this respect, Agile methods resemble the principles first advocated by Toyota and now known in the West as Lean Production.

The techniques of Lean Production created an economic improvement of twenty to fifty fold during the second half of the 20th century. For example, Womack and colleagues [Womack 1991] report that in one recent year Toyota built half as many cars as General Motors using less than 5% of the people. In other words, Lean Production at Toyota had produced a ten fold improvement over its American mass production competitor. Some Agilists are reporting four fold economic improvements [Highsmith 2002; Schwaber 2002]. This is equivalent to the improvements made in automobile manufacturing in Japan in the earlier part of that half century—for example, those at Mazda between the 1960s and 1980 when productivity was improved by four fold. During the most recent twenty years, some of these manufacturers have gone on to make improvements of five times or greater. This produced a cumulative economic improvement of twenty times or more. It is precisely these types of gains that created the Asian economic miracle of the latter 20th century and provided vast wealth across the globe.

If Agile software development can provide a four fold improvement within 9 months, why would a company outsource to an Indian supplier that promises a four fold cost reduction over 3 to 4 years?

The software industry now employs over 30 million people worldwide[iv] and can list the world's richest company, Microsoft, amongst its number. What if it were possible to create another economic miracle? What if software development resembled the manufacturing efficiency of 1925? It is just possible that Agile methods represent the beginning of an understanding of how to build software better, faster, and cheaper. Is it just possible that there is a latent economic improvement in the order of 95% waiting to be unleashed? Agile methods are a step down the road to a leaner knowledge worker industry. They really do produce financial benefits, and this book will demonstrate how to calculate them.

[iv] Estimates taken from figures by Gartner Group and IBC suggest that there are around 15 million software developers. It is reasonable to assume that those employed in other related functions, such as project, program, and product management, will account for 15 million more.

Who Should Read This Book

"Most management books are only for managers. This one is for everyone—for the simple reason that, today, all of us live in a world of management's making. Whether we realize it or not, every one of us stakes our well-being on the performance of management,"[v] said Joan Magretta introducing her book "What Management Is" [2002, p.2]. As Tom DeMarco observed in his book "Slack,"[vi] the Dilberts[vii] of the world have abdicated responsibility. It suits them to blame the manager. Dilbert fails to see it as his duty to help his manager be more effective. Management is a task that concerns everyone involved in a business from the stockholders to the most junior of employees. Hence, this book is intended for a wide audience—an audience of anyone who cares whether or not a software business is well run.

The text is intended for anyone who is interested in changing the working practices of software development to make them more effective and more competitive. The book is primarily aimed at all levels of management in all software-related disciplines and those who aspire to senior individual contributor or line manager positions in the foreseeable future. It should also appeal to Masters degree and MBA students looking for a management career in a software-related industry. Every CEO, CFO, COO, and CIO who runs a business with significant expenditures on software development activity needs to understand the new paradigm and theory presented in Section 1. Lou Gerstner, writing in his IBM memoir pointed out that cultural change must be led from the top if it is to be effective [Gerstner 2002]. If change is to be led from the top, the boss must adopt the correct mental model of Agile development practices in order to frame decisions and understand the counterintuitive activity happening beneath.

A Thesis for New Management Practices

This book defines 4 basic management roles and describes a set of practices for each role. Those roles are development manager, program manager, project manager, and product manager. Each is described in Chapter 8, "The Agile Manager's New Work." Specific details for the development manager's role are defined in Chapters 5 and 9. The program manager's role is defined in Chapter 10. The project manager's role is defined in Chapter 7. The product manager's role is defined in Chapter 16.

The thesis of the book is that the development manager is responsible for running an on-going system of software production. This must be managed with metrics based on fine grained units of production activity. However, programs and projects must be measured at a coarse grained level that reduces the uncertainty through aggregation of fine grained tasks. How to buffer against variability is explained in Chapter 4. The product manager must define the groupings of fine grained functionality which have meaning as valuable deliverables, that is, the coarse grained items to be tracked by the program and project manager.

Together, all 4 roles interact to define a 2-tiered management system that sets the governing rules for the system, but allows highly delegated,

[v] [Magretta 2002] What Management Is, page 2.

[vi] [Demarco 2001] Slack

[vii] Dilbert is a registered trademark of United Feature Syndicate. Dilbert, a cartoon character created by Scott Adams, suffers under a pointy haired boss who just doesn't get it!

self-organization within. Successful Agile management requires a highly delegated system of empowered knowledge workers. The essence of Agile management is self-organizing production, framed within the planned assembly of valuable components, and delivered frequently to generate a the required ROI for the business.

An Agile Maturity Model

Chapter 11 introduces the notion that Agile methods can mature in an organization as it learns to use them better. This book leads with the financial metrics. It focuses on the true goals of a business and then examines how management must organize and report the day-to-day workings of the software production system in order to deliver the desired financial results. This approach has been taken to demonstrate the compelling reason for switching to Agile software development.

In practice, the approach to delivering a failure-tolerant, agile, learning organization will happen inside-out. The working practices will come first, then the traceability, then the metrics, then learning, and eventually the financial metrics and results. The Agile Maturity Model describes this progression.

How to Read This Book

Section 1 is intended as general reading for anyone interested in running a software development business for better results. It is suitable reading for all levels of management from team lead developers to CIOs, CEOs, CFOs, and GMs. Section 1 explains Agile management, its practices and theory. It explains how to apply the Theory of Constraints and Lean Production methods to software engineering as a general practice. For many readers Section 1 will be sufficient.

Section 2 is intended for readers who need to manage the change to Agile software development in their organization and for those who need to understand why they are making a change and how to implement what they are changing to. Chapters 19 and 20 give an outline of traditional software development methods and will help an agile manager explain the current reality and create a baseline from which to measure improvement. The remainder of Section 2 surveys a subset of Agile software development methods. This survey is not meant to be exhaustive. It shows, by example, how to relate specific Agile methods to the theory presented in Section 1. Chapters 21 through 30 lay out possible future realities for an Agile software development organization and demonstrate how to measure them to show an economic improvement. FDD, XP, Scrum, and RAD are compared against the theory from Section 1. The emphasis is on explaining these methods rather than comparing them against each other. Relative comparisons are left for Section 3.

Section 3 is for those who need to choose one method over another and those who seek to understand Agile methods and develop the future of Agile software development management. It seeks to understand the similarities and differences and the varying foci of currently available Agile methods. The applicability of these methods is considered against their appropriateness for different types, sizes, and scales of software projects. Section 3 is intended primarily for Agile methodologists and those who wish to further the debate about the future of Agile software development.

chapter 3

TOC in Software Production

TOC's Five Basic Steps

The Theory of Constraints can be explained with a simple five-step process that needs little explanation.

1. Identify the System Constraint.
2. Decide how best to exploit the System Constraint.
3. Subordinate everything else to the decision in step 2.
4. Elevate the Constraint.
5. If steps 1 through 4 have created a new constraint, return to step 2.

TOC is founded on the notion that a value chain is only as strong as its weakest link. The conjecture is that there is only one weakest link at any given time. This weakest link is known as the constraint. In a process or system that takes input and produces output under some control mechanism, the constraint is described as the capacity constrained resource (CCR).[1] In other words, the value chain is a chain of processes or systems that add value to a raw material and turn it into a finished product. The rate of production of the finished product is constrained by the rate of production of the slowest (or weakest) element in the value chain.

Identify and Exploit Constraints

If TOC is to be used to improve a system, constraints must be identified, one by one. Physical constraints are also known as bottlenecks. Hence, identifying bottlenecks is a good place to start looking for the current constraint. It is generally assumed that there is only one global system constraint at any given time. There may be several bottlenecks, but only one will be constraining the overall Throughput.

Figure 3–1 shows a software production system that is constrained by the capacity of System Test, which is only capable of processing 30 units of production per month. The capacity of Acceptance Test at 80 units per month is irrelevant. At most, only 30 units will be passed to Acceptance Test every month.

Once the constraint has been identified, a decision must be made on how to minimize its constraining ability on the system. The utilization or

[1]The term CCR is an abstraction. Constraints can take many forms.

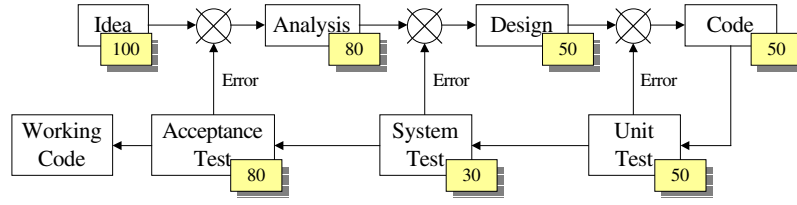

Figure 3–1
Software production system showing rates of production.

capacity of the constraint must be maximized. The CCR must be fully utilized. It must never be idle. Every unit of production (Q) lost on the constraint is a unit of Q lost to the whole system. The constraint can be protected from idleness by providing a buffer (or queue) of work for it to perform. As a generalization, constraints are protected by buffers—a queue is just a type of buffer, a physical buffer of inventory. Protecting a constraint is a necessary part of exploiting a constraint to the full. Achieving maximum exploitation of a resource means that whatever can be done should be done to utilize the CCR optimally. This is best explained with a couple of examples.

Consider a manufacturing machine that cuts silicon wafers into individual chips. Assume that this is the CCR for a chip fabrication plant. How can the wafer cutter be protected and exploited as a CCR?

It can be protected from starvation by provision of a buffer of completed wafers. It can be protected from power outage with a provision of uninterruptible power and a backup generator. It can be exploited fully by running three shifts and utilizing the machine up to 24 hours per day. It can be exploited by performing a prior quality control check on wafers to insure that only good quality wafers are processed through the cutting machine.

Exploiting the Software Developer as a Capacity Constrained Resource

Consider a software development constraint. A software developer is paid to work 8 hours per day. Strictly speaking, 8 hours per day is the constraint. Of course, software developers tend to be flexible in their working hours, so it might be more accurate to state that the constraint is the period of time during which the developer shows up in the office. How can the software developer as a resource be protected and exploited?

She can be protected from idleness by always having a pool of development tasks ready to be done. She can be protected from interruptions by providing a communication structure that minimizes the lines of communication.[2] She can be protected from distraction by providing a quiet envi-

[2]Harlan Mills wrote about the "Surgical Team" in 1971 as an example of a structure designed to minimize lines of communication and maximize the Throughput of software developers.

ronment for working. She can be further exploited by providing her with the best software development tools available. She can be exploited by providing support staff for nonproductive activities, such as progress reporting and time-tracking, or tools to automate such nonvalue-added work. Nonvalue-added work is waste. She can be exploited by providing adequate training in the technologies being used. She can be exploited by providing a team of colleagues to support her and help resolve difficulties. She can be exploited by ensuring that the requirements she is given are of good quality. This represents just a short list of possible protection and exploitation mechanisms to maximize the completed working code produced by a software developer.

Subordinating to the Exploitation of a Constraint

Step 3 in TOC requires subordination of all other things to a decision to protect or exploit a constraint. Step 3 has profound implications for any business. The effect of step 3 can produce results that are counterintuitive and go against the existing management policies.

To continue with the fabrication plant example, assume that the capacity of the wafer cutting tool has been determined. It has also been determined that the Throughput of the whole plant is constrained by the capacity of the wafer cutter. A decision is made to protect and exploit the wafer cutter to its maximum capacity. In order to subordinate all else to this decision, there must be agreement to regulate the flow of inventory from the factory gate to the wafer cutter at the same speed as the wafer cutter can process it. This is the Drum-Buffer-Rope application of TOC. The rate of the cutter is the drum. The inventory from the factory gate to the cutter is the rope and a buffer in front of the cutter to prevent it becoming idle is the buffer. A similar process in Lean Production (or TPS) is known as "balancing" and results in the determination of Takt time. Takt time plays the same role as the drum in Drum-Buffer-Rope. What this can mean in practice is that other machines earlier in the process may lay idle. Part of the manufacturing plant may be idle because it would produce too much inventory were it to run 24 hours per day. Other parts of the system must only produce as much as can be consumed by the wafer cutting machine.

The psychological effect of this subordination approach when first introduced can be overpowering. If the business is run using traditional cost accounting methods, then the idle machines appear to be very inefficient because efficiency is measured locally as the number of units processed per day/hour/minute. However, if the machines are not permitted to be idle, if there is no subordination of the rest of the plant to the decision to feed inventory at the speed the wafer cutter can cope with it, then inventory will be stockpiled in front of the wafer cutter. The result will be that total inventory will grow and so will investment. Consequently, operating expense will grow, too. The business will become less profitable and return less on the invested capital. Leaving machines idle can be good for business, but it is counterintuitive.

Perishable Requirements

Producing and holding too much inventory is much worse in software because of the perishable nature of the inventory—requirements can go stale because of changes in the market or the fickle nature of the customer. There is a time value to requirements, and they depreciate with time, just like fresh produce, that is, requirements have a time to market value. As time goes by, the potential Throughput from the transformation of the requirement into working code decreases. Increasingly often requirements become obsolete, and they are replaced by change requests for new requirements with a current market value.

Staleness is a very profound problem in software development. Although a requirement (unit of V) may be written down, there is an implicit body of knowledge on how to interpret it. If it isn't being actively processed, that knowledge atrophies—people forget things! Even worse, people leave projects or companies and take the knowledge with them.

For proof of this, ask a developer to explain how some code written 12 months ago actually works, and see whether he can recall from memory or a brief analysis of the source code. Vital details never get captured, and people forget. Loss of memory and loss of detail incur extra costs. Such extra costs can be classified as waste.

Requirements that become stale are pure waste. Such requirements have a $0 potential Throughput value. When this happens, the cost of acquiring the requirement must be written off as operating expense.

Idleness Might Breed Contempt

It ought to be possible to measure the average Production Rate (R), of a developer for any given week, or month, or quarter. For a team of developers, it should be possible to guess approximately how many requirements can be processed for a given time period. If they are to be fully exploited, developers should not be loaded with any more than they can reasonably handle. So the rest of the system of software production must be subordinated to this notion. Requirements should be fed into development at the same pace as the developers can process them into completed code.

Again, the psychological effect of this decision when first introduced could be devastating. If the requirements are generated by analysts who interview subject matter experts and business owners, management must realize that those analysts no longer need to work flat out producing requirements. If development is the constraint, then by implication, analysis must not be the constraint. It could be that the analysts will spend time idle. They are not required to create requirements constantly. In TOC language, this stop-go effect is often called "Road Runner Behavior" after the Warner Bros. cartoon character who only has 2 speeds—full speed and stop. Road Runner behavior may be bad for morale amongst the analysts. Hence, the Agile manager must be aware that the subordination step in TOC is dangerous and needs careful management attention during introduction.

Probably the best way to deal with this is to ensure that everyone understands the Drum-Buffer-Rope principles and that they are aware of the current system constraint. If they know that they do not work in the CCR, they should be comfortable with their new role being only partially loaded. This technique of sharing an understanding of the system process and gaining buy-in to changes has been called "Fair Process" [Kim 1997].

Step 4 requires the elevation of the constraint. In plain English, this means that the constraint must be improved to the point where it is no longer the system constraint. This is best explained by example.

If the wafer cutting machine is the constraint in the fabrication plant, everything has been done to protect and exploit that machine, and everything else in the system has been subordinated to those decisions, but still there is insufficient Throughput from the plant, the constraint must be elevated. With a machine, elevation is simple—buy and commission another machine. With the introduction of the second machine, the wafer cutter may no longer be the constraint. The constraint within the system may have moved elsewhere. Management must now move to step 5 and identify the new constraint. If the constraint has moved outside the system, then (arguably) they are done.[3] For example, if the plant can now produce more than the company can sell, the constraint has moved to sales. Managers at the fabrication plant no longer control the constraint. Hence, there is nothing they can do to improve the Throughput of the business. If, however, the fabrication plant still has an internal constraint, then the managers must return to step 1 and start again.

Elevating the Software Developers Capacity Constraint

How is the developer elevated from a constraint? The most obvious method of elevation used in the software industry is unpaid overtime. The developer is asked to work longer hours. The constraint is stretched. Weekend work may also be requested. Again, the constraint is being stretched. The manager could also choose to hire more developers or to introduce shift working. Another very powerful method is to use a better developer.

It has been known for over 30 years [Weinberg 1971/1998] that some software developers produce much more output than others. Performance differences between the average and the best of 10 to 20 times have been documented [Sackman 1968]. The difference between poor and best is possibly fifty fold. Hence, one way to truly elevate a software development organization is to hire better people.

Good people are hard to find. In a large organization, management must accept that over the statistical sample, the engineering team will tend to be average. Hence, it is important that managers identify top performers, reward and keep them, and apply them judiciously to projects where development is the constraint. By using a top performer on a capacity constrained development project, management elevates the constraint and moves it elsewhere in the value chain.

[3]Lepore & Cohen merged the Theory of Constraints with Edwards Deming's Theory of Profound Knowledge and devised a 10-point scheme they call "The Decalogue" [Lepore, 1999]. Stage 9 involves "bringing the constraint inside."

Increasing Throughput by Elevating the Constraint

In Figure 3–1, the System Test process is a bottleneck. It is constraining the Throughput of the system to only 30 units per month. The equations in Chapter 1 demonstrated the best way to be more profitable is to increase Throughput. Figure 3–1 shows that it would be possible to raise the Throughput to 50 units per month if the System Test process was elevated. At 50 units per month, System Test ceases to be the constraint. At that point, Design, Coding, Unit Test, and System Test are all joint constraints. Raising production higher than 50 units per month would require investment to elevate all four of them. The business question to be answered is how much is it worth to increase the capacity of System Test from 30 to 50 units per month?

Focus of Investment

If TOC was to be summarized in a single word, it would be ***focus.*** TOC teaches managers where to focus investment. Whether it is investment of time, resources, or money, the largest ROI will be gained from investing in the currently identified system constraint.

The financial equations for cost justification are simple. The existing equation for ROI is:

$$ROI_{Pre\text{-}Investment} = \frac{\text{Throughput (T)} - \text{Operating Expense (OE)}}{\text{Investment (I)}}$$

If the current constraint is eliminated through an Investment (dI), then Throughput will increase by an amount (dT). The equation post-investment would look like

$$ROI_{Post\text{-}Investment} = \frac{(T + dT) - OE}{I + dI}$$

If $ROI_{Post\text{-}Investment}$ is greater than $ROI_{Pre\text{-}Investment}$ the investment to remove the constraint is worth doing. In fact, the officers of the company, being aware of the option to invest and elevate the constraint, are legally obliged to the shareholders to make the investment—assuming funds are available.

Is the 8-Hour Day the Best Choice of System Constraint?

It seems attractive to decide that there will be only one CCR in the system and that it will be the working day. To simply assume that 8 hours is a constraint and that all personnel are constrained by it seems to provide a convenient option for managers—they don't have to look for other constraints.

Goldratt argues that the 8-hour work day is not a useful constraint because it is not a bottleneck [1994, chs. 30 & 31]. Rather, he would call this an insufficiently buffered resource, that is, demand may outstrip supply. However, in the system of software production, a bottleneck would be a capacity constrained resource (CCR) in front of which a stockpile of inventory is apt to accumulate and beyond which resources may starve for input. This definition of a constraint satisfies the definition required for the Critical Chain [1997].

The capacity constrained resources (CCRs) that are most likely to represent the bottlenecks are not generalist developers and testers but the specialists such as UI designers, architects, data modelers, DBAs (performance

tuning wizards), visiting consultants, and maybe even subject matter experts (SMEs). Resources related to expert skills such as usability laboratories, staging environments for performance tuning, prototyping laboratories, and testing labs can also be bottlenecks. Such resources are likely to be shared in large organizations and require scheduling. Sharing a resource and scheduling a date for its use introduces uncertainty into project management.

Summary

Bottlenecks in software production are identified by measuring the trend in inventory at each step in the process. The trend in inventory is affected by the production rate (or capacity) of each process step.

The overall production of the system should be balanced against the capacity of the bottleneck. The bottleneck should be protected and exploited in order to maximize its Throughput.

Management may choose to invest in the bottleneck in order to increase its capacity and hence increase the overall production of code through the whole system. The cost of the investment can be considered against the value of the increased production that will be achieved. The use of TOC provides a focus for management, who may choose to employ Lean Thinking in order to elevate constraints and create improvement.

Index

D

E

F

G

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Y