Just Enough Requirements Management

Where Software Development Meets Marketing

JERMJERM

ALAN M. DAVIS

Just Enough Requirements Management

Also Available from Dorset House Publishing

Agile Software Development in the Large:
Diving Into the Deep
by Jutta Eckstein
ISBN: 0-932633-57-9 Copyright © 2004 248 pages, softcover

The Deadline: A Novel About Project Management
by Tom DeMarco
ISBN: 0-932633-39-0 Copyright © 1997 320 pages, softcover

Five Core Metrics: The Intelligence Behind Successful Software Management
by Lawrence H. Putnam and Ware Myers
ISBN: 0-932633-55-2 Copyright © 2003 328 pages, softcover

Hiring the Best Knowledge Workers, Techies & Nerds:
The Secrets & Science of Hiring Technical People
by Johanna Rothman foreword by Gerald M. Weinberg
ISBN: 0-932633-59-5 Copyright © 2004 352 pages, softcover

Peopleware: Productive Projects and Teams, 2nd ed.
by Tom DeMarco and Timothy Lister
ISBN: 0-932633-43-9 Copyright © 1999 264 pages, softcover

Project Retrospectives: A Handbook for Team Reviews
by Norman L. Kerth foreword by Gerald M. Weinberg
ISBN: 0-932633-44-7 Copyright © 2001 288 pages, softcover

Software Endgames:
Eliminating Defects, Controlling Change, and the Countdown to On-Time Delivery
by Robert Galen
ISBN: 0-932633-62-5 Copyright © 2005 328 pages, softcover

Waltzing with Bears: Managing Risk on Software Projects
by Tom DeMarco and Timothy Lister
ISBN: 0-932633-60-9 Copyright © 2003 208 pages, softcover

For More Information

- ✔ Contact us for prices, shipping options, availability, and more.
- ✔ Sign up for *DHQ: The Dorset House Quarterly* in print or PDF.
- ✔ Send e-mail to subscribe to *e-DHQ*, our e-mail newsletter.
- ✔ Visit Dorsethouse.com for excerpts, reviews, downloads, and more.

DORSET HOUSE PUBLISHING
An Independent Publisher of Books on
Systems and Software Development and Management. Since 1984.
353 West 12th Street New York, NY 10014 USA
1-800-DH-BOOKS 1-800-342-6657
212-620-4053 fax: 212-727-1044
info@dorsethouse.com www.dorsethouse.com

Just Enough Requirements Management

Where Software Development Meets Marketing

ALAN M. DAVIS

DORSET HOUSE PUBLISHING
353 WEST 12TH STREET
NEW YORK, NEW YORK 10014

Library of Congress Cataloging-in-Publication Data

Davis, Alan Mark.
Just enough requirements management : where software development meets marketing / Alan M. Davis.
p. cm.
Includes bibliographical references and index.
ISBN 0-932633-64-1
1. Computer software industry. 2. Computer software--Development. 3. Computer software--Marketing. I. Title.
HD9696.63.A2D38 2005
005'.068'5--dc22
2004028999

Cover Design: Nuno Andrade

Distributed in the English language in Singapore, the Philippines, and Southeast Asia by Alkem Company (S) Pte. Ltd., Singapore; in the English language in India, Bangladesh, Sri Lanka, Nepal, and Mauritius by Prism Books Pvt., Ltd., Bangalore, India; and in the English language in Japan by Toppan Co., Ltd., Tokyo, Japan.

Printed in the United States of America

Library of Congress Catalog Number: 2004028999

ISBN: 978-0-932633-64-4 12 11 10 9 8 7 6 5

Acknowledgments

I would like to thank Wendy Eakin of Dorset House, who quickly shared my vision to produce a "good enough requirements" book. Also deserving of thanks are the approximately 6,000 students to whom I have had the opportunity to teach requirements management over the years. Although I was officially the teacher, I feel I have learned so much from all of them and their experiences. My personal evolution as a requirements manager has progressed for the most part as the result of interacting with them.

I would also like to thank all of the colleagues I have worked with over the past 34 years, in both academe and industry. They too have been instrumental in shaping my thoughts and opinions.

I thank two universities for their contributions to this work: the University of Colorado at Colorado Springs, for granting me a sabbatical so I could devote time to writing, and the University of the Western Cape, South Africa, for providing me with a stimulating environment during my sabbatical.

But, above all, I want to thank my family for being with me always: my beloved wife of 27 years, Ginny; my wonderful children, Marsha and Michael; and my parents, Barney and Hannah.

This page intentionally left blank

Contents

This page intentionally left blank

Preface

When I first started studying requirements specifications and teaching classes on them in the late 1970's, I recognized that writing good requirements is very difficult. I worked hard to remove and help others to remove every trace of ambiguity from each and every requirement. I was convinced back then that a polished, word-processed requirements document was the only right way to record requirements. As I gained more and more experience, though, I started to realize that ambiguity can never be entirely removed from a requirements document that is written in natural language. So, I started to explore alternate ways of documenting requirements.

By the mid 1980's, my solution to the problem of ambiguity in requirements was to use more and more formalism. Formalism allows requirements writers to remove some of the ambiguity by replacing natural language with notations that possess unambiguous semantics. By 1990, I had written *Software Requirements*, a book that explored many of these rich notations.

By the mid 1990's, I was experiencing more and more push-back from customers. Computer-science-savvy customers embraced the notations. Of course, the software engineers loved the formalisms, such as finite state machines, decision tables, Petri

nets, statecharts, and so on. However, a very large majority of the customers I was meeting were not computer-science-savvy and had no interest in becoming so. What they wanted was very simple: to have their real-world problems solved.

Around the turn of the millennium, I realized why we communicate on a day-to-day basis in natural language: It works. One secret to writing good requirements, then, is to write them primarily in the language of the customers. For example, consider these customers:

- a hospital administrator looking for a new patient records system
- a military officer interested in procuring a new weapon control system
- a marketing person looking for a new way to build a Website quickly
- an employee in an operating division of any company

What language is spoken and understood by these customers? Answer: natural language.

However, the days of large, word-processed requirements documents are over. These days, there are too many things that a manager needs to do quickly—more quickly than a word-processed document can provide. For example, a manager needs to know the following:

- How many requirements are there in Release 2.0?
- How many high-priority requirements have been delayed until Release 3.0?
- What percentage of the requirements for Release 2.0 are low-priority?
- Which requirements in Release 2.0 are high-priority, are being built for Customer X, and are the responsibility of Sally?

It was the need for quick answers to questions like these that helped me conclude that the only way to record requirements when pressed for time is to *list* the discrete requirements, annotating each with multiple attributes.

Having a list of requirements solves many problems, but it misses a major purpose of creating requirements in the first place.

We create requirements to address needs, or markets. Without a thorough understanding of those needs, we are wasting our time. What good are "perfect" requirements—ones that are nicely worded and neatly laid out in a table—if they fail to address the customers' needs?

This book is all about how to discover, prune, and document requirements when you are subjected to tight schedule constraints. If you are in an environment with unlimited resources and unlimited time, then you need not read this book.

March 2005 A.M.D.
Colorado Springs, Colorado

This page intentionally left blank


Just
Enough
Requirements
Management

This page intentionally left blank

THREE

Requirements Triage

As you discovered in the previous chapter, poorly executed requirements elicitation can hamper development. In this chapter, you will learn more about the second major area of requirements management: requirements triage.

Definitions and Terminology

After eliciting and creating a list of requirements, you will likely want to establish (or learn about) the desired schedule and the available budget. No matter how ample either of these appears to be, one of them, if not both, will be insufficient to address all the requirements. To solve this problem, you will need to find some way to balance the desired requirements, your available budget, and the desired schedule. *Requirements triage* is the art of selecting the right requirements to include in the next release of your product [DAV03]. Although many people with many diverse titles are assigned the task of performing elicitation, very few ever acknowledge this responsibility.

In most organizations, triage is not performed explicitly. Instead, it is performed by some combination of intimidation,

inertia, osmosis, and incompetence. Here are some typical scenarios:

- *The "You're Not a Team Player" Approach:* Development and marketing cannot agree on a set of requirements and a schedule. Specifically, marketing "demands" that all the requirements must be included by a particular date "or the product might as well not be built." Development knows that the date is totally unreasonable, given the set of requirements, and fights back. Notice that both parties are trying to represent the best interests of the company. Frustrated, marketing approaches executive management, explains how important it is to have all the requirements satisfied by the date, and then informs the boss of development's "obstructionist attitude." Development is then seen as "not a team player," and executive management demands that it conform. The manager of development agrees to the date even though he knows it is impossible to meet. The project is not delivered on schedule. Notice that the development organization is able to be late and still say, "I told you so."
- *The "You Don't Understand Technology" Approach:* Development and marketing cannot agree on a set of requirements and a schedule. Specifically, marketing "demands" that all the requirements must be included by a particular date "or the product might as well not be built." Development knows that the date is totally unreasonable, given the set of requirements, and fights back. Frustrated, development approaches executive management, explains how important it is for the company to be successful, and then informs the boss of marketing's "death wish." Marketing is seen as trying to set up the company to fail by insisting that it try to accomplish the impossible. The manager of marketing agrees to either reduce requirements or delay the product release, even though she knows that the product will no longer be competitive. When marketing fails to meet revenue goals, it has the perfect scapegoat: development.
- *The "Over-Estimate" Approach:* Development knows from past experience that it is going to be forced to accept a ridiculously tight schedule. Therefore, developers over-

estimate the effort needed to address each requirement. They hope that by doing so, they will have enough slack built into the schedule for them to actually succeed. This scenario may result in a successful product delivery. Unfortunately, the games being played make it impossible to ever repeat the process in a predictable manner. None of the players learn anything about why it worked or how to improve it. Executive management may walk away from the project feeling that it was successful, but for all the wrong reasons.

- *The "Over-Demand" Approach:* Marketing knows from past experience that development is going to deliver late regardless of what is needed. Therefore, it gives development an earlier deadline for product delivery, hoping that when the product is delivered, although late relative to the deadline, it will still be early enough to meet the real market window. This scenario may result in a successful product delivery. Unfortunately, the deceit involved makes it impossible to ever repeat the process in a predictable manner. No one involved learns anything about why it worked or how to improve it. But the worst part of this scenario is that development gets an undeserved "bad rap."

Synonyms for requirements triage include release planning [CAR02], requirements prioritization [WIE03], optimal attainment of requirements [FEA02], requirements negotiation [IN01], requirements selection [KAR96, RUH03], and requirements allocation.[1] I prefer the term "triage" because the analogy to triage in medicine is so fitting. As I described in [DAV03], after a medical disaster, medical personnel "systematically categorize victims into three groups: those who will die whether treated or not, those who will resume normal lives whether treated or not, and those for whom medical treatment may make a significant difference. Each group requires a different strategy. The first group receives palliative care, the second group waits for treatment, and the third requires some ranking in light of available resources. As new victims appear, personnel must repeat the categorization."* Notice how similar this is to the software world. Some requirements are no-brainers—we

[1] "Requirements allocation" more commonly connotes the assignment of requirements to specific software components or sub-systems rather than to specific releases.

*Alan Davis, "The Art of Requirements Triage," *IEEE Computer*, Vol. 36, No. 3 (March 2003), p. 42. © 2003 IEEE. Used by permission.

absolutely must address them or the product won't do its job. Other requirements are a different kind of no-brainer—just dreams that should remain unfulfilled until the appropriate resources are available. The third set requires ranking in light of available resources. As new requirements emerge or resources change, new rankings must be performed.

Triage is the most interdisciplinary of the three areas of requirements management. Successful triage requires close interplay between those responsible for understanding customer needs and the timing of those needs,[2] those responsible for expending resources to satisfy requirements,[3] those responsible for the allocation of money to the project,[4] and those responsible for overall project success.[5] Successful triage requires knowledge of the following:

- *The needs of the customers:* What problems does the customer have, and what is the relationship of various product requirements to the solution of those problems?
- *The relative importance of requirements to the customers:* Which requirements have the most (and least) value to the customer? If you simply ask this question directly, in most cases the answer will be, "They are all critical." Later in this chapter, I discuss better ways to determine relative importance.
- *Timing:* What is the market window? By when does the customer need each requirement addressed? If you simply ask this question directly, in most cases the answer will be, "I need them all today."[6] Later in this chapter, I discuss better ways to determine relative timing.

---

[2]In an organization planning to sell the product being specified to a commercial market, these individuals are typically called *marketing personnel.* In an organization planning to build a product for internal use, they are typically called *analysts.* In an organization planning to build a custom product for an individual customer, they are typically the *customers* themselves.

[3]When building software for internal business use, these individuals are typically called the *IT department.* Otherwise, they are called the *R&D organization,* the *software development organization,* or *software engineering.*

[4]These are the individuals responsible for funding the software development organization, and will vary based on whether the funding is allocated via corporate line management, by the customer directly, or by a project organization.

[5]Typically called a *product manager, project manager,* or *program manager.*

[6]Or worse, "yesterday."

- *Relationships among requirements:* Which requirements make sense only when other requirements are already present? For example, it makes little sense to include a requirement to bill for a particular service if the service itself is not being provided by another requirement. Are some requirements easier to implement after other requirements have already been implemented? Which ones? Which requirements are no longer needed when other requirements are met? Which requirements are incompatible with which other requirements?
- *Sensible subsetting* [PAR76]: Which sets of requirements make business sense only when all members are present?
- *Cost to satisfy each requirement:* How many resources (measured in terms of currency, effort, or elapsed time) will need to be expended in order to satisfy each requirement?

Triage should be performed for every planned release of a product, as indicated by the left-most oval in Figure 3-1. It is during this time that most requirements are assigned critical attributes and major decisions are made concerning which requirements will be addressed in the next release. As shown by the center and right-most ovals in Figure 3-1, triage activities are repeated each time new requirements arise or new resources become available.

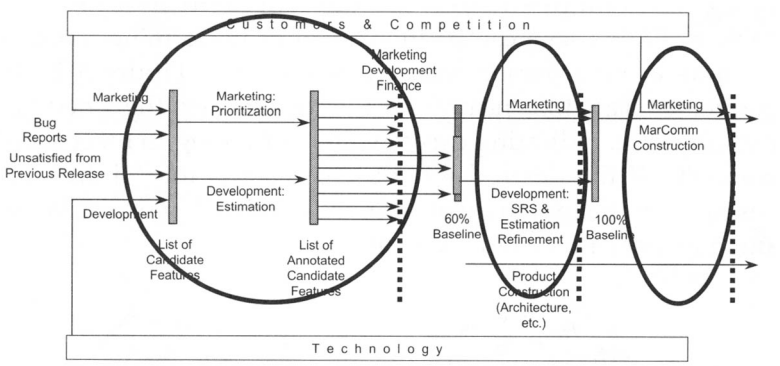


Figure 3-1: Triage in the Requirements Process.

Why Do Triage?

In an attempt to do *just enough* requirements management, you may be tempted to dispense with triage. After all, most organizations don't do it, anyway. And it is difficult to do. Furthermore, in

most companies, development, finance, and marketing rarely get along with each other. So why bother? The answer lies in the fact that most systems built today do not meet customer expectations. Perhaps the lack of triage is the very reason for this. Not doing triage guarantees that your organization is taking huge risks: the risk of satisfying the wrong requirements, the risk of promising to meet a schedule only to miss it significantly, the risk of agreeing to satisfy requirements for a given budget only to exceed it significantly, and so on.

Performing triage is the most effective way to achieve *just enough* requirements management. It is triage that enables the rest of system development to proceed on schedule while still providing a quality product. Not performing triage is not an option. If you do not do it explicitly, it will occur implicitly. And if performed implicitly, you are leaving the success of your project to pure chance.

Basic Triage Techniques

This section of the book differentiates between *basic* and *advanced* triage. Basic triage is the art of achieving a balance between requirements, development cost, development schedule, and risk. It is the minimal amount of triage that should be performed when trying to accomplish software development in a *just enough* manner. Basic triage can be thought of as performing a balancing act upon a three-person seesaw, as shown in Figure 3-2. If the requirements are too "heavy" for the seesaw, then the schedule and cost arms rise, indicating excessive risk of failing to meet schedule and cost. If the desired schedule or cost is too "light" for the seesaw, then the requirements arm falls, indicating excessive risk of failing to meet all the requirements.

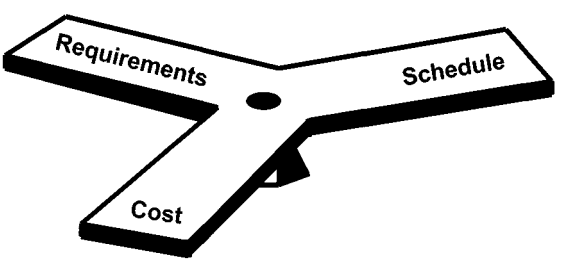


Figure 3-2: Basic Triage Represented as a Three-Person Seesaw.

Performance of basic triage necessitates that you know something about the requirements being considered for inclusion. At a minimum, this knowledge should include the relative importance of the requirements (from the customer perspective) and the cost of satisfying each requirement. There are many other attributes, but these are the two that you cannot afford to omit.

Prioritizing Candidate Requirements by Importance and Cost

Numerous techniques exist to determine the relative importance of requirements. Ironically, one technique that does *not* work is to simply ask individual customers to prioritize the requirements. If you do this, most customers will tell you that all the requirements are of equal importance and they are all critical. Instead, gather stakeholders together in a room, show them the list of candidate requirements, and ask them to apply the hundred-dollar test, the yes-no vote, or the five-way priority scheme.

To apply the hundred-dollar test [LEF00], explain to the stakeholders that they each have one hundred (imaginary) dollars that they must distribute among the candidate requirements—in this case, let's say we have 100—in such a way that if requirement i is x times more important than requirement j, then they should give it x times more money. Also, explain to them that it would be pointless to give $1.00 to each requirement because when we sum all the votes, their vote would have no effect on the outcome, so this is equivalent to not voting at all. If the group is small enough, simply point to each candidate requirement, conduct a short discussion to make sure that everybody understands the requirement sufficiently, and then go around the room asking each stakeholder for his or her points. Add as you go, and write the sum of the votes next to the requirement. Repeat this for every candidate requirement. Sort the requirements from most to least points, and—voilà!—you have your prioritization.

If the group is too large to count the votes in real time, you may do the counting one of two ways. First, if the candidate requirements are already in an electronic medium, have all the stakeholders record their votes using a computer. If the candidate requirements are written down (on cards, as a result of brainstorming, for example), simply give the stakeholders a marker and ask them to record their votes on the cards. The biggest advantage

of the hundred-dollar test is that stakeholders can vote how they really believe, so if they believe one requirement is ten times more important than another, they can give it ten times more votes.

The biggest disadvantage is that the hundred-dollar test can easily be "gamed." Let me tell you about a situation that occurred a few years ago, an excellent example of gaming: We had gathered seven or eight stakeholders together in a room to discuss the next release's features. We started with a two-hour discussion of all the requirements. During this time, one stakeholder expressed his view that only four of the requirements were of high importance to him. It also became evident that everybody else in the room shared his assessment of three of the four, but only he cared about the fourth one. When it came to a vote, he realized he would be wasting his vote on the first three, since the other stakeholders would vote for them. So, he put all one hundred dollars on the fourth requirement. After tallying the votes, we discovered that the fourth requirement received the highest number of total votes, even though only one stakeholder cared about it.

Collusion is another game that stakeholders sometimes play. This also happened to me a few years ago: I had gathered together six stakeholders for a hundred-dollar test. I had warned them up-front of the futility of giving every requirement the same number of points. I pointed to the first requirement and added up the votes: $6.00. I pointed to the second requirement and added up the points: $6.00 again. After this happened eight more times, I realized that they must have colluded before entering the room. I could imagine them saying, "Davis is going to try to get us to agree that some of the requirements are less important than others! We all know that they are all of equal importance, so let's not let him get away with it." When this became evident to me, I suggested a short recess. During the recess, I cornered one of the developers and asked him to join us for the rest of the voting. Upon reconvening, I announced, "I have asked Quinn to join us for the remainder of the voting. As you all know, I strongly believe that developers' opinions on the importance of requirements are not as important as customers' opinions. For that reason, I am giving Quinn just $1.00 to distribute in one-cent increments among the one hundred candidate requirements. Nobody should feel intimidated by Quinn's presence. After all, no matter how he chooses to vote, any one of you can easily out-vote him with just one of your

hundred dollars." And so we continued the voting process. The first requirement Quinn voted on earned $6.10, the next earned $6.01, and so on. Within a few minutes, it became clear to the stakeholders that if they continued with their collusion game, Quinn would be making all the prioritization decisions. Finally, one of the original stakeholders asked if we could start over again. During this second run, the stakeholders voted more accurately. Some would say I tricked the stakeholders into voting in a differentiated manner, but Quinn's presence would not have had any effect on their votes if their votes actually reflected how they felt about the relative priorities of the requirements.

To address the inherent weaknesses of the hundred-dollar test, some organizations use a much simpler voting system: the yes-no vote. In this system, simply point to each requirement and ask stakeholders to indicate their interest-level in the requirement by raising their hands. There is no reason to establish limits on the hand-raises. If stakeholders choose to raise their hands to all or none of the requirements, they should feel free to do so. No matter how many people are in the room, it is pretty easy to tally the number of raised hands. Record these numbers next to the requirements and sort them from highest to lowest to find the relative prioritization of your candidate requirements. The yes-no vote is very easy to administer, but has two major problems:

- What does it mean for a stakeholder to not raise a hand for a requirement? Does it mean that he or she does not care if the requirement is satisfied in this release? Or does it mean that he or she believes that including the requirement in the current release would actually detract from its usefulness?
- How should a stakeholder vote if he or she really believes that one requirement is twice as important as another requirement, but that both should be included in the next release?

To overcome the weaknesses of the yes-no vote while maintaining simplicity, try using the five-way priority scheme, which is the system I use most. Conduct the voting the same way that the yes-no vote demands, but give the stakeholders five options. They can vote +1 if they are for the requirement's inclusion in the next release, 0 if they don't care, and -1 if they are against its inclusion.

They can also vote +2 if they feel the requirement is extraordinarily important, and -2 if they feel its inclusion is extraordinarily destructive to the release. To make the tallying process easy, ask the stakeholders to vote with their fingers, as shown in Figure 3-3. When I am facilitating a triage session with ten or fewer stakeholders, I usually record all their votes rather than just the sum, as shown in Figure 3-4. My reason for doing this is simple: Let us say that two requirements each score 0 points, but one is the result of every stakeholder voting 0 and the other is the result of five stakeholders voting +2 and five stakeholders voting -2. In the former case, the requirement should be excluded from the baseline. But the votes for the latter requirement indicate that we may need to consider two versions of the product.

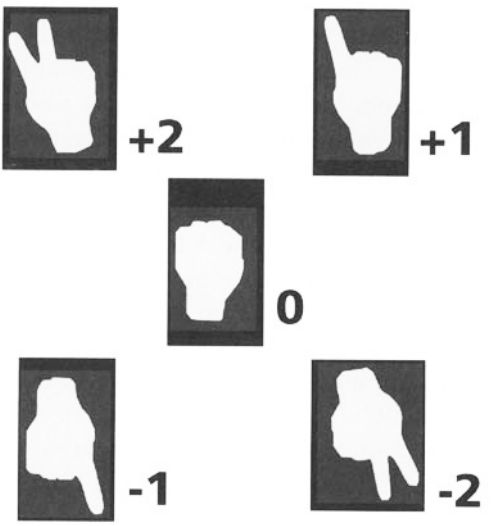


Figure 3-3: Five-Way Priority Scheme.

The above prioritization schemes work as stated when the stakeholders are all equally important. If some stakeholders are more important than others, you may want to modify the data collection method somewhat. First of all, if the politics of the situation are such that all parties understand which stakeholders are most important, then simply weight those individuals' votes accordingly. For example, if everybody knows that stakeholder *x* is twice as important as anybody else, just count his or her votes twice. If the politics dictate that the relative importance of stakeholders

remain clandestine, then it is best to collect all the votes electronically (or secretly) and do the tallying remotely.

Reqt. No.	Requirement Text	Priority by Stakeholder									
		A	B	C	D	E	F	G	H	I	J
1	The system shall be programmable by the operator.	+2	+2	0	-1	+2	+1	+2	+2	0	0
1.1	The system shall be programmable by the operator to set the default for the green direction to be "East" or "West."	+2	+2	0	0	+1	+1	+2	0	0	0
1.2	The system shall be programmable by the operator to set the maximum duration for the light to remain green in the non-default direction.	+2	+2	0	0	+2	+2	+2	+2	0	0
1.3	The system shall be programmable by the operator to set the minimum duration for the light to remain green in the default direction.	+2	+2	0	0	+2	+2	+2	+2	0	0
1.4	The system shall be programmable by the operator to set the duration of the amber light prior to it changing to red.	+2	+2	0	-2	+2	+1	+2	0	0	0
2	The system shall provide safe access to a one-lane east-west bridge via green/amber/red traffic lights.	+2	+2	+2	+2	+2	+2	+2	+2	+2	+2
2.1	Two sets of traffic lights shall be controlled by the system.	+1	+1	0	+2	+2	0	+2	+1	0	+2
2.2	When either set of lights is "green," the other set of lights shall be set to "red."	+2	+2	+2	+2	+2	+2	+2	+2	+2	+2
2.3	When the system determines that it is time to switch the direction of traffic, it shall do so in a safe manner.	+2	+2	+2	+2	+2	+2	+2	+2	+2	+2
3	The system shall control eastbound traffic coming from northwest and southwest converging roads.	+2	+2	+2	-2	-2	-2	+2	+2	+2	+2
4	The system shall control westbound traffic coming from northeast and southeast converging roads.	+2	+2	+2	-2	-2	-2	+2	+2	+2	+2
5	The system shall interface to vehicle sensors capable of determining if there is a vehicle waiting at either of the two entrances.	+2	+2	+2	+1	+1	-1	+1	+1	+1	0
6	The system shall interface to vehicle counters capable of counting vehicles as they pass through each of the two entrances.	+2	+1	+1	+2	+2	+2	+2	+2	0	+1
7	The system shall sense the weight of vehicles on the road and not allow either light to turn to green while a vehicle remains on the bridge.	-2	0	-1	+2	-1	+1	+2	+2	-1	+1

Figure 3-4: A List of Prioritized Candidate Requirements.[7]

[7]Sometimes, prioritization is only applied down to a certain level of refinement; in this case, it's to the second level only. That is why I have omitted requirements 2.3.1 and 2.3.2 from this figure.

During the triage discussion, one stakeholder decided that he needed a more complex system: one that controlled four roads (as opposed to two) converging on the one-lane bridge. I have therefore added new requirements 3 and 4. Note that the votes indicate quite a bit of controversy surrounding these new requirements.

Another stakeholder, during the triage session, realized that we had inadvertently omitted an important requirement—interfacing to sensors that detect arriving traffic. This is why requirement 5 was added. Look at the votes for this one. Here, one stakeholder voted "-1" because she felt it was "obvious."

If you are looking for a technique that incorporates multiple stakeholders, each with differing levels of importance, and a way of propagating such data to the decision process, take a look at quality function deployment (QFD) [MIZ94]. In QFD, a table is created with each stakeholder name and its relative importance atop each column, and each individual requirement labeling the rows. In each column, record that stakeholder's votes for the requirements. After completing the data entry, do a weighted sum of the rows,[8] and record that sum in a new column to the right. This becomes your relative priority of requirements.

Estimating Effort for Candidate Requirements

There are many books that can assist organizations in determining the quantity of resources required to build software to solve a problem (for example, see [BOE00, JON98]). When distilled, though, all of them will tell you basically the same thing:

- Estimate the size of the problem that needs to be solved or the solution that needs to be built.[9]
- Tweak the size based on some attributes of the problem to be solved, the people working on the project, or the type of solution.
- Use historic data to see how many resources were required in the past to tackle such an endeavor.

The effort estimation should be done on a per-requirement basis. In other words, you should estimate the effort required to satisfy each requirement. The units of effort can be anything you like: feature points, function points, lines of code, person-years, person-months, person-weeks, person-hours, dollars, and so on. Figure 3-5 shows the example of Figure 3-4 with the individual stakeholder priorities summed and with the estimate efforts, in person-hours.

---

[8]If the table entry at the intersection of row (requirement) i and column (stakeholder) j is termed v_{ij}, and the relative importance of stakeholder j is r_j, then the relative priority p_i of each requirement i is the weighted sum, that is,

$$p_i = \sum_j (r_j \times v_{ij})$$

[9]Although arguments exist for just about any method of measuring the size of the endeavor, I will not cover them—the methods or the arguments—here.

Reqt. No.	Requirement Text	Priority	Person-Hours
1	The system shall be programmable by the operator.	10	120
1.1	The system shall be programmable by the operator to set the default for the green direction to be "East" or "West."	8	20
1.2	The system shall be programmable by the operator to set the maximum duration for the light to remain green in the non-default direction.	12	20
1.3	The system shall be programmable by the operator to set the minimum duration for the light to remain green in the default direction.	12	20
1.4	The system shall be programmable by the operator to set the duration of the amber light prior to it changing to red.	7	15
2	The system shall provide safe access to a one-lane east-west bridge via green/amber/red traffic lights.	20	200
2.1	Two sets of traffic lights shall be controlled by the system.	11	incl
2.2	When either set of lights is "green," the other set of lights shall be set to "red."	20	incl
2.3	When the system determines that it is time to switch the direction of traffic, it shall do so in a safe manner.	20	incl
3	The system shall control eastbound traffic coming from northwest and southwest converging roads.	8	150
4	The system shall control westbound traffic coming from northeast and southeast converging roads.	8	150
5	The system shall interface to vehicle sensors capable of determining if there is a vehicle waiting at either of the two entrances.	12	40
6	The system shall interface to vehicle counters capable of counting vehicles as they pass through each of the two entrances.	15	120
7	The system shall sense the weight of vehicles on the road and not allow either light to turn to green while a vehicle remains on the bridge.	3	200

Figure 3-5: A List of Candidate Requirements with Effort Estimates.[10]

Disagreements Concerning Relative Priority of Requirements

Let's say that two members of the marketing team have seriously differing opinions about the relative priority of the following requirement:

A. The system shall provide service x *to the customers.*

[10]As indicated by "incl" in the Person-Hours column, it is sometimes impossible to divide the effort of a requirement into its sub-requirements. If this is so, don't worry about it; just record that you have made that decision.

There could be a variety of reasons for this disagreement. One possibility is that the two marketing people represent two disparate groups of customers with different needs, based on the jobs that they do. In that case, the best route is to record multiple attributes for each requirement, one for each of the priorities, for each of the groups, as shown in Figure 3-4. Averaging or combining the priorities is unlikely to be of much value. For example, if a requirement is essential to meet the needs of one class of customer but unimportant to another, nothing worthwhile results from recording an average priority of "important." The fact is that if the requirement is met, the first group of customers will have its needs met. And if the requirement is not met, that group's needs will not be met. The satisfaction of the second group of customers is completely unrelated to this requirement.

Another possible basis of the disagreement is that the two marketing people have different interpretations of the candidate requirement. In this case, the correct route is to refine the requirement. By refining it, you will improve your understanding of the requirement and discover the basis of the disagreement. So, in the above example, we could refine the requirement into, say, three sub-requirements:

A1. The system shall provide service x1 *to the customers.*

A2. The system shall provide service x2 *to the customers.*

A3. The system shall provide service x3 *to the customers.*

This is what was done to create some of the hierarchies of requirements shown in Figure 3-4.

Once this is done, it often becomes clear that the person who rated requirement *A* highly did so because service *x3* is of critical importance (and perhaps *x1* and *x2* were actually of little value). And the person who gave a low rating to requirement *A* did so because he or she thought that *x* referred only to *x2*, rather than *x1*, *x2*, and *x3*. With the refined requirements, however, the parties realize that they actually agreed with the priorities—requirement *A1* is low-priority, requirement *A2* is low-priority, and requirement *A3* is high-priority. But, equally important is the side effect this refinement has of reducing the ambiguity of requirement *A*.

Disagreements Concerning Effort to Satisfy a Requirement

Let's say that two members of the development organization have seriously differing opinions about the effort required to address the aforementioned requirement *A*.

There could be a variety of reasons for the disagreement. One possibility is that the two developers have different assumptions concerning the development resources that are to be applied. For example, Sally may be assuming that she will be assigned to the development effort, and she knows she is highly skilled in this type of work. Knowing this, she has provided a relatively low estimate of effort. Meanwhile, John may be assuming that he will be assigned the development effort, but really does not like doing this type of work. Feeling this way, he has provided a relatively high estimate of effort. These are but two of the many games developers play when making estimates.[11] If such is the case, the best route is to find out from Sally and John "where their heads are at." That is, try to ascertain what games they are playing, so you can make the right decisions. Then select either Sally's or John's estimate, and be sure to record the assumptions that you are making. Whatever you do, do *not* average the two estimates.

An equally likely reason for the disagreement is that the two developers have different interpretations of the candidate requirement. In this case, you should take the same action as in the case of a disagreement among marketing personnel—refine the requirement:

A1. The system shall provide service x1 *to the customers.*

A2. The system shall provide service x2 *to the customers.*

A3. The system shall provide service x3 *to the customers.*

Once this is done (as we did above, to clarify relative priority), it often becomes clear that the person who estimated that it would require a lot of effort to satisfy requirement *A* did so because service *x3* is very difficult to provide (and perhaps *x1* and *x2* are easy to provide). And the person who estimated that it would require little effort to satisfy requirement *A* did so because he or she thought that *x* meant *x2* only, rather than *x1*, *x2*, and *x3*. After

[11] For a more complete description of these games, read [DAV04].

refinement, however, the parties realize that they actually agree with the estimates. Equally important is that the ambiguity of requirement *A* has been reduced.

Establishing Requirements Relationships

The preceding subsections assumed that requirements are independent. In reality, this is rarely the case. The following sections describe just some of the relationships that may exist between requirements.

Necessity Dependency

In this relationship, it makes sense to satisfy requirement *A* only if we are also satisfying requirement *B*. For example, consider this dependency:

> *Requirement A: The stop button shall be red.*
>
> *Requirement B: The system shall provide a stop button.*

Sometimes, such a dependency is bidirectional, as in this case:

> *Requirement A: The system shall provide the Fiends and Famine capability for our customers.*
>
> *Requirement B: The system shall bill customers $3 per minute when they use the Fiends and Famine feature.*

If such dependencies exist between requirements, you should record the relationships, as shown in Figure 3-6 and Figure 3-7. The record of this relationship will be used during requirements triage. As we try to select an optimal subset of the requirements to implement, we will want to avoid subsets that include a requirement without including all requirements on which it depends. Pär Carlshamre *et al.* [CAR01] differentiate between two types of necessity dependence: AND (which indicates bidirectional necessity dependence between two requirements) and REQUIRES (which indicates a unidirectional necessity relationship between one requirement and another).

ID	Requirement Text	Necessity Dependency
A	The stop button shall be red.	
B	The system shall provide a stop button.	

Figure 3-6: Unidirectional Necessity Dependency.

ID	Requirement Text	Necessity Dependency
A	The system shall provide the Fiends and Famine capability for our customers.	
B	The system shall bill customers $3 per minute when they use the Fiends and Famine feature.	

Figure 3-7: Bidirectional Necessity Dependency.

Effort Dependency

In this relationship, requirement *A* will be easier to satisfy if we are also satisfying requirement *B*. For example, such a dependency may exist between the following:

> *Requirement A: The system shall provide reports of accounts receivable older than 60 days.*
>
> *Requirement B: The system shall provide a general-purpose report-generation utility.*

If such a dependency exists between requirements, you should record that relationship, as shown in Figure 3-8. Unlike necessity dependency, effort dependency is not used to select compatible requirements. Instead, it is used to determine the actual implementation cost for selected subsets. So, in the case of Figure 3-8, the cost of satisfying requirement *A* is four person-months and the cost of satisfying requirement *B* is five person-months, but the cost of satisfying both requirements *A* and *B* is seven person-months, because satisfying requirement *B* reduces the effort required to satisfy requirement *A* by two person-months.

ID	Requirement Text	Effort Estimate	Effort Dependency
A	The system shall provide reports of accounts receivable older than 60 days.	4	-2
B	The system shall provide a general-purpose report-generation utility.	5	

Figure 3-8: Effort Dependency.

Pär Carlshamre *et al.* [CAR01] separate this relationship into two subtly different characteristics: TEMPORAL dependencies, to capture the fact that one requirement should be implemented before another, and ICOST dependencies, to capture the incremental cost reduction gained by implementing one requirement before another.

Subset Dependency

If requirement *A* is satisfied, then requirement *B* will be satisfied. That is, requirement *B* is part of requirement *A*.

If such a dependency exists between requirements, you should record that relationship. Notice from the list of requirements shown in Figure 3-9 that we have refined the parent requirement into three child requirements. In this case, the three children do not represent the entirety of the parent's functionality. The intent is for them to exemplify or capture the *spirit* of the parent. Notice in this case that developers must satisfy the full essence of the parent requirement—in doing so, they'll satisfy the three children requirements, but that is not sufficient. Note also that during the actual triage process, when selecting an optimal subset of the requirements to satisfy, you may select all, some, or none of the children. And if you select all the children, then you may also select the parent if you wish. However, if you select the parent, you *must* select all its children. You will notice that there is no need to define effort dependencies between parent and children requirements. If we select *A* for satisfaction, we will automatically select *B*, *C*, and *D*, and the total effort will be 27 person-months. If we select only a subset of *B*, *C*, and *D*, then the effort will be only the sum of the efforts of the selected requirements.

ID	Requirement Text	Effort Estimate	Subset Dependency
A	The system shall enable the user to specify the order of the displayed list in a wide variety of ways.	5	
B	The system shall enable the user to specify that the list be displayed in alphabetic order.	7	
C	The system shall enable the user to specify that the list be displayed in chronological age order.	2	
D	The system shall enable the user to specify that the list be displayed in employment seniority order.	13	

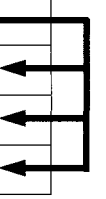


Figure 3-9: Subset Dependency.

An alternate way to represent subset dependency is by numbering the requirements accordingly, so that if a parent is numbered 3, then its children are numbered 3.1, 3.2, 3.3, and so on. This is precisely what is being represented in Figure 3-4. That figure has also indented the requirements texts for children so they can be even more easily identified. In such cases, there is no need to use a separate column to show the dependency. Juha Kuusela and Juha Savolainen [KUU00] call this an AND dependency because child requirements 3.1 *and* 3.2 *and* 3.3 represent the parent requirement 3.

Cover Dependency

This is a special case of subset dependency in which the union of the children's functionality *is* the parent's functionality. If requirements *B*, *C*, and *D* are satisfied, then requirement *A* will be satisfied; and if requirement *A* is satisfied, then requirements *B*, *C*, and *D* will be satisfied. That is, requirements *B*, *C*, and *D* represent a refinement of requirement *A*.

If such a dependency exists between requirements, you should record that relationship. You will notice from the list of requirements shown in Figure 3-10 that we have refined the parent requirement *A* into three child requirements—*B*, *C*, and *D*—and we *do* wish to imply that the sum of the three children completely captures the parent. Note that during the actual triage process, when selecting an optimal subset of the requirements to satisfy, you may select all, some, or none of the children. And if you select all the children, then you have indeed selected the parent by inference. If you select the parent, you have selected all children requirements by inference. Notice that once again there is no need to include effort dependency.

ID	Requirement Text	Effort Estimate	Cover Dependency
A	The system shall enable the user to specify the order of the displayed list in three ways.	5	
B	The system shall enable the user to specify that the list be displayed in alphabetic order.	7	
C	The system shall enable the user to specify that the list be displayed in chronological age order.	2	
D	The system shall enable the user to specify that the list be displayed in employment seniority order.	13	

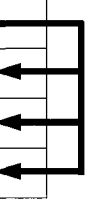


Figure 3-10: Cover Dependency.

Value Dependency

If satisfying one requirement lowers (or raises) the need for another requirement, then a value dependency exists between them [CAR01].

If such a dependency exists between requirements, you should record that relationship as you would other types of dependencies. When a value dependency exists, triage will become a bit more complex: The relative priority of requirement *B* will change as a result of including requirement *A* in the next release. If the dependency has not been recorded, the triage participants are likely to include requirement *B* even though it is no longer that important.

Pär Carlshamre *et al.* [CAR01] also describe a requirements relationship they call an OR dependency. When two requirements possess an OR relationship, then one of the requirements should be satisfied, but not both. I believe that this is just a special case of a bidirectional value dependency. In particular, if requirement *A* is selected for inclusion, the priority of requirement *B* becomes negative, and if requirement *B* is selected for inclusion, then the priority of requirement *A* becomes negative.

Documenting the Dependencies

In all of the above cases, I have shown the relationships as arrows in the figures. There are a variety of ways of maintaining these relationships on an actual project:

- *Manual:* If you want requirement 1 to refer to requirement 2, simply type "requirement 2" in the appropriate attribute field of requirement 1. This requires almost no effort at all to do initially, but maintenance is extremely painful [ANT01, DÖM98, RAM98]. Every time a requirement changes, you must manually check to see if other requirements are impacted.
- *Hyperlink:* Much better than manual maintenance, this requires you to enter a hyperlink [HAY96] called "requirement 2" in the appropriate attribute field of requirement 1. The biggest advantage of this is you can quickly move from one requirement to all the related requirements. This makes maintenance somewhat easier.

- *Requirements Tools:* Ideally, when you change requirement 1, you want to immediately see a list of all requirements related to it on a first- and second-order basis (what relates to requirement 1 and what relates to what relates to requirement 1, respectively), and so on. The easiest way to do this is to use a requirements management tool,[12] which makes maintenance a lot easier. But with the benefit comes the extra overhead of having to learn how to use (and having to pay for) the tool.

Performing Triage on Multiple Releases

The essence of the triage decision is to decide whether or not each candidate requirement will be included in the next release, given the available time and resources. As described earlier, this is always the result of a cooperative process among multiple stakeholders. I highly recommend that you always plan at least two releases at a time. If you plan just one release—say, Release 3.0—every requirement necessitates a binary (yes or no) decision. There is no compromise position. If you plan two releases at a time, a compromise position is available. So, let's say one stakeholder insists a requirement be included in Release 3.0 and another stakeholder insists that it be excluded from Release 3.0. It might be possible to formulate a middle-of-the-road position by agreeing that it will be satisfied in the following release, Release 4.0. To make this even more palatable, you might consider numbering the following Release 3.1, or even 3.01 (rather than 4.0), to make it sound even more imminent and thus even more of an acceptable compromise (see Figure 3-11).

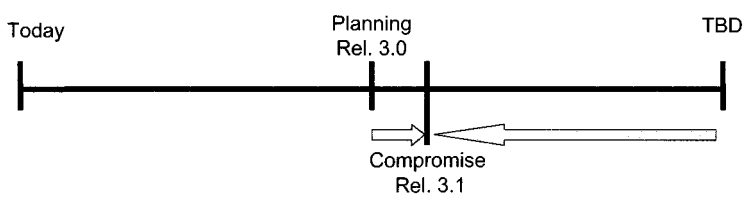


Figure 3-11: A Compromise Release.

[12]See www.incose.org for a list of available requirements management tools.

Making the Triage Decision

Now that you have a relative ranking of requirements as created by your stakeholders, you need to determine which ones to include, based on the available resources. Because of the complex relationships that exist among and between requirements, the decision is not as simple as just selecting the most desirable requirements for inclusion. Also, you can't just include all the requirements, as you have a limited amount of resources. Three general approaches exist for doing triage: optimistic, pessimistic, and realistic.[13] In the optimistic approach, you place the available budget and desired schedule on their respective arms of the seesaw shown earlier, in Figure 3-2, and then you place *all* the candidate requirements on the third arm. Remove requirements one at a time from the requirements arm until it balances. This is by far the most common way of performing triage, but not necessarily the most efficient. In the pessimistic approach, you place the available budget and desired schedule on their respective seesaw arms, and then you place requirements onto the requirements arm one at a time until it balances. In the realistic approach, you place the available budget and desired schedule on their respective seesaw arms, and a reasonable amount of candidate requirements on the requirements arm. Then you refine your requirements selection by adding and removing requirements.

Think of this process as manipulating a radio dial, one associated with each requirement.[14] The values for the positions of the dials include one position for each release being considered, one for "done" (already satisfied), and one for "TBD" (to be determined). Figure 3-12 shows what such a radio button would look like for considering Releases 2.0, 2.1, 2.2, and 3.0, while Figure 3-13 shows what the list of requirements from Figure 3-5 look like after they have been allocated to different releases. (Of course, if your tool doesn't support radio dials, you can replace them in Figure 3-13 with the appropriate version/release number.)

---

[13]Martin Feather and Tim Menzies [FEA02] use a tool to iterate through random choices toward an optimal solution.

[14]When the radio dial for a requirement is turned to a planned release, it is added to that release. When the radio dial is used for a requirement with a necessity, subset, or cover dependency on other requirements, those other requirements are also added. When the radio dial for a requirement is one of a set of requirements for which another requirement has a cover dependency, and all other members of the set are already included in the release, then the parent requirement is also added.

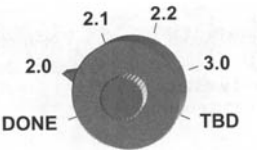


Figure 3-12: Requirement Selection Dial.

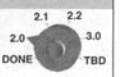
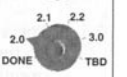
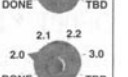
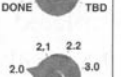
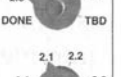
Reqt. No.	Requirement Text	Priority	Person-Hours	Release
1	The system shall be programmable by the operator.	10	120	
1.1	The system shall be programmable by the operator to set the default for the green direction to be "East" or "West."	8	20	
1.2	The system shall be programmable by the operator to set the maximum duration for the light to remain green in the non-default direction.	12	20	
1.3	The system shall be programmable by the operator to set the minimum duration for the light to remain green in the default direction.	12	20	
1.4	The system shall be programmable by the operator to set the duration of the amber light prior to it changing to red.	7	15	
2	The system shall provide safe access to a one-lane east-west bridge via green/amber/red traffic lights.	20	200	
2.1	Two sets of traffic lights shall be controlled by the system.	11	incl	
2.2	When either set of lights is "green," the other set of lights shall be set to "red."	20	incl	
2.3	When the system determines that it is time to switch the direction of traffic, it shall do so in a safe manner.	20	incl	
3	The system shall control eastbound traffic coming from northwest and southwest converging roads.	8	150	
3.1	During the period while the eastbound traffic light is authorized to be green, the system shall provide equal time for the traffic coming from the southwest and the northwest.	8	incl	

Figure 3-13: Candidate Requirements Showing Which Will Be Satisfied in Next Release.

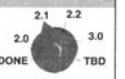
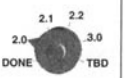
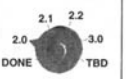
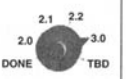
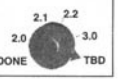
Reqt. No.	Requirement Text	Priority	Person-Hours	Release
4	The system shall control westbound traffic coming from northeast and southeast converging roads.	8	150	
4.1	During the period while the westbound traffic light is authorized to be green, the system shall provide equal time for the traffic coming from the southeast and the northeast.	8	incl	
5	The system shall interface to vehicle sensors capable of determining if there is a vehicle waiting at either of the two entrances.	12	40	
6	The system shall interface to vehicle counters capable of counting vehicles as they pass through each of the two entrances.	15	120	
7	The system shall sense the weight of vehicles on the road and not allow either light to turn to green while a vehicle remains on the bridge.	3	200	
8	If a vehicle is disabled on the bridge, the system shall automatically notify a tow truck.	1	Unknown	

Figure 3-13, continued: Candidate Requirements Showing Which Will Be Satisfied in Next Release.[15]

The only way I know to determine that the seesaw is in balance is to compare the desired schedule and available budget against the original schedule (*not* actual) and original budget (*not* actual cost) of previously completed projects that required approximately the same amount of work.

Most effort estimation techniques (examples include COCOMO [BOE00], KnowledgePLAN® [JON98], and SLIM [PUT78]) generate estimates based on the *actual* schedules and budgets of previously completed projects. The problem with this is that most organizations create their systems within a unique, unchanging culture. Thus, the rate at which requirements change (usually called "requirements creep") is similar on every project; the amount of inaccuracy of development effort estimations is similar on every

[15]During the triage process, one stakeholder expressed confusion about the meaning of requirements 3 and 4. A discussion ensued, and the agreed-upon clarifications were added as requirements 3.1 and 4.1, respectively.

Another stakeholder, during the triage discussion, thought of a new dream requirement. It was added as requirement 8. The team voted quickly on its priority, and all agreed to postpone the decision of which release it should be included in, so its release was given the value "TBD" (to be determined).

project;[16] and the politics that force under- or over-estimation are similarly unchanged [DAV04].

Although most cost and schedule estimation textbooks and methods show the distribution of similar, previously completed projects in a probabilistic distribution, such as in Figure 3-14, I have found that a cumulative probability curve, as shown in Figure 3-15, is more useful when performing triage.

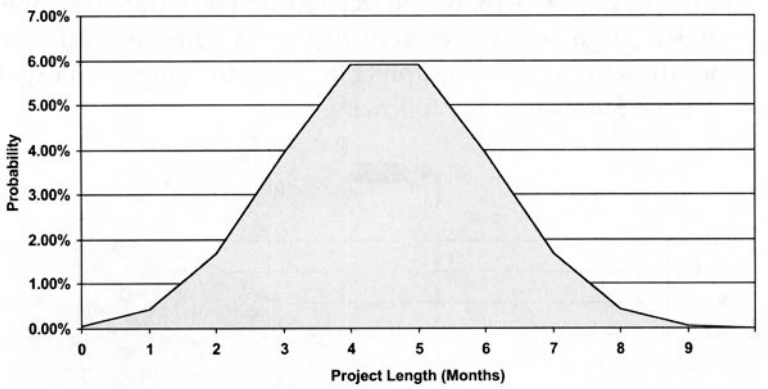


Figure 3-14: Typical Historic Project Distribution (Adapted from [DAV04]).

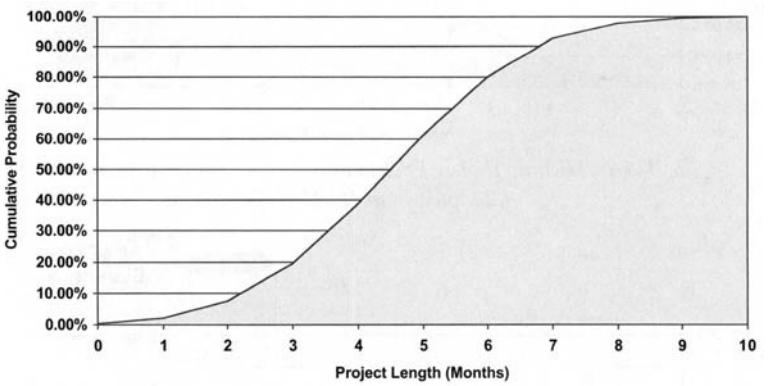


Figure 3-15: Cumulative Probability Historic Project Distribution (Adapted from [DAV04]).

[16]At one company I worked for, we discovered that the effort estimates given by the development team were consistently 25 percent lower than the actuals. So we decided on the next project to escalate all the development team's estimates by 25 percent as soon as we received them. Lo and behold, the actuals were *still* 25 percent too low. It seems that *this* corporate culture always induced the work to expand to fill 25 percent of whatever was originally estimated. This phenomenon may be true at all companies. The solution is to recognize that the up-front estimates will be 25 percent low—not to "fix" them!

In Figure 3-14, the y-axis shows the "likelihood of the project completing exactly as estimated," but in Figure 3-15, it expresses the "likelihood of the project completing at or better than the estimate," which is precisely what you want to know—the probability of success. Since the graph is a function of the weight or size of the currently selected candidate requirements, it suffices to superimpose a vertical bar on the graph (representing the desired budget, as shown in Figure 3-16, or the desired delivery date, as shown in Figure 3-17) to assess the current degree of balance in the seesaw. It is not difficult to create graphs like these for your own organization; see the Sidebar on the following page.

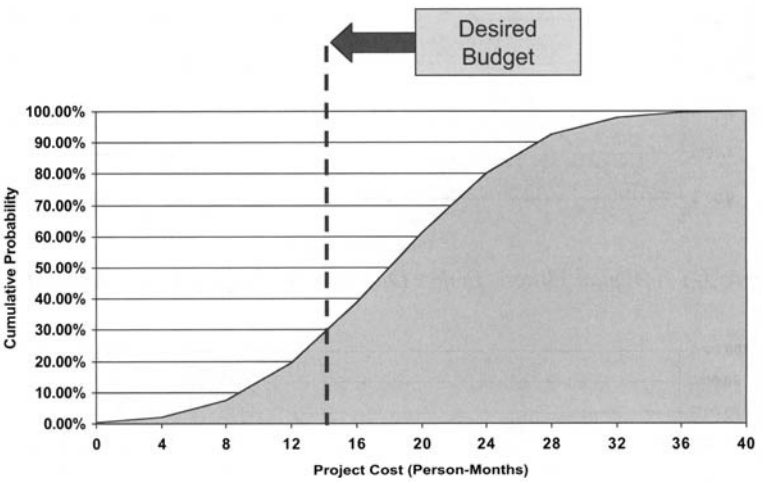


Figure 3-16: Historic Project Distribution (Cost) with Current Budget (Adapted from [DAV04]).

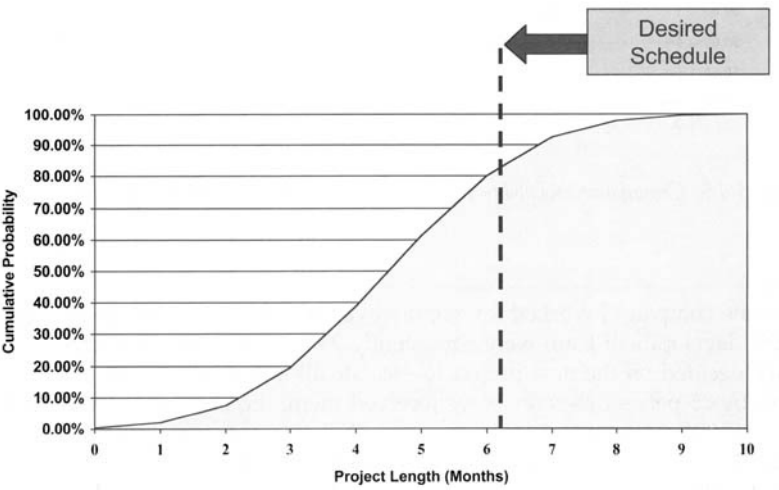


Figure 3-17: Historic Project Distribution (Schedule) with Current Desired Delivery Date (Adapted from [DAV04]).

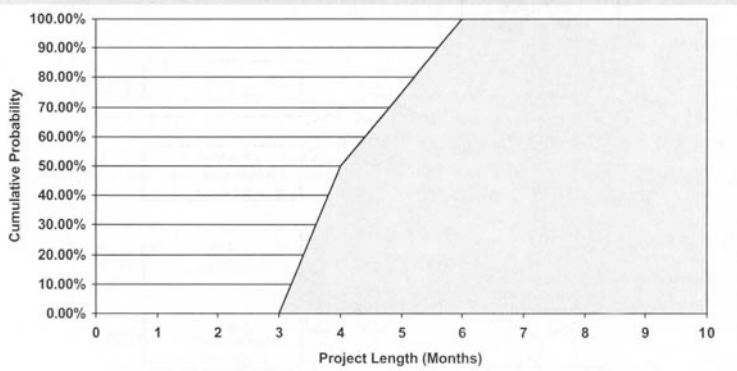
SIDEBAR

To create a graph like Figure 3-17 for your own organization, simply examine a series of past projects and collect just two pieces of information about each: the original estimate of size (in any standard units—in this case, I've used person-months), and the actual elapsed time spent on the project. Let's say that data for twenty projects look like this:

Project Number	Range of Original Estimate of Size (person-months)	Actual Project Duration (months)
1	10-19	2
2	10-19	2
3	1-9	2
4	40-49	6
5	30-39	3
6	80-89	8
7	20-29	4
8	20-29	5
9	1-9	3
10	40-49	6

Project Number	Range of Original Estimate of Size (person-months)	Actual Project Duration (months)
11	30-39	4
12	60-69	9
13	40-49	7
14	20-29	6
15	1-9	1
16	40-49	6
17	110-119	8
18	10-19	5
19	30-39	5
20	20-29	4

Now, let's say that you have a new product and you are considering the inclusion of requirements estimated to be 27 person-months in the next baseline. You can see from the above table that in the past, when you have tackled projects in the range of 20 to 29 person-months, you have never succeeded in completing them in less than four months; you completed them in four months 50 percent of the time; you managed to complete them in five or fewer months 75 percent of the time; and you succeeded in completing them in six or fewer months 100 percent of the time. So, your graph looks like this (just smooth out the lines and you're all set):



Is this extremely accurate? No, but it is good enough for making the triage decision.

Whether or not the seesaw is balanced has a lot to do with the degree of risk the organization can tolerate. For example, the graph shown in Figure 3-16 might be acceptable to an organization that thrives on risk, but the same graph might be totally unacceptable to a development organization that was recently burned severely and is looking to take a more conservative approach to product development until it regains its credibility.

Triage can be performed in a relatively orderly manner, as shown in Figure 3-18. Start with your optimistic, pessimistic, or realistic subset of candidate requirements and follow these guidelines:

1. Select the higher-priority requirements before the lower-priority ones.
2. If you include a requirement that has a necessity dependency upon another requirement, include that requirement as well.
3. Stay cognizant of which groups of requirements make sense as a commercially viable and useful product.
4. Initially, exclude any requirement that raises considerable controversy (the subsequent iterative refinement process will resolve this).

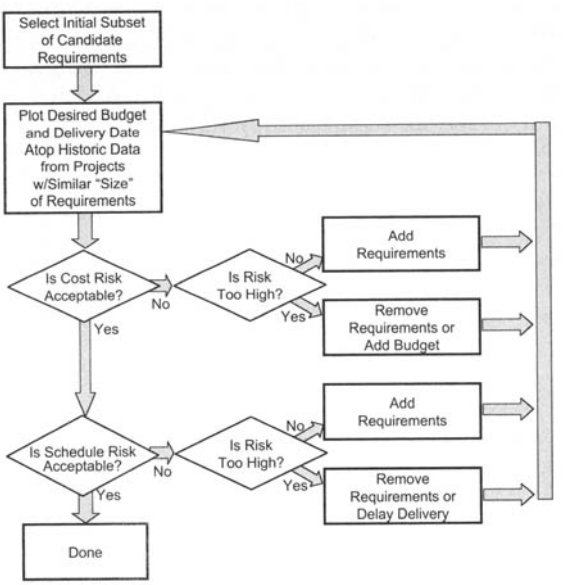


Figure 3-18: A Basic Requirements Triage Process.

Next, add up the "effort estimations" for all the selected requirements,[17] and then plot graphs (like those in Figure 3-16 and Figure 3-17) showing how previously completed projects have fared when attempting the same amount of requirements. Then, atop these two graphs, draw vertical bars representing your desired budget and schedule.

Next, examine the cost-risk graph. If the graph looks like Figure 3-16, your project has a 30-percent likelihood of completing these requirements with the desired budget. Is this an acceptable level of risk for your project? Most organizations will accept a likelihood of 80 percent or more as tolerable. Anything below that number, however, and you must decide on your level of risk adversity and, more importantly, the effect that exceeding the budget will have on the organization:

- Will additional funds be available?
- Will the project be cancelled?
- What will happen to the careers of the project members?
- What financial impact will this have on the company?
- If we are talking about human resources (which is likely for software development projects), will you be permitted to do additional hiring?

If the total picture is positive (or at least acceptable), then proceed to the next step. Otherwise, you have only two alternatives: Remove requirements or find additional resources. It is that simple.

When removing requirements, be sure to remain cognizant of the four considerations you used when selecting the initial set of candidate requirements (as described above). When adding resources, you need to know how many you need. To figure this out, just look at the graph in Figure 3-16; look at the horizontal distance between the current position of the vertical bar (the current budget) and the desired position of the vertical bar (the budget at which the risk is acceptable). To raise the likelihood of success in Figure 3-16 from the current 30 percent to a more acceptable 70 percent, just shift the vertical bar to the right until it intersects the graph at the horizontal 70-percent line. If you do that, you will have the graph shown in Figure 3-19, and you will need ten more person-months in the budget.

---

[17] Adding the estimations is not as simple as plain addition. You must consider the effects of any selected requirements with subset dependency and effort dependencies, discussed earlier.

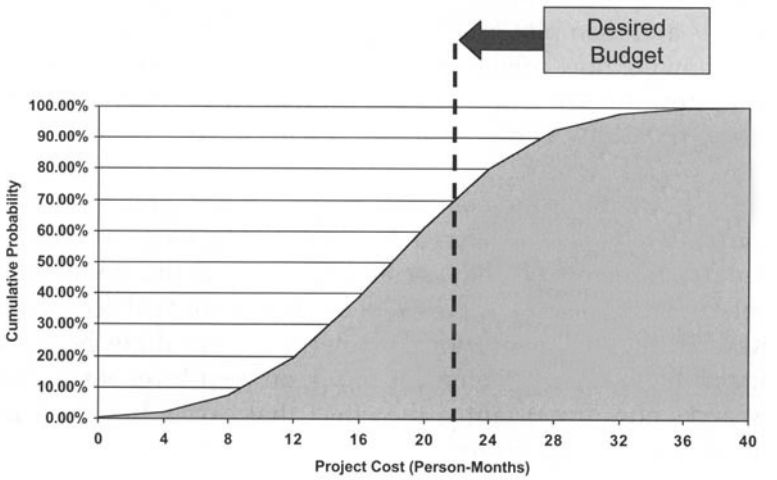


Figure 3-19: Imbalance of Figure 3-16 Fixed.

Next, examine the schedule-risk graph. If the graph looks like that in Figure 3-17, your project has an 83-percent likelihood of completing these requirements by the desired delivery date. Just as we asked of the cost-risk graph earlier, ask, Is this an acceptable level of risk for your project? Again, if the likelihood is 80 percent or more, most organizations will accept it. Below that number, you must decide on your level of risk adversity and, even more importantly, the effect that delivering the project late will have on the organization:

- Will the product still be useful?
- What are the financial implications (revenues if the product is to be sold, excessive expenses if the product is designed to reduce costs, and so on)?
- Will the project be cancelled?
- What will happen to the careers of the project members?

As before, with cost risk, if the total picture for schedule risk is positive (or at least acceptable), then you are done with triage. If the total picture is negative, you have only two alternatives: Remove requirements or extend the delivery date. It is that simple.

When removing requirements, be sure to remain cognizant of the four considerations described above. When extending the date, you will need to know how far to extend it. To decide this, just

look at the graph in Figure 3-17. Look at the horizontal distance between the current position of the vertical bar (the current desired delivery date) and the desired position of the vertical bar (that date where the risk is acceptable). You can see that in order to raise the likelihood of success from the current 83 percent to, say, 97 percent, you will need to extend the delivery date by around one month, as shown in Figure 3-20.

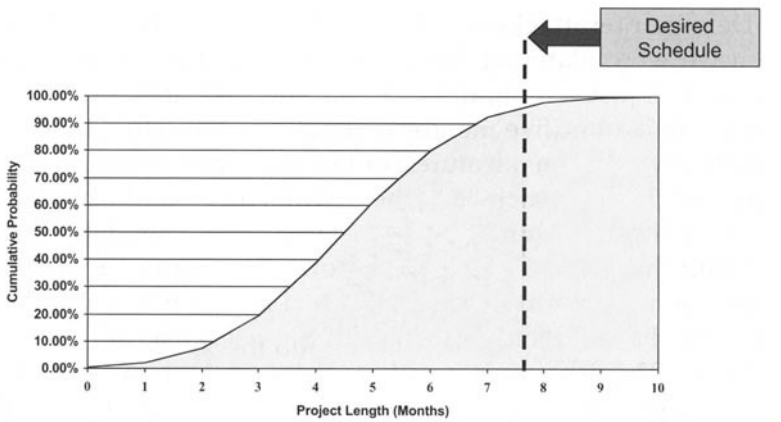


Figure 3-20: Imbalance of Figure 3-17 Fixed.

In practice, triage is rarely practiced as a sequential series of adaptations that slowly converge toward an optimal solution. Instead, progress toward an optimal solution is made in spurts.

In October 2000, I was consulting for a large manufacturer of mass storage devices. The product manager had called me in to resolve a problem the company was experiencing: Mark, the marketing manager (not his real name), was demanding that the next release of the product, Version 3.0, be delivered to the customers in nine months. Meanwhile, Dev, the software development manager (also not his real name), was insisting that such a date was impossible to achieve. They had reached an impasse when I was called in. I asked each party to describe his position. Here is what I heard:

Mark: "Look, the window of opportunity starts in nine months. We know that the competition is planning to release similar products ten to twelve months from now. Since their products and our 3.0 release are so similar, the only way we are going to be successful is if we are the first to market."

Dev: "I understand what you are saying. But I also know what is realistic. Just wishing for something is not going to make it happen. Read my lips: My team cannot produce all the features you want in Release 3.0 in nine months. It simply cannot be done."

Mark decided to try appealing to Dev's corporate allegiance: "Don't you realize that you'll be letting down the entire company if you don't build it when it's needed?"

I saw where this was headed, and it was not good.

Dev counterattacked: "Look, Mark, remember a year ago when we were planning the 2.4 release? You demanded that I deliver it in just five months. I told you then that we had two choices: Build it in five months without an architecture that could support any additional features, or build it 'right' and deliver it in eight months, in which case the architecture would be able to handle additional requirements. *You* chose the first option. So, it is *your* fault that we are in this mess now! The current architecture simply cannot support the 3.0 features. I need a full year in order to revamp the architecture and then add the new features."

Ouch! The debate continued for about an hour and they made little progress toward a mutually agreeable solution. During that time, I presented the schedule-probability graph (as shown in Figure 3-21) to the team. It was clear that Dev was not lying; the graph reported a meager 37-percent chance of success.

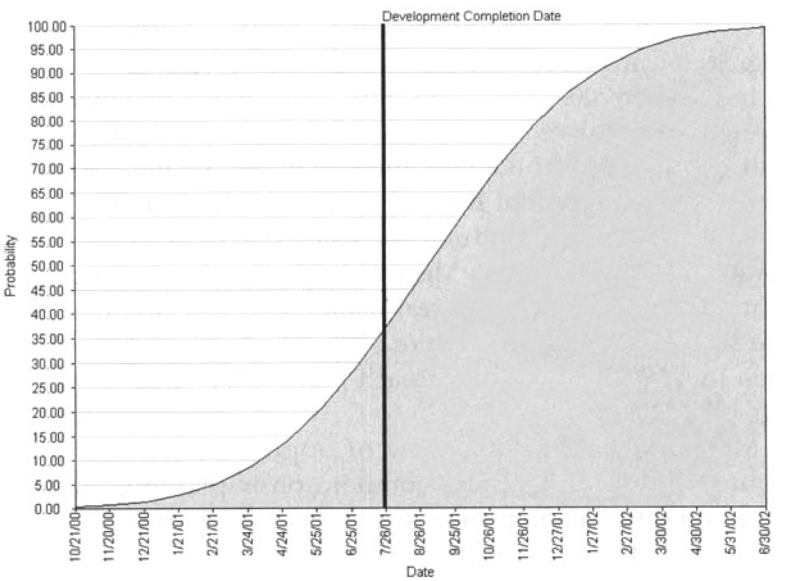


Figure 3-21: Case Study—Schedule-Probability Graph.

We took a short break, during which I took Dev aside and asked him to imagine that he owned a majority of the company's stock, and that the project's success or failure in the marketplace was key to his personal financial success. Clearly, promising to deliver something in nine months when it would actually take twelve is not productive. Nor is delivering it in a year to a stale market. I asked Dev what he would do with the project. He answered, "Interesting situation—here's what I would do. I'd increase my headcount so I could afford to staff two parallel development teams. One team would work on incorporating as many of the 3.0 features as possible into the old architecture. We could release it seven-to-eight months from now. We could call it Release 2.5. The other team would start revamping the architecture and be ready to deliver the full 3.0 capability in a year."

I told Dev I liked his suggestion and wanted him to present it to the assembled team after the break. He said, "Okay, but there's no way Mark is going to find it acceptable."

After we reconvened, Dev made his suggestion. Mark's response? "Wow! Would you really do that for me?" We had finally reached a tentative agreement in principle. We worked a few more hours to see if it was realistic. First, we checked to see if the twelve-month schedule was really reasonable for the 3.0 release. All we did was move the vertical line shown in Figure 3-21 to the right by three months, which resulted in Figure 3-22.

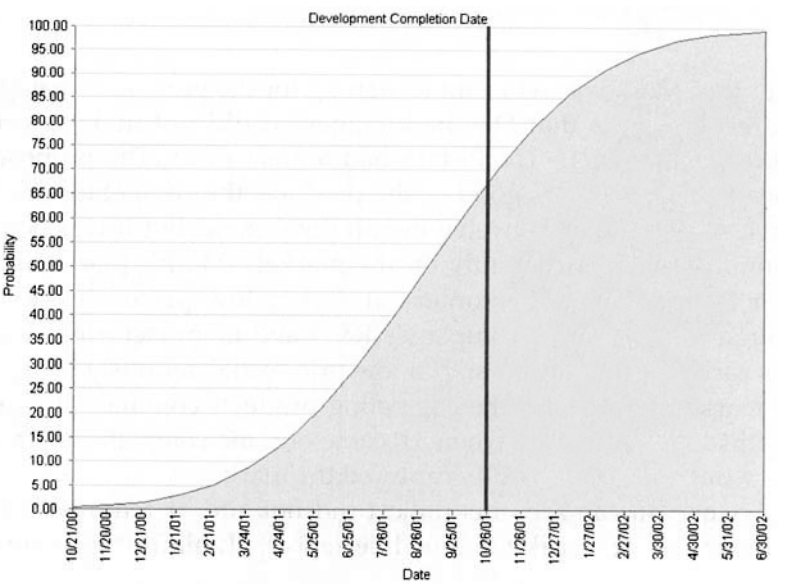


Figure 3-22: Case Study—Schedule-Probability Graph for Proposed 3.0 Release.

As you can see, the likelihood of success moved from 37 percent to 67 percent. But the real test came next. Was Release 2.5 possible? We allowed the development team to select the subset of requirements it was able to include with the old architecture. Figure 3-23 was the result. Once again, the graph indicated an acceptable level of risk, an 82-percent chance of success.

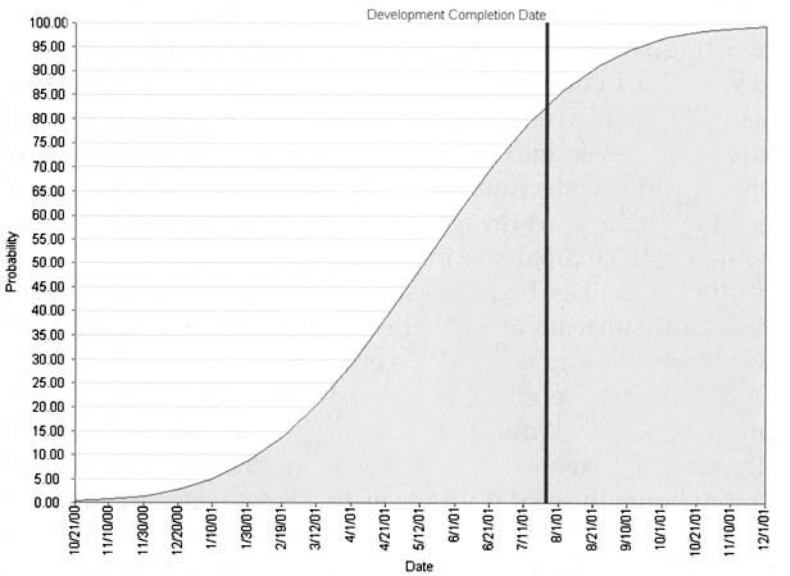


Figure 3-23: Case Study—Schedule-Probability Graph for Proposed 2.5 Release.

After this, Mark was reluctant to sign up for the proposed strategy; the requirements that Dev had suggested did not make a very impressive product. Then Mark had a great idea: The proposed Release 2.5 was not as good as the products the competitors were to release in the ten-to-twelve-month time frame, but it was better than anything else currently on the market. Mark's plan was to offer Release 2.5 to customers at a very low price—not even enough to recover the company's R&D and manufacturing costs. This early introduction at such a low price could seriously dampen the market demand for the competing products coming out a few months later. And then when 3.0 came out, the competitors' products would not have already captured the market.

So, as you can see, this project did not add or remove single requirements or tweak the schedule repeatedly until it converged

upon a viable solution. Instead, it was the result of out-of-the-box thinking to envision two releases, followed by equally innovative thinking concerning how the product could be marketed and priced relative to the competition. This is often the case.

Delivery Date: Talking Apples and Apples

When parties agree to a delivery date, make sure they are talking about the same thing. For example, development organizations often think of "delivery" as the date they are no longer responsible for the product. In some companies, this could mean releasing the product to the independent testing team within the company. But in most companies, marketing thinks of delivery date as the date that the company can ship the product to the first buyer. Here is a list of some of the events that various people think of as "delivery":

- release to test
- release to quality assurance
- release to manufacturing
- release to (tactical) marketing
- release to sales
- release to beta customer
- release to revenue customer

Depending on the complexity of the product, these dates could be many months apart. It really does not matter which delivery date the parties are talking and negotiating about, as long as they are all talking about the same date.

Advanced Triage Techniques

The triage process described in the previous section balances delivery date and development budget against desired requirements. However, in a real business environment, other factors need to be considered. This section describes how to perform triage when considering the many other factors that can influence the packaging of requirements into releases. These factors include

- risks inherent in addressing specific requirements
- market size
- market window

- market penetration
- price
- costs
- revenue
- return on investment

Advanced triage can be thought of as balancing a multi-person seesaw. Obviously, manipulating any of the variables could have dramatic effects on the other variables. For example, adding a few extremely unique requirements might allow us to sell the product at a much higher price and accept a smaller market penetration, and still achieve desired revenue and profitability goals. We will now discuss each of the above factors and describe how it affects the requirements triage process.

Considering Risks Inherent in Addressing Specific Requirements

Even though the size estimates for two requirements may be identical—say, ten person-weeks—one requirement may have some inherent risks that the other does not. For example, we may already have the right skills on board for one, but the other may require us to hire some people. Or, in one case, satisfaction of one requirement may depend heavily on a subcontractor delivering a subcomponent on time, but the other does not.

To capture such differences, annotate individual requirements with the inherent risk associated with their successful satisfaction. This can be captured most easily as a percentage representing the likelihood that the situation will go awry. A requirement having an associated risk of 25 percent means that there is a 25-percent chance that we will fail to satisfy the requirement, even after expending the specified number of resources, as illustrated in Figure 3-24.

Although not shown in Figure 3-24, a comment describing the source of risk is a good addition. Requirements triage demands that not only must the probabilities of completion on schedule and within budget (as shown previously in Figure 3-19 and Figure 3-20) be acceptable, but also that the requirements being considered for inclusion must exhibit acceptable levels of inherent risk.

Reqt. No.	Requirement Text	Risk
1	The system shall be programmable by the operator.	10%
1.1	The system shall be programmable by the operator to set the default for the green direction to be "East" or "West."	10%
1.2	The system shall be programmable by the operator to set the maximum duration for the light to remain green in the non-default direction.	10%
1.3	The system shall be programmable by the operator to set the minimum duration for the light to remain green in the default direction.	10%
1.4	The system shall be programmable by the operator to set the duration of the amber light prior to it changing to red.	10%
2	The system shall provide safe access to a one-lane east-west bridge via green/amber/red traffic lights.	30%
2.1	Two sets of traffic lights shall be controlled by the system.	10%
2.2	When either set of lights is "green," the other set of lights shall be set to "red."	10%
2.3	When the system determines that it is time to switch the direction of traffic, it shall do so in a safe manner.	30%
3	The system shall control eastbound traffic coming from northwest and southwest converging roads.	40%
3.1	During the period while the eastbound traffic light is authorized to be green, the system shall provide equal time for the traffic coming from the southwest and the northwest.	15%
4	The system shall control westbound traffic coming from northeast and southeast converging roads.	40%
4.1	During the period while the westbound traffic light is authorized to be green, the system shall provide equal time for the traffic coming from the southeast and the northeast.	15%
5	The system shall interface to vehicle sensors capable of determining if there is a vehicle waiting at either of the two entrances.	10%
6	The system shall interface to vehicle counters capable of counting vehicles as they pass through each of the two entrances.	60%
7	The system shall sense the weight of vehicles on the road and not allow either light to turn to green while a vehicle remains on the bridge.	75%
8	If a vehicle is disabled on the bridge, the system shall automatically contact a tow truck.	80%

Figure 3-24: A List of Candidate Requirements Annotated by Inherent Risk.

The easiest way to visualize the inherent risk associated with a proposed release is to examine a histogram showing the distribution of the requirements currently being considered for inclusion, as shown in Figure 3-25.

In this case, 59 percent of the requirements being considered for inclusion (10 out of 17) have a 19 percent or lower risk; 18 percent (3 out of 17) have a 60 percent or greater risk, and the rest lie in between. The shape of an acceptable risk histogram varies widely

from project to project and company to company for two reasons: First, some companies or projects thrive on risk, while others are risk-averse; second, members of different projects are likely to calibrate risks associated with requirements in different ways.

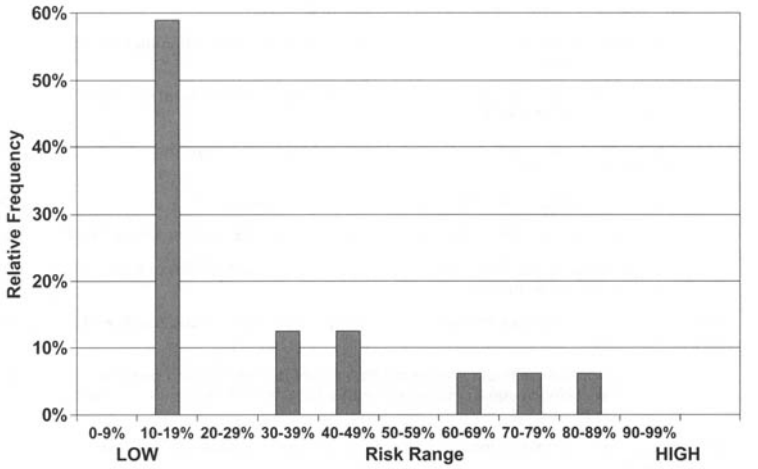


Figure 3-25: Requirements Risk Histogram.

While you're performing triage, if you find that the risk histogram indicates unacceptably high levels of risk, consider one or more of the following actions:

- Move some of the riskier requirements out of the baseline and defer them to a later release.
- Consider removing a high-risk requirement while adding one or more lower-risk requirements.
- Examine the requirements that represent unacceptable levels of risk. I have seen many cases in which the development group labeled a requirement high-risk because of some aspect of that requirement that the customer didn't even care about. To uncover this situation, refine high-risk requirements into sets of simpler requirements (as described in Chapter 1). Now, reanalyze the relative priorities and risks of the children requirements. You may discover that you can include high-priority, low-risk children requirements and exclude other lower-priority, high-risk children requirements.

Discuss the high-risk requirement within your group. Focus on what makes it high risk. Explore alternative solutions to satisfying a requirement.

Considering Market and Market Size

A *market* is a group of people with an unresolved need and sufficient resources to apply to the satisfaction of that need. For a company that sells its software externally, the market is the set of all potential customers. For an internal IT organization whose mission is to address the information-processing needs of its revenue-producing sibling organizations, the market is the set of individuals within the company who will eventually use the IT system in order to improve its ability to produce revenue, reduce overhead, increase profits, and so on.

An organization that is trying to define its market has a great deal of flexibility. For example, let us say we manufacture cherry soda. We could define the market as all humans on earth. Our justification would be that every human, in theory, could buy our soda. If we defined our market thusly, we would discover that the market size is huge, but that we could only succeed in selling to a minuscule fraction of our market. On the other hand, if we defined our market as only those people who have ever purchased a cherry soda, we would have a relatively small market size but would experience a much better record of selling to that market.

If there were a "right" approach, it would be to narrow your market definition to the subset of the population that you and your competitors are seriously trying to capture. Thus, as a cherry soda manufacturer, we should probably define our market as the people who purchase any type of sweet, carbonated beverage.

Your market may change as you add or remove features. For example, if you manufacture laser printers and you need to decide to include color printing in your next product (or not to), your market (and its size) will change.

Do not try to define the changes to the market size as a function of the addition or removal of individual requirements. It is more productive to understand and record major market changes as they relate to the addition or removal of large subsets of requirements. A simple example will suffice: Let's say you are in the United States and are building your first software product. You are

wrestling with a long set of potential features (requirements). Twenty of them relate to various aspects of the internationalization of the product, and ten of these relate in particular to the software's ability to handle multi-byte characters. By omitting all twenty features, the market is limited to just the United States; by including the ten requirements that do not relate to multi-byte characters, the market is expanded to include roughly half of the world; and by including all twenty of the internationalization requirements, the market expands to include most of the world.

As you perform requirements triage, you need to remain cognizant of how adding or deleting requirements affects your market size. Otherwise, you may delete some requirements in order to make the market window, only to discover that you now have no market. As a general rule, the more features you add to the product, the larger the market, as shown in Figure 3-26. Of course, there are many exceptions to this usual pattern. For example, in some markets, the simplest and most basic product may have the largest market. Furthermore, as software products become more and more overloaded with functionality, they often become cumbersome and fewer people want to buy them.

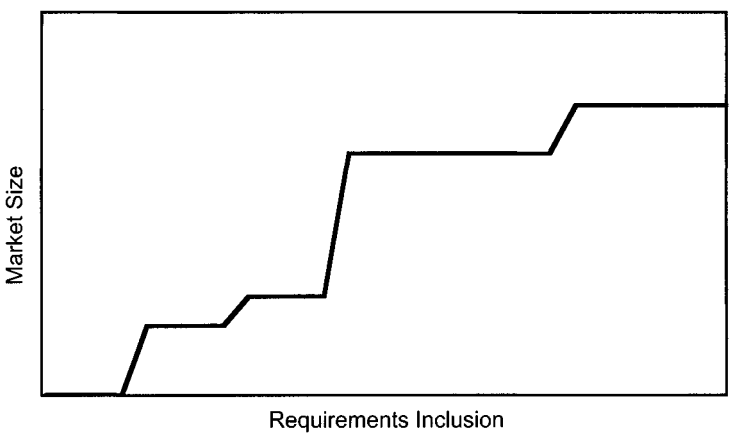


Figure 3-26: Usual Relationship Between Requirements Inclusion and Market Size.

Considering the Market Window

The *market window* is the period of time during which customers will buy a product. Extending this concept to an organization that builds a software system for internal use, the market window is the

period of time during which the company can best utilize the software system. For the current discussion, we will assume that the earlier in the market window you introduce a product, the more successful the product sales will be.

However, the reader should be aware that this is not always the case. In a landmark study at Iowa State [IOW57], researchers found that as a market window progresses, different types of buyers become active (see Figure 3-27). Buyers in the early phases (innovators and early adopters) are more likely to be impulse buyers and to buy something just because it is new. On the other hand, buyers in the later phases (early majority and late majority) are more likely to make careful buying decisions. And those in even later phases (laggards) will not buy until the product is well proven and extremely mature. In addition, Donald Reinertsen [REI97] points out that in some types of markets, it may be better to be late than early because you spend less money educating the market about the product. For systems designed for internal use, though, this is rarely, if ever, the case.

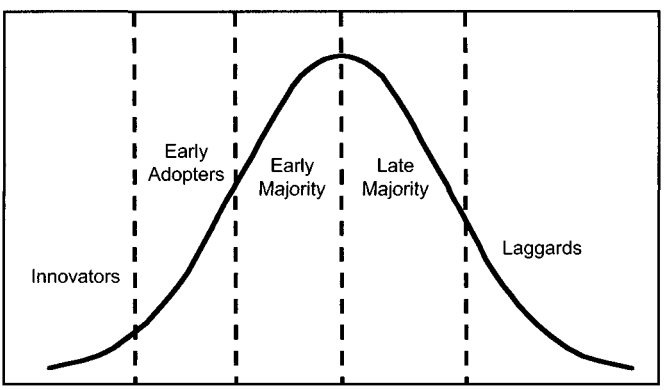


*Figure 3-27: Market Window (Adapted from [IOW57]).**

Marketing departments are acutely aware of market windows, and often work hard to target a product's delivery to the right place in that window. However, as features are added to a product, the following may occur:

- The delivery date is generally delayed, so the product enters the window at a later stage.

**The Diffusion Process,* Agriculture Extension Service, Iowa State University, Special Report No. 18 (Ames, Iowa: 1957). Used by permission.

- The inclusion of those features may change the market window (in other words, the market window for one set of features is different than for another), shifting it either forward or backward. For example, adding a specific feature may make sense only to innovators or early adopters, so adding that feature but delaying its introduction to the market makes no business sense. Or, adding a specific feature may make sense only to early and late majorities, so delaying delivery to add that feature may make business sense.

As features are removed from a product, the following may occur:

- The delivery date is generally contracted, and the product enters the window at an earlier stage. Even though the product may be feature-poor, it may be attractive to innovators and early adopters because it is so early (assuming it does have the right kinds of features for these buyers).
- The removal of features may change the market window (again, the market window for one set of features is different than for another), shifting it either forward or backward.

Considering Market Penetration

It is one thing to hit a market window at the desired time, but it is another to successfully capture that market. *Market penetration* is the percentage of the market that you have sold to. The amount of market penetration your product achieves depends on the features selected for inclusion, your pricing, the softness of the market, the features of your competitors' products, and your competitors' pricing. For internal IT projects, market penetration is the percentage of relevant company transactions that the company executes using the new system.

Market penetration will be a function of time. Figure 3-28 shows a simplified model of how you could specify the percentage of the market that you expect to successfully sell your product to, given a specific set of features.

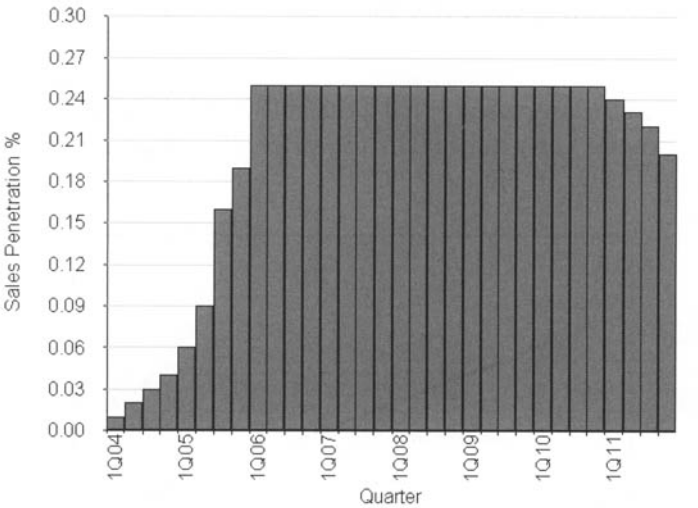


Figure 3-28: Market Penetration Graph.

Considering Price

Price is the amount a customer is charged for one or more copies of the product. Pricing strategies often include multiple tiers, quantity discounts, different pricing for different kinds of customers, partner-agreement-based pricing, and so on. There is no real equivalent of price for internal IT organizations, unless you want to consider it the effort expended by the internal customers.

Often, the marketing organization conceives of a particular feature mix in the next product release based on a long list of assumptions, many of which are not documented. In fact, many may be tacit and may have never even been expressed. One such assumption may be the price of the product. When marketing makes a statement such as, "The customer absolutely needs to have requirement *x* satisfied," it is actually saying, "The customer absolutely needs to have requirement *x* satisfied *if* we are going to charge *this* price." A company desiring to produce the best possible achievable product needs to consider the trade-off between feature mix, price, and timing. Furthermore, the "perfect" product—one that has all the right features and is released on budget and within schedule—is an utter failure if it cannot be sold at the desired price.

When selecting the price for software products, many factors need to be considered:

- How many units will you sell at a given price? As a general rule, the higher the price, the smaller the volume, as shown in Figure 3-29.

Figure 3-29: Price and Units Sold.

- The higher the price, the higher your revenues per unit will be. As a general rule, a curve like that shown in Figure 3-30 is applicable. That is, if you charge zero dollars for your product, you will have no revenues, but as you increase your price, revenues increase. This continues until your price becomes too high, at which point sales decrease significantly enough that your total revenues decrease.

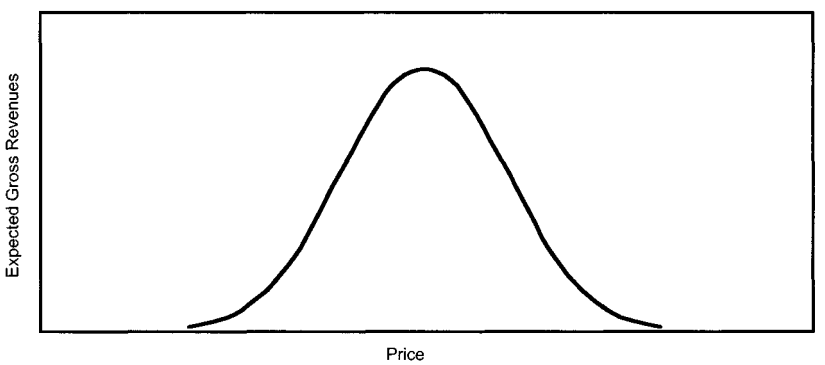


Figure 3-30: Price Modeling View.

- Unlike the pricing of manufactured products, the pricing of software is not related to the cost of raw materials or manufacturing. In software, these are both close to zero.
- To some degree, users may perceive the quality of a higher-priced product to be greater than a similar but lower-priced product.
- The higher the price, the higher your margins (profit per unit sold) will be.
- You may want to maximize market penetration and not revenue, in which case you might offer your products earlier and at a lower price than the optimal point shown at the peak of the curve in Figure 3-30.

Here are some questions to ask yourself regarding pricing:

- For each market segment, what is the expected average price that customers will pay per unit? Or, for each market segment, what is the expected average order size (in currency and number of units)?
- What special discounts will apply? For volume orders? For special customers? For promotions?
- Will you offer elasticity from advertised prices? For example, I know two companies in a particular marketplace. One advertises a price twice as high as the other, but they both experience the same gross revenues per unit sold. The reason is that one company allows its sales force to offer "special" discounts to every customer. Both have a unique strategy, and both work. The company that offers discounts makes every customer feel special by offering them those steep discounts, while the other gets its foot in more doors by advertising a lower price.
- How will you discount the product for resellers?
- Will you sell the software at a relatively low price, and offer customization services, which will become your major source of revenue? Or will you sell the software at a higher price, and allow third parties to do the customization?
- Will you offer the software to customers on a per-use basis rather than a per-site or per-copy basis? The most common way to do this is to offer it as an application service provider (ASP): Users pay for access to the software over the Web.

For software products destined for internal use, rather than external sale, there is no real price, per se, as I mentioned above. Instead, "benefit" should be analyzed as part of triage. Software for internal use is typically developed to assist other parts of the company in performing some business function. Typical motivations are to reduce the cost of doing business, reduce errors, or collect additional revenue. The questions to ask when analyzing the benefit of the software are as follows:

- What will the average cost savings per transaction be?
- What percentage of errors is expected to be eliminated?
- What is the average cost to the company of each error?
- What additional revenues are expected as a result of introducing the product?

In summary, during the triage process, price must be considered. Adding extra features may enable you to increase the price and compensate for a slightly later delivery. Removing some features may enable you to deliver early enough that you can afford to charge less and perhaps capture the market before the competition.

Considering Costs

Labor costs incurred by a company in creating a new software system are usually nontrivial. According to generally accepted accounting principles (GAAP), *costs* are usually expensed in the year they are incurred and thus have a direct and immediate impact on both cash and profitability.[18]

When considering the construction of a new software system, the impact on the company's cash and profitability must of course be considered, but it is also useful to examine the effect of costs on whether it makes sense to build the product at all. For example, when a company spends x dollars developing a product, it takes some time to recover those costs through sales (for companies that sell the products) or through use (for companies that deploy the product internally).

Recovery is usually measured in units sold (or elapsed time or transactions processed). Let's assume that recovery is measured in terms of y units sold. Each time a unit is sold, we are, in effect,

[18]In some cases, these costs may be capitalized, in which case they have an immediate negative effect on the company's cash flow but no immediate effect on profit. The costs are then depreciated over the software's useful life.

recapturing x/y of the original outlay of cash. And after selling y units, we have recovered all the original x dollars (ignoring, of course, the time value of money—see the net present value discussion later in this chapter).

As more and more features are added to a product,

a. Development costs increase.
b. Because of the increased costs, we will need to sell more units to recover the costs.
c. Because we need to sell more units, the time required for recovery will increase.
d. Product delivery may be delayed.
e. Due to the delay, once again, the time required for recovery will increase.

On the other hand,

f. Adding features may increase the rate at which units are sold.
g. This would lead to a decrease in the time required for recovery.

When features are removed from a product,[19]

a. Development costs generally decrease.
b. Because of the decreased costs, we will need to sell fewer units to recover the costs.
c. Because we don't need to sell as many units, the time required for recovery will decrease.
d. Product delivery may be accelerated.
e. Due to earlier delivery, the time required for recovery will decrease.

On the other hand,

f. Removing features may decrease the rate at which units are sold.
g. Due to the slower rate of sale, the time required for recovery would increase.

[19]Although items *a* and *d* seem logical, once development begins, the removal of a feature could actually increase costs and delay delivery because of the extra effort to undo something that has already been done!

As you can see, c and e are opposing forces to g in both cases. Finding an optimal set of features is nontrivial.

Although most companies will not compute all these items precisely, it is important to understand the general effect of adding features to a product or removing them. The bottom line is that adding features to a proposed product will definitely increase development cost, but it may increase or decrease the time for recovery. On the other hand, removing features from a proposed product will generally decrease development cost, but it may increase or decrease the time for recovery. Finally, in software development, there are very few recurring costs (such as the cost of manufacturing or of goods sold, as in the case of material goods).

Considering Revenues

The *revenue* associated with a product is the sum of all the gross receipts related to the sale of the product. Given the earlier definitions of market size, market penetration, and price, we could also define revenue (at least conceptually) as

$$\text{revenue} \approx \text{market size} \times \text{market penetration} \times \text{price}.$$

That is, as market size, market penetration, and price increase, so does revenue. However, as pointed out in previous sections, the three variables on the right side of the above equation are not independent. Changing the market so that market size increases will lower penetration, and changing the price can have dramatic effects on market penetration.

Given the following, we should be able to predict expected revenue as a function of time: a fixed market segment (as discussed earlier, in the subsection entitled "Considering Market and Market Size"), a set of features (as shown in Figure 3-13), and our expected market penetration (as shown in Figure 3-28).

Considering the Effect of Investment

Each of the above factors should be considered carefully when deciding to include or exclude a requirement from a software product. However, the ultimate indicator of whether a feature should be included or excluded is how it contributes to return on

investment (ROI). In general, *return on investment* is defined as a measure of how effectively the company is using its capital to generate profits. There are many ways to calculate a return on investment:

- *Accounting definition:* The official definition of ROI is the annual income (profit) divided by the sum of shareholder's equity and long-term debt.
- *Annualized percentage rate (APR):* If you invest, say, $100 this year, and in five years it has increased in value to $128, you have received an overall rate of return of 28 percent. But to be able to compare two different potential investments, you want to look at the annualized rate of return. In this case, the annualized rate of return is 5 percent ($100 x 1.05^5). That is, if you invest $100, and every year thereafter you earn 5 percent on your money, at the end of five years, you will have $128. Now, let's say you have the choice between the following two potential ways to invest $100:
  - $100 today will become $128 in 5 years.
  - $100 today will become $115 in 1 year.

  If you want to know which is the better option, just compare their APRs. In the first case, the APR is 5 percent. In the second case, the APR is 15 percent. The second is a better investment even though it has a smaller total gain.
- *Internal rate of return (IRR):* IRR is quite similar to APR, but is expressed as a multiple, rather than a percentage. So, a 5-percent APR is expressed as a 1.05 IRR, a 15-percent APR is expressed as a 1.15 IRR. If you invest $100 today and it becomes $1,600 in four years, that represents an IRR of 2; in other words, you have effectively doubled your money every year for four years. Spreadsheets take away the difficulty of computing IRR by providing built-in formulae for this.
- *Break-even:* At the break-even point, your cumulative revenues and your cumulative costs are equal. In a typical software development effort, you incur significant up-front costs of product development, as well as the ongoing costs of maintenance and upgrades to the software (see Figure 3-31 for a graph of these cumulative costs).

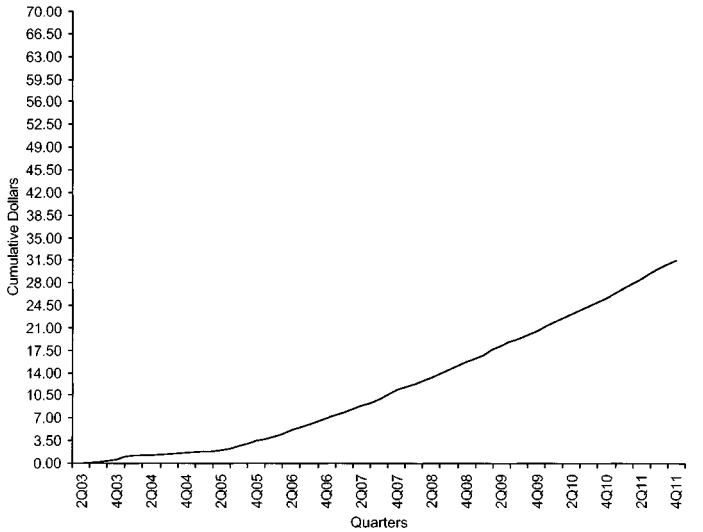


Figure 3-31: Cumulative Costs.

Meanwhile, if you are selling the software externally, you are receiving revenue from your customers for both initial purchases and annual maintenance contracts. Or, if you are using the software internally, you are reaping some reward—some combination of increased revenues, decreased costs, or increased efficiency (see Figure 3-32).

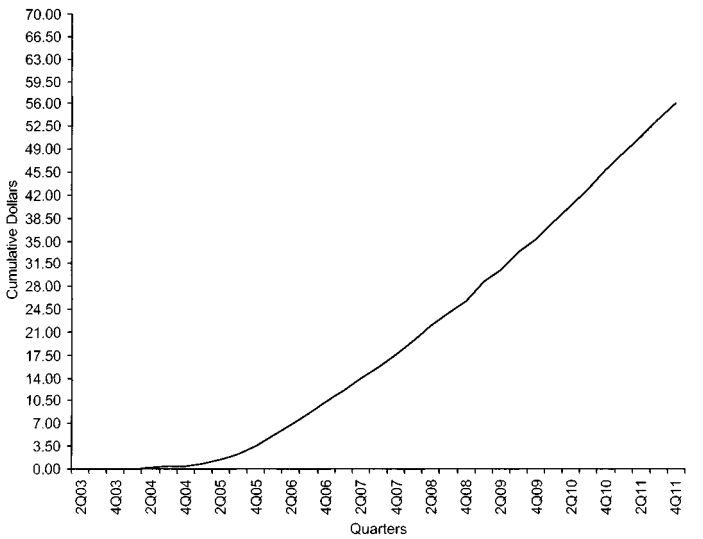


Figure 3-32: Cumulative Benefits (Revenues or Savings).

Considering that revenues and other benefits generally do not start until well after the software development stage, it should be no surprise that the curve of Figure 3-31 shows non-zero amounts (by the third quarter of 2003) earlier than they are shown in Figure 3-32 (by the third quarter of 2004). The break-even point is found by overlaying the two graphs, as shown in Figure 3-33.

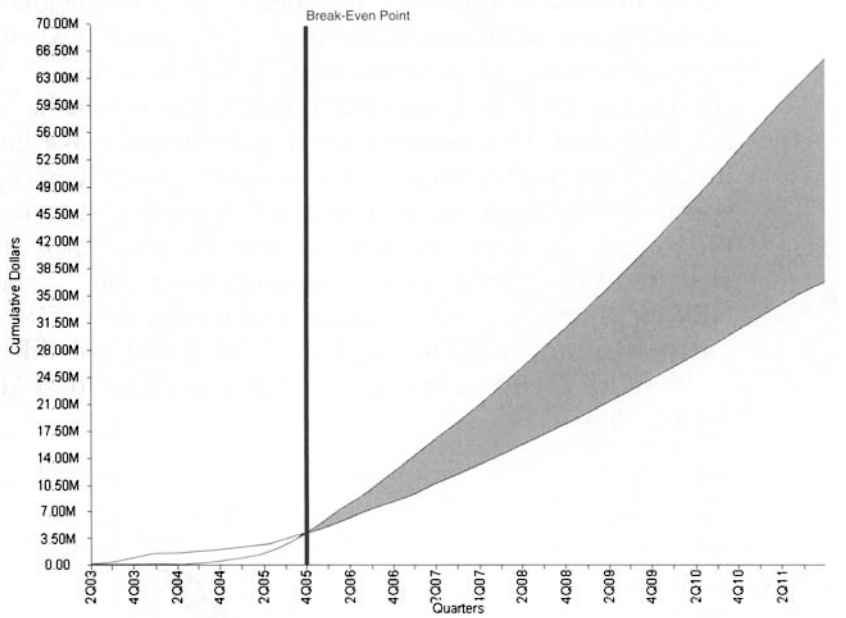


Figure 3-33: Break-Even Graph.

The shaded areas between the two graphs represent where the benefits exceed the expenses. However, be aware that on *real* software projects, there is a point of diminishing returns, when the cumulative expenses (being fed by escalating maintenance costs) once again exceed the cumulative benefits, as shown in Figure 3-34. You will note in this figure how the area turns from "in the red" (unshaded) initially, to being shaded, and then the lines cross again and the area is unshaded ("in the red") once again.

- *Net present value (NPV)*: Net present value is used to compare the return on investment of a candidate product (with selected features) with other potential investments or alternative feature sets. It is the net financial result of a multi-year investment expressed using the current dollar valuation.

Let's look at a few parts of this definition. "Net" means that we are looking at the final effect of all outflows (the investments made in R&D for building the software product, as well as the ongoing maintenance) and all inflows (all types of revenues resulting from sales of the product and maintenance contracts). "Financial result" indicates that NPV will be expressed in monetary units. "Current dollar valuation" indicates that NPV considers the time-value of money (for example, $1.00 today is worth more than $1.00 will be worth next year). The formula for computing NPV factors in the interest you would have earned (called the discount rate) if you did not make the investment currently under consideration. Like IRR, every spreadsheet today provides a built-in formula for NPV; just plug in the discount rate, inflows, and outflows for each year, and the spreadsheet will compute it for you. If the NPV for your new product is higher than some other alternative you are considering, then you might consider building it. If NPV is less, it is probably not wise to make the investment.

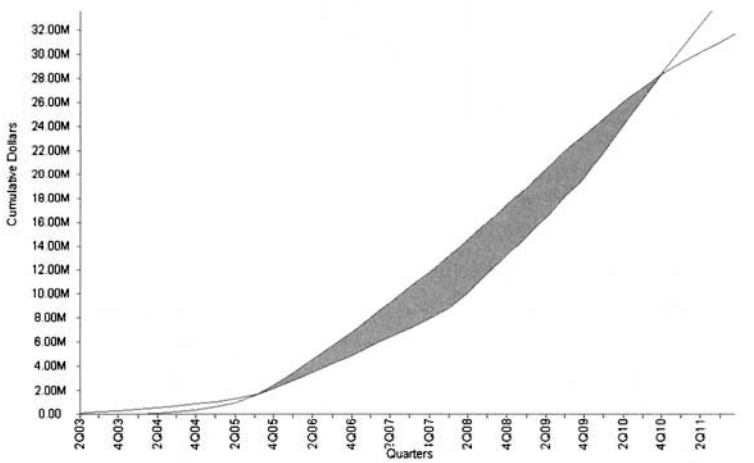


Figure 3-34: A More Realistic Break-Even Graph.

Putting It All Together

The balancing act that is triage is nontrivial, but quite doable. Figure 3-35 shows how each of the factors influence each other, all

driven by the decision to add or subtract features from the next release. This figure highlights the conflicting facts. For example,

- As you add features, costs increase, which delays break-even.
- As you add features, the price may increase, resulting in higher revenues, which accelerates break-even.

Or consider these facts:

- As you add features, delivery is delayed, decreasing volume and delaying break-even.
- As you add features, market size may increase, increasing volume and accelerating break-even.

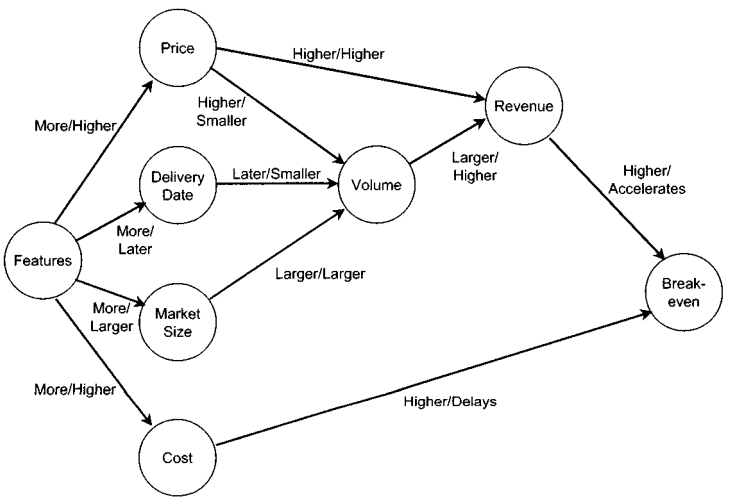


Figure 3-35: Complexity of Advanced Triage Factors.

The Result of Triage

Triage is completed when the organization has determined which subset of requirements will be satisfied in the next release of the product. If we're conducting basic triage, then we know everything shown in Figure 3-36. If we're conducting advanced triage, we know everything shown in Figure 3-37. In both of these figures, the first item in each column is usually documented in a preliminary version of the requirements document, to be expanded in the subsequent requirements specification phase. All other

items are likely to be documented in a business case document [REI03], preliminary project plan [IEE98a], or preliminary marketing plan [BER01].

For Products to Be Sold	For Products for Internal Use
• *which* requirements we will strive to satisfy • *when* the product will be delivered, and with what likelihood • *how much* development is expected to cost, and with what likelihood	• *which* requirements we will strive to satisfy • *when* the product will be delivered, and with what likelihood • *how much* development is expected to cost, and with what likelihood

Figure 3-36: What We Know at the End of Triage (Basic).

THE SECRETS OF *JUST ENOUGH* TRIAGE

If you pay too little attention to triage, you run the risk of trying to build a system that cannot be built within your time and resource constraints. If you spend too much time on triage, priorities will change before you even start doing requirements specification, resulting in a never-ending delay to project initiation. The secrets of accomplishing *just enough* requirements triage are as follows:

- Learn to accept that there is no such thing as a perfect solution to the triage dilemma. Compromise is necessary.
- Always annotate your candidate requirements with a relative priority and an estimated cost.
- Record interdependencies between requirements.
- Plan more than one release at a time [DAV03].
- Plan to replan before each new release [DAV03].
- If the voices (usually from marketing) crying "Add more functionality!" are allowed to overcome the voices of moderation, late delivery is guaranteed.
- If the voices (usually from development) crying "We can't implement that much functionality!" are allowed to overcome the voices of moderation, a weak product is guaranteed.
- Never lose sight of your goal: to select a subset of the full set of desired requirements so that the product can be delivered on time and within budget.

For Products to Be Sold	For Products for Internal Use
• *which* requirements we will strive to satisfy	• *which* requirements we will strive to satisfy
• *when* the product will be delivered, and with what likelihood	• *when* the product will be delivered, and with what likelihood
• *how much* development is expected to cost, and with what likelihood	• *how much* development is expected to cost, and with what likelihood
• *who* the customers are	• *who* the internal customers are
• *how much* the customer is expected to pay for the product	• *how much* savings or additional revenues the company will incur
• *how much* it will cost to market and sell	• *how much* it will cost to implement
• *how many* units are expected to be sold within the next *x* years	• *how many* transactions are expected to be performed by time period over the next *x* years
• *when* the product is expected to be replaced (the product's expected life span)	• *when* the product is expected to be replaced (the product's expected life span)
• *when* the company will recover its costs (the break-even)	• *when* the company will recover its costs (the break-even)
• *what* the internal rate of return is	• *what* the internal rate of return is

Figure 3-37: What We Know at the End of Triage (Advanced).[20]

- Triage participants must see themselves as a team trying to solve a business problem, not as separate camps trying to get their own way.
- Development should avoid making absolute statements such as, "We cannot build the system by the delivery date if you add that requirement." Instead, make statements such as, "By adding that requirement, our likelihood of delivering on time reduces from 73 percent to 27 percent." This helps create an environment of teamwork since *nobody* on the team wants to deliver the product late.

[20]I use the term "implement" in the right-hand column, in the information systems sense, which means, "making it happen in the organization" and includes such costs as training, deployment, purchasing equipment, changing business processes, and so on. I am not using it in the computer-science sense, meaning "code the software."

- Marketing (and customers) should avoid making absolute statements, such as, "We cannot sell (or use) the system if that requirement is excluded." Instead, make statements such as, "By removing that requirement, our expected revenues will be reduced from $20M to $11M." This helps create an environment of teamwork since *nobody* on the team wants to hurt the company's revenue.
- Agreeing to a set of requirements that are impossible to satisfy in the given time guarantees failure. Why would anybody send a company down such a path?
- Avoiding explicit triage altogether means that triage issues will be addressed through intimidation and politics, which will doom the project to failure.
- Stakeholders have the right to change their mind. A requirement that is not important today may become critical tomorrow, and vice versa. Be flexible.

Index