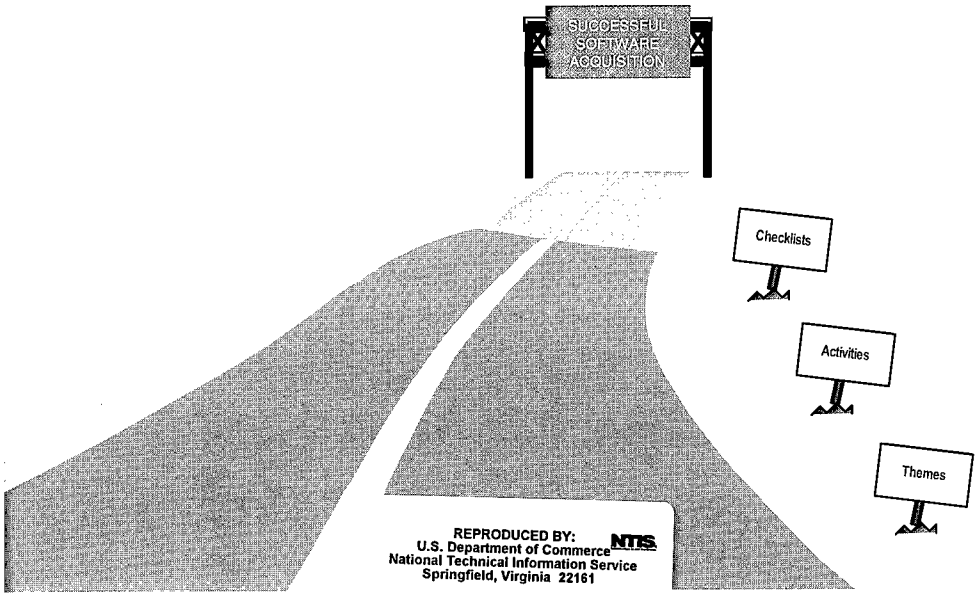
PB99-117061

U.S. Department
of Transportation
Federal Highway
Administration

The Road to Successful ITS Software Acquisition

Volume II: Software Acquisition Process Reference Guide



REPRODUCED BY:
U.S. Department of Commerce
National Technical Information Service
Springfield, Virginia 22161
NTIS

Notice

This document is disseminated under the sponsorship of the Department of Transportation in the interest of information exchange. The United States Government assumes no liability for its contents or use thereof.

1. Report No. FHWA-JPO-98-036	2. Government Accession No.	3. Recipient's Catalog No.
4. Title and Subtitle The Road to Successful ITS Software Acquisition Volume II: Software Acquisition Process Reference Guide		5. Report Date July, 1998 6. Performing Organization Code
7. Author(s) Dr. Arthur E. Salwin		8. Performing Organization Report No.
9. Performing Organization Name and Address Mitretek Systems 600 Maryland AVE SW STE 755 Washington, DC 20024		10. Work Unit No. 11. Contract or Grant No. DTFH61-95-C-00040
12. Sponsoring Agency Name and Address Department of Transportation Federal Highway Administration ITS Joint Program Office 400 Seventh ST SW Washington, DC 20590		13. Type of Report and Period Covered 14. Sponsoring Agency Code HVH-1
15. Supplementary Notes Bill Jones and Lee Simmons		

16. Abstract

This document assembles best practices and presents practical advice on how to acquire the software components of Intelligent Transportation Systems (ITS). The intended audience is the "customers"--project leaders, technical contract managers, decision makers, and consultants--who are responsible for one or more ITS systems.

The document presents a series of "themes" that serve as guiding principles for building a successful acquisition. Included are people themes of collaboration, team building, open communications, and active customer involvement, which have been likened to partnering; management themes of flexibility, "no silver bullets", and up-front planning; and system themes of "Don't build if you can buy" and "Take bite-size pieces". Software acquisition activities that build upon these themes are presented in subsequent chapters. Among the activities covered are building a team, developing requirements, making build/buy decisions, resolving the intellectual property rights, acceptance testing, and project and risk management. Also included are "war stories" to illustrate the various points, as well as key point summaries and checklists to facilitate use of the material. The document concludes with short stand-alone topic sheets that introduce various relevant software topics.

Key Words Software, Acquisition, Procurement, ITS, Intelligent Transportation Systems	18. Distribution Statement No restrictions.		
19. Security Classif. (of this report) Unclassified	20. Security Classif. (of this page) Unclassified	21. No of Pages 224	22. Price

Form DOT F 1700.7 (8-72) Reproduction of completed page authorized

The Road to Successful ITS Software Acquisition

Volume II: Software Acquisition Process Reference Guide

July 1998

PROTECTED UNDER INTERNATIONAL COPYRIGHT
ALL RIGHTS RESERVED.
NATIONAL TECHNICAL INFORMATION SERVICE
U.S. DEPARTMENT OF COMMERCE

Prepared for the Federal Highway Administration
by Mitretek Systems

ROADMAP TO VOLUME II

This is Volume II of *The Road to Successful ITS Software Acquisition.* In Volume I we gave an overview and introduced a number of themes to guide your software acquisition. In this volume, we discuss a number of activities that build upon those themes. Unlike Volume I, we recognize that probably no one would ever pick up this volume and read it cover to cover. Instead, we recommend that as your acquisition unfolds, and various topics discussed here become relevant you turn to the appropriate chapters.

Chapters may become relevant sooner than you expect. Many acquisition activities have to be first addressed long before they actually occur. Take acceptance testing, for example. Even though acceptance testing does not take place until after development activities are complete, it must be planned for early. In other words, the chapter on acceptance testing will become relevant long before testing actually takes place.

Parts Three through Six of the document appear in this volume.

Parts Three and Four discuss the various activities that constitute a software acquisition. Each chapter discusses a separate activity. Collectively, these activities encompass the entire period from the system's initial concept, through the software development, on to the end of the system's operational life. The themes introduced in Part Two recur throughout the various activities. When they do, we'll call them out.

The activities in Part Three have defined beginning and end points.

- Chapter 6, *Sequence of Acquisition Activities* gives an overview of these activities and discusses how no single timeline can be generated to describe all acquisitions.

  The next two chapters discuss activities that necessarily take place early in the acquisition.

- Chapter 7, *Building A Team* discusses the players that must be assembled to work together. This must be done early, so they can work together and carry out the acquisition.

- Chapter 8, *Planning The Project* discusses the project plan used to organize an acquisition. Even activities that will not take place until late in the acquisition must be included in this plan. The plan helps to ensure that all the team members are trying to achieve a common goal.

  The next three chapters discuss key activities that drive the rest of the acquisition: These activities all feed off one another and to some extent take place in parallel.

- Chapter 9, *Requirements* is divided into two subchapters: 9A, *Developing Requirements* and 9B, *Requirements Management.* Developing requirements culminates in a requirements document. However, attention to requirements cannot end at that point. Requirements management is still needed for the

remainder of the acquisition. Requirements creep must be avoided, but at the same time requirements cannot be "thrown over the fence" and forgotten. Although requirements management is an on-going activity that could have been discussed in Part Four, we chose to include it here to keep all the requirements-related material in one chapter. The length of this chapter reflects the importance of requirements in a software acquisition.

- Chapter 10, *Build/Buy Decision(s)* urges you to give serious consideration to using off-the-shelf systems or system components, rather than building your own. However, this is not a panacea, and the risks in doing so are also discussed.
- Chapter 11, *Selecting the Contracting Vehicle* introduces the various contracting options and discusses their applicability to software acquisitions.
- Chapter 12, *Identifying The Software Environment* discusses the hardware, software, and communications context of the system.
- Chapter 13, *Resolving The Intellectual Property Rights* addresses the contentious issue of who has what rights to the software once it's developed.
- Chapter 14, *Project Scheduling* shows how to pull together the various planned activities into a realistic and achievable schedule.

  The last two chapters in this part address activities that do not take place until the end of the acquisition. Nonetheless, planning and preparation for these activities must begin much earlier.

- Chapter 15, *Acceptance Testing* discusses how to determine whether the system is ready to go operational in a way that is fair to both the customer and the contractor.
- Chapter 16, *Training, Operations, and Software Maintenance* discusses three important activities that collectively will probably take up considerably more than half the budget over the life cycle of the system.

The activities in Part Four take place throughout the entire acquisition. The contractor may do the bulk of the technical work, but the customer still has an active and vital role to play even after contract award. Since the various activities have no natural time sequence to them—they all take place simultaneously—the chapters are ordered alphabetically. As in Part Three, we call out the various themes when they occur.

- Chapter 17, *Project Management* focuses on gaining visibility into the project and what to do if that visibility uncovers a schedule slippage. It also addresses quality management steps that can be carried out to achieve various quality factors such as reliability and maintainability of the system.
- Chapter 18, *Software Configuration Management* discusses configuration management and baselining activities. Without these activities, the various parts of the acquisition can rapidly become out of "synch" with one another.
- Chapter 19, *Software Risk Management*, focuses on how to identify risks and manage them before they become problems.

Part Five wraps things up.

- Chapter 20, *Best Practices Checklist and Key Points Summary* provides a final checklist summarizing best practices. It also collects together the key point summaries and checklists that appear throughout the document.
- Chapter 21, *Where To Get More Help,* suggests outside sources of information.
- Chapter 22, *Concluding Remarks* sets you on the road to your software acquisition.

Part Six contains a series of stand-alone topic sheets. Typically one or two pages in length, these introduce various software topics "offline", without interrupting the main flow of the document.

Throughout the document, checklists supplement the text. In addition the following icons appear throughout:

The themes icon is used to highlight when an activity or recommendation is a specific instance of one of the acquisition themes introduced in Part Two of Volume I.

Chapters end with a bulleted list of key points that summarize the main messages of the chapter.

Sidebars are used to clarify various points or to relate "war stories" on the software acquisition experiences of the ITS community.

The dictionary icon appears when new terminology is defined.

The stack of reference books appears when references are given to the outside literature.

The Road to Successful ITS Software Acquisition
Table of Contents

Chapter		Page
Volume I: Overview and Themes		
Acknowledgments		iii
Preface		v
Document Roadmap		vii
Executive Summary		ES-1
Part One: Setting the Stage: The Big Picture		
1	The Nature of Software	1-1
2	Software Acquisition In A Larger Context	2-1
3	Differing Perceptions of ITS Software	3-1
4	Types of ITS Software Systems	4-1
Part Two: Themes on the Road to Successful ITS Software Acquisition		
5	Themes of Successful Software Acquisition	5-1
Concluding Remarks to Volume I		
References		RE-1
Volume II: Software Acquisition Process Reference Guide		
Roadmap to Volume II		iii
Part Three: Activities on the Road to Successful ITS Software Acquisition		
6	Sequence of Acquisition Activities	6-1
7	Building A Team	7-1
8	Planning the Project	8-1

Chapter		Page
9	Requirements	9-1
	9A. Developing Requirements	9-1
	9B. Requirements Management	9-14
10	Build/Buy Decision(s)	10-1
11	Selecting The Contracting Vehicle	11-1
12	Identifying The Software Environment	12-1
13	Resolving The Intellectual Property Rights	13-1
14	Project Scheduling	14-1
15	Acceptance Testing	15-1
16	Training, Operations, and Software Maintenance	16-1
Part Four: On-Going Management Activities		
17	Project Management	17-1
18	Software Configuration Management	18-1
19	Software Risk Management	19-1
Part Five: Putting It All Together		
20	Best Practices Checklist and Key Points Summary	20-1
21	Where To Get More Help	21-1
22	Concluding Remarks	22-1
Part Six: Topic Sheets		
TS-1	Rapid Prototyping	TS1-1
TS-2	Security	TS2-1
TS-3	Software Acquisition Capability Maturity Model (SA-CMM)	TS3-1
TS-4	Software Capability Maturity Model (SW-CMM)	TS4-1
TS-5	Software Safety	TS5-1
TS-6	The Year 2000 Problem (Y2K)	TS6-1
References		RE-1

The Road to Successful ITS Software Acquisition
List of Checklists

No.		Page
8-1	What To Include In The Project Plan	8-3
9-1	What To Include In A Requirements Document	9-6
9-2	Suggested Agenda Items For A Requirements Walk-Through	9-18
12-1	What To Consider When Identifying The Software Environment	12-4
13-1	Intellectual Property Rights	13-2
14-1	Software-Related Activities and Milestones on the Project Schedule	14-2
15-1	What to Include In The Acceptance Test Plan	15-9
15-2	What to Include In The Acceptance Test Procedures	15-10
15-3	What to Include In The Acceptance Test Cases	15-11
15-4	What to Include In The Acceptance Test Log	15-12
15-5	What to Include In The Report Of The Test Results	15-13
16-1	Personnel Roles Needed For System Support	16-8
18-1	How To Determine If Configuration Management Is Adequate For Your Program	18-4

Note: All of the checklists also appear together in chapter 20.

PART THREE:

ACTIVITIES ON THE ROAD TO SUCCESSFUL ITS SOFTWARE ACQUISITION

CHAPTER 6
SEQUENCE OF ACQUISITION ACTIVITIES

The chapters in Parts Three and Four of this document discuss software *acquisition* activities; software *development* activities are not addressed. The activities described in this part, Part Three, have the common feature of resulting in a specific product or event that ends the activity; whereas the activities described in Part Four continue throughout the acquisition process.

Two pivotal activities

Two activities should be recognized as pivotal during the acquisition process: developing system requirements (chapter 9) and selecting a contracting vehicle (chapter 11). They are not pivotal in the sense that they are more important than the others, that you only need to focus on these and all will go well. Rather, they are pivotal because together they are the major drivers of the sequence of activities and events during the acquisition, and because they work in conjunction with each other in driving the events. In the broadest terms, the relationship between the requirements and the contract can be summed up as follows: developing the system requirements will allow the team to determine the optimum contracting method; and the contracting method chosen determines whether the requirements must be substantially defined before selection, or can be developed in collaboration with the contractor after selection.

Consider for example, a small traffic signal system. The system concept along with an initial set of needs or requirements may be sufficient for making an implicit decision to buy an off-the-shelf product that requires no software development. As a result of these initial requirements, the appropriate contracting mechanism to purchase an off-the-shelf product can be determined and used. A features list taken from the initial requirements may be sufficient to select from among competing vendors. There would be no need to develop a comprehensive set of system requirements.

For systems that require more development, the choices are more complex. A sophisticated regionwide ATMS that includes freeway management is at the other end of the ITS spectrum. Here, selecting the contracting vehicle may prove to be the activity that drives all the others. If a time and materials contract is selected, system requirements should be developed collaboratively with the contractor *after* contract award. On the other hand, if a fixed-price contract must be used, the requirements must be well enough defined to make some initial build/buy decisions for the various subsystems. Then, for those subsystems that need significant amounts of custom software development, comprehensive system requirements would be developed in-house *before* contract award. In fact, they become part of the RFP. For the components that are bought, only high-level requirements in the form of a features list may be needed (as in the traffic signal control example above).

Order of activities and their presentation

Because printed material is sequential by nature, the activities are presented sequentially here, in chapters that follow one another. Ideally, this sequence would be one that you could follow, step-by-step, as you carry out your software acquisition. Unfortunately, the acquisition process is not linear; it doesn't proceed in such a neat fashion. Arguably, *many* of the activities should be done first, a clear impossibility. Certainly, most activities must be addressed before any one activity can be completed.

Themes *Tailoring the software acquisition to meet your needs instead of following a prescribed process by rote is an example of the need for* ***flexibility.***

Furthermore, in all cases, the activities presented here will overlap, regardless of drivers and constraints that define the actual sequence of activities for your project. The events in a software acquisition really form more of a spiral than a sequence, with each activity taken to a higher level after progress is made on other activities. One can think of the activities as sharpening the definition of the project over time. Initially there may be only a fuzzy definition. Over time, the various activities bring that definition into sharper focus. To borrow a software term, the acquisition process will be a test of your "parallel processing" capability.

Consider, for example, the development of system requirements. The initial cut at the requirements will yield only needs or a general concept of what the system should do. With these in mind, site visits may be conducted or demonstrations of vendor products may be attended to "see what's out there." Combining needs with available products allows the requirements to become more specific. As the requirements become more specific, a number of determinations can be made: Is it better to build or buy? What contracting options are applicable? Who will be added to the team to provide the expertise needed? As these various decisions come into better focus, the project plan can be updated, which in turn drives the various other activities. Decisions made in relation to these other activities may then cause the requirements to be revised. For example, as things become clearer, they help determine the needs in regards to intellectual property rights to the system. This determination may in turn impact the maintenance concept for the system, which may necessitate documentation requirements. Simply put, the various activities feed off and build upon one another. Software acquisition is an iterative process.

Given the potential for different sequences of activities during the acquisition process, we have biased the information in Part Three of the document towards a sequence of activities where development of requirements is performed prior to contracting, and the requirements become part of the RFP.

What can we say about the sequence of activities?

In spite of the many possible variations in the activity sequence, there are several constants that stand out:

- Building a team should take place as soon after project go-ahead as possible. In some cases, you may informally line up team members even before the official start of the project.
- Up-front planning is critical. Even the activities that take place at the far end of the project, such as acceptance testing, training, and maintenance must be planned up front.
- All too many projects fail because scheduling is developed independently of the requirements. Instead of being a natural outgrowth of the requirements, the schedule does not reflect the realities of the time needed to perform the other activities.
- Developing system requirements and selecting a contracting vehicle together drive many other activities. They must be considered in conjunction with each other.

We've summarized these concepts in figure 6-1, which organizes software-related activities on an approximate timeline. This timeline is very generic, and will vary considerably across acquisitions. The activities begin with team building, which is a precursor for all follow-on activities. Some activities are on-going, as depicted by the arrowheads at the end of the activity boxes. The exact sequence and length of the various activities will depend on the particular acquisition, so we've aligned them with respect to the acquisition phases shown at the top of the figure (i.e., preliminary activities, software development timeframe, and system operations). For example, the System Acceptance milestone marks the transition between development and operations.

Noticeably absent from figure 6-1 are the procurement-related activities such as issuing a contract. These are summarized in figure 6-2. In this figure, building a team is again shown as a precursor for all that follows. Decisions on build/buy and the contracting vehicle are reflected in the RFP and ultimately the contract. For public agencies, the Source Selection step is necessarily a formal process. Even for purchasing off-the-shelf products, an RFP or something similar must still be issued.

One state has expanded the Source Selection step to include a design competition. Two contractors proceeded in parallel in developing prototypes, and then one was selected to proceed with the bulk of the project.

Figure 6-2 shows that the intellectual property rights must be resolved before a contract is signed. (See chapter 13, *Resolving the Intellectual Property Rights*.) On the other hand, the timing of the requirements walk-through depends upon the contract. If, in spite of recommendations contained elsewhere in this report, a relatively inflexible contracting vehicle is chosen, then the walk-through takes place before contract award. A more flexible vehicle can allow this act to take place after contract award. That is why the walk-through is shown as a dashed box.

Why two figures instead of one? Why show the software-related activities and the procurement-related activities on separate timelines? This was intentionally done because combining the two timelines would be misleading. The timing relationships

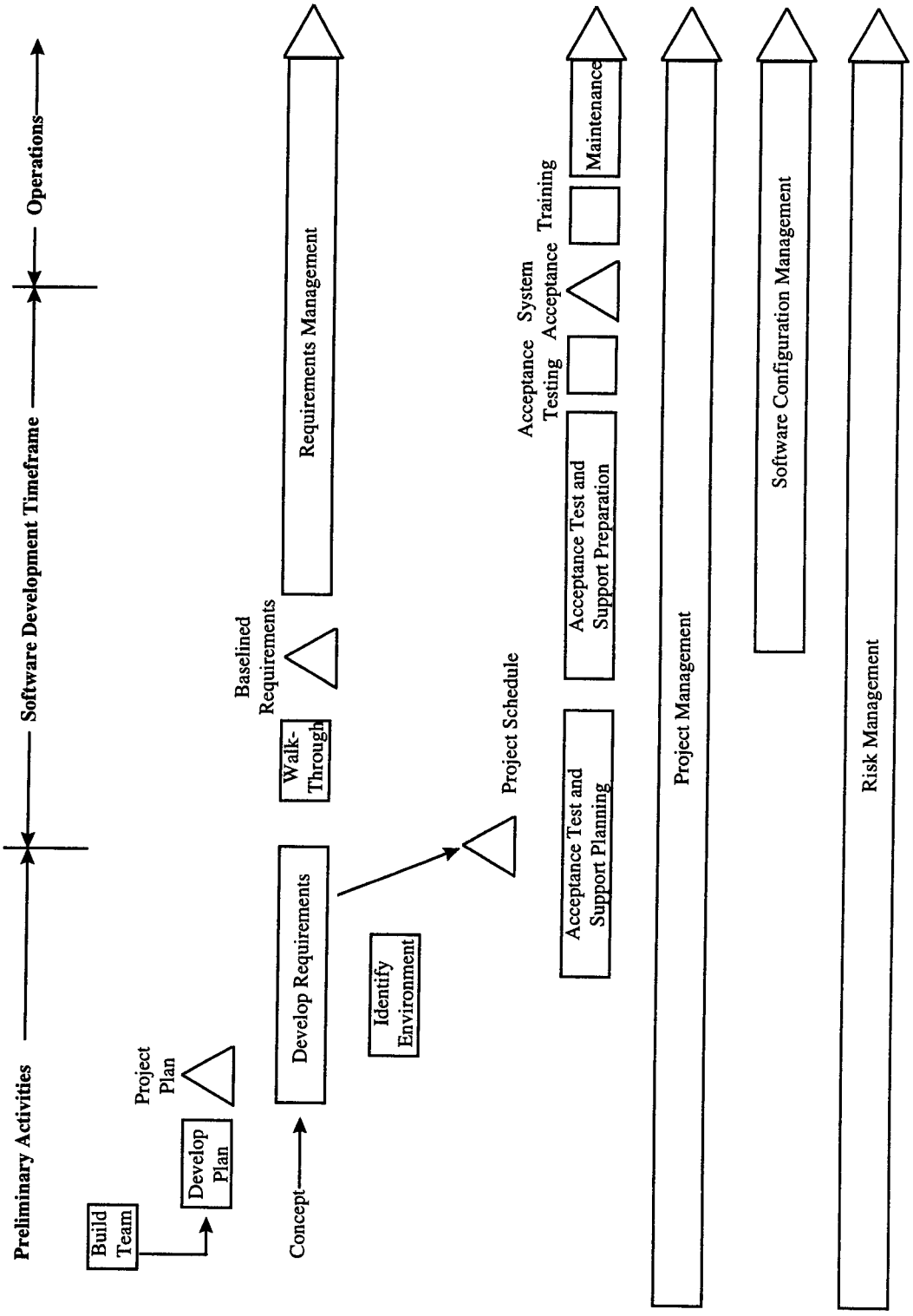


Figure 6-1. Approximate Timeline For Software Activities

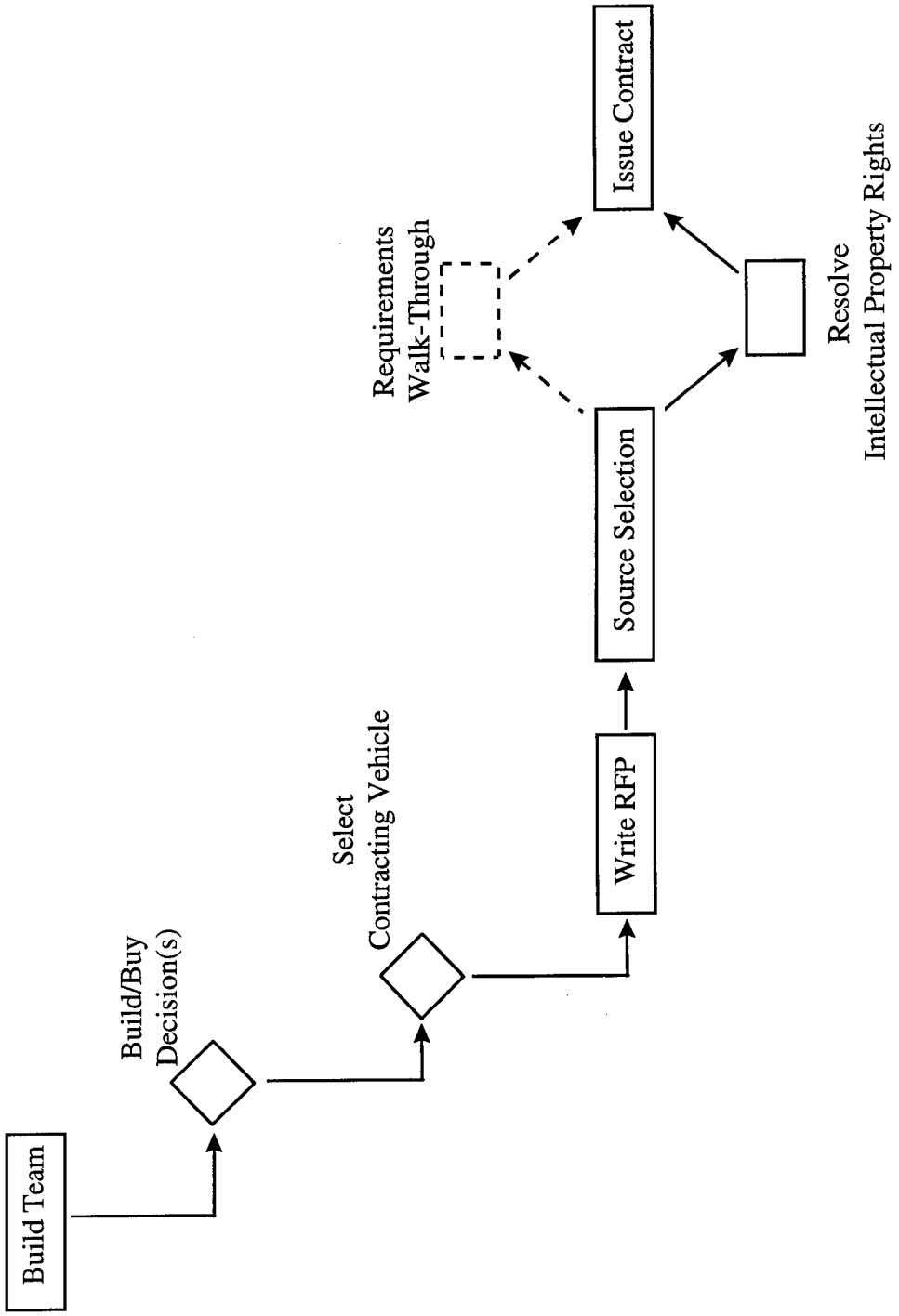


Figure 6-2. Approximate Timeline For Procurement Activities

between the two sets of activities depend upon various factors, especially the selection of a contracting vehicle:

- Our recommended approach is to select a vehicle that allows most of the software activities shown on figure 6-1, including requirements development and acceptance test planning, to be carried out collaboratively by the customer and software development contractor. Thus they will necessarily take place *after* contract award.
- For an acquisition in which the build/buy decision results in a mostly "buy" option, the requirements process can be considerably simplified, with perhaps only a features list needed to make the purchase. (There would still have to be sufficient requirements to formulate an acceptance strategy. Also a formal process, such as an RFP, would still be needed to meet your agency policies for buying the software.) This is illustrated in figure 6-3. You don't want to go too far initially in developing requirements. Not only will they be unnecessary for the buy option, they may actually unnecessarily preclude the option to buy an existing product that meets most of your needs.

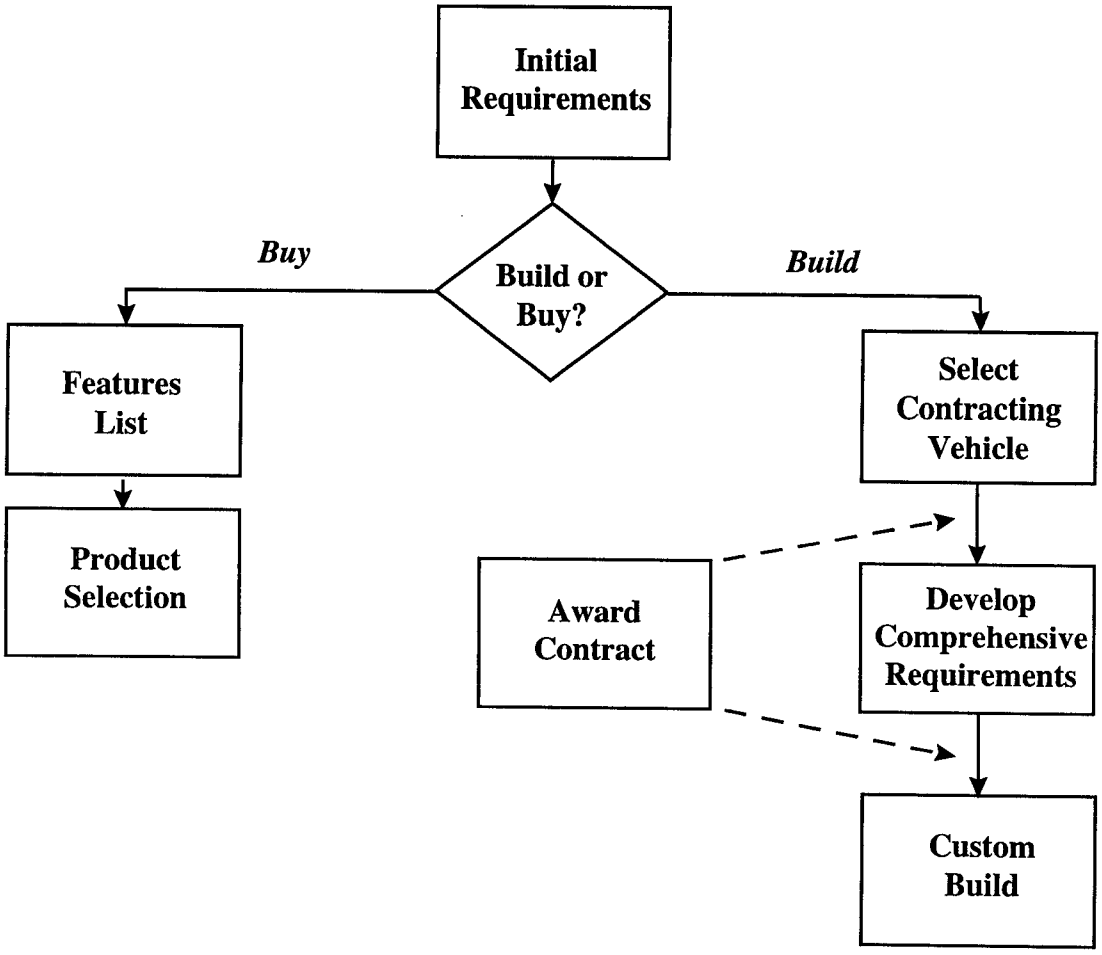


Figure 6-3. Comprehensive Requirements Only If Needed

The most important point of the figure is what's not there: the Develop Comprehensive Requirements box does not appear on the left side under the buy option. Other things to note in the figure are that the Develop Comprehensive Requirement box can take place before or after contract award depending upon the contracting vehicle you select. This again brings forth the idea that figures 6-1 and 6-2 can slide with respect to one another.

A final point regarding the figure is that teaming would still be needed under either option: essential for custom build box, it is also mandatory in working with the supplier if any customization is needed after the buy. Only the supplier knows which of your proposed changes are simple and which will require significant software development risk. Customer intuition cannot be relied upon for this.

- The software configuration management procedures used by the contractor would probably not be of concern to the customer and could be removed from figure 6-2.
- For fixed-price contracting with a relatively inflexible contract, the requirements will be developed before the "Issue Contract" box in figure 6-2. As noted above, the requirements walk-through would also take place before the contract is issued.

In short, the two timelines can be thought of as sliding horizontally with respect to one another, depending upon circumstances. The sequence of activities between figures will vary, even though the relative order within a figure should remain roughly constant.

- *There is no simple step-by-step activity sequence that applies to all software acquisitions.*
- *Developing system requirements and selecting a contracting vehicle drive many of the other activities.*
- *Many acquisition activities take place in parallel. They feed off and build upon one another.*

CHAPTER 7
BUILDING A TEAM

"A team is a small number of people with complementary skills who are committed to a common purpose, performance goals, and approach for which they hold themselves mutually accountable." — [Katzenbach and Smith, 1993 as quoted in Higuera et al., 1994]

Transportation planners and civil engineers are well qualified to carry out the core mission of a transportation or transit agency. However, software acquisition requires additional areas of expertise beyond those traditionally found in these agencies. To be successful, you must build a team of professionals with the right skill mix. Do not attempt to go it alone. The overall philosophy is not just a collection of skills, but one of forging a team that works together to achieve common objectives. Potential bureaucratic adversaries will need to be won over. This often requires considerable management and negotiating skills. Egos will have to be stroked; you'll have to use your best schmoozing techniques. Your success should be perceived as their successes as well.

Several interviewees attempted to more or less go it alone and then discovered they lacked certain critical skills as their projects progressed. So they wisely took a step back and added additional players. We hope to help you benefit from their experiences by relating where they found they needed help.

Who should be on the team?

The following are some of the skills that will be needed on your team:

- *Software technical expertise* is needed to assess the realism of the requirements, give independent size and cost estimates, review technical documents, serve as a technical liaison to the software contractor, and provide general guidance to the program manager throughout the acquisition activities. As can be seen from the range of activities, this expert (or experts) needs to have experiences beyond computer programming. Software engineering, system engineering, or software management experience is desirable. These individuals are difficult to find, hire, and retain, especially for public agencies. All the more reason for teaming with the contractor.
- *End user(s)* participate in the development of user requirements and are heavily involved in rapid prototyping activities. They interact with the software contractor, and assess the system from an operational perspective. End users have a different perspective than engineers. Engineers often have considerable computer experience; end users do not. What's easy or natural for an engineer, may not be at all intuitive for the end user. Similarly, common terminology among designers, the contractor, and the users must be developed to ensure open and clear communications. Therefore, one or more end users should be brought

on board early in the process, even though they will not actually use the system until later. Further, having end user participation is a wise political move. It achieves greater buy-in, as the users feel they were part of the development process. The system becomes a shared goal instead of a threat, and the users understand the basis for design decisions and the trade-offs that are made. The system becomes their own instead of something foisted upon them. One source cites the user "as the most important member of the team." [Farbman, 1980] Another cites user involvement as the number one reason why successful projects succeed. [Standish Group, 1994]

- *Maintainers and system administrators* of the ITS system are end users also, but of a different sort. They do not use the system operationally, but are responsible for keeping it running and making upgrades to it. They will undoubtedly have valuable insights into the system maintenance concept and knowledge of the legacy system interfaces, which can be properly reflected in the RFP and contract. Because of their unique roles, they will review the design with a different perspective than the software technical experts or the end users. They can suggest the "hooks" to be included in the system that facilitate troubleshooting problems and bringing the system back on-line quickly. They can help to define software documentation requirements, as well as review documentation to see whether it conveys the information they need to do their jobs.
- *Domain experts* determine that the system will have the intended effect within their areas of expertise. For example, traffic engineers can provide reality checks to requirements, determine if the system meets the needs of those responsible for traffic and transit management, and estimate the effects that the system will have on facilities. Domain experts will also be useful in developing training for system users. They can help the operators understand how and when the various capabilities of the system should be used.
- *Contracting and purchasing officials* can help select the most appropriate contracting vehicles. Bringing these officials on board early gives you the opportunity to explain the unusual aspects of software acquisitions. For example, you can stress the need for contract flexibility in such areas as requirements, incremental development, and rapid prototyping. This will not be "business as usual" for many contracting and purchasing officials; it will be your responsibility to make sure that you jointly explore alternatives to the rigid contracting vehicles and source selection procedures that are typically used for construction projects. You may jointly identify the need to bring in outside expertise in this area, if your agency has no software contracting experience. Bringing these officials on board early will also give them the lead time they need to accommodate the various acquisition activities into their schedules. Approaching them at the last minute with requests to use unfamiliar contracting practices will probably just be met with negative responses.
- *Software-specific legal staff* is needed to resolve intellectual property rights issues that are peculiar to software. They will need to address the thorny issues associated with licensing, warranties, ownership, and copyright. Because software legal issues are a hard-to-find sub-specialty, you will probably need to hire outside help in this area.

In some cases, there may not be any pre-existing source of expertise or even any existing positions. (There are no freeway management system operators if there is no freeway management system!) In such cases, personnel may have to be transferred from other types of positions to represent the end user. If this is done, try to use personnel who will have similar backgrounds to those who will eventually use the system. And remember, a critical qualification for selection to the team is an interest in the system and its success.

The last team member

At this point in the acquisition, you will (by necessity) be missing an important member of your team: the software contractor. Once a contract is issued, however, be sure to expand your team by adding the software contractor to it. They are a critical member of your post-contract award team. This is true whether you build or buy the system.

Treating the contractor as a team member may seem somewhat unusual. An adversarial contractual relationship may be more familiar and feel more comfortable. If so, this is another example of how software is different. However, including the contractor as part of your team will seem familiar if you have practiced *partnering* on construction projects. Partnering is a cooperative and highly interactive way of dealing with the inevitable problems of scope, cost, and schedule. (See also *Software Risk Management,* chapter 19 for teaming in risk management activities.)

One ITS manager successfully teamed with his contractor by treating them as part of his staff. They were invited to attend staff meetings and participated in setting milestones for the project.

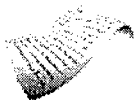
Even after the contractor joins your team, other members of the pre-contract award team should still be retained after contract award. End users will continue to have particularly important roles. However, there may be lesser roles for contracting and purchasing officials and software-specific legal expertise once a contract is issued.

- *Build a team of professionals that contains the variety of skills needed to make the project succeed.*
- *If possible, consider tapping other resources in your agency to gain access to the needed skills. When this is not possible, you may have to contract for these skills to gain access to them.*
- *Be sure to add the software contractor to your team as soon as the contractor comes on-board.*

- *Translators* are people having knowledge in multiple areas that can translate technical points, jargon, and concepts across disciplines. An example would be someone who is knowledgeable in both software and transit management, or someone who can work with the National ITS Architecture and apply it to local needs. This ensures that true communication is flowing and that everyone is speaking the same language.

Where can I find the team members?

Where can this expertise be found? Often the best places are other parts of your agency. Those responsible for information resource management functions may have relevant experiences that you need in acquiring geographic information systems (GIS) or database management systems (DBMS). This recommendation is made with the realization that, in real-world situations, not all these people will be accessible to you. Further, they may view your project as a diversion from the best interests of their long-term career paths in their "home' department. You may need to find ways to incentivize them to join your team.



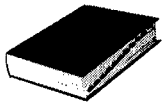
One transit agency found that their information resource management personnel were invaluable in preparing acceptance test plans since they had experience in writing them for other types of projects.

Be sensitive to the risk that apparent expertise and experience in a discipline may not directly and explicitly apply to your ITS software acquisition. Even if a person's previous software experiences were successful, they may not necessarily apply to ITS systems. For example, the standard database management system acquired by the agency, which works so well for personnel records, may be totally inappropriate for the complex real-time demands of a freeway management system or for transit fleet tracking.

If the expertise is not available in-house, you may have to resort to issuing a contract to acquire the needed expertise. This has worked well on some ITS projects. (See *Resolving The Intellectual Property Rights*, chapter 13 for how well this has worked in acquiring software-specific legal expertise.) However, contracting for expertise has caused problems on other ITS projects. Vendors complained about acquisitions in which a software expert is hired to write a requirements document and then is allowed to competitively bid to develop the system. In addition to giving the expert a time advantage in developing an RFP, the requirements often reflect the expert's own product. This may also subject the follow-on contract award to a legal challenge regarding conflict-of-interest. In another case that describes a potential problem with contracting for expertise, a state DOT hired a software expert to represent them in interactions with the software development contractor. The development contractor complained that the authority of the contracted software expert during interactions was not clear: Was the software expert stating the DOT's perspective or position? Or was he just expressing his own views?

CHAPTER 8
PLANNING THE PROJECT

As with the outset of any acquisition, you should begin by writing a project plan. This need not be a lengthy document; it should be terse and to the point. The plan should present the overall goal and objectives of the project from a global perspective, and highlight the high level strategic decisions associated with your acquisition. Initially, you may not have all the answers. That's fine. Just leave placeholder sections in the plan and come back to them later.



Your organization may have different names for what we are calling a project plan. Terms such as "acquisition strategy," "project management plan," or "software development management plan" are often used. In some cases, the plan may be split into two or more documents, such as a management plan and an acquisition plan. Whatever it is called, or however it is bound, a plan should be written.

When to write the plan

Although this chapter on writing the project plan follows the one on building a team, in practice the two steps are typically iterative. You may have to write a first draft of the plan on your own, as initially you may be a team of one member. This draft can help you identify who must be added to your team, and can be used to explain the acquisition and your current thinking to potential team members. Then as the team is assembled, other members can participate in writing the plan. Not only are their inputs valuable, but by participating in planning the acquisition from the outset, they will have a greater stake in its success.

Although the plan is initially written at the outset of the project, it should be regarded as a living document. You will need to revise it periodically to keep it consistent with current program objectives and approaches.

Uses for the plan

A project plan is a good communications vehicle among the members of your acquisition team. You will also find that a plan will serve as a valuable political document for communicating with those external to your project, especially your management. When decision makers and budget analysts ask for descriptions of your project, you have the plan ready as a handout for them. This gives a good impression of "having your act together", which is especially important during various budget review cycles. It can also serve as a handout for those seeking general information. By having a documented plan, you save having to write up a project description in fire drill mode every time a request is made. You'll find it useful to cut-and-paste sections from it, knowing that the words and ideas have already been coordinated.

Don't underestimate the value of putting the plan together in helping to crystallize your thinking and organize your thoughts. For example, if the plan states that the information management organization will provide training, then that can serve as your reminder to coordinate with that organization and identify the person who will carry out the training role. Even the placeholder sections are valuable as they serve as "tinglers" to remind you of what needs to be done. The particular approach that gets documented in the plan is probably not as important as the fact that, by writing a plan, you have gone through the thought process.

What should go in the plan?

There are no universal rules about what has to be included in a project plan. Checklist 8-1 has some suggestions. Since many of the items in the checklist would apply to all projects, we've italicized the ones that are unique to software (or are more critical for software than they may be for some other types of projects). The information in the National ITS Architecture is another good source of ideas for what to include in the plan. Your organization may have its own standards for what to include.

---

*Why have a **why** section?*

*A project plan section on justification (the **why's**) helps to keep the acquisition consistent with the global vision. Sometimes a project can take on a life of its own. It continues on, even if the assumptions under which it had been approved at the outset no longer apply. Perhaps the system was justified on the basis of saving money. But if it turns out the original budget estimate to build the system was low by an order of magnitude, then maybe the projected cost savings won't be realized. There is no need to continue under those circumstances. If your project does run into trouble, the **why's** can be used as a periodic sanity check to determine whether continuation is justified. If the **why's** are no longer valid, then project termination may be the best answer. Why prolong the agony or waste resources?*

*The **why's** are also important as you develop system requirements. It's easy to become enamored with "glitzy" or expensive features. You see them in various products or in other implementations, and the natural tendency is to say, "I've just got to have one of those." Use the **why's** to ask yourself whether these features really support your overall needs.*

*As a counter-argument you may argue that project goals and objectives evolve over time. For example, office automation systems (word processing, etc.) were originally justified on the basis of increased productivity. In most cases this increased productivity has not materialized. Yet few organizations would be willing to go back to the days of typewriters; other, unanticipated, benefits have been realized. So if your project's original justification is no longer valid, but there are still good reasons for its continuation, change the **why's** in the project plan to reflect current reality and proceed.*

Many of the items in the checklist concern activities that won't take place for some time. For example, system acceptance, training, and maintenance don't take place until after the software development activities. Nonetheless, they must be planned and budgeted early. You may not have the information to address them in initial versions of the plan. But planning does not stop when the plan is first published. You will certainly need to think

Checklist 8-1. What To Include In The Project Plan

✓	Project Description: A brief narrative of what the project is all about, its goals, objectives and scope.*
✓	Justification: Why the acquisition will take place; saving money, alleviating congestion, providing better on-time service are all possibilities; although increased productivity is sometimes offered as justification, in practice, this seldom materializes.
✓	Project Schedule: An overall schedule showing when the major milestones take place.*
✓	Roles: Who will manage the project? What is the size and composition of your acquisition team? What organizations will be involved? Who are the contact points within each of these organizations? What are their respective roles? Who will be responsible for training? For maintenance? Can include an organization chart.
✓	Funding Estimates and Sources: You can refer back to these to ensure that you are living within your constraints.
✓	*Facilities:* Where will the work be carried out? Where will the system and its users be housed upon completion? Will any special tools or equipment be needed?*
✓	*Acquisition Strategy:* Will the system be built from scratch? To what extent will off-the-shelf components be used or sought?* How will such components be integrated into the system? Are there pieces of the system that can be reused from other projects?* Will the system be build incrementally using a multi-phase approach, in which each phase contains a task to define and scope the next phase? Will a prototype be built? How will off-the-shelf products be integrated with each other?
✓	*Environment:* Are there any legacy systems that must be interfaced with? Field sensors, roadway or transit vehicle devices? How about other organizations or neighboring jurisdictions?
✓	Standards: What technical standards must be complied with?
✓	*Major Risks and Risk Management Approach:* How will risks be managed? Have any key risks been identified?*
✓	Contracting Strategy: What work will be done in-house? What will be contracted for? Will consultants be hired?

[Checklist continued on next page]

Checklist 8-1. What To Include In The Project Plan (Concluded)

✓	Type of Contract(s):* What options are being considered? Fixed price, cost-plus, or time and materials? Design/build or system manager? How many contracts will be needed?
✓	*Contract Management:* How will oversight be accomplished? How will progress be tracked and monitored?*
✓	*The End Users:* Who will operate the system? Administer it? Maintain it? What will be the sources of staffing and funding for these activities?
✓	*Acceptance Strategy:** What will be the basis for accepting the system?
✓	*Training Concept:* How will user training be accomplished?
✓	*Maintenance Concept:* How will the system be maintained once it is accepted?
✓	Constraints:* What are the realities that you must live within?

* Recommended in the "Software Acquisition Capability Maturity Model (SA-CMM)," [Ferguson, 1996, pages L2-4 and L2-5].

NOTE: *Italicized* items are unique to software, or are more critical for software then they may be for some other types of projects.

through these topics before a software development contract is issued. Don't get into the situation of the interviewee who told us, "Maintenance kind of caught us by surprise."

Again, when you begin the plan you may not have answers to all the items in the checklist. Leave a placeholder in the plan and add the information as it develops.

- *Write a short project plan that documents your major approaches to the acquisition.*
- *The project plan is a living document during the acquisition process; new information will be added, and existing information may need to be revised.*

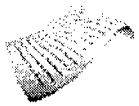
CHAPTER 9
REQUIREMENTS

"The hardest single part of building a software system is deciding precisely what to build. No other part of the work so cripples the resulting system if done wrong. No other part is more difficult to rectify later." — [Brooks, 1987]

9A. Developing Requirements

The importance of requirements

Developing a good set of requirements is one of the most important things you can do on a software acquisition. This lesson is repeatedly emphasized in the software engineering literature. Indeed, our public-sector and private-sector interviewees from the ITS community confirmed that this general lesson applies equally well on ITS acquisitions.



This chapter addresses the requirements that characterize the operation of the eventual system. Project requirements (testing requirements, legal requirements associated with the contract, documentation requirements, contractor reporting requirements, etc.) are covered elsewhere.

Research showed that, of the various software engineering practices used on real world projects, emphasis on requirements had the strongest positive impact on software quality and productivity. [Glass, 1982, page 235] Conversely, problems encountered on projects are often due to poor requirements: Another study surveyed 8,000 projects. It showed that three problems attributable to requirements—lack of user input, incomplete requirements, and changing requirements—are the major reasons that projects were late, over budget, or had less functionality than desired [Standish Group, 1994 as quoted in McConnell, 1996].

Requirements are important because they "establish a common understanding" between the customer and the software developer. "This agreement with the customer is the basis for planning and managing the software project." [Paulk 1993] Requirements are the basis for: size, schedule, and cost estimates; build vs. buy decisions; design and development activities; and acceptance testing.

Having good requirements is also important from a cost perspective. Problems uncovered during latter stages of a project can often be traced back to flaws in the requirements. However, the later these flaws are uncovered, the more they cost to correct. In fact, the conventional wisdom is that as you progress from one phase of the software life cycle to the next, the cost for fixing a requirements error increases by a

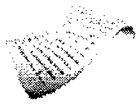
factor of ten. This cost inflation factor also applies to changes in requirements—the later they are made, the more they cost.

Who develops the requirements?

It is important that various types of expertise represented on the customer's team actively participate in developing the requirements. Our interviewees all agreed that the transportation engineer should not write the requirements alone. Here are some roles for the various team members:

Themes *This is an example of our themes of* ***team building*** *and* ***collaboration.***

- *Domain experts* ensure that the system addresses the operational needs that it is being built for.
- The *software expert* helps write the requirements, and reviews them to see that they are complete, clear, consistent, testable, and feasible to implement. [Ferguson, 1996] The expert can also carry out the rapid prototyping activities before contract award rather than having the contractor do it after contract award. (See *Develop human interface requirements using rapid prototyping techniques*, in chapter 9B below.)
- The *user operators* assess the requirements to see that they meet their needs. (Operators typically have different perspectives and ways to approach systems than do engineers or software developers.)
- The *system administrators and maintainers* bring a different perspective on the system than the user operators. Accordingly, they can specify requirements that facilitate diagnosing problems or bringing portions of the system on-line or off-line. These capabilities are often overlooked by engineers or end users with only an operational perspective.



Joint Application Development is a current management practice that is in vogue. It involves having users, executives, and developers going off-site to define requirements and design the user interface in a structured, negotiated process.

Note that the requirements are developed collaboratively as a negotiated process. Give the user operators feedback on excessive requirements; do not just accept them as gospel. Offer the following tradeoff: "We may not be able to acquire a system that meets all your requirements, but we'll be able to get it for you much sooner."

Themes *This is another example of the* ***flexibility*** *theme. In this case, the user operators need a flexible mindset.*

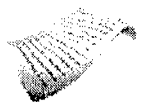
How do you determine what the requirements are?

Before you can develop the requirements, you first need to understand what you want and why. Work with all the members of your team to develop the requirements. Team brainstorming sessions, interviews with users and system operators, and the National ITS Architecture are good sources of information for determining requirements.

Be prepared for multiple iterations of the requirements. Initial team meetings may be needed just to achieve consensus on what problems you are trying to solve and what the system must do to solve them. Assign responsibility for committing meeting notes and interview sessions to paper. Review these drafts at subsequent meetings. At various points, be sure to prioritize the requirements; you can't do it all.

As discussed in *Software Acquisition In A Larger Context,* chapter 2, the system concept is the basis for the requirements. With your needs firmly in mind, you may want to attend trade shows or visit other sites that have implemented similar systems. This provides you with the opportunity to see what features have been implemented elsewhere that would address your needs. In some cases, this opens up possibilities for valuable features that may not be apparent otherwise. In other cases, it may serve to lower expectations as to the limits of what's reasonably achievable within the state-of-the-practice. This should help minimize the risk of specifying high risk or unmeetable requirements. But those who don't have their needs firmly in mind before the site visits may be swayed by the glitzy features that they see. They could decide to require capabilities, not because they're needed, but because they look nice.

You can also ask to see demonstrations (perhaps at your site) of existing products and look at the documentation. Doing so will allow you to find out what's available in the marketplace. This gives you the opportunity to identify requirements that are nice to have but preclude the use of off-the-shelf solutions. Often, small, seemingly minor requirements, prove to be overly restrictive. You may decide to relax or eliminate such requirements. In other words, your goal is to develop the least demanding set of requirements that both meets your needs and allows as many existing products to be bid as possible. Even if you decide not to go with an off-the-shelf solution, the demonstrations offer the same advantages noted in the previous paragraph regarding site visits.



The consideration of the availability of existing products and the willingness to trade off functionality to decrease cost and schedule has been cited as a "best commercial practice" that is used by the private sector. [Ferguson and DeRiso, 1994.]

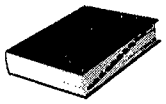
As is true for other recommended software practices, site visits and product demonstrations are not a panacea. They have potential downsides too. Several of our private-sector interviewees suggested the use of customer site visits and viewed them favorably. At the same time, they also identified the risk that site visits will cause the customer to want one feature from System A and another feature from System B. There is a danger that customers will be "mesmerized by gee-wizz technology". The resulting requirements will in effect specify

a composite or hybrid system, "a little bit of everything". This not only precludes using existing off-the-shelf products, it may even result in mutually exclusive requirements that cannot all be met. Again we emphasize the importance of first understanding your high-level needs or having a system concept. Then embark on site visits or product demonstrations to determine specific requirements.

Also beware that just because you see operational software working at another site, that does not necessarily mean it will work at yours without significant modification. It will only "plug and play" at your location if your environments are identical. Otherwise, depending on the software, you may incur significant cost and development risk if you try to obtain rights to their software "as is."

The requirements document

Because the requirements are so important, be sure to record them in a formal, configuration-controlled requirements document. (See *Software Configuration Management,* chapter 18.) Our interviewees repeatedly stressed that writing this document is worth the time and effort it takes to produce it. As one ITS manager warned, "Don't short change this phase!"



In the software engineering community, the requirements document is often referred to as a "requirements specification." However, "specification" seems to connote "design" to many transportation engineers, as in "design spec." Therefore, we will not use the term "requirements specification."

What are the characteristics of good requirements?

The requirements should tell *what* but not *how*. This is very different from the design specs that transportation engineers are familiar with. There, detailed technical standards are cited on how a road or bridge is to be built. Do not include design decisions in the requirements.

Although the *what, not how* philosophy is generally sound, it is sometimes hard to follow in practice: one person's *what* is another's *how*. Guidance obtained from our interviews with transportation officials may be somewhat clearer. They told us that a requirements document should contain *performance* requirements and not *technical* requirements. Nonetheless, vendors complain that customers can't resist the temptation to engineer the system, and resort to overly restrictive technical specifications. This can result in a more expensive and less effective product. It may also unnecessarily preclude buying a viable existing off-the-shelf system.

We are using the terms "performance requirements" (the what's) and "technical requirements" (the how's) for the two types of requirements. These seem to be the words most commonly used within the transportation community. However, there are no uniformly accepted definitions and other words are used as well. Technical requirements are also called design requirements; performance requirements are sometimes called functional requirements. To further complicate matters, the military and aviation communities distinguish between functional requirements and performance requirements. To them, an example functional requirement would be detecting and tracking enemy aircraft. The corresponding performance requirement would be processing 4000 aircraft target reports per second, with a 1.5 second response time for displaying their positions.

A customer's design can also constrain the contractor and inhibit innovation. This is a particular danger on fixed-price contracts or when the contractor is not allowed to deviate from rigid requirements or specifications. One way to overcome this problem is to allow offerors to document deviations to the requirements in their proposals. The customer is then given the flexibility to make best value judgments based upon what the offerors submit. If the customer determines that an offeror has a satisfactory cost-effective alternative to a requirement, then the deviation can be accepted. If you follow this *best value* approach, make sure to document the deviations in the resulting contract. Otherwise there is the risk that the deviation will not subsequently be allowed because it is not in strict compliance with the documented (and possibly over-specified) requirements.

A good requirement should be clearly stated, unambiguous, testable, feasible to implement, and consistent with all other requirements. In general, requirements are documented as *shall* statements. (*"The system shall input ..."*). A good practice is to use a separate sentence for each *shall*. Each *shall* can be assigned a number, which facilitates tracking requirements later on (against project completion status or acceptance test cases, for example).

What goes in the requirements document?

Checklist 9-1 has items for you to consider incorporating in your requirements document. The requirements have been grouped into several categories including functional requirements, performance requirements, and interface requirements. However, there are other ways of grouping them, so the order of presentation in the checklist does not imply a table of contents for the requirements document.

In keeping with the philosophy of not specifying design in the requirements, some strongly urge that even the computing platforms and operating systems *not* be identified in the requirements. There is, however, one exception to "no design": Identify required interfaces to legacy systems. In fact, even if the development of requirements is deferred until after contract award, it is still a good idea to identify legacy systems and other aspects of the environment in the RFP. (See *Identifying The Software Environment,* chapter 12.) The National ITS Architecture is a useful source of information in identifying these interfaces.

Checklist 9-1. What to Include in a Requirements Document

	Functional requirements
✓	What capabilities the system must have. The trick is to stay at a functional level and not prescribe a solution.
✓	Each required function takes the form of a sentence with the word "shall" and should be testable. (e.g., "The system shall display a congestion warning message on variable message signs.")
✓	Define whether the function is manual, automated, or semi-automated (e.g., the system shall choose a message and display it; the operator shall type in a message which gets displayed; the operator chooses from among several pre-defined messages and causes the system to display it).
✓	High-level human interface requirements. More detailed human interface requirements may also be included, but rapid prototyping is the preferred approach for them.
✓	Algorithms or equations.
✓	Year 2000 compliant (see topic sheet 1 on The Year 2000 Problem (Y2K)).
✓	Conforms to the National ITS Architecture.
	Performance requirements
✓	Response times, expressed as averages, standard deviations, 90 percentile, etc. ("The systems shall have a mean response time of 30 seconds with 90 percent of all responses within 45 seconds.")
✓	Loading requirements (e.g., being able to handle simultaneous inputs from a specified number of sensors), including degradation requirements, if any, under excessive load.
✓	Throughput (e.g., number of transactions per hour).
✓	Capacity ("The system shall be able to store 30 days of incident reports.")
✓	False alarm rates, including the algorithms to be used in determining the rates.
✓	Accuracy, specifying the algorithms.
✓	Reliability and maintainability (e.g., mean time between failures; mean time to repair).
✓	Security (see topic sheet 2 on Security).
✓	Safety (see topic sheet 5 on Software Safety).

[Checklist continued on next page]

Checklist 9-1. What to Include in a Requirements Document (Concluded)

	Interfaces, external and internal, including the data (inputs and outputs) and controls that flow across the interface*
✓	To/from field devices.
✓	To/from displays.
✓	To/from users.
✓	To/from other systems, including legacy systems.
✓	To/from other jurisdictions.
✓	Between major subsystem components (e.g., between a vehicle subsystem and a transit management center).
✓	Between software components (e.g., between incident detection algorithms and data collections.
	Inputs
✓	Identify its source (automated and human).
✓	Frequency of arrival.
✓	Valid ranges and units of measures.
✓	Give each one a unique name and identifier.
	Outputs
✓	Include real-time outputs (e.g., alerts to a display) and non-real-time (e.g., summary reports printed out on paper).
✓	Identify its destination (devices or users).
✓	Frequency of generation.
✓	Valid ranges and units of measure.
✓	Give each one a unique name and identifier.

*Data flows in the National ITS Architecture are a good source of candidate interface requirements, including the data flows and their descriptions.

Documentation standards are another source for determining what to put in the requirements document. If your agency doesn't have a standard, consider using the *IEEE Recommended Practice for Software Requirements Specifications* [IEEE, 1993a].

Caution: don't ask for too much

One of the most common problems on software acquisitions is that the systems are over-specified. There's always the temptation to add "just one more requirement, it's only a matter of some more software". An experienced systems engineer once observed that there's nothing wrong with adding new features to the solid base of an implemented system. However, adding requirements for these same features for an as yet unbuilt system just weighs the whole thing down.

In fact, there's a truism that "less is more": if you ask for a little bit, you'll get it; if you require too much, you risk spending a lot of money and heartache and getting less (or maybe nothing at all) in the end. It's fine to have an ambitious long-range vision, but the vision should be implemented one step at a time. New capabilities can always be added incrementally, as long as the vision is in place.

Themes *This highlights our theme of taking **bite-size pieces**. Another advantage of keeping the requirements in check is that it allows for another one of our themes, **don't build if you can buy**.*

Once an initial system is built, users gain operational experience with it. The results are often unanticipated. In some cases, deficiencies in the system may not surface until then. New requirements will arise out of the day-to-day use. In other cases, user roles may change in unanticipated ways as a result of the automation. Regardless of their origin, these new requirements can then be incorporated into the project plans. Often they will turn out to be more important than the *nice-to-haves* that were contained in the original requirements. Some of those anticipated needs will turn out to be not that important after all, while wholly unexpected needs will arise. But the only way to find out is to get the system fielded sooner. And that can best be done by holding the requirements to a manageable size.

Requirements "scrub sessions" are one technique to weed out nonessential requirements. The philosophy behind a requirements scrub is that when a requirement is removed, it eliminates all the costs and time for the associated design, test, integration, and documentation activities. As noted previously, the effort, complexity, and schedule will be reduced disproportionately; the system reliability may also increase disproportionately. Depending upon your contracting approach, you may want to carry out at least two scrub sessions: the first before the RFP is issued; the second, as part of a requirements walk-through after the software contractor is on board. To avoid unnecessary repetition, the details on conducting a requirements scrub are included only once in this document. (See *Conduct a requirements walk-through,* below in chapter 9B.)

Another way to avoid requirements creep is to periodically revisit the system concept in your project plan. (See the *Planning The Project,* chapter 8.) As you develop the requirements, examine them in the light of the concept. Consider the following questions: Do the requirements support the concept or are you losing sight of the big picture? Are you getting sidetracked adding *nice-to-haves* that don't really address the *why's* for the system?

How thorough should the requirements document be?

The answer to this question is somewhat controversial: Consultants generally recommend that the requirements document be "complete, specific, and comprehensive". This allows the requirements to serve as the basis for all that follows, without ambiguity. However, this viewpoint is not shared by product vendors who sell software systems. They feel that comprehensive requirements almost inevitably preclude an off-the-shelf solution. They also cite the problems that are inherent in any requirements document. (See *The need for requirements management,* below in chapter 9B.)

Instead of going with a comprehensive document, product vendors generally prefer a short list of high-level requirements (incident detection, surveillance, etc.). These would then be used to select the best off-the-shelf product. (See the matrix in *Build/Buy Decision(s),* chapter 10.)

One option that may be satisfactory to both communities is to develop a high level set of requirements that are issued as part of the RFP. These give the offerors something to base their proposals on and can also be used as the basis for build/buy decisions. (See *Build/Buy Decision(s),* chapter 10.) Then for the portions of the system that you decide to build, work with the development contractor after they come on board to collaboratively flesh out a comprehensive set of requirements. (See also figure 6-3.)

How about human interface requirements?

Human interface requirements are those that address how operators and other end users will interact with the software system. Although we have included human interface requirements in checklist 9-1, rapid prototyping is a better approach for determining them. (See *Develop human interface requirements using rapid prototyping techniques*, in chapter 9B below.)

Quality factors

Related to requirements is the subject of *quality*: "how well" the system meets the requirements. Quality factors are more difficult to measure or test than performance or functional requirements. In many cases, formal acceptance testing may not be practicable. You may have to resort to less rigorous procedures, such as inspecting the software or analyzing its ability at a design review. Nonetheless, quality factors are still important. Even if the contractor cannot directly measure or test them adequately, the factors place emphasis and may make a difference in the design of the system. Let us briefly examine some of the quality factors that help define quality: We concentrate on the more practical ones, and briefly mention some others.

- *Reliability.* The extent which software will perform without failures within a specified time period. Usually measured in terms of the number of failures per unit time, it can also be measured in terms of defects. A defect may not lead to a failure, for example a document error or some malfunction that does not interrupt the operation. Thus, reliability requires a definition of what exactly one means by failure. If the user is simply concerned with the operation of the system we can use the factor Availability.
- *Availability.* The extent that the system is available for operation without interruption. The time between failures is measured and is converted to a percent that gives the amount of "up time" for the system. If we say that the system has a 95% availability, it means that in one hundred hours the system will be down for only five hours. We have to consider how to handle partial non-availability. Does the entire system have to be available or is some degradation allowable?
- *Maintainability.* Ease of effort for locating and fixing a problem or failure within a specified time period. How long, on average, does it take to repair a single failure or problem (which ones—all failures may not crash the system)?
- *Usability.* Relative effort required for training or for using the system. This is a tough one. It has to do with human factors which are hard to measure. Some ways that it can be specified are in the number of key strokes or operator actions required, or the response time of the system.
- *Expandability.* How easy is it to increase the performance or capability of the system. This usually has to be built into that design of the system so that future enhancements can be easily accommodated. Usually this means ensuring that the system can accommodate additional resources such as memory. It could also mean that the system is designed with standard internal and external interfaces such that future changes can be accommodated without major changes to the architecture and implementation of the system. (See also *Planning for flexibility,* the next section below.)

Requirements for the first three (reliability, availability, and maintainability) should take into consideration the skill levels of the personnel who will be responsible for operations and maintenance; this is a key ingredient in determining what can reasonably be achieved. Other quality factors that are more difficult to specify, measure, and test are sometimes used. Two of them are:

- *Portability.* Relative effort to transport the software for use in another environment.
- *Reusability.* Relative effort to convert the hardware or software component for use in another application. Applies in cases where the system, or parts of the system, may be reused in another similar application. It is to the suppliers' advantage to accommodate this requirement as they might want a marketable system that can be reused in other applications. However, trying for too broad a development on one customer's money is risky, and can jeopardize trust and teamwork.

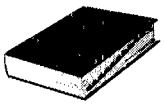
See also the discussion on quality management, in *Project Management,* chapter 17.

Planning for flexibility

Another subject related to requirements is system *flexibility*. Software is inherently more flexible than hardware during the development stage. Ironically, deployed systems that use software are often inflexible: they can't incorporate new capabilities, and existing components can't be replaced. In some cases, antiquated computers are kept running because the cost and effort of migrating the software to a new computer would be too disruptive and expensive. As a result, the rapid advances in computer and peripheral devices are not taken advantage of.

Systems that are acquired without flexibility in mind may not even survive long enough to become operational, dying before their development is complete. [Horowitz, 1991; Saunders, Horowitz, Mleziva, 1992]

The essence of planning for flexibility is to accommodate change. Planning for flexibility addresses the components of a system, the interactions and connections between them, and the constraints. It considers how each of these can evolve over time. [Garlan and Perry, 1995] Note that we are talking at a higher level than the design of software algorithms or data structures. Instead, we are addressing the overall organization of the system.



The software engineering community uses the terms "software architecture" and "system architecture" in conjunction with the overall organization of a system. However, to avoid confusion with the National ITS Architecture, we will refrain from using "architecture" in our discussions.

For example, the design of ramp metering algorithms is not of concern here. Rather, we are interested in where these algorithms fit into a freeway management system, what data sources are available, what additional data sources may become available in the future, what other components will use the output of the algorithms, even what are the likely changes to these other components in the future.

One way to approach planning for flexibility is to ask yourself, "What features of the system are likely to change or grow over the next five years?" Will new vehicles be added to a vehicle tracking system? Will more roads be put under surveillance of a traffic management system? Will surveillance be upgraded from loops to video? Will any of the legacy systems be replaced? Will traffic data be shared with neighboring jurisdictions in the future? Then, as you work with your contractor or conduct design reviews, play "what if". See if the system will be able to accommodate these changes. The goal is to make the changes as painless as possible.

In general, you don't want to specify the overall organization of the system in the RFP. Vendors are almost always the most qualified to do this for their own software. (If your vendor is not qualified, you should find another vendor!) In fact, the private-sector interviewees were adamant that customer-supplied technical specifications are a bad idea. "Leave it to the supplier."

Nonetheless, there are some aspects of planning for flexibility that do need to be considered as part of the acquisition:

- Identify the environment in which the software system is to operate. This frequently includes other systems that are yet to be acquired in addition to systems that you already own. You may want to require the software to support standard interfaces and communications (e.g., the National Transportation Communications for ITS Protocol or NTCIP) so that you have a free choice of vendors when it comes to acquiring other components of your system. However, it is generally best not to identify the hardware platform or operating system for the system, unless absolutely necessary, so as to not constrain the design unnecessarily.
- Require the contractor to document the overall organization of the system as a contract deliverable early in the development process. This can be done regardless of whether a build or buy approach is used and should happen before the software is built or bought.
- If this is a custom implementation, include contract language that allows you to review and approve the vendor's approach. This will help you judge whether the vendor is properly addressing your future needs and may surface problems with your requirements. (If you will be responsible for maintaining the software, review the vendor's products to verify that you will be able to do so.) If you retain software technical expertise on your team in addition to the contractor, those experts should be involved in the reviews.
- For off-the-shelf software (See *Build/Buy Decision(s),* chapter 10.), determine how much "tailoring" may be required. This can include work that needs to be done to interface the software to other components, or to customize the software to your local environment. Verify that you have the ability to do the necessary tailoring.

When should the requirements be documented?

Almost everyone agrees that the requirements should be documented and stabilized before system design or development takes place. In general, if you're going with an off-the-shelf buy, then high-level requirements (perhaps a features list) will be needed *before* vendors are formally solicited. At the other extreme, if time-and-materials contracting is used to develop the system from scratch, then the requirements should be collaboratively developed with the contractor *after* contract award. However, beyond that there does not seem to be a consensus as to what point in the acquisition the requirements document should be written. Here are some of the options:

Option: The customer documents the requirements early in the life cycle. The requirements are included in the RFP. Since transportation engineers are generally inexperienced in writing these types of requirements documents, this activity calls for heavy involvement of the software expert on the team.

Pros: Allows the bidders to understand the system that they are bidding on. The more specific the requirements are, the more clear and more focused will be the bidders' responses. This, in turn, makes source selection easier for the customer.

Cons: There is concern from the private sector that when customers specify requirements, they inevitably specify the design instead. This almost guarantees that a custom-built approach will be needed and precludes existing off-the-shelf solutions. Also, it may result in a premature selection of the technology, which becomes obsolete or constraining when the system is implemented. There is also a danger that the customer will view the requirements as being "final" and will not carry out a requirements management process.

Option: This is a variation on the preceding option, with similar pros and cons. The customer documents the requirements early in the life cycle. But in this option, the requirements are included in the RFP only as draft requirements. Soon after the contractor comes on board, they will work with the customer to finalize and scrub the requirements. (Depending upon the contracting mechanism, the requirements may have to be finalized as part of the contract negotiations process, before the contract is signed.)

Pros: Allows the bidders to understand the system that they are bidding on, while explicitly acknowledging the need for collaboration on refining the requirements.

Cons: Same as in preceding option, except that the requirements will not be viewed as final.

Option: The contractor develops the requirements soon after contract award. The RFP has only a high-level set of requirements. These may include some combination of a features list, system needs, system concept, or identification of the system environment including interfaces to legacy systems. Time and materials contracting and task-ordered contracts are especially appropriate for this option. If a task-ordered contract is used, requirements development may be the only task that is initially funded. When this task is completed satisfactorily, the customer and contractor will have a mutual understanding of the requirements. Then the optional tasks for the other software development activities (design, coding, testing, etc.) are funded. Under this option, the RFP contains such items as an operational perspective and a general scoping document for the potential contractors to bid against.

Pros: Enables those with software project experience and expertise to write the requirements. Once the requirements are mutually understood, there is a sound basis for scoping the remaining tasks.

Cons: Does not provide a firm basis of understanding for the contractors to bid against. The contract must be carefully constructed to accommodate the uncertainties. A firm, fixed-price contract for the entire development effort would be inappropriate. (See *Selecting The Contracting Vehicle,* chapter 11.)

Option: An RFP is issued for developing the requirements. Then these requirements are included as part of a second RFP to implement the system.

Pros: Fills the void created by lack of customer software expertise. As with the first option, allows the bidders on the second RFP to understand the system they are bidding on.

Cons: If the contractor who writes the requirements is later allowed to respond to the RFP for implementing the system, this creates a conflict of interest. The contractor may write the requirements in such a way that only his system can

effectively meet them or unfairly position themselves for the second contract. On the other hand, if the contractor who writes the requirements is precluded from implementing the system, (or if another contractor is selected to implement the system) then there will be multiple interpretations of what the requirements mean. There is also the danger that a consultant with strong transportation credentials but lacking the needed software expertise will be chosen to write the requirements.

In short, the option you select will to a large extent define your contracting approach. There is no easy answer here, or one right solution. It is up to the customers to assess the situation from the perspective of their acquisition. Hopefully, the options presented here will help you proceed with "your eyes open" as to the implications of the chosen approach.

However, regardless of your decision, you will need to remain actively involved with the requirements for the duration of the project. Although it would be nice to set aside the requirements and go on to other tasks once the requirements are documented, you unfortunately cannot dismiss them as a "done deal." Ignoring the requirements will risk the success of the entire project.

9B. Requirements Management

"...it is necessary to allow for extensive iteration between the client and the designer as part of the system definition." — [Brooks, 1987]

"Requirements are not completely known at the start of a system's development. They cannot be specified completely up front in one voluminous document, but rather will evolve during the analysis phases of a project and beyond." — [Christel and Kang, 1992, page 14]

The need for requirements management

Fundamental flaw of software acquisition: "One can specify a satisfactory system in advance, get bids for its construction, have it built, and install it ... this assumption is fundamentally wrong." [Brooks, 1987]

A complete set of well documented requirements would be ideal, but is probably unachievable. At one time it was thought that the key to a successful software project was to i) develop a rigorous, complete set of requirements; ii) freeze them for the entire project; and iii) insist that the contractor meet all the *shall's*. However, this ideal does not jibe with real world project experience. "The demand for firm and unchanging requirements is mostly wishful thinking." [Humphrey, 1989, page 25] "The initial requirements are often wrong and will change." [Humphrey, 1989, page 26] At a minimum, the requirements will have to be clarified.

Why is this so?

It turns out that English is an imperfect vehicle for documenting requirements. Requirements documents are notorious for being vague, inconsistent, ambiguous, incomplete, untestable, and just plain wrong. The writer and reader of a requirement can have honestly differing interpretations of it. As a result, the requirements are subject to multiple interpretations. The requirement whose meaning seems so clear to the customer who wrote it may be unclear or subject to an entirely different interpretation by the software developer who reads it. If these different interpretations are not resolved early in a project, the misunderstandings will eventually lead to mistrust between the two.

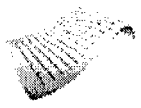
"Rule 1 of Systems Integration: The agency and the integrator will never interpret the functional definition in the same manner." [Phil Tarnoff]

Even if a totally clear, perfect requirements document not subject to multiple interpretations could be written, requirements can (and often do) change over the course of a project. As the acquisition progresses, new insights are gained and the requirements must change to accommodate them.

Instead of freezing the requirements, the management approach must take these realities into account. Our recommended requirements management approach is discussed in the following paragraphs.

Obtain vendor feedback

After the requirements are documented, formally or informally float draft copies past potential vendors, seeking their feedback. (You may want to consult with the contracting and legal expertise on your team to ensure that this is done in a legal manner that does not show favoritism to particular vendors.) Ask them if they can meet the requirements and what changes they would recommend. Re-evaluate your needs based on the comments you receive. In this way, your requirements are likely to be more reasonable. If you find there are no commercial products available to meet certain requirements, ask yourself whether these requirements are truly necessary and what the risk would be in developing software from scratch to meet them. Perhaps there is a reason that no one has implemented them. You can also use peer agencies—perhaps those in other jurisdictions who have previously procured a similar system—to peer review your requirements document.



Vendors may be reluctant to give honest appraisals of your requirements. Their responses could be interpreted as admissions of inadequacies in their products. They may fear that by being honest, they would, in effect, preclude themselves from further consideration in a competitive procurement. The Department of Defense found this can be overcome by having the comments submitted anonymously. When this was done, vendor responses improved and were much more valuable.

A pre-bid conference is another way to practice open communications for improving the requirements. Hold this conference after draft requirements are circulated but before a final RFP is issued. At this stage, contractors are generally willing to make constructive comments and suggestions, since they won't be revealing their proposal approach to competitors. Similarly, customers are still in a position to make changes, since requirements have not yet been "finalized". Once the final RFP is issued, communications are necessarily more constrained on both sides.

Obtaining vendor feedback and improving the requirements is not the end of the requirements management process.

Themes

While discussing requirements, one interviewee from the ITS community told us . . . "A formal document is not a substitute for teaming." Requirements management will be carried out by the expanded team that now includes the contractor.

The requirements document is a good starting point for establishing communications with the software developer or product vendor as to your needs. However, it is not a substitute for the primary means of communication that will take place: human communications and interactions through meetings, work groups, prototypes, system demonstrations, and system tests. Knowledge and new insights will be gained as the project progresses. This knowledge must be captured, rather than prohibited by a static requirements document. [Christel and Kang, 1992] Plan to have on-going collaboration with the contractor and revisit the requirements throughout the remainder of the project. To be sure, changes to the requirements must be carefully controlled. (See *Establish a stable requirements base to work from,* below.) However, you cannot take the requirements, "throw them over the fence" at the contractor, and forget them.

Conduct a requirements walk-through

Hold a requirements walk-through with the software development contractor when the requirements document becomes available. Having the right people present at the walk-through is key to its success. The other members of your customer team should also participate. On the contractor side, make sure that people who are responsible for the coding are present.

One of the mistakes that have been made on past projects is that the software developer's project manager is present at the walk-through, but the individuals responsible for coding the software are not.

The project schedule, RFP, and contract should explicitly call out this activity. Plan for it to take place early in the development process before software design or coding activities begin. The walk-through may even have to take place before the contract is signed; depending upon the contracting vehicle, afterwards may be too late.

At the walk-through, go through the requirements together, line by line. Because a customer and contractor approach the requirements from different perspectives,

differences in interpretation are inevitable. Discuss your respective interpretations of each requirement and reconcile your differences. Also discuss the implications of the various requirements and their impacts on the project.

Themes *This is an example of our themes on the need for **open communication** between customer and contractor and the need for them to work together (**collaboration**).*

During the walk-through, the requirements should not be viewed as being cast in stone, even if they are part of the signed contract. Checklist 9-2 identifies what should take place at the walk-through; you may identify additional items to include on the agenda.

You can also use the requirements walk-through as another opportunity to scrub requirements. Once the requirements have been clarified, go through them again with the contractor and prioritize them by their desirability and the cost or effort needed to implement them. Accordingly, Checklist 9-2 also identifies what the scrub session can be used to accomplish. As shown, in the checklist, deferring low priority requirements is one of the options to consider.

Walking through the requirements and scrubbing them is important whether you build your system or buy it. For off-the-shelf products, a walk-through provides the opportunity to identify which modifications are easy to make and which ones are difficult. The product supplier is in a better position than anyone else to know this. Intuition of outsiders often fails: a seemingly innocent change can be difficult to implement; a seemingly ambitious one may acutally be easy.

Themes *By using incremental development and keeping the size of the project builds small (**bite-size pieces**), users are more willing to defer capabilities into the next release of the program. If they see a long development cycle, they will try to cram as much capability as possible into the initial release. In that case, the problem exacerbates itself.*

You may object that such a thorough walk-through will take considerable time. It will. However, the time spent will be well worth it. Again citing the advice of an ITS manager, "Don't short change this phase." If requirements issues are not addressed at this point, they will surface later during design, integration, or operational phases of the project. At that time they will be considerably more costly to correct. More costly in time, dollars, and frayed working relationships. The conventional wisdom is that if you delay fixing a requirements problem for one phase of the life cycle, the cost to correct the problem goes up by a factor of ten. Correcting a requirements problem after design can cost ten times more than correcting it during a requirements walk-through. If you wait until after operational use of the system begins, you can incur a hundred-fold increase over what it would have cost to correct earlier.

Checklist 9-2. Suggested Agenda Items For A Requirements Walk-Through

✓	Clarify ambiguous or vague requirements.
✓	Remove inconsistencies between requirements.
✓	Supply missing requirements.
✓	Replace existing requirements with better alternatives that are identified.
✓	Eliminate unnecessary or hard-to-meet requirements, or mark them as low priority.
✓	Prioritize the remaining requirements.
	Scub session
✓	Eliminate low priority or high cost requirements; "retain only those that are absolutely necessary."
✓	"Simplify all requirements that are more complicated than absolutely necessary."
✓	"Substitute cheaper options" when they are available.
✓	Defer lower priority requirements into later versions of the software.

*Quotations taken from S. McConnell, *Rapid Development: Taming Wild Software Schedules,* Microsoft Press, 1996, page 329.

Suppose a requirements walk-through is not held? What happens if agreements are not reached on the requirements?

Probably nothing at first. But when requirements problems do become visible (and they will!), they ultimately manifest themselves as a system that is unsatisfactory in some respect. The system will fail to meet customer expectations. ("I thought the system was going to do...") And unmet expectations mean a disappointed customer. In the long run, that can't be good for anyone involved in the project, including the contractor.

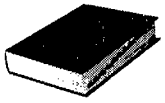
One ITS software developer cites the example of an unnamed customer who refused to carry out a requirements walk-through. So the contractor proceeded as best they could in designing the system. Then came the critical design review, a major milestone. But instead of addressing design issues, the review quickly back-tracked to the unaddressed requirements issues. The contractor and customer finally reached a mutual understanding of the requirements, but not without cost. By then, much of the previous design work had to be discarded and re-done. This could have been avoided with a timely walk-through of the requirements.

Sign the requirements and place them under configuration control

Document any agreements reached during the walk-through. In many cases, a memorandum of understanding between you and the contractor may be sufficient to record the clarifications and understandings that were reached. In some cases, the requirements document may have to be changed. (Interviewees did not find that this presented any legal obstacles even when requirements were a formal part of the contract. This is an area that should be explored ahead of time with the contracting officer who is a member of the customer's team. It may be that a walk-through will need to be held before a contract is signed.)

Once all the requirements have been agreed upon, both parties (customer and contractor) should sign the document and any memoranda of understanding. (Remember to include contract language for this to occur.) It is desirable to have the end user member of your team be one of those signing the requirements. This documents their agreement and establishes their buy-in. Then baseline the requirements. From this point on, requirements changes should be incorporated only in a controlled (formal) manner. [Ferguson, 1996, page L2-7] (See also *Software Configuration Mangement,* chapter 18.)

A document or piece of software is said to be ***baselined*** *if procedures are in place so that the current version is identified and controlled at all times.*



Formal *implies that the customer and contractor have to agree on the change using a defined process.*

Okay, so I don't freeze the requirements at contract award. But now that they're under configuration control, can we put them aside and go on to other things?

A baselined requirements document is not the end of the story. You will still need to address requirements issues as they arise as part of the requirements management process.

Two opposing needs: address requirements issues as they arise and establish a stable requirements base to work from

The following paragraphs address two needs. On the one hand, there must be the flexibility to address the requirements issues that will inevitably arise. On the other hand, there is the need to establish and work from a stable base of requirements. Walking a tight rope and balancing between these two, somewhat opposing, needs will be a key factor in the success of your project.

Address requirements issues as they arise

> *"Myth: Once the agreed functionality has been documented and signed, the user can relax and let [the developer] build the system."*
> *"Fact: The user ... should be in almost continuous contact with [the developer]." — [Farbman, 1980]*

Even after formal sign-off takes place, the requirements are not frozen. Placing the requirements under configuration control should *not* be equated with freezing them for the duration of contract. Some changes are inevitable. Even if the requirements document is part of the signed contract with the contractor, there is still the need for some flexibility. It is natural and prudent to take advantage of insights gained as the project progresses. You may also find the opportunity to exploit advances in technology. You will need to address requirements issues as they arise. Most often it will mean only clarifications or agreements, perhaps documented in a memorandum. In some cases this may mean changes to the requirements. Such changes, whether proposed by the developer or requested by the customer, are normal and should be treated as such. (Again, the need to have flexibility in the contracting process. It is best to have a process that minimizes the number of contract change orders.)

However, forcing a contractor to interpret the original requirements as incorporating *bona fide* additions and insisting that they do so within the original schedule and budget should not be a cost of doing business. It only destroys trust and leads to adversarial relationships that are not good for either party or for the system. Instead, collect the list of additions and use them to define additional releases of the software. That way you minimize on-going development, while planning and budgeting for new features. But be sure to keep the additional stages small and of manageable size.

Whereas walking through the requirements is a one-time, scheduled event, addressing requirements issues as they arise will be an on-going process. Requirements will need the contractor's attention and yours throughout the development process. As a customer, you should be particularly attentive when the software development contractor raises

requirements issues or suggests changes to the requirements. This does not mean blindly relaxing requirements or accepting any changes suggested for them. It does mean entering into open and honest discussions.

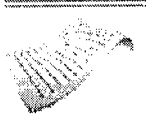
When the contractor raises a requirements issue, do not respond with an attitude of "Back to requirements again? I thought we were finished with them." Not only should you be receptive to addressing requirements issues, you should actually foster an atmosphere that *encourages* the contractor to bring them forth.

Suppose the contractor seeks clarification about an unclear or ambiguous requirement. This is a positive sign that contractor personnel are doing their job correctly. Respond accordingly: "Thank you for bringing that to our attention. Let's address it now."

Suppose the contractor has identified the need for a new requirement. Perhaps in designing the software, they have uncovered an area in which the requirements are incomplete. There must be flexibility to allow the requirement to be added to the baseline system. The implications of adding the new requirement should be jointly considered by you and the contractor. You will need to explore tradeoffs and alternatives together. Maybe you will jointly agree to eliminate a lower priority, pre-existing requirement. This will compensate for the new requirement while maintaining schedule and cost. Or perhaps you will agree that the schedule must slip. Maybe, the best of all worlds, the contractor will be able to say, "Since we agreed to this change at this early stage in the project, we'll be able to accommodate it in our design without any cost or schedule implications." And don't forget to have the relevant members of your customer team present for the discussion.

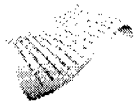
Sometimes the contractor will discover that a certain requirement is going to be more difficult to meet than originally anticipated. Again, such news should not be met with a "shoot the messenger" approach. The news may be bad, but the earlier you find out about it the better. In such a case, you may have to relax the requirement to live within schedule constraints. Or if the requirement is essential, you may have to forego other requirements to accommodate it. As you make your decisions, recognize that you may not be able to get everything you asked for: insisting on everything may have significant cost or schedule impacts, or it may preclude the use of an otherwise perfectly acceptable off-the-shelf solution. Remember that the goal is to solve problems and build a system; not to fix blame.

Themes *The contracting mechanism must contain the **flexibility** to relax requirements when necessary.*



One satisfied customer told his long-time contractor "The reason we're so successful together is because you always give me 80% of what I ask for."

But whatever way the new requirement is handled, there must be sufficient flexibility to allow the change to take place. Further, the issues that are raised should be dealt with promptly, with agreed upon changes documented and adopted in a timely fashion.



One ITS software developer complained about an unnamed customer. "We wrote questions and . . . couldn't get them to respond. [We] couldn't make progress."

Why not just leave the decisions up to the contractor? After all, you hired them for their software expertise. The reason is that the members of your customer team are often in the best position to assess the operational impact of a change. Your domain experts are the ones with the operational perspective. Who better than the domain experts to clarify the intent of an ambiguous requirement? Without your team's inputs, a computer programmer will be the one who is forced to make the decision during the coding process. The programmer may be a "computer wiz" but undoubtedly lacks the traffic or transit perspective needed to assess the operational impact of the change. The programmer can only consider its impact on the internal workings of the software. This perspective should be brought forth in the discussions. But it should not be the sole basis for making decisions.

Themes

*Here is where having the right expertise on the customer team comes into play (**team building**). The end users of the system and domain experts are often in the best position for assessing the impact of proposed changes. After all, it is their day-to-day working lives that will be affected. They have the operational perspective that the project manager may lack. And they will feel better about using the eventual system if they know that they had an input into shaping it.*

Establish a stable requirements base to work from

Notwithstanding the discussion in the previous paragraphs, resist the temptation to add or make changes in the requirements and minimize the number of changes you do make. Having a stable set of requirements is essential for successful software development. That is why we recommend having requirements signed and baselined; it provides some stability to the acquisition. Ever-changing requirements lead to cost and schedule overruns as rework is induced with each change. The government continually changing requirements during the course of development is one of the most common reasons for over-running cost or schedule. The authors of this document have personal experience with several such systems. Scope or requirements creep should be resisted whenever possible. This is why we recommend having the requirements signed, to provide some stability to the acquisition. As noted previously, proposed changes or additions to the requirements can be examined in the light of the system concept documented in the project plan. Ask yourself, "Does the proposed change support the concept? Or is it tangential to the real objectives, the *why's* for the system?"

Coming full circle in our argument however, stability in requirements should not be an excuse for rigidity.

"Good" changes, "bad" changes

How do you balance the conflicting needs of addressing requirements issues and stabilizing the requirements? One way may be to ask yourself whether a proposed change is "good" or "bad". Act accordingly in deciding whether to accept or reject the change.

"Good" changes	"Bad" changes
clarifies ambiguities	increases system functionality
simplifies the design	adds new requirements
derives lower-level requirements in support of high-level requirements	increases the scope of the project

Another way to assess the desirability of proposed changes is to increase the threshold for making the change as time progresses. Initially, there can be more give and take. But once a requirements baseline is established for a system increment, then the "pain level" for making changes to that baseline should be high. Although not rigidly frozen, baselined requirements should be carefully controlled. Scrutinize changes in regards to their necessity and for their impact on scope, cost, and schedule. The software engineering expertise on your team can assist in making these determinations.

In short, treating the requirements as a non-frozen, living document should not be mistaken for treating the requirements as fluid. As usual, good engineering and management judgment is needed to achieve the proper balance.

Use formal procedures and sign off on them

One ITS manager recommends that before you address requirements issues, you first establish "some rules of engagement". By this, she means making prior agreements as to how the customer and contractor will work together to adopt changes to the requirements. Things such as, "How will agreed upon changes be formally documented?" "How long does the customer have in responding to issues raised by the contractor?"

She cites what can happen where there are no "rules of engagement" in place. Without them, there is the danger that a brainstorming idea can be misinterpreted as a formal commitment. Perhaps the customer and contractor attend a meeting together, in which they discuss some possible changes to the requirements. From the contractor's perspective, this was just a brainstorming session, and so the proposed changes are quickly forgotten. From the customer's perspective, the contractor has tacitly agreed to make the changes. So the customer has false expectations that the changes will be incorporated. Later on, of course, the customer is faced with unmet expectations: a system that does not have the "agreed upon" changes. Accusations of not living up to agreements will soon follow. In the meantime, the contractor complains that the

customer wants more and more for free. This leads to mistrust and jeopardizes maintaining open communications over the long term.

Some suggested "rules of engagement" are:

- No changes take effect until they are formally documented and signed off by both parties. Oral agreements are not sufficient to effect a change. Of course, this presupposes a non-bureaucratic procedure. Having to formally re-negotiate an entire contract with both parties' contracting offices for every change is not acceptable. (See also *Software Configuration Management,* chapter 18.)
- All proposed changes are discussed in regards to their impact on scope, cost, and schedule. This discussion takes place in the context of possible features trade-offs.
- Reach agreement as to how long the requirements document will be maintained as a living document. As the project progress, the requirements document may outlive its usefulness. There would be no point in going through a bureaucratic exercise of changing a requirements document if no one will refer to this document again. On the other hand, if you decide that the requirements document will be the reference for the system throughout operations and maintenance, then by all means incorporate any changes in it. In some cases, the requirements document can serve as the basis for future changes or replacement systems. For this reason, some recommend that the requirements document be kept as a living document throughout the life of the system.

To ensure the requirements management process works, the customer and the software development contractor each have certain responsibilities. Table 9-1 summarizes them. Customer and contractor responsibilities are juxtaposed. Each row gives a customer responsibility and its "flip side", the corresponding contractor responsibility. For example, you should resist making changes to the requirements, but the flip side is that the contractor should not reject all changes out of hand.

> *"Myth: The user must tell [the developer] what he needs. You [the developer] document it. He signs it. All changes to applications are indications of the user's failure to know what he wants."*
> *"Fact: Most users don't know what they want; they almost never know what they need. Your [the developer's] job is to ... suggest alternative solutions ..." [Farbman, 1980]*

The responsibilities shown in the table can also serve as the basis for the "rules of engagement". Discuss them with your contractor to achieve an understanding of your mutual expectations and obligations.

Table 9-1. Responsibilities for Requirements Management

Responsibilities for Requirements Management	
Customer Responsibilities	**Contractor Responsibilities**
• Maintain open communications with the contractor. Work together as a team and communicate on a regular basis.	• Maintain open communications with the customer. Work together as a team and communicate on a regular basis.
• Discuss the requirements with the contractor in a non-threatening, non-recriminating manner. Remember that the contractor's differing interpretations may be honest ones.	• Discuss the requirements with the customer in a non-threatening, non-recriminating manner. Remember that the customer's differing interpretations may be honest ones.
• Sign off on the requirements once a mutual understanding has been reached with the contractor.	• Sign off on the requirements once a mutual understanding has been reached with the customer.
• Remember that proposed changes do not take effect until signed off by you and the contractor.	• Follow configuration control procedures in updating the requirements. Do not implement changes until they have been signed off by you and the customer
• Recognize that it may not be possible to have everything. Some requirements may have to go unmet or be deferred.	• Do not take a legalistic view of the requirements by narrowly satisfying the *shall's*, while ignoring the true needs of the customer. Avoid the attitude of "they didn't ask for it, so we didn't provide it." Try to satisfy the intent of the requirements. A satisfied customer should be your goal.
• Resist the temptation to change the requirements or add new ones. Avoid "requirements creep."	• Do not be too rigid in rejecting changes proposed by the customer. Not all changes are invalid. Recognize that some change is inevitable.

Table 9-1. Responsibilities for Requirements Management (Concluded)

Responsibilities for Requirements Management	
Customer Responsibilities	**Contractor Responsibilities**
• Foster an atmosphere that not only allows, but encourages, the contractor to bring forth requirements issues as they arise. • Be receptive to requirements issues that are raised and changes that are proposed by the contractor or other members of your team.	• Bring forth requirements issues. Point out problems as they arise; do not hide or attempt to bury them. No cover ups. • Identify requirements that are causing particular difficulties. These may be high risk requirements that are technically difficult to implement, ones that drive the design or cost of the entire system, or requirements that present schedule or budget risks. • Use your software expertise to educate the customer as to what to expect. Point out what various requirements imply with respect to risk, system usability, functionality, etc.
• Have an open mind. Show a willingness to be flexible. Do not become wedded to bad ideas. This does not mean blind acceptance of all proposed changes. • Work with the contractor to find alternatives.	• Suggest alternatives that would better meet the true needs of the customer or would save money. • Work with the customer to find alternatives.
• Be responsive. Answer questions and respond to issues that are raised in a timely manner. When changes are agreed upon, adopt them in a timely fashion.	• Recognize that the customer has the final say in accepting or rejecting your proposed changes.
• When requirements change, or new requirements are necessary, do not attempt to squeeze the contractor by insisting the new requirements be added within the framework of the existing schedule and budget. Recognize that the contractor is in this to make a profit, not as a public service. • If requirements change, or new requirements become necessary, discuss how they will be accommodated. In some cases, they may have to be substituted for lower priority requirements; in other cases, schedule or budget relief may be necessary.	• Do not try to gouge the customer by viewing a requirement change as an excuse to "get well" on an underbid project. Some requirement changes can be easily folded into the design and accommodated without schedule or budget impact. Others may actually have positive impact, lowering the contract costs or shortening the schedule.

Develop human interface requirements using rapid prototyping techniques

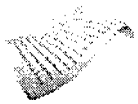
> *"For the truth is, the client does not know what he wants. ... The client usually does not know what questions must be answered, and he has almost never thought of the problem in the detail necessary for specification." [Brooks, 1987]*
>
> *"The customer doesn't generally know what is needed and neither does anyone else!" [Humphrey, 1989, page 26]*

Our previous discussion emphasizes that the customer and contractor resolve their differing interpretation of the requirements at a walk-through. This presupposes that the customers have an interpretation that they can bring to the table, that they know what they want. In fact, they generally do not. Throughout the software engineering literature, we are warned that part of the difficulty in documenting requirements is that customers don't really know what they want. Several of our interviewees who had participated on ITS procurements also acknowledged this truism. That is one of the reasons why we've been emphasizing the need for continual and direct customer involvement in revisiting the requirements as the project proceeds.

"Not knowing what they want" is especially true for the human interface requirements.

*The interface that provides human interactions with a computer system is sometimes called the **GUI**. That stands for "Graphical User Interface" and is pronounced goo' ee.*

It is almost impossible for users to generate the exact requirements of a system before they have tried out some version of it. A static written document is a poor communications medium for describing the highly interactive nature of an ITS system. It is hard to visualize how written requirements will manifest themselves or how interactions with a system will "feel" to the user.



A few ITS examples illustrate what can happen when human interface requirements are specified on paper. In the transit arena, a box on a bus needed multiple keystrokes for a simple function like changing the volume control. This was not apparent from reading the written requirements and was not realized until the box was used operationally. In the traffic arena, incident reports could not be filed until all the fields of an on-line form were filled out. Many of the fields were not particularly important and filling them out delayed the transmission of critical information. But the requirements did not specify the capability to transmit a partially filled out form or allow the ability to retrieve a form and add the missing fields later. Because rapid prototyping techniques had not been used in either case, it was not possible to visualize the implication of the written requirements. Only real-world interactions with the system revealed the flaws that were inherent in the requirements. If you buy existing products, you will at least gain the benefits of someone else's experiences.

For these reasons, it is best if the requirements do not go into too much detail in this area. Instead, rapid prototyping is the technique of choice for defining the human interface requirements. (See the *Rapid Prototyping* topic sheet.) The end users will need to work

closely with the software developers. Basically, the developers will try out different human interfaces and allow the users to "try it out", make suggestions, and iterate until an acceptable solution is found.

Be sure to include funds in the contract to cover rapid prototyping activities. Do not expect the contractor to bear the costs.

Themes

*Several of our recurrent themes come into play. First, rapid prototyping requires **open**, non-threatening **communications** between customer and contractor. Second, the two should work together (**collaboration**) as a team (**team building**) to address requirements issues. Finally, there must be the **flexibility** to incorporate the requirements developed through rapid prototyping into the system.*

The use of rapid prototyping does not mean that the human interface can be completely ignored in the requirements document. Define human interface requirements broadly, with further definition deferred as an activity for the customer/contractor team. For example, you can document the numbers and types of users. At first, the operators who monitor traffic or schedule buses may be the only users who come to mind. But more careful analysis will show that there are other types of users as well: those who administer the system to keep it running, managers who read computer-generated reports, technicians who repair the system.

General human interface capabilities should also be documented. For example, it would be appropriate to include a functional requirement that the system accept incident reports filed by an operator. However, do not include details on the keystrokes, screen layouts, etc. that are to be used for achieving this function. Instead, have the requirements document reference the prototype, letting it serve as a "living document." (Of course, this assumes you have contract and funding flexibility to incrementally define and refine the requirements.)

Perhaps what we're really saying is that the human interface is just another example of "specifying the requirements, not the design," the *what's* not the *how's*.

- *Develop a good set of requirements. It is one of the most important things that you can do on a software acquisition.*
- *Have the various members of the customer's team participate in developing the requirements.*
- *Document the requirements in a formal configuration-controlled document.*
- *Develop functional and performance requirements (the "what's") and not the design or technical requirements (the "how's").*
- *Scrub the requirements to avoid asking for too much. Avoid requirements or scope creep.*
- *Address quality factors and the ability of the system to accommodate anticipated changes.*
- *As soon as possible after contract award, hold a requirements walk-through with the contractor and other members of your team. Then sign the requirements and place them under configuration control. Make sure the contract calls for these activities.*
- *Establish a stable base of requirements. It is essential for the success of your project.*
- *Address requirements issues as they arise, as part of the on-going requirements management process.*
- *Flesh out the human interface requirements using rapid prototyping.*
- *Use the requirements as the basis for size, schedule, and cost estimates; build versus buy decisions; design and development activities; and acceptance testing.*

CHAPTER 10
BUILD/BUY DECISION(S)

"The most radical possible solution for constructing software is not to construct it at all."— [Brooks, 1987, page 16]

"Where use of an existing component is both possible and feasible, it is no longer acceptable for the government to specify, build, and maintain a comparable product when a commercial solution is available." — [Carney and Oberndorf, 1997]

When acquiring software, one of your most important decisions is whether to "build" the software (i.e., develop a custom system) or to "buy" already developed software.

Themes *This chapter amplifies our* ***don't build if you can buy*** *theme.*

The build/buy decision is not necessarily an either/or choice; you may end up doing some of each. Even if your overall system is a custom build, some of it could be constructed with off-the-shelf products. The compilers used during software development and the underlying, low-level components that implement communications protocols (e.g., TCP/IP drivers) would be examples. In fact, most ITS systems will be a hybrid, having some off-the-shelf components, some tailored commercial products, and some custom-developed software. It's the degree to which each of these is appropriate that is at issue.

Kinds of off-the-shelf software

The term "off-the-shelf" is used to cover any software that was developed outside of your own organization, i.e. software that you "buy" rather than "build". It comes in many varieties. "Shrink wrapped" software (e.g., word processors) is developed by large commercial software vendors. For ITS, some of the underlying components (database management systems, communications protocol drivers, software development tools, etc.) may be shrink wrapped. Be especially wary of building software components that are most commonly bought (word processors, compilers, operating systems, database management systems, geographic information systems, etc.). If you find yourself in this situation, carefully review the rationale for building the software and the requirements that constitute the basis for this decision. Finally, if you decide that you must build this kind of software, hire specialists in that area to build it.

For most of the ITS systems shown in figure 4-1, off-the-shelf product offerings are available, often from more than one vendor. These offerings range from almost commodity products (e.g., traffic signal control for isolated intersections; transit fleet tracking) to components that would be integrated into a larger system (e.g., variable message sign control). However, for the most part there are currently no off-the-shelf

products that integrate different types of ITS systems (e.g., traffic management with transit management) or that integrate peer systems in neighboring jurisdictions.

Build/buy decision factors

> *"Using [off-the-shelf] components in any given circumstance may either be beneficial or it may cause greater problems ... choosing an [off-the-shelf] component may be a reasonable solution; however the decision ... should be the product of analysis, reasoning, and engineering decisions, not the desire to jump on the latest bandwagon." [Carney and Oberndorf, 1997]*

There are a number of factors that go into making a build/buy decision, including:

- The availability and cost of off-the-shelf software that would satisfy your needs (e.g., meeting functional and performance requirements and required standards).

An over-specified system will unnecessarily preclude satisfactory off-the-shelf solutions.

- Local phenomena may lead to unique local requirements that must be addressed by the software. These will have to be accommodated through customization of off-the-shelf products or through custom-built software. However, give careful scrutiny to unique requirements and examine the need to be different.

One city has unique requirements for their freeway management system to interface with water pumps in highway underpasses.

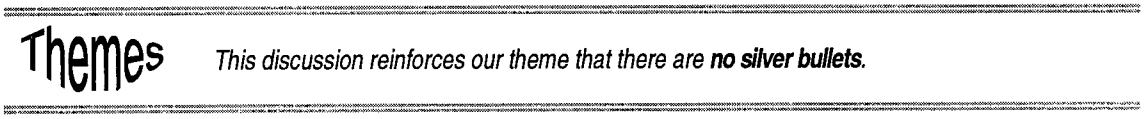
- The extent of modification, customization, and integration required.
- Whether the software operates in your system environment. However, beware of prematurely specifying the environment or over-specifying it, as this will also unnecessarily preclude satisfactory off-the-shelf solutions. (See *Identifying The Software Environment,* chapter 12.)
- Interface capabilities of the software to other ITS components and subsystems. Are interfaces of available products open and documented or are they closed (i.e., proprietary and undocumented)?
- Whether the software fits budgetary constraints, including costs of initial purchase, modification and integration, and continuing license and maintenance.
- The availability and cost of internal and external developers who are knowledgeable about intelligent transportation functions and needs.
- The availability of product documentation that meets your needs.

- Your organization's plans and abilities to maintain and enhance the software. (See also the *Software Maintenance* section in *Training, Operations, and Software Maintenance,* chapter 16.)
- Time frame in which the system must be implemented.
- The ability to incrementally expand or implement the software.

Risks associated with buying the software

Generally, buying off-the-shelf components reduces risk. On the other hand, the risks of building custom software are well known. (See, for example, *The Nature Of Software,* chapter 1.) Furthermore, buying software is very attractive, as someone else has already gone through the pain and agony associated with building it. And it is certainly true that a great deal of time and effort can be saved by using someone else's software when it will work for you in your environment.

That is not to say, however, that buying off-the-shelf software is a panacea. There are risks, but these risks are manageable. And, as vendors and consumers gain more experience with off-the-shelf software, standards emerge, and technology improves, the risks are being reduced. Table 10-1 lists some of the risks and provides some suggestions for mitigating them.



How to buy software

If you want to investigate using off-the-shelf software, first conduct a "market" survey to see what software, if any, is available that seems to meet your functional, system, and budgetary requirements and constraints. Review product literature, and discuss your concepts and needs with vendors to determine if their products fit well with your concepts. You can also check with other DOTs or transit agencies to see if they have software performing this same function.

In some cases, the job of acquiring and integrating off-the-shelf software can be simplified if the selection can be done in conjunction with other agencies. For example, transit and freeway management could jointly select the same geographic information system to ensure compatibility. However, you must ensure that the selected software meets the needs of both organizations.

Once you've decided to acquire off-the-shelf software, the major steps and considerations for selecting a specific package are:

- In developing requirements, identify mandatory vs. optional requirements. Keep requirements as functional and flexible as possible, focusing on what is needed rather than how to do it.
- Select an evaluation team and prepare a schedule.

Table 10-1. Risk Areas for Buying Off-the-Shelf Software and How to Mitigate Them

Risk Area	How to Mitigate the Risk Area
• Cannot satisfy requirements perfectly.	• Attitude of "compromise requirements with reality"--the 80% solution may be good enough.
• Ability of the software to meet vendor claims; true vs. advertised capabilities.	• At a minimum, require live, hands-on demo. Preferably, use the demo or borrow a product to confirm value of software before purchase.
• Poorly debugged product. In worst case, may not be usable or even installable.	• Be skeptical about new releases or version 1.0 of a product; ask for evaluation copy to test.
• Integration and test risks--interfaces may be closed, undocumented, and proprietary.	• Test evaluation copies of all products under consideration in combination with each other before making a final decision to buy them. • Obtain documentation on the interfaces. • Do not over-specify the environment.
• Poor or non-existent documentation.	• Understand what types of documentation are included in the license. • Examine documentation before purchase.
• Vendor lock-in. (Note: this is also a risk area for custom built software.)	• Use open systems based on standards. • Use open data formats that allow external processing of data files.
• Vendor will stop supporting the product or go bankrupt. (Note: this is also a risk area for custom built software.)	• Consider the financial stability of the vendor and the market penetration of the software package. Place products in escrow (admittedly an imperfect solution).
• Shipping dates.	• Be circumspect about building schedules that rely on future shipping dates for software; rely on released products.

Table 10-1. Risk Areas for Buying Off-the-Shelf Software and How to Mitigate Them (Concluded)

Risk Area	How to Mitigate the Risk Area
• Off-the-shelf software can change quickly. • Realizing new features in an upgrade of one product may force you to upgrade another product as well.	• Plan for change. Have a strategy for incorporating new releases of the software by the vendor. • Plan for ongoing license, maintenance, and administrative costs.
• May be forced to purchase unwanted functions bundled with the desired ones. May be forced to install, use, work around unwanted functions.	• Look for modular software with separately priced parts, looking for severable features and extensible products.

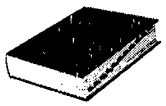
- Identify evaluation criteria and selection factors and priorities, including degree to which package must satisfy requirements and the relative importance of function vs. budget vs. schedule. Criteria can be application-specific, or they can be generic issues such as adherence to standards or vendor upgrade policy.
- Make a matrix to determine which products meet which requirements. (See table 10-2.) Include functional requirements and performance requirements. (An example of the latter in a transit management system would be the capacity to track 100 transit vehicles simultaneously.) Depending upon the requirement, it can be scored on a yes/no basis or on a numeric scale (e.g., 1 to 5). Sources of information for making this determination include: product literature, vendor-supplied information, product demonstrations, and use of evaluation copies of the software. [Oh, 1993; Reed, 1994.]

Table 10-2. Sample* Matrix for Evaluating Off-the-Shelf Products

	Product A	Product B	Product C
Mandatory Requirements			
• Requirement #1 • Requirement #2 • • • •			
Other Criteria			
• Life cycle costs • User interface • Data rights • Licensing • Upgrades • Security • Training • Vendor stability			

*Matrix shown is a sample only. Some of the rows shown under "Other Criteria" may belong under "Mandatory Requirements" on your acquisition.

- Weed out software that doesn't meet mandatory requirements. (However, make sure that your "mandatory" requirements really are mandatory.) Be especially wary if not products meet your requirements. Perhaps they're beyond the state-of-the-art.
- Evaluate the remaining proposals more thoroughly. Hands-on evaluation of the products is preferred, but at a minimum, require a product demonstration in an environment as similar to your own as possible. Significant evaluation factors should include package flexibility, degree to which users can customize/tailor package to their needs, extent of required customization, ease of package integration, and risks associated with each package. You may want to consider vendor solvency and financial viability in your evaluation criteria. In some non-transportation project areas, it is common for state and local agencies to request financial data for the division of the companies responsible for carrying out the work.
- Obtain lists of previous customers of the vendors and talk to them. Ask about the quality of the product and vendor support. Be sure to know if they bought the same product or a different one from the vendor.
- Select a product from among the qualified offerings.
- Contract for the software; include a contracting mechanism to accommodate changes (e.g., in requirements, functionality, costs). Pre-negotiate the billing rates to ensure you have a flexible licensing agreement that takes account of geographical factors, numbers of users and upgrades.
- Have built any custom software that is needed to the integrate the off-the-shelf product into your overall system (to legacy systems, to other ITS subsystems, etc.).



Software used to integrate off-the-shelf components is sometimes called ***glueware****.*

- Use operational experience with the off-the-shelf product as the basis for requesting future enhancements. Before embarking on any such upgrades, team with the vendor to explore which changes are easy and which would be difficult to implement.
- If you go with a combination of custom development and off-the-shelf products, be sure to consider the intellectual property rights implications. Even though you may have unlimited rights to the custom software, you may need to buy additional licenses to replicate the commercial components on each user workstation. Your overall rights to the custom system may not cover these components.

Note that in most agencies, these steps will have to be carried out as part of a formal, competitive process. For example, the matrix may have to be formally scored by several agency personnel. Work with your contracting office to define the process.

In summary, the key to successful off-the-shelf buys is to look beyond the marketing brochures and understand what you are getting, and what it will take to integrate the product into your environment.

- *Consider buying your software, if at all possible, rather than building it.*
- *Consider a mix of build and buy, if buying alone does not meet project needs.*
- *The buy option is not without risk; however, the risks are manageable.*
- *Understand the off-the-shelf products and the implications of their use before buying them.*

CHAPTER 11
SELECTING THE CONTRACTING VEHICLE

This chapter contains what is probably the most controversial material in this document. There is no consensus as to what is the correct contracting vehicle for acquiring software. This stems from the fact that the various existing contracting types and approaches were created without software in mind. Consequently, none of them is particularly well suited for software. In fact, there has been some movement in the ITS community to define new contracting vehicles that explicitly address the needs of software acquisitions.

The good news is that even contracting vehicles not particularly well suited for software can be made to work. Indeed several have been made to work when the various themes of collaboration, team building, bite-size pieces, etc. were applied. There is also evidence that at least some of the problems attributed to contracting vehicles were really the result of other more important problems, such as the failure of the customer and contractor to maintain open communications. Even an ideal contracting vehicle will fail if the acquisition is not properly managed.

Although there is no consensus on what to do, there does appear to be some consensus on things that don't work—on what *not* to do. But before getting into our recommendations, let us review the contracting alternatives that are available to you.

In selecting an appropriate contracting vehicle, we need to consider the contract *type* and the contract *approach.*

Types of contracts

Three general types of contracts, which have many variations, are relevant to our discussion. These general types address how the contractor will be paid:

- *Fixed-price* contracts, sometimes referred to as "low bid" or "lump sum"
- *Cost-reimbursement* contracts or "cost plus"
- *Time-and-materials* contracts or "T&M" require that the contractor deliver specified types of labor at specified rates. Costs for materials are also reimbursed.

Within the overall framework of these general types of contracts, there are four contract types that specifically apply to Federal-aid procurements [Booz-Allen, 1997]:

- *Construction* contracts are the ones traditionally used for transportation projects and are the most familiar to state DOTs. These are usually awarded on a low-bid basis. For ITS, these may be appropriate for installing field devices or constructing the room to be used as a traffic management center.
- *Engineering and design services* contracts are generally issued to consulting firms for design work. Software development has sometimes been procured using this type of contract. Consulting contracts are a variation of this type.

- *Non-engineering, non-architectural* contracts are used for such things as procuring property.
- *Innovative* contracts encompass a range of contract types. The *design/build* contract is the most relevant to our discussion. Commonly used in domains other than transportation, it combines two types of contracts—construction and engineering and design services—into one. A single contractor takes on all the responsibility. (However, see below about the need for customer involvement.) This type of contract is intended for projects that are defined by functional or performance specifications or for complex systems that require major integration activities. ITS projects would seem to fall into this category.

  Innovative contracts can also encompass innovative contracting practices, such as the use of life cycle costing, which are encouraged under FHWA's Special Experimental Project No. 14 (SEP-14). Other types of innovative contracts include incentive arrangements and multiple phase contracts. FHWA approval is needed if an innovative contract is to be used on contracts with FHWA funding.

Contracting approaches

"Contracting approaches" refers to how one or more contracts are used in combination during an acquisition [Booz-Allen, 1997]:

- The *engineer/contractor* or *design-bid-build* approach. This is the one usually used on construction projects.
- The *systems manager* approach.
- *Design/build*, listed above as one of the types of innovative contracts, may also be considered an alternative approach to the two traditional approaches.
- *Design to cost and schedule*
- *Build to budget*

Table 11-1 weighs the pros and cons of the various contracting approaches. As can be seen in the table, none of the choices is perfect. One of the problems common to many of the approaches, is that the software is acquired separately from the rest of the system. Even if you go with existing products, buying the computing hardware or field devices on a low-bid basis without regard to software may result in the need to significantly re-tailor the software. Thus significant cost and software development risk could be incurred. In other words, computing hardware, field devices, and the software that drives them should be acquired in conjunction with one another.

What type of contract should I use for ITS software?

Now that we've introduced the various types of contracts and contracting approaches, the question remains as to which should be used for ITS software.

Time-and-materials contracts, especially if task-order based, are well suited for ITS software. They provide the opportunity for a flexible approach. Instead of defining

Table 11-1 Contracting Approaches

Contracting Approach	Description of Alternative	Advantages	Disadvantages
Engineer/Contractor	The engineer is selected using a conventional consultant procurement process that is based on qualifications and experience to perform the work. The engineer typically prepares the contract documents (plans and specifications). Construction contractors are invited to submit bids in accordance with the requirements of the contract documents. Once the bid has been awarded, the contractor builds the project per bid documents. The engineer may inspect construction and interpret bid documents. The agency is the responsible entity.	• Long history of use • Well-defined roles • Legal precedent for handling disputes • End-product well defined at early stage • Contractor manages subcontractors • Well-suited to highway construction	• Artificial line between design and construction • Not well-suited to software development work – Difficult to specify – Buyer may not know needs • Software/systems integration not usually performed by prime contractor • Contractor has financial incentive to find deficiencies in bid documents and "changed" site conditions to seek change orders • Limits customer and software developer communications when software is developed by a subcontractor.
Systems Manager	The systems manager is selected using conventional consultant procurement process (i.e., qualifications-based followed by competitive negotiation). The systems manager is responsible for design (plans and specifications), software development, hardware procurement, integration, training, and overall quality control. Equipment and electrical contracting services procured on low bid basis. System managers are often used for technology-based projects.	• Overall system design, software development, system integration, and testing controlled by a single entity • Software developer is usually prime contractor • Minimizes shifting of fault • More flexibility to allow changes than traditional approach • Well-suited to ITS projects • Avoids use of low-bid selection • Gives customer access to systems manager	• Fewer firms in marketplace with requisite blend of skills • May be unfamiliar to local engineers and procurement officials • Heavy reliance on successful performance of system manager • End-product less well defined than engineer/contractor approach • Public agency responsible for low-bid services, including their inspection and acceptance • Low-bid hardware procured without regard to software

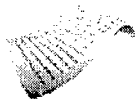
Table 11-1. Contracting Approaches (continued)

Contracting Approach	Description of Alternative	Advantages	Disadvantages
System Integrator	Same as system manager, except the system integrator can bid on equipment and electrical contracting services.	• Single point of responsibility • Simplified contracting	• Not well-known by agencies • Direct bidding to system integrator may violate agency procurement process
Design/Build	The agency must commission the concept plan(s). The concept plan is normally 15 to 30 percent complete at the design level before the contractor is selected. This approach relies on a single entity to be responsible for the design and construction of a project. The agency's role is to monitor the design/build work. The design/build approach is frequently used for federal procurements involving structures. Partnering is generally involved.	• Full transfer of responsibility to design/build team • Eliminates imperfect transfer of design knowledge from designer to contractor [Pearce, 1997] • Rapid completion possible; significant time-savings reported [Pearce, 1997] • Streamlined procurement possible • Engineer and construction work done cooperatively with a single entity to resolve problems • Financial incentive to rapidly complete work • May include warranty of operations management	• Agency assumes greater responsibility for inspection and approval process • May be indistinguishable from engineer/contractor approach when detailed plans with a significant amount of design are developed by the engineer • May increase costs because of contractor risk and high proposal costs (design not complete) • May violate statutes (17 states) • Significant agency commitment to quality control

Table 11-1. Contracting Approaches (Concluded)

Contracting Approach	Description of Alternative	Advantages	Disadvantages
Design to Cost and Schedule	A prioritized requirements list is generated. The contractor supplies all the mandatory items and as many optional items that fit within the cost and schedule constraints.	• Reduces requirements creep • Reduces cost and schedule risks	• Bidders may be unwilling to propose not meeting all the optional features • Overly optimistic proposals will win
Build to Budget	Different from design/build in that functional requirements used in place of detailed design. Proposers develop designs based on their best solution to meeting functional requirements using existing elements where practical. This approach has been used in toll projects.	• Similar to design/build • Allows maximum flexibility to proposers to use their most cost-efficient designs • Reduced risk based on previous developments and applications • May allow added functionality for given budget	• Similar to design/build • Very unusual practice for agencies • Risk based on lack of detailed designs • Detailed design document may prove contentious point and delay project • Very expensive for proposers
• Build, Own, Operate, and Transfer (Boot) • Franchise or Lease	This approach involves long-term contracts with consortium to finance, design, build, operate, and collect revenue. For the system implementation phase, it is equivalent to either the design/build or the build to budget alternatives. The differences occur during the system operations and maintenance phases. These alternatives are typically considered because they do not involve an up-front capital cost for the owner.		

everything up front, they allow incremental development to be used, in which you plan as you go. As new needs or capabilities are uncovered from previous work, the contractor can proceed to work on them. Pre-negotiated billing rates are used throughout the contract. That way, as changes are identified and mutually agreed to, the contractor can say "That will take us *n* hours to accomplish" and the resultant cost will be known. This alleviates the need to go back at each point to ground zero and re-negotiate labor rates.



A leading-edge traffic management center was successfully built using a time-and-materials contract. A "rolling" development approach was used. The system evolved over time by having new pieces of the system put in place at frequent intervals, typically on the order of several weeks. The contracting approach was credited with being able to adapt quickly to the Internet when that became a viable vehicle for transmitting traffic information. The popularity of the Internet could not have been anticipated in advance, and yet time and material contracting provided the flexibility to take advantage of it when it arose. Under other contracting vehicles, lengthy delays would have likely ensued as new contracts would have had to be issued to provide Internet access.

Themes *In spite of its advantages, time-and-materials contracting also has its downsides. There are* ***no silver bullets****.*

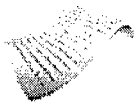
Time-and-materials contracting is a legal type of contract, which is allowed on Federal-aid procurements. Nonetheless, its use may not be politically palatable in many states. (Teaming with your contracting shop may help overcome some of the resistance.) Also time-and-materials contracting may not be the best choice for systems with low development risk for which turn-key, off-the-shelf products exist. In addition, some contractors complain that time-and-materials contracting can encourage the customer to micro-manage the project in such areas as who works on the contract, profit margins, or markups on equipment obtained from third parties.

Themes *While time-and-materials contracting may best facilitate the use of the various themes, it also effectively mandates their use. In particular, its relatively informal nature mandates* ***active customer involvement*** *and on-going* ***collaboration*** *with the contractor. If formal mechanisms were relaxed, but the customer were not actively engaged, a lot of money could be wasted.*

At first glance, it would seem that *fixed-price* contracts would be the best option for the customer. They produce a system at a guaranteed price. On the surface, all the risk is borne by the contractor. In fact, many a customer delights in the prospects of a win-lose situation: contractors underbid the project, the low-bid contractor is selected, the customer "holds the contractor's feet to the fire", and the customer gets a system at a bargain price.

This philosophy of "win-lose" flies in the face of establishing teaming relationships, which are essential for successful software acquisition. In practice, the anticipated win-lose often turns out to be lose-lose. Fixed-price contracting assumes it is possible to know all the requirements ahead of time and just "throw them over the fence". Customers lose on fixed-price when changes need to be made. With software, multiple

changes are inevitable because requirements can't be firmly established ahead of time. As a result, realistic low bids cannot be established.



Fixed-price contracting illustrates the differing perceptions of the public and private sectors in the ITS community. The public sector perceives the contractors as being in the driver's seat on fixed-price contracts. The public-sector interviewees complain that contractors can extract high costs for even small changes. It is viewed as a way for them to recoup their losses from underbidding the contract. On the other side, contractors complain that they lose money on fixed-price contracts because they must respond to requirements changes without being given budget relief. Otherwise they won't be paid. Also there is no flexibility to accommodate unanticipated problems.

Because of these difficulties, the Department of Defense long ago moved away from fixed-price contracting for software intensive systems, especially those that are analogous to ITS. *Cost-reimbursement* contracts, often cost-plus-fixed-fee, are used instead. A true cost-reimbursement contract can provide the needed flexibility. For example, contractors should be more receptive to resolving differing interpretations of the requirements, since they will be paid for their effort. In a true cost-plus contract (see caveat #1, below), you have access to whatever needed skills are available because contractor costs are reimbursed. (Reimbursement is not limited to negotiated labor rates, as in time and materials contracting.) Conversely, with fixed-price, contractors are loath to agree with any changes, since they will have to "eat" the associated costs. Cost-reimbursement also accommodates addressing unexpected problems that are encountered as the acquisition proceeds, without having to re-negotiate the contract.

Cost-reimbursement contracts also have some disadvantages. They subject the customer and contractor to additional paperwork as contractor charges are subject to auditing. With fixed-price, costs that the contractor experiences are to some extent irrelevant to the customer; a defined product will be bought at a negotiated price. How the contractor arrived at that price is not as much an issue. But clearly, if a contractor is to be reimbursed for its costs, then there has to be an audit trail as to what costs are incurred. This entails additional overhead.

Cost-reimbursement contracts with stated deliverables and a fully allocated dollar ceiling have the worst aspects of both fixed-price and cost-reimbursement contracts. (See caveat #1, below for more discussion on this topic.) However, some would argue that without a low ceiling there is the opposite danger that the contractor will have no incentive to deliver because of a guaranteed paycheck. (This argument can be overcome if no fee is paid on costs above the contracted amount. From the contractor's perspective, increased costs without fee reduces profitability.)

If you decide to go with cost-reimbursement contracting for software, there are three important caveats. If these caveats are not heeded, or cannot be implemented under your agency's procurement policies, then perhaps fixed-price is the way to go, as the lesser of evils.

- *Caveat #1:* Do not go fixed-price under the guise of a cost-reimbursement contract. Our private-sector interviewees complain that "Every cost-plus contract is really fixed-price" with a low overall dollar ceiling. When there is no reserve

on the overall project, there is no flexibility. As a result, the potential advantages of cost-reimbursement contracting are not realized. At the same time the added administrative costs of cost-reimbursement contracting are incurred. The contractors also lose. In theory, they can walk away from a cost-plus project if additional expenses are not reimbursed. But in practice, maintaining good customer relations and protecting their business reputation precludes this from happening.

In summary, going with what amounts to fixed-price on a cost-reimbursement contract has the disadvantage of incurring additional overhead, for both customer and contractor, without realizing any of the advantages. It also encourages the contractor to spend to the ceiling to maximize profits. (This, too, can be overcome through incentive provisions.)

- *Caveat #2:* Do not use a cost-reimbursement contract for the software in conjunction with a fixed-price contract for the computing platform (computing hardware). Often software development can be expedited by simply using a faster computer, buying more memory, or adding another computer to the network. But with fixed-price contracting for the hardware, any additional hardware will "come out of the contractor's hide." Therefore, the contractor has a disincentive to make hardware changes. At the same time, with cost-reimbursement contracting for the software, the contractor gets paid for making software changes, even those that result from having too little hardware. As a result, in an attempt to make the software run faster or take less memory, the contractor can spend considerable customer resources to avoid spending even a little of their own resources on hardware. Overall, time and money is wasted as the contractor tries to "shoe horn" the software into too little hardware.

  In short, the hardware and software contracting mechanisms must be compatible. The contractor and customer need the flexibility to make tradeoffs between hardware, software, and also the communications.

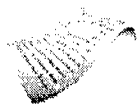
- *Caveat #3:* For purchasing off-the-shelf software that requires little or no customization (such as for the systems near the bottom of figure 4-1 in *Types of ITS Software Systems*, chapter 4), fixed-price may be the preferred way to go. This is another advantage of going with off-the-shelf software whenever possible.

Which contracting approach should I use for ITS software?

There are several underlying assumptions that serve as the basis for the *engineer/contractor* approach [Booz-Allen, 1997; Pearce, 1997]. They do not apply to software or even to ITS in general. Therefore, one of the few areas on which there appears to be consensus regarding contracting is that the engineer/contractor approach should *not* be used for software acquisition:

- Engineer/contractor is intended for systems with firmly established specifications. If nothing else, this document has stressed how software cannot be acquired using firm specifications that are "thrown over the fence," because detailed requirements cannot be known in advance.

- With an engineer/contractor approach, an adversarial relationship between the customer and the contractor is not uncommon. This flies in the face of the need for a collaborative teaming arrangement.
- With an engineer/contractor approach, the customer has limited opportunity to provide inputs. This, too, flies in the face of the need for a collaborative approach between customer and contractor.



Engineer/contractor often leads to multiple layers of subcontracting. One ITS software contractor found themselves third tier down on the subcontracting arrangement of a construction contract. They were effectively shut off from all direct contact with the customer. The lack of contact predictably led to a very bad software experience for all parties. Even if contracts had been initiated, they would have to have been formal ones under the auspices of the prime contractor. And, even then, there would have been no flexibility, since the prime contractor would have been affected.

- Engineer/contractor is best suited for systems that employ familiar technologies. In general, ITS does not fit the category of "familiar technology".
- Engineer/contractor is based on the assumption that the cost of design is relatively inexpensive when compared with construction costs. With software, the opposite is true: the production of diskettes, etc., which is analogous to construction, is far less expensive than the development costs. (See *The Nature of Software,* chapter 1.)
- The low-bid selection of a contractor is based on initial price. However, for software, typically more than half the cost is incurred during maintenance, after system installation.

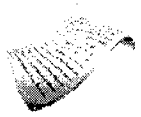
If engineer/contractor approach is not to be used, let us consider some of the remaining alternatives. They aren't ideally suited to software, but they can be used with success.

The *systems manager* approach has the advantage that the consultant is kept on board throughout the project. The consultant has the "big picture", understands the reasons for various decisions, and can maintain continuity throughout the project. (In the engineer/contractor approach, the consultant "throws the requirements over the fence" and exits the picture.) The consultant works with the customer to develop the requirements. The systems manager may also be responsible for the software development or can be the team member with the software technical expertise. (See the *Building A Team,* chapter 7.) However, contractors who have worked as systems managers complain that they really don't have control over the project. Multiple contracts are issued by the agency, and not by the systems manager. The agency decides when to accept various subsystems developed on these contracts, sometimes overriding or ignoring system manager recommendations. The systems manager may have to make the software work with systems that they wouldn't have accepted. In effect, the systems manager is in the unenviable position of having responsibility without authority.

In one case, the customer signed off on a communications subsystem that wasn't thoroughly tested. Later on, problems arose in implementing the software. At its own expense, the systems manager had to demonstrate that these problems were due to flaws in the communications subsystem and not the fault of its software.

On another ITS software acquisition, the communications subsystem was acquired independently from the software. The communications subsystem met specifications, and worked in the sense that data could be transferred back and forth. The communications were to be used as the backbone for controlling video surveillance cameras. However, the communications subsystem used polling, a technique not well suited for this function. When the software to control the cameras was implemented, it was not at all responsive to operator commands. For example, upon command, the camera would begin to zoom, but this operation could not be stopped in a timely fashion, so the camera would zoom well beyond the operators intent. This problem is directly attributable to the use of polling for the communications. However, because of the contracting approach, no one could be held responsible: the communications worked as specified, but there was no technical way for the software to be made to work with it. Presumably if one contractor had been responsible for the entire system, they would have taken a total systems approach and chosen communications that work better with camera control software. Even if they picked wrong, they would clearly have had full responsibility to make the entire system work.

The *design/build* approach offers the possibility of overcoming many of the problems encountered with the other approaches. It gives one contractor full responsibility for the system. This provides the flexibility, for example, to trade off hardware, software, and communications. It also provides a single point of responsibility and authority. The design/build approach has been tried on several ITS acquisitions, with mixed results.



Unfortunately, the design/build approach did not work well for a large freeway management system project. It ran into the usual software problems of functionality not meeting requirements and the contractor losing money. (There are no silver bullets!) Whether the problems encountered are intrinsic to design/build, or were due to other factors is not totally clear. For example, fixed-price contracting was used, which limited contractor flexibility and the ability for the customers to provide inputs. In addition, customer-developed requirements were kept rigid. The procuring agency feared that flexibility would open them up to legal challenges by the losing contractors, who could then say: "We could have done the project, if you'd also given us requirements relief."

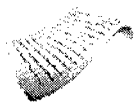
One of the lessons learned on design/build is the need to manage expectations. The use of a single design/build contractor may alleviate some of the overhead associated with multiple contract administration. However, it is unrealistic to expect that overall system costs will be materially affected. This is because fundamental costs (e.g., purchase and installation of variable message signs) will not be impacted, whether incurred on the same contract as the software or on a separate contract.

Another lesson learned from a design/build experience was stated as follows by one of the participants: "Just because the project is design/build this does not mean the DOT can just sit back and expect the final product to be produced as they expect. Constant interaction and guiding of a complex ITS project is required."

Fixed-price contracting also causes problems for design/build. Not only is the budget not derived from the requirements, but the overall price for the contract is set before the requirements are even known. Then the build phase is squeezed on both ends, by too ambitious requirements and too little money.

Consider other alternatives

Task-order contracting has been used by several state DOTs. Under this approach, a contract with multiple tasks is issued for the project. However, only one task is funded; the rest are options that the customer can choose to exercise ("turn on") at a later date. The first task is used for initial work, such as requirements analysis or the development of a detailed design. It also includes planning for the next phase of activities, perhaps building a baseline capability. If the initial task is performed satisfactorily, then the next one is funded. This optional task also includes planning for the next one, perhaps for adding functionality to the baseline system. Subsequent tasks are handled in a similar manner, with costs negotiated on a task-by-task basis.



One transit agency successfully used a similar approach on an ITS project, in which the initial task was for a "needs analysis."

This phased approach offers several advantages: it breaks the development into bite-size pieces, provides the contractor with incentive to do well so that subsequent phases will be funded, and allows customer and contractor to "learn as you go." Also it recognizes the reality of schedule and budget uncertainty on a large software development, by providing some costing flexibility. Money initially planned for the "out phases" can be shifted to earlier tasks as the needs arise. You may not get everything you initially hoped for, but you will succeed in getting a working capability. (Or at least learning from an early task that the contractor will not be able to deliver, in which case you should terminate them and not fund the optional tasks.)

On-call services are another option, but are not considered very often.

A variation of the contracting approaches discussed above is to include a *design competition* phase as part of the source selection process. Two or more contractors are chosen to proceed in parallel at the beginning of the project. Their progress is then used as input to the final selection. Thus actual ability to perform, rather than just written proposals, becomes a factor in selecting the ultimate contractor. However, care must be taken during the design competition phase to treat the two (or more contractors) equally, and to keep their designs confidential from the other contractor(s). This approach was employed successfully in the development of the National ITS Architecture.

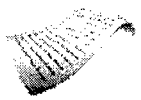
Other states have found that their state contracting mechanisms are simply too rigid for software. So they have resorted to funding the *work through other entities*. Metropolitan planning organizations (MPOs) or state universities can be given full responsibility for issuing a contract and managing the project. The risk here is ensuring that these organizations understand your needs and have the requisite software management skills.

Another possibility for using an outside organization exists, although it has not yet been tried out on any ITS acquisitions. It is the use of an *Information Technology Omnibus Procurement (ITOP)* run by U.S. DOT's Transportation Administrative Service Center.

The Transportation Equity Act for the 21st Century (TEA 21) calls for "appropriate methods of procurement for intelligent transportation system projects ... including innovative and nontraditional methods such as the Information Technology Omnibus Procurement."

They maintain a list of pre-approved contractors, and provide other procurement services, such as assistance in writing an RFP. However, the state agency would still retain responsibility for managing the contract. (*Where To Get More Help,* chapter 21 tells how to obtain more information about ITOP.) In addition to being untried for ITS, ITOP has the risk that you may not be able to get access to all the vendors who have the requisite ITS products or skills. Only the listed contractors would be accessible.

Finally, keep in mind that you may be able to avoid the risks of software development contracts altogether by going with an off-the-shelf purchase. (See *Build/Buy Decision(s),* chapter 10.)



Several states are trying out some innovative approaches for their ITS software acquisitions;

- *One state DOT is using an obscure contracting mechanism they found under the state's procurement laws: non-professional services contracting. It seems to offer several advantages for ITS software and is allowed under Federal Aid regulations.*
- *Another state funded four contractors. They worked independently and carried out a portion of the software development in parallel. An evaluation was then held, and one contractor was selected to proceed with the rest of the project.*
- *Several states have used best-value procurement or life cycle costing instead of going strictly low-bid.*
- *One state has had success with cost-plus contracting in conjunction with a reserve fund for contingencies. The state agencies and the contractors viewed this approach as "win-win."*

The bottom line is: look at your *full* range of options before selecting a contracting approach for software. As one private-sector interviewee told us, "challenge traditional thinking."

How the choice of contracting alternatives impacts project planning

Your choice of the contracting vehicle will to a large extent determine the sequence of the various software activities. Under some of the contracting choices, the requirements, for example, will be developed *before* an RFP is issued. These would then be given *to* the software development contractor and serve as the basis for their work. Under other choices, the requirements would be developed collaboratively *with* the development contractor *after* contract award; only preliminary requirements or a features list would be available before the RFP is issued. (See also *Sequence of Acquisition Activities,* chapter 6.)

Use best software acquisition practices regardless of the contracting approach

Although none of the contract types and approaches were developed specifically for software, probably all of them can be made to work. The important thing is to ensure that the contract allows best software acquisition practices to be followed. Ensure that the various themes stressed in this document are not negated by the contract. For example:

- Regardless of the contract type, the contract should allow frequent and open communications between customer and contractor. This may require explicit contracting language, especially if the software is to be developed under a subcontract.
- Make sure that the contract allows for the needed flexibility. It must do so in a way that will not open up the contract to legal challenges.

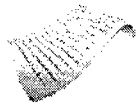
In order to ensure that an appropriate contract vehicle is chosen, you will need to work closely with your contracting and legal offices. Team with them from the inception of the project. Early teaming will give you the opportunity to explain the goals of your project, and discuss how software is different and therefore requires different approaches. Jointly explore alternatives and determine whether previously untried approaches are illegal or simply unfamiliar. Doing this effectively may take considerable management know-how and political savvy; no one said it would be easy. On the other hand, if you wait until the last minute to approach them with the proposed use of unfamiliar contracting vehicles, contracting officials will naturally be resistant.

One transportation agency ran into trouble when, late in the procurement cycle, they found that construction and union wage scales were going to be imposed on a software project. A consultant to the project acknowledged that they may have been able to avoid this problem if there had been a better job of teaming with the contracting office earlier in the procurement.

- *The familiar engineer/contractor (design-bid-build) used for construction projects is not appropriate for software; it should not be used for software acquisitions.*
- *Work with your contracting or purchasing office and legal office early in the project to explore your full range of options; challenge traditional thinking.*
- *Fixed-price contracting may not provide the needed flexibility for building software. Consider a time-and-materials type of contract and/or innovative contracting approaches as alternatives to fixed-price.*
- *Do not use fixed-price contracting practices under the guise of a cost-reimbursement contract.*
- *Do not use a fixed-price contract for computer hardware in conjunction with a cost-reimbursement contract for software.*
- *Fixed-price contracting may be appropriate for off-the-shelf software.*
- *Whatever approach is chosen, that approach will still require the application of sound acquisition practices; a contract is not a substitute for them.*

CHAPTER 12
IDENTIFYING THE SOFTWARE ENVIRONMENT

An important part of engineering a software system is identifying the environment in which the software will be operating. This allows the software design to accommodate any constraints imposed by the environment and ensures that all the necessary interfaces are implemented.



Identifying the environment for a software system is much the same as performing a site survey for a civil engineering project.

The environment refers to the combination of conditions external to the software under which the software must function. For a simple off-the-shelf word processing application, identifying the environment may be as simple as listing the existing computer, operating system, and printer on which the application will be used. The more extensive and complex the software, the more likely that identifying the environment will result in a longer list, for example, computer(s), printer(s), drive(s), other hardware peripherals, operating system, network hardware and software, database, and other software or applications with which the software must operate. If interfaces will be needed to pre-existing systems ("legacy systems") then those systems constitute an important part of the environment.

The software environment identified typically leads to specifying requirements for the software acquisition. The requirements for the environment can be described in two ways: as functional requirements or as technical requirements or constraints. The selected way generally depends on the scope of the acquisition and your own information system or software standards. If the software is being acquired in conjunction with the other components for a complete system, then the components are likely to be described in functional terms. However, if the software needs to run on a specific platform (e.g., hardware, operating system, network, or data base), the requirements will specifically identify the platform, which becomes a technical requirement or constraint. It is not unusual that both forms of specification are used.

The environment-related requirements specified will depend on the purpose and scope of the acquisition. For example, if you are acquiring software to run in a standalone environment, there is no network or communications environment to be specified. If you are acquiring software that will run on a remote system and provide data back to a traffic control center, it is likely that the communications and remote application management environment will need to be specified. If you are planning to acquire custom developed software, your organization may have programming languages, databases, data formats, or other development standards that need to be specified as technical requirements or constraints.

Determining an appropriate environment can be complex and require the help of experts who are familiar with the components of the environment. Without specifying the environment, one risks purchasing a potentially excellent software application only to find that it can't use your existing communications lines, won't interface with another critical application, or that you need to purchase additional memory or data storage. One source of help is your information systems staff; they can provide descriptions of your current computer system and network environment. Given their experience and level of expertise, they may even be able to identify recommended platforms, provide insights into their ability to support and maintain different platforms and components, identify existing platforms and tools that could be used, and provide information on applicable information systems and software standards. Another potential source of help are other city, county, and state DOT offices. In particular, their information systems staff may be able to provide the expertise needed to ensure that a reasonable and appropriate combination of environmental components are being specified in your acquisition.

Over-specifying the environment in the requirements, however, also carries risks. An overly specified environment may severely restrict the options available to the contractor and significantly increase development time and costs. An overly specified environment also carries the following risks in regards to off-the-shelf products:

- You may unnecessarily preclude an otherwise satisfactory product solution.
- You may incur significant unnecessary costs in having the off-the-shelf product re-engineered to run under your choice of operating system or with a particular set of field devices.
- The resulting modified product may prove less reliable than the off-the-shelf version, which has already been "wrung out" over a larger customer base.
- You may be off the upgrade path for future upgrades and offerings by the vendor.

Environment specifications should be kept to those truly required, and not include those that are only "preferred". Avoid specifying the computing platform (hardware and operating system) ahead of time unless it's imperative. If you're considering a particular platform, do the reasons for choosing it (e.g., it supports your agency's information management systems) apply to the real-time demands of an ITS system? Similarly, do not acquire other components of the environment (e.g., field devices) separately without considering their impact on the software. Flexibility on the part of the customer and contractor are needed to ensure the system operates in the environment, accommodates the preferences of the customer, and allows the contractor to design for the best, cost-effective solution.

The environment also determines how much customization is needed for existing products. Software that works in one location will probably not "plug and play" in another unless the total environment (field devices, communications, computing hardware, other software such as the operating system or database management system) is identical in the two locations. A software product that would work without modification with one set of field devices may require significant re-tailoring if it is to be used with another set. By insisting on a particular vendor's field device or on a particular database management system, you may incur significant cost and development risk in the

software. This can happen inadvertently if you purchase hardware on a low-bid basis without regard to the implications for the software.

Checklist 12-1 identifies what to consider when identifying the environment for your system.

- *Identify the environment -- including the interfaces to legacy systems -- in which the software will be operating.*
- *Do not unnecessarily constrain the system design by prematurely specifying the computing hardware or operating system.*

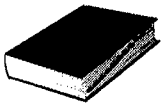
Checklist 12-1. What to Consider When Identifying the Software Environment

✓	Interfaces to legacy and other existing systems or applications (what legacy software and systems must the new software interoperate or interface with, including application monitoring and management systems?). *
✓	Existing communications and networks (including protocols, characteristics such as line speed, bandwidth, dedicated or dial-up, type of network, significant components).
✓	Interfaces to planned future applications or systems (what other software is planned for the future and will need to interoperate or interface with this software?). *
✓	User population and user interface (e.g., graphical user interface (GUI), point-and-click).
✓	Location/physical environment (e.g., office, computer room, outside; lighting conditions, space constraints impacting use of mouse or keyboards; uninterruptable power supply).
✓	Security measures that implement security policies and procedures, to include physical safeguards (locked rooms), software protection (passwords), and hardware systems (so-called firewalls to the Internet).
✓	Performance (how much data to be processed and how fast it must be processed, data accuracy).
✓	Standards.
✓	Hardware (e.g., PC, mini, or mainframe).**
✓	Operating system.**
✓	Data base management system (DBMS).**
✓	Programming languages, development methodologies, maintenance requirements. **

*** The National ITS Architecture can assist in identifying interfaces to other systems.**
****Only if absolutely necessary.**

CHAPTER 13
RESOLVING THE INTELLECTUAL PROPERTY RIGHTS

Software licensing and ownership issues were brought up more often than any other in our interviews with ITS project personnel. Almost every interviewee raised this as an issue, whether they were from the public or private sector. In fact, many offered it as their first choice for a topic that this document should address. Several interviewees even indicated that they currently were, or recently had been, involved in litigation over licensing and ownership issues. Although public sector and private sector interviewees had different perspectives on how to address such issues, they all agreed on one thing: this is a very contentious area that leads to conflicts between customer and contractor.[1]



We will use the term "intellectual property rights" to encompass a broad range of topics associated with ownership, licensing, copyright, etc.

Who has the rights to the software once the project is complete?--a major point of contention

The following scenario seems typical. When the contract is initially signed, both parties *think* they've reached a mutually satisfactory agreement with respect to the intellectual property rights issues. Indeed, the contract may contain words about *owning*, *licensing*, *copyrighting*, *selling*, or *leasing* the software. Unfortunately, unbeknownst to both parties, they interpret these terms differently

Even the meaning of the word "software" in the contract language has been a frequent point of contention between customers and software contractors. Often the customer interpreted "software" to include the source code, whereas the contractor meant for "software" to apply only to executable or object code.

These different interpretations don't surface until much later, towards the end of the contract. At that time, the customer claims the rights they think they signed up for. "Okay, turn over the software so I can maintain it [or *sell it* or *make a copy of it* or ...]." That's when the contractor responds, "Hey, wait a minute. You don't have the right to do that. You can only ..." This leads to accusations of not bargaining in good faith or not living up to prior agreements. Protracted negotiations, and sometimes litigation, then follow.

[1] Problems can also arise in regards to hardware, computers, peripherals, database software, compilers, etc. that are bought under the contract: who has the rights to these items at the conclusion of the contract?

Determine your true needs with respect to intellectual property rights

Before you enter into negotiations over intellectual property rights, consider what your true needs really are. Doing so may enable you to avoid unnecessary fights over the rights.

Your software maintenance concept partially determines your needs in regards to intellectual property rights. (See the "Software Maintenance" section in *Training, Operations, and Software Maintenance,* chapter 16.) If you are considering software maintenance by in-house staff, ask yourself whether you really want to take on that responsibility. Ask yourself, "If I do take possession of the source code, does my agency have people who are capable of maintaining it? Or will we just turn around and hire maintainers, perhaps the original development contractor?" You may be able to avoid unnecessary fights over intellectual property rights, if you determine that you won't be able to maintain the code and don't insist on additional rights in the first place.

Taking on responsibility for software maintenance implies the following:

- Having qualified staff. Note that programming experience on information systems does not qualify an individual to work on demanding, real-time ITS software. Such staff are in short supply. If you currently do have them on board, will your agency salary structure allow you to pay them enough to attract them? If you train existing staff, will you then be able to hold onto them?
- Familiarizing the staff with the internals of the software and the support environment and tools used to maintain it. This will require close collaboration of the maintenance staff and the contractor developers from the outset of the project.
- Taking over such tasks as documentation and software configuration management (See *Software Configuration Management,* chapter 18), for which contractors normally have prime responsibility.
- Setting up and running a support environment (see next section, below)
- Having access to the following items:
  - source code, in compilable computer files; listings are not sufficient
  - documentation on databases, data structures, and interface protocols
  - development tools used to compile the software, keep it under configuration control, test it, etc.

  Note that there are costs associated with having access to these items. The full support environment for a commercial database package, for example, is considerably more expensive than run-time license for the same package. There is no point in paying more for extra rights if you don't have the skill base to take advantage of them.

Sometimes public sector customers insist on having full rights to the software so they will not get "locked in" to a particular vendor. But in practice, only the original developer may have the technical expertise to maintain the code. So winning full rights to the code may be a hollow victory. You may pay more, yet still have to contract with the original vendor for maintenance. In fact, one software vendor indicated that it doesn't mind

giving its customers access to the source code, knowing that they (the vendor) are the only ones who could make sense of the code and maintain it anyway.

In some cases, you may decide to obtain rights to only certain parts of the software. But where to draw that line raises other questions. For example, you may decide that you should have complete ownership to all software developed on your project, but not to the pre-existing software that the contractor brought to the project. That software would be retained by the contractor. But will having the developed software do you any good if it can't be run without the contractor's software? Recognize that off-the-shelf software is most often delivered as object code only.

Sometimes state agencies retain the rights to redistribute software to other states for a nominal fee. In return, they get all the enhancements developed by the other state. This approach may have previously worked well for you on a stand-alone program used in an office environment. But ask yourself whether it makes sense for an ITS system. Do you really want to get into having to supply technical maintenance and support? Conversely, if you are considering using another state's software, do you really want someone else's undocumented, unsupported package? The ITS environment is complex, so the software will probably need changes to adapt to the interfaces (sensors, signs, etc.) and operating environments that differ between the two states.

Be explicit with respect to intellectual property rights

To avoid problems with respect to intellectual property rights, have your contract explicitly call out what rights each of you—the customer and the contractor—has to the software.

Checklist 13-1 suggests some of the rights that should be explicitly considered. Then, *before the contract is signed*, walk through the language together line-by-line and discuss it. That way, there will be no surprises later on. Also you're more likely to have a co-operative relationship at this point than later on when differences suddenly become apparent and feelings harden. As you walk through the contract, make sure that it clearly states your agreements. It should also clearly and explicitly differentiate the rights each party has with respect to the source code, the object code, and the documentation. The rights may or may not be the same for each of these.

Themes

*This is an example of our theme that customer and contractor must maintain **open communications** throughout the project. Here there is a need for open dialogue even before the contract is signed. Similarly, we recommend that in the technical arena, you and the contractor walk through the requirements together line-by-line to reach a mutual understanding of them.*

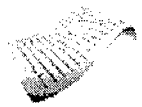
Checklist 13-1. Intellectual Property Rights

	General Rights
✓	Who owns the software? What rights does that entail?
✓	Who holds the copyright to the software? What rights does that entail?
✓	Should copyright notices be included as source code comments? Who should be listed as retaining the copyright?
	Customer Rights
✓	Can the customer make additional copies of the operational software for their internal use on this project? On other projects?
✓	Can the customer distribute copies of the operational software to other agencies or departments within their state?
✓	Can the customer distribute copies of the operational software or issue licenses to other states? Can such a license allow the other state to make changes or enhance the software, or does it only give them the right to use it?
✓	Can the customer give away copies of the operational software for free? Charge a fee?
✓	Can the customer change the operational software or make derivative works?
✓	Can the customer disclose the source code of the operational software to other vendors or allow them to make changes to the software?
✓	For the previous six items on the checklist, which portions of the operational software can be copies or distributed: the source code? the object code? the documentation? how many copies may be made?
✓	Does the customer have rights to any subsequent upgrades made by the contractor?
✓	Are there portions of the software that are needed to run the system that are not covered by the licensing agreements? Such items could include the operating system (e.g., Windows, UNIX), a commercial database management system, a geographic information system, or a digital map. For these items, there may be a different number of copies that can be made, run, distributed, etc. than there is for the rest of the software.
✓	Will the customer have access to the source code? If so, as compilable files or only as listings?

[Checklist continued on next page]

Checklist 13-1. Intellectual Property Rights (Concluded)

✓	Will the customer have access to the support tools and development environment that were used to compile the software, keep it under configuration control, test it, etc.?
✓	Will the customer have rights to all the training material?
✓	Will the customer have rights to the executable environment needed to run the software or must these be purchased separately from other vendors?
✓	Will the customer have access to documentation on database formats and interface protocols?
✓	How many computers can contain copies of the software? How many can run the software? (Note: These numbers may be different to allow for backup copies.)
✓	How many computers can simultaneously run or access the software? (Note: On a network, all computers may be able to run a piece of software or access a database, but the licensing agreement may restrict the simultaneous number.)
✓	How many users have license to run the software? How many simultaneously?
✓	Can the software be run across a network? (Note: There are two options here. the software could be run remotely, that is, on the "other" machine where the copy resides; or, a temporary copy could be made on your machine and run locally.)
	Contractor Rights
✓	Can the contractor distribute the software to other customers? If so, can they charge for it?
✓	Can the contractor reuse portions of the software on other contracts?
✓	Can the contractor copyright or patent the software or patent other parts of the system? If so, will the customer have to pay royalty rights?
✓	Does the contractor have rights to any upgrades made by the customer?



An interview with an ITS project manager was published over the Internet and indicates what can happen when explicit agreements are not reached up-front on intellectual property issues [available 4/23/97 at ITS Online <URL:http://www.itsonline.com/travinfo1.html>].
Q: "Who owns the software that [the contractor] developed for [your project]?"
A: "That's been a very difficult issue for us with [the contractor. We] did our best in our procurement process to say we wanted to own the source code. However, we definitely did not articulate that precisely enough. In the early days of the project, when everything was still rosy, we thought we articulated it and everything felt fine. As time went on, it was clear that [the contractor] did not intend to deliver the source code. We had to hire intellectual property lawyers to negotiate it, and we still haven't finished the source-code agreement. We've finished the boiler plate, and now we're agonizing through the exhibits as to what constitutes the system itself."
Q: "What's the bottom line of the "boiler plate" that you mentioned?"
A: "The philosophy of the agreement is that [the contractor] owns the software and the source code, but we have a license to the software and source code. So we can, if we so desire, contract out to a third party to modify the software."
Q: "So you will have access to the source code?"
A: "That's correct. We can have one backup copy of it, but cannot distribute the code to other agencies. That meets our needs. The Federal Highway's language we originally used was not as comprehensive as we had believed it to be."

As a result of trying to explicitly define both parties' rights, you may not be able to reach agreement and cannot enter into a contracting arrangement. But that is far better than the alternative of procuring a system under differing assumptions and then having to go to court later on. In court, at least one party, and likely both, will get less than what they thought they had agreed to. The point is not that terms such as *ownership* or *licensing* may have precise legal definitions that could be resolved in a court of law. Rather, such terms seem to connote different things to different people (or at least to different non-lawyers). So it's best to clarify the understandings up front and avoid legal battles in the first place.

The ownership language in a contract can lead to subtle misinterpretations on either side. For example, a contract allowing the right to distribute software to agencies in other states may seem less restrictive than one that allows distribution only within your own state. However, on one Commercial Vehicle Operations (CVO) project, distribution language in the contract was interpreted differently by the lead state and the contractor. Both agreed that the lead state could distribute the software to peer agencies in other states that were participating in the project, but disagreed on the use of the software within the lead state on a project closely associated with, but not part of, the original CVO project.

Access legal expertise

Intellectual property rights are a specialty area of the legal profession. Intellectual property rights as they pertain to software are an even narrower area, and a relatively new and rapidly evolving one to boot. Therefore lawyers who are knowledgeable in this area are hard to find. Probably none exist in most transportation or transit agencies. Therefore, as you build your team, consider acquiring the services of a lawyer with this subspecialty as his or her area of expertise.

One ITS project hired an intellectual property attorney and termed it "money well spent." Another project ran into troubles, hired a nationally known software attorney, and "that really helped."

- *Regardless of whether or not they have precise legal meanings, terms such as "ownership" or "licensing" have different connotations to different people.*
- *Before a contract is signed, reach agreements on intellectual property rights for the software.*
- *Walk through the contracting language together; discuss the implications, resolve issues, and ensure the contracting language clearly and explicitly states your understandings.*
- *Be explicit with respect to source code and object code and the media on which it will be delivered.*
- *You may wish to acquire the services of a lawyer who specializes in software intellectual property rights.*
- *Think through your true needs before insisting on certain rights at the negotiating table.*

CHAPTER 14
PROJECT SCHEDULING

"More software projects have gone awry for lack of calendar time than for all other causes combined." — [Brooks, 1975]

"[You] can reduce effort, cost, (and defects) by planning a little longer schedule." — [Putnam and Myers, 1996]

At this point you've written a project plan, developed a set of requirements, made a build/buy decision, and selected a contracting vehicle. Now it's time to incorporate it all on a project schedule.

What to include on the project schedule

Checklist 14-1 suggests software-related activities and milestones to include on the project schedule. Clearly, the schedule will also include other major activities that are not software-related (site preparation, installation of field devices, etc.). Also include milestones that commit the customer to key activities. This includes dates for supplying any government-furnished equipment or facilities, submitting comments on documentation, and accepting deliverables.

Setting schedule milestones

A project schedule shows activities and milestones. The following are two good management practices for setting the milestones. They are applicable on any project, but seem to be particularly important for software.

- Milestones should be well defined and unambiguous. That way, there can be no argument as to whether they have been met or not.

On one ITS project, milestones were so ill-defined that the participants were openly puzzled as to whether they had met them or not.

- Milestones should be binary; that is, they've been met, or they haven't been. Do not get into a situation where you're "90% of the way there". Instead, divide the milestone into several smaller ones, each with a binary completion criterion.

Putting these practices together, a milestone might be "the coding is 100% complete" where "complete" is explicitly defined. (Does it include any testing? documentation? placing the software under configuration control?)

Checklist 14-1. Software-Related Activities and Milestones on the Project Schedule

	Contract negotiations
✓	Walk-through of the intellectual property rights issues.
✓	Signing the contract (milestone).
✓	Dates on which the agency furnishes contractually-required items to the contractor (equipment, space, services, etc.) (milestones)
	Requirements
✓	Requirements walk-through.
✓	Signing the requirements (milestone).
✓	Rapid prototyping.
	Size estimates
✓	Independent size and schedule estimates by contractor.
✓	Resolving differences.
	Management controls
✓	Risk management reviews
✓	Project reviews.
✓	Inspections.
✓	Document reviews.
✓	Document approvals (milestones).
	Acceptance testing
✓	Detailed acceptance test planning.
✓	Conducting acceptance tests.
✓	Analysis of acceptance test results.
✓	System acceptance (milestone).
	Training
✓	Training preparation and planning.
✓	Conducting the training.
	Support
✓	Support facility development.
✓	Transition from development to operations and maintenance (milestone).

Typical scheduling flaws to avoid

Two flawed practices are encountered more often than not in scheduling software projects:

- *No technical basis for the schedule.* One of the fatal flaws of many software acquisitions, and the reason why so many projects are late, is that the schedule is developed independently of the requirements. All too often there is "false scheduling to match the patron's desired date." [Brooks, 1975] The target date of a system is selected arbitrarily, picked for political reasons, or "imposed from above."
- *Excessive schedule pressure.* Target dates are overly optimistic, sometimes unrealistically so. As a result, necessary activities are bypassed or short changed because there isn't enough time to do them.

Let us now discuss good practices that avoid these flaws.

Use requirements-based scheduling

Derive the schedule from the requirements. The required functionality determines how long the project will take. Have the requirements, schedule, and budget agree at all times, not just at the outset of a project. If you revise the requirements and add new features to a system, be sure to adjust the schedule accordingly.

Now we recognize that in some cases the end date cannot slip. For example, major events such as the Olympics are not going to wait until a city puts a traffic management center in place. In such cases, the requirements and schedule need to be developed iteratively. If there isn't enough time to do all that was planned, cut back on the requirements until the planned activities can fit into the time allotted for them.

In other cases, you may find that the schedule and resources needed to meet the requirements are "budget busters". In fact, a good rule of thumb is that they probably will be if the requirements include "any functions that are not absolutely essential." Like potential home buyers, customers of software "generally want more than they can afford." [Humphrey, 1989, page 84] And like home buyers who are forced to scale back on their expectations, the system must be pared back to fit the resources allocated for it.

Themes *These are another illustration of the need for **flexibility**. Either there must be some give in the requirements, the schedule dates, or both.*

The bottom line is that regardless of the project and its circumstances, the schedule and requirements should be in concert with one another.

Allow sufficient time

> *"The tried and true ways of blasting through a schedule barrier—more workers, money, overtime, computer time—don't seem to work for software." [Putnam and Myers, 1992]*

Project data show that a realistic schedule is one of two main keys to a successful project. (The other is a stable requirements base.) [Jones, 1997]

Unfortunately, most schedule estimates are not realistic; they are almost invariably optimistic. There is misplaced faith that "all will go well" and take only as long as they "ought to". [Brooks, 1975] Generally speaking, when people are asked to give a range of estimates, the "best estimate" turns out to be optimistic, while the "worst case estimates" cluster around actual performance. [McConnell, 1996]

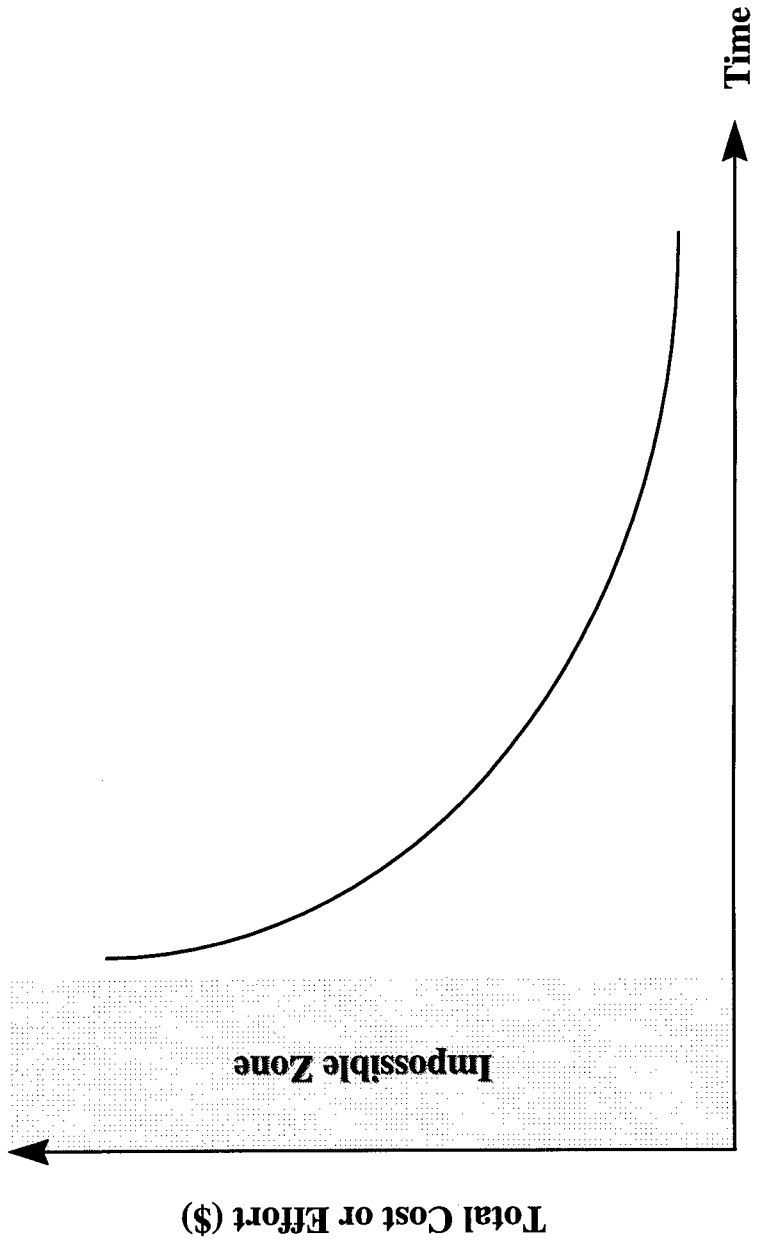
Some managers compensate for human nature and natural optimism by applying a factor to all size and schedule estimates given to them; a factor of two is often suggested.

One of the most effective ways to reduce costs is simply stretch out the schedule! "One would think off-hand that extending the development period would increase effort and cost—there is more time over which people would be working." [Putnam and Myers, 1992] However, that doesn't happen. Instead the converse is true: If you squeeze the schedule, costs go up dramatically. It is not surprising that when you compress the schedule, more people are needed at any point in time and the *peak* staffing increases. But what is surprising is that the *total* staffing, development effort, and cost also increase. Another example of how intuition gained elsewhere doesn't necessarily apply to software.

Multiplying the number of staff by the length of a project gives the number of staff-months. On other types of projects, staff and project length can be traded off for one another, since staff-months reflect the size of the job. This is not the case for software. A famous quote summarizes the situation: "The man-month is a fallacious and dangerous myth, for it implies that men and months are interchangeable." [Brooks, 1975, page 231] In fact, the tradeoff between schedule ("months") and development effort ("men") is far from linear. A little schedule relief results in significant cost reduction.

There are two explanations for this somewhat surprising result. First, as more people are added to a project, you must communicate and interact with them. In effect, with each new hire, all the previous employees become marginally less productive. Second, as the schedule is compressed, you turn sequential activities into parallel ones. This can be done to some extent, but with a price of lowered productivity.

Even if you have an unlimited budget or inflexible end dates, and are willing to pay the price of a condensed schedule, you can only squeeze the schedule so much. There is a lower limit; the project simply can't be done in less time. You pay a high price to approach the limit, and there's no way to go below it. Yet schedules are often set in the "Impossible Zone". (See figure 14-1.)



Note: The total cost and effort go up dramatically as the schedule is compressed. However, no matter how many resources you apply, you cannot squeeze the schedule into the "Impossible Zone".

Figure 14-1. Impact of Squeezing the Schedule

To summarize, first develop a realistic schedule. Then stretch it out somewhat.

Determining how much time is needed on the schedule

Typically you begin by estimating the size of the software. These estimates can be expressed in various units, such as lines of code or function points. Then this size estimate is used to derive the time and effort that will be needed to develop the software.

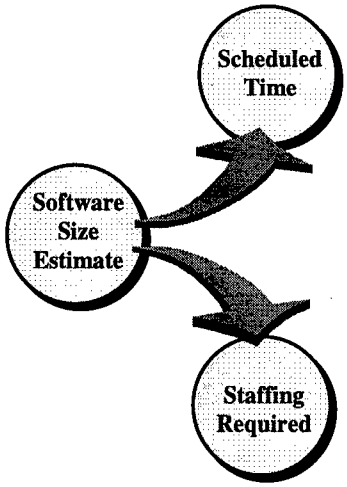


Figure 14-2. Size, Schedule, and Staffing

Unfortunately, at the outset of a project, it may be impossible to know the eventual size of the project; only a very broad range is possible. (See *Estimate ranges,* below.) This is especially true for those with limited software acquisition experience who may not have a clue as to the eventual size of the software that will be built. Once again, the advantage of using existing products is evident: there is no need to estimate the size of the software for them.

Where To Get More Help, chapter 21, lists some references for estimating the size of the software. Generally, the approach is to divide the system into as many components as possible, estimate the size for each, and then sum them up. For example, it is much easier to estimate the size of an incident management sub-component (e.g., determining which messages to put on the variable message signs) than it is to estimate the size of the overall incident management function.

Some recommended practices in estimating software size are:

- Utilize the resources of the software experts on your team to estimate the size.
- Get multiple size estimates; the more people who estimate the size the better. There are mathematical techniques for combining their estimates into estimate ranges that have means and standard deviations. There are also mathematical techniques to combine pessimistic, optimistic, and best guesses into an expected size and a standard deviation for the estimate.

- Obtain independent estimates whenever possible. Independent estimates are better than the ones from those who have a vested interest in the project, such as project managers.
- After requirements walk-throughs (See *Requirements Management,* chapter 9B.), have the contractor produce independent estimates for the size of the system. You will undoubtedly find significant differences between their estimates and your estimates. Compare your estimates and resolve all differences.

Themes *Comparing schedule estimates is an effective technique for* ***team building*** *and* ***collaboration.***

- Estimate at the lowest level of detail possible.
- Never give off-the-cuff schedule or size estimates. People won't remember the caveats you place on them. [McConnell, 1996]

Once the size estimate has been obtained, various productivity factors can be applied to determine how much staffing and time will be needed to carry out the project. There are software tools that can assist in making these tradeoffs between cost and schedule. However, the productivity factors you need to plug into the models are not easy to obtain. Generally they are based on a track record. But if this is your first ITS project, you will have no prior project experience to go on. And since a contractor is not yet on board, you cannot use their organization's experience. (Once they come on board, you can work with the contractor to refine these estimates based on their experiences.)

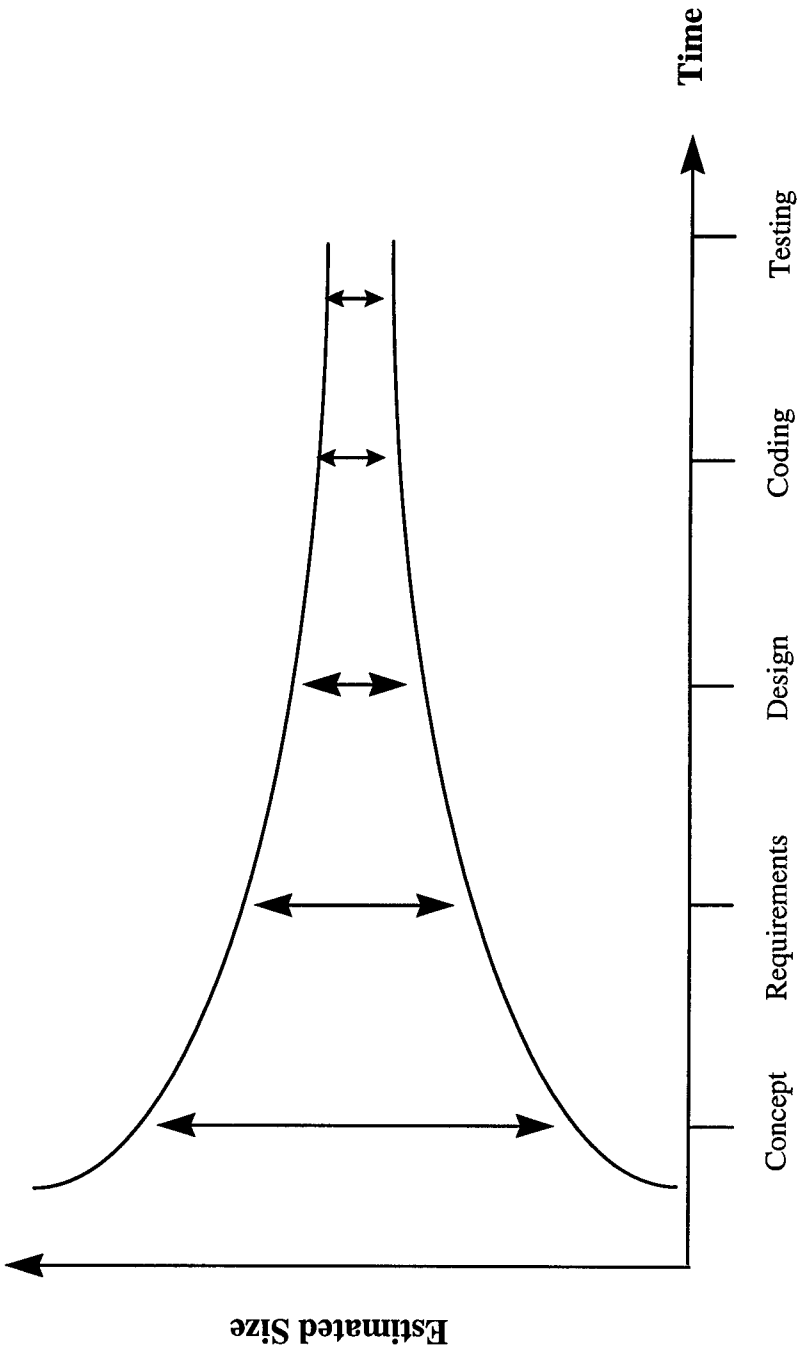
Estimate ranges

Of course you would like to have a reliable point estimate for the size of any project. Unfortunately, obtaining one at the outset of a software project is beyond the state-of-the-art. Only an estimate with a range of values is possible. These can then be refined over time to produce a narrower range as the project proceeds and more information becomes available. (See figure 14-3.) For example, only an estimate with a broad range is possible after the requirements are initially developed. Once the contractor has carried out a detailed software design, an estimate with a tighter range can be given.

The dilemma for a project manager is that a range of cost and schedule will not be acceptable to decision makers who approve projects. "We want to develop a system that will take from 8 months to 2 years and cost somewhere between $300,000 and $1,500,000" is unlikely to be met with approval.

Suggested techniques for addressing the dilemma

How does a project manager resolve the dilemma of needing an accurate point estimate when only an estimate range is possible? We have no good answers, but here are some suggestions that may help:



Note: At the outset of the project, the size of the project can only be estimated within a broad range. As the project proceeds, the estimates can be refined to a narrower margin.

Figure 14-3. Range of Size Estimates Over Time

- Start with an overall schedule and budget. But use a phased contract approach, in which development of only a small part of the system is "turned on" at any given time. Perhaps the first phase would be for detailed design of a subsystem. An output of this phase would be an estimate of the schedule and budget for developing the software (coding, testing, etc.) for this subsystem. Since the design is in hand, the estimates should have a narrower range than any previous ones based only on requirements.

Themes *These suggestions are further justifications for the themes of **bite-size pieces** and **flexibility**.*

- Carefully track progress and use that as feedback for future estimates. Subsequent phases can be estimated more realistically based on the experience of the initial phase. Even if you don't go with a phased approach, the actual progress can be plotted against time and compared with the planned progress. That way you can update the scheduled completion date and get a more realistic estimate for it. That is generally better, than slipping the project one day at a time or unrealistically hoping to make up for lost time. (See also the "How to handle schedule slips" section in *Project Management,* chapter 17.)

- *Two flawed practices are common with software schedules:*
  - *They are established independently of requirements*
  - *They are set in the impossible-to-do zone*
- *Develop a schedule that realistically matches the requirements and what you've set out to accomplish. Don't use best case estimates; they won't be met. Pessimistic estimates often turn out to be the most realistic ones.*
- *Adjust the schedule, requirements, and budget so that they are consistent with one another throughout the project.*
- *Stretching out a realistic schedule is one of the most cost-effective ways of lowering the cost and overall development effort of a project.*
- *Use well-defined "yes/no," "done/not done" milestones.*
- *Get as many independent size estimates for the system as possible, including those of the contractor and the software expert on your team. Resolve the differences.*
- *Use feedback on actual progress to derive more realistic schedule estimates for future activities.*

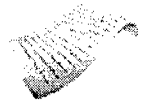
CHAPTER 15
ACCEPTANCE TESTING

The purpose of acceptance testing is to formally validate that the system meets all of its requirements. Planning for acceptance testing, along with planning for training, operations, and maintenance covered in the next chapter, serves as the bridge between the development of a system and its operation. Recognizing that software is never "done", acceptance testing addresses the question, "When is the system good enough to begin using it?" On many acquisitions, system acceptance is accompanied by handing off responsibility for the software from the contractor to the customer or from one contractor to another. When a system passes acceptance testing, it indicates the contractor is entitled to payment for its services.

It is important to understand that passing acceptance testing does not mean the software is 100% error-free. All software contains hidden problems ("bugs") of various severity that will surface from time to time. Bugs may not be found during acceptance testing because they occur only when unexpected data or an unusual series of events occur. Sometimes the "bug" is really the system reacting as it was designed to react, in a rarely encountered situation not clearly covered by the requirements. Whatever the source, the contractor cannot be expected to fix them for free forever. How you intend to deal with them is part of your maintenance strategy and is discussed in the next chapter (*Training, Operations, and Software Maintenance*).

We are recommending a more formal and better documented acceptance testing approach than is customary in the transportation community, and that formal approach must be integrated into the acquisition processes very early. To accommodate this approach, your planning will have to take acceptance testing into account before a contract is signed and even before an RFP is issued. Even if the contracting mechanism that you choose allows for the acceptance test to be developed jointly with the contractor, it must be planned for in the initial stages.

This chapter discusses *formal* acceptance testing only. It does not address other types of software testing that necessarily take place as well. For example, there will be various levels of developmental testing carried out on pieces of the software as they become available. For the most part, these *informal* tests are the contractor's responsibility and do not directly affect the customer.



You may decide not to ignore the informal tests completely. You may choose to witness some of this testing and use it as one means to informally monitor progress. Further, during the selection process, you could ask the bidders to describe their informal testing approach along with other aspects of their software engineering process. Although there would be no one "right answer," the responses may indicate the experience and "maturity level" of the various bidders.

Acceptance testing is requirements-based for software systems

On most transportation projects, acceptance testing is based on the design specs. In contrast, for software systems the requirements document serves as the basis for acceptance testing:

- To protect the customer, explicitly trace every requirement to one or more acceptance tests. This ensures that there is adequate testing coverage.
- To protect the contractor, explicitly trace every acceptance test back to one or more requirements. This ensures that extra requirements are not being "slipped in", and the system is not being asked to do more than was required for it.

A corollary is "tie the tests to the requirements and test only to the requirements."

Two acceptance testing approaches

In the ITS community, two approaches appear for acceptance testing. The first one places the system into operational use to see how it performs. The criterion for acceptance is the system performing in accordance with its requirements for a specified period of time. The second approach, the one we will focus on in this document, is more formal. It carries out end-to-end tests on as much of the total system as is possible and reasonable. These tests are specifically constructed for use during acceptance testing and are conducted in accordance with an approved test plan. The criterion for acceptance is the successful outcome of these tests.

In the preceding paragraph, we presented the selection of an acceptance testing approach as being a choice between two extremes. In practice, a mix of the two approaches may be right for your system. For example, acceptance of the system could be contingent upon its passing a set of formal tests followed by successful operational use for a limited period of time, say one month. Also, it may be useful to acceptance test a limited deployment of the eventual system, prior to providing the contractor with the go ahead to deploy the entire system. For example, a limited deployment could be formally tested using only one equipped bus for a transit vehicle tracking system, or a single ramp meter for a freeway management system. Passing these tests would provide contractor go-ahead to install the rest of the system on the remaining buses or at the remaining entrance ramps. Acceptance of the full system would then be contingent upon formal system tests and its successful operation over a period of time.

Incorporating acceptance testing into the project

Whichever testing approach, or combination of approaches is used, the approach should be decided upon up front, and incorporated in the project plan, project schedule, RFP, and contract. Let's discuss each of these in turn.

The *project plan* (See *Planning the Project,* chapter 8) includes a section that briefly describes the overall acceptance testing strategy, with the testing approach that has been chosen. You may also wish to address such high level testing issues as

- Where will acceptance testing will take place? (e.g., at a contractor location or an operational site)
- Who will participate in the testing process? (contractor personnel versus operational personnel)
- What are the overall acceptance criteria requirements that will need to be satisfied?

The *project schedule* shows the period of time during which the acceptance tests will be conducted. This includes the time needed to analyze the test data. The schedule should allow a reasonable amount of time for the customer to review test results and make a determination as to whether the system meets the overall acceptance criteria. Time should also be set aside for making repairs and repeating some or all of the acceptance tests in the event that the system fails on the first attempt.

One ITS customer complained that the development contractor wanted the customer to accept the system immediately upon conclusion of testing. Since, under many contracts, contractors don't get paid until the system is accepted, it is understandable that they don't want any unreasonable delay. However, sufficient time to analyze the results should be in the test plan and schedule, and agreed to by all.

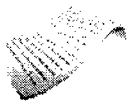
The schedule should also address *test planning*, with the delivery of a collaboratively developed acceptance test plan shown as a milestone. Allow adequate time for customer review and approval of this important document. Do not promise a turnaround time that is not possible to meet!

Schedule test planning and preparation activities to begin as early as possible after contract award. Recognize that the plan will evolve as requirements are fleshed out (e.g., during a requirements walk-through). Test planning and preparation will proceed throughout the development period in parallel with the system development activities. Why is this necessary? For one thing, acceptance testing takes considerable time and resources to plan; you cannot wait until the last minute. Test cases must be designed, test software written, and test facilities established. Also, experience shows that activities near the end of a project often get squeezed, and this is particularly true for testing. So it's best not to wait until near the end of the project to plan the testing and write the testing documentation. Another advantage of planning the testing early on is that it helps focus on the meaning of the documented requirements, thereby providing feedback into the requirements process. As discussed in *Requirements,* chapter 9, good requirements are testable. If a requirement is not testable, it is not adequately defined. By asking, "how will we test this requirement?", questions like "what does this requirement really mean?" get raised. The earlier the answer to that is known, the better. If a test can't be devised to determine whether a *shall* statement has been met, then perhaps that requirement should be recast or deleted. For quality factors that are difficult to measure, ask the contractor to suggest a validation method.

From a legal perspective, acceptance testing cannot be unilaterally imposed on the contractor at the end of a project. Therefore, you must address acceptance testing in both

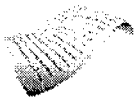
the RFP and contract, reflecting the overall testing approach given in the project plan. Explicitly call out test planning activities and test plan deliverables, too. At a high level, the RFP and contract should identify roles and responsibilities for acceptance testing and test planning, identifying customer responsibilities and assigning any contractor responsibilities in this area: Who will write the test plan? Who will conduct the tests? Who will analyze test results? Who determines whether acceptance criteria have been met? (More detailed answers to these questions will be needed in the acceptance test plan.) We also recommend that the contract call for both the customer and contractor to sign off on an approved acceptance test plan.

You should also define the overall system acceptance criteria in the RFP and contract, with the understanding that the detailed criteria will be jointly developed. As an example, the criteria might be that the system has to pass all the tests designated as critical and eighty percent of the remaining tests. The test documentation will go into more detail, providing pass/fail criteria for each individual test.



Including acceptance criteria in the RFP and contract protects both customer and contractor. Historically, customers complain about poorly functioning systems that do not meet expectations. At the same time, contractors complain that without formal acceptance criteria, public sector customers have no incentive to accept a system and continually ask for more and more enhancements to that system. In the meantime, the contractors don't get paid.

One final reason for making planning for acceptance testing a priority early is that some software requirements may result from test criteria. If, for example, an algorithm is to be tested and the test plan indicates that certain input and output data are to be viewed as part of the test, extra software "hooks" into the data stream may be needed to capture the data for the test. Often such hooks can be easily incorporated early in the design and development, but only with greater effort later on.



Another item for consideration that was suggested by our interviewees was distinguishing between conditional sign-offs (after formal acceptance testing) and full acceptance (after a limited period of operation). After the formal acceptance test, the customer withholds 10 to 20 percent of the contract costs to be released after full acceptance. The bottom line is that if conditions similar to this are to be reflected in the contract, they need to be thought through up front, before an RFP is issued. And remember: money costs money! The contractor has already paid the software developers, but has to wait for payment (in effect, reimbursement) until the acceptance period of operation is over. The cost the contractor incurs by using internal funds will be included explicitly or implicitly in their bid.

A note of caution

Be very careful about accepting the software independently of the rest of the system. Although software is just one more component, it is the "glue" that holds the rest of the system together. Yes, this presents the chicken and egg dilemma: It is best to test the software with as complete a hardware configuration (sensors, variable message signs, surveillance cameras,

communications infrastructure, etc.) as possible. On the other hand, you don't want to accept the hardware until you are sure it supports the software you are acquiring. There are risks, either way.

Having the hardware available early in the software acquisition can add to the costs. Since the hardware is typically the most expensive part of the system, there are advantages to acquiring it as late as possible. (By "hardware" we're including the field devices and other components of the system. In this context, the computing platform hardware may not be a significant cost item, for it is often less expensive than the software it hosts.) This is particularly true if the software is delayed. In the worst case, if the project were cancelled because of software problems, you don't want to be stuck with a full complement of unused hardware devices. Furthermore, hardware technology is advancing rapidly. Deferring hardware purchases may enable you to take advantage of some of the advances that occur during the period that software development is taking place.

On one large Federal acquisition, the customer accepted the computers as the first deliverable. The software wasn't ready, so the computers were put into warehouses. The customer then had to pay for several years of storage costs while the software development encountered the inevitable delays. By the time the software was finally made to work, the hardware vendor no longer provided hardware maintenance or operating system support for the aged computers.

On the other hand, if the hardware acquisition is deferred, or if the components are not available for software acceptance testing, the software may have to be accepted before it runs as part of the complete system. The obvious risk is that the software won't work with the complete system: when the hardware—the most expensive parts of the system—is delivered, accepted, and paid for, you find that the software doesn't work. Since software is the "glue", nothing useful comes from the system as a whole. You will have paid most of the project funds and yet have nothing that is usable. Even if the software is made to work eventually, the hardware has aged and you are not going to be able to take advantage of the rapid gains in technology that occur in the meantime.

Often field hardware and communications systems are accepted with limited testing and when an attempt is made to integrate the software, many problems are encountered. One integrator complained: "We frequently find ourselves in a position of having to spend project resources investigating a 'software problem' that turns out to be a communications or hardware problem." Since the hardware has been "accepted," the software is (sometimes unfairly) "blamed." This can lead to unplanned, unbudgeted activities on the part of the contractor(s) to debug the system. Identifying the nature of the problem is greatly aided by a teamwork approach.

A better alternative is to accept the various subsystems, with each subsystem having the software, hardware, and communications components.

Testing is a teaming activity

Many of the team members can and should participate in acceptance testing. End users and system administrators should have inputs into the plan. Any software technical

expert(s) on your customer team can assist in preparing and reviewing test plan documentation. As noted above, regardless of who writes the test plan, the customer and contractor should both sign off on it. Consider having several members of your team, including end users, sign off on the plan on behalf of the customer. Similarly, the customer and contractor should jointly conduct the actual tests. One party can carry out the tests while the other records the data or monitors the activity. End users and system administrators can participate in testing the functions that pertain to them. The software technical expert(s) on your team should also participate in the testing activities. Make sure the contract allows for this participation and any training of customer staff that is needed prior to the test.

Types of formal acceptance tests

Assuming you decide to go with formal tests, what types of tests should be carried out? Acceptance testing should comprise a range of thorough and rigorous testing, not simply benign tests that show, for example, that the system can communicate with a field device. That is a demonstration, not a test. Acceptance testing should be carried out to show that the system performs in accordance with its requirements, and with all field devices. Stressing the system is a good idea:

- If a requirement calls for only numeric parameters to be entered in a field, test the system with nominal values of those parameters, but also with extreme values of the parameters and erroneous values of the parameters (e.g., entering "11" for a parameter when only values from 1 to 9 is allowed; entering a letter when only numeric constants are allowed).
- Measure system performance (e.g., ability to respond in a certain amount of time) under heavy load (large number of users, transactions, inputs, outputs, etc.). If there is a load requirement, make sure the system meets that requirement.
- You may also want to test the system to see what happens when the system is overloaded—does the system degrade gracefully? or does it simply crash? (If you test for graceful degradation, make sure that graceful degradation is reflected in the requirements.) Another option is to stress the system until it "breaks". (The corresponding requirement would be for the breaking point to exceed a certain threshold.) The ability of a transit management system to support a fleet of at least 100 buses would be an example of this.

In addition to carrying out tests on the operational use of the system, conduct tests on systems administration. For example, how long does the system take to make a "cold start"? Or, what happens if the communications line to a device goes down? (This could be simulated by turning off the appropriate modem.)

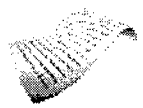
The above items are not meant to be complete, but are presented to provide a flavor for the formal acceptance tests that can be used.

The following are the types of tests that should be carried out for software systems [Royer, 1993]:

- *Functional tests:* the ability of the software to transform its inputs into desired outputs. Functional tests generally comprise more than half of all tests conducted. They could include some of the simple demonstrations, such as the example cited above of communicating with a single device as well as more rigorous tests. If there are a multiple number of devices, phone lines, displays, etc., each should be independently tested to show that it functions properly. (This is an example of multiple test cases all following the same test procedure.) Other functional tests would show that the system can correctly input sensor data, process it (perform calculations), generate the proper outputs, and store it appropriately. The ability of the system to display proper error messages when failures are inserted should also be tested. Year 2000 compliance may also be tested. (See *The Year 2000 Problem (Y2K),* topic sheet 6.)
- *Maximum capacity and stress tests:* what happens when the system is heavily loaded, such as when all the operational positions at a traffic management center are actively in use, or all the vehicles in a transit fleet are transmitting data. Tests should be performed to ensure that the system continues to function and that it meets required response times when subjected to heavy load. Often these tests will require the use of simulation software that artificially generates inputs at high rates. The schedule and resources must be allocated to develop (and test) the simulation software, which must be developed in addition to the regular operational code.

How do you go about stressing a transit system when there are only a few equipped vehicles available? You need assurances that the system will continue to function satisfactorily as more and more vehicles gradually come on-line over a period of time. Simulation software can be written and used for this purpose. It generates a load simulating inputs from a full fleet of transit vehicles. Unless the simulator was already available, writing such software would have to be one of the scheduled tasks. Similarly, hardware test simulators may need to be specified.

- *Erroneous input tests:* entering data beyond allowed ranges.
- *Stability tests:* the ability of the system to continue its operation over an extended period of time without intervention. Also called *continuous operations testing*. Often these tests are some of the most difficult ones to pass.



A Navy contractor proudly demonstrated the system they had developed. They ran it through its paces and it apparently met all of the requirements. However, the wise old admiral was not impressed. "Leave the system running all night long," he said, "and we'll see how it's doing in the morning." Alas, the system crashed during the night. It took several more months of development before the system was reliable enough to field.

- *Integrity tests:* the ability of the system to prevent unauthorized access.

How thorough should the testing be?

In deciding what tests to conduct, it is important to keep the overall objectives of the project in mind. Previously, we recommended periodically revisiting the system concept in your project plan to help control requirements creep. Similarly, the system concept can serve as a guide for how much testing is needed. If a project is intended to control a corridor for many years, then it should be thoroughly tested. However, if the project is to develop throw-away software for an operational test, then only test it for its ability to perform the operational test. A system that may not be acceptable for long-term operational use may suffice for an operational test.

In determining the level of performance needed to pass the acceptance test, keep in mind the limitations of legacy or other existing software within the system. Nearly all software, including off-the-shelf "shrink wrapped" software, will not prove to be crash-proof and will fail given strenuous enough testing. For example, trying to hold software applications to a higher availability than can be expected from the operating system is clearly not feasible. This is true even if the operating system is "shrink wrapped".

Acceptance test documentation

Acceptance testing should be sufficiently documented to ensure that test results will be reproducible if a test is rerun. Acceptance test documentation is grouped into five parts: a test plan, test procedures, test cases, test logs, and test results. How these parts are actually packaged—whether combined into one document or bound separately—is not a key consideration.

The *acceptance test plan* summarizes the overall approach to acceptance testing. It fleshes out the various testing issues that were previously addressed in the project plan, schedule, RFP, and contract. Begin writing the plan shortly after contract award. The customer can take the lead in writing the plan, or the contractor can. In either case, the plan must be collaboratively written and should be a formal document, signed off by both the customer and the contractor and placed under configuration control.

The *test procedures* are detailed step-by-step instructions for carrying out the tests. The same procedure may be used several times with different sets of input data. Each set of input data is termed a *test case*. Each test case addresses one or more requirements. This can be ensured by tracing test cases to tests which, in turn, are traced back to the requirements. (cf. *Acceptance testing is requirements-based for software systems,* page 15-2.) The *test log* is a simple form to record outputs of tests as they are run. The *test results* are the report of your findings. The test-by-test results could be combined with the test logs. The overall results on whether the system passed could be a separate document, perhaps just a short memo.

Checklists 15-1 to 15-5 suggest items to include in the acceptance test documentation. The following paragraphs provide some more details.

As discussed above, the acceptance test plan is a formal document. The degree of formality of the remaining acceptance test documents will be project dependent. The

Checklist 15-1. What to Include in the Acceptance Test Plan

	Organizations and their respective roles; who will be responsible for:
✓	Conducting the tests?
✓	Recording the data?
✓	Analyzing the data and reporting the results?
	Where will the acceptance tests take place
✓	The contractor's location?
✓	An operational facility?
✓	In transit vehicles?
	Testing schedule (should allow time for data to be analyzed; may want to include a dry run phase)
✓	Computers to run the tests.
✓	Field devices (e.g., installed variable message signs; bus sensors).
✓	Other systems (e.g., legacy systems; systems in neighboring jurisdictions).
	Software needed
✓	Special test software (simulators to stress the system, spreadsheets to analyze results, etc.).
	Overall system acceptance criteria (Note: the pass/fail criteria for an individual test are listed below under "List of tests to be run.")
✓	Acceptable failure rate (e.g., pass all the tests designated as critical and 80 percent of the remaining tests).
	What happens when tests fail or do not proceed as planned
✓	Role of regression testing
	List of tests to be run; for each test, indicate:
✓	Test identifier (e.g., Test 1A)
✓	Purpose of test (brief statement)
✓	Data to be recorded
✓	Pass/fail criterion (depending upon the test, the test case may be a better place to provide this information).
	Traceability
✓	For each test, show to which requirement(s) the test traces (shows that test is requirements-based; can be viewed as protection for the contractor so that the new requirements are not "slipped in").
✓	For each requirement, show to which test(s) the requirement traces (shows the coverage of testing; can be viewed as protection for the customer so that all of the requirements are tested)

Checklist 15-2. What to Include in the Acceptance Test Procedures

	For each test, include the following:
✓	Pre-test activities needed to set up the test (e.g., turn on or off certain pieces of equipment, load a piece of software).
✓	Step-by-step procedures used to carry out the test.
✓	Procedures used to reduce and analyze the data (explicitly state equations, statistical formula, averaging techniques, etc.)
✓	Computers needed (to run tests; to analyze results)
✓	Field devices needed to run tests (e.g., installed variable message signs).
✓	Other systems (e.g., legacy systems; systems in neighboring jurisdictions).

Checklist 15-3. What to Include in the Acceptance Test Cases

✓	Input values
✓	Source of input (manual entry, field device, simulated data, etc.)
✓	How long the test is to be run (e.g., collect loop detector data for one hour)
✓	Expected value(s) of the output
✓	Pass/fail criterion for the test (depending upon the test, the test plan may be a better place to provide this information)
✓	Traceability between test cases and tests to ensure that a test case exists for all tests.

Checklist 15-4. What to Include in the Acceptance Test Log

✓	Name of test
✓	Date and time test started
✓	Date and time test ended (only for tests that last for extended duration; for many tests only the start time is needed)
✓	Who carried it out
✓	Any deviations from the test procedures (e.g., test conductor inadvertently left out a step; error was found in test procedure, so it was modified "on the fly")
✓	Recorded outputs

Checklist 15-5. What to Include in the Report of the Test Results

✓	Overall information (e.g., when the tests were conducted)
✓	Overall report on whether the system passed and what follow-on steps are needed
	Test-by-test results
✓	Test identifier, also indicate which test procedure was used and which test case was used
✓	Any deviations from the test procedure
✓	Recorded data from the test log
✓	Computed data
✓	Whether the test passed or failed (in accordance with the documented criteria)

same is true for whether there will be several pieces of testing documentation or whether the various items are merged into a single document. In any case, the RFP needs to call out any documents that are contract deliverables. You may also decide to require preliminary versions of the testing documentation to be issued at various times. These can be used as the basis for informal tests that allow you to "kick the tires" of your system as it is being assembled. You can use the results of this testing to give feedback to the contractor.

Themes

*For the feedback to have any value, there has to be some **flexibility** built into the acquisition process so that the contractor can accommodate it.*

As shown in checklist 15-1, the test plan defines what happens when tests fail. This topic is somewhat complicated. There are several alternatives for dealing with a failed test: the test can be rerun after corrections are made to the system or the entire suite of tests may be repeated. This latter process is referred to as *regression testing*. Corrections may have unexpected side effects in other parts of a system. A "fix" that corrects one problem may inadvertently introduce a new one. Regression testing to show that all tests pass successfully is, therefore, often a good idea. However, rerunning all the tests can be a costly proposition. Here's a case where common sense and available resources will have to dictate to what extent regression testing will be used.

The recorded outputs portion of the test log could be numeric values or simply confirmation that some event occurred, for example a status light went out. For some tests the outputs may not be recorded on the form but would consist of supplemental material such as printouts of computer data. In some cases, the outputs would correspond to the test results and the outcome of the test would be known immediately. For example, the status light either did or did not come on. In other cases, the outputs may have to be analyzed later (e.g., computing an average value or performing a statistical test) to determine whether the outcome of the test was successful. In some cases, the outputs (and possibly a pass/fail indication) may be recorded separately from the rest of the logs. In that case, the logs would contain only the administrative data such as the date and time of the test.

- *Plan a formal acceptance test strategy, including the use of formal documentation, before an RFP is issued. Reflect this approach in the contract.*
- *Schedule test planning and preparation activities to begin as early as possible after contract award, and to proceed throughout the development period in parallel with the system development activities.*
- *Base acceptance testing on the requirements.*
- *Carry out several varieties of rigorous testing; simple, benign tests are not sufficient.*
- *Carry out testing as a teaming activity.*

CHAPTER 16
TRAINING, OPERATIONS, AND SOFTWARE MAINTENANCE

You will probably look forward to the day when the software has been completely developed and you can begin to operate the system. However, there are a number of *support activities* that must be considered in conjunction with system operations. Two primary support activities are training the end users who will operate the system, and maintaining the software. Other support activities include system administrative functions to keep the system running, and training for those who will be involved in the administrative and maintenance activities.

The general experience is that on many software projects, the operations and support activities are often forgotten until late in the acquisition life cycle. Actions that should have occurred earlier, during software development, are not accomplished, making it more difficult and costly to carry them out later. Then they are done in a rush, in order to meet deadlines for fielding the system. In this rush, key issues are not adequately addressed or even forgotten until the system is delivered.

ITS appears to be no exception to the general experience. One ITS manager told us that "maintenance kind of caught us by surprise."

This chapter is intended to give you a "heads up" for what the support activities entail.

Themes

*Even though they don't take place until late in the project, operations and support (training and maintenance) must be prepared for early, and documented in the project management plan (**up-front planning**). Support requirements and activities must be addressed in the RFP and contract. Preparation for support (setting up facilities, etc.) is then carried out in parallel with the software development activities.*

Planning for the support activities

Issues that need to be addressed during planning to ensure proper support for system operations include:

- How to implement the system, integrating it into the operational environment
- How to operate the system effectively
- Staffing the system
- Training the end users in the operation of the system
- Providing on-line and telephone support for timely response to operational problems

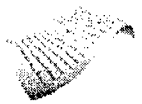
- Answering questions from users and operations staff
- Making sure that any off-the-shelf software used in the system will be supported when the system is delivered. For example, are proper licensing agreements in place?
- Software maintenance
- Ensuring that a support system is available, along with a facility to house it. Without access to a support facility, the system maintainers may have to bring down the operational system whenever maintenance activities are performed.
- Migration planning for upgrading the system

A key decision for each of these is determining who will carry it out.

Let us examine some of these issues in more detail.

Training

There are several types of training. First, there is the training that is required for the end users to operate the system. Clearly, they need training on how to invoke various system functions, on "which buttons to push." But they also need training on the overall system concept and how to respond to various situations. In other words, users also need domain knowledge so they know "when to push those buttons."



End users of a freeway management system will need domain knowledge on how to interpret the various displays and maps. End users of a transit management system will need domain knowledge on how to react if a bus runs behind schedule.

Another type of training is training on what it takes to maintain, support, and administer the system.

You must also decide who will conduct the training: the development contractor, a separate support contractor, or your in-house staff. If you decided to carry out your own training in-house, then that begs the question, "Who trains the trainers?"

Finally, be sure to get the rights to all training materials.

Support for off-the-shelf software

When off-the-shelf software is used in a system it presents special issues, both from a development perspective and a support perspective. We previously addressed the development issues. (See *Build/Buy Decision(s),* chapter 10.) Let's concentrate on the support issues. Off-the-shelf software is normally maintained (or not maintained) by the supplier. Ordinarily only use licenses are granted. Don't expect to be given the intellectual property rights that are necessary for carrying out in-house maintenance. The supplier might not provide what we would consider to be periodic maintenance (i.e., the identification and correction of errors). The supplier will hopefully provide fixes in new

versions of the software. These are normally "marketing" enhancements, that may not respond to your problems. Even if these enhancements do address your needs, they present a new issue—should you upgrade to this new version of the product?

Arguments in favor of upgrading include:

- The desirability of having access to new system functionality
- Bug fixes
- The consideration that staying with an older version may obsolete the product capability or cause you to lose access to vendor support.

Arguments against upgrading include:

- The operational impact of incorporating the change. Will the rest of the software integrate smoothly with the new version? In some cases, new hardware may be needed to host the new software, since vendors tend to take advantage of advances in hardware technology.
- The new version may not be backwards compatible with the previous one. Other system components may depend on features in the older version that no longer exist in the newer one. In other words, a simple upgrade to one piece of software could make the rest of the system "break".
- The need to train the users on the new version of the software because of changes in the user interface.
- Cost of obtaining the upgrade.

Recognize that regardless of the pro's and con's, in many situations upgrading is an unpleasant fact of life that you may just have to accept. If you decide to upgrade, be sure to run test versions of the system with the new software before making it operational.

If the off-the-shelf software is not a shrink-wrapped product, it is generally easier to negotiate a special agreement for support. Normally, the suppliers will provide some form of help service, usually an 800 number. If this is not sufficient, make special arrangements to have more responsive help. A "hot-line" that the user can call if a problem or some operational question arises is one possibility. Special support such as this should only be used when it is absolutely necessary, since it will add cost, and may not be as responsive as desired. It is often better to count on a sound training program that provides operators and users that are trained and can respond to anomalies during operation. As with the shrink-wrapped product, decisions will have to be made on whether to upgrade when a new product version is announced.

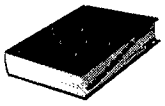
You need to ensure that licenses to off-the-shelf software products are put into place that are appropriate and timely for operational use of the system. Other off-the-shelf software products may be needed to support the system, so licenses for their use are also required. Some questions to consider in regards to what is covered by the support license include:

- Are upgrades included? If so, how many per year?
- Is there telephone support? If so, how many hours per day? how many days per year?
- What is the required time for the contractor to initially respond to your problems by giving you access to trained personnel?

Software maintenance

What is software maintenance?

Except for the simplest systems, software is seldom "done". Instead it is "maintained" throughout its operational life. (Recall from *Acceptance Testing,* chapter 15, that your acceptance of the system does not mean that the system is perfect.) In fact, various studies have shown that over the life cycle of a system, typically 60% to 80% of the overall software costs will be for maintenance, dwarfing the amount originally spent on software development. [Pigoski, 1997, page 30]



***Software maintenance** is defined as the "modification of a software product after delivery." Maintenance includes correcting software faults ("bugs") that are found during system operation (corrective maintenance), improving performance or enhancing functionality (perfective maintenance), keeping the software usable in a changing environment (adaptive maintenance), and keeping the system operational (unscheduled emergency maintenance). [IEEE, 1993b, pages 3 and 4]*

Consider software maintenance in the context of an overall system maintenance concept that includes hardware maintenance. (See *Software Acquisition In A Larger Context,* chapter 2.) Ask yourself whether software functionality depends upon a certain level of hardware maintenance. Often, the software is critically dependent on having most of the field sensors up and running. For example, ramp metering algorithms implemented in software cannot work without the inputs from loop detectors.

The maintenance concept provides a systematic maintenance approach for the system. The maintenance concept deals with the extent of the software maintenance, who will perform it, and an estimate of life-cycle costs. Depending on the size and complexity of your system, the concept can be documented in its own plan or incorporated into the project management plan. The concept addresses the following:

- Establishing a method for reporting bugs
- Making suggestions for new features or extending old ones.
- Assigning priorities, scheduling fixes and enhancements, and seeing that they get implemented.
- Testing changes to see that they work and that to ensure that they don't cause something else to "break"

- Using established configuration management procedures (See *Software Configuration Management,* chapter 18.) to maintain records of:
  - problems and how they were resolved, enhancements, and changes to the software
  - the tests conducts
  - all associated software code and documentation

Who should maintain the software?

As with the other support activities, you have to assign responsibilities for the various software maintenance activities. In particular, who will:

- extend existing capabilities of the system?
- improve system performance?
- add new functions and train users on them?
- find and report bugs?
- correct bugs?
- document changes to the system?

Factors such as long term cost, space, qualifications of the contractor and agency staff, and personnel availability all bear on this decision. In some case, responsibilities can be shared, with some maintenance activities carried out by the contractor, for example, while others would be performed in-house.

In many cases, the most attractive option is to have the software developer maintain the software. After all, they are the ones most familiar with the internal design of the software, and have staff with the requisite skills and the necessary tools for maintaining it. "The best people to maintain a software product are those who know it." If you decide to have the development contractor carry out software maintenance, reflect this as a task in their original contract or in a separate software maintenance contract. Use this section of this document as a guide for the key items to be included in the maintenance contract.

Product vendors complain that their public-sector customers do not use contract maintenance by the development contractor as often as they should. This attractive option is too often dismissed out-of-hand for fear of getting "locked in". (See Differing Perceptions of ITS Software, *chapter 3.) Instead, customers take on in-house maintenance, a task for which they may not have the qualified staff, on software whose internals they are not familiar with.*

If you intend to out-source maintenance to a different contractor, then preparations for this must begin early. Use the material presented here as a guide for items for consideration in selecting the contractor. The maintenance contract must be in place when operational use of the system begins.

If you are considering software maintenance by in-house staff, ask yourself whether you really want to take on that responsibility. Taking on responsibility for software maintenance implies the following:

- Having qualified staff. Note that programming experience on information systems does not qualify an individual to work on demanding, real-time ITS software. Such staff are in short supply. If you currently do have them on board, will your agency salary structure allow you to pay them enough to attract them? If you train existing staff, will you then be able to hold onto them?
- Familiarizing the staff with the internals of the software and the support environment and tools used to maintain it. This will require close collaboration of the maintenance staff and the contractor developers from the outset of the project.
- Taking over such tasks as documentation and software configuration management (See *Software Configuration Management,* chapter 18), for which contractors normally have prime responsibility.
- Setting up and running a support environment (see next section, below)
- Having access to the following items:
  - source code, in compilable computer files; listings are not sufficient
  - documentation on databases, data structures, and interface protocols
  - development tools used to compile the software, keep it under configuration control, test it, etc.

  Note that there are costs associated with having access to these items. The full support environment for a commercial database package, for example, is considerably more expensive than run-time license for the same package. These costs must be factored into your decision as to whether you should contract for maintenance or perform it in-house.

As can be seen on the above list, your software maintenance concept partially determines your needs in regards to intellectual property rights. Remember though, that your intellectual property rights need to be clearly spelled out; possession of code is different from rights to the code. (See *Resolving The Intellectual Property Rights,* chapter 13.) However, you may be able to avoid unnecessary fights over intellectual property rights, if you determine that you won't be able to maintain the code and don't insist on additional rights in the first place. Ask yourself, "If I do take possession of the source code, does my agency have people who are capable of maintaining it? Or will we just turn around and hire maintainers, perhaps the original development contractor?"

In deciding who will maintain the software, examine the documentation to determine whether it will be adequate for you or a third party to support in-house maintenance. You may be forced to go with vendor maintenance even if that was not your first choice at the outset.

Often the customer gets locked into the development contractor for maintenance, yet most of the maintenance activities are simple "tweaks" that produce new outputs (management reports, on-line forms, etc.) or revise the formats of old ones. One way around this is to require that report generators and graphical user interface tools be

delivered as part of the system. With these easy-to-use software tools, the system administrator can revise screen formats for the users, generate new reports when needed, or revise old ones.

You also need to decide where software maintenance will take place. For large systems, you may decide to have the maintainers on site. Space considerations must factor into this decision.

The support environment

The support environment is all of the hardware and software used to maintain the system, including software tools for analysis and testing. This environment is generally different from the one used to develop the system. For example, tools used during development may not be the same as those used during operation, and licenses that the developer has with suppliers may not carry over into support. During the support phase of the system, maintenance requires diagnostic tools to troubleshoot the system, trace tools that can track the flow of data and messages, and special test scenarios. It needs to be clear in the contract that this support environment is distinct from the operational system or from the development environment that the contractor uses at its facilities. If a support environment is to be used, it needs to be called out as a contract deliverable. Its acquisition goes through all the same steps as those used for the operational system—software development, testing, and acceptance; operations; and maintenance—and its software has to be placed under configuration control. Intellectual property rights issues also have to be addressed.

There is also the issue of having a place to house the support environment. One option is to have the contractor carry out support on existing facilities at its own site. However, if a new, separate facility needs to be established, the support facility should also be called out as a contract deliverable. After all, the developer's environment may be part of their larger facility, and not available for transition into a support facility. The rights to the data associated with the facility need to be specified so that the customer acquires those rights.

The support facility has similar integration requirements to those listed above for the operational system, including adequate space, air-conditioning and power, scheduling, budgeting, and staffing.

Personnel roles

An early step in planning for support is to identify all the personnel roles that will be needed. The end user operator is an obvious one, but other equally necessary roles may not be so apparent at first. Checklist 16-1 has some roles to consider. Clearly some of these roles can be combined. For example, the administration role can include diagnosing problems and contacting hardware vendors to make needed repairs.

Next, and most importantly, give thought and consideration to specifying, as clearly as you can, the needed skill and experience levels of the personnel who will perform the

Checklist 16-1. Personnel Roles Needed For System Support

✓	Shift supervision of the end-user operators
✓	Administration of the system to keep it running
✓	Generation of management reports from the system
✓	Review of data produced by the management reports
✓	Installation of new computing hardware or displays
✓	Integration of additional field devices into the system
✓	Fixing software bugs
✓	Upgrading the software – Installing new releases of off-the-shelf software – Adding functionality to custom software – Modifying control algorithms
✓	Diagnosis of problems and repairing hardware
✓	Training for all the above, both when the system first comes on-line and, later, when personnel turnover occurs

support roles. Then decide who will be responsible for carrying out each of these roles. If it's the development contractor, that responsibility must be reflected in the contract. If support is to be performed in-house by the customer, then staff may have to be hired. If a separate support contractor is to be used (i.e., distinct from the development contractor), then acquisition activities for this contract must begin early, so that this contractor will be available at the completion of development. These may be used in combination; for example, training could be performed in-house, while software maintenance could be done under contract.

The support organization must be on-board, trained, and ready to take over the operation of the system once it has been accepted. There can be a transition period as the system is handed off from the development organization, but this role must be planned and specified in the contract.

The support budget

Support has to be budgeted up front and decisions as to who pays for support settled. If the buyer is not going to provide total support that should be made clear and agencies that are expected to pick up support notified beforehand so that they can budget for the support and provide the necessary facilities and resources. In either case, the life cycle cost estimate for the system should include the cost of support.

Support costs have to include training, obtaining a support environment (if needed), licenses, and maintenance costs.

Training costs. Estimate the number of people who have to be trained, including users, administrators, and operators. Include the costs of the trainer, materials, and staff time charges.

Support environment. The cost of the support environment is usually specified in the developer's cost proposal. Remember to include on-going maintenance costs of this facility, license renewal costs, and space costs if you take over this facility for maintenance.

Maintenance costs. Traditionally, in large system development, the cost of development accounts for only 30% or less of the life cycle cost, while support over a ten year period or so accounts for the other 70%. This support is not merely fixing errors that are found during operation, but enhancing the system based on new requirements and more efficient operation. In fact, fixing errors typically accounts for only about a fifth of the total maintenance effort. [Pigoski, 1997, page 34] There are several ways to estimate this support cost. Here are three:

- A quick way is to take a percentage of the number of developers. If 20 persons were used to develop the software, a range of maintainers would be anywhere from 20% to 50%, or 4 to 10 persons. Naturally, this is a level of effort that must be converted to a dollar figure. At the bottom level one would only expect a minimum level of error correction and minimal enhancements. The top level would provide for a higher level of enhancements and allows for a more complex

support environment. Remember, it also takes resources to operate the support facility.

- A second method is to use the 30%/70% rule. If we assume that 20 persons is 30% of total life cycle cost then 70% of total life cycle cost is approximately 46 persons. If the period of support is ten years then approximately 5 persons are required. We can see that this quick and dirty method is within the bounds of the first method.
- The third method relies on more scientific cost estimation tools that calculate the maintenance cost on the basis of the estimated level of change to the code.

- *Plan for support activities early, and reflect your approach in the contract. Prepare for support in parallel with development activities.*
- *Allow adequate budget for support activities to take place. Over the life of the system, support activities generally consume more budget resources than do the development activities.*

PART FOUR:

ON-GOING MANAGEMENT ACTIVITIES

CHAPTER 17
PROJECT MANAGEMENT

This chapter discusses the management activities that take place once a contract is issued. The contractor is of course responsible for the technical activities associated with software development, whether that be building custom software, tailoring pre-existing products, or integrating off-the-shelf products. However, as customer, you also have a vital role to play. As we've stressed elsewhere throughout this document, software acquisition is a collaborative process. You and the other members of the customer team will need to be actively involved with the contractor on such *technical* activities as rapid prototyping, revisiting the requirements, and acceptance testing, which are discussed elsewhere in this document. This chapter focuses on the complementary *management* activities that you will need to perform. The degree to which they are carried out should be geared to the size and complexity of the acquisition.

Themes *This chapter shows how the **active customer involvement** theme manifests itself after a contract is issued.*

Managing risks and gaining adequate visibility into contractor progress are the two key aspects of managing the contract. Software Risk Management is discussed in chapter 19. Here we focus on what can be done to gain visibility into the development activities. As we discussed in *The Nature of Software,* chapter 1, one of the complaints commonly expressed by project managers is the difficulty in gaining visibility into the software development progress. This chapter is intended to help in this area. We also discuss corrective action tracking, what to do in case the schedule slips, requirements management, quality management, and managing the expectations of others.

Techniques for gaining visibility

The customer can utilize three sets of management tools to monitor the development effort: project reviews, document reviews, and measurement data such as cost and schedule data.

Project reviews

Project reviews are held to determine status and to surface issues and problems, but not to solve them. Conduct your project reviews in a spirit of cooperation and not in an adversarial fashion. They can be formal or informal. *Formal reviews* are scheduled events specifically called out in the contract.[1] These can be used to track progress

---

[1]Under a fixed-price contract, the contractor would naturally be resistant to participating in reviews that are not specifically called out in the contract. However, under a time-and-materials contract they may be less resistant, since they will be paid. Thus time-and-materials contracts may offer some flexibility and

against an agreed upon set of metrics. They can also be used to review the contractor's software engineering process for adequacy and for adherence to that which was proposed. *Informal reviews* are those meetings such as technical interchange meetings held on a periodic basis. Attendance should be limited to those directly involved, so that the review does not degrade into a "dog and pony show."

Provision for customer participation in project reviews should be clearly stated in the contract. The contract can also allow customer attendance at some of the contractor's internal reviews on such areas as requirements, design, and risk. This is useful, provided a teaming environment has been achieved. Otherwise, there may be added cost and no value if the contractor holds their "real" internal meetings separately, away from the customer.

Document Reviews

There are also two types of document reviews. The first type is a formal review. A formal review can be a project milestone. A formal document walk-through by members of the customer and contractor team is an example of this. *Requirements Management,* chapter 9B discusses formal walk-throughs in regards to the requirements document. Alternatively, a formal review can also be conducted in conjunction with another milestone. An example would be the review of a design document as part of a design review.

The second type of document review is the review of documentation that is produced by the contractor for its internal use. Such documentation is normally not required as a formal deliverable, but is produced as part of the development effort. An example might be a software development file. These are informal documents that record the activity associated with the development of a unit (of code) or of a software component. They are normally checked through inspection by the developer's quality assurance process. Results are recorded and open to inspection by the customer.

There are three good practices to keep in mind in planning document reviews:

- Prior to a document review, circulate copies of the document to all the participants to help them prepare for the review.
- A formal document review should generally *not* be the first opportunity that the customer has to see the product. Instead, there should be continual informal reviews and opportunities for feedback between customer and contractor. If the resulting products are collaborative ones, there should be few surprises at the formal review.
- Do not defer document reviews until the end of the project. When this happens, there is no opportunity to take corrective actions that are uncovered by the review. As a result, the review either becomes a meaningless exercise, or it results in an almost guaranteed schedule slip.

---

informality, in that not all the management events need to be planned up front; they can be scheduled as the need arises. The activities, of course, still must be budgeted for.

Measurement data

Quantitative measurement data (sometimes called "metrics") are used to provide project status and are often discussed during project reviews. Metrics can take many forms. Let us examine some of them. The first two apply to all types of contracts, not just software:

- *Costs* are reported as a normal part of the contractor's financial reporting system. These may be as simple as labor hours expended to date to more extensive systems based on earned value. Cost data are normally received because cost and resource data associated with financial reporting are required for progress payments to be made to the contractor.
- *Schedules* can range from a simple milestone chart to a PERT chart.
- *Software-specific metrics.* Table 17-1 lists some metrics that are commonly used for software. (See also [MITRE Corporation, 1985].) Plotting expected and actual values against time creates a useful tool called a management indicator. A variance in one management indicator does not necessarily mean that action is called for, or that a real problem exists. An example of a management indicator is shown in Figure 17-1.

Contract language is needed for gaining access to measurement data. While it is generally not good practice to detail the exact metrics that will be used by the contractor, it is reasonable to ask the bidders what metrics they use. There's no one correct answer, but their responses may give you some indication of the maturity level of their software development practices. (See also *Software Capability Maturity Model (SW-CMM),* Topic Sheet 4.)

Other techniques

Some of the techniques that we've discussed elsewhere can also be used to provide increased visibility into the project:

- Rapid prototyping gives insight into how the final system will "look and feel". (See *Rapid Prototyping,* Topic Sheet 1.)
- Iterative development and multiple builds allow progress to be monitored by watching the system evolve. The trick is to not put too much functionality into any one build.
- Continual open communications with the contractor allows problems to be surfaced and addressed as they occur.

Gaining visibility into subcontractor activities

One of the most difficult aspects in managing an acquisition is getting visibility into the activities of the subcontractors. The prime contractor may even intentionally try to insulate you from the subcontractors. This situation can lead to disaster, especially if the subcontractor is responsible for critical parts of the project—the software, in particular. There are ways to prevent this from happening.

Table 17-1. Software Metrics

Development Progress

The actual and expected number of units or components, plotted against time. Separate plots are generated for designed, implemented (coded), and integrated units or components.

Test Progress

The actual number of tests that have been conducted, and the scheduled number of tests, plotted against time. The number of tests passed or failed can also be plotted against time. Test aging data on failed tests that have still not been passed can also provide insight. If long periods have elapsed since the test initially failed, that may indicate "sticky" problems that cannot be fixed, or lack of necessary resources. (It can also simply mean that the problem was fixed, but the test was not re-run.)

Staffing

The actual and planned number of software personnel plotted against time. Another variant of this chart shows key personnel loading. This can be more insightful then total loading as a project is usually dependent upon having the right skills at the right time.

Software Size

The actual and originally estimated size, plotted against time. Growth in the size of the system over the original estimates is a good indicator of impending cost and schedule overruns. Size is commonly measured in lines of code, although other units are sometimes recommended. Among these are function points, modules, or number of objects.

Computer Resource Utilization

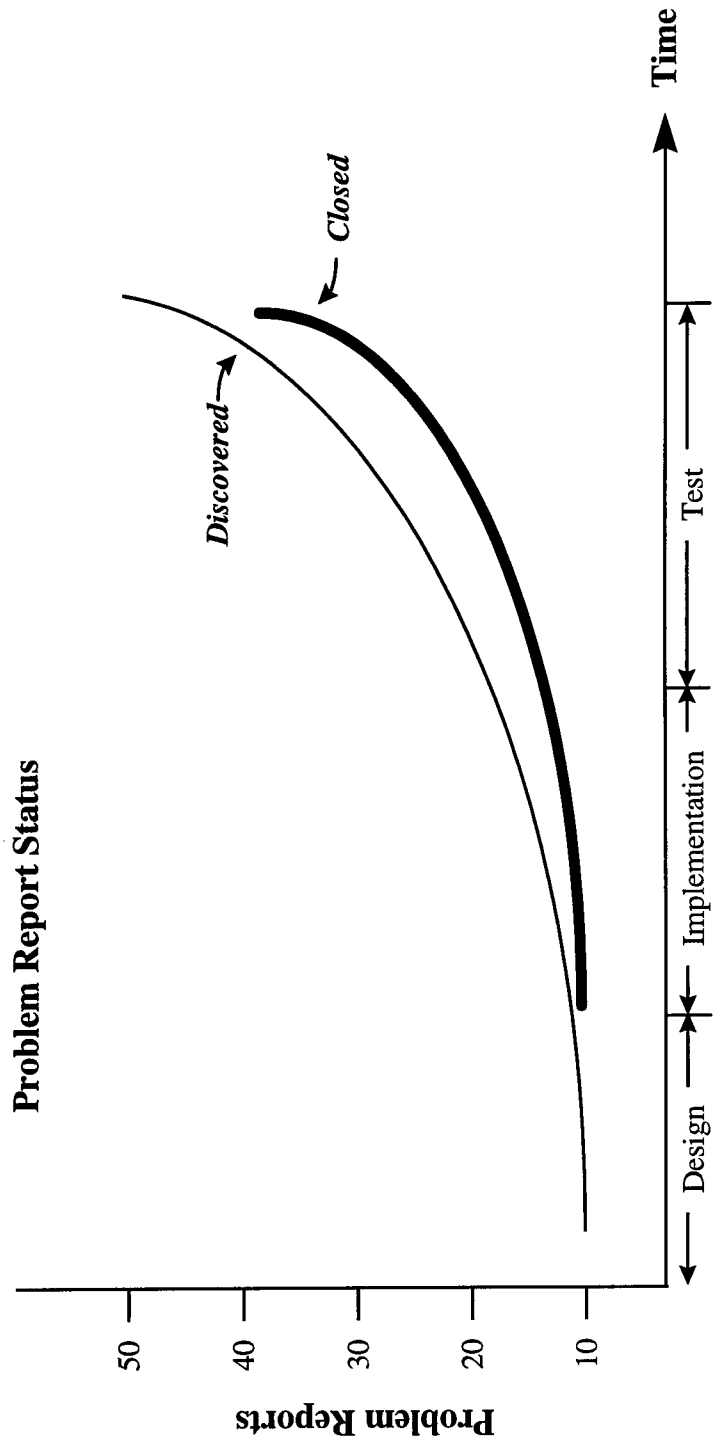
The CPU utilization, memory, and input/output resources are plotted as a percent of designated maximums for these resources. This gives an indication as to whether the software will fit into the hardware resources to which it has been allocated. It can also be used to control resource utilization when a limited amount is available or when a reserve of some percent is necessary (e.g., for contingencies or future expansion).

Problem Reports and Problem Report Closures

The cumulative number of problems discovered and the number corrected, plotted against time. It is useful to know whether the number of outstanding problems is increasing or decreasing over time.

Requirements Stability

The cumulative number of changes in requirements and the total number of requirements, plotted against time. This is a useful tool to manage requirements growth. If the number of new requirements exceeds 20 percent of the total requirements before the design is complete, it is considered a warning sign.



Note: Problems are captured by a problem report. These are closed, and the chart captures the discovery of reports and their closure. The curves should approach each other as one gets closer to delivery of the system; however, in reality, there will always be open problems.

Figure 17-1 Sample Management Indicator

One ITS software vendor complained about being third tier down in a subcontracting arrangement and not having access to the customer. Similarly, the customer complained about not having access to the subcontractor.

Let the contractor know, up front, that full visibility and participation of all developers is necessary and required for the management of the program. Subcontracting must be addressed early, as a part of the project plan, with appropriate language inserted into the RFP and contract. Points to consider for the RFP and contract are as follows:

- Have subcontractor personnel participate in all reviews, particularly in informal technical exchange meetings.
- Require that subcontractor data, such as the metrics discussed above, be made available without change.
- Request that the customer be afforded access to the subcontractor's facility.
- Ensure that subcontractor personnel participate in all testing as appropriate. The subcontractor should perform all testing of their subsystems and participate in systems level testing.
- Ensure that the customer is able to monitor subcontractor testing.
- Ensure that specific reports are provided for critical (contractor) subsystems and are not embedded in the contractor's reports and their meaning lost.

An attractive method for gaining visibility is to establish teams of developers and end users who are included on the team regardless of their affiliation. Built on the tenets of cooperation, teamwork, and breaking down institutional barriers, this technique has been successfully employed as *Integrated Product Teams* in the Department of Defense and other Federal agencies [Department of Defense, 1996], and as *Product Development Teams* in private industry [Prasad, 1996, volume 2 chapter 10]. It is, however, a difficult concept to introduce, particularly in a culture that does not understand it and has not been exposed to it.

In summary, our open communications theme applies, whether or not the software is developed by a subcontractor. Open communications implies ready access to the customer without having to go through a formal process that requires prime contractor approval for each interchange.

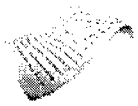
Corrective action tracking

Another management technique is the use of corrective action tracking. As problems arise in the project or in the software, they are logged as Problem Trouble Reports (PTRs). Problems can be rated according to their severity. For example,

- Highest — problem causes system to crash
- High — problem results in reduced functionality
- Low — system does not work as specified, but is workable

- Lowest — documentation error or trivial problem

Action items are assigned to correct the problems. The whole process is tracked to ensure that problems reach closure. Because the reports are under configuration control, their status and the number of problems outstanding are known at all times.



One large Department of Defense project was in danger of being canceled. It had slipped its schedule and there were still a large number of outstanding trouble reports. A project audit generated a historical plot of the number of trouble reports versus time. It showed a decreasing function, and was extrapolated to zero to indicate when project completion could be expected. This proved to be a reliable estimate, and a successful project was saved from unnecessary cancellation.

Problem trouble reports are generally the contractor's responsibility. Make sure that the contract calls this out and requires status reports on them, perhaps at project reviews.

How to handle schedule slips

Suppose you're successfully using the management techniques discussed above and they show a schedule slip. Consider the hypothetical example of a six month project in which the month two milestone is not met until the end of month three. Now what?

We consider five options. At the outset, the first option may appear to be the most attractive, but it is almost guaranteed *not* to work. The second probably won't work either. The last three options are seemingly less palatable, but offer a real possibility of working. [McConnell, 1996]

- option 1: Plan to catch up and make up for lost time later. However, experience has shown that projects hardly ever make up the lost time; they just get further and further behind. The fallacy in this option is that the schedule was too optimistic to begin with. In our hypothetical example, it took three months to accomplish what was scheduled to be done in two. Yet, under this option, the get-well plan is to exceed the schedule estimates for the remainder of the project. That is, to do four months of planned work in the remaining three months.

  Nonetheless, this is the option that is most often chosen. "When the schedule slippage is recognized, the natural (and traditional) response is to add manpower. Like dousing a fire with gasoline, this makes matters worse, much worse." [Brooks, 1975] "Oversimplifying outrageously," this is often stated as the famous "Brook's Law": "Adding manpower to a late software project makes it later." [Brooks, 1975]

Themes

Brook's Law is an illustration of how "software is different." Intuition gained from other endeavors for meeting a schedule "more workers, money, overtime, computer time--doesn't seem to work for software." [Putnam and Myers, 1992, page 43]

- option 2: Extend the project by the amount of the lost time. In our hypothetical example, all the remaining milestones would slip by one month, with project

completion at the end of the seventh month (instead of the sixth). The fallacy behind this option is that it assumes only the first part of the schedule was underestimated and that the remaining schedule estimates are accurate. In some cases, that may be true. For example, if the project started a month late, but was otherwise able to proceed along at the anticipated pace, then this option may work. But in most cases, it probably won't.

- option 3: Use the lost time as a multiplier for the remaining work. Although it may not be much to go on, the experienced delay is the best feedback available as to the true pace of the project. In our example, since two months stretched into three (a fifty percent stretch factor), the remaining four months of planned effort can be expected to take six months to complete; the overall project will take nine months and not six.
- option 4: Relax some of the requirements to make them easier to implement.
- option 5: Cut back on the planned functionality of the project. By removing requirements, all work associated with implementing them vanishes, and project complexity is reduced. "Don't expect to recover from a schedule slip of ten percent or more without a ten percent or greater reduction in software functionality to be delivered." [Condensed Guide To Software Acquisition Best Practices, 1997]

Themes

*Options 3, 4, and 5 are examples of our themes on the importance of **flexibility**. In Option 3, flexibility is used to trade off time so that functionality can be maintained. In Options 4 and 5, a firm schedule date may be maintained by trading off some of the functionality. But, without flexibility, these tradeoffs cannot be made.*

Requirements management

An important part of project management is the on-going requirements management process. We covered this important topic in chapter 9B.

Quality management

> *"Quality must be built into the products of software development from the beginning through the definition of an effective development environment and the controlled application of monitoring procedures."*
> *—[Evans and Marciniak, 1987]*

Managing the project and gaining visibility into contractor activities are really only a means to an end. The "end" in this case is a system. Not just any system, but a useful, operational system; in short, a *quality* product. In chapter 9A, we discussed some of the quality factors for software. Here we discuss quality management steps to ensure that the quality factors will be achieved.

Quality management refers to a broad program consisting of all efforts to ensure that a quality product is built within the performance, cost, and schedule envelope of the program. It requires that quality requirements be planned for and made a part of the

contractual effort. Everyone should be concerned with the quality of the effort, not just the quality function in the developer's organization.

Quality management is a broader term and practice then quality assurance. In fact, quality management encompasses quality assurance. Quality management as it has developed over the past ten or so years places the responsibility for a quality product across the entire project organization, not just the quality assurance organization. It is the quality assurance organization, or function, that acts as the eyes and ears of the project to assure that the quality program is being carried out, and that the goals of the quality program are achieved.

Quality management includes both the quality of the product and the quality of the process that is used to develop the product.

How do we achieve a quality product?

There are three complementary approach that the customer can use to achieve a quality product. Depending on the acquisition, these approaches can be used singly or in conjunction with one another.

First, and fundamental, quality requirements have to be specified in the system requirements document or the statement of work. (See the *Quality factors* section in *Developing Requirements,* chapter 9A.) Project personnel have to decide and prioritize what is important. For those factors that are measurable, such as Availability, the performance requirements should be specified. For those that are difficult to measure, such as Portability, project personnel can describe what is meant by the factor in this application, its priority, and how they will evaluate it in the developer's design.

The second approach is to ensure that a quality assurance program is in place within the developer's organization and determine how it will be applied to the program. An effective way to do this is to require, in the statement of work, that the developer institute a software quality program and provide a software quality assurance program plan for review by the customer. A further step toward achieving a quality product is to also require that the developer undergo (or have already undergone) a successful audit of its software quality program by an appropriate professional organization. The developer's audited achieved level should be at the level appropriate for the acquisition. During the development process, the customer monitors the quality program to ensure that it is being carried out and to judge its effectiveness.

A third approach is to require, in the statement of work, an Independent Verification and Validation (IV&V) review. Usually the IV&V effort is performed by an organization or contractor clearly independent of and having no conflict of interest with the developer. However, IV&V can also be conducted by staff in the development organization who are not designers that work on the project. They would typically report to a line of management that is independent of the one responsible for developing the project.

How do we ensure that the developer has a quality process that will result in a quality product?

A quality development process is key to attaining a quality product. This has been recognized in the Department of Defense with the innovation of the Software Capability Maturity Model. Using this model as the basis for a process improvement program is one of the ways that a software development organization can achieve a quality development process. (See *Software Capability Maturity Model (SW-CMM™),* topic sheet 4.)

An organization with a continuous process improvement program is better able to perform. The demonstration of such a program by a prospective developer should be viewed as a plus by the acquiring agency. However, continuous process improvement is an expensive investment and is usually an overhead expense. Contractors who invest in such improvements expect to recoup their costs in some fashion and this usually results in higher overhead rates. If the customer puts low caps on overhead rates, that may dissuade some of the better developers/integrators from participating. While a customer might reason that the contractor's resulting increased productivity should pay for the investment and this would be true in a fixed-price contract (with the problems that we have discussed), this is not possible under cost-reimbursement or time-and-materials contracts. In fact, under these contract arrangements, the contractor would be further penalized for being too productive because reduced labor hours would generate less overhead reimbursement. Another example of software acquisitions being different.

Expectations management

ITS project experience has shown the importance of managing the expectations of stakeholders not directly working on your project. This includes the public and upper management in your agency. The danger is that a successful project that meets all its goals and objectives can still be deemed a failure, if the project fails to meet overly optimistic expectations for it. To reduce the possibility of false expectations, do not over-promise: Do not propose an overly optimistic schedule. Do not promise more functionality than you can reasonably deliver.

However, be aware that unreasonable expectations need not arise from any overt actions on your part. There is a natural tendency for others to have a rosy vision that goes beyond what you are claiming for the project. Therefore, you will need to be continually on guard to see if this is taking place. If it is, take immediate steps to correct others of their misperceptions.

- *The customer has an active role to play even after a contract is issued.*
- *Project reviews, documentation reviews, and quantitative measurement data are three techniques for gaining visibility into the project and the software development process.*
- *Reflect the management techniques you choose in the contract.*
- *Make sure the contract allows for direct access to, and open communications with, any software contractors.*
- *If the project slips, do not try to "play catch up." Either stretch the remaining schedule or reduce functionality in the same proportion as the slip.*
- *Ask bidders to describe their development process, specifically with regard to software quality assurance and process improvement. Monitor the developer's quality assurance program during in-process reviews and through spot checks of their quality assurance program.*
- *Manage the expectations of stakeholders not directly working on the project.*

CHAPTER 18
SOFTWARE CONFIGURATION MANAGEMENT

"The discipline of Configuration Management is vital to the success of any software effort. Configuration Management is an integrated process of identifying, documenting, monitoring, evaluating, controlling, and approving all changes made during the life-cycle of the program for information that is shared by more than one individual or organization."
— [The Condensed Guide to Software Acquisition Best Practices, 1997]

Experts have identified software configuration management as one of the Principal Best Practices that are widely used and deemed essential to the success of any large-scale software project. [Arlie Software Council, cited in The Condensed Guide to Software Acquisition Best Practices, 1997] Software configuration management is primarily the responsibility of the development contractor.[1] Here we focus on the customer's role in this process.

What is configuration management?

Before describing software configuration management, we first introduce the concept of a *baseline*. A baseline is a "snapshot" of the system at a defined point in time. It provides a controlled basis for future work. Because the requirements are so important, they are recorded in a formal configuration-controlled document. (See *Requirements,* chapter 9, page 9-4.) All the other items associated with the software are also part of the baseline. This includes the software itself (source code and object code), technical documentation, test cases, problem reports and their status, and whatever else is used in conjunction with producing the software. A baseline is not something abstract; you should be able to physically point to the entire collection of all the items that constitute any given baseline, assembled together. (We recognize that this is not literally true, since some of the items may be in electronic format. But you should be able to readily put your fingers on and collect together all of the items that constitute a baseline.)

At certain points, a new baseline is established while the old one is retained. Any subsequent changes are made to the new baseline and the older baselines are no longer disturbed. That way, at any point in time, you can go back to the previous baseline(s) and re-establish things as they were.

Baselines represent "what you have and what you know" about the software system at a point in time. During the software requirements analysis phase you don't have any design or code, so the requirements baseline is, in effect, the system. Then as you move

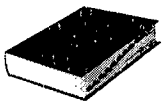
[1] Note, however, that if you decide to maintain the software with in-house staff, then you will also take on sole responsibility for configuration management during the maintenance phase. (See also *Training, Operations, and Software Maintenance,* chapter 16.)

into design, the requirements (now allocated to the design) and the design collectively become the system. When the code is developed, the requirements and design and code and whatever else is needed to support the software become the system.

Baselining carries with it the notion that all the constituent elements of a baseline are consistent with each other. For example, the set of baselined requirements is the one that matches the design, and the design is the one that matches the code. Indeed, one purpose of a design review is to ensure that the design implements all of the requirements. Any subsequent change to an element of the baseline must be reflected in all the other elements of the baseline so that they remain in "synch" with one another. This is the essence of configuration control.

An analogy to an automobile may be useful. A given model automobile consists of a well-defined set of parts. If the muffler fails on your 1993 sedan, you can easily ascertain what the replacement part number is and have a high level of confidence that the replacement part will fit together with the other parts of your automobile's configuration. Similarly, software can be considered to be constructed from a set of parts, albeit not necessarily tangible ones.

Configuration management is concerned with making changes to the baseline. For this reason, configuration management is sometimes called "change management."



*"**Configuration Management** is defined as the discipline of identifying the configuration of a system at discrete points in time for purposes of systematically controlling changes to this configuration and maintaining the integrity and traceability of this configuration throughout the system life cycle." [Bersoff, Henderson, and Siegel, 1980, page 20]*

Changes are controlled using a documented process and all changes to the baseline have to be approved. Changes are jointly managed by both the customer and the contractor. You don't want either party to make changes without the concurrence of the other, though the customer does have the ultimate authority in determining whether a change should be made.

What happens if you don't use configuration management?

Without proper configuration management procedures, the software development rapidly gets out of control. The various items and pieces of work get out of "synch" with each other. Typical symptoms of poor configuration management include [STSC, 1994]:

- The latest version of source code cannot be found.
- Bugs that were fixed in a previous software version reappear again.
- No one knows which modules comprise the software system delivered to the customer.
- Programmers are working on the wrong version of the code.
- The wrong version of the code was tested.
- There is no traceability between the requirements, documentation, and code.

Configuration management steps

The following is a simplified view of some of the steps that take place during software configuration management:

- Write and approve a configuration management plan.
- Identify the software components that will be placed under configuration control.
- Identify the components via a numbering or some other scheme.
- Maintain a current status of all parts (revision number, etc.) that are in a baseline. (configuration control).
- Maintain a backup copy of the baseline. At any point in time you should be able to go back and faithfully reproduce a previous baseline on the system. (configuration control)
- Get approvals before making changes to the baseline. Make the changes in accordance with your plan and document the changes. (change control)
- Check to see that the requirements, design, code, test cases, etc. all track one another, especially when it comes time to install the software (configuration audit)

Configuration management responsibilities

The developer has primary responsibility for implementing configuration management, establishing and controlling the baselines. However, the customer also has some responsibilities that must be considered:

- Require the developer to implement a configuration management process and describe that process in a configuration management plan.
- Review and approve the plan.
- Check to see that the plan is followed.
- Decide what baselines will be established and how they will be controlled. A Configuration Control Board may be established to oversee the configuration management process.
- Establish a process for controlling interfaces between separate contractors where their products interact, and between the vendor products and other systems that the products must interface with.

If the requirements are developed before the RFP is issued, the customer would also have responsibility for establishing the requirements baseline.

Checklist 18-1 can assist you in determining whether the contractor's configuration management process is adequate for your program [The Condensed Guide to Software Acquisition Best Practices, 1997; Paulk, 1993]:

Checklist 18-1. How To Determine If Configuration Management Is Adequate for Your Program *

✓	Is a configuration management plan documented for the project?
✓	Have the products that will be placed under configuration control been identified?
✓	Is the configuration management process integrated with the project plan and followed as an integral part of the culture?
✓	Are all versions (of configuration items) controlled?
✓	Has an electronic library been established that can store and retrieve multiple baselines?
✓	Are configuration control tools used for status accounting and configuration identification tracking?
✓	Are change requests and problem reports for all configuration recorded, approved, and tracked according to a documented procedure?
✓	Are all changes to baselines controlled in accordance with procedures?
✓	Are all baselines periodically reviewed and audited to assess the effectiveness of the configuration management process?
✓	Are all pieces of information shared by two or more organizations placed under configuration management?

* "The Condensed Guide to Software Acquisition Best Practices, 1997" and [Paulk, 1993].

Don't over-do it

Although lack of software configuration management can result in the problems listed above, too much configuration management is not good either. If you require the contractor to put configuration management procedures into place prematurely, they can stifle progress on the project. For example, early drafts of documents, when there is still much iteration and revision, should not be placed under configuration control. Only when documentation and other software products are mature enough to be baselined, should configuration management be employed. Similarly, if programmers are trying out little tests to see how something would pan out, or exploring the capabilities of a database package, do not require that such software be subject to configuration control procedures. Nonetheless, the requirements and all formal baselines that have been established must be controlled. Let the development contractor control the development configuration. Clearly, deciding when to use configuration management is a judgment call that requires good practical sense.

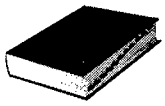
- *A formal software configuration management process is essential to the health of your program.*
- *Establish baselines and employ formal procedures for making changes to them. A baseline is a "snapshot" of everything associated with the software.*
- *Check to ensure that the developer establishes sound configuration management procedures and follows them.*

CHAPTER 19
SOFTWARE RISK MANAGEMENT

"The Airlie Software Council identified nine Principal Best Practices observed to be used generally and successfully in industry, and deemed essential for nearly all [Department of Defense] software development projects." Formal risk management is the first practice on the list. [The Condensed Guide to Software Acquisition Best Practices, 1997, page 8]

Project planning is, by its nature, optimistic. It assumes everything will go right. But as we have seen, problems often arise on software acquisitions. Sometimes without apparent warning. Use risk management techniques to avoid such surprises (or at least to minimize the number of them!). By its nature risk management is pessimistic. It is the process of continually learning from experience, assessing what things can go wrong, and implementing strategies to deal with them. [Higuera *et al.*, 1994]

Risk management is carried out from the inception of a project until its completion. The purpose is to identify the risks before they become problems, and handle them while there is still time. This avoids crisis management situations in which the options are restricted and schedule slippage is the only "solution". Ironically, if you do a good job of risk management, then it appears to be unnecessary.



*What is a **risk**? A risk is not a problem. A problem is something that has occurred, and must be dealt with. A risk is something undesirable that may or may not happen -- a potential problem, not a certainty. [Higuera, 1994] A risk can be assessed in terms of its impact on performance (not achieving some requirement), schedule, or cost.*

One way in which software projects are different is in the magnitude of the risks. On other types of engineering projects, if the risk results in a 50 percent multiplier (on cost, for example), it is considered a high impact risk. On software projects, risk impact can be several hundred percent.

Is risk management needed for my project?

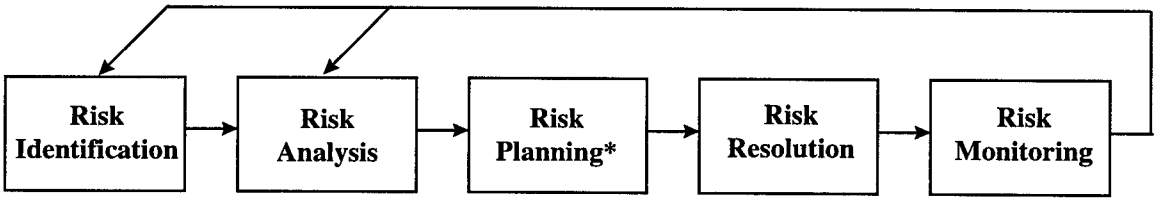
The short answer is "yes." However, in spite of the quote at the top of the page, we recognize that risk management has to be geared to the size of the project and the development risk inherent in the system. (See the ordered list in figure 4-1 of *Types of ITS Systems,* chapter 4.) The amount of new development, customization, and type of system will all contribute to the amount of risk. To avoid overkill, feel free to modify the full, formal treatment described here. For example, a "top ten list", in which the top ten risks are kept and watched on a periodic basis, may be sufficient for small systems or those with low risk.

The most effective risk management technique may simply be to go with an existing product. Much of the software development risk will automatically be removed, and the risk will reflect the amount of customization that you undertake. On the other hand, there is also a risk that an off-the-shelf system that works elsewhere may not work for you. As we've indicated, off-the-shelf is not a panacea. Software that works in one environment with one set of computing hardware and field devices may not work in another with a different set.

To summarize, every software acquisition has risk, and risk must be managed on every software acquisition.

The risk management steps

While various references differ somewhat in the details and in the terminology, the following steps represent the essence of risk management.



* Also called "Risk Mitigation" or "Risk Resolution"

Figure 19-1. Risk Management Steps

The first step in risk management is *risk identification.* This involves anticipating what things might go wrong and listing them. If desired, the risks can be assigned to various categories, such as technical risks or schedule risks. The taxonomy of software risks shown in table 19-1 can serve as a starting point for identifying the risks on your project. (See *Where To Get More Help,* chapter 21 for references to alternative lists of software risks.)

Next, *risk analysis* characterizes the identified risks by their likelihood of occurrence and severity of impact. This is difficult to do quantitatively, so the likelihood and impact of a risk are normally rated qualitatively as high, medium, or low. (That's why we use "likelihood of occurrence" instead of "probability of occurrence", which is sometimes found in the literature. The latter term implies a quantitative assessment.) From these ratings, a risk characterization table can be created. As shown in table 19-2, a risk characterization table provides the information needed for risk prioritization; that is, deciding which risks to address. This establishes a risk strategy for the project. Generally, the high impact risks with high likelihood of occurrence will be addressed first and receive the most attention. Generally, some resources are required to mitigate such risks. Or the requirements, schedule, or cost may need to be adjusted. A risk scoring low in both categories would generally receive low priority. (Unless, perhaps, the risk mitigation strategy for it were immediately obvious and so simple that you decide you may as well go ahead and eliminate the risk.)

Table 19-1. Taxonomy of Software Risk: An Overview

A. Product Engineering	B. Development Environment	C. Program Constraints
1. Requirements a. Stability b. Completeness c. Clarity d. Validity e. Feasibility f. Precedent g. Scale 2. Design a. Functionality b. Difficulty c. Interfaces d. Performance e. Testability f. Hardware Constraints g. Non-Developmental Software 3. Code and Unit Test a. Feasibility b. Testing c. Coding/ Implementation 4. Integration and Test a. Environment b. Product Integration c. System Integration 5. Engineering Specialties a. Maintainability b. Reliability c. Safety d. Security e. Human Factors f. Specifications	1. Development Process a. Formality b. Suitability c. Process Control d. Familiarity e. Product Control 2. Development System a. Capacity b. Suitability c. Usability d. Familiarity e. Reliability f. System Support g. Deliverability 3. Management Process a. Planning b. Project Organization c. Management Experience d. Program Interfaces 4. Management Methods a. Monitoring b. Personnel Management c. Quality assurance d. Configuration Management 5. Work Environment a. Quality Attitude b. Cooperation c. Communication d. Morale	1. Resources a. Schedule b. Staff c. Budget d. Facilities 2. Contract a. Type of Contract b. Restrictions c. Dependencies 3. Program Interfaces a. Customer b. Associate Contractors c. Subcontractors d. Prime Contractor e. Corporate Management f. Vendors g. Politics

NOTE: This table is reproduced from [Sisti, 1997], which also provides (a) descriptions of all the potential risks listed above, and (b) questions to ask that can assist you in determining whether the risk is a real one for your project.

Table 19-2. Risk Characterization Table

		Likelihood of Occurrence		
		High	**Medium**	**Low**
Impact	**High**	• Risk #1 • Risk #5	• Risk #7	---
	Medium	• Risk #3	• Risk #2 • Risk #4	• Risk #6
	Low	• Risk #8	---	---

The next step is determining what should be done for each of the risks listed in the table. (This step is variously called "risk planning", "risk mitigation", or "risk resolution" in the literature.) The strategies and specific actions you decide upon generally fall into several categories:

- Avoid the risk. One way to do this is by relaxing requirements that cause the risk in the first place. For example, the risk of not meeting a system availability requirement of 99% could be avoided by decreasing the availability requirement to 95%.

Themes

*Here is where **flexibility** comes into play. You may decide that meeting the relaxed requirements is "good enough." Or the relaxed requirement may be the lesser of evils, if the alternative is having to devote significant additional resources to meet the requirement. This example also illustrates the value of **team building**. Probably only the software contractor would be in a position to understand the full ramifications of what it would take to meet the requirement, hence the need to have **open communications** with the contractor who should be involved in the risk management activities.*

- Eliminate the root cause of the risk. An example would be to provide additional computing resources if a lack of resources was imposing schedule risk.
- Control the risk. This generally takes the form of trading off cost, schedule, and performance to reduce the probability of the risk occurring. An example might be the attainment of a difficult performance objective, perhaps implementing an incident detection algorithm. You could decide to spend some up-front money to research its feasibility before contracting for the full development of the system. This is an example of starting risk management early in the project while there is still time to be effective. In some cases, you may decide to control the risk by exploring other options or funding an alternate path in parallel with the main system. The alternate could be used as a fallback in case the risk materializes. This may be the preferred approach if a scheduled date must be met, no matter what.

- Assume the risk. This is generally done for the lower priority risks. (However, see the risk monitoring step below, about watching the risk to make sure that its likelihood doesn't rise over time, thereby increasing its priority.)

The next step is *risk resolution*, which is simply the execution of the strategy planned in the previous step.

The final step in risk management is the *monitoring* or *tracking* step. Monitor the controlled risks for progress towards resolution. Periodically review all the risks listed in the risk characterization table. This review can take place as part of the normal review process that was described previously. Reassess the other risks to determine if there is a change in likelihood or impact. Did a deferred risk go away, or should its priority be raised? Are there new risks that should be added? (If so, go back to the risk identification step.)

Risk management is carried out throughout the project

Risk management should be an on-going process throughout the project. During the initial phases, risk identification is a useful tool for influencing the acquisition strategy. For example, if a major performance risk is identified, it could be addressed before major procurement action is initiated.

During source selection, consider the bidders' proposed risk management approach. The RFP can ask the bidders to describe their risk management approach. It is understandable that bidders might be reluctant to identify actual risks in their proposal. However, they should be able to describe their approach for risk management during the conduct of the program. (You can also use the RFP as an opportunity to inform bidders about your intended risk management strategy.)

After contract award, risk management can be used to monitor and control the software development risks. This is normally accomplished by participating in periodic program reviews, where risk is one of the agenda topics.

One of the most effective risk management methods is the establishment of a software quality management program early in the project. (See the *Quality management* section in *Project Management,* chapter 17.) This program's charter can include looking at issues that impact product performance, cost, and schedule. Some of the things they can evaluate are the adequacy of the development environment (tool sets, compilers, computing hardware, etc.), appropriateness of the selected computer language, the clarity of requirements, and the adequacy of the prototyping, testing, and configuration management activities.

Treat risk management as a teaming activity

It is best if risk management is jointly carried out by the customer and contractor. Here's one way of opening up communications to make this happen: Have each party independently generate its own list of risks. Then sit down together and compare the lists. You may be surprised at your differing perspectives on the project. Then work

together to prepare one set of prioritized risks. Jointly plan and carry out the risk management steps discussed above. If you go with such an approach, be sure to include the appropriate language in the RFP.

To summarize, risks should be addressed at the start of the project in order to influence the acquisition approach. A risk management program should be put into place to monitor and control risks throughout the development project. Risk management relies upon open communications and collaboration. Risk management discussions can serve as a vehicle used towards achieving the open communications needed for the rest of the parts of the project.

We previously gave an example of relaxing an availability requirement to avoid a risk. For that scenario to work, the software contractor would need to come forward, identify the requirement as risky, and note that meeting the requirement would require significant development resources. In other words, the example presupposes a teaming activity. It also presupposes that the customer and contractor have a shared vision and are working together to achieve a common goal. [Higuera et al., 1994]

Making risk management work

- The key to successful risk management is non-threatening communications. [Higuera and Haimes, 1996]. Ask yourself, "Do information flow patterns and reward criteria within the organization support the identification of risk by all project personnel?" [The Condensed Guide to Software Acquisition Best Practices, 1997 page 8]
- Describe your risk management approach in the program plan and in the RFP.
- Ask the bidders to describe their risk management approaches.

- *Continually carry out risk management activities throughout the life of a software project.*
- *Risk management steps comprise risk identification, analysis, planning, resolution, and monitoring.*
- *Risk management is most effective when done as a teaming activity between the customer and contractor.*
- *For risk management to work, there must be an atmosphere that fosters people to come forward with risks without "finger pointing."*

PART FIVE:

PUTTING IT ALL TOGETHER

CHAPTER 20
BEST PRACTICES CHECKLIST AND KEY POINTS SUMMARY

Checklist 20-1 lists best practices that summarize the key activities you carry out as part of software acquisition. As you pull together your program, it can be used in determining your readiness to proceed with the acquisition.

Following the summary checklist, we have reproduced in one place for your convenience, all key point summaries and checklists that appear throughout the document.

Checklist 20-1. Best Practice Activities for Software Acquisition *

✓	Use existing products to the maximum extent practicable.
✓	Document the software acquisition plan that addresses the entire project through operations and maintenance.
✓	Build a team and collaborate with them to acquire the system.
✓	Maintain on-going, open communications with the contractor and other members of your team.
✓	Prepare independent cost and schedule estimates.
✓	Document requirements and have them serve as the basis of other activities (test cases, budget and schedule, design, etc.).
✓	Trade off requirements to decrease cost and schedule. Keep all three in synch.
✓	Develop formal source selection criteria, which, for a software development process, include assessment of the bidders' software engineering process.
✓	Identify problems, record them, and track their status.
✓	Track expenditures and progress.
✓	Manage risks: identify and resolve them. Conduct risk management in conjunction with your contractor and other team members as an integral part of the acquisition process.
✓	Include system acceptance criteria in the contract.
✓	Develop an acceptance test plan and carry out acceptance testing in accordance with it. (Note: This plan could be developed by the contractor subject to your review and approval.)
✓	Have explicit contract language documenting licensing and ownership rights.
✓	Develop training materials and carry out a training program for use and operation of the system.
✓	Develop a support strategy for the system.
✓	Ensure software meets Federal requirements for architecture and standards consistency.

* This checklist has been adapted to ITS systems from the Software Acquisition Capability Maturity Model (SA-CMM) [Ferguson, 1996] and *The Program Managers Guide to Software Acquisition Best Practices*, 1997.

Key Points To Remember From Volume I

The Nature of Software:

- *Software acquisitions are different from other types of projects.*
- *Missed schedules, cost overruns, and lack of visibility into the software and software development are common.*
- *Different approaches are therefore needed to manage software projects.*
- *There are established managerial techniques that can be relied upon to overcome the problems.*
- *ITS software experiences are similar to those encountered on other types of software projects.*
- *This document is intended to help you find your way through what has been termed the "software thicket."*

Software Acquisition In A Larger Context:

- *Although we will focus our attention on the software acquisition activities, they take place in the context of a system acquisition, which in turn is part of an overall process.*

Differing Perceptions of ITS Software:

- *The public and private sectors have very different perceptions of software. These differences manifest themselves in the ways that they approach software acquisitions and each other.*
- *Each sector perceives the situation as lose-win: they lose while the other sector wins. In fact, it's lose-lose; both sectors lose.*

Themes of Successful Software Acquisition:

- *Build your software acquisition around certain themes that should recur throughout the various acquisition activities:*
  - *People themes, which are akin to partnering.*
  - *Management themes, on how to approach the acquisition.*
  - *System themes, relating to the end product.*
- *The themes can guide you as to the best practices to employ in approaching your software acquisition.*
- *Collectively, the themes address the problems commonly associated with software and represent our response to the overarching theme that "software is different."*

Key Points To Remember From Volume II

Sequence of Acquisition Activities:

- *There is no simple step-by-step activity sequence that applies to all software acquisitions.*
- *Developing system requirements and selecting a contracting vehicle drive many of the other activities.*
- *Many acquisition activities take place in parallel. They feed off and build upon one another.*

Building A Team:

- *Build a team of professionals that contains the variety of skills needed to make the project succeed.*
- *If possible, consider tapping other resources in your agency to gain access to the needed skills. When this is not possible, you may have to contract for these skills to gain access to them.*
- *Be sure to add the software contractor to your team as soon as the contractor comes on-board.*

Planning The Project:

- *Write a short project plan that documents your major approaches to the acquisition.*
- *The project plan is a living document during the acquisition process; new information will be added, and existing information may need to be revised.*

Requirements:

- *Develop a good set of requirements. It is one of the most important things that you can do on a software acquisition.*
- *Have the various members of the customer's team participate in developing the requirements.*
- *Document the requirements in a formal configuration-controlled document.*
- *Develop functional and performance requirements (the "what's") and not the design or technical requirements (the "how's").*
- *Scrub the requirements to avoid asking for too much. Avoid requirements or scope creep.*
- *Address quality factors and the ability of the system to accommodate anticipated changes.*
- *As soon as possible after contract award, hold a requirements walk-through with the contractor and other members of your team. Then sign the requirements and place them under configuration control. Make sure the contract calls for these activities.*
- *Establish a stable base of requirements. It is essential for the success of your project.*
- *Address requirements issues as they arise, as part of the on-going requirements management process.*
- *Flesh out the human interface requirements using rapid prototyping.*
- *Use the requirements as the basis for size, schedule, and cost estimates; build versus buy decisions; design and development activities; and acceptance testing.*

Build/Buy Decision(s):

- *Consider buying your software, if at all possible, rather than building it.*
- *Consider a mix of build and buy, if buying alone does not meet project needs.*
- *The buy option is not without risk; however, the risks are manageable.*
- *Understand the off-the-shelf products and the implications of their use before buying them.*

Selecting The Contracting Vehicle:

- *The familiar engineer/contractor (design-bid-build) used for construction projects is not appropriate for software; it should not be used for software acquisitions.*
- *Work with your contracting or purchasing office and legal office early in the project to explore your full range of options; challenge traditional thinking.*
- *Fixed-price contracting may not provide the needed flexibility for building software. Consider a time-and-materials type of contract and/or innovative contracting approaches as alternatives to fixed-price.*
- *Do not use fixed-price contracting practices under the guise of a cost-reimbursement contract.*
- *Do not use a fixed-price contract for computer hardware in conjunction with a cost-reimbursement contract for software.*
- *Fixed-price contracting may be appropriate for off-the-shelf software.*
- *Whatever approach is chosen, that approach will still require the application of sound acquisition practices; a contract is not a substitute for them.*

Identifying The Software Environment:

- *Identify the environment -- including the interfaces to legacy systems -- in which the software will be operating.*
- *Do not unnecessarily constrain the system design by prematurely specifying the computing hardware or operating system.*

Resolving The Intellectual Property Rights:

- *Regardless of whether or not they have precise legal meanings, terms such as "ownership" or "licensing" have different connotations to different people.*
- *Before a contract is signed, reach agreements on intellectual property rights for the software.*
- *Walk through the contracting language together; discuss the implications, resolve issues, and ensure the contracting language clearly and explicitly states your understandings.*
- *Be explicit with respect to source code and object code and the media on which it will be delivered.*
- *You may wish to acquire the services of a lawyer who specializes in software intellectual property rights.*
- *Think through your true needs before insisting on certain rights at the negotiating table.*

Project Scheduling:

- *Two flawed practices are common with software schedules:*
  - *They are established independently of requirements*
  - *They are set in the impossible-to-do zone*

- *Develop a schedule that realistically matches the requirements and what you've set out to accomplish. Don't use best case estimates; they won't be met. Pessimistic estimates often turn out to be the most realistic ones.*
- *Adjust the schedule, requirements, and budget so that they are consistent with one another throughout the project.*
- *Stretching out a realistic schedule is one of the most cost-effective ways of lowering the cost and overall development effort of a project.*
- *Use well-defined "yes/no," "done/not done" milestones.*
- *Get as many independent size estimates for the system as possible, including those of the contractor and the software expert on your team. Resolve the differences.*
- *Use feedback on actual progress to derive more realistic schedule estimates for future activities.*

Acceptance Testing:

- *Plan a formal acceptance test strategy, including the use of formal documentation, before an RFP is issued. Reflect this approach in the contract.*
- *Schedule test planning and preparation activities to begin as early as possible after contract award, and to proceed throughout the development period in parallel with the system development activities.*
- *Base acceptance testing on the requirements.*
- *Carry out several varieties of rigorous testing; simple, benign tests are not sufficient.*
- *Carry out testing as a teaming activity.*

Training, Operations, and Software Maintenance:

- *Plan for support activities early, and reflect your approach in the contract. Prepare for support in parallel with development activities.*
- *Allow adequate budget for support activities to take place. Over the life of the system, support activities generally consume more budget resources than do the development activities.*

Project Management:

- *The customer has an active role to play even after a contract is issued.*
- *Project reviews, documentation reviews, and quantitative measurement data are three techniques for gaining visibility into the project and the software development process.*
- *Reflect the management techniques you choose in the contract.*
- *Make sure the contract allows for direct access to, and open communications with, any software contractors.*
- *If the project slips, do not try to "play catch up." Either stretch the remaining schedule or reduce functionality in the same proportion as the slip.*
- *Ask bidders to describe their development process, specifically with regard to software quality assurance and process improvement. Monitor the developer's quality assurance program during in-process reviews and through spot checks of their quality assurance program.*
- *Manage the expectations of stakeholders not directly working on the project.*

Software Configuration Management:

- *A formal software configuration management process is essential to the health of your program.*
- *Establish baselines and employ formal procedures for making changes to them. A baseline is a "snapshot" of everything associated with the software.*
- *Check to ensure that the developer establishes sound configuration management procedures and follows them.*

Software Risk Management:

- *Continually carry out risk management activities throughout the life of a software project.*
- *Risk management steps comprise risk identification, analysis, planning, resolution, and monitoring.*
- *Risk management is most effective when done as a teaming activity between the customer and contractor.*
- *For risk management to work, there must be an atmosphere that fosters people to come forward with risks without "finger pointing."*

Checklist 8-1. What To Include In The Project Plan

✓	Project Description: A brief narrative of what the project is all about, its goals, objectives and scope.*
✓	Justification: Why the acquisition will take place; saving money, alleviating congestion, providing better on-time service are all possibilities; although increased productivity is sometimes offered as justification, in practice, this seldom materializes.
✓	Project Schedule: An overall schedule showing when the major milestones take place.*
✓	Roles: Who will manage the project? What is the size and composition of your acquisition team? What organizations will be involved? Who are the contact points within each of these organizations? What are their respective roles? Who will be responsible for training? For maintenance? Can include an organization chart.
✓	Funding Estimates and Sources: You can refer back to these to ensure that you are living within your constraints.
✓	*Facilities:* Where will the work be carried out? Where will the system and its users be housed upon completion? Will any special tools or equipment be needed?*
✓	*Acquisition Strategy:* Will the system be built from scratch? To what extent will off-the-shelf components be used or sought?* How will such components be integrated into the system? Are there pieces of the system that can be reused from other projects?* Will the system be build incrementally using a multi-phase approach, in which each phase contains a task to define and scope the next phase? Will a prototype be built? How will COTS products be integrated with each other?
✓	*Environment:* Are there any legacy systems that must be interfaced with? Field sensors, roadway or transit vehicle devices? How about other organizations or neighboring jurisdictions?
✓	Standards: What technical standards must be complied with?
✓	*Major Risks and Risk Management Approach:* How will risks be managed? Have any key risks been identified?*
✓	Contracting Strategy: What work will be done in-house? What will be contracted for? Will consultants be hired?

[Checklist continued on next page]

Checklist 8-1. What To Include In The Project Plan (Concluded)

✓	Type of Contract(s):* What options are being considered? Fixed price, cost-plus, or time and materials? Design/build or system manager? How many contracts will be needed?
✓	*Contract Management:* How will oversight be accomplished? How will progress be tracked and monitored?*
✓	*The End Users:* Who will operate the system? Administer it? Maintain it? What will be the sources of staffing and funding for these activities?
✓	*Acceptance Strategy:** What will be the basis for accepting the system?
✓	*Training Concept:* How will user training be accomplished?
✓	*Maintenance Concept:* How will the system be maintained once it is accepted?
✓	Constraints:* What are the realities that you must live within?

* Recommended in the "Software Acquisition Capability Maturity Model (SA-CMM)," [Ferguson, 1996, pages L2-4 and L2-5].

NOTE: *Italicized* items are unique to software, or are more critical for software then they may be for some other types of projects.

Checklist 9-1. What to Include in a Requirements Document

	Functional requirements
✓	What capabilities the system must have. The trick is to stay at a functional level and not prescribe a solution.
✓	Each required function takes the form of a sentence with the word "shall" and should be testable. (e.g., "The system shall display a congestion warning message on variable message signs.")
✓	Define whether the function is manual, automated, or semi-automated (e.g., the system shall choose a message and display it; the operator shall type in a message which gets displayed; the operator chooses from among several pre-defined messages and causes the system to display it).
✓	High-level human interface requirements. More detailed human interface requirements may also be included, but rapid prototyping is the preferred approach for them.
✓	Algorithms or equations.
✓	Year 2000 compliant (see topic sheet 1 on The Year 2000 Problem (Y2K)).
✓	Conforms to the National ITS Architecture.
	Performance requirements
✓	Response times, expressed as averages, standard deviations, 90 percentile, etc. ("The systems shall have a mean response time of 30 seconds with 90 percent of all responses within 45 seconds.")
✓	Loading requirements (e.g., being able to handle simultaneous inputs from a specified number of sensors), including degradation requirements, if any, under excessive load.
✓	Throughput (e.g., number of transactions per hour).
✓	Capacity ("The system shall be able to store 30 days of incident reports.")
✓	False alarm rates, including the algorithms to be used in determining the rates.
✓	Accuracy, specifying the algorithms.
✓	Reliability and maintainability (e.g., mean time between failures; mean time to repair).
✓	Security (see topic sheet 2 on Security).
✓	Safety (see topic sheet 5 on Software Safety).

[Checklist continued on next page]

Checklist 9-1. What to Include in a Requirements Document (Concluded)

	Interfaces, external and internal, including the data (inputs and outputs) and controls that flow across the interface*
✓	To/from field devices.
✓	To/from displays.
✓	To/from users.
✓	To/from other systems, including legacy systems.
✓	To/from other jurisdictions.
✓	Between major subsystem components (e.g., between a vehicle subsystem and a transit management center).
✓	Between software components (e.g., between incident detection algorithms and data collections.
	Inputs
✓	Identify its source (automated and human).
✓	Frequency of arrival.
✓	Valid ranges and units of measures.
✓	Give each one a unique name and identifier.
	Outputs
✓	Include real-time outputs (e.g., alerts to a display) and non-real-time (e.g., summary reports printed out on paper).
✓	Identify its destination (devices or users).
✓	Frequency of generation.
✓	Valid ranges and units of measure.
✓	Give each one a unique name and identifier.

*Data flows in the National ITS Architecture are a good source of candidate interface requirements, including the data flows and their descriptions.

Checklist 9-2. Suggested Agenda Items For A Requirements Walk-Through

✓	Clarify ambiguous or vague requirements.
✓	Remove inconsistencies between requirements.
✓	Supply missing requirements.
✓	Replace existing requirements with better alternatives that are identified.
✓	Eliminate unnecessary or hard-to-meet requirements, or mark them as low priority.
✓	Prioritize the remaining requirements.
	Scrub session
✓	Eliminate low priority or high cost requirements; "retain only those that are absolutely necessary."
✓	"Simplify all requirements that are more complicated than absolutely necessary."
✓	"Substitute cheaper options" when they are available.
✓	Defer lower priority requirements into later versions of the software.

*Quotations taken from S. McConnell, *Rapid Development: Taming Wild Software Schedules,* Microsoft Press, 1996, page 329.

Checklist 12-1. What to Consider When Identifying the Software Environment

✓	Interfaces to legacy and other existing systems or applications (what legacy software and systems must the new software interoperate or interface with, including application monitoring and management systems?). *
✓	Existing communications and networks (including protocols, characteristics such as line speed, bandwidth, dedicated or dial-up, type of network, significant components).
✓	Interfaces to planned future applications or systems (what other software is planned for the future and will need to interoperate or interface with this software?). *
✓	User population and user interface (e.g., graphical user interface (GUI), point-and-click).
✓	Location/physical environment (e.g., office, computer room, outside; lighting conditions, space constraints impacting use of mouse or keyboards; uninterruptable power supply).
✓	Security measures that implement security policies and procedures, to include physical safeguards (locked rooms), software protection (passwords), and hardware systems (so-called firewalls to the Internet).
✓	Performance (how much data to be processed and how fast it must be processed, data accuracy).
✓	Standards.
✓	Hardware (e.g., PC, mini, or mainframe).**
✓	Operating system.**
✓	Data base management system (DBMS).**
✓	Programming languages, development methodologies, maintenance requirements. **

*** The National ITS Architecture can assist in identifying interfaces to other systems.**
****Only if absolutely necessary.**

Checklist 13-1. Intellectual Property Rights

	General Rights
✓	Who owns the software? What rights does that entail?
✓	Who holds the copyright to the software? What rights does that entail?
✓	Should copyright notices be included as source code comments? Who should be listed as retaining the copyright?
	Customer Rights
✓	Can the customer make additional copies of the operational software for their internal use on this project? On other projects?
✓	Can the customer distribute copies of the operational software to other agencies or departments within their state?
✓	Can the customer distribute copies of the operational software or issue licenses to other states? Can such a license allow the other state to make changes or enhance the software, or does it only give them the right to use it?
✓	Can the customer give away copies of the operational software for free? Charge a fee?
✓	Can the customer change the operational software or make derivative works?
✓	Can the customer disclose the source code of the operational software to other vendors or allow them to make changes to the software?
✓	For the previous six items on the checklist, which portions of the operational software can be copies or distributed: the source code? the object code? the documentation? how many copies may be made?
✓	Does the customer have rights to any subsequent upgrades made by the contractor?
✓	Are there portions of the software that are needed to run the system that are not covered by the licensing agreements? Such items could include the operating system (e.g., Windows, UNIX), a commercial database management system, a geographic information system, or a digital map. For these items, there may be a different number of copies that can be made, run, distributed, etc. than there is for the rest of the software.
✓	Will the customer have access to the source code? If so, as compilable files or only as listings?

[Checklist continued on next page]

Checklist 13-1. Intellectual Property Rights (Concluded)

✓	Will the customer have access to the support tools and development environment that were used to compile the software, keep it under configuration control, test it, etc.?
✓	Will the customer have rights to all the training material?
✓	Will the customer have rights to the executable environment needed to run the software or must these be purchased separately from other vendors?
✓	Will the customer have access to documentation on database formats and interface protocols?
✓	How many computers can contain copies of the software? How many can run the software? (Note: These numbers may be different to allow for backup copies.)
✓	How many computers can simultaneously run or access the software? (Note: On a network, all computers may be able to run a piece of software or access a database, but the licensing agreement may restrict the simultaneous number.)
✓	How many users have license to run the software? How many simultaneously?
✓	Can the software be run across a network? (Note: There are two options here. the software could be run remotely, that is, on the "other" machine where the copy resides; or, a temporary copy could be made on your machine and run locally.)
	Contractor Rights
✓	Can the contractor distribute the software to other customers? If so, can they charge for it?
✓	Can the contractor reuse portions of the software on other contracts?
✓	Can the contractor copyright or patent the software or patent other parts of the system? If so, will the customer have to pay royalty rights?
✓	Does the contractor have rights to any upgrades made by the customer?

Checklist 14-1. Software-Related Activities and Milestones on the Project Schedule

	Contract negotiations
✓	Walk-through of the intellectual property rights issues.
✓	Signing the contract (milestone).
✓	Dates on which the agency furnishes contractually-required items to the contractor (equipment, space, services, etc.) (milestones)
	Requirements
✓	Requirements walk-through.
✓	Signing the requirements (milestone).
✓	Rapid prototyping.
	Size estimates
✓	Independent size and schedule estimates by contractor.
✓	Resolving differences.
	Management controls
✓	Risk management reviews
✓	Project reviews.
✓	Inspections.
✓	Document reviews.
✓	Document approvals (milestones).
	Acceptance testing
✓	Detailed acceptance test planning.
✓	Conducting acceptance tests.
✓	Analysis of acceptance test results.
✓	System acceptance (milestone).
	Training
✓	Training preparation and planning.
✓	Conducting the training.
	Support
✓	Support facility development.
✓	Transition from development to operations and maintenance (milestone).

Checklist 15-1. What to Include in the Acceptance Test Plan

	Organizations and their respective roles; who will be responsible for:
✓	Conducting the tests?
✓	Recording the data?
✓	Analyzing the data and reporting the results?
	Where will the acceptance tests take place
✓	The contractor's location?
✓	An operational facility?
✓	In transit vehicles?
	Testing schedule (should allow time for data to be analyzed; may want to include a dry run phase)
✓	Computers to run the tests.
✓	Field devices (e.g., installed variable message signs; bus sensors).
✓	Other systems (e.g., legacy systems; systems in neighboring jurisdictions).
	Software needed
✓	Special test software (simulators to stress the system, spreadsheets to analyze results, etc.).
	Overall system acceptance criteria (Note: the pass/fail criteria for an individual test are listed below under "List of tests to be run.")
✓	Acceptable failure rate (e.g., pass all the tests designated as critical and 80 percent of the remaining tests).
	What happens when tests fail or do not proceed as planned
✓	Role of regression testing
	List of tests to be run; for each test, indicate:
✓	Test identifier (e.g., Test 1A)
✓	Purpose of test (brief statement)
✓	Data to be recorded
✓	Pass/fail criterion (depending upon the test, the test case may be a better place to provide this information).
	Traceability
✓	For each test, show to which requirement(s) the test traces (shows that test is requirements-based; can be viewed as protection for the contractor so that the new requirements are not "slipped in").
✓	For each requirement, show to which test(s) the requirement traces (shows the coverage of testing; can be viewed as protection for the customer so that all of the requirements are tested)

Checklist 15-2. What to Include in the Acceptance Test Procedures

	For each test, include the following:
✓	Pre-test activities needed to set up the test (e.g., turn on or off certain pieces of equipment, load a piece of software).
✓	Step-by-step procedures used to carry out the test.
✓	Procedures used to reduce and analyze the data (explicitly state equations, statistical formula, averaging techniques, etc.)
✓	Computers needed (to run tests; to analyze results)
✓	Field devices needed to run tests (e.g., installed variable message signs).
✓	Other systems (e.g., legacy systems; systems in neighboring jurisdictions).

Checklist 15-3. What to Include in the Acceptance Test Cases

✓	Input values
✓	Source of input (manual entry, field device, simulated data, etc.)
✓	How long the test is to be run (e.g., collect loop detector data for one hour)
✓	Expected value(s) of the output
✓	Pass/fail criterion for the test (depending upon the test, the test plan may be a better place to provide this information)
✓	Traceability between test cases and tests to ensure that a test case exists for all tests.

Checklist 15-4. What to Include in the Acceptance Test Log

✓	Name of test
✓	Date and time test started
✓	Date and time test ended (only for tests that last for extended duration; for many tests only the start time is needed)
✓	Who carried it out
✓	Any deviations from the test procedures (e.g., test conductor inadvertently left out a step; error was found in test procedure, so it was modified "on the fly")
✓	Recorded outputs

Checklist 15-5. What to Include in the Report of the Test Results

✓	Overall information (e.g., when the tests were conducted)
✓	Overall report on whether the system passed and what follow-on steps are needed
	Test-by-test results
✓	Test identifier, also indicate which test procedure was used and which test case was used
✓	Any deviations from the test procedure
✓	Recorded data from the test log
✓	Computed data
✓	Whether the test passed or failed (in accordance with the documented criteria)

Checklist 16-1. Personnel Roles Needed For System Support

✓	Shift supervision of the end-user operators
✓	Administration of the system to keep it running
✓	Generation of management reports from the system
✓	Review of data produced by the management reports
✓	Installation of new computing hardware or displays
✓	Integration of additional field devices into the system
✓	Fixing software bugs
✓	Upgrading the software – Installing new releases of off-the-shelf software – Adding functionality to custom software – Modifying control algorithms
✓	Diagnosis of problems and repairing hardware
✓	Training for all the above, both when the system first comes on-line and, later, when personnel turnover occurs

Checklist 18-1. How To Determine If Configuration Management Is Adequate for Your Program *

✓	Is a configuration management plan documented for the project?
✓	Have the products that will be placed under configuration control been identified?
✓	Is the configuration management process integrated with the project plan and followed as an integral part of the culture?
✓	Are all versions (of configuration items) controlled?
✓	Has an electronic library been established that can store and retrieve multiple baselines?
✓	Are configuration control tools used for status accounting and configuration identification tracking?
✓	Are change requests and problem reports for all configuration recorded, approved, and tracked according to a documented procedure?
✓	Are all changes to baselines controlled in accordance with procedures?
✓	Are all baselines periodically reviewed and audited to assess the effectiveness of the configuration management process?
✓	Are all pieces of information shared by two or more organizations placed under configuration management?

* "The Condensed Guide to Software Acquisition Best Practices, 1997" and [Paulk, 1993].

CHAPTER 21
WHERE TO GET MORE HELP

Software is a very rich topic area. Clearly this document can only scratch the surface. Indeed a comprehensive software bibliography would probably be longer than the entire document. Nonetheless, we'll make a stab at listing some key references where more help is available. We begin by giving some general references, and then follow with references that are pertinent to the various individual chapters of the document.

General references

A good starting point are two classic works by Frederick P. Brooks, which are **must reading** for anyone about to embark on a software acquisition. The first is a journal article; the second a book. Both are easy reading, widely quoted, and contain many pearls of wisdom. Their messages are as relevant today as when they were first written:

- F. Brooks, "No Silver Bullet: Essence and Accidents of Software Engineering", *Computer,* Vol 20, issue 4, pages 10-19, April 1987.
- F. Brooks, *The Mythical Man-Month: Essays on Software Engineering*, Addison-Wesley, 1975. If your time is really limited, then be sure to a least read chapter two, "The Mythical Man Month".

The two works have more recently been combined into a single volume:

- F. Brooks, *The Mythical Man-Month: Anniversary Edition*, Addison-Wesley, 1995 (ISBN 0-201-83595-9). However, this volume does not have the striking graphics that were included in the original journal article listed above.

We found the following book to be a good source of practical advice in managing software projects:

- Steve McConnell, *Rapid Development: Taming Wild Software Schedules,* Microsoft Press, 1996.

The Software Engineering Institute (SEI), an arm of Carnegie Mellon University, is a Federally Funded Research and Development Center set up by the Department of Defense to assist with the transition of software engineering technology. (Even though SEI products were developed for Department of Defense acquisitions, they contain good commonsense software management principals that are also applicable to ITS.) SEI technical documents can be downloaded for free at <URL:http://www.sei.cmu.edu>. The Software Engineering Institute can be reached at (412) 268-5800. Their mailing address is Software Engineering Institute, Carnegie Mellon University, 4500 Fifth Avenue, Pittsburgh, PA 15213-3890.

The Software Technology Support Center (STSC) provides telephone support, technical consulting, and documentation for government agencies. Their on-line documentation,

white papers, and back issues of their Crosstalk magazine can be downloaded for free at <URL:http://www.stsc.hill.af.mil/>. STSC Customer Service can be reached at (801) 775-5555 or via e-mail at custserv@software.hill.af.mil. Their mailing address is Software Technology Support, Ogden Air Logistics Center TISE, Hill Air Force Base, UT 84056.

The Software Program Managers Network, another Department of Defense initiative, wrote four very focused pocket-sized pamphlets. They highlight the essence of good software practice in checklist format. You may want to keep them close at hand and refer to them often during your acquisition. They are entitled:

- *The Condensed Guide to Software Acquisition Best Practices*
- *Little Yellow Book of Software Management Questions*
- *The Little Book of Bad Excuses*
- *Project Breathalyzer*

The Software Program Managers Network can be reached at (703) 521-5231. Their mailing address is Software Program Manager's Network, P.O. Box 2523, Arlington VA 22202. They can be reached over the Internet at <URL:http://www.spmn.com>.

The International City/County Management Association (ICMA) prepared a guide for local government managers who have been called upon to make decisions about the acquisition and implementation of information systems. For more information, call (312) 977-9700.

- Roscoe Sandlin, *Manager's Guide to Purchasing an Information System,* 1996.

The Institute of Electrical and Electronics Engineers has written a number of software standards and recommended practices over the years. *IEEE Recommended Practice for Software Acquisition,* IEEE Std 1062-1993 provides general acquisition guidance for organizations that use software they acquire from suppliers. It describes useful acquisition practices and activities, and includes a number of checklists for organizing and planning your project. IEEE standards on more specific topic areas that are felt to be useful on ITS acquisitions are noted under the appropriate headings below. IEEE documents can be ordered from: The Institute of Institute of Electrical and Electronics Engineers, at (800) 678-4333 or by mail from IEEE Operations Center, Sales Office, 445 Hoes Lane, PO Box 1331, Piscataway, NJ 08855-1331. A list of IEEE software standards appears at <URL:http://standards.ieee.org/catalog/software.html>.

As identified in various places throughout this document, the U.S. National ITS Architecture provides valuable information that can assist in your ITS acquisition. It is available in several formats, including a CD-ROM that can be ordered from the ITS Joint Program Office at (202) 366-9536 or on-line at <URL:http://www.its.dot.gov/architecture/cd_order.html>; downloadable documents from ITS America at <URL:http://www.itsa.org/archdocs.nsf>; on-line browsable hypertext format in HTML at <URL:http://www.odetics.com/itsarch>; or in paper copy that can be purchased from ITS America.

U.S. Government software publications are distributed through the following sources, and are available in electronic or paper media:

Asset Source for Software Engineering Technology (ASSET), Science Application International Corporation, 1350 Earl L. Core Road, P.O. Box 3305, Morgantown, West Virginia 26505. Phone: (304) 284-9000. FAX: (304) 284-9001. e-mail: sei@asset.com Internet: <URL:http://www.asset.com/sei.html>

National Technical Information Service, U.S. Department of Commerce, Springfield, VA 22161-2103, Phone: (703) 487-4600

Defense Technical Information Center, 8725 John J. Kingman Road Suite, 0944 Ft. Belvoir, VA 22060-6218. Phone: (800) 225-3842 or (703) 767-8222.

Chapter-specific references

The following references provide useful information specific to individual chapters of this document.

Chapter 8: Planning The Project

To assist in writing the project plan:
IEEE Standard for Software Project Management Plans, IEEE Std 1058.1-1987 (revised 1993). (See *General references* above for ordering IEEE standards.)

Chapter 9: Requirements

The content and suggested outlines for requirements documents:
IEEE Recommended Practice for Software Requirements Specifications, IEEE Std 830-1993. (See *General references* above for ordering IEEE standards.)

Literally hundreds of texts have been written on the subject of requirements analysis. A widely used book in this area is:
Al Davis, *Software Requirements : Objects, Functions, and States* , Prentice Hall, 1993 (ISBN 0-138-05763-X)

Chapter 11: Selecting the Contracting Vehicle

An excellent report that describes the different types of contracts and contracting approaches used under Federal Aid regulations and discusses the applicability of each:
Booz-Allen, *FHWA Federal-Aid ITS Procurement Regulations and Contracting Options*, August 1997. Distributed by FHWA as one of the procurement guidance initiative documents, it is also available over the Internet at <URL:http://www.tfhrc.gov/qkref/gene/title.htm>

The Information Technology Omnibus Procurement (ITOP) is one of the information technology services run by Transportation Administrative Services Center in U.S. DOT. They provide one-stop-shopping for contractual services, in the areas of information systems engineering, systems facility management, and

information systems security services. This includes access to over twenty pre-certified contractor teams comprising over one hundred contractors with access to many others and other mechanisms to assist in contracting for software. They also provide consulting support such as preparing an RFP, developing an acquisition strategy, drafting an SOW, or gaining access to information management personnel. For more information, contact Dell Berry at (202) 366-1211 or click on the link to the ITOP procedural handbook on the Internet at <URL:http://itop.dot.gov/>.

Chapter 14: Project Scheduling

A good introduction to estimating the size of a software project written by two of the leaders in the field:
Lawrence Putnam and Ware Myers, *Executive Briefing: Controlling Software Development,* IEEE Computer Society Press, 1996.

An overview on cost estimation that contrasts the characteristics of various cost estimation methods such as Price, Slim, COCOMO:
Donald J. Reifer, "Cost Estimation", *Encyclopedia of Software Engineering* by J. Marciniak, Vol. 1., John Wiley and Sons, 1994, pp. 209-220.

The classic text on software cost estimation:
Barry W. Boehm, *Software Engineering Economics,* Prentice-Hall, 1981 (ISBN 0-13-822122-7).

A number of authorities believe that a technique called function points is better for estimating projects that are the more traditional lines of code. Beyond the scope of this document, function points are described in many places including:
David Garmus and David Herror, *Measuring the Software Process: A practical Guide to Functional Measurements,* Yourdon Press, 1996 (ISBN 0-13-349002-5).
Capers Jones, *Applied Software Measurement: Assuring Productivity and Quality,* McGraw-Hill 1991 (ISBN 0-07-032813-7).

Chapter 16: Training, Operations, and Software Maintenance

The Institute of Transportation Engineers is developing a recommended practice entitled "Operations and Management of Intelligent Transportation Systems". It includes a chapter on computer systems that addresses software issues. When it receives final approval (expected in early 1998), the document can be ordered from the ITE Bookstore, 525 School St. SW, Suite 410, Washington, DC 20024-2797.

The text by T. Pigoski, *Practical Software Maintenance: Best Practices for Managing Your Software Investment,* John Wiley & Sons, 1997 provides an up-to-date view of software maintenance by a professional who has a number of "practicing" years of experience in this discipline."

Chapter 17: Project Management

An excellent source for information on the various plots of software metrics and their use:

The MITRE Corporation, *Software Reporting Metrics Revision 2,* November 1985. It can be ordered for free by writing to Document Control, The MITRE Corporation, 202 Burlington Road, Bedford MA 01730. Ask for document ESD-TR-85-145.

Contents of software quality assurance plans:
IEEE Standard for Software Quality Assurance Plans, IEEE Standard 730-1989. (See *General references* above for ordering IEEE standards.)

To assist with independent verification and validation (IV&V):
IEEE Standard for Software Verification and Validation Plans, IEEE Standard 1012-1986 (revised 1992). (See *General references* above for ordering IEEE standards.)

Chapter 19: Software Risk Management

Tables of software risk areas:
Robert N. Charette, *Software Engineering Risk Analysis and Management,* 1989, page 186, table 5.2.
J. Marciniak, Software Acquisition Management, page 131, table 6-1.

Topic Sheet 2: Security

The National Institute of Standards and Technology maintains a comprehensive Computer Security Resource Clearinghouse at:
<URL:http://csrc.ncsl.nist.gov/>

The CERT Coordination Center provides 24-hour technical assistance for responding to computer security incidents, provides product vulnerability assistance, takes proactive steps to raise the community's awareness of computer security issues through technical documents and seminars, has security tools, and conducts research targeted at improving the security of existing systems.
<URL:http://www.cert.org/>

There are a number of commissions and committees addressing security issues:

- The President's Commission on Critical Infrastructure Protection (PCCIP) <URL:http://www.pccip.gov>
- The International Institute for Surface Transportation Policy Studies (IISTPS) <URL:http://transweb.sjsu.edu>
- The National Science and Technology Council (NSTC) Research and Development Committee is investigating information security in the transportation domain. <URL:http://www.volpe.dot.gov/pubs/nstc/>

See also the bibliography of texts, periodicals, and Internet material listed in the *ITS Information Security Analysis* report cited in the topic sheet.

Topic Sheets 3 and 4: SA-CMM and SW-CMM

See *General references* above for ordering SEI and other government publications through ASSET, NTIC, and DTIC.

Topic Sheet 5: Software Safety

IEEE Standard For Software Safety Plans, IEEE Standard 1228-1994. (See *General references* above for ordering IEEE standards.)

Two leading texts on software safety are:

- Nancy Leveson, *Safeware: Systems Safety and Computers,* Addison-Wesley 1995 (ISBN 0-201-11972-2)
- Peter Neuman, *Computer-Related Risks, ACM Press,* Addison-Wesley 1995 (ISBN 0-201-55805-X)

CHAPTER 22
CONCLUDING REMARKS

We hope we have been able to help you with your ITS software acquisition. We started out by explaining how software is different. The rest of the document is essentially our recommended response to that difference. We didn't give you the classic answer traditionally used on large government acquisitions. That "solution" imposes an excessive documentation burden on an acquisition in an attempt to control it. And in spite of all the paper products that are required (or perhaps, because of them), the projects are still late, over budget, and perform poorly if at all.

Instead, we have taken a process-oriented approach. It is centered around a series of themes that deal with the system, the management approach, and most importantly, the people. Then we built upon those themes, showing how they play out in certain key activities.

To be sure, we haven't been able to give you all the answers. Your software acquisition will still be hard work, requiring your hands-on, active management involvement. There are no quick fixes, panaceas, or cure-alls. No silver bullets. The road ahead may not be a totally smooth one. But perhaps it will be less bumpy because you'll know the potholes to avoid. And that may keep small risks from growing into major problems.

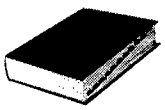
As Brooks wrote in his classic paper *No Silver Bullet: Essence and Accidents of Software Engineering,* "There is no easy road, but there is a road." Good luck!

PART SIX:

TOPIC SHEETS

TOPIC SHEET 1
RAPID PROTOTYPING

Rapid prototyping is the recommended technique for fleshing out human interface requirements. This topic sheet provides more information on this subject.



*The **human interface** represents all the interactions between a person and a computer system. It includes user inputs (including data and commands), the system's output to the user, and how those outputs sound or appear (such as on a screen or in a printed report). It also includes how data and commands are specified by the user to the computer: does the user type something in, select something from a menu, click on a button, use a mouse or a keyboard or a touch screen or voice input, and other considerations.*

In rapid prototyping, the developers build a "mock-up" of a human interface. The users are then given the opportunity to "kick the tires" of the prototype. The users provide comments on what they've seen, and suggest changes to the interface. The developers then refine the prototype based upon this feedback, show the revised interface to the users, who then provide further comments. This iterative process is repeated until it converges on an acceptable human interface for the system. The actual system is then built using this interface, with the rapid prototype serving as the requirements "document".

Themes

*Rapid prototyping is an effective method for **collaboration** and **open communications** between users and developers. It provides a way for end-users to give direct feedback to the software developers who act upon what they have heard and solicit further feedback from the users. Success is achieved through continuous, on-going interactions between customer and developer; thus, end-users must be made available for any rapid prototyping activity.*

How can rapid prototyping achieve such fast turn-around when actual systems can take months or even years to develop? The answer is that the prototype is only a partial solution to the final product. There is nothing "behind the screen". "Canned" or "dummy" data are used. Databases are not updated; messages and data are not sent to other systems; and the prototype does not fully edit all data. The software needed to provide the full functionality and keep a system running accurately and smoothly, even if some components fail, is often the most difficult and time consuming to develop; such software is generally not part of a prototype.

Rapid prototyping offers a number of advantages:

- **Better human interfaces.** Because of the iterative process described above, rapid prototyping results in a human interface that has been refined and improved many times. The interface is tailored to the needs and wants of the user. Contrast this with the traditional approach, in which the first cut of the human interface (often built from the developer's viewpoint) is the one that gets implemented and used.

- **User ownership and "buy in".** Users see their suggestions take shape in various iterations of the prototype. They take a sense of ownership on what is built.
- **Visibility.** Rapid prototyping directly addresses the problem that paper design documents may give little insight as to how the final system will look or feel. Rapid prototyping gives the customer something tangible to review. Similarly, for those who are defining the requirements, it gives them an opportunity to see how their ideas might work.

Depending on the application, rapid prototyping can be limited or can result in extensive representations of the eventual system. In some cases, pictures of sample computer screens may suffice. Paper drawings or software could be used to produce them without any actual programming taking place. This would be appropriate, for example, in defining the layout of various management reports. In other cases, a more extensive prototype may be needed. This would allow users to interact with the prototype and see how it responds. Users can conduct a hands-on "test drive" of menus, commands, buttons, data entry and error correction techniques, etc. There are modern software products that can be used to develop these higher fidelity prototypes quickly.

There are also some negatives to rapid prototyping:

Themes *Rapid prototyping is not a* ***silver bullet.***

- People sometimes (too often) mistake the rapid prototype for the "real thing". They see the prototype working and don't understand why that same capability can't be replicated at once in the real system. They don't appreciate that there is little to nothing behind the panel in the prototype. This is very different from hardware, where no one would mistake an automobile mock-up for the real thing.
- Rapid prototyping activities can take on a life of their own. Rapid prototyping can be fun when users see how their ideas come to life. "The customer goes crazy adding features and making changes." [Gordon and Bieman, 1995]

Several good practices can help make rapid prototyping activities a success and overcome the problems cited above:

- Carefully plan the activities. What will be prototyped? How will users evaluate the prototype? Will questionnaires be used? (If so, they must be developed.)
- Set a rigid deadline for cutting off all rapid prototyping activities.
- If rapid prototyping surfaces new requirements, carefully consider their schedule and budget impact before deciding to implement them. Don't try to squeeze more into the existing project. Either provide schedule and budget relief for implementing the new requirements, or keep to the existing schedule by foregoing other requirements and implementing the new ones instead.

Reference cited

V. Gordon and J Bieman, "Rapid Prototyping: Lessons Learned", *IEEE Software,* Vol. 12, No. 1, page 85, 1995

TOPIC SHEET 2
SECURITY

With the development and deployment of ITS, there is a growing reliance on information and information systems. Unfortunately, events that can harm these systems, known as *threats*, are evolving almost as rapidly as the technologies themselves. Furthermore, as the ITS systems are opened up to make traveler information more available to the public, the security risks from outside attacks increase. The potential impacts of such threats lead to significant concerns regarding public safety and emergency-response effectiveness, corruption of financial transactions, violations of citizen privacy, and the loss of credibility. Thus there is need for information *security* to protect these systems on which we all depend.

Incorporate security from the outset

Frequently, information security has been neglected during system acquisition. Then an event occurs that causes some harm. And, almost always, much embarrassment! Only then are attempts made to retrofit a system with the necessary information security mechanisms. Do not make this mistake. Instead, plan security into your system from the outset. This not only avoids problems, but it is also more cost effective: As with other types of system requirements, the cost of incorporating security increases at a significant rate as system development proceeds from concept, through development, to operations and maintenance.

What can go wrong if you don't?

In one state a hacker displayed an obscene message with the governor's name on a variable message sign. Although this incident was only embarrassing and caused no real damage, a similar intrusion could pose risk of death or serious personal injury. Consider what would happen if, for example, messages were altered to direct freeway traffic onto a entrance ramp of an unfinished bridge. Other hypothetical, but realistic examples of what can happen without adequate security measures are:

- Personal information collected to establish electronic toll transaction account could be compromised, threatening its confidentiality or leading to unauthorized use of credit cards.
- Traveler privacy could be violated by unauthorized access to a database of toll transactions. This would reveal travel patterns, including time and place of trips. Indeed, in a related real-world example, hackers have publicly targeted a transit fare card database that records all trips taken by transit riders.
- By gaining access to information systems, travelers could set up unauthorized accounts to bypass paying electronic tolls and get free trips.
- By forging border-clearance information, an illicit commercial vehicle operator could be authorized to make illegal shipments.

What types of security are needed?

Following are some of the technical and non-technical security *services* that are need to counter the major security threats.

Technical security services include:

- *Confidentiality* helps restrict sensitive information from disclosure. Confidentiality applies to both information storage and transmission.
- *Authentication* verifies one's identity or membership in a group.
- *Data integrity* ensures that information is not modified while stored or transmitted except by authorized users.
- *Non-repudiation* prohibits the sender or receiver of a transaction from subsequently denying the action.
- *Access control* regulates who can access a system or specific information and what they are allowed to do with it (e.g., read, write, modify, execute).
- *Accountability* attributes actions to the users who perform them.

Non-technical security services include:

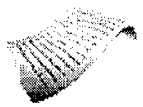
- *Administrative security* includes establishing and implementing procedures to protect the organization's information resources.
- *Personnel security* assures that employees, both local and remote to the information systems that they access, can be trusted in ways appropriate to their responsibilities.
- *Physical security* is concerned with protecting the organization's personnel as well as its buildings, offices, equipment, and products from harm, destruction, and unauthorized access.

How is security achieved?

Technical security services are implemented by various software, hardware, and/or procedures, known as security *mechanisms*. Some of the more common mechanisms are:

- *Encryption* disguises data so that it cannot be read unless decrypted ("secret codes"). Encryption is used to provide confidentiality. Depending upon the specific technique, it also provides authentication, integrity, and non-repudiation.
- *Digital signatures* are the electronic equivalent of a hand-written signature and can be attached to electronic transactions. This security mechanism is primarily used to provide non-repudiation; however, authentication and integrity can be provided as well.
- *Passwords* allow users to identify themselves to a system. They are used for authentication.

- A *firewall* is a collection of hardware and software components placed between networks to protect one network from another (e.g., to protect a private network from the Internet). Firewalls provide access control to an organization's internal network.



Firewalls are currently very popular. Unfortunately, some organizations mistakenly view them as a panacea for their security needs. However, they are only a partial answer and must be used in conjunction with other information security mechanisms. In particular, confidentiality, integrity, and authentication are not provided by firewalls and must be provided by other mechanisms. Also, firewalls do not protect against internal threats.

Security planning

To be effective, security should be implemented in accordance with an information security program plan that ties together the technical and non-technical services. Regardless of the specifics, here are some of the key points to consider:

- Security must be multi-faceted. No one service or mechanism addresses all the threats. Focus on the entire system, not just database security or physical security.
- Designate a central point of contact for the information security program.
- Obtain "buy-in" (i.e., acceptance, concurrence) from the major players such as acquisition managers, development teams, end-users, and other stake-holders.
- Only provide as much security as warranted by the value of the information. Ask yourself the following question: How serious would the consequences be if a certain type of information is manipulated, disclosed, or destroyed? Then act accordingly. In other words, balance the costs and risks.
- Determine which types of services you will need; only then should the appropriate mechanisms be specified. (i.e., requirements first, design later).
- Recognize that because of the rapid evolution of information technologies, no security solution will be permanent and that you will never achieve total security.

For more information on security

The U.S. DOT Intelligent Transportation Systems Joint Program Office (JPO) is distributing a packet of documents that specifically address ITS security. Contact Mr. William S. Jones of the JPO at (202) 366-2128 or william.s.jones@fhwa.dot.gov *for copies. Included in the packet are the following documents:*

"Protecting Our Transportation Systems: An Information Security Awareness Overview," Mitretek Systems, Inc., 1997.

[A brief, easy-to-read security awareness overview gives answers to security-related questions. This high-level paper is directed at transportation agency senior management.]

"Intelligent Transportation Systems (ITS) Information Security Analysis," Mitretek Systems, Inc., 1997.

[A security analysis of the National ITS Architecture. This document characterizes the various threats to ITS subsystems, their exchanging of information, and the supporting communications infrastructure. It also recommends solutions that can be used to reduce or eliminate the identified threats.]

"Maryland ITS Security Requirements Recommendations" and "Maryland ITS Security Implementation Recommendations," Computer Sciences Corporation for Volpe National Transportation Systems Center (NTSC), 1997.

[Follow-on studies to the above documents. They recommend implementation of security guidelines and describe steps necessary to achieve a minimal level of security for Maryland ITS.]

TOPIC SHEET 3
SOFTWARE ACQUISITION CAPABILITY MATURITY MODEL (SA-CMM[SM])

What is the SA-CMM?

The Software Acquisition Capability Maturity Model (SA-CMM) specifies a set of desirable software *acquisition* practices. (In contrast, the Software Capability Maturity Model, or SW-CMM, specifies a set of desirable software *development* practices.) The SA-CMM is intended to be used by individuals and organizations who are planning and managing software acquisitions. Developed by the Software Engineering Institute for use by the Department of Defense, the SA-CMM has potential for use by traffic and transit agencies. In particular, agencies can use it as a self-assessment tool for determining their readiness to embark on ITS software acquisitions.

The practices set forth in the SA-CMM are collected into five levels called maturity levels with higher levels building upon the lower ones. Most agencies start at level 1. For the SA-CMM, the maturity levels are characterized as follows:

- *Level 1 Initial.* Acquisition processes are ad hoc, and occasionally even chaotic.
- *Level 2 Repeatable.* Basic software acquisition project management processes are established to plan and manage all aspects of the acquisition.
- *Level 3 Defined.* The software acquisition process is documented and standardized. All projects use approved, tailored versions of this process.
- *Level 4 Quantitative.* Detailed measures are collected. Acquisitions are quantitatively and qualitatively understood and controlled.
- *Level 5 Optimizing.* Continuous process improvement is achieved by quantitative feedback from the process.

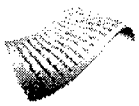
As shown in the chart below, each maturity level includes a set of Key Process Areas (KPAs). In turn, each Key Process Area has a set of goals and activities that define objectives and the actions to be taken to meet those objectives. An agency at level 2, for example, would carry out the activities that define such Key Process Areas as "Contract Tracking and Oversight" or "Requirements Development and Management". The "focus" of level 2 Key Process Areas is to achieve basic project management skills. This may be sufficient for most agencies.

An organization is operating at a given maturity level when it satisfies the requirements for all Key Process Areas in that level and all lower levels. A maturity level is associated with an organization's capability to acquire software. In general, as organizations move up in maturity level, they improve their capability.

[SM] SA-CMM is a Service Mark of Carnegie Mellon University

Software Acquisition Capability Maturity Model (SA-CMM)

Level	Focus	Key Process Areas
5 Optimizing	Continuous Process Improvement	• Acquisition Innovation Management • Continuous Process Improvement
4 Quantitative	Quantitative Management	• Quantitative Acquisition Management • Quantitative Process Management
3 Defined	Process Standardization	• Training Program • Acquisition Risk Management • Contract Performance Management • Project Performance Management • Process Definition and Maintenance
2 Repeatable	Basic Project Management	• Transition to Support • Evaluation • Contract Tracking and Oversight • Project Management • Requirements Development and Management • Solicitation • Software Acquisition Planning
1 Initial	Competent People and Heroics	



Although maturity levels and Key Process Areas may sound intimidating, they actually only represent desirable practices that are good managerial common sense. Consider, for example, Contract Tracking and Oversight, one of the level two Key Process Areas. Contract Tracking and Oversight has four goals and six activities. One of the goals is stated as "The project team and contractor maintain ongoing communication and commitments are agreed to by both parties." One of the activities is stated as "The project team conducts periodic reviews and interchanges with the contractor," with a sub-activity of "The actual progress and cost of the contractor's software engineering process is compared to planned schedules and budgets." An assessment would ascertain whether these activities are carried out.

How can the SA-CMM be used for ITS software acquisitions?

Realistically, most transportation agencies will probably not use the SA-CMM in carrying out a multi-year process to move up to higher maturity levels. The higher levels are more appropriate for organizations that repeatedly perform software acquisitions of significant scope and magnitude; level 2 is more appropriate for smaller organizations with one-time projects. Instead, for ITS, an agency would probably use the SA-CMM primarily as a tool for assessing its software acquisition practices before embarking on an ITS project. Areas of strength and weakness would be identified. The assessment results could then form the foundation for making improvements. In using the SA-CMM in this fashion, it should be kept in mind that it was originally developed for organizations that repeatedly perform software acquisitions of significant scope and dollar value. So it may be overly formal for many ITS acquisitions. For example, it calls for having a project

training plan (a good idea) that is written in accordance with training program procedures (may be overkill). If you stick with the essence of the SA-CMM (in particular, the activities), ignore much of the other material (e.g., defining maturity levels for your acquisition agency), and do not treat it as a rigid standard, it may prove valuable for you.

For smaller acquisitions, the SA-CMM can be tailored by omitting non-applicable portions, and by scaling to fit. For example, the SA-CMM calls for a number of management plans and it outlines what practices should be addressed in the plans. For a large acquisition, such a plan might be many pages long, but for a small acquisition it could be only one or two pages. Or you may decide to carry out the various practices without formally documenting them in a plan. Decisions can be made as to which practices will be performed in-house, which must be contracted for, and which are not applicable to your particular project. Also, the Key Process Areas can be modified for internal use to fit the project; activities can be added, deleted, or changed if desired.

The SA-CMM can also serve as a framework for a process improvement program. In particular, the maturity levels can serve as goals: "We want this organization to operate at Level 3." The Key Process Areas provide further definition of the goals by defining functional areas of improvement, such as planning, doing the solicitation, overseeing the contractor, and so on. However, implementing a process improvement program is not a trivial undertaking. It requires serious organizational commitment and management involvement to ensure positive results.

An SA-CMM assessment can also produce a maturity rating, from 1 to 5. This benchmarks the organization relative to the SA-CMM, and relative to other organizations that have been assessed using the SA-CMM.

What the SA-CMM is NOT

The SA-CMM defines *what* you should do within a given Key Process Area, but it does not prescribe *how* to carry out a software acquisition. Instead, it characterizes practices at the various maturity levels, with the idea that the project or organization will develop or acquire practices with the desired characteristics. Procedures to satisfy the desired practices are the responsibility of the implementing organization.

Nor is the SA-CMM a process improvement method. The SA-CMM can form the *basis* for a process improvement effort, when used as a diagnostic and goal-setting tool. But it does not articulate the steps an organization must take to succeed at process improvement.

Finally, the SA-CMM is not a commercial or proprietary product.

For more information on the SA-CMM

The SA-CMM is described in the SEI technical report CMU/SEI-96-TR-020, "Software Acquisition Capability Maturity Model (SA-CMM℠), Version 1.01," by J. Ferguson et al. It can be viewed and downloaded without charge through the SEI's World Wide Web site at: <URL:http://www.sei.cmu.edu/technology/risk/Risk_SW_Acq/SA-CMM.html>

TOPIC SHEET 4
SOFTWARE CAPABILITY MATURITY MODEL (SW-CMM[SM])

What is the SW-CMM?

The Software Capability Maturity Model (SW-CMM) specifies a set of desirable software *development* processes. (In contrast, the Software Acquisition Capability Maturity Model, or SA-CMM, specifies a set of desirable software *acquisition* practices.) Developed by the Software Engineering Institute for the Department of Defense, the SW-CMM has been successfully applied in the defense and other sectors. On ITS software acquisitions, it has two potential types of applications:

- The SW-CMM could be used by traffic and transit agencies (the customers) to assess the capability of a contractor to develop software.
- It could be used by the software development contractors as a basis for their process improvement programs.

The SW-CMM was developed upon the central premise that the quality of a software system is largely governed by the quality of the process used to develop and maintain it. This premise is not new—it has been the foundation of Total Quality Management (TQM) programs world-wide for several decades. The desirable software development processes are grouped into five maturity levels as shown in the chart below.

Each maturity level represents a level of refinement and integration of an organization's processes for software development. For the SW-CMM, the maturity levels are characterized as follows:

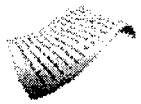
- *Level 1 Initial.* Processes are informal, ad hoc, and largely unrepeatable. Results depend upon the initiative of competent personnel and often involve "heroics" to achieve objectives.
- *Level 2 Repeatable.* The process has become documented, disciplined, and repeatable.
- *Level 3 Defined.* The process is promulgated organization-wide into a standard, consistent process.
- *Level 4 Managed.* The process becomes predictable. Metrics can be used to maintain performance within expected boundaries.
- *Level 5 Optimizing.* Process improvement is systematically applied to the software process, and is institutionalized.

[SM] SW-CMM is a Service Mark of Carnegie Mellon University

Software Capability Maturity Model (SW-CMM)

Level	Focus	Key Process Areas
5 Optimizing	Continuous Process Improvement	• Defect Prevention • Technology Innovation • Process Change Management
4 Managed	Product and Process Quality	• Process Measurement and Analysis • Quality Management
3 Defined	Engineering Process	• Organization Process Definition • Organization Process Focus • Peer Reviews • Training Program • Intergroup Coordination • Software Product Engineering • Integrated Software Management
2 Repeatable	Project Management	• Software Project Planning • Software Project Tracking and Oversight • Software Subcontract Management • Software Quality Assurance • Software Configuration Management • Requirements Management
1 Initial	Competent People and Heroics	

Each level of maturity includes a set of Key Process Areas (KPAs) as shown in the chart. An organization is operating at a given maturity level when it satisfies the requirements stated for all the Key Process Areas in that level and all lower levels. Each Key Process Area has a set of goals and activities. These define objectives, and the actions to be taken to meet those objectives, which a development organization should follow in order to achieve a given level of process maturity.



Although the use of maturity levels and Key Process Areas may sound intimidating, they actually only represent desirable practices that are good managerial common sense. For example, consider Requirements Management, one of the level two Key Process Areas. Requirements Management has two goals and three activities. One of the goals is stated as "System requirements allocated to software are controlled to establish a baseline for software engineering and management use." One of the activities is stated as "Changes to the allocated requirements are reviewed and incorporated into the software project," with a sub-activity of "The impact to existing commitments is assessed, and changes are negotiated as appropriate." An assessment would ascertain whether these activities are carried out.

How can the SW-CMM be used for ITS software acquisitions?

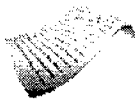
There are several ways that the customer can use the SW-CMM:

- To assess and compare the ability of bidders to develop the software. (Or, stated negatively, to assess the risk of a contractor not being able to successfully develop the software.) Some of the SW-CMM practices could be posed as questions to bidders during source selection. ("Describe how your organization does...") Consider doing this orally in face-to-face meetings with bidders, instead of requesting written responses in the RFP.
- To require that bidders be at a certain level of maturity. RFPs have been written that require bidders to demonstrate that they "satisfy the requirements of Level X of the SW-CMM." However, this may not be the best use of the SW-CMM for ITS. Among smaller companies and niche segments of the software marketplace, the SW-CMM may not yet have been applied (or even heard of!). A possible result is that an RFP with maturity level requirements may attract *no* qualified bidders or significantly reduce the competition. To overcome this, the RFP can ask the offerors what level of software process maturity has been achieved according to the SW-CMM. This places the burden on the developers without undue restrictions in the RFP and the procurement process.

When used by software development contractors, the SW-CMM serves as a framework for process improvement. This is the originally intended purpose of the SW-CMM. An organization can strive to achieve a certain level of maturity. However, moving "up the ladder" takes time (typically 18 months to go up one level), commitment, and resources. In spite of documented cases of the benefits of moving up to higher levels, it is probably not cost-beneficial or necessary for all organizations to achieve level five; level two may be sufficient for much of the ITS software.

Section 5204 of the Transportation Equity Act for the 21st Century (TEA 21) calls for the use of the SW-CMM or "another similar recognized standard risk assessment methodology, to reduce the cost, schedule, and performance risks associated with the development, management, and integration of intelligent transportation system software." However, this does not mean that you should blindly apply the SW-CMM to your project. Guidance will be forthcoming from U.S. DOT on tailoring the SW-CMM and using it in an appropriate manner for ITS.

The SW-CMM should be exercised with judgment. For simple and/or off-the-shelf software products (like signal controllers), the SW-CMM would be overkill. (If an off-the-shelf product has proven itself in the marketplace, who really cares what processes were used to develop that product?) It is most effective where the software is significant in terms of dollar value or program risk, and where it is applied with common sense and tailoring appropriate to the situation.



Transportation officials are encouraged to:

- *Discuss intended use of the SW-CMM as a performance metric with potential bidders to determine their understanding and use of the CMM prior to committing to an acquisition strategy.*
- *Consider a "carrot" approach (rewards for achieving maturity levels) rather than a "stick" (penalties or disqualification for not being at a certain maturity level).*
- *Consider requiring the contractor to have a plan to achieve a maturity level and demonstrate progress against that plan as opposed to a hard requirement to be at a certain level. This allows the transportation community of software providers to grow their maturity gracefully.*
- *Use the results of an assessment to evaluate and compare a bidder's capability, rather than as a pre-qualification to bidding on an RFP.*
- *Seek assistance from organizations experienced with the CMM, when developing an acquisition strategy that uses the SW-CMM as a requirement.*

Who carries out capability assessments using the SW-CMM?

Capability assessments are used to benchmark the performance of a software development organization and determine its maturity level. Although the Software Engineering Institute (SEI) developed the SW-CMM, it does not carry out these assessments. Instead, the SEI authorizes individuals, not companies, to perform assessments and maintains a roster of those individuals that meet SEI's qualifications. Assessments can be performed in two ways:

- By an outside auditing group conducting a Software Capability Evaluation under the leadership of an SEI-authorized Lead Evaluator. This is the approach used by customers to assess contractors.
- By an organization using its own personnel and guided by an SEI-authorized Lead Assessor. This is the preferred approach for process improvement programs.

Of course, the SW-CMM could be used less formally. Instead of a formal assessment, it can serve as the basis for questions to ask contractors, or as a contractor's checklist of good development practices to follow.

What results has the SW-CMM produced?

The SW-CMM's use began with Department of Defense contractors in the early '90's in response to RFPs. Recently there has been a surge of commercial developers beginning to use the CMM as a means to improve. Acquisition agencies have also found the SW-CMM to be a useful tool in source selection.

In addition to the extremely important qualitative benefits of process maturity described above, there are now available documented cases [Fox, 1997; Vu, 1997] of the quantitative benefits of maturity-based process improvement efforts that are based on the

SW-CMM. Returns on investment in the range of 5:1 to 7.7:1 have been reported, and cost and schedule performance improvements quantified. Published data across many projects [Jones, 1998] show that the risk of project failure is much lower for organizations at higher maturity levels.

References cited

C. Fox and W. Frakes, "The Quality Approach: Is it Delivering?", *Communications of the ACM,* Vol.40 No. 6, June 1997.
C. Jones, "Becoming Best in Class: Application Development and Productivity", Software Productivity Research, 1998.
J. Vu, "Software Process Improvement Journey (From Level 1 to Level 5)", *2nd European Software Engineering Process Group Conference,* June 1997.

For more information on the SW-CMM

The SW-CMM is described in two SEI technical reports by M. Paulk, et al.: CMU/SEI-93-TR-024, "Capability Maturity Model for Software[SM], Version 1.1"; and CMU/SEI-93-TR-025, "Key Practices of the Capability Maturity Model[SM], Version 1.1." These reports and related material may be viewed and downloaded without charge through the SEI's World Wide Web site at:
<URL:http://www.sei.cmu.edu/publications/>

An SEI Appraiser Directory lists individuals qualified to carry out assessments and their firms. The directory is available at:
<URL:http://www.sei.cmu.edu/topics/managing/appraiser.listing.html/>

A hard copy can be obtained without charge through the SEI's Customer Relations office at:
(412) 268-5800 *or* customer-relations@sei.cmu.edu

TOPIC SHEET 5
SOFTWARE SAFETY

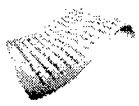
"Safety is not an attribute that can be added to software after the event; it must be designed into the software from the start, and it must be constantly checked to ensure that unexpected, unsafe, functions have not been added or necessary functions have not been removed." — [Place and Kang, 1993, page 45]

"The Sydney Harbour Tunnel was closed to traffic today following a computer problem, which threw peak hour traffic into chaos." — [AAP Newsfeed, August 19, 1997]

"'Serious interruptions' on AT&T's high-speed fiber optic data network began Monday afternoon and caused problems for hundreds of multinational banks, travel agencies, insurance and credit card companies. The problem appeared to be the result of software errors." — (USA Today, April 14, 1998)

Software safety is concerned with ensuring that the software does not cause hazardous or life-threatening conditions to occur during its operation. Obvious examples would be simultaneous green lights for crossing traffic at an intersection or open entrance ramps at either end of a reversible HOV lane. Depending upon your definition, software safety can also be said to concern highly undesirable conditions, even if they're not life threatening. You clearly don't want all the lights to be green, but you don't want all of them to be red either.

As software takes over more and more functions, there is increasing danger that it will allow just such conditions to occur. In the future, if software provides collision avoidance and vehicle control functions, software safety issues will be even more critical.



There have been several well documented cases of unsafe software. Perhaps the most notorious was the Therac-25 therapeutic X-ray machine that literally fried several patients to death because of a software bug. Another example was the Ariane-5 rocket that exploded shortly after launch because of a software problem.

Safe software is not the same thing as reliable software or secure software. All software has bugs and will fail from time to time. Software safety is concerned with taking steps to mitigate the problems that arise from the bugs or from breaches in security.

What can be done to ensure that your software is safe?

Software safety is a highly technical area that sometimes involves mathematical analyses and formal software design and verification techniques. We can only scratch the surface here. The important point is to make sure your acquisition has a mechanism to address safety issues. If there are significant safety considerations, then a safety analysis should be carried out by competent, expert sources. Consider contracting separately for a safety analysis to be conducted by persons who have expertise in this area. If nothing else, ask your development contractor what techniques they will use to address software safety. If that draws a blank stare, you may have problems.

How can you ensure that life-threatening, hazardous, or other highly undesirable events that you absolutely, positively, never what to happen, indeed do not occur? In other words, what can be done to ensure that your software is safe?

First of all, don't eliminate hardware interlocks. Interlocks are devices that prevent some (bad) state from occurring. When it comes to safety, don't succumb to the economic incentives to replace hardware with software. If you want to supplement hardware interlocks with software, that's fine. Software can be used as a back-up for the hardware. But don't rely exclusively on software to maintain the safety of your system. There is no such thing as a software interlock!

Earlier "old fashioned" versions of the Therac X-ray machine had hardware interlocks to prevent people from receiving excessive X-ray doses. The newer "more modern" model 25 relied exclusively on the software, resulting in a loss of life.

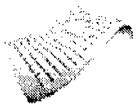
Second, incorporate safety subsystems into your overall system. No single point of failure within a safety system should prevent it from managing unsafe events.

Third, there are managerial techniques that can be applied to minimize the risk. Roughly, they comprise the following steps:

- Recognize that the software does not operate in an ideal world, and that no matter how much you test, it will still have bugs and will never be error-free.
- Make a list of all the hazardous conditions that could conceivably arise from your system. This is best done by domain experts, not software experts.
- Working backwards, taking each hazardous condition one at a time, identify what could conceivably cause the condition to occur. Consider such things as operator failures, something in the environment, hardware failures, software processes running out of sequence, inputs triggering the wrong module to be run, software processes aborting due to bugs, or hardware failures.
- To the greatest extent possible, isolate from the rest of the system those portions of the software that are deemed to have a potential safety impact. A well designed system will have a relatively small number of safety-critical components relative to the total number. Subject these portions of the software to extra scrutiny and special so-called "formal" design and test techniques. (Note: It is overkill to treat

an entire system as safety critical. For one thing, it would be prohibitively expensive. For another, it would probably result in less safety, not more, as attention would be spread across the entire system. It is better to focus the extra attention where it is truly needed.)

- If you change safety-critical software, repeat the entire safety analysis before releasing the software.
- Supplement the software with hardware and administrative protections.



At least one state visually inspects the entrances to reversible lanes to ensure that both ends are not open simultaneously. There are also reports of surveillance cameras being used to check messages on variable message signs. In other words, administrative procedures are used just in case the software has failed.

Finally, here are some safety-related questions that you can ask your developer:

- Have the life-threatening, hazardous, and other highly undesirable events for this system been identified and documented?
- How is the system designed to address each such event?
- Has an analysis been performed and documented to ensure that no single failure within the safety system can prevent successful management of these events?
- Has a single-point-failure analysis been performed and documented to assure that no single failure within the safety system can prevent successful management of a such an event?
- Has a separate review been performed to identify common cause failures that could occur? These would include manufacturing errors, operator errors, maintenance errors, and potential system design defects.
- Does normal system operation include tests conducted at scheduled intervals to detect failures and verify system operability?
- Has an independent verification and validation review been performed on the system? on the software? If so, was the software review in accordance with *IEEE Standard for Software Verification and Validation*?
- Have coding standards that prohibit "error prone" coding techniques been used?
- Does the software incorporate continuous, self-checking diagnostics?
- Are security algorithms in place to prevent intentional software tampering?
- When the software encounters an abnormal condition, at any point, due to power failure, missing or bad data, can it re-start or initialize automatically and successfully? Does this activity include start-up diagnostics?
- Has an abnormal conditions and events analysis been performed and documented on this software?

References cited

Institute of Electrical and Electronics Engineers, *IEEE Standard for Software Verification and Validation,* IEEE Std 1012-1998.

P. Place and K. Kang, *Safety-Critical Software Status Report and Annotated Bibliography,* SEI Technical Report CMU/SEI-92-Tr-5, June 1993. Available at <URL:http://www.sei.cmu.edu/publications/documents/93.reports/93.tr.005.html>.

For more information on software safety

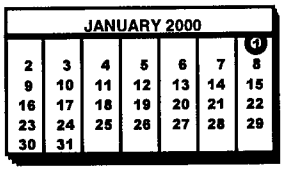
Nancy Leveson, "An Investigation of the Therac-25 Accidents," IEEE Computer, Vol. 26, July 1993, pages 18-41.

[A readable and scary account of what can go wrong when safety is not properly considered. The lessons are applicable to a wide range of applications.]

Bradford Ulery and Charles C. Howell, "Safety-Critical Software on Our Highways," Mitretek Publication WN 96W25, March 1996.

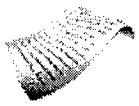
[A closer look at what's been done on software safety in other domains and the applicability to ITS.]

TOPIC SHEET 6
THE YEAR 2000 PROBLEM (Y2K)



Nature of the problem

The Year 2000 Problem, also known as Y2K, has recently received a lot of media attention (some would say media hype). Essentially it boils down to problem software not being able to handle the transition from the 20th century to the 21st correctly. In some cases, the year 2000 is treated as 1900. So according to the computer, December 31, 1999 is followed by January 1, 1900. In other cases, there may be more bizarre manifestations.



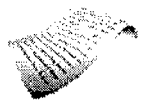
One interviewee pointed out that this is not the first time the transportation industry has faced date and time problems with software. A similar situation arose when Congress changed the law that specifies the dates when Daylight Savings Time takes effect. Some traffic controllers had the old dates built into them, and had to be reprogrammed. This rework was eligible for Federal Aid funds with a proviso: The fix had to make the controllers programmable to readily accommodate any future changes in the Daylight Savings Time law.

What happens if your software has the Y2K problem? The consequences can range from mildly irritating (e.g., entries in management report are incorrectly sorted by date) to significant (e.g., traffic signal controllers begin using weekend timing plans on weekdays) to profound or even life threatening (e.g., emergency management systems fail to work altogether).

Related problems

There are three other Y2K type of problems to be aware of:

- The year 2000 is a leap year. That is, there *will* be a February 29, 2000. Some software incorrectly assumes that 2000 is not a leap year and that February will only have 28 days that year.



The Y2K problem is most commonly encountered with older systems whose software is written in the COBOL programming language. People, therefore, erroneously assume that the problem will not affect their newer systems. However, this is not necessarily true; new systems are not immune to the problem and there are recently-developed systems that have the problem. Consider, for example, a recent best-selling textbook on Java, a new and popular software language. An early chapter contains a sample program that incorrectly does away with February 29, 2000. Undoubtedly, this flawed code will be copied and appear in new state-of-the-art operational systems.

- Some systems do not handle the year 1999 correctly since "99" (the last two digits in the year) is sometimes used as a special code in computer systems. For such systems, the Y2K problems will manifest themselves even sooner.

- Although not strictly a Y2K problem, the GPS satellites will reinitialize their dates on August 22, 1999. Software that cannot handle this so-called "GPS week 1024 rollover" will reset their clocks to January 6, 1980. This could cause problems for transit vehicle fleet tracking systems. [<URL:http://www.navcen.uscg.mil/gps/geninfo/y2k/default.htm>]

Addressing the problems

The bottom line is: Don't wait. Do something now to determine what impact, if any, there will be on your software. By doing this as soon as possible, there may still be time to fix the problems before they manifest themselves. This is a case of a hard deadline that cannot be pushed back. Also there is a shortage of people who can deal with the problems, and they will come even more into demand as the time approaches.

For new software:

- Test for Y2K problems as part of your acceptance testing program.
- Include system requirements to handle dates correctly.

For existing software, run special tests to see how the systems perform.

How to test for Y2K problems

Some of the tests that can be run to address the Y2K problem include:

- Set the computer's clock ahead to sometime early in the next century and see what happens.
- Set the clock for the end of the day on December 31, 1999 and watch what happens as the new year "arrives". This "rollover test" is a more rigorous test since some systems that work correctly in either century do not handle the transition between them correctly.
- For database applications, retrieve data that span the turn of the century.
- Conduct analogous tests for the related problems discussed above.

Taking remedial actions if testing fails

Your testing may show that the software works correctly. If so, fine. Or there may be problems with minor impact that you can tolerate. Perhaps all that needs to be done is to shut down your system for a few minutes on either side of midnight. (Depending on the system, it may be shut down anyway during that holiday weekend.) But if you uncover problems during testing, take steps to correct them before they have operational impact. This may even mean having to replace some existing systems before the turn of the century.

For more information on Y2K

A comprehensive Web site on the Year 2000 Problem:
<URL:http://www.it2000.com/>

REFERENCES

E. Bersoff, V. Henderson, and S. Siegel, *Software Configuration Management: An Investment in Product Integrity*, Prentice-Hall, 1980

Booz-Allen, *FHWA Federal-Aid ITS Procurement Regulations and Contracting Options*, August 1997

F. Brooks, "No Silver Bullet: Essence and Accidents of Software Engineering", *Computer*, vol. 20, pp. 10-19, April 1987. (Also reprinted in *The Mythical Man-Month: Anniversary Edition*, Addison-Wesley, 1995)

F. Brooks, *The Mythical Man-Month: Essays on Software Engineering* , Addison-Wesley, 1975. (Also reprinted in *The Mythical Man-Month: Anniversary Edition*, Addison-Wesley, 1995)

J. Cappelletti and P. Gerdes, *Nondevelopmental Item (NDI) and the System Acquisition Process*, MITRE Corporation Technical Report MTR 94W22, 1994

D. Carney and P. Oberndorf, "The Commandments of COTS: Still in Search of the Promised Land", *Crosstalk: The Journal of Defense Software Engineering*, Vol 10 No 5, May 1997

M. Christel and K. Kang, *Issues in Requirements Elicitation*, Software Engineering Institute Technical Report CMU/SEI-92-TR-12, 1992

The Condensed Guide to Software Acquisition Best Practices, 1997. pamphlet available from Software Program Managers Network

J. Costantino *et al.*, "Air Traffic Control: Lessons Learned for Surface Transportation", *ITS Quarterly*, Vol III No. 1, 1995.

Department of Defense, *Guide to Integrated Product and Process Development (IPPD)*, Version 1.0, February 5, 1996. Available at over the Internet in HTML and Word formats at <URL:http://www.acq.osd.mil/te/survey/survmain.html>

M. Evans and J. Marciniak, *Software Quality Assurance and Management*, John Wiley & Sons, 1987

D. Farbman, "Myths That Miss", *Datamation*, pp. 109-112, November 1980

Federal Highway Administration, *Key Findings from the Intelligent Transportation Systems (ITS) Program: What Have We Learned?*, U.S. DOT Publication FHWA-JPO-96-0036, 1996

J. Ferguson *et al., Software Acquisition Capability Maturity Model (SA-CMM℠) Version 1.01*, SEI Technical Report CMU/SEI-96-TR-020, 1996

J. Ferguson and M. DeRiso, *Software Acquisition: A Comparison of DoD and Commercial Practices,* Software Engineering Institute Special Report CMU/SEI-94-SR-9, 1994

D. Garlan and D. Perry, "Introduction to the Special Issue on Software Architecture", *IEEE Transactions on Software Engineering*, Vol 21 No 4, 1995

W. Gibbs, "Software's Chronic Crisis", *Scientific American,* pp. 86-, September 1994

R. Glass, *Modern Programming Practices*, Prentice-Hall, 1982

R. Higuera and Y. Haimes, *Software Risk Management*, Software Engineering Institute Technical Report 96-TR-012, 1996

R. Higuera *et al., Team Risk Management: A New Model for Customer-Supplier Relationships*, SEI Special Report CMU/SEI-94-SR-5, 1994

B. Horowitz, *The Importance of Architecture in DOD Software*, The MITRE Corporation, M91-35, July 1991

W. Humphrey, *Introduction to Software Process Improvement*, SEI Technical Report CMU/SEI-92-TR-7, revised June 1993

W. Humphrey, *Managing the Software Process*, Addison-Wesley, 1989

IEEE, *IEEE Recommended Practice for Software Acquisition,* IEEE Std 1062-1993, 1993

IEEE, *IEEE Recommended Practice for Software Requirements Specifications,* IEEE Std 830-1993, 1993a

IEEE, *IEEE Standard for Software Maintenance,* IEEE Std 1219-1993, 1993b

C. Jones, *Software Project Management: What Works and What Doesn't*, talk at SD '97 Conference, Washington DC, September 29, 1997

J. Marciniak and D. Reifer, *Software Acquisition Management: Managing the Acquisition of Custom Software Systems*, John Wiley & Sons, 1990

S. McConnell, *Rapid Development: Taming Wild Software Schedules*, Microsoft Press, 1996

MITRE Corporation, *Software Reporting Metrics*, MTR-9650 Rev 2, November 1985

M. Paulk, *Key Practices of the Capability Maturity Model, Version 1.1*, SEI Technical Report CMU/SEI-93-TR-025, February 1993

V. Pearce, "Procurement: Hard Work Pays Off", *Traffic Technology International,* pp. 70-, Oct/Nov 1997

T. Pigoski, *Practical Software Maintenance: Best Practices for Managing Your Software Investment,* John Wiley & Sons, 1997

P. Place, P. and K. Kang, K., *Safety-Critical Software: Status Report and Annotated Bibliography,* SEI Technical Report CMU/SEI-93-TR-005

B. Prasad, *Concurrent Engineering Fundamentals (Volume 1: Integrated Product and Process Organization; Volume 2: Integrated Product Development),* Prentice-Hall, 1996

The Program Manager's Guide to Software Acquisition Best Practices, 1997. available from Software Program Managers Network

L. Putnam and W. Myers, *Executive Briefing: Controlling Software Development*, IEEE Computer Society Press, 1996

L. Putnam and W. Myers, *Measures for Excellence: Reliable Software on Time, within Budget*, Yourdon Press, 1992

T. Royer, *Software Testing Management: Life on the Critical Path*, P T R Prentice Hall, 1993

T. Saunders, B. Horowitz, and M. Mleziva, *A New Process for Acquiring Software Architecture*, The MITRE Corporation, M92-B126, 1992

The Standish Group, *Charting Seas of Information Technology*, 1994

STSC (Software Technology Support Center), *Software Configuration Management Technology Report*, <URL: http://stscols.hill.af.mil/cm/REPORT.HTML>, 1994