Integrated Requirements Engineering: A Tutorial

Ian Sommerville, *Lancaster University*

Before developing any system, you must understand what the system is supposed to do and how its use can support the goals of the individuals or business that will pay for that system. This involves understanding the application domain (telecommunications, railways, retail banking, games, and so on); the system's operational constraints; the specific functionality required by the stakeholders (the people who directly or indirectly use the system or the information it provides); and essential system characteristics such as performance, security, and dependability. *Requirements engineering* is the name given to a structured set of activities that help develop this understanding and that document the system specification for the stakeholders and engineers involved in the system development.

This short tutorial introduces the fundamental activities of RE and discusses how it has evolved as part of the software engineering process. However, rather than focus on established RE techniques, I discuss how the changing nature of software engineering has led to new challenges for RE. I then introduce a number of new techniques that help meet these challenges by integrating RE more closely with other systems implementation activities.

This tutorial introduces the fundamental activities of requirements engineering and discusses recent developments that integrate it and system implementation.

The fundamental process

The RE process varies immensely depending on the type of application being developed, the size and culture of the companies involved, and the software acquisition processes used. For large military and aerospace systems, there is normally a formal RE stage in the systems engineering processes and an extensively documented set of system and software requirements. For small companies developing innovative software products, the RE process might consist of brainstorming sessions, and the product "requirements" might simply be a short vision statement of what the software is expected to do.

Whatever the actual process used, some activities are fundamental to all RE processes:

- *Elicitation*. Identify sources of information about the system and discover the requirements from these.
- *Analysis*. Understand the requirements, their overlaps, and their conflicts.
- *Validation*. Go back to the system stake-

0740-7459/05/$20.00 © 2005 IEEE

holders and check if the requirements are what they really need.
- *Negotiation*. Inevitably, stakeholders' views will differ, and proposed requirements might conflict. Try to reconcile conflicting views and generate a consistent set of requirements.
- *Documentation*. Write down the requirements in a way that stakeholders and software developers can understand.
- *Management*. Control the requirements changes that will inevitably arise.

These activities are sometimes presented as if they occur in sequence, where you start with elicitation and end with a documented set of requirements that are then handed over for implementation and managed as changes occur. In reality, whatever the details of the process, RE is always a cyclic activity (see Figure 1). Individual activities repeat as the software requirements are derived, and the iteration continues during system implementation and operation.

The outcome of the RE process is a statement of the requirements (a requirements document) that defines what is to be implemented. The software engineering research community has argued that the more complete and consistent a requirements document, the more likely that the software will be reliable and delivered on time. So, we have a range of techniques—from the use of special-purpose requirements specification languages to structured modeling, to formal mathematical specifications—to help us analyze requirements' completeness and consistency.

Academic research aimed at supporting completeness and consistency hasn't had a major impact on practice. Requirements are usually written in natural language and are often vague descriptions of what's wanted rather than detailed specifications. In situations where requirements change very quickly, this might be the right approach, because the costs of maintaining a detailed specification are unjustified. In other situations, however, failure to define precisely what's required results in endless disputes between the client and the system developer.

RE's evolution

The need for RE became obvious in the last century as the systems engineering discipline developed. The RAND Corporation, founded in 1946, introduced the notion of systems analysis, which has evolved into RE. Understanding a problem and specifying its system requirements became an inherent part of the development process for complex military and aerospace systems.

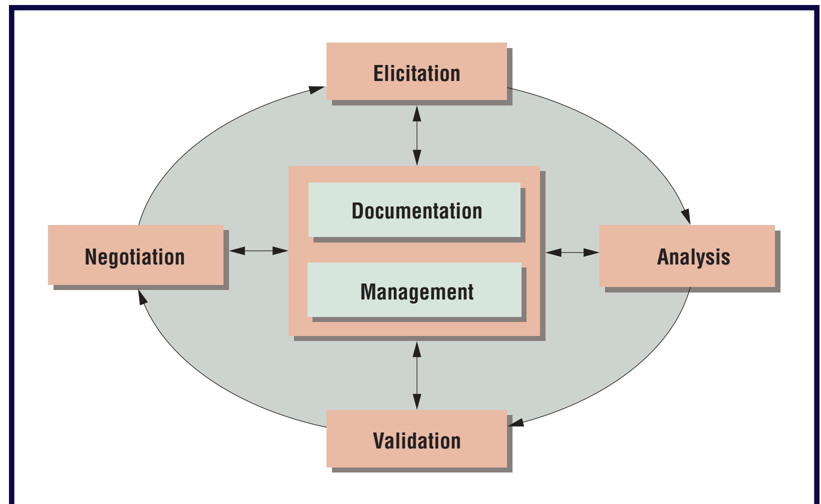


Figure 1. The requirements engineering activity cycle.

The lifecycle model used in systems engineering was the predominant influence on the development of the waterfall model of software engineering, first proposed by Winston Royce in 1970. In this model, the process of understanding and documenting system requirements is the first stage in the software engineering process. This led to an assumption that RE was something that you did before you started software development and that, once discovered, the software requirements would not change significantly during the development process. It also led to the assumption that RE should be separated from system design. The system requirements should define what the system should do; the design should define how the system should implement the requirements.

Work in the 1970s on requirements focused on developing requirements statement languages, such as Dan Teichrow's PSL/PSA (Problem Statement Language/Problem Statement Analyzer), and methods of structured analysis.[1,2] Object-oriented modeling was developed in the 1980s, with Ivar Jacobson's use cases being a key element now embodied in the Unified Modeling Language.[3,4] The IEEE standard on requirements documents was introduced and refined,[5] and the 1990s saw much academic research on viewpoint-oriented approaches to elicitation and analysis,[6]

Requirements engineers shouldn't be influenced by design considerations when setting out a system's requirements.

formal mathematical methods,[7] goal-oriented approaches,[8] and RE process improvement.[9]

We now know that the initial assumptions that underpinned much RE research and practice were unrealistic. Requirements change is inevitable, because the business environment in which the software is used continually changes—new competitors with new products emerge, and businesses reorganize, restructure, and react to new opportunities. Furthermore, for large systems, the problem being tackled is usually so complex that no one can understand it completely before starting system development. During system development and operational use, your stakeholders continue to gain new insights into the problem, leading to changes in the requirements.

Separating requirements and design means that requirements engineers shouldn't be influenced by design considerations when setting out a system's requirements. Moreover, the requirements shouldn't limit designers' freedom in deciding how to implement the system. In one of the first books on RE,[10] Alan Davis explains why this is desirable: designers often know more about technologies and implementation techniques than requirements engineers, and the requirements shouldn't stop them from using the best available approach.

However, Davis also recognizes that this ideal is impossible to achieve. What one person might think of as a specification, another thinks of as a design. Fundamentally, the processes of understanding the problem, specifying the requirements, and designing the system aren't discrete stages. They are all part of the general process of developing a deeper understanding of the business, the capabilities and structure of the system being developed, and its operating environment.

RE for the 21st century

The 20th-century view of RE as something you do before system development, and the software requirements document as a complete specification of the software to be implemented, is no longer valid for many types of system. New approaches to software development and the need for businesses to respond quickly to new opportunities and challenges mean that we must rethink RE's role in software development.

Four key change drivers have forced this rethink:

- *New approaches to systems development—in particular, construction by configuration.* The dominant approach for many types of system is now based on reuse, where existing systems and components are configured to create new systems. The software requirements depend on the existing system capabilities and not just on what stakeholders believe they need. The "Construction by Configuration" sidebar discusses this important approach to systems development in more detail.
- *The need for rapid software delivery.* Businesses now operate in an environment that's changing incredibly quickly. New products appear and disappear, regulations change, businesses merge and restructure, competitors change strategy. New software must be rapidly conceived, implemented, and delivered. There isn't time for a prolonged RE process. Development gets going as soon as a vision for the software is available, and the requirements emerge and are clarified during the development process.
- *The increasing rate of requirements change.* This is an inevitable consequence of rapid delivery. If you don't have time to understand your requirements in detail, you'll inevitably make mistakes and have to change the requirements to fix these problems. More significantly, perhaps, the changing business environment means that new requirements might emerge and existing requirements might change every week or sometimes even every day.
- *The need for improved ROI on software assets.* Companies have enormous investments in their software and, understandably, want to get as much return as possible on that investment. So, when they need new systems, there's pressure to reuse existing software wherever possible. This introduces the need for interoperability requirements that specify how the new and the existing software should work together.

The emerging vision of Web Service architectures where programs can dynamically search for available services and bind to them at runtime poses further challenges for RE. In the Web Services model, a system's components are services, defined by their interfaces. These might be offered by external providers, and many providers might offer the same service,

such as an ordering service for PCs. In principle, an executing program can use services from different providers at different times without user intervention.

Instead of thinking about requirements in terms of system functionality or features, we'll have to think about systems in terms of services provided and used. We'll also need to find ways to embed the requirements for the services that a program itself needs so that these services can be dynamically discovered and used.

Integrating RE with system development

To address the system development challenges of the 21st century, we must integrate the processes of RE and system implementation. The artificial separation of these activities leads to a situation where customers don't realize how much time and effort is required to deliver their requirements, and where suppliers can't deliver the best value to customers using their specialist knowledge and existing software.

RE researchers and practitioners increasingly recognize this. I don't have space to discuss all the recent developments, so I'll focus here on three particularly important areas of work:

- Concurrent RE
- Supporting trade-offs between requirements and design
- RE and commercial off-the-shelf (COTS) acquisition

Concurrent RE

Concurrent engineering is an approach to product development where, instead of a sequential process of specification, design, manufacture, and so on, engineering process activities are carried out concurrently with extensive feedback and iteration among the different teams involved. In software engineering, agile development methods such as Extreme Programming (XP)[11] embody a concurrent approach that integrates the processes of RE, design, and development.

Concurrent RE means that the starting point for development is an outline of the software. RE activities such as elicitation and validation are carried out concurrently, and the RE process is concurrent with other system development processes. The system is developed and delivered in increments, with each increment incorporating a useful subset of the overall system functionality.

Concurrent RE offers several benefits:

- *Lower process overheads*. You'll spend less time analyzing and documenting a large body of requirements.
- *Early identification and implementation of value-delivering requirements*. Value-delivering requirements are the most critical ones for the customer's business—they might allow new business processes to be created or existing processes to be more effective. A customer representative identifies the requirements that deliver the most value and negotiates their implementation with the development team.
- *Responsiveness to requirements change*. Because you identify and document your requirements iteratively, the overhead of accommodating requirements change is

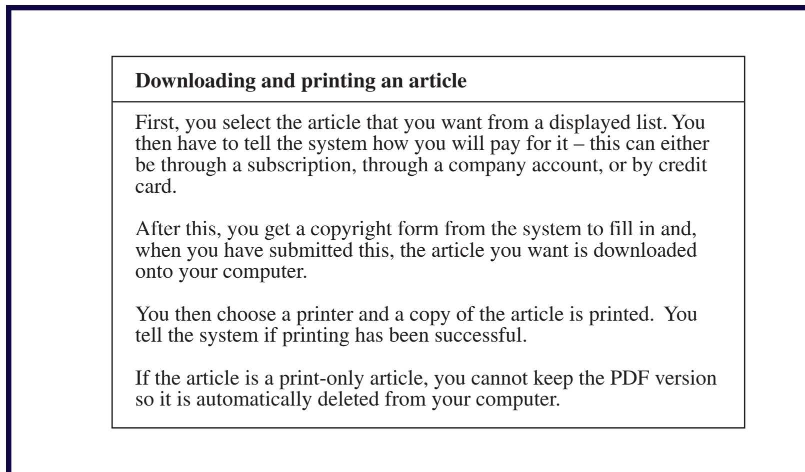
Construction by Configuration

The central development in software engineering over the past 15 years has been the divergence of approaches to software development for different types of system. Before 1990, irrespective of application domain, most systems were designed and programmed in a generic or application-specific programming language. Business systems were developed in Cobol, operating systems in C, many embedded systems in assembly language, and so on. There was a shared development paradigm of specify, design, implement, test.

However, the past 10 years have seen remarkable changes. Pressured by the need to cope with rapid change, the Y2K problem, and increasingly complex distributed environments, businesses changed from building all their software from scratch to an approach based on software reuse. For business systems, the dominant development paradigm is no longer based around programming but around reuse. Systems are developed by assembling and integrating COTS, legacy systems, handwritten code, configured enterprise resource planning (ERP) systems, and other software.

Sometimes this still involves conventional programming but with extensive component reuse. At other times, it means configuring an off-the-shelf system to support a business process; in other cases, it might mean moving toward an ERP-based solution where the software development involves constructing business rules and business process descriptions. In all these cases, however, the freedom of the system stakeholders and designers is limited by what is available.

Of course, other system classes, such as control systems and middleware, are still developed according to the traditional paradigm. While this approach will continue for some classes of system, I believe that software construction by configuration rather than programming will be extended to other areas such as systems software development and embedded systems.



Downloading and printing an article

First, you select the article that you want from a displayed list. You then have to tell the system how you will pay for it – this can either be through a subscription, through a company account, or by credit card.

After this, you get a copyright form from the system to fill in and, when you have submitted this, the article you want is downloaded onto your computer.

You then choose a printer and a copy of the article is printed. You tell the system if printing has been successful.

If the article is a print-only article, you cannot keep the PDF version so it is automatically deleted from your computer.

Figure 2. A story card describing a usage scenario of a digital article library.

relatively low. It might simply involve reprioritizing the development schedule to incorporate a requirements change in the next system increment or to implement a newly emerged requirement.

For example, in XP, customer representatives are key members of the development team. Rapid iteration is the norm, with new releases of the system delivered to the customer at frequent intervals. The customer representative's role on the development team is to identify the requirements that, at any point in the development cycle, deliver the best value and then to negotiate the implementation of these requirements with the developers.

The particular approach used in XP is based on stories or scenarios, with each scenario written on a card. The scenarios are written in user terms and illustrate some required user functionality. For example, Figure 2 shows a simple scenario that might be used in implementing a library system that provides access to paid-for copyright articles from external providers.

Developers analyze the scenario, break it down into tasks, and estimate the effort required to implement that scenario. Given this information about the costs of implementing each scenario, the customer representative then decides which requirements should have priority for inclusion in the next system release.

Concurrent RE isn't suitable for all types of system. If you have to implement a critical system where careful analysis of the interactions and dependencies between requirements is essential, you need a complete and detailed requirements document before implementation starts. However, many companies don't believe that the benefits of creating a complete requirements document for their business systems outweigh the time and effort costs required to create such a document.

Some RE researchers and practitioners are concerned that XP's informal approach makes requirements analysis almost impossible. Because XP requirements aren't formally documented, developers effectively discard them after implementation, and they never deliver a complete system specification to the customer. I also have some doubts about the impermanence of XP requirements, but I believe the concurrent approach points the way forward for using RE in volatile business systems.

Supporting requirements/design trade-offs

A major difficulty in RE is that customers don't have the knowledge to estimate the difficulty and associated costs of implementing a requirement. They might suggest an apparently simple requirement that has major cost repercussions, or they might unnecessarily limit their requirements because they don't know what's already available in off-the-shelf products. Barry Boehm and his colleagues give examples of situations in which such problems arose:[12]

- A customer asked for a natural language interface to a relatively simple query system. This resulted in major additional costs and ultimate cancellation of the project.
- In a project to digitize corporate records using scanning and optical character recognition, the customer failed to recognize that the OCR technology didn't work well with tables, charts, and graphs.

Boehm argues that to deliver systems rapidly that meet customer needs, a key challenge is to reconcile customer expectations with developer capabilities. He developed an approach to RE called the WinWin approach,[13] in which negotiation between customers and software suppliers is central. The aim is to ensure that all stakeholders identify win conditions that can be satisfied. The WinWin approach has now been incorporated in a more general approach to RE called MBASE (*m*odel-*b*ased *a*rchitecting and *s*oftware *e*ngineering), which integrates RE and systems design.

One of the most important aspects of the

MBASE approach is its support for managing expectations. To help with communication between customers and developers, Boehm and his team have identified what he calls *simplifiers* and *complicators*: things that make life easier and things that make life harder, for both developers and customers. These are organized and classified using domain-specific headings and are used to help customers understand the problems developers face, and vice versa.

For example, for COTS package extension, developer-side simplifiers are

- Clean, well-defined APIs
- A single COTS package
- Simple mappings of interface inputs and outputs

and developer-side complicators include

- Dynamic APIs
- Natural language processing
- Multiple, incompatible COTS packages
- Volatile COTS packages
- Complex exception handling

You can analyze simplifiers and complicators from a customer perspective to assess the associated customer risks and benefits. For example, in an information retrieval system for a digital library, an obvious simplifier is to use a standard query language. However, the risk from a librarian perspective is that this might not be as effective as a specially designed query system because users must know what they're looking for before they start searching.

Simplifiers and complicators are a simple idea that you can easily incorporate in any RE process. They make clear to customers that different requirements choices have significant implications for the system's design and implementation, and they provide a focus for making requirements and design decisions that reduce the risks for both software customers and suppliers.

RE and COTS acquisition

COTS systems are now available in most domains, so you can configure and adapt generic products to different operational settings. You can develop applications by acquiring new COTS systems and configuring them to interoperate with existing systems. Although you shouldn't underestimate the difficulties of this approach, when it's successful it leads to lower development costs and accelerated system deployment.

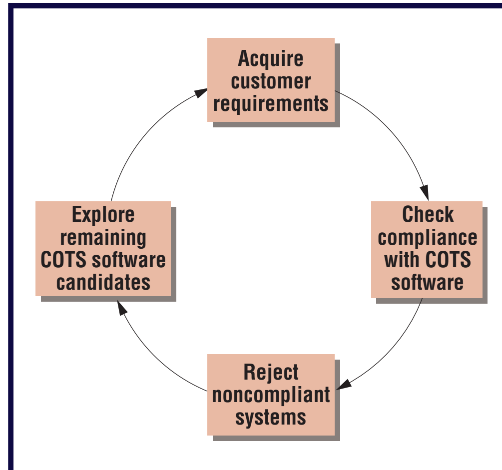


Figure 3. The cyclical PORE (procurement-oriented requirements engineering) process can help you select the right COTS product.

From an RE perspective, the traditional "requirements first" approach poses real problems. If you develop a detailed set of requirements, you'll probably find that no COTS product meets your requirements. When selecting COTS software, you need to be flexible; identify a set of critical requirements and choose products that meet them. Then you can adapt and extend your other requirements according to the selected systems' capabilities.

Two areas of RE research are particularly important for COTS acquisition. The first deals with COTS product selection: When many different products are available, how do you pick the one that's best for your requirements? The second area is COTS interoperability: How should you specify your requirements so that your COTS software will work with each other and with your existing operational systems?

Selecting a COTS product

Neil Maiden and his colleagues have developed a systematic approach to COTS product selection called PORE (procurement-oriented requirements engineering).[14] This approach depends on eliciting key requirements from stakeholders, then using these to identify a candidate set of COTS software that meets or partially meets these requirements. The candidate systems' features and capabilities help system stakeholders identify further requirements that they can then use to refine their selection of COTS products. Figure 3 shows this

Over the next few years, integrated RE will become the preferred mode of development for most types of system.

cyclic process, in which the final result is selection of the most suitable COTS system.

In the PORE process, you select candidates through a three-stage process, in which each stage develops the system requirements in more detail and reduces the size of the COTS candidates set. The stages are as follows:

1. Use publicly available information to select candidate COTS software. Identify critical requirements that you can employ to discriminate between products using marketing literature and data sheets. Requirements at this stage might include cost requirements, requirements for general capabilities, and interoperability requirements.
2. Use product demonstrations to narrow the set of possible systems and to stimulate the elicitation of new requirements. The product demonstrations should show how each system meets the initial requirements and demonstrate each system's overall capabilities. This gives you information on how to refine your initial requirements and generate new requirements that let you pick the systems that go forward to the next stage.
3. Use hands-on product evaluation to further refine your choice of system and your system requirements. By this stage, you effectively have a prototype system for experiment, and you can use this with stakeholders to drive the requirements elicitation process. Because they can see what features and capabilities each system offers, stakeholders can prioritize their requirements accordingly. Once you've narrowed the choice to two or three systems, you might perform more extensive trials to assess the emergent properties of candidates such as performance, reliability, and so on.

The PORE method provides active guidance for each of these stages. It's been used successfully to procure different types of systems, including requirements management systems and telecommunications systems for securities trading.

Interoperability requirements

All companies now use a range of different software systems, and a critical business requirement in many situations is that new software systems should interoperate with those that are already in place. For example, a desktop e-procurement system might have to work with an existing supplier database to provide supplier information and with an existing ordering and invoicing system to manage orders, payments, and deliveries.

In the RE research community, there's been a prevailing view that we can achieve interoperability by using open interfaces and standards, and that simply specifying openness as a requirement solves the problem. Experience has shown this isn't true. Boehm and Chris Abts have reported on some of the practical difficulties in integrating COTS systems.[15]

Soren Lausen, in a recent paper,[16] discusses the problems of specifying interoperability requirements and choosing systems that meet these requirements. He introduces a new type of requirement called an *open-target requirement*, which tries not to exclude any possible technical solution. It defines customer expectations and, critically, requires potential suppliers to explain how they'll meet those expectations.

He proposes five guidelines for interoperability requirements specification:

1. Use open-target requirements and develop a structured framework for evaluating and scoring suppliers' responses to these requirements.
2. Express the interoperability requirement as a user request rather than as a technical requirement.
3. Be flexible in the degree of integration that you can live with.
4. Think about product evolution, and write requirements that ensure that someone apart from the original supplier can extend the product.
5. Write a trial period into the system contract to demonstrate that the supplier can handle the project's high-risk areas.

To illustrate open-target requirements, consider a sample e-procurement system that must interoperate with an existing supplier database:

R1: *The e-procurement system shall not maintain supplier addresses separately but shall retrieve supplier addresses from the existing supplier database.*

Expressing the requirement in this way is unduly restrictive and might exclude many possible e-procurement COTS solutions that include their own data management system. An alternative specification would be to focus on the consistency of the information:

R1a: *The supplier addresses displayed by the e-procurement system shall be consistent with the customer addresses in the supplier database.*

This is a more open requirement but again might be unduly restrictive. It excludes the possibility, for example, of simply displaying the supplier name and supplier reference and then adding the address from the supplier database when the order is actually generated.

R1b: *The e-procurement system shall share supplier data with the current supplier database. Supplier addresses on orders and invoices must be consistent. The system provider shall explain the proposed sharing mechanism.*

This open-target requirement is more flexible; it states why the requirement is included and explicitly requires an explanation from the system provider of how this will be ensured. Such requirements are a way of ensuring interoperability without overprescribing and excluding systems that might viably meet other important system requirements.

Business demands for faster delivery of systems that cost less and are more responsive to changing requirements mean that the traditional "requirements first" approach to software engineering has to evolve. Requirements engineering has to be more tightly integrated with system implementation to take advantage of reuse and to let systems evolve to reflect changing requirements. Business system developers have already embraced these changes. I predict that over the next few years, integrated RE will become the preferred mode of development for most types of system.

However, we should not underestimate the real barriers to this integration that will slow its introduction. Many system acquisition processes require a detailed requirements document as the basis of the contract between client and supplier. Outsourced development also relies on the remote development team working from a detailed specification. Integrated RE will require acquisition processes to evolve to reflect the fact that close cooperation between clients and suppliers is the best hope we have for more effective software engineering.

About the Author

Ian Sommerville is a professor of software engineering in the Computing Department at Lancaster University. His research interests include requirements engineering, system dependability, service-oriented software engineering, and social informatics. He received his PhD in computer science from St. Andrews University, Scotland. He is a Fellow of the British Computer Society and IEE and a member of the IEEE Computer Society and ACM. Contact him at the Computing Dept., Infolab21, Lancaster Univ., Lancaster, LA1 4WA, UK; is@comp.lancs.ac.uk.

References

1. D. Teichrow and E.A. Hershey, "PSL/PSA: A Computer Aided Technique for Structured Documentation and Analysis of Information Processing Systems," *IEEE Trans. Software Eng.*, vol. SE-3, no. 1, 1977, pp. 41–49.
2. T. DeMaro and P.J. Plauger, *Structured Analysis and System Specification*, Prentice Hall, 1979.
3. I. Jacobson, *Object-Oriented Software Engineering: A Use-Case Driven Approach*, Addison-Wesley, 1992.
4. G. Booch, J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language User Guide*, Addison-Wesley, 1998.
5. IEEE Std IEEE-Std-830-1998, *IEEE Recommended Practice for Software Requirements Specification*, IEEE CS Press, 1998.
6. I. Sommerville and P. Sawyer, "Viewpoints: Principles, Problems and a Practical Approach to Requirements Engineering," *Annals of Software Eng.*, vol. 3, 1997, pp. 101–130.
7. A. Hall, "Using Formal Methods to Develop an ATC Information System," *IEEE Software*, vol. 13, no. 2, 1996, pp. 66–76.
8. A. Van Lamsweerde, "Goal-Oriented Requirements Engineering: A Guided Tour," *Proc. 5th Int'l IEEE Requirements Eng. Conf.*, IEEE CS Press, 2001, p. 249.
9. I. Sommerville and P. Sawyer, *Requirements Engineering: A Good Practice Guide*, John Wiley & Sons, 2000.
10. A. Davis, *Software Requirements: Analysis and Specification*, Prentice Hall, 1990.
11. K. Beck, *Extreme Programming Explained: Embrace Change*, Addison-Wesley, 2000.
12. B. Boehm et al., "Requirements Engineering, Expectations Management and the Two Cultures," *Proc. 7th Int'l Symp. Requirements Eng.*, IEEE CS Press, 1999, pp. 14–22.
13. B. Boehm et al., "Using the WinWin Spiral Model: A Case Study," *Computer*, vol. 31, no. 7, 1998, pp. 33–44.
14: N.A. Maiden and C. Ncube, "Acquiring COTS Software Selection Requirements," *IEEE Software*, vol. 15, no. 2, 1998, pp. 46–56.
15. B. Boehm and C. Abts, "COTS Integration: Plug and Pray?" *Computer*, vol. 32, no. 1, 1999, pp. 135–138.
16. S. Lausen, "COTS Tenders and Integration Requirements." *Proc. 12th IEEE Int'l Requirements Eng. Conf.*, IEEE CS Press, 2004, pp. 166–175.

For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/publications/dlib.