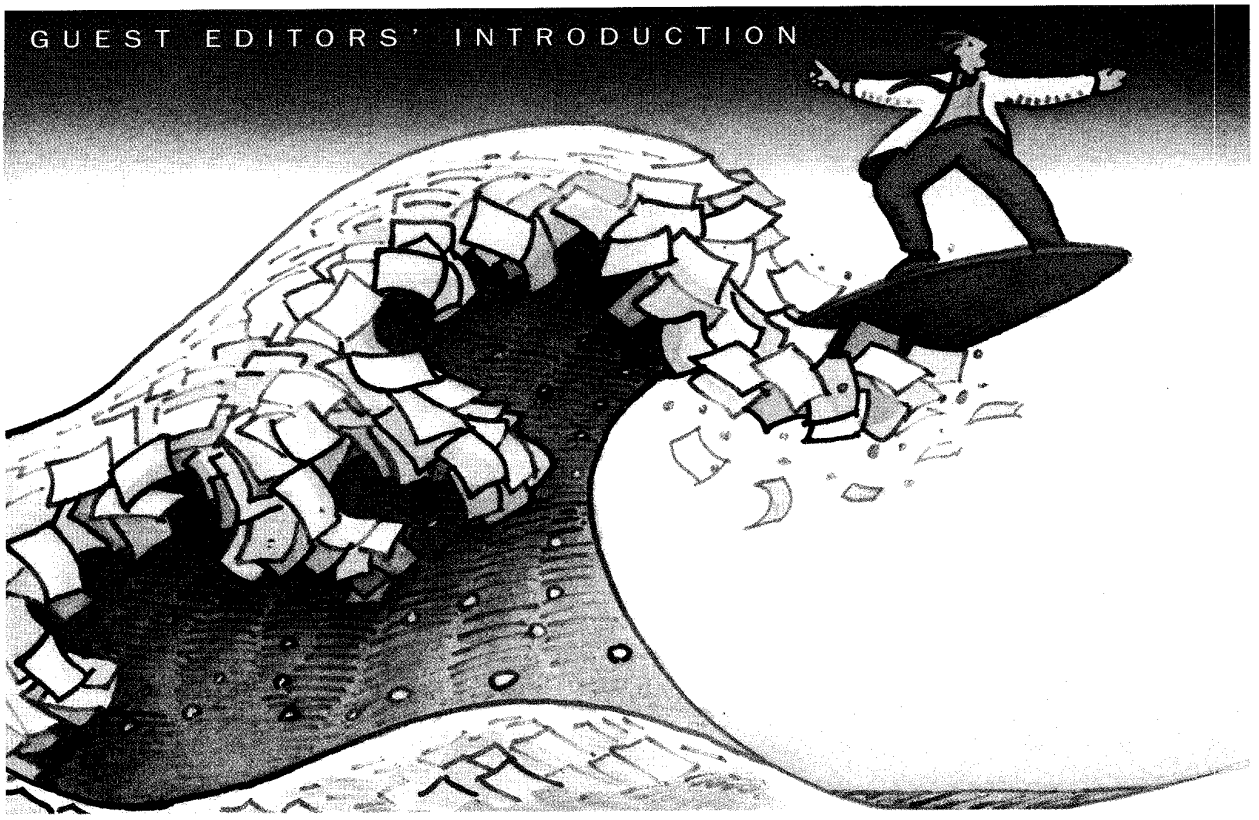

REQUIREMENTS ENGINEERING: THE EMERGING WISDOM

JAWED SIDDIQI, *Sheffield Hallam University, and* M. CHANDRA SHEKARAN, *Microsoft*

Developments in requirements engineering, as in system development, have come in waves. The next wave of requirements techniques and tools will account for the problem and development context, accommodate incompleteness, and recognize the evolutionary nature of requirements engineering.

The field traditionally known as system analysis was first applied to information systems, and so had an organizational and application orientation. The field of requirements engineering seeks to incorporate an engineering orientation into systems analysis.

The most widely known, and perhaps the most significant, products of this engineering orientation are the various development methods and their associated automation support tools. Unfortunately, many of these prescriptive methods pay little or no attention to how context influences decomposition and evolution. Practitioners, who are used to focusing on context, find these methods to be inadequate. So the gap between practice and research is still very wide. The conventional wisdom about requirements engineering is rapidly evolving, however, and the latest research is taking context into account.

Developments in requirements engineering are following trends in system development: In the first wave of system development, the focus was on writing code. Small and large system development alike were viewed as a single activity, not an organized process with several stages. The next wave saw the introduction of the development life cycle, of which requirements analysis was the first phase. Next came the adoption of evolutionary development models and the acknowledgment, at least from practitioners, that implementation may often proceed from incomplete requirements. The evolution of

0740-7459/96/$05.00 © 1996 IEEE

ARTICLE SUMMARIES: REQUIREMENTS ENGINEERING

Identifying Quality-Requirement Conflicts, pp. 25-35.

Barry Boehm and Hoh In

Despite well-specified functional and interface requirements, many software projects have failed because they had a poor set of quality-attribute requirements. To find the right balance of quality-attribute requirements, you must identify the conflicts among desired quality attributes and work out a balance of attribute satisfaction.

We have developed The Quality Attribute Risk and Conflict Consultant, a knowledge-based tool that can be used early in the system life cycle to identify potential conflicts. QARCC operates in the context of the WinWin system, a groupware support system that determines software and system requirements as negotiated win conditions. This article summarizes our experiences developing the QARCC-1 prototype using an early version of WinWin, and our integration of the resulting improvements into QARCC-2.

Technology to Manage Multiple Requirements Perspectives, pp. 37-48.

Hans W. Nissen, Manfred A. Jeusfeld, Matthias Jarke, Georg V. Zemanek, and Harald Huber

Stakeholder conflicts can be productive in requirements engineering. A requirements-engineering project should ensure that crucial requirements are captured from at least two perspectives, preferably in a notation of the customer's choosing. Capturing, monitoring, and resolving multiple perspectives is difficult and time-consuming when done by hand. Our experience with ConceptBase, a meta-data-management system, shows that a simple but customizable metamodeling approach, combined with an advanced query facility, produces higher quality requirements documents in less time.

Our experience shows that conceptual metamodeling technology can be a valuable complement to informal teamwork methods of business analysis and requirements engineering. In particular, the use of representations and cross-perspective analysis can help identify a wide variety of conflicts and, perhaps more important, monitor them.

An Object-Oriented Tool for Tracing Requirements, pp. 52-64.

Francisco A.C. Pinheiro and Joseph A. Goguen

Tracing requirements helps verify system features against the requirements specification, identify error sources, and — most importantly — manage change. We describe a tool called TOOR (Traceability of Object-Oriented Requirements), based on Joseph Goguen's broad view of requirements, which suggests that not only technical but also social factors play a significant role in software development. Of course, the requirements themselves are not necessarily object-oriented. Our tool's name is derived more from its use for object-oriented development and its object-oriented implementation, which allows the definition of classes and subclasses of objects and relationships among objects.

Requirements traceability is our primary topic, but the scope of the tool reflects our view that requirements issues are pervasive and occur throughout the life cycle. Therefore, requirements traceability should be available at all times, and traceability of any desired artifacts should be supported.

requirements engineering has benefited from both the information-systems and software-engineering paradigms. Today a variety of approaches judiciously mix techniques borrowed from both strands. Definitive claims about the superiority of one paradigm over the other are not only premature but of little practical use. Such debates distract us from addressing important fundamental questions that include:

- What activities should be included in requirements engineering?
- What constitutes a requirement?
- What issues of practice need further attention?

REQUIREMENTS-ENGINEERING ACTIVITIES

Most software-engineering professionals believe that the requirements phase has its own life cycle. The phases of it have been given different labels. In the '80s Herb Krasner identified five phases: need identification and problem analysis; requirements determination; requirements specification; requirements fulfillment; and requirements change management.[1] More recently, Matthias Jarke and Klaus Pohl proposed a three-phase cycle: elicitation, expression, and validation.[2]

However the phases may be sliced, it has long been realized that requirements evolve through a series of iterations. The Inquiry Cycle Model proposed by Colin Potts and colleagues takes this view.[3] It integrates three phases — documentation, discussion, and evolution — and has stakeholders use scenarios to identify and validate requirements. The validation is accomplished by stakeholders challenging proposed requirements with the intent of obtaining a clearer understanding of the justifications for the requirements' existence. On the basis of this validation, stakeholders can freeze or change a requirement in the final phase.

The software-engineering community had focused its efforts on the problem-analysis phase, which is what many decomposition methods address. In general terms, these methods are designed to help the analyst define the range of all possible solutions. More specifically, according to Alan Davis, the problem-analysis phase encompasses learning about the problem, understanding the needs of the potential users, discovering who the user really is, and understanding all the constraints on the solution.[4] The outcome — the requirements-specification document — is assumed to be a complete description of the product's external behavior.

Lately, the software-engineering community has extended this decomposition paradigm to propose that systems can be built using a standard repertoire of components. Jarke and Pohl have suggested, for example, that one-shot requirements-engineering projects may be replaced by a "requirements-engineering practice," which puts together standard components in innovative fashions rather than continuing the practice of reinventing the components themselves.[2]

As we've said, the biggest drawback of the reductionist view of partitioning things into smaller parts is that context will influence the decomposition.

Indeed, for Jarke and Pohl the juxtaposition of vision and context is at the heart of managing requirements. They define requirements engineering as a process of establishing visions in context and proceed to define context in a broader view than is typical for an information-systems perspective. Jarke and Pohl partition context into three worlds: subject, usage, and system. The subject represents a part of the outside world in which the system — represented by some structured description — exists to serve some individual or organizational purpose or usage.

WHAT CONSTITUTES A REQUIREMENT?

The oldest and perhaps most widely shared piece of conventional wisdom is that requirements constitute a complete statement of what the system will do without referring to how it will do it. The resiliency of this view is indeed surprising since researchers have long argued against this simple distinction.[5]

Clearly, requirements and design are interdependent, as practitioners surely realize. Perhaps the continuing prevalence of the "what vs. how" distinction is due to the well-meaning desire on the part of requirements engineers to avoid overconstraining implementers. Other reasons for the persistence of this debate are explored elsewhere.[6]

Another common distinction is the separation of functional (or behavioral) and nonfunctional requirements. Again, practitioners have found that, for many applications, this distinction is not clear. Some requirements that may appear to be nonfunctional at first become, in due course, functional. In the past, most researchers have focused on functional requirements. The article by Barry Boehm and Hoh In in this issue reflects the more recent trend to direct attention to nonfunctional requirements issues. For some time now, the software community has realized the need to broaden its view of requirements to consider the context within which the system will function. Alex Borgida, Sol Greenspan, and John Mylopoulos' work on the use of conceptual modeling as a basis for requirements engineering was a major signpost in directing researchers to this perspective.[7] The article by Hans Nissen and his colleagues in this issue reports on recent experiences in applying conceptual-modeling techniques.

More recently, Michael Jackson has advanced another way to look at context.[8] Jackson faults current software-development methods for focusing on the characteristics and structure of the solution rather than the problem. Software, according to Jackson, is the description of some desired machine,

The oldest, most widely held piece of conventional wisdom is that requirements are complete.

and its development involves the construction of that machine. Requirements are about purposes, and the purpose of a machine is found outside the machine itself, in the problem context. He has, therefore, argued for a shift towards a problem-oriented approach that seeks to distinguish different characteristics and structures in the application domain. Adopting this problem-oriented approach means that the requirements for a system can simply be viewed as relationships among phenomena in the domain and a specification is a restricted kind of requirement; it is restricted because it must be expressed in terms of domain phenomena that are shared with the machine to be constructed.

This characterization of requirements and specification is indeed very general. However, Jackson animates it into a method by devising a general problem frame, analogous to those proposed by the mathematician, George Polya. In Polya's terms, software development is a three-part problem: the domain, the requirements, and the machine. Jackson argues that for any method to be powerful it must exploit the specific features of the problem and because problem features vary widely, we need a repertoire of methods each suitable for a certain class of problems. This view puts the knowledge of both the domain expert and the analyst at the heart of requirements engineering.

Joseph Goguen shares Jackson's broad view of requirements.[9] But, while Jackson's distinctive contribution is primarily concerned with how requirements are represented, Goguen's novel contribution centers on how requirements should be produced. Goguen argues that requirements are information, and all information is *situated* and it is the situations that determine the meaning of requirements. Taking context (or situations) into account means paying attention to both social and technical factors. Focusing on technical factors alone fails to uncover elements like tacit knowledge, which cannot be articulated. Therefore, an effective strategy for requirements engineering has to attempt to reconcile both the technical, context insensitive, and the social, contextually situated factors.

For Goguen, requirements are not things "out there" flying about like butterflies. Nor is the job of the analyst to find some suitable net to capture them. Requirements emerge from the social interactions between the system users and the analyst. This goes beyond taking multiple viewpoints of the different stakeholders and attempting to reconcile them because it does not attempt, *a priori*, to construct some abstract representation of the system. Current methods of eliciting tacit information, such as questionnaires, interviews, introspection, and focus groups are inadequate, as Goguen points out.

Instead, he advocates "ethnomethodology." In this approach, the analyst gathers information in naturally occurring situations where the participants are engaged in ordinary, everyday activities. Furthermore, the analyst does not impose so-called "objective," preconceived categories to explain what is occurring. Instead, the analyst uses the categories the participants themselves implicitly use to communicate.

Trends in the last decade have made the requirements-as-contract model irrelevant to most developers today.

REALITY CHECK

Descending from the lofty considerations of the fundamental nature of requirements, here we recommend some practical issues that require greater attention. A careful examination of these issues actually reveals a considerable level of compatibility with the perspective shifts urged by Jackson and Goguen.

Most requirements-engineering work to date has been driven by organizations concerned with the procurement of large, one-of-a kind systems. In this context, requirements engineering is often used as a contractual exercise in which the customer and the developer organizations work to reach agreement on a precise, unambiguous statement of what the developer would build.

Trends in the last decade — system downsizing, shorter product cycles, the increasing emphasis on building reusable components and software architectural families, and the use of off-the-shelf or outsourced software — have significantly reduced the percentage of systems that fit this profile. The requirements-as-contract model is irrelevant to most software developers today.

Other issues are more important:

◆ *Supporting market-driven inventors.* The bulk of the software developed today is based on market-driven criteria. The requirements of market-driven software are typically not elicited from a customer but rather are created by observing problems in specific domains and inventing solutions. Here requirements engineering is often done after a basic solution has been outlined and involves product planning and market analysis. The paramount considerations are issues such as available market window, product sizing, feature sets, toolkit versus vertical application, and product fit with the development organization's overall product strategy. Classical requirements engineering offers very little support for these problems. Only recently have researchers begun to acknowledge their existence.[10]

◆ *Prioritizing requirements.* Competitive forces have reduced time to market, causing development organizations to speed development by deliberately limiting the scope of each release. This forces developers to distinguish between desirable and necessary (and indeed, between levels of needed) features of an envisioned system. Further, modifying certain noncritical requirements may enable an envisioned system to be realized using one or more off-the-shelf components. Yet there has been little progress to date on mechanisms for prioritizing requirements and making choices on which of those among a set of optional requirements will be satisfied by a given system release.

◆ *Coping with incompleteness.* One impetus for the switch in the '80s to the evolutionary development model was the recognition that it was virtually impossible to make all the correct requirements and implementation decisions the first time around. Yet most requirements research agendas continue to emphasize the importance of ensuring completeness (in the sense of having no missing parts) in requirements specifications. However, incompleteness in requirements specifications is a simple reality for many practitioners. Some may even claim that completeness in real-world requirements specifications is a utopian state about as achievable as getting it right the first time! Goguen echoes this view in his criticism of current methods for their prescriptiveness and their insistence on the existence of a complete specification.[9] The real challenge is how to decide what kinds and levels of incompleteness the developer can live with. To this end we need techniques and tools to help determine appropriate stopping conditions in the pursuit of complete requirements specifications — enabling such clarification to be postponed to a later development stage (or a later "spiral" in the system's evolution).

◆ *Integrating design artifacts.* Developers need faster ways to conveniently express the problem to be solved and the known constraints on the solution. Often, getting to this fast outweighs the risk of overconstraining design. As Shekaran and others have observed elsewhere, requirements engineering becomes more of a design and integration exercise in this context.[11] We need "wide-spectrum" requirements techniques that can capture and manipulate design-level artifacts, such as off-the-shelf components. To date, there have been very few concrete results in providing support for the task of evaluating alternative strategies for satisfying requirements (a "design-like" task). However, the burgeoning interest and activity in requirements tracing may offer some solutions in the near future. In this issue, Pinheiro and Goguen offer an early look at tool support that can be provided for tracing requirements.

◆ *Making requirements methods and tools more accessible.* Today, many practitioners use general tools like word processors, hypertext links, and spreadsheets for many requirements engi-

neering tasks. Given the wide variety of contexts in which requirements are determined and systems are built, researchers may be well-advised to focus on specific requirements subproblems (for example, tracking and managing software priorities) and consider building automation support in the form of add-ons to existing general-purpose tools. Less accessible to practitioners are methods that prescribe a major overhaul of an organization's requirements process and the use of large, monolithic tools.

We believe the key mission for the requirements-engineering community is to continually narrow the ever-growing gap between research and practice. To that end, we close with some apt advice from the poet Stevie Smith, cited by Peter Checkland:[12]

> *It is very nice to have feet on the ground if you are a feet-on-the-ground person. I have nothing against feet-on-the-ground people. And its very nice to have feet off the ground if you are a feet-off-the-ground person. I have nothing against feet-off-the-ground people. They are all aspects of truth or motes in the coloured rays that come from coloured glass and stains the white rays of eternity.* ◆

Chandra Shekaran is program manager at Microsoft Corp. of a telecommunications group whose charter is the creation of shrink-wrapped telephony software products. He has been the prinicipal member of various software-engineering groups at GTE Laboratories, where he led a variety of research and development efforts in requirements engineering for legacy system migration, the use of knowledge representation techniques for requirements tools, and domain-specific languages for developing executable specifications of reactive systems.

Shekaran received a BTech in electronics engineering from the Indian Institute of Technology in Madras and an MS in computer science from the University of Tennessee at Knoxville.

Jawed Siddiqi is director of the Computing Research Centre and professor of Software Engineering at Sheffield Hallam University. He is a founding member of the IEEE International Conference on Requirements Engineering and a permanent member of its steering committee. At Sheffield Hallam, he is the lead researcher on several projects, including the development of requirements-engineering tools, the animation of formal specifications, the assessment of quality-management and process-improvement methods, and the empirical and analytical evaluation of programming paradigms and human-computer interfaces.

Siddiqi received a BSc in mathematics from the University of London and an MSc and PhD in computer science from the University of Aston in Birmingham. He is a Chartered Engineer and a member of the IEEE Computer Society.

Address questions about this issue to Shekaran at Microsoft Corporation, One Microsoft Wy., Redmond, WA 98052; or by e-mail to either Shekaran at shekaran@microsoft.com or Siddiqi at J.I.Siddiqi@shu.ac.uk.

REFERENCES

1. H. Krasner, "Requirements Dynamics in Large Software Projects, A Perspective on New Directions in the Software Engineering Process," *Proc. IFIP*, Elsevier, New York, pp. 211-216.
2. M. Jarke and K. Pohl, "Requirements Engineering in 2001: (Virtually) Managing a Changing Reality," *Software Engineering*, Nov. 1994, pp. 257-266.
3. C. Potts, K. Takahashi, and A. Anton, "Inquiry-Based Requirements Analysis," *IEEE Software*, Mar. 1994, pp. 21-32.
4. A Davis, *Software Requirements, Analysis and Specification*, Prentice-Hall, Englewood Cliffs, N.J., 1990.
5. W. Swartout and R. Balzer, "On the Inevitable Intertwining of Specification and Design," *Comm ACM*, July 1982, pp. 438-440.
6. J. Siddiqi, "Challenging Universal Truths in Requirements Engineering," *IEEE Software*, Mar. 1994. pp. 18-19.
7. A. Borgida, S. Greenspan, and J. Mylopoulos, "Knowledge Representation as the Basis for Requirements Specifications," *Computer*, Apr. 1985, pp. 82-91.
8. M. Jackson, *Software Requirements and Specifications*, Addison-Wesley, Reading, Mass., 1995.
9. J. Goguen, "Formality and Informality in Requirements Engineering," *Proc. IEEE Int'l Conf. Requirements Eng.*, IEEE CS Press, Los Alamitos, Calif., 1996.
10. C. Potts, "Invented Requirements and Imagined Customers: Requirements Engineering for Off-the-Shelf Software," *Proc. Int'l Symp. Requirements Engineering*, IEEE Press, New York, 1995, pp. 128-130.
11. M.C. Shekaran and J.F. Tremlett, "Reasoning about Integration Issues During Requirements Definition: A Knowledge-Based Approach," *Proc. Int'l Conf. Systems Integration*, IEEE CS Press, Los Alamitos, Calif., 1992, pp. 229-239.
12. P. Checkland, *Systems Thinking, Systems Practice*, John Wiley, Chichester, UK, 1990.