Editor: Neil Maiden ■ City University, London ■ n.a.m.maiden@city.ac.uk

Requirements 25 Years On

Neil Maiden

So here we are—25 years of *IEEE Software*. Sitting down to write this column, I reflected on where I was a quarter of a century ago. I'd just left school, was lazing my way through summer, trying to do very little. I trained my parent's new dog and went to the beach a lot. All wonderful, but alas, not a great place from which to view 25 years of requirements practice and research.

So perhaps I'm too young to be writing this column. Fortunately, people such as Sol Greenspan, Peter Hruschka, and Suzanne and James Robertson were there 25 years ago, and I have melded their recollections with my knowledge to produce this perspective on 25 years of requirements work. Others will have different perspectives. I'll be happy to read them, so please send your letters to N.A.M.Maiden@city.ac.uk.

So What's Changed?

First of all, it's surprising how far we've come.

In 1983, most software developers weren't talking that much about requirements. When they were, it was about poorly understood ideas such as *requirements definition* and *requirements analysis*. Developers wrote requirements documents described as "monolithic Victorian novels,"[1] and—like novels—incomplete, ambiguous, and full of inconsistencies.

In 2008, by contrast, developers talk about requirements a lot. They know why requirements are important to their projects' success. They can write measurable requirements in structured specifications. They have access to requirements textbooks and case studies. Training in requirements techniques is widely available. Universities teach requirements engineering courses, and there are established requirements conferences and organizations. You can buy software tools to document, manage, and analyze your requirements. Heck, some organizations even have good, repeatable requirements practices!

Let's look back at the requirements practices that existed in 1983 and see what has changed and why.

A Narrow View on Requirements

By the 1970s, empirical data had revealed that requirements-related problems were a reality.[2] The best practice of the day was to write natural-language descriptions of system requirements. However, Bertrand Meyer reported common problems with these descriptions, including *noise*, *silence*, *contradiction*, *ambiguity*, and *wishful thinking*.[3] This was because many of the techniques tended to overcome technical problems with the requirements document itself as much as they overcame the more important problems with the specified system.

Specification Languages

Although natural language dominated, not all specifications were written in it. Formal specification languages such as the Vienna Development Method (VDM) were available. However, most analysts lacked the skills needed to use these languages, and the number of reported applications for specifying complex software-based systems was small. Program-specification techniques, such as Jackson Structured Programming,[4] were more usable but didn't apply to specifying system requirements.

So, in the late 1970s, software developers began to fill the gap between natural-language and formal requirements techniques with structured-analysis methods.

Structured Analysis: Starting to Describe the World

Today, we use requirements techniques to analyze the world in which we implement software-based

 0740-7459/08/$25.00 © 2008 IEEE

systems as well as these systems' requirements. In the early 1980s, structured-analysis methods were a first step toward such techniques. As early as 1977, a special issue of *IEEE Transactions on Software Engineering*, edited by Doug Ross, reported techniques such as the Structured Analysis and Design Technique (SADT), which viewed requirements analysis as the establishment of system boundaries and the specification of what the system must do.[5] The key innovation was to specify what a system should do independent of its implementation—something obvious now but revolutionary at the time.

SADT provided analysts with a graphical language for communicating functional specifications to others.[5] Looking back, what stands out is Ross's focus on good storytelling, exploiting communication patterns such as "tell them what you're going to tell them, tell them, then tell them what you've told them." His use of models to weave together a story at different levels of telling preceded the focus on scenario-based design techniques by 20 years.

Others developed structured-analysis methods to support communication with stakeholders. Tom DeMarco used dataflow diagrams as a new type of functional specification.[6] He advocated different types of physical and logical dataflow diagrams to describe current and new systems and analyze and understand a problem domain. Ed Yourdon and Larry Constantine's method also used dataflow and state transition diagrams to explore system boundaries and functional decompositions.[1] Other authors, such as Chris Gane and Trish Sarson, proposed similar methods.[7]

Michael Jackson took the need to describe the world one step further. In a 1978 paper, he argued that a specification should consider the system as a model of the world—the reality with which it's concerned—rather than start from system functions.[8] Analysts should explicitly model the reality, then use that model to inform system specification.

Although the software development community was making progress, structured-analysis methods back then contrast markedly with today's requirements techniques in at least three ways. First, those methods made remarkably little reference to requirements as first-order entities; most had no explicit representation of requirements or why a system was needed. Rather, they focused on functional decomposition using graphical-systems models, although DeMarco also used structured text descriptions to add procedural detail to the functional specifications.[6] Second, the methods offered little support for describing or analyzing quality requirements, such as performance, reliability, and security. In hindsight, the lopsided treatment of functional and nonfunctional characteristics is startling. Finally, the methods offered limited capabilities for reasoning about the systems being specified. The need to communicate models to stakeholders led to informal graphical languages amenable to only limited automated reasoning; the onus was on people to analyze and validate systems models manually.

Reasoning about the World

In the early 1980s, Sol Greenspan and his colleagues took an alternative approach inspired by knowledge representation work from artificial intelligence that offered semantic networks, first-order logic, and frames. The new focus was on representation and reasoning about all knowledge accumulated during requirements acquisition, not just the requirements.[2] The result was a framework that formed the basis for the Requirements Modeling Language (RML), which supported an object-oriented framework that treated a model as a collection of various kinds of objects. The framework provided an ontology for requirements modeling based around entities, activities, and assertions. RML had formal semantics that defined mapping from RML descriptions into a set of assertions in first-order predicate calculus.[2] In contrast to structured-analysis methods, more automated reasoning about requirements models became possible.

This early work on RML led to substantial growth in requirements modeling techniques such as *i**,[9] and methods such as TROPOS. *i** and the Knowledge Acquisition in Automated Specification (KAOS) pioneered techniques—now important in modern requirements methods—for modeling and reasoning about the goals and constraints on both systems and organizations.[10] In the past decade, these goal-based techniques have increasingly affected requirements practices both directly and indirectly.

In the past decade, goal-based techniques have increasingly affected requirements practices both directly and indirectly.

Acquiring Requirements

The need to model knowledge about the world led to new acquisition techniques. In 1983, developers poorly understood the problems that impeded them from effectively acquiring knowledge, and structured-analysis methods couldn't address these issues. Most analysts relied on interviews, which capture only verbalized knowledge at the expense of, for example, important compiled and taken-for-granted knowledge.

We've seen at least three distinct waves that have changed how organizations can acquire requirements. The first wave, during the 1980s, was the adoption of knowledge-engineering techniques developed originally to enable construction of expert systems. As software developers became aware of stakeholders' difficulties in articulating their work and requirements, they adopted new techniques, such as laddering, repertory grids, and card sorting.

The second wave, during the 1990s, introduced techniques from ethnography and human-computer interaction to understand the system environment.[11] Analysts became aware of working environments' complexity and the difficulty inherent in introducing software-based systems into them. Contextual inquiry[12] and cognitive work analysis techniques[13] provided new ways to capture knowledge and investigate changes to environments at the requirements stage.

The third, most recent wave was for

requirements invention. Developers often assume that stakeholders already know their requirements. However, stakeholders don't always know what new technologies can do for them. As technologies evolve, stakeholders need to create novel requirements by connecting knowledge of the problem with information about relevant technologies for solving it. We can now find success stories describing how creativity techniques such as constraint removal and analogical reasoning can improve requirements projects.[14]

In contrast to 25 years ago, analysts now have many more tried-and-tested techniques with which to acquire knowledge about the world and requirements.

Requirements Tools

Software tools for supporting requirements work have also changed. Back in the 1970s, DeMarco reported that analysts still worked primarily with pen, paper, and wits[6]—few computing tools were available. In the late 1970s, the first computer-aided software engineering (CASE) tools, the forerunners of modern requirements tools, appeared. In 1980, GEI in Germany developed ProMod, and in 1981, the first version supported DeMarco's Structured Analysis and Modular Design technique. Tony Wasserman worked on what became IDE's Software through Pictures, and Cadre was developing and selling Teamwork. The market for CASE tools grew substantially throughout the 1980s.

In some ways, these CASE tools were quite different from today's requirements management tools. The focus on graphical modeling using published methods contrasted markedly with the change management, traceability, and documentation features that requirements tools support. However, what really stands out about all these tools is how little our research advances have affected them. Today, commercial tools rarely have established capabilities for representing, structuring, and reasoning about requirements.

What Else Has Changed?

Other requirements practices have also improved over the past 25 years. We're now better at

- identifying the stakeholders in projects,
- discovering requirements with use cases and scenarios,
- describing and analyzing quality requirements,
- reasoning about natural-language specifications,
- handling different stakeholder viewpoints on requirements,
- dealing with and detecting conflicts between requirements,
- supporting social requirements activities,
- tracing requirements,
- managing large numbers of requirements over time,
- tailoring our requirements processes to different domains, and
- training people to be better requirements analysts.

There is a lot to celebrate.

And yet, we can still find practices common 25 years ago. Many people still write unstructured requirements specifications in natural language. Many projects lack trained requirements analysts. Many requirements-related problems still exist. Some argue against up-front requirements work in favor of more agile approaches. The backlog of software projects remains. Project costs and durations are difficult to estimate. Software still costs too much. And many projects fail. The requirements community still has work to do.

One reason is that we now apply requirements techniques to increasingly complex projects and to new types of systems—off-the-shelf, service-based, autonomous, and systems of systems. Both create new challenges for the next 25 years.

We now apply requirements techniques to increasingly complex projects and to new types of systems.

In a future installment of this column, I'll look at future trends in requirements practice and research—come and join the magic carpet ride!

Acknowledgments

My enormous thanks go to Suzanne and James Robertson, Peter Hruschka, and Sol Greenspan for their invaluable comments and insights that enabled me to write this column.

References

1. E. Yourdon and L.L. Constantine, *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*, Prentice Hall, 1979.
2. S. Greenspan, J. Mylopoulos and A. Borgida, "On Formal Requirements Modeling Languages: RML Revisited," *Proc. 16th Int'l Conf Software Eng.* (ICSE 94), IEEE CS Press, 1994, pp. 135–147.
3. B. Meyer, "On Formalism in Specifications," *IEEE Software*, Jan./Feb. 1985, pp. 6–26.
4. L. Ingevaldsson, *JSP: A Practical Method of Program Design*, Studentlitteratur, 1979.
5. D.T. Ross and K.E. Schoman, "Structured Analysis for Requirements Definition," *IEEE Trans. Software Eng.*, vol. 3, no. 1, 1977, pp. 6–15.
6. T. DeMarco, *Structured Analysis and System Specification*, Prentice Hall, 1979.
7. C. Gane and T. Sarson, *Structured Systems Analysis: Tools and Techniques*, Prentice Hall, 1979.
8. M.A. Jackson, "Information Systems: Modeling, Sequencing and Transformation," *Proc. 3rd Int'l Conf. Software Eng.*, IEEE CS Press, 1978, pp. 72–81.
9. E. Yu and J.M. Mylopoulos, "Understanding 'Why' in Software Process Modeling, Analysis, and Design," *Proc. 16th Int'l Conf. Software Eng.* (ICSE 94), IEEE CS Press, 1994, pp. 159–168.
10. A. van Lamsweerde, "Goal-Oriented Requirements Engineering: A Guided Tour," *Proc. 5th Int'l IEEE Symp. Requirements Eng.*, IEEE CS Press, 2001, pp. 249–261.
11. I. Sommerville et al., "Integrating Ethnography into the Requirements Engineering Process," *Proc. 1st IEEE Int'l Symp. Requirements Eng.*, IEEE CS Press, 1993, pp. 165–173.
12. H. Beyer and K. Holtzblatt, *Contextual Design: Defining Consumer-Centered Systems*, Morgan Kaufmann, 1998.
13. K. Vicente, *Cognitive Work Analysis*, Lawrence Erlbaum, 1999.
14. N. Maiden, S. Robertson, and A. Gizikis, "Provoking Creativity: Imagine What Your Requirements Could Be Like," *IEEE Software*, vol. 21, no. 5, 2004, pp. 68–75.

Neil Maiden is a professor of systems engineering at City University London. Contact him at n.a.m.maiden@city.ac.uk.