ELSEVIER

Information and Software Technology 42 (2000) 419–428

INFORMATION AND SOFTWARE TECHNOLOGY

www.elsevier.nl/locate/infsof

Requirements engineering: making the connection between the software developer and customer

H. Saiedian[a,*], R. Dale[b]

[a]Department of Computer Science, University of Nebraska at Omaha, Omaha, NE68182-0500, USA
[b]AIL Systems Inc., Technical Services Operations, Bellevue, NE 68005, USA

Received 11 June 1999; received in revised form 31 October 1999; accepted 11 November 1999

Abstract

Requirements engineering are one of the most crucial steps in software development process. Without a well-written requirements specification, developer's do not know what to build, user's do not know what to expect, and there is no way to validate that the created system actually meets the original needs of the user. Much of the emphasis in the recent attention for a software engineering discipline has centered on the formalization of software specifications and their flowdown to system design and verification. Undoubtedly, the incorporation of such sound, complete, and unambiguous traceability is vital to the success of any project. However, it has been our experience through years of work (on both sides) within the government and private sector military industrial establishment that many projects fail even before they reach the formal specification stage. That is because too often the developer does not truly understand or address the real requirements of the user and his environment.

The purpose of this research and report is to investigate the key players and their roles along with the existing methods and obstacles in Requirements Elicitation. The article will concentrate on emphasizing key activities and methods for gathering this information, as well as offering new approaches and ideas for improving the transfer and record of this information. Our hope is that this article will become an informal policy reminder/guideline for engineers and project managers alike. The success of our products and systems are largely determined by our attention to the human dimensions of the requirements process. We hope this article will bring attention to this oft-neglected element in software development and encourage discussion about how to effectively address the issue. © 2000 Elsevier Science B.V. All rights reserved.

Keywords: Requirements specification; Requirements specification; Key players

1. Introduction

Designing from a deep knowledge of the customer is central to any requirements definition process. When attempting to find out the right thing to do for the customer, we often focus on actions to take-interviews, questionnaires, and observations. However, the success of all these activities ultimately depend on how well people communicate and work together. Not only must we deal with the technical issues at hand, but we must also consider the interpersonal, cultural, and organizational aspects of our work environment. As a developer, it is absolutely critical that we recognize this human dimension in our discourse with the customer. To be successful in understanding the user and in meeting his needs with our products, we need to formulate customer-centered strategies and communication techniques that encourage customer participation and shared consensus in our product-making decisions.

This article will investigate the current state of requirements elicitation. It will introduce the goals of successful elicitation along with the key players from both the customer and developer and their intended roles in the process. The article will examine some of the more common methods of elicitation and the obstacles to successful information transfer. The article will then address these obstacles by identifying the keys to effective communication and by introducing new and alternative approaches and environments to approaching the elicitation problem.

The article will conclude by re-addressing the challenge and need realities for good requirements definition and analysis. It shall also attempt to offer encouragement and evolutionary strategy to any individual trying to be a "change agent" within his or her own organization.

* Corresponding author. Tel.: +1-402-554-2849; fax: +1-102-554-3284.
E-mail address: hossein@cs.unomaha.edu (H. Saiedian).

2. Background

This section provides the necessary information to orient the reader who may not be a specialist in the paper subject.

2.1. Role of requirements elicitation

The first step in any software developmental effort is to determine exactly what the software system shall do. Software requirements engineering is defined as all the activities devoted to identification of user requirements, analysis of the requirements to drive additional requirements, documentation of the requirements as a specification, and validation of the documented requirements against the actual user needs.

As part of this activity, software requirements elicitation is the specific processes of gathering, determining, extracting, or exposing software requirements. From the user's perspective, good elicitation results in them having a better understanding of their needs and constraints. From this, they will be able to effectively evaluate solution alternatives and understand the implications of their decisions. From the developer's perspective, it constructs a clear, high-level specification of the problem that is to be solved. It ensures that a solution is being developed for the right problem and that the solution is feasible. Most importantly, a good elicitation process builds a common vision of the problem and the conceptualized software solution between the user and developer.

An elicitation strategy is a set of guidelines for identifying the correct source of requirements and background information, eliciting requirements from them, and resolving conflicts among them. Of all the aspects of requirements analysis, it is the most communication-intensive. As a result, most of the techniques that prove useful do not come from computer science research, but from organizational theory, group interaction research, interviewing techniques, and practical experience [1, pp. 256–257].

2.2. Key players and their functions

Ideally, there are many participants involved in software requirements development. It is important that we recognize the existence of these diverse interests and the specific roles they play to ensure the right people are involved at the right time and that the right expectations are addressed. Failure to do so results in indirect and ineffective communication between the customer and developer. Issues that are not addressed until later phases in system development can become major schedule and budget impacts, at best, and even potential showstoppers.

The key players (from the customer side) and their respective functions can be classified as follows:

Buyer; buyers are the people responsible for contracting and paying for the software system. Their chief concerns include project schedule and budget, even to the point of compromising usability. These individuals are usually the project point of contact for meetings with the developer. While their background may be technical in nature, they are often long removed from the user/domain expert relationship with the work environment they oversee.

End user; end users are the individuals who ultimately will use the system developed. As such, they are most concerned with the usability, reliability, and availability of the system. These individuals are the ones most familiar with the specific work procedures being addressed by the system. They have the greatest stake when it comes to issues concerning user interfaces and user guide documentation.

Domain experts; these are the individuals who understand the system environment or problem domain where the software system will be employed. They are the source for technical input regarding system interface detail and requirements.

Software maintainers; for projects that will eventually be maintained by the customers, these are the individuals who will be responsible for future change management and implementation and anomaly resolution. As such, they are most interested in internal product issues such as design documentation and system architecture.

The key players (from the developer side) and their respective functions can be classified as follows:

Program management; these are the individuals responsible for product sales and marketing as well as overall project development oversight. Often, these are the individuals who deal directly with the customer.

Requirements engineers; these are the individuals, usually system engineers, who are responsible for the identification and documentation of the requirements.

Software engineers; these are the individuals who provide expertise on software design constraints, prototype development, and technical feasibility.

Testers; testers are responsible for developing and executing the necessary test conditions for development and sell-off activities. These include module tests, integration tests (as elements of the product are brought together), and ultimately system-level functional tests (both standalone and integrated with external interfaces). All are designed to validate and demonstrate delivered capability in level-up fashion. As such, testers must intimately understand and trace requirements from origin to final product to ensure product validation is feasible and completely and accurately documented. If we cannot adequately demonstrate to the user that the product meets their needs at sell-off, then its acceptance will be jeopardized.

As with the customer side, each of the specific disciplines for the developer brings a critical expertise to the requirements definition task. It is important that we do not promise something that cannot be ultimately implemented or that cannot be done given the current budget or schedule. For effective requirements information gathering, it is important that these various disciplines be organized and coordinated into a team for brainstorming and analysis activity with the customer. Product definition is unavoidably a battle of trade-off. User's invariably desire that which their resources and/or available technology cannot provide. It is critical that any early requirements change decisions (added cost, removed capability, etc.) are made with full customer involvement. Ensuring that the user understands their constraints (and the resulting impacts to their needs) is just as important as identifying their needs. Otherwise, the developer will shoulder the blame for failing to deliver what the customer wanted. Given all this, requirements gathering requires the developer representatives to have a wide variety of knowledge and skills in addition to the specific technical expertise in the application domain and software development. These include interviewing, group work, facilitation, negotiation, problem solving, and presentation skills.

2.3. Common techniques and tools

Currently, we gain understanding about the participants work environment and activities through such traditional techniques as on-site observations of on-going activities, open-ended interviews, and examinations of manuals, job descriptions, and organizational hierarchies. In order to gain an understanding of the user's work, we try to appeal to the very resources upon which the participants draw to achieve their own understanding of their work. Knowledge of many of these aspects can only be gained from experienced co-workers. Further resources are the situational and interactional context in which the activities of work are conducted.

Such a full complement of data gathered from interviews and document reviews is often to unwieldy to assemble and display let alone carry around to all relevant interested parties dispersed throughout an organization. Further, project schedule and budget often limit the time for and the number of participants involved in such information gathering. Face-to-face discourse is often almost non-existent. Depending on the complexity of the target problem domain, this can result in much omitted detail and perspective leaving us, as the developer, with a skewed view of the user's real needs.

3. Underlying difficulties with elicitation

The elicitation of requirements is a difficult process. This section of the article highlights the most common problems that hinder the identification/definition of the user's needs.

3.1. Poor communication

We can communicate goals, objectives, tasks, procedures, constraints, interdependencies, timetables, priorities, responsibilities, accountabilities, and even deliver the solution that perfectly meets our customer's needs—and still not meet our customers expectations [2, p. 38]. That is because how we communicate can be just as important as what we communicate.

If we inadvertently exhibit signs of non-listening when we work with customers, we give them the impression that we are distracted or uninterested. This can undermine our ability to work together effectively. Conversely, if people's perception is that you are listening, it leads them to feel comfortable with you and to open up to you.

In both verbal and written communication, it is important not to contradict what we intend to communicate. Over-eagerness to be responsive or the need to match or exceed the level of performance by a competitor can cause us to promise something that we cannot deliver. Conflict can also occur if we promise a level of service and then exceed it. For example, if we consistently complete projects for our customers ahead of schedule or deliver more than we promise, they will begin to expect the same in future projects. Worse, they may suspect we are padding our cost and schedule estimates in our proposals.

Indirect communication links are those in which the customer and developer do not deal with one another directly but communicate through intermediaries. The marketing and sales link (in which a salesperson serves as the intermediary) is one example of this as is a MIS intermediary who defines corporate customer's goals and needs to designers and developers. Intermediaries often do not have a complete understanding of the customer's needs and, as a result, can intentionally or unintentionally filter and distort messages.

To complicate matters, communication preferences often vary from customer to customer and from one circumstance to another. Each has its own priorities, time frames, pressures, and objectives. It is the responsibility of the developer to be cognizant of and adapt to the customer's communication styles considering such things as their pace of activity, receptiveness to new ideas, level of risk-taking, adherence to protocol, and written and oral presentation formats [2, p. 45]. Showing an interest in the customer's preferences tells them they matter and will ultimately help build the bridge to better communication needed to work together and meet expectations. Do not wait for the customer to complain to consider their preferences.

3.2. Resistance

Resistance is a physical expression of an emotional process-taking place within a person and takes the form of opposition. Resistance to new ideas is a perfectly natural part of any improvement process. Nobody likes being told they are obsolete or that they can do things better. Further, technological "revolutions." are often too frequent for most people's taste, especially when the costs of learning a new technology come close to outweighing the benefits [3, p. 42].

Being able to recognize and mitigate resistance is critical for the requirements engineer because it highlights issues important to the customer that are not being addressed adequately. Some of the more common forms of resistance are listed and briefly described below:

- *Time resistance*. Person never has time to meet with you.
- *Overload resistance*. No matter how much information is given to the person, it is never enough.
- *Silence resistance*. Person does not react or respond to anything said.
- *Impracticality resistance*. Person always reminds you that he lives in the real world.
- *Compliance resistance*. Person always agrees with you. Reservations are never expressed and the implications is that whatever you do is fine.

3.3. Articulation/expertise problems

A common obstacle to communication on technical matters is the use of terminology that is not understood by one of the parties. Professionals love the use of jargon and acronyms and use them profusely. Overuse of such terminology can serve to confuse, annoy, or intimidate and it is important to adjust to the customer's level of technical sophistication.

The complexity of modern software systems also complicates the elicitation process. Many systems have countless interconnections between subsystems and even environments that even experts in specialized disciplines have difficulty in understanding. Even more problematic is the ever-changing nature of requirements as user's learn and grow and the difficulties of integrating multiple, diverse and conflicting views.

Another problem can be the developer "talking down" to the "dumb users." In the minds of some engineers and programmers, making software easy to use is necessary only because "most people are too unsophisticated to understand the glories of a real computer system" [3, p. 70]. When a system is approached with that attitude, the result can be downright offensive to the oft-intelligent user. Too often, we think that something is too complex to explain, so why waste the effort? However, taking time to educate the customer has a number of benefits. It not only forces us to fully understand the problem and how our solution addresses it, but it demonstrates that explicitly to the user. More importantly, it demonstrates a confidence in our work and willingness to go above and beyond to help the customer. This can help build trust with the customer and encourage them to open up to us.

3.4. Problem perspective differences

Most companies in the high-tech industry have not yet integrated customer-centered techniques into the front-end of their development processes. Further, most development engineers have little or no experience as an end-user in the application domain for which they develop software. As a result, development tends to be technology (or solution) driven without a contextualized sense of the problem to be solved. We tend to offer products to our customers not based on their actual needs but rather on the whims of what updated hardware and software packages allow us to do. Nate Borenstein offers a great example of this misguided technology fixation in the following excerpt from his book, *Programming As If People Mattered*:

> The developer may emerge wild-eyed from his office, ranting to anyone who will listen about the breakthrough he has made, about how easy his new gadget or program is to use, about how it will revolutionize the way people talk to computers. His colleagues will smile uneasily and shift restlessly from one foot to another, pondering several alternatives, all of them unpleasant:
>
> 1.1. The developer has finally gone over the edge, lost his mind, and will not longer be useful for anything.
> 1.2. The developer has re-invented the wheel, and will eventually have to face the disappointment of knowing that his technique isn't new at all.
> 1.3. The developer has indeed made a minor innovation, but nobody is really likely to care about it because it is irrelevant for most purposes.
> 1.4. The developer is telling the truth, but the world is not ready for his breakthrough, and it will languish without aggressive and expensive marketing.
> 1.5. The developer is telling the truth, and the world is about to sit up and take notice. The colleague is going to have to take the time it takes to learn how to use this new breakthrough. Since he is not (yet) aware of any need for the breakthrough, this is an unappealing prospect.

Conversely, customers and end users are experts in the application domain but not in the process of engineering software. As a result, they are not aware of any existing limitations in their process that recent technology improvements have alleviated. They simply do not know what it is they can ask for in a new or updated product development.

As developers, we are in the business to make money—our companies would not survive if we were not profit-motivated. It is much more appealing to present a solution that requires significant new development effort than a simple modification to an existing tool. However, if we push something that is not truly needed, the strategy can backfire and potentially lose us future contracts. That is not to say we should not look for and offer new and better ways to do things. Customers and end users, although experts in the application domain are not usually proficient in the process of engineering software. As a result, they are not aware of any existing limitations in their process that recent technology improvements have alleviated. They simply do not know what it is they can ask for in a new or updated product development. It is our responsibility to inform them of the possibilities, but to always keep the focus on the benefit to their activity.

4. Techniques and tools for enhancing elicitation

This section of the article attempts to address the identified obstacles to effective requirements elicitation. In the end, it is the relationship between designers and customers that determine how well the design team understands and meets the user's needs. Customers and designers must develop a shared understanding of the work problems and the impact of technical solutions on the work [4, p. 31]. This section highlights important tips for better communication along with presenting some of the latest identified tools/aids for promoting meaningful information transfer dialogue. Where possible, examples of industry application of these techniques are identified.

4.1. Effective communication and information-gathering skills

Knowing how to show we are listening can help us improve our information gathering and relationship building. Eye contact, body posture, and facial expressions all convey information about listening or non-listening. These techniques for effective listening are equally pertinent to effective speaking.

To help focus our attention, it is important to actively participate in the customer's conversation. Techniques such as asking questions, re-stating what we have heard, and taking notes not only demonstrate we are listening but also help us in making sure we do listen. It is important not to overdo our responsiveness. Overly eager listening or too many interruptions can be worse than underdoing it. Jumping to conclusions about what a person is saying can lead us to focus only on their comments that support our conclusions and to ignore all others.

Frustration concerning customers who do not know or understand what they want requires us as developers to help them do a better job of describing their needs. Customers are often at a disadvantage when brought into a developer's meeting environment. The user's unique expertise is their real work experience. Taken out of this context, they are much less able to represent real experience [15, p. 94]. Customers often feel stress when asked by specialists to describe or document their needs. Offering customers something that resembles what they want—a focal point—is a powerful way to help those, who lack the necessary visualization, description, or language skills, to articulate their requirements [2, p. 66]. Focal points can backfire if the customer feels we are trying to railroad him into a solution of our choosing, not his.

Naomi Karten, in her book Managing Expectations, emphasizes that it is not only important to be an information-gathering specialist, but also an "information-gathering skeptic" [2, p. 77]. This role is required, not because customers deliberately mislead us, but rather because they, like us, often do not say what they mean. Karten offers several suggestions to extract information from customers:

1. Take nothing at face value. Never make assumptions. Repeat questions already asked, rephrasing them to get different perspectives.
2. Ask for clarification. Be sure to understand the customer's language used to describe a problem. Donot be afraid of admitting not knowing something. Stress to the customer how important our questions are to ensure we fully understand their expectations.
3. Gather information from multiple sources. We can get a broader perspective of diverse needs by presenting similar questions to individuals in separate interviews. Multiple responses helps to fill in the gaps in what any one given source provides.
4. Watch for inaccuracy. Be sure to consider the skill-level and any resistance in the customer. Are they answering questions from their own perspective or simply trying to tell you what you want to hear?

Even the types of questions we ask can help draw accurate information from the customer. Focusing on the process, rather than individuals, is less likely to generate defensiveness. Avoid simple yes and no type questions. Involve the customer in the questioning process by soliciting their concerns and by using consensus-building queries. Keep in mind that identifying what customer's do not want can be just as important as what they do want.

Also important is the need to understand the ways that indirect communication links can manifest themselves in practice and to work to rely more heavily on direct links (e.g. face-to-face conversations, etc.). Technological advances and cost reductions in the area of telecommunications have dramatically increased the techniques and channels that allow customers and developers to exchange information. In addition to standard links (such as surveys, interviews, and observational studies), support lines, bulletin boards, and e-mail now allow users to post questions and suggestions at the drop of a hat. However, this presents both a challenge and an opportunity. Undoubtedly, increasing the number of links between customer and developer is important. In his article, Customer–Developer Links in Software Development, Mark Keil presents the results of an inductive, multiple-case study regarding the communication characteristics between more successful and less successful projects. Based on his data it was shown that project managers should establish at least four different Customer–Developer Links to facilitate effective transfer of information. However, the same study also showed a point of diminishing returns by the time there are six or seven links [5, p. 38]. Further, the study data demonstrated that the absolute number of links is only a partial measure of customer participation. More important are the link characteristics, its perceived effectiveness, and how it is employed in practice. The question is not whether customers should participate in the development process but how they should participate. While a wide variety of specific techniques (such as those documented in this article) exist to aid requirements elicitation, what is missed is the detail as to who should be involved. As Keil points out, "links such as user groups will be less effective when meetings are attended by buyers rather than users and by marketers rather than developers" [5, p. 39].

4.2. Graphical representations of the user environment

They say a picture is worth a thousand words when it comes to describing something. The use of non-traditional "artifacts" to capture and characterize the user environment can help clarify and detail requirements understanding. Such artifacts can include wall-sized diagrams of the customer's environment, hierarchies of problem encountered, matrices showing key customers and markets, priority lists of benefits and capabilities desired, and storyboards of the products use and non-functional GUIs [6, p. 73]. Such items can bring written text to life giving the customer and developer alike a tangible conceptual framework that allows the proposed end product to be visualized.

The Aarhus University Research Foundation, in its introduction of new Picture Archive and Communication Systems (PACS) hardware and software to the Radiology Department at Skejby Hospital in Aarhus, Denmark, developed a graphical documentation tool, known as Blueprint Mapping, to aid their requirements research.

Essentially, they started from a regular blueprint of the building to establish an overview of the daily work. Conventional techniques such as interviews and questionnaires were used to initially document the general work procedures and roles of the professionals involved. The interviews were audio-taped to allow formal referencing of requirements. The interviews were supplemented by observations of various locations and activities and still photographs captured the physical setting. In addition, department employees participated in role-playing workshops to further illustrate standard and non-standard practices (see Section 4.3). The idea was to create "an overview of the current situation and at the same time try to frame the ideal future" [7, p. 56].

The size of the diagram was scaled up to make room for post-its representing information about employee locations and work activities performed. Miniaturized copies of equipment photographs were positioned with data entry and data retrieval actions indicated by arrows of colored tape. The map was presented to the department staff, and they added, changed, deleted information as needed. The map was then used as the shared understanding of the workspace and its problems, as well as to identify and develop ideas for alternative organizational designs.

Certainly, this example is not overwhelming in its technological achievement. But, it was easy to create and modify and it did prove to be a valuable tool in eliciting information from the customer and in refining the developer's overall understanding of the requirements. It just goes to show what a little creativity and interactive discourse can achieve.

The use of videotape has been found to be particularly effective in documenting the user site and activities. With the consent of the participants, video recordings can be obtained of people going about their everyday activities as well as during interviews and information-gathering meetings. This enables the users to speak and be heard in their own terms without intermediary interpretation. Complaints and requests seen and heard directly from users attempting to do some piece of work have much greater authority, far greater effect, and are more difficult to dismiss, than any summary reports field workers produce [8, p. 68].

Video provides a stable reference that can be shared, viewed, and discussed by viewers having different backgrounds. Tapes can be transcripted and time-stamped to make it easier to locate particular clips. It allows people, who otherwise would be unable, to interact with the user environment and results in everyone gaining a richer understanding of the activities and issues being explored. The real-time nature of video provides a glimpse of the user environment that still photos and written or audio-only products can never match. Video helps reveal easily overlooked features of the work environment (e.g. a task that is much more involved than initially described). It can be reviewed repeatedly and frozen allowing a more thorough review. Implications of certain features not apparent to some members of a design team may be detected by others. As a result, it can lead to early identification of possible sources of misunderstanding.

Experience has shown that user willingness to participate in video documentation is surprisingly high despite the fact that it requires a considerable investment in terms of time and resources. Part of the reason may be that the participants can use the video for their own purposes, such as displaying their work to others in the workplace.

4.3. Customer participatory techniques

Direct customer involvement in requirements definition early-on is a consistent factor in successful programs. The need to use participatory analysis is critical when it comes to understanding the flexibility aspects of a workplace, since flexibility concerns not the regular procedures and standard ways of doing things, but the unexpected, unprecedented, exceptional cases, situations, and events that are only experienced by the people who do the day-to-day work.

One way of accomplishing this is the use of situational workshops or organizational games. Such an approach was demonstrated by The Aarhus University Research Foundation during its PACS project with Skejby Hospital in Aarhus, Denmark. In conducting the organizational game, situation cards—index cards describing a specific situation at the workplace—were used. These situations were solicited from the customer prior to the activity and described situations in which normal procedures were not followed. During the game, the situational cards were used to trigger discussions about how particular situations are actually handled and other ways they could be handled [7, p. 57]. Such discourse helps the developer identify requirements raised by exceptions.

The traditional relationship between a master and apprentice has also been identified as a useful interactive model to encourage information elicitation. Just as an apprentice learns a skill from a master, designers want to learn about their customer's work from the customer. A designer taking on the role of apprentice automatically adopts the humility, inquisitiveness, and attention to detail needed to collect good data.

Apprenticeship in the context of on-going work is a most effective way to learn. "Nobody can talk better about what they do and why they do it than they can while in the middle of doing it" [9, p. 46]. Talking about work while doing it ensures that details are not confused or omitted. Teaching in the context of doing work obviates formal presentations and course materials, meaning little extra effort on the part of either the master or apprentice. Further, transfer of information in this context is not just limited to the work at hand. Events that occur while the designer is present can remind customers to talk about events that happened previously.

In their article *Apprenticing With The Customer*, Holtzbalt and Beyer detail how such an apprenticeship would work in practice:

> In practice, the customer might work for a while, with the designer looking on. The customer is immersed in the work, thinking about content (as usual). The designer, as apprentice, looks for patterns, for structure, and for things he or she does not understand. At some point, the designer makes an observation or asks a question. This interrupts the flow of work. In order to respond, the customer has to stop working, step back, and think about the question…. The customer now responds on two levels: First, he or she addresses the question and a conversation about the work ensues. When the conversation winds down, the customer returns to doing the work and the designer returns to watching. Second, the question is an example of seeing strategy where before the customer saw only actions. Customers soon learn to see strategy, and start interrupting themselves to make observations about the work they do [9, pp. 49–50].

While apprenticeship defines an attitude for designers to adopt, their motive in observing work is not that of an apprentice. The designer is constantly thinking about ways to improve the work structure and strategy with technology. The apprenticeship model allows both designer and customer to introduce design ideas as the work suggests them. The customer can respond to the idea while doing the work the idea supports—there is no better time to get feedback.

Maintaining the apprenticeship relationship requires attention to the interaction or the customer and developer may fall back into more traditional patterns for requirements gathering. As an example, working together as such usually results in a close, personal relationship. This can lead to conversations that are irrelevant to the design focus which can reduce the effectiveness of the information-gathering activity.

4.4. Prototyping

In the context of requirements engineering, prototyping is the construction of an executable system model to "enhance understanding of the problem and identify appropriate and feasible external behaviors for possible solutions" [10, p. 77]. Prototyping has been identified, especially for user interface software products, to be a viable, maybe even necessary, definition and development tool that reduces risk by early focus on feasibility analysis, identification of real requirements, and elimination of unnecessary requirements. User interfaces are fundamentally evolutionary artifacts, much more so than other computer programs [3, p. 116]. No matter how carefully they are designed, they change frequently, often radically, in the early stages of their development. Thus the effort to prematurely formalize the requirements and to "perfect." the code is wasted. A better approach, in many cases, is to make the initial user interface "a quick-and-dirty component of the larger system, the rest of which is built as a more stable and lasting edifice" [3, p. 117]. Then when the interface has stabilized it is rewritten in the traditional style of professional design and programming.

Stephen Andriole strongly believes that prototyping is a critical prerequisite to specifying requirements. In his article "Fast, Cheap Requirements: Prototype—Or Else!," he identifies a fast, powerful, cost-effective merger of requirements-modeling and the prototyping process. The key with prototyping, he stresses, is to simply get something running quickly that can be tested on the users [11, p. 85]. Prototypes, by their very nature, are designed to be disposable and as such can and should cut corners wherever possible when it comes to issues such as reliability, robustness, and algorithmic efficiency. It is generally acceptable for prototypes to simply die when they get unusually abnormal input data, when they run out of memory, or when an interfaced subsystem fails. The danger, however, is that the prototype evolve semantically into a final product as they so often do. Therefore, a "good." prototype will be full of shortcuts and assumptions, but each will be briefly documented on a list of things to be cleaned up, rewritten, or redesigned for the final version of the interface. Such a list will simplify (and justify) the ultimate choice between a total rewrite and a thorough cleanup of the prototype.

A typical prototyping phase can be outlined as follows:

1. Design the user interface. Make your best guess on limited user input and your own past experience to

develop a rough outline of the interface functionality. Since this is a prototype, nothing formal is needed.
2. Build a “quick-and-dirty system.” The prototype should be built well enough to let users assess the interface, but need not address reliability and error-checking. Keeping a complete list of all known inadequacies will prove invaluable if the prototype doesn’t get thrown away as planned.
3. Let users play with the prototype. There is a natural tendency to want to “polish.” the prototype before releasing it, but the real strength of prototyping is getting something to the user quickly for their immediate feedback.
4. Observe the users. Pay particular attention to first reactions since people learn quickly to work around or accept interface deficiencies or annoyances. Ensure the methods used to accomplish goals are the ones intended or redesign may be in order.
5. Collect user feedback. Ask the users a series of questions, both specific and open-ended, about the prototype. Allow for both face-to-face and anonymous feedback from multiple sources to ensure a comprehensive review.
6. Back to the drawing board. Reconsider the interface (and its requirements) in light of the experience with the users, and return to Step 1 [11, pp. 85–86].

The prototyping cycle obviously cannot continue endlessly. At some point, waning interest or schedule will require it to be complete. Some developers have devised user-satisfaction indices by which this decision can be made, e.g. when 90% of the users consider the interface satisfactory. Most often, however, a simple and definite amount of time is budgeted for prototype activity when a project starts to eliminate arguments whether to continue or not.

5. Implementing change

Good requirements elicitation takes time and money and as such often faces a hard sell to justify to program managers with ever reducing budgets and schedules. Integrating new tools and process techniques into an organization imposes additional workload upon the requirements process—they cannot simply be plugged in and expected to work [12, p. 79].

Good practice must be introduced incrementally and its performance verified. Some tools and techniques will work better than others will in a given situation. Some agencies may not want to be videotaped for fear it will eventually be used against them if negative workplace practices exist. Highly interactive techniques such as the apprentice relationship require mature and motivated participation on the part of the customer. Such a relationship may not always exist. As with any process, elicitation has to be adapted to match the scope of the task, the initiative maturity of the using agency, and the cost and schedule constraints. Identifying an exact strategy for any situation is impossible. However, the Esprit Requirements Engineering Adaptation and Improvement for Safety and Dependability (REAIMS) Project has proposed the Requirements Engineering Good Practice Guide (REGPG) to “define a measurable process improvement framework for the systematic, incremental adaptation of good requirements practice” [12, p. 79].

The REGPG, like the Capability Maturity Model (CMM) for overall software development, uses multiple maturity levels to characterize the requirements process and a strategy to improve it. Three levels exist: initial, repeatable, and defined. REAIMS research indicates that nearly all existing requirements processes are at level one having only pockets of good practice. The key according to Pete Sawyer of Lancaster University (involved in the Reaims project) is “increasing the use of appropriate good practice.” It is not simply a matter of buying more tools or using new approaches, but of “introducing the right practices, in the right order, at the right place, and with the required degree of strategic commitment” [12, p. 80].

The REGPG defines 66 good practices that are extracted from existing requirements standards and practices. These are classified as basic, intermediate, or advanced depending on their utility and required expertise. Basic practices represent the fundamental measures that are part of any good requirements process (e.g. documentation standards and organizational procedures). The intermediate and advanced practices represent increasingly technical and specialized measures that make the process more systematic. Each measure contains a qualitative assessment of the costs and benefits associated with introducing the particular measure. The maturity levels are based on the implementation progress of the various practices. Organizations, once they have baselined their existing process strengths and weaknesses, can use the REGPG as guideline to introduce practice improvements incrementally to address areas of need. The maturity level is calculated by measuring (via interviews and checklists) the extent to which individual practices have been implemented and summing the associated numerical scores. This self-assessment used to baseline the existing process and to evaluate the continuous-improvement process, is the difficult part. The REGPG, unlike the CMM, does not require outside certification, and as such loses a degree of objectivity. People naturally tend to bias the successes of their existing practices and this must be given consideration during scoring analysis. However, the internally driven nature of REGPG is also a strength. It is not overly prescriptive—i.e. it does not require a massive infrastructure to create and maintain. Too often, nowadays our prescribed solutions to process failures are worse than the original problem. We create untold new bureaucracy and spend more time defining and monitoring progress than we do making progress. REGPG is simple. It provides a meaningful measure of process evaluation and improvement by matching the most effective measures to the most pressing needs. It helps to raise management's awareness of the importance of requirements engineering in a visible and quantifiable way. And yet, it remains flexible and feasible enough to adapt to specific company and/or project needs for targeting improvement.

Changing the attitudes and work habits of any organization can be daunting at best, especially if an organization perceives itself as successful and secure, dominating the market for its products. Dennis Allen faced this same dilemma at WordPerfect Corporation several years ago as a product design strategist. Allen recognized the lack of direct contact with the customer and the need for "a software requirements definition process grounded in the real work of real customers using real machines" [13, p. 82].

Allen was particularly sensitive to the urgency for change based on his experiences in software development at a start-up software company that was working on natural language, machine-translation systems. He had witnessed the complete cancellation of their development efforts due to customer rejection. This rejection stemmed from a failure by the program managers and company leaders to recognize and address the customer's true needs. Rather than correct their fundamental ignorance relating to the user's work practice and its effect on the success or failure of their software applications, they instead directed all efforts to responding to corrections of individual problem reports and complaints.

These experiences were used by Allen to be powerful, personal motivating influences in his efforts to be a "change agent" at WordPerfect and to avoid repeating the same dismal ending. His approach was instructional and collaborative in nature, seeding concepts in the development community rather than "harassing people into submission" [13, p. 83]. He wanted to avoid the risks of cross-organizational antagonism. By focusing on introducing only one or two concepts at a time, casually and repeatedly working the topic into as many conversations as possible, and creating opportunities for conversation throughout the organization, he was able to quietly create a context for change. Before long people were pressing ahead and claiming the idea as their own. Allen found his approach to be successful for several reasons: It is not preaching, but rather casual conversation that seeks to engage and solicit individual opinion. In addition, it includes everyone. Allen stresses that its important that everyone be given the opportunity to stand in the spotlight. This creates a sense of community. The greater the sense of community and the more unified the vision of the goal, the faster you can move [13, p. 84].

6. Conclusions

As Stephen Andriole points out, "the software industry talks a good requirements game, but seldom takes it seriously" [11, p. 87]. We need to understand the realities of requirements elicitation and analysis. User-specified requirements are often vague, ambiguous, and incomplete. It is the nature of the game given the user's oft-limited expertise in the technical aspects of their application domain and in software development. As a result of this ambiguity, requirements can be discovered, evolve, change, and disappear without warning throughout the development cycle.

However, while such unpredictability is inevitable, it does not mean we cannot work to mitigate its influence or reduce its presence. Certainly building flexibility into our program definition and follow-on design is important to allow for downstream changes in requirements. But, no matter how well we buffer ourselves, such changes incur significant costs in time and money. It would be better to avoid the need for such changes altogether. And that is where an effective requirements elicitation strategy comes into play. Taking the time to learn about our customer and his work environment yields many benefits. In addition to helping the user (and us) understand what it is he needs, it also helps to foster a better working relationship and communication between the developer and customer. By better addressing the users' needs through their participation and through consensus-building, we help to break down the "us vs. them" mentality that exists in many customer agencies.

The elicitation techniques and approaches to change we choose are not important. What is important is recognizing the participatory necessity of the customer in defining their requirements to us. This philosophy needs to become a standard part of our development goals—one that is accepted as an absolute need, not a nice to have, and one that is budgeted for in time and schedule. Ultimately, it is the customer that decides our fate whether we like it or not and whether or not we think they are qualified to proceed without us. Without the customer's business and dollars, we cease to exist. Including the customer right from the beginning in our product definition and design goes along ways in ensuring our delivered systems meets his needs the first time. That in turn instills confidence, hopefully laying the groundwork for a growing and mature business relationship.

References

[1] C. Potts, Seven (plus or minus two) Challenges for Requirements Research, Sixth International Workshop on Software Specification and Design, IEEE Computer Society Press, Los Alamos CA, 1991, pp. 256–259.

[2] N. Karten, Managing Expectations, Dorset House Publishing, New York, NY, 1994.

[3] S. Borenstein, Programming As If People Mattered, Princeton University Press, Princeton, NJ, 1991.

[4] K. Holtzblatt, H. Beyer, Requirements gathering: the human factor, Communications of the ACM 38 (5) (1995) 31–32.

[5] M. Keil, E. Carmel, Customer–Developer Links in software development, Communications of the ACM 38 (5) (1995) 33–44.

[6] A. Hutchings, S. Knox, Creating products customers demand, Communications of the ACM 38 (5) (1995) 72–80.

[7] A. Kjaer, K. Madsen, Participatory analysis of flexibility, Communications of the ACM 38 (5) (1995) 53–60.

[8] F. Brun-Cottan, P. Wall, Using video to re-present the user, Communications of the ACM 38 (5) (1995) 61–71.

[9] H. Beyer, K. Hotzblatt, Apprenticing with the customer, Communications of the ACM 38 (5) (1995) 45–52.

[10] P. Hsia, A. Davis, D.C. Kung, Status report: requirements engineering, IEEE Software 10 (6) (1993) 75–79.

[11] S. Andriole, Fast, cheap requirements: prototype, Or Else!, IEEE Software 11 (2) (1994) 45–52.

[12] P. Sawyer, I. Sammerville, S. Viller, Capturing the benefits of requirements engineering, IEEE Software March/April (1999) 78–85.

[13] C.D. Allen, Succeeding as a clandestine change agent, Communications of the ACM 38 (5) (1995) 81–86.

[15] K. Holtzblatt, H. Beyer, Making customer-centered design work for teams, Communications of the ACM 36 (10) (1993) 93–103.