

## Requirements Engineering Roundtable

Alan Christie, Sholom Cohen, David Gluch, Nancy Mead, Mark Paulk, Patrick Place  
Moderated by Bill Pollak

This article captures an exchange of ideas among members of the SEI technical staff who work on different technical initiatives:

- Alan Christie is active in promoting the application of process and collaboration technologies to make the practice of software development more effective.
- Sholom Cohen's current research activities include object technology, software product line practices, and product line introduction.
- David Gluch is investigating software engineering practices for dependably upgrading systems, focusing on software verification and testing.
- Nancy Mead, guest editor of this release of *SEI Interactive*, was general chair of the International Conference on Requirements Engineering last year. She is currently involved in the study of survivable systems architectures.
- Mark Paulk was the "book boss" for *Version 1.0 of the Capability Maturity Model® (CMM®) for Software* and was the project leader during the development of CMM Version 1.1. He is also actively involved with software engineering standards.
- Patrick Place is a member of the SEI technical staff in the COTS (commercial off-the-shelf)-Based Systems Initiative.

### The problem of requirements engineering

**Bill Pollak (BP) (moderator):** At the SEI, we look at requirements engineering from the different perspectives of our technical initiatives. How can we improve requirements engineering as part of our mission to improve software engineering practices and technology?

**Patrick Place (PP):** I honestly believe that requirements are the root of all evil. A lot of the problems that we face in our work with COTS (commercial off-the-shelf)-based systems stem from requirements--this belief that you can do things the way you used to

do them. I just heard someone say yesterday, “We had this great success in setting requirements and in developing this particular system.” Their “success” was that they got the system fielded. It’s an appalling system--it’s unmaintainable, it can’t be upgraded, and it’s going to be a disaster. But they’re ready to do the same thing again because they’ve had a success!

You almost wish that the system had not been fielded. But I think if we want to be software engineers, we have to consider the entire cycle, the entire life of our systems from inception to final turnoff, cradle to grave. And that involves figuring out what that system should be, which is collecting and eliciting requirements and maintaining those requirements throughout the life of the system. Because if you just do it once and then throw it away and let the system evolve, you’re in trouble.

**Sholom Cohen (SC):** In the case of our work with product lines and architectures, customers either have nothing in the way of product line requirements and are looking to establish some basic set of requirements--they aren’t really looking in detail at how to elicit, manage, or track requirements--or they come to us with an architecture already in hand and say, “Now what do we do with it?” Or they come to us with a system that’s in trouble and say, “How do we reengineer it?” So it doesn’t seem like people are coming to us saying that they need help with managing requirements. We might say that there’s a problem there because they can’t track the requirements. And they say, “Well, give me a tool to do it.”

**Nancy Mead (NM):** We see a lot of people coming to the CERT® Coordination Center and asking us to analyze their existing systems to see where the security holes are. Right now we’re really trying to push the idea of having them look at their architectures up front to see if they’re survivable. There is not as much awareness of these up-front survivability issues as there should be, but the awareness is growing.

**Alan Christie (AC):** Often what is required in developing a set of requirements is a broader systems perspective--to be able to see the whole as more than the sum of the parts, to be able to assess the behavior of the system as an integrated entity. Complex feedback effects are very difficult to understand and often counter intuitive. As system complexity increases, it is more and more difficult to predict, at the requirements phase, what exactly will happen. As a result there is requirements churning, delaying project deadlines and pushing up costs.

I believe this situation can be addressed with the appropriate use of simulation techniques. Simulation has been used for many years within the engineering community to predict everything from traffic flow to aircraft wing loadings to the likelihood of a reactor accident. However, the software community has, ironically, been slow in adopting

this technology. I don't believe that simulation is a silver bullet, but it is more than a technology in the sense that its use can raise awareness with respect to the importance of dynamic effects in complex software systems. By using simulation to support requirements, a customer can achieve a much-improved sense that a supplier understands the issues. In addition, if changes are proposed, a simulation of the changes is likely to wring out potentially unforeseen consequences.

Simulation software is now mature and readily available, but it will probably take a few dramatic examples of simulation supporting software requirements before the community takes notice.

**PP:** A lot of our activities in the COTS-Based Systems Program have been based on a customer coming in and saying, "I've got this legacy system, it's ancient, we have to build a new one, how do we do it?" And the first thing is for them to even understand what it is they have in their existing system. They have documents that are hundreds of pages long that are supposed to describe their requirements, but they don't. They're a mishmash of operational procedure, of detail, of flowcharts...but the documents don't provide any understanding of what their requirements are.

**NM:** And they're telling you that they really have no choice; they're up against a wall.

**PP:** Some are not quite as up against the wall as others. But they're all under a federal mandate to use COTS products, one way or the other. And so they're trying to figure out how to do it. We tend to work on the design and the technology, but one of their big problems is understanding the requirements that they need to build to.

**NM:** The other problem in the requirements area is that requirements are higher risk than some of the downstream problems we deal with, and we don't have nice, neat solutions to hand to people for problems related to requirements. We can't say "If you go off and use this analysis technique or that elicitation technique, everything is going to be fine."

### **The human side of requirements engineering: natural language vs. formal methods**

**BP:** Requirements are more than a technical problem; in large part, they are a people problem too.

**Mark Paulk (MP):** Most of the real requirements engineering value is in the people stuff. It's fundamental communications, not writing things down in a mathematical formula. It's going out there and talking to people, and eliciting what the requirements are, then capturing them in some way that an MBA can understand.

Probably the best book I know of in the area is Gause and Weinberg's *Exploring Requirements* [Gause 89]. They have exercises where you look at a sentence like "Mary had a little lamb," and you see that the meaning changes depending upon which word you emphasize. So we have to look at the whole idea of the way that we communicate, which is really an interpersonal skills issue.

There are folks out there who are doing good work. Barry Boehm, for example, with his win-win stuff [Boehm 99], is in my opinion one of the folks who is at the forefront of this kind of activity. All of us have to do requirements elicitation and analysis for our particular projects. But in terms of looking at requirements engineering as a discipline, I think you really need somebody or a group of people who focus more on the elicitation side and the capturing of requirements, which is a communications issue more than it is a formality issue.

**PP:** I'd like to take issue with one of your points. I agree that requirements are a people issue, but you jump to the conclusion that writing things down mathematically is not the right thing to do; I think this is a bit of a leap. I was involved in an exercise with the IEEE POSIX.21 standard for realtime distributed systems communication. We communicated in English to the people who were developing the requirements about the flaws that they had in their existing requirements set, but that communication was based on a mathematical analysis. I agree with you wholeheartedly that I would not take 20 pages of mathematics--or even one page--to the average system acquisitions individual and say, "Here, now you know what the system is." But you can make the interpretation back into the system's own language or domain of discourse because you can see from the mathematical logic the consequences of your decisions; I will tell you what those consequences are not in a stream of symbols, but in a stream of words that have meaning to you, and you can tell me if I'm right or wrong. And if I'm wrong, I can adjust my model, and if I'm right, you can adjust your requirements.

**MP:** I agree, you're absolutely right. Those kinds of analytical techniques are very useful. The point I was making is that a lot of the work I see that is reported under the requirements engineering label is really dealing with a requirements specification language, a formal methods kind of approach: "We're going to have a complete, consistent, unambiguous set of requirements." And then you go out and you try and explain that, because when you try to translate from the formal specification back to the English, you inject ambiguity back in. That's the reason you went to the formal specification in the first place.

You can get fairly detailed with a rigorous analysis, but when you try to abstract that back up to the CEO or the MBA or whomever your customer is, things you have captured absolutely precisely are just not what they really meant. Or as we've said several times,

they don't know what they really mean, but they'll figure it out as we iterate with them on what the system should be.

**David Gluch (DG):** One of the things we're talking about in model-based verification is to use formalism and pragmatic, focused models as communication/understanding vehicles. As people analyze requirements specifications, they create an abstracted model, and there is communication. They begin to understand the engineer who is building the system, and they also make users, clients, and other stakeholders aware of the issues that emerge. So we see that using formal language effectively as a communication as well as an analysis vehicle is a very effective way to identify errors in requirements and make them explicit. People may consciously decide about a certain requirement, for example, "I'm not sure about this, I'd like to defer it."

**MP:** Is this coupled with tools?

**DG:** It can be, in terms of the actual checking part of the model, but it's mostly just the process of building the model--of abstracting out the essential elements--that helps focus discussion about a system and also identifies ambiguities.

**PP:** That's a communication from the people who have the requirements in their heads to you the engineer. It doesn't address the communication from you the engineer back to the people so that you can fix what's in their heads because they've got it wrong.

**MP:** Let's broaden what we said earlier, which is, if you know who your audience is, then you can communicate in a way that they will understand. If you're talking to a software programmer, and you say, "Now here is the language specification for C++," you might use the formal BNF specification to help the programmer understand the syntax for the language. But if I were trying to teach some manager how to do Programming 101 so he could write a little sort program, and I put one of those specifications down, I can predict that there would be some resistance to that form of communication.

**PP:** A course at the University of Manchester actually did that. They gave first-year graduate students a formal specification, plus a natural-language discussion, for their first exercise. In fact, for all of their programming careers, students used formal specifications in their exercises. And the university found that successful--that's why they did it.

**MP:** I made that mistake too when I was starting out teaching. It didn't work out as well for me. I guess that natural-language discussion is critical: the better you are at that, the more success you're going to have.

**PP:** Yes, you need the two together.

**NM:** I think that's a problem that we see a lot. We do need both, and people don't want to invest in both. They want to pick one or the other, and I'm not sure that one or the other can do the job that needs to be done.

**AC:** I don't think that there is any real conflict between natural language and formal requirements. Usually formal requirements are applied to narrowly focused elements of the problem while natural-language requirements complement this in the broader scope. For high-reliability and safety-critical systems, formal requirements make a lot of sense for precisely describing the system's interactions, but that's not necessarily best for providing an understanding of overall capability.

**PP:** The most important part of a formal specification is the natural language that surrounds it. Any mathematical model can state very precisely what the requirements are, but there's no intuition. And you can provide all the intuition about that precision in natural language, so that it's not just 20 pages of mathematics. It's mathematics plus text that says what the mathematics mean.

### **Requirements engineering in other disciplines: Is software unique?**

**BP:** Is requirements engineering particularly difficult with software?

**DG:** We often seem to function in isolation, but formal representations are routinely used in structural engineering. They go to mathematics very quickly, they go to formal representations, and they're integrated--mathematics are part of how they define requirements, part of how they accomplish this communication and translation to the engineering staff and to the other people involved. I don't think there is anything fundamentally different about building software than building anything else, but there's a larger focus on intellectual intangibles up front that are more difficult to describe.

**PP:** The issue is one of complexity. We are creating software with thousands of functions, many times more complicated than any other manufactured artifacts. Even well-understood programs have incredible complexity. In a previous roundtable discussion in *SEI Interactive* ([http://interactive.sei.cmu.edu/Features/1998/June/COTS\\_Roundtable/Cots\\_Roundtable.htm](http://interactive.sei.cmu.edu/Features/1998/June/COTS_Roundtable/Cots_Roundtable.htm)), David Carney likened COTS software to bridges as opposed to screws; however, I think that he's doing software an injustice--it's more like building a country's road system when the pieces we have to interconnect are the towns! This complexity takes software out of the realm of the other engineering disciplines.

A second significant difference is that software is, by its nature, of extreme generality. A toaster makes toast and could, if you bought an expensive enough toaster, be coupled to all sorts of other household artifacts so that your morning toast was available, say, 10 minutes after your morning shower--whenever that occurred. That said, it's still a toaster, and if I wished I could create the mathematical language to describe and engineer toasters.

The mathematics of bridges are well understood and relatively simple. However, for software, since it has no defined purpose, the best we can do is create a mathematics capable of modeling any aspect of software. Unfortunately, this leads us to all computable functions (a technical term, not to be confused with computation)--this is a lot of mathematics to cover and means that we are without a domain-specific language to use for our engineering (unlike bridges that deal in tensile strength as well as forces and moments).

Another point to bear in mind is that software development is really new; we've only been developing what we consider to be software for a tiny fraction of the time during which we've been building bridges.

**AC:** One distinction between most other engineering disciplines and software is the fact that, in traditional engineering, the results of what you produce are mostly tangible. Buildings, bridges, and aircraft all have a physical presence that software does not. I think this physical presence allows one to more easily visualize the issues and to think in metaphors. For example if I understand the reason why bridges are structurally sound, I can probably use that knowledge in designing safe buildings--or even aircraft. I'm not sure that this intellectual reuse is as portable with software. At least it is not being used that way today.

**DG:** My comment relating to specifications for other disciplines was intended to be a very general observation that other engineering disciplines rely heavily on mathematics as a basis for describing and understanding their systems. I did not mean to select structural engineering specifically. I believe this is true across the board in engineering endeavors.

My point is simply that other engineering disciplines rely on mathematics (formalism) and so, if software engineering is to be an engineering discipline, it too should include mathematical formalism as integral to doing "good" engineering for software systems. The issue is not whether formalism should be included but rather how it should become part of the discipline.

Also, I was speculating on what differentiates software from the other engineering disciplines. While it may seem that software engineering does not have much in common with mechanical engineering, I think it does. But consider another example--electrical engineers (digital) designing microprocessors or custom application-specific integrated circuits. I believe the issues and problems here closely align with those of software implementation. It can be argued, though, that the similarity is at the design/implementation levels rather than at requirements. But in looking at requirements, one may ask how software requirements engineering differs from systems requirements engineering.

Pat, relating to the complexity premise, I am not sure about this perspective. Consider the Boeing 777. I believe that the complexity of the entire aircraft is greater than the complexity of the software that comprises only part of it, and that the requirements specification for the software can be considered as resulting from doing the system design for the aircraft.

**MP:** I agree with Alan that it is easier to have a set of shared paradigms when you have something tangible to look at. So when you're saying "I want to build a bridge," if I'm the mayor of Pittsburgh, I can say "I want a bridge to go from here to over there and I want to be able to run 50,000 cars every day over it." And so you can specify from the user's perspective.

Dave is absolutely right, when the engineers started doing that, they started getting into the mathematics from Day 1, but those mathematics are communicated to the city engineering office, they're not communicated to the mayor. There are different audiences for the different ways that you capture things, and the problem that we run into in the software world is not that we don't have different audiences, it is that we try to communicate the same way to all of them.

## **Communication**

**NM:** In one of the projects that I was on, we had a series of design reviews. And one of the later design reviews was a review of the user interface and displays. At that review, we found out that these folks who had sat through all the previous reviews had no understanding of what was going on, because they were looking for data on the displays that wasn't being computed in the system; and if they had understood everything that had come before then, they would have realized that not only was the data not being displayed, it wasn't there, it didn't exist. Because they didn't understand the methods that we were using to present our designs, they were unable to internalize that until they actually saw the user interface; that was the only way they could comprehend what it was that we were trying to do.

**MP:** That's why you see such a big emphasis on use cases these days. Operational scenarios are popular because it's in the actual use that people internalize what it is that they're getting. Rapid prototyping is another technique--all of these techniques are ways of getting the user wrapped up into the system.

**DG:** Achieving this balance is a key problem--keeping it in the environment. Any descriptions of requirements that evolve should be characterized in terms of the environment so that they become more real. The entities that are identified need not be tangible, but they must be real in terms of the user, and I think it's important to keep a balance between what's real on the one hand and the need to consider COTS products on the other. Because COTS products will constrain the designer and ultimately perhaps the performance of the system. This is not explicitly communication, but it's really wrestling with ideas that have traditionally been discussed in terms of the "what" and the "why." Use cases, scenarios, and other similar strategies enable people to describe the system and how it works within the environment.

**SC:** Use cases came up for us recently. We had someone come in to talk about them. This person's organization had developed, I think, 156 use cases to describe its system. The use cases were stated in terms of processes that the system had to go through. We were sitting there with the person who had developed all the requirements, and also across the table were all of the program managers. He said "What I would like to do is hit on one of these functions and see what use cases apply." And the guy's jaw kind of dropped, because there was no mapping between the use cases, which were process oriented--how to do tracking, analysis, engagement planning--and the functions that were actually built into the system.

**AC:** From an organizational perspective, it's increasingly common for a customer for software and the supplier of that software to be geographically distributed. This usually means that they have to get together on a periodic basis to "sync up." During the formative stages of a project, when requirements are being developed, this interaction can be particularly critical, and interactions should be most frequent. However, the impediment of distance means that such interactions do not take place as frequently as they should, with the result that communications quality can be degraded. Phone and email communications just do not measure up when high bandwidth interactions are required. Clearly the quality of the requirements lays a foundation for the quality of the resulting software system, so there should be a great motivation to really work hard at effective means to communicate easily early during the early stages of such projects.

This is why I believe that collaboration technologies should be used to help groups develop requirements when these groups are geographically separated. These technologies are rapidly maturing and coming down in cost, so it is becoming

increasingly appealing to use them. They allow for a wide variety of types of interactions among individuals, and the automatic capture of session interactions provides a historical record of activities.

### **What are the “real” requirements for the system?**

**BP:** It seems that requirements can exist at different levels of granularity. How can software engineers manage these different levels and determine the “real” requirements of their systems?

**PP:** Let’s take the FAA system as an example. The FAA has a CD-ROM full of documents, that they call “functional specifications,” and these are all in terms of user functionality, such as “The user types these two letters and these two digits, and this is what will happen.” So in essence, they are the requirements for the system, and that document is maintained and is kept up to date. And, given the life-threatening nature of their systems, people do not make changes without good reason. But is that document really a “requirements document?” My guess is that it is not, at least at the level of “What are the requirements of the system?”

The requirements of the system are that it be able to maintain separation of aircraft, to maintain the safety of people as they’re flying en route and as they’re landing; it’s not that the controllers shall be able to type these two letters and these two digits and achieve this effect. I think that people lose sight of the real requirements, and they don’t see the value of investing the effort in maintaining currency between whatever the requirements document is and the code. Because while the design documentation and the architecture documentation may never change, they know they’re going to change the code. So there is a tendency to view everything else as overhead. That’s why you get the situation that we’ve all seen with systems that evolve, where the initial requirements document becomes irrelevant. We do not recognize that the requirements are recorded in a living document that must be maintained and that must survive and evolve throughout the life of the system. We just don’t do that, because it’s expensive and there’s no return on it; it doesn’t get another byte of code written.

**NM:** One interesting example is the use of the Patriot systems during the Gulf War. By keeping the system in operation much longer than was intended, accuracy was lost, and hence the anti-missiles didn’t go in the right direction. The change was a procedural one-- just take it down and re-initialize it periodically.

**PP:** Early versions of IBM’s AIX operating system had a memory fragmentation problem. After a couple of weeks of continual operation, they’d run out of virtual memory. Procedurally, you needed to reboot the system. But if you’re running on a 24-

hour-a-day, 7-day-a-week system, you do not want to reboot that system. Yes, you can get around these things, but I see confusion in the fact that people have these operational procedures documents but don't realize that there are also requirements on the overall system. We think of requirements that are system requirements or software requirements, but we don't consider that the man in the loop and other environmental factors really form this thing that is the system.

**MP:** Let me ask a question that builds on what you and Nancy have been saying. Should the user manual or the operator's manual be considered the real requirements specification for the system? For example, in the case of the FAA and the need for separation of aircraft, a requirement might be that there should be an alarm that goes off or some kind of blinking light that says these craft are getting too close together. Or the procedure for operating the system should say that when these two aircraft get too close together you should tell them to go in different directions or something. I know that there are non-technical requirements in there, but one of the things that I've encouraged folks to do is to make sure that their user manuals and what the code actually does are consistent with one another.

**PP:** But this still misses the point. Doing that gets you into the world of "We have 2000 requirements, and X number of 'shalls' in our document," and this is what you have to address. At some level, yes, you have to have that, that forms a contract. But is that something that's manageable or maintainable, or even something that I can elicit from people?

**MP:** Let me rephrase the question. Should anything be considered a real requirement that is, in essence, invisible to the user? A lot of times when we get into functional requirements, they're actually design statements, they're not really requirements. It's the requirements about what the system will do, what the user can expect to achieve, which I would hope would be captured in some kind of user manual or something like that.

**PP:** But there are cases where it is important to require things that may be invisible to the user, that may impinge upon the design. For example, if the FAA writes requirements only in terms of things that are visible to the user, they will have many different types of maintenance to do. If they can require that everybody runs on this operating system or uses that hardware, which is completely invisible to the user, they can save themselves a whole bunch of maintenance.

**MP:** I would characterize that as being part of the maintenance manual. Depending on what environment you're from, a lot of times there's a maintenance document that is part of the deliverable set, which includes the user manual, the operations manual, the maintenance manual...

**PP:** Yes, but in the world of COTS, you don't have a maintenance manual. You don't maintain the operating system. Periodically you will have to do operating system upgrades, but you don't maintain the operating system. I think that you would rather have one operating system and one set of procedures for how to reinstall that operating system than to have 20.

**NM:** I think that the place where this breaks down is that, again to use the FAA example, all of the operational procedures tell you what to do with aircraft with the assumption that you're tracking them correctly. But there's really a requirement of the system that you track all aircraft, and that might not appear in the user manual. And all of the software that's underneath it isn't captured in the statement that you're going to display.

**PP:** The user manual is all about how and what you can do with the information that you have available. But there are other requirements for completeness and currency of that information that probably don't appear.

### **Documenting requirements and maintaining architectural integrity**

**DG:** If I could put a little different spin on this, one of the key areas I see is getting the thing right. Many if not most of the errors that occur downstream, even in code, were requirements errors. So there are two elements of getting requirements right. One is something we've been talking about--eliciting them and capturing the real needs. Once that's accomplished and we have in fact described real needs, are they consistent and correct in and of themselves? That's the kind of analysis that we're focusing on. Are they complete, consistent, do they make sense, do they really accomplish the objectives that the stakeholders wanted? These questions must be addressed at the requirements level, especially given the data that shows that the cost of correcting an error that occurs downstream that was originally injected in the requirements is quite significant.

**MP:** How do you ensure that you keep a consistent record of what the requirements are? We've all observed that, over time, the code changes, the design changes, and the requirements change. So the point that Dave is making is absolutely valid, but the problem is that if we had a stable set of requirements that was unchanging, then the more rigorous techniques would be much easier to use. But when you have a dynamically changing environment, how do you deal with the consistency and make sure that everything is kept up to date? It's a human behavior issue as much as anything else.

**DG:** Right, and that's true regardless of the language that's used, whether it's formal language or natural language. But a potential solution is to capture as many of the requirements as appropriate and can be effectively accomplished in a formal language and then allow automation to handle a lot of the application-independent things that are

problematic and that require a lot of time but not a lot of intellectual activity on the part of humans. There are solutions that are tending in that direction, so that you have a line from requirements all the way through to code, and the ability to then do checking and analysis.

**SC:** At conferences that I have attended in the past couple of years, the emphasis has not been on the requirements phase. Even when they talk about analysis, they're talking about something that is much closer to design than it is to requirements analysis. The approach starts with the assumption that the requirements already exist, and elicitation is not the emphasis.

In the product line area, we tend to apply object-oriented analysis to the up-front elicitation; but again, it's in a rather narrowly focused area within a bigger problem. And once we give our results back to the client, there's a tendency to look at architecture development activities, and then never to return back to the requirements as they exist in that up-front analysis effort. The requirements will be evolved through the architectures, through change scenarios, or some other approach, which lets clients explore what they've got in hand, what kinds of changes the system is likely to undergo in the future, and to refine with those things in mind. They're not going to go back and elicit further requirements through a formal elicitation or management process. And I think that's likely to cause a problem downstream. At the end of implementation or even design, the clients don't really know what requirements they have.

This is similar to what Pat said earlier--a lot of people come in with these legacy systems and ask "What do we have?" So one of the things we can take back to other customers is to say, our experience has shown that, without formally tracking and managing requirements effectively, you're going to run into some of the same problems downstream as those you've encountered in the past. They may hit later than they do now, but they're still going to hit, and they're going to probably be as severe if not more so, because systems are bigger and more complex, and they affect more people than the old standalone systems used to.

The situation we see commonly is that people do some up-front analysis, and they have some elicitation and some requirements, but as they get into the architectural design, they never go back and update the requirements. So within months after the first set of requirements is complete, the requirements have already been overcome by the architecture, and they are never re-examined. To these people, requirements elicitation and management aren't even issues. The issues now are management and evolution of the architecture. There may have been some requirements notions up front, but now they're embodied in the implementation.

**NM:** I think that what happens is that, over time, people lose intellectual control, and they lose the sense of integrity of the architecture, and going back further, the set of requirements, and they start changing things in ways that they don't reflect on. They don't go back and say "Does this match the original vision for this?" Sometimes the vision isn't documented or expressed really well, and so you don't really know what the impact is. But many times, people just don't think about it; they think about the immediate problem at hand and not the impact that it has on the original vision of the system, which is why you see systems that become a mishmash even if they had a pretty decent set of requirements or a decent architecture to begin with.

### **Improving requirements engineering**

**BP:** We've had a very rich discussion about the problems associated with requirements engineering. In the time remaining, I'd like us to focus on some of the solutions in the various domains that you represent. Nancy, you mentioned an analysis method in the security area.

**NM:** In the area of survivable systems, we are developing an analysis method that we call the Survivable Network Analysis (SNA) method for software architectures. The SNA method is causing us to think about requirements for survivable systems. We find that most clients have not given a lot of thought to how to get the essential functions of their systems to survive an attack or a break-in and to continue to operate--how to recognize such an attack and how to recover from it. We call this the "three Rs": resistance, recognition, and recovery. For readers who are interested, we wrote up one of our survivable network analysis case studies (<http://www.sei.cmu.edu/publications/articles/ellison-survivable-systems/ellison-survivable-systems.html>).

We're finding that in doing architectural analysis, we are stepping back and making recommendations for requirements changes as well as recommendations for survivability. We think there is a lot of work that needs to be done here. Many users, from defense to finance, just haven't given that much thought to the vulnerabilities that will occur when they take their systems and distribute them over a network.

One of the things we're trying to do this year in our research is to think about intrusion scenarios that we develop and work with a client on. We'd like to have a canonical or relatively complete set of possible intrusions at the architecture level. In our initial work with this, we basically used our own intellect to figure out what the typical types of intrusions might be, but we'd really like to reduce that to something that could be used in a more analytical way, so the customers could feel that they had taken into account most of the bad things that could occur. So our focus this year is in part on that and in part on

taking into account risk analysis, because intrusions are not your normal occurrence, they're at the far end of the bell curve of activities relative to the systems. And in conjunction with that, we're looking at formalisms that support the work we're doing.

**BP:** In the sense that Pat and Dave have spoken of?

**NM:** There's certainly some relationship, yes.

**AC:** I mentioned this before but I think it's worth mentioning again. I don't believe behavioral properties of software systems are being addressed early enough. Too many systems have been built that don't address performance issues at an early stage, and consequently they get wiped out during operational tests. Likewise, complex system may not respond as intended because of unforeseen dependencies between loosely coupled components. I think simulation can help here, and in fact, a simulation model can become an integral part of the system's specification. It is almost certain that, after the requirements development phase is completed, requirements will change. Such simulation models can thus be used to investigate and address new issues as they arise--before they become significant problems.

**SC:** We've found that a lot of the people we're working with are taking the scenario approach or use-case approach. Many people are adopting this approach to move toward object-oriented development. But we still come back to the fact that they get to a certain point with the use cases and then kind of abandon further evolution and maintenance of the requirements. Instead, the class hierarchy or class structure, whatever the next stage is, is maintained because it's closer to the code or the implementation. It seems like there are instances though, where requirements are placed under very tight control, which would work if the requirements were true requirements. Often they degenerate into functional descriptions, and then you're really just writing in text what's happening in the design--the example mentioned earlier of hitting these two keys and having this result. Those are not really requirements, they're a description of what's going on the code.

**PP:** I don't think we have solutions in the COTS domain. We have, at best, advice, which is perhaps little more than a collection of things that you should know anyway. One of the more innovative approaches that we've seen is to get recognized consultants from the industry--from the marketplace--to help set the requirements for a particular product for a new system to be developed. That is, instead of getting only experts in the system to be developed, you also get marketplace leaders who can say, "If you go down this route, you will never be able to acquire anything that's COTS based." We are beginning to collect these sorts of successful approaches together. But are they repeatable? Who knows?

**BP:** Maybe these are sort of "best practices."

**PP:** Or suggested practices. Things like setting the levels of stretch of the requirements is a real problem. We see problems where you go into so much detail that you don't admit COTS solutions. That may be what you need to do. On the other hand, we've seen cases where to ensure the use of COTS-based systems, people set the requirements so abstractly that they get completely unsuitable systems offered in bids, and they have no way to reject them because they meet the stated requirements. There was no way to say that this one, which was the lowest cost alternative, cannot be employed, because its user interface, for example--which we didn't mention in our requirements--is completely unsuitable for our users, and they will reject it, and it will never get installed. So we're at that sort of level--we have advice, but not solutions.

With regard to engineering business processes, sometimes it's much easier to change the way you do business than it is to acquire a system that does the business that you're in. We saw one case where the DoD installation said, "It's much easier to adopt Oracle Financial and get that approved within the Air Force as a reporting mechanism for financial information than it is to try and acquire a system that reports on the existing reports that we're producing. But again, this is mostly anecdotal; there's no real "practice."

We talked earlier about the POSIX work. One of the things that we did with POSIX.21 was that, when the requirements were still in the formative stage, we did an analysis of the requirements and formalized them, and then reported the problems that arose from the formalization back to the requirements group and changed the requirements because of the problems that arose from the formalization. So this was a real requirements engineering elicitation. The entire working group believed then and still believes now that this was a valuable exercise and a really great way to home in on the requirements and get higher quality requirements faster than they could have gotten without them. Again, this is another piece of anecdotal experience.

**SC:** We did a case study about three years ago of CelsiusTech, a Swedish firm. They have an elaborate requirements database that they've maintained. It has allowed them to work with new customers to say, "Given this set of requirements that we can support in our architecture, how do you envision your systems?" So the new system requirements are developed in light of this requirements database that's been proven over 10 years or so in implemented systems. That's one instance of a managed, tracked, effective database of requirements.

**BP:** Is the requirements aspect specifically discussed in the CelsiusTech technical report [Brownsword 96]?

**SC:** Yes, but the report is a high-level overview, so the information about the database is probably only a paragraph or two. The counter example to that are things that come up in our architecture analyses for applying the Architecture Tradeoff Analysis Method (ATAM) or Software Architecture Analysis Method (SAAM). You get stakeholders together sometimes who haven't worked together, and we're taking an existing architecture and eliciting scenarios about how the system will change, how it will be used, and how it's envisioned. And new requirements pop up, or requirements that exist that were unknown to other people in the room. So that's an example of requirements that weren't developed, tracked, or fully understood, and now that the system has been built, things are suddenly coming to light. Again, there's no recommendation for how this could have been done more effectively, but at least bringing the stakeholders together and working through these kinds of analyses are helpful for bringing these things out early in the development--anything you can do to avoid sudden recognition when the system is being tested, as in Nancy's case of the people who didn't understand the system until they looked for data on the screen. So we hope that the architecture analysis brings the stakeholders together to recognize problems with requirements up front when the system is being designed.

So we encourage people to examine and consider requirements up front, and at least if there's a core requirement that's being used to define the evolution of the system, maintain that core even if the other details aren't maintained. If analysis turns up new requirements, we have to pull those back into this core. But I don't think we'll ever get to the stage where in the majority of cases, requirements are fully defined and explored as the system evolves.

**DG:** I think that what we really have been talking about here is making requirements engineering part of our systems engineering activities. That's really the essence of all of this. A lot of the problems we've identified are simply a failure to be engineering-like. We need a disciplined tracking practice as well as a technically sound approach to capturing and analyzing requirements.

**PP:** I agree. Engineering discipline takes a lot of time to develop—I just read Petroski's *To Engineer is Human* [Petroski 85] to understand the millennia spent “hacking” bridges, and everything else for that matter. So, it's not a surprise that software isn't engineered in the same way as the products of other disciplines. But that doesn't mean we shouldn't try and put it on a sounder footing.

## References

[Boehm 99] Boehm, Barry & Port, Dan. “Escaping the Software Tar Pit: Model Clashes and How to Avoid Them.” *Association for Computing Machinery (ACM) Software Engineering Notes* 23, 1 (January 1999): 36-48.

[Brownsword 96] Brownsword, Lisa & Clements, Paul. *A Case Study in Successful Product Line Management* (CMU/SEI-96-TR-016, ADA 315802). Pittsburgh, Pa.: Software Engineering Institute, Carnegie Mellon University, 1996. Available at <http://www.sei.cmu.edu/publications/documents/96.reports/96.tr.016.html>

[Gause 89] Gause, Donald C. & Weinberg, Gerald M. *Exploring Requirements: Quality Before Design*. New York, NY: Dorset House Pub., 1989.

[Petroski 85] Petroski, Henry. *To Engineer is Human: The Role of Failure in Successful Design*. New York, NY: St. Martin's Press, 1985.

The views expressed in this article are the participants' only and do not represent directly or imply any official position or view of the Software Engineering Institute or Carnegie Mellon University. This article is intended to stimulate further discussion about this topic.

The Software Engineering Institute (SEI) is a federally funded research and development center sponsored by the U.S. Department of Defense and operated by Carnegie Mellon University.

<sup>SM</sup> IDEAL, Interim Profile, Personal Software Process, PSP, SCE, Team Software Process, and TSP are service marks of Carnegie Mellon University.

® Capability Maturity Model, Capability Maturity Modeling, CERT Coordination Center, CERT, and CMM are registered in the U.S. Patent and Trademark Office.