JOURNAL OF OBJECT TECHNOLOGY

Online at www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2007**

Vol. 6. No. 1, January-February 2007

Common Requirements Problems, Their Negative Consequences, and the Industry Best Practices to Help Solve Them

Donald Firesmith, Software Engineering Institute, U.S.A.

Abstract

In this column, I summarize the 12 worst of the most common requirements engineering problems I have observed over many years working on and with real projects as a requirements engineer, consultant, trainer, and evaluator. I also list the negative consequences of these problems, and most importantly suggest some industry best practices that can help you avoid these problems, or at least fix them once they have raised their ugly heads. Although there is nothing really new here, these problems are well worth revisiting because they are still far too common, probably because the associated industry best practices are still far from being widely put into practice.

1 INTRODUCTION

From the standpoint of state-of-the-art best industry practices as opposed to state-of-the-average practice, requirements engineering is a relatively mature discipline with many well-known methods and techniques for identifying, analyzing, specifying, managing, verifying, and validating a system's requirements. But if that is so, why are there still so many defects in requirements specifications? Why are requirements mistake still a major root cause of many project failures in terms of significant cost and schedule overruns, failures to deliver all of the functionality specified, and systems that do not have adequate quality? Do we need new and radically improved requirements engineering methods, techniques, and tools? Or do we *just* need to put into practice the best industry practices that currently exist?

In this column, I will summarize a dozen of the worst most common requirements engineering problems that I have observed over many years working on and with real projects as a requirements engineer, consultant, trainer, and evaluator. These are the problems that I have seen most often and that have caused the most damage. For each of these problems, I list its major negative consequences, and most importantly suggest

Cite this article as follows: Donald G. Firesmith: "Common Requirements Problems, Their Negative Consequences, and Industry Best Practices to Help Solve Them", in *Journal of Object Technology*, vol. 6, no. 1, January-February 2007, pp. 17-33 http://www.jot.fm/issues/issue_2007_01/column2

some industry best practices that can help you avoid the problems, or at least fix them once they have raised their ugly heads. Although there is nothing really new here, these problems are well worth revisiting because they continue to occur far too often.

2 REQUIREMENTS PROBLEMS AND THEIR SOLUTIONS

The following are some of the most important of the many problems associated with how requirements engineering is practiced today:

1) Poor Requirements Quality

- **Problem**
  In practice, the actual quality of many specified requirements is poor. These requirements do not exhibit the accepted properties that should characterize all well engineered requirements. By poor requirements quality , I specifically mean that far too many 'requirements' specified in real requirements specifications are ambiguous, not cohesive, incomplete, inconsistent, incorrect, out-of-date, specified using technical jargon rather than the terminology of the user or business/application domain, not restricted to externally-visible behavior or properties of the system, infeasible to implement or manufacture, not actually mandatory (i.e., merely nice-to-haves on someone's wish list), irrelevant to the system being built, lacking in necessary metadata such as priority and status, untraced, in a form that is unusable to the requirements many stakeholders, unverifiable, and unvalidatable [Firesmith 2003].
  This problem arises because many requirements engineers who are inadequately trained, have inadequate access to stakeholders and other sources of the requirements, and who are given inadequate resources or authority to properly engineer the requirements. Other major causes of this problem are the prevalent myths that it is too costly, too difficult, and even impossible to produce good requirements, especially nonfunctional requirements. These myths are especially prevalent with regard to quality and specialty engineering requirements (e.g., availability, interoperability, performance, portability, safety, security, and usability), where there is still a prevailing but mistaken belief that it is impossible to specify these requirements in a verifiable form containing actual minimum acceptable thresholds. Not only is it possible to specify explicit quality thresholds, without them it is impossible for architects to know when their architectures are good enough and how to properly make architectural trade-offs between different quality requirements; without thresholds, it is also impossible for testers to produce proper quality tests and to generate associated test completion criteria.
- **Negative Consequences**
  Requirements engineering is the first engineering activity during which major mistakes can be made, and the negative consequences of these mistakes are felt

during all downstream activities such as architecting, design, implementation, and testing. Poor-quality requirements greatly increase development and sustainment costs and often cause major schedule overruns. As long ago as the early 1990s, it was well known that defects discovered once a system is fielded cost 50 to 200 times as much to correct as they would have had they been found during requirements evaluations [Boehm and Papaccio 1988], and these depressing figures have not changed significantly since then. As noted in [Wiegers 2001], "Industry data suggests that approximately 50 percent of product defects originate in the requirements. Perhaps 80 percent of the rework effort on a development project can be traced to requirements defects." Because these defects are the cause of over 40% of accidents involving safety-critical systems [HSE 1995], the unnecessary engineering of poor requirements has even been the ultimate cause of both death and destruction.

- **Solutions**
  Poor requirements quality is currently the number one problem in requirements engineering, and solving it will go a long way towards improving software and system development. Requirements engineers, stakeholders with whom they must collaborate, and requirements evaluators (e.g., inspectors and reviewers) need to be properly trained in the characteristics of good requirements including examples of both good and bad requirements, and they need to be taught how to tell the difference between them. Where practical, inspection should be used rather than (or in addition to) the less formal reviews and walkthroughs to verify and ensure that all of the requirements have the appropriate characteristics (e.g., unambiguous, complete, correct, mandatory, readable, etc.). You should use simple tools to identify inherently vague words being used in the requirements. Involve members of the architecture and test teams when verifying the quality of requirements to ensure that the requirements are feasible and verifiable. Ensure that the requirements engineers are enabled and required to collaborate with stakeholders until the requirements have sufficient quality. Finally, requirements engineers should rework or delete all requirements that lack the required characteristics of good requirements.

2) Over Emphasis on Simplistic Use Case Modeling

- **Problem**
  Currently, there is a major overemphasis on use case modeling as the only technique for identifying and analyzing requirements. Use cases seem to have become the hammer that makes every requirements problem a nail. Unfortunately, use cases are best suited for engineering functional requirements. Other techniques are much more appropriate for the engineering of non-functional requirements (NFRs), such as *interface requirements*, *data requirements*, *quality requirements* (i.e., requirements mandating minimum acceptable levels of the 'ilities' such as affordability, availability, interoperability, portability, reliability, safety, security, and usability), and architectural, design, implementation, and

configuration *constraints*.

Additionally, many projects only develop use case diagrams rather than creating sequence/swim lane diagrams to capture the normal and exceptional paths through the use cases. They also fail to use text to capture use case path preconditions, triggers, steps, and postconditions. Perhaps worst of all, only the primary 'sunny day' path through the use case is often developed. Unfortunately, there are usually many more exceptional 'rainy day' paths through the typical use case than 'sunny day' paths. In other words, what the system should do under normal circumstances may be captured, but not what the system should do when it can't do what it normally should do.

- **Negative Consequences**

  There are four major problems with the current use of use case modeling. Firstly, many NFRs are not being engineered at all, and those NFRs that are being engineered often end up as ambiguous, incomplete, unfeasible, and unverifiable goals rather than as true requirements. Secondly, producing incomplete use cases models results in simple stories rather than actual requirements. Thirdly, ignoring most if not all of the exceptional paths leaves much of the required behavior unspecified. Finally, if the requirements do not specify what the system should do under all credible combinations of inputs and states, then the developers will end up either making incorrect assumptions or ignoring possible cases, leading to systems that are unreliable, unstable, and unsafe.

- **Solutions**

  Requirements engineers should utilize all aspects of use case modeling to ensure that all credible paths through the use case are identified and analyzed. They should also utilize use case modeling as an identification and analysis technique, rather than as a requirements specification technique. They can use the use cases to identify, analyze, functional requirements. Inspection of the use case models will also help ensure that they are adequately complete.

  Requirements engineers should use appropriate requirements analysis techniques for the type of requirements being engineered. For example, they should use a risk-based approach built upon the analysis of vulnerable assets, attackers, threats, and attacks for engineering security requirements. They should also use checklists and a robust quality model that identifies and defines all of the major quality factors (i.e., 'ilities') so that no major type of quality requirement is accidentally overlooked.

3) Inappropriate Constraints

- **Problem**

  In practice, many requirements are not actually mandatory. Instead, too many of them are architecture, design, implementation, and installation/configuration constraints that are unnecessarily specified as requirements. Because stakeholders and/or requirements engineers sometimes incorrectly assume that a common way

to implement a requirement is actually the only way to implement the requirement, they confuse the implementation with the requirement and inappropriately specify *how* to build the system rather than *what* the system should do or *how well* the system should do it.

This problem is largely due to the fact that requirements engineers are not sufficiently qualified in the problem domain and specialty engineering areas (e.g., safety and security) to act as sources of the requirements and they are neither qualified nor authorized to architect, design, and implement the system. Similarly, many of the different kinds of stakeholders (e.g., users, customers, marketing, operators, maintainers, etc.), who are appropriate sources of the requirements, may be too caught up in the current system to envision how it could be significantly improved by new technologies and business process reengineering.

- **Negative Consequences**

  By unnecessarily specifying constraints, the requirements needlessly tie the hands of the architects, designers, implementers, and installers. This often prevents a better solution to the problem from being selected. Worse, it often prevents innovative solutions that can significantly improve the system and associated business processes, and eliminates an opportunity to differentiate both the system and its enterprise from the competition.

  Perhaps the canonical example of this occurs when specifying security requirements. Too often, instead of the specifying necessary levels of user identification and authentication, requirements engineers unnecessary specify the use of textual user identifiers and passwords, which are architectural constraints mandating specific security countermeasures. Mandating user IDs and passwords not only eliminates the selection of more modern countermeasures such as biometrics, smartcards, and digital signatures; it also mandates the use of countermeasures that have proven to provide the weakest level of security.

- **Solutions**

  The most important solution to this problem is to ensure that all collaborators in the requirements engineering process are aware of it. Looking for improperly specified constraints (i.e., specifications of *how* rather than *what* and *how well*) should be one of the most important items on the requirements inspection checklist. Finally, architects and specialty engineers should take part in the requirements evaluation process and question every requirement that potentially specifies an architectural or design decision.

4) Requirements Not Traced

- **Problem**

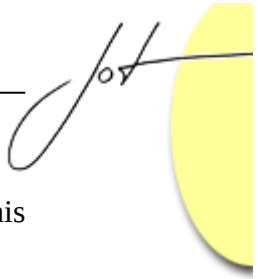
  Although the value of requirements tracing is widely recognized, is often mandated in contracts, and is included in many requirements engineering methods and training classes, many requirements are still not properly traced in practice. The sources of requirements (e.g., higher level requirements, other documents, and stakeholders) are not documented. Similarly, requirements are often neither

allocated to architecture and design elements nor to the test sets that verify them. On many projects, the very large number of requirements makes requirements tracing impossible to perform manually and difficult and resource-intensive to perform even with the modern tool support. The mapping from functional requirements to architecture and design elements is anything but one-to-one, and this mapping has become more difficult with the advent of modern technologies such as object, agent, and aspect orientation and the common use of middleware and other frameworks. Similarly, non-functional requirements are often implemented by many components scattered across an architecture. As a result, it is not at all uncommon for functional requirements to be traced to only the most important architectural elements and for non-functional quality requirements to not be traced at all.

- **Negative Consequences**
  This lack of tracing makes it difficult, if not impossible, to know the impact of proposed and actual changes, both to the requirements themselves and the architecture, design, and implementations derived from them. When changes occur as they will on any real endeavor, the requirements and both the upstream and downstream work products get out of synch as inconsistencies develop among them. Architecting, designing, implementing, and testing also become more difficult, expensive, and time consuming to perform.
- **Solutions**
  Ensure that requirements tracing is mandated in the contract and explicitly specified in the requirements engineering method. Also be sure to mandate and verify the tracing of all requirements, not just the functional requirements. Provide user friendly and scalable tool support for requirements tracing. Ensure management understands the negative consequences of not tracing requirements, and obtain support for proper tracing, including providing adequate resources to trace the requirements. Ensure that tracing occurs both early in the project development cycle as well as later during design, development, and maintenance. Finally, ensure that the evaluation of requirements tracing is a documented part of the requirements verification method.

5) Missing Requirements

- **Problem**
  Midsized systems often have hundreds of requirements and many large systems can end up with several thousand separate requirements, especially when one considers the derived requirements that are allocated to subsystems and their subsystems. Still, it is not at all uncommon for important bits of functionality to slip through the cracks. Given an iterative, incremental development cycle, these minor slips do not usually cause much harm so long as the omissions are identified and added to later builds or releases. In fact, many information systems often are specified to have numerous features that are not used by almost all users



and possibly not needed at all. Overlooking such requirements is not what this problem is primarily about.

The real problem is that many architecturally-significant requirements are accidentally overlooked. These are usually nonfunctional requirements, most commonly quality requirements specifying minimum acceptable amounts of some type of quality such as availability, interoperability, performance, portability, reliability, robustness, safety, security, stability, and usability. This typically happens because the stakeholders who are the source of the requirements often assume that such requirements are obvious and go without saying.

- **Negative Consequences**
  Because missing requirements are much harder to spot during requirements evaluations than incorrect or poorly-specified requirements, their absence is often missed until the system is integrated, undergoing operational testing, being manufactured, or being deployed. Worst case scenario, the missing requirements may not be discovered until the system is in use by hundreds, thousands, or an even larger number of users. Such requirements are typically much more difficult and expensive to fix then, especially if they are architecturally-significant requirements. For example, it is often difficult to add on performance, reliability, safety, and security to an existing architecture. Major system failures and accidents are often caused by missing requirements.
- **Solutions**
  Requirements engineers must actively elicit requirements rather than merely relying on stakeholders to tell them what they want. The requirements team should collaborate with specialty engineering teams (e.g., reliability, safety, security, and usability) and representatives from all groups of stakeholders when eliciting requirements. Mature methods and techniques (e.g., state modeling) should be used to ensure that the system knows how to handle all credible inputs and requests under all conditions. Instead of only drawing use case diagrams, use case modeling should include the production of sequence/swim lane diagrams and path descriptions that address all credible ‘sunny day’ and ‘rainy day’ paths with their associated preconditions, trigger conditions, and postconditions.

6) Excessive Requirements Volatility including Unmanaged Scope Creep

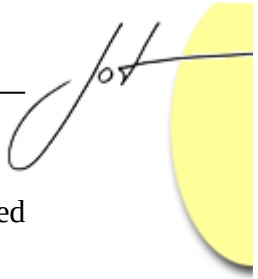
- **Problem**
  Because most systems have long development cycles and lifecycles, it is obvious that requirements will change. They must. Systems have to evolve as business needs change (e.g., with the advent of new competitors and new technologies). In spite of past heroic attempts to conform to strict waterfall development cycles, it is effectively impossible to freeze requirements in practice. This need to continually change requirements is a major reason why industry is adopting the use of iterative, incremental, and parallel development and life cycles. But changing a system’s requirements to meet the system’s stakeholders’ changing needs is not without its own problems. Stakeholders will want to

constantly add a few new requirements here and change one or two existing requirements there. But when this happens in an uncontrolled manner, you get the perennial problems of excessive requirements volatility and scope creep.

- **Negative Consequences**
  Unmanaged and unexpected changes to requirements can raise havoc with existing architectures, designs, implementations, and testing. Without a minimum amount of stability, developers cannot do their jobs and deliver new systems or increments to existing systems. The cycle of testing and fixing defects becomes endless.
  Scope creep almost always results from more requirements instead of less. Thus, it typically significantly increases the cost and time required to build new systems or versions of existing systems. Unfortunately, project budgets and schedules are often neither sufficiently flexible nor updated to remain consistent with the new requirements. This causes projects to rapidly go over budget and milestones to slip.
- **Solutions**
  The primary solution is *not* to chisel existing requirements in granite and prohibit the addition of any new requirements. Using a modern lifecycle to allow for requirements changes *is* a good idea. But changes to the requirements must be properly managed. For each release of the system, the requirements must be baselined and frozen at appropriate milestones within the development/update cycle. Baselined requirements should be placed under configuration control like any other major work product, and the impact of changes to these requirements needs to be determined before the changes are authorized to take place. Finally, budgets and schedules need to be updated whenever there is any nontrivial change to the requirements.

7) Inadequate Verification of Requirements Quality

- **Problem**
  This problem is not about the verifying whether the as-built system implements its requirements. Rather, it is about verifying sufficiently early in the development process whether the requirements have sufficient quality to avoid the many negative consequences resulting from poor requirements.
  Often, requirements are informally verified during small peer reviews and/or as a side effect of major ‘dog and pony show’ stakeholder reviews. While both reviews are somewhat helpful, they have not proven effective in identifying requirements defects.
- **Negative Consequences**
  Requirements defects that are not identified during the requirements engineering process will negatively impact all subsequent activities. When eventually discovered, these defects will be significantly more expensive and take



significantly more time to fix than they would had they been found and fixed during early requirements verification.

- **Solutions**
  When ever practical, evaluators should use inspection rather than (or in addition to) the less formal reviews and walkthroughs to verify and ensure that all of the requirements have the appropriate characteristics (e.g., unambiguous, complete, correct, mandatory, readable, etc.). Projects should develop and/or reuse checklists of the most common and damaging requirements defects. Requirements engineers and evaluators should use simple tools to scan the requirements and identify inherently vague words in the requirements. The requirements verification team should contain representatives from all major types of stakeholders, whereby multiple inspections are held with small, cohesive groups of stakeholders. The requirements evaluation team should contain members of the architecture and test teams to verify whether the requirements are feasible and verifiable. Finally, the requirements team should be authorized and mandated to rework or delete all requirements that lack the required characteristics of good requirements.

8) Inadequate Requirements Validation

- **Problem**
  A major task of requirements engineering is to have the stakeholders validate their requirements to ensure that the requirements completely and correctly specify their needs. Unfortunately, requirements are not always properly validated by their stakeholders.
  One root cause of this is that the requirements engineers often do not have adequate access to stakeholder representatives. This is especially a problem on projects where there are contractual and procedural limitations on the availability of stakeholders to validate the system requirements. For example, there may be one organization that elicitates stakeholder needs and produces an operational specification of user needs that is passed on to via an acquisition organization to the development organization, which must then produce the system's technical requirements. In this situation, there are two organizations separating the system's requirements teams from the system's stakeholders, making it difficult to get the requirements properly validated.
  A second root cause of this may be that the project's requirements engineering method may not include requirements validation, perhaps due to ignorance of the tasks comprising requirements engineering or a lack of resources to properly perform all of the requirements engineering tasks. Sometimes requirements validation is dropped due to a lack of stakeholder time, project schedule, or project funding.
- **Negative Consequences**
  A lack of proper requirements validation with the stakeholders typically results in requirements that are incomplete because they fail to specify important

stakeholder needs or they are incorrect because of misunderstandings between the requirements engineers and the stakeholders. The resulting system may then be unacceptable to major classes of stakeholders even if it has been verified by testing to meet its requirements. Fixing these problems later can have major negative impacts on cost and schedule, and some functionality may be missing upon delivery.

- **Solutions**
  Ensure that requirements validation is a fundamental component of any requirements method, one that will not be dropped the first time that project resources become scarce. Ensure that requirements validation is included into the project's schedule and budget as well as the schedules and budgets of the system's stakeholders. Finally, remove all unnecessary obstacles separating the stakeholders and the requirements team.

9) Inadequate Requirements Management

- **Problem**
  Many projects do not adequately manage their requirements. They store their requirements in paper documents or in simple spreadsheets. Different kinds of requirements are also stored separately in different media controlled by different teams such as the marketing team, the management team, the requirements team, and specialty engineering teams. For example, functional requirements may be stored in a requirements database, interface requirements may be stored in interface control documents, data requirements may be stored as data design definitions in one or more data dictionaries, security requirements may be stored in multiple organizational security policy documents, and other quality requirements may be stored in a supplementary requirements specification. Often, there is little support for access control to these requirements including limits on who has what kind of access (e.g., create, read, update, and delete). The requirements are often missing important metadata, such as priority, type, status, source, rationale, etc.
- **Negative Consequences**
  Requirements stored in paper form rather than in a requirements repository are difficult if not impossible to create, manipulate, and maintain. Scattered requirements are hard to find, sort, query, and maintain. Lack of access control makes it difficult to limit access to sensitive requirements and to achieve proper change control. Lack of centralized, automated management of requirements also makes it difficult to capture, analyze, and report requirements metrics (e.g., requirements stability, maturity, and completion).
- **Solutions**
  To deal with the large number of requirements and the constant changes to them, store the requirements in a database or the repository of a requirements tool. Store the requirements models and diagrams with or linked to the requirements. Store

all important attributes about a requirement (a.k.a., metadata) with the requirement so that they are easy to manage and maintain. Do not scatter different kinds of requirements; instead, keep them all in the same repository. Ensure that the requirements repository (and tool) supports access control, including prohibition of unauthorized access to sensitive requirements (e.g., proprietary information and classified data). If you need different kinds of requirements specifications for different audiences or purposes, generate them automatically from the requirements repository.

10) Inadequate Requirements Process

- **Problem**
  On many projects, the actual requirements method used is largely undocumented. It is often incomplete in terms of either missing or inadequately documenting important tasks, techniques, roles, and work products. The as-followed requirements engineering process is often inconsistently followed and significantly different from the as-documented requirements engineering method. The requirements engineering method is often based on a single technique (e.g., use case modeling) that is unfortunately intended to be used for all types of requirements, rather than having the requirements engineering method include appropriate techniques for engineering functional, interface, data, and quality requirements as well as for mandated constraints. Often documented in a requirements engineering plan, system/software engineering plan, or requirements standards and procedures documents, the requirements engineering method is typically much too brief (1 to 2 pages) and incomplete. The method used is often chosen because it was used more or less successfully once before by a member of the requirements team, rather than because it is appropriate for the engineering of the requirements of the specific system to be developed or updated. Another cause of inadequate requirements engineering processes is the widespread use of standard, generic, out of the can (or book) development methods, which do not meet the needs of the specific project. As with any other development process and discipline, one size does not fit all.
- **Negative Consequences**
  A poor as-documented method is enacted as poor as-performed processes that produce poor products, which in this case are poor-quality requirements and requirements specification documents. Inappropriate methods are inefficient and ineffective. When different requirements engineers and requirements engineering teams use poorly documented methods, they produce inconsistently specified requirements, which are difficult for architects, designers, implementers, and testers to use. Methods lacking of necessary detail cause the requirements engineers to waste time arguing over what to do and how to do it. They will also make unwarranted assumptions about how parts of the method should be performed.
  The use of a generic requirements engineering method often results in a mismatch

with the specific needs of the project. If the generic requirements engineering method is not properly tailored or if a project-specific method is not developed (e.g., constructed by selecting and integrating reusable requirements-related method components), then the resulting suboptimal method will not produce optimal results.

All of these subproblems and associated specific negative consequences ultimately cause budget and schedule overruns as well as the delivery of system with missing capabilities and added defects.

- **Solutions**

  Have an experienced requirements engineer and process engineer collaborate to ensure that the requirements engineering method is complete, incorporating all of the important method components including tasks, techniques, roles and responsibilities, and work products. The quality organization should also audit the requirements engineering process. Ensure that the method components are mature and have been successfully used on projects that were similar in size, complexity, and type and that developed similar systems. Ensure that the method components are properly documented, easily understood by their target audiences, and contain the appropriate level of detail based on the training and experience of the people who will use them.

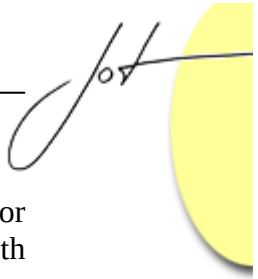
  Where practical, construct a project-specific requirements engineering method that meets your specific needs by reusing mature method components instead of either reusing a generic but inappropriate canned method or developing and documenting a requirements engineering method from scratch. For example, you may wish to consider using a commercial tool (e.g., RUP from IBM/Rational, which seems to be the most commonly mentioned process engineering tool in the software engineering community[1]). On the other hand, you may wish to consider reusing free, open source method components to construct your project-specific requirements engineering method, whereby the OPEN Process Framework Repository Organization (http://www.opfro.org) has the most extensive repository of free, open source method components including requirements engineering tasks, techniques, roles, teams, and work products.

11) Inadequate Tool Support

- **Problem**

  Many requirements engineers do not have or do not use adequate tool support when engineering their requirements. For example, many requirements engineers still use a requirements specification document as their combined requirements specification and requirements repository, while others use a simple spreadsheet or relational database table. Few requirements engineers use a real requirements

[1] The Rational Unified Process (RUP) tool is merely given as a popular example of a commercial process engineering tool; naming it as opposed to naming tools from competing vendors is *not* intended as an endorsement of any one tool over another.



management tool (e.g., Borland CaliberRM, IBM/Rational's RequisitePro, or Telelogic's DOORS[2]) that enables them to store individual requirements with their associated attributes (metadata).

Diagrams (e.g., UML use case diagrams, sequence/swimlane diagrams, and state charts) are a major part of most requirements models, and when it comes to drawing diagrams, it is hard to beat whiteboards. Still, the diagrams must be eventually being captured if they are going to be documented for later use. When it comes to requirements identification and analysis, many requirements engineers use simple drawing tools while others use CASE tools (e.g., IBM/Rational Rose or Telelogic's Rhapsody[2]) to draw diagrams. The requirements and their associated models and diagrams are often developed and stored in two (or more) different and incompatible tools. Traceability from the requirements in one tool to the architecture, design, implementation, and testing in two, three, or four additional tools is often not supported by the tools and must be maintained manually.

- **Negative Consequences**
  Many requirements models are not properly documented and stored to back up the actual requirements. It is extremely labor-intensive to manually produce and maintain a non-trivial amount of requirements. Without tool support, inconsistencies significantly increase and the documented requirements easily get out-of-date.
- **Solutions**
  Use a powerful, yet user-friendly, requirements *management* tool that enables the storing of requirements metadata. Use a powerful, yet user-friendly requirements *modeling* tool to capture requirements diagrams and associated text. Ensure that these tools support the configuration management of the requirements and their models. Where practical, choose an integrated toolset that supports traceability as well as the forward engineering of requirements through architectures and designs to implementations and tests and reverse engineering from these back to the requirements. When practical, develop scripts or other software that links the tools together if you cannot obtain a fully integrated or interoperable set of requirements tools, which is likely given the current state of the industry. Do due diligence when evaluating requirements and related tools, and beware of tool vendor marketing descriptions and promises.

12) Unprepared Requirements Engineers

- **Problem**
  There is a common myth held by certain managers that because requirements are usually specified using native languages such as English, then any reasonably literate person should be able to talk to a few stakeholders and write down what they want. The belief is that, unlike design and programming require specific

[2] As in previous paragraphs, these tools are merely used as examples; no recommendation or endorsement is intended.

technical experience and training, requirements engineering is a soft discipline that anyone can perform. Another myth is that domain experts (e.g., business analysts and marketing personnel) who understand the application domain, but who know nothing about requirements engineering can also magically become requirements engineers overnight. While these two myths are patently untrue, it is not uncommon to see people Peter-Principled into the position of requirements engineer without training in requirements engineering and without any experience or an apprenticeship to gain that experience.

Requirements engineering is often a position that is little valued by technical people, who do not understand that it is an engineering discipline in its own right with its own methods, techniques, and tools. In fact, being a good requirements engineer requires some of the same characteristics of a good architect. Both need to be able to have a big-picture viewpoint and be able to communicate well with non-technical people as well as technical people. Often, the position of requirements engineer is looked down upon as not having good prospects for career advancement. In general, it is not considered to be fun by most technical people, who mistakenly consider the role to be closer to that of management than technologist.

- **Negative Consequences**
  Requirements engineers without training, expertise, or motivation do not tend to understand and follow good requirements methods and therefore do not tend to produce good requirements. For such people, the job can be frustrating and a source of low morale and self-esteem. In such organizations, the position of requirements engineer becomes viewed as a no-fun, dead-end job for performers, a viewpoint that becomes a self-fulfilling prophesy. Thus, poor productivity and excessive staff turnover can result.
- **Solutions**
  Carefully select people with the right combination of training, experience, motivation, mindset, and people skills to be good requirements engineers. Provide them with significant amounts of training, including classes, conference tutorials, books, and journals. Apprentice beginners to more experienced requirements engineers. Then, formally give them the mandate including responsibility and authority to properly do their job. Ensure that others, including both management and the technical staff, understand the importance of the role they play in project success.

3 CONCLUSION

In this column, I have briefly described the twelve most important problems negatively impacting the engineering of requirements for software-intensive systems. For each problem, I have described some of its major negative consequences, and the most important things we can do to either avoid these problems or fix them.

Reading the previous descriptions, you may have noted that some of these problems are synergistically related. The bad news is that they feed off of each other so that if you have one of the problems, you are likely to have more and that some of their negative consequences have common causes. The good news, however, is that several of the solutions are also synergistically related. Applying one industry best practice to solve or avoid one problem will often help solve or prevent several other problems.

Although these may be perennial requirements engineering problems, they fortunately all have well-known industry best practices as solutions. Thus, our primary challenge is not to develop new and improved requirements engineering methods and techniques. Rather, it is to put into practice what many professional requirements engineers have been recommending for years. And that leads to a new problem; solving these twelve problems will take a considerable amount of consciousness raising, training, and management support. The bottom line is the following two questions:

- How many of us will put these industry best practices into practice?
- How many of us will continue to suffer the negative consequence if we don't?

REFERENCES

[Boehm and Papaccio 1988] Barry W. Boehm and Philip N. Papaccio, "Understanding and Controlling Software Costs," *IEEE Transactions on Software Engineering*, Vol. 14, No. 10, October 1988, pp. 1462-1477

[Firesmith 2003] Donald Firesmith, "Specifying Good Requirements," *Journal of Object Technology (JOT)*, 2(4), Swiss Federal Institute of Technology (ETH), Zurich, Switzerland, pp. 77-87, July/August 2003. http://www.jot.fm/issues/issue_2003_07/column7

[Wiegers 2001] Karl E. Wiegers, "Inspecting Requirements," *StickyMinds.com Weekly Column*, 30 July 2001, http://www.stickyminds.com

ACKNOWLEDGEMENTS

Many thanks go to my colleagues Peter Capell and Mary Popeck, who reviewed this column and provided helpful observations and recommendations.

Disclaimers

The Software Engineering Institute is a federally funded research and development center sponsored by the U.S. Department of Defense.

The views and conclusions contained in this column are solely those of the author and should not be interpreted as representing official policies, either expressed or implied, of the Software Engineering Institute, Carnegie Mellon University, the U.S. Air Force, the U.S. Department of Defense, or the U.S. Government.

About the author

Donald Firesmith is a senior member of the technical staff at the Software Engineering Institute (SEI), where he helps the US Government acquire large, complex, software-intensive systems. Working in industrial software development since 1979, he has worked primarily with object technology since 1984 and has written 5 books on the subject. During the last five years, he has developed the world's largest (1,100+ webpage), free, and open source informational website of reusable process engineering components for constructing project-specific development methods. Based on the OPEN Process Framework (OPF), it is located at http://www.opfro.org. Currently completing his next book on the engineering of safety- and security-related requirements for software-intensive system, he can be reached at dgf@sei.cmu.edu.