Editor: Suzanne Robertson ■ The Atlantic Systems Guild ■ suzanne@systemsguild.com

Developing the Requirements Discipline: Software vs. Systems

Regina Gonzales

Do you think of yourself as a software engineer or a systems engineer? What are the differences? And where in this mix does the requirements engineering discipline fit? Regina Gonzales considers history and cultural backgrounds and makes recommendations for the future.
—Suzanne Robertson

Clearly, two existing communities are establishing methods for capturing, specifying, and managing requirements. These communities are software engineering and systems engineering, and cultural differences exist between the two. Requirements work is too important to be pioneered by a mainstream software-centric or -ignorant viewpoint. Each community must continue to learn from the other for our understanding of the requirements engineering discipline to continue to grow.

Different beginnings

Systems engineering has been around for centuries, including Mesopotamia's water distribution systems in 4,000 BC and Egypt's irrigation systems in 3,300 BC. But what we call modern systems engineering began in the late 1930s (see Figure 1). The British formed an early multidisciplined engineering team in 1937 to analyze their air defense systems. The RAND Corporation established a system development division in 1955 to prepare SAGE (Semi-Automatic Ground Environment) system training programs. The first systems engineering textbook is Harry H. Goode's *Systems Engineering: An Introduction to the Design of Large-Scale Systems* (Univ. of Michigan Press, 1957). The first paper referencing requirements specification is W.O. Turner's "Estimation of Requirements Dial Telephone Central Offices" (*Proc. Operations Research Conf.*, Case Inst. of Technology, 1953). The coordination and melding of many established science and engineering disciplines distinguishes modern systems engineering. The effort required to coordinate the project becomes a complex problem itself. This explains the more management-oriented approach that systems engineers tend to take.

Software engineering began when writing software no longer required detailed hardware understanding. In 1954, Fortran became the first widespread language that allowed that to happen. Initially, software was developed by the engineers who designed the hardware. As mathematicians learned to program, the complex software field forked as a discipline, and most systems engineers began to consider it a system component (such as another gear).

Cultural filters

Daniel Bates and Fred Plog's *Cultural Anthropology* (Knopf, 1976) establishes a working definition of culture: "The system of shared beliefs, values, customs, behaviors, and artifacts that members of society use to cope with their world and with one another, and that are transmitted from generation to generation through

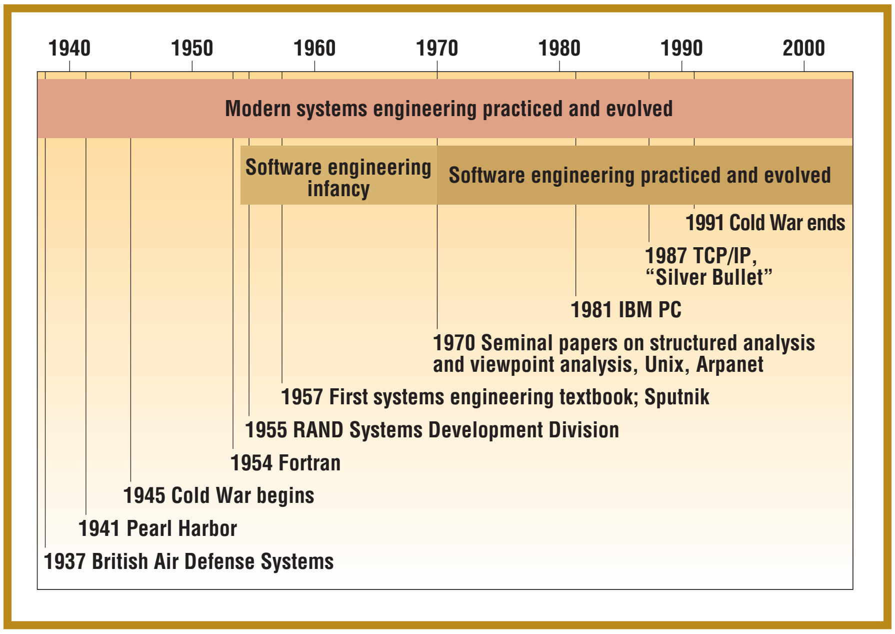


Figure 1. Systems engineering and software engineering history.

learning." From a cultural anthropology approach, we can examine some fundamental beliefs and assumptions, shared values, behavioral patterns (or customs), behavioral norms, and artifacts to gain insight into how the software and systems engineering communities approach the requirements discipline. These elements work together to form cultural filters that predispose our problem-solving approach (see Table 1).

So, we can see that software's complexity demands that the software engineers learn the same lessons the systems engineers learned, but from a different perspective. Mathematicians tend to create precise boundaries around a problem and make it as neat as possible so they can solve it. However, systems engineers don't usually expect a precise solution to a problem. Instead, they approach problems by selecting from a set of concepts based on trade-offs (equally technically and management-oriented) early in the project life cycle. On the other hand, the system engineering discipline has methods that engineers have used over the years to reinforce a "stovepipe" approach. These come from an approach based on requirements decomposition and allocation. While requirements are based on strategic need and rolled down, the functional allocation is based on historical precedence and is managed in stovepipes, such as the telemetry component, radar, and so forth. Using this decomposition approach, systems engineers regard software as a component rather than a complex system embedded and intertwined much like the human

Table 1

Software engineering culture versus systems engineering culture

	Software engineering	Systems engineering
Fundamental beliefs and assumptions	■ Engineers can reduce problems or problem solutions to a set of formalisms. ■ Precisely specifying and applying methods will produce equal results regardless of the people involved. ■ The software is the system.	■ Formalisms help, but communication is most important. ■ Collect, mix, and match methods and techniques and put them in your toolkit. Apply as appropriate; none offer any guarantees. ■ The software is a system component.
Shared values	■ Developing formalisms for a problem or domain is the most important step in solving the problem.	■ Understanding the people involved (including nonnegotiables) and their interpersonal dynamics is most important.
Behavioral patterns (or customs)	■ Precisely state or formalize the problem even if it requires narrowing the scope. ■ Model the problem using formalisms.	■ Develop a full understanding of a prospective solution's operational aspects, and use natural language to establish an agreement.
Behavioral norms	■ Avoid the messiness of systems in the real world, such as deployment, aging, poor physical environments, and chaos. ■ Be faddish—sometimes cultish—about new methods.	■ Apply a balance of management and engineering methods to achieve the desired outcome. ■ Be risk averse.
Artifacts	■ Artifacts include UML diagrams and user scenarios.	■ Artifacts include operational scenarios and abundant management documents, such as SEMP (systems engineering management plan), TEMP (test and evaluation management plan), risk management plans, and so forth.

nervous system. This misconception has caused enormous grief as recently as 1996, when European launcher Ariane 5 blew up about 40 seconds after takeoff. The tragedy occurred because the control software was reused and the instrumentation software that fed the control software was revised to accommodate more bits of data.

In this vein, if we examine problems in capturing, specifying, and managing requirements, we're likely to find cultural filters that are common to both communities. For example, both communities still have trouble determining the real problem when proposing a system because both are predisposed to jump to solutions instead of spending adequate time understanding the problem. To overcome some of this cultural bias, we, the requirements community, are borrowing from other disciplines' cultures; we use anthropology's ethnography technique, the medical field's art of diagnosis, and journalism's five imperatives to gain understanding before jumping to a solution.

Systems engineering's future

When I think about complex systems and how they've evolved over the years, I'm struck by the fact that matter still matters. System-level requirements are a first-order effect because the system (hardware and software) is what the human perceives. The software increasingly determines most complex systems' behavior. This presents a dilemma for many systems engineers who don't understand software, because systems engineering's roots are large military applications. A logical solution is to make software engineers the next generation of systems engineers, and, to a degree, that's happening. The problem, however, is that the need for generalists produces systems engineers (*generalists* being people who can technically coordinate multiple disciplines by analogy and understanding the "third-order" effects in their own discipline). Software engineers often don't have a broad engineering or science background, so they can't accomplish this generalist perspective.

Requirements are at the heart of systems engineering. Systems engineers play many roles (see Sarah Sheard's 1996 paper, "Twelve Systems Engineering Roles," *Proc. 6th Ann. Int'l Symp. Int'l Council on Systems Eng.*, Software Productivity Consortium), of which requirements owner and system analyst are principal. While systems engineers are still learning to apply more methodological approaches to elicit and analyze requirements, software engineers are learning to take the broader view necessary to develop system solutions. Both communities have matured in their understanding of the requirements engineering discipline; both are beginning to understand that capturing requirements requires understanding the problem before developing a solution; and both have found influences in other disciplines. Each community's culture necessarily influences the other.

To properly develop the requirements discipline, we must continue the dialogue and ensure that we're aware of strides to formalize standard systems engineering approaches and generalize software engineering approaches to capturing, specifying, and managing requirements. An example of the former is the new SysML UML Framework for systems engineering. It formally defines the concepts of a requirement and trace links. An example of the latter is the CMMI, which now casts a wider net on requirements based on the systems engineering community's influence.

Acknowledgments

Sandia is a multiprogram laboratory operated by Sandia Corp., a Lockheed Martin Company, for the US Dept. of Energy's National Nuclear Security Administration, under contract DE-AC04-94AL85000.

Regina Gonzales is a systems engineer and a program integration engineer in the Weapons Systems Engineering Center at Sandia National Laboratories, and she also teaches systems engineering and requirements engineering for New Mexico State University. Contact her at rgonzal@sandia.gov.

Elicitation Technique Selection: How Do Experts Do It?

Ann M. Hickey
ahickey@uccs.edu

Alan M. Davis
adavis@uccs.edu

Department of Information Systems, The University of Colorado at Colorado Springs

Abstract

Requirements elicitation techniques are methods used by analysts to determine the needs of customers and users, so that systems can be built with a high probability of satisfying those needs. Analysts with extensive experience seem to be more successful than less experienced analysts in uncovering the user needs. Less experienced analysts often select a technique based on one of two reasons: (a) it is the only one they know, or (b) they think that a technique that worked well last time must surely be appropriate this time. This paper presents the results of in-depth interviews with some of the world's most experienced analysts. These results demonstrate how they select elicitation techniques based on a variety of situational assessments.

1. Introduction

The success or failure of a system development effort depends heavily on the quality of the requirements [1]. The quality of the requirements is greatly influenced by techniques employed during requirements elicitation [2] because elicitation is all about learning the needs of users, and communicating those needs to system builders. How we select an appropriate elicitation technique out of the plethora of available techniques greatly affects the success or failure of requirements elicitation [2]. We believe that requirements analysts who have extensive experience (and are considered to be *masters* of elicitation by most) seem to have the ability to select appropriate elicitation techniques on a regular basis. Since most practicing analysts have less experience and are more journeyman than master, it is no surprise that over half the products created by the software industry fail to satisfy users' needs [3]. If we could improve the average analyst's ability to select elicitation techniques, we will most likely improve our record of successful products. Our industry should find ways of transferring these experts' knowledge of elicitation technique selection to the less experienced. This paper's mission is to begin that process by assembling and reporting in one place the elicitation technique selection processes employed by some of the experts.

Before beginning that process, however, it is important to recognize that unsatisfactory performance by practicing analysts could be caused by a variety of conditions. The poor performance could be (1) unrelated to elicitation techniques, (2) caused by lack of effective elicitation techniques [4], or (3) by availability but poor use of effective elicitation techniques. If the latter is true, and effective elicitation techniques do exist, then our product failures may be attributable to some problem relating to the skills of the practicing analysts. For example, perhaps technology transfer has not been taking place with the fault lying with researchers, or the techniques' inherent complexity, or insufficient time, or the analysts' lack of interest in new techniques [5]. Or perhaps, analysts do know of the existence of elicitation techniques but do not know how to apply them, or they know how to apply the techniques, but fail to understand *when* to apply them. In all likelihood, it is a combination of these conditions that cause projects to fail. This paper applies to conditions where analysts do not know how or when to apply elicitation techniques. In particular, given that appropriate elicitation techniques are available, given that we believe that the best analysts are likely to be innovators or early adopters [6], and given that we believe that more experienced analysts are more successful, what is it that these analysts are doing that can be captured and conveyed to less experienced analysts to improve their performance? By assembling the combined wisdom of the most experienced analysts in one place, we aim to improve requirements elicitation practice, and thus raise the likelihood that future products will meet user needs.

2. Related Research

Many articles [7, 8, 9] and books [10, 11, 12, 13, 14] describe *a* way to perform requirements elicitation. This is logical since so many practitioners are looking for a simple recipe for success – the *silver bullet* [15] that will solve all their elicitation problems. However, consensus exists that one elicitation technique cannot work for all situations [4, 16, 17, 18, 19, 20]. Therefore, almost all general requirements books [11, 16, 17, 21, 22, 23, 24, 25] and some articles [26, 27, 28] describe multiple requirements elicitation techniques.

Some writings, e.g., [16, 17, 21, 23, 24, 25, 29, 30], provide limited insight into *when* an elicitation technique might or might not be applicable. Maiden and Rugg [18] have performed the most extensive research concerning the relationship between conditions and elicitation techniques, but the number of elicitation techniques analyzed was quite limited, and no attempt was made to assemble the collective wisdom of the most experienced analysts. As far as we know, no research has been done to date concerning how experts select elicitation techniques. And no author has tried to compare and contrast the advice from experts using common terminology.

3. The State of Requirements Elicitation

Requirements elicitation is the means by which analysts determine the problems and needs of customers, so that system development personnel can construct a system that actually resolves those problems and addresses customers' needs. Elicitation is an iterative process [31]. At any moment, conditions cause the analyst to perform a step using a specific elicitation technique. The use of that technique changes the conditions, and thus at the very next moment, the analyst may want to do something else using a different elicitation technique. The result of elicitation is a list of candidate requirements, or some kind of model of the solution system, or both.

Requirements elicitation is conducted today in a variety of contexts. For example, organizations that create software for mass market sale, perform elicitation while doing market research [32]. Meanwhile, the responsibility of elicitation in organizations that either create custom software, or customize a base of software for sale to a single client, tends to falls on an interdisciplinary team representing the customer and developer. Finally, in IT organizations that build custom software (or procure off-the-shelf systems and produce glueware) for use within the same company, analysts serve as a bridge between the company's IT and operating divisions. In all cases, the responsibility of the individual doing elicitation is the same: to fully understand the needs of users and translate them into terminology understood by IT.

As can be seen, elicitation is performed in a wide variety of situations, which span many dimensions representing various combinations of participants, problem domains, solution domains, and organizational contexts. It is also performed in a wide variety of ways, e.g., interviewing [33], collaborative workshops [34], prototyping [7, 35], modeling [10, 21, 22, 36], and observation [30]. It is this relationship between the detailed characteristics of such situations and elicitation technique selection that we are concerned with.

The requirements elicitation field has benefited from the presence of many expert consultants, who have had extensive experience in many projects, and who make themselves available to organizations endeavoring on a new or continuing systems effort. These individuals bring with them both successful and unsuccessful experience on many projects, and either help perform elicitation, or advise analysts on how they should perform elicitation.

If involvement by individuals like those described above does improve a project's likelihood of success (a conjecture that we and their customers firmly believe but that has not necessarily been proven to be true), then it would behoove us all to better understand how such people approach elicitation.

4. Overall Research Program

Given the limited research and theory regarding elicitation technique selection, we have chosen a qualitative research approach [37] using three primary qualitative information-gathering methods (participation in the setting, document analysis, and in-depth interviews) [38] to provide the data needed to discover situational technique selection theory [37, 39]. Figure 1 shows the 7 research phases necessary to achieve our final research goal. First we analyzed articles and books to create lists of potential situational characteristics and available elicitation techniques. The next phase is underway: to gather expert opinions about how these (and perhaps other) situational characteristics affect the selection of these (and perhaps other) elicitation techniques. We are gathering these opinions using three methods:

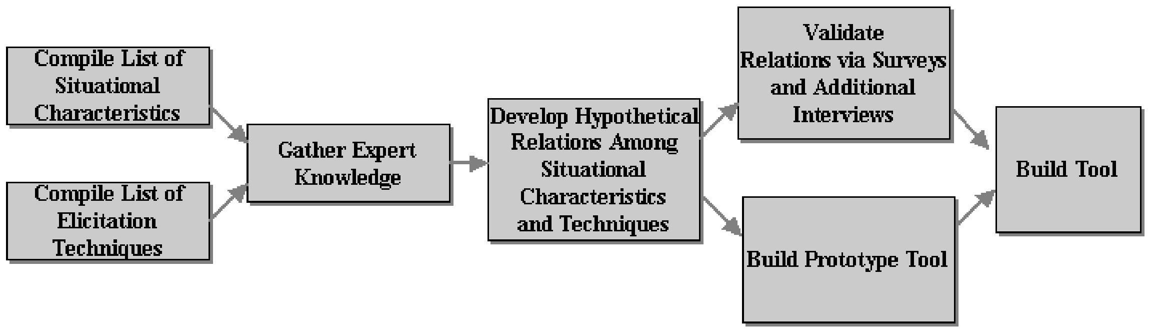

Compile List of Situational Characteristics
Compile List of Elicitation Techniques
Gather Expert Knowledge
Develop Hypothetical Relations Among Situational Characteristics and Techniques
Validate Relations via Surveys and Additional Interviews
Build Prototype Tool
Build Tool

Figure 1. Overall Research Program

1. *Participation in the Setting*. In some cases, our own experience, based on a combined total of 40 years leading and participating in requirements elicitation in industry, has shown repeatedly that certain situations demand the use of certain techniques. For example, when we as analysts have little experience in a domain, we tend to start be seeking knowledge from a subject matter expert. In other cases, the knowledge borders on common sense, e.g., if there are multiple customers who may not know of each other's existence, then do not gather them together in a collaborative workshop.
2. *Document Analysis*. We have found preliminary information about how situations affect the selection of elicitation techniques in writings by Davis [21], Hudlinka [26], Kotonya and Sommerville [16], Laueson [23], Leffingwell and Widrig [24], Macauley [17], Maiden and Rugg [18], and Wiegers [25].
3. *Interviews with Experts*. In-depth interviewing is ideally suited for gaining understanding of people's behavior, its context, and the meaning they make of that behavior [40, 41]. Johnson [42] states that it is likely the best approach "where the knowledge sought is often taken for granted and not readily articulated by most" [p. 105] as it is in this case where experts often rely on tacit [43] knowledge to select elicitation techniques. These interviews are the subject of this paper.

Based on the information gathered from these sources, we plan to refine the lists of situational characteristics and elicitation techniques, and develop tentative relationships among the situational characteristics and the techniques. We then plan to validate these relationships through further interviews and surveys, and finally capture the results in a tool, which can be used by less-experienced analysts who wish to behave more like experts.

5. Research Method

This paper reports the results of interviews with nine of expert analysts[1]: Grady Booch, Larry Constantine, Tom DeMarco, Don Gause, Tim Lister, Lucy Lockwood, Suzanne Robertson, Karl Wiegers, and Ed Yourdon[2]. (In this paper, we will not associate specific opinions with any individual[3].) These individuals were selected using our a priori research design [41], which required elicitation experts, and purposeful sampling [40] to provide the broadest possible coverage of elicitation techniques, domains, and situations. They represent a subset of the list of thirty experts we assembled using the following criteria: (a) at least five year's experience performing analysis or elicitation in industry as a consultant or employee, and (b) author of a major book or many articles on requirements. Combined, the nine individuals have had over 225 years experience analyzing requirements on more than 500 projects. Six experts provided more detailed information for approximately 200 of their projects. These projects spanned many domains: 16% were embedded systems, 61% were business information systems, 10% were government systems, and 13% were systems software. Of these projects, 36% were intended for external sale and 60% were intended for internal use (4% unknown). Finally, 87% were mostly custom development and 8% were primarily commercial off-the-shelf (5% unknown). All considered themselves to be primarily "doers" as opposed to "thinkers" or "researchers." All have had extensive elicitation experience in multiple countries.

		Interview Location		
		At Subject's Office or Home	At Interviewers' Office or Home	At Another Location
Number of Participants	Two Interviewers with One Subject	2	3	1
	One Interviewer with One Subject	0	0	1
	Two Interviewers with Two Subjects	2	0	0

Figure 2. Interview Locales and Participants

The interview goal was to understand how each expert performs elicitation. Interviews were held from August through November 2002. Figure 2 summarizes the various interview settings. In all cases, the authors took extensive notes, and as many recommend [40, 42], all interviews were audio-recorded. Each expert completed a brief pre-interview questionnaire summarizing their elicitation experience, the types of projects on which they had worked and the elicitation techniques they were familiar with and had used. Recommended by Seidman [40], these 'focused life histories' enabled us to compare our interviewees to our desired sample. The expert's responses also served as the basis for follow-up questions. A flexible interview protocol was developed to guide the interviews, facilitate time management, and ensure all key issues were addressed [41]. Each interview started with:

1. A short period of informal social discussion.
2. A brief introduction to our research goals, definitions, research approach, and interview style.
3. An open-ended query to the interviewee along the line of "so, can you tell us how to perform elicitation?"

[1] Most practitioners and researchers consider these individuals to be experts. However, almost every one of the nine refused to acknowledge that they were indeed *experts*. They agreed they had much experience, but felt that they were also learning every day how to perform elicitation effectively. To be honest, we might have distrusted anybody who claimed to really be an expert, and didn't need to learn any more!

[2] In the future, we intend to expand this list of experts to include a greater mix of occupational and regional representation.

[3] As part of this, we have used the pronoun "he" for all references to the analysts. We do not mean to imply the genders of the individuals.

What ensued differed significantly from interview to interview. In response to our open-ended question, some of the experts told us "stories" of how they performed elicitation in a variety of situations. Others told us *the* way they always performed elicitation. And still others responded with a description of the conditions they consider when deciding upon an optimal approach.

- When the interviewee told us stories, we carefully listened for aspects of the stories that might trigger particular elicitation paths. When we suspected this, we asked the interviewee for verification by either asking directly, "Was it *this* situation that inspired you to do that?" or the contrapositive, "If *this* situation were not true, would you still have chosen to do that?"
- When the interviewee told us *the* way he always performs elicitation, we would take careful notes. Then we'd pose variations to the interviewee's assumptions, and invariably their response would be something like "Of course if *that* were true, I wouldn't do elicitation the same way." We thus discovered that although some experts think of themselves as practitioners of just *one* means of performing elicitation, they really have a *default* means of doing elicitation, and anomalous conditions result in alternative approaches.
- When interviewees told us conditions under which they would take one route or another, we listened carefully. In our opinion, these individuals were the most self-aware and process-aware of the group.

To insure that we received some normalized data, we also posed the same "situations" to all the interviewees, and asked them to describe how they would approach the situation. These situations will be described in section 6.2.

We independently analyzed the interview results. An open coding method [37, 39] was used; every elicitation technique and situation mentioned by the experts was highlighted, not just those included in the initial lists created in steps 1 and 2 of our research. We then compared our respective analyses to validate our observations and summarize the lessons learned from the expert interviews. These results are reported next.

6. Lessons Learned

We report on the results of the interviews with the nine experts, categorized as follows:

- *When to Use Techniques*. These are the insights from the experts concerning conditions under which *they* would select a technique. The ideas are summarized by technique. This is the primary focus of the research.
- *Normalized Situations.* As previously mentioned, we asked each of the experts to analyze a subset of the same four situations. Section 6.2 presents the experts' approaches in each case.
- *Other Useful Information.* We also learned other related information, which we report in section 6.3.

6.1 When to use techniques

This section summarizes guidance explicitly stated by the experts on when to use elicitation techniques. Whenever counts are provided (e.g., five of nine), the reader should assume the other experts did not mention that technique, not that they necessarily disagree with the others.

Collaborative Sessions[4]. Three of the experts stressed the effectiveness of gathering multiple stakeholders in a single room to conduct a collaborative session. One stated that it should always be done; and that not even geographic distribution of the stakeholders should be a deterrent – just use today's technology to conduct the group session in a distributed manner. Another said that such sessions should be held when there is general belief that "missed opportunities" exist; however he also acknowledged that this is likely to be all or most of the time. He also stated how essential such meetings are when a large, diverse, and autonomous set of stakeholders exists. A third emphasized the power of creativity exercises in collaborative sessions to aid envisioning innovative future systems, breaking the constraints of current system knowledge that would dominate interviews or observations. In general, it appears that collaborative sessions are seen by most to be a standard or default approach to eliciting requirements.

Interviewing. One expert interviews to gather initial background information when working on new projects in new domains. Another uses it whenever heavy politics are present, to ensure that the group session does not self-destruct. Yet another uses it to isolate and show conflicts among stakeholders whenever he has 2-3 days available to meet the stakeholders. When senior management has a dream, but the employees consider him crazy, one fixes the problem by interviewing subject matter experts and visionaries. Another said that interviews with subject matter experts are essential when the users and customers are inaccessible. In general, it appears that interviews are widely used, but primarily to surface new information, or to uncover conflicts or politics.

Team-Building is a second-order elicitation technique in that it does not directly surface requirements. Instead, it is any of a wide variety of synthetic group experiences that help build communication and mutual trust among the stakeholders so that later first-order elicitation techniques become more effective. Four emphasized the need to address the building of teams. Three of these called specifically for team-building exercises prior to conducting any elicitation involving stakeholders whenever the team has not worked together in the past or there is good reason to believe that strong differences of

[4] Many terms are used to represent this concept, or to represent examples of this concept, including joint application development (JAD), brainstorming, group sessions, and so on.

opinions exist. One recommended under the same conditions that face-to-face meetings be conducted in lieu of using technology to support a distributed meeting. There seems to be consensus that effective elicitation requires teamwork, and when not present, team-building exercises are important.

Ethnography. Five out of the nine experts highlighted ethnographic techniques as extremely effective. In particular, three implied that observation of users should always be done when they are available and there is an existing system. Another thought observation is a great way to gather requirements when users are too busy to be involved in interviews, group sessions, or questionnaires. One uses observation to assess political and power relationships when working in new organizations. All in all, many of the analysts seemed to acknowledge that stakeholders should be observed when feasible.

Issues List. Two of the experts emphasized the need to maintain a list of outstanding issues on the side (so that the issues are not forgotten and can be re-surfaced later). Regardless of what technique is being used, new issues are simply appended to the list as they arise. This enables the team to stay focused and not follow the tangent induced by the new issue. One of these recommended that every meeting end with the assignment of open issues to team members. Although only two mentioned issues lists per se, we suspect that most of the analysts actually employ them. We will need more data to verify this.

Models. Eight of the experts mentioned the critical role played by models, such as data flow diagrams (DFD) [10], statecharts [44], or UML [45], in elicitation. Of these, five had favorite models; for example, one always uses time-ordered sequences of events to capture a typical interaction between the system and the systems or people it interfaces with[5], one always uses data flow diagrams, two *usually* used data flow diagrams, and one constructs user role models and task models. All three of the DFD users expressed specific caveats concerning the use of DFDs. One emphasized the need to create the DFD on a white board as the result of collaboration, and felt pretty strongly that the purpose for any model is to help the thought process, not serve as final customer documentation. The other two emphasized the need to build DFDs bottom up based on events as defined by essential systems analysis [46]. Of these last two, one tends to model the current business rules (i.e., the "current model" as defined by classical structured analysis [10]), while the other tends to model the new or proposed system. Three emphasized the need to employ multiple models as a means to better understand customers' problems. Three of the analysts were able to state conditions under which they would use certain approaches. In particular, one said that all models should be built collaboratively when customers or users can be assembled. Another said that he builds a data model, probably ER diagrams [47], for database-intensive applications or systems to be implemented using object-oriented approaches. And the third said he uses data dictionaries whenever there are multiple, diverse, and autonomous stakeholders. Finally, one of the analysts cautioned to use only those models that the stakeholders find palatable. In summary, analysts seem to rely on models in almost every situation. Although historically, modeling was used as *the* elicitation technique, more and more analysts are now seeing modeling as a means to (a) facilitate communication, (b) uncover missing information, (c) organize information gathered from other elicitation techniques, and (d) uncover inconsistencies.

Questionnaires. Surprisingly, very few of the experts mentioned questionnaires or surveys. One limited his use to problems that were fairly concrete. Another focused on market research surveys as an aid in understanding external customer needs.

Data Gathering from Existing Systems. When requirements are being gathered for a new system to replace an existing one, one suggested that "click counts" be collected. Another warned to *not* over-analyze the existing system for fear that the new system will become too constrained. Another suggested performing an "archeological dig." And a fourth recommended meeting with current end users and to do a value analysis. None of the experts appeared to use this as a primary method.

Requirements Categorization. Three of those interviewed used the Volere template [48] as a guide to ensure that all categories of requirements are gathered. Some experts stated opinions concerning other dimensions of requirements categorization. For example, one categorizes requirements as essential, expected and gee-whiz. Another sees management as a threat to success and thus wants to tag requirements that he calls "management fantasies." One feels it is essential to purge the incoming statements of all stakeholder wants, resulting in a list only of needs. Another tries to eliminate symptoms, instead striving for underlying root causes. Two others like to make the system boundary explicit early in elicitation so that all parties know what is within scope and what is not.

Conflict Awareness and Resolution. When conflicts arise within a particular class of stakeholder, one of the experts defers the resolution to a single spokesperson for the class, rather than directly addressing the conflicts himself. Two suggested that before any elicitation is performed, make sure that you understand the power structure, the politics, and the political camps. Perhaps the most insightful, yet radical, idea was spoken by one of the experts who said that if you think you have a project

[5] Many terms are used to represent this concept, or to represent examples of this concept, including use cases, scenarios, stories, stimulus-response sequences, storyboards, and so on.

without conflict, you obviously don't understand the problem; he also points out that the more stakeholders involved, the more conflict will occur.

Prototyping. Only two of the analysts interviewed mentioned prototyping as an elicitation technique. In particular, one suggested that you should not do rapid prototyping unless you really believe it will be rapid. The other suggested prototyping only when there is mutual trust. We had expected more broad reliance on prototyping. Perhaps this is the result of the small sample.

Role Playing. When key stakeholders are inaccessible, one of the analysts recommended the use of surrogates, where non-stakeholders play the role of stakeholders.

Formal Methods. Only one of the analysts mentioned formal methods, and his comment was to never use them during requirements even for safety-critical systems. His reason is that they distance stakeholders from the process. He did however acknowledge their effectiveness in later stages of system development.

Extreme Programming. Extreme Programming [49] calls for little up-front explicit elicitation, and replaces it with an omni-present customer, co-located with the development team, who can answer questions that arise during development. Only one analyst interviewed recommended this approach, and he suggested that it be used when the domain is undergoing enormous and constant flux.

6.2 Normalized responses

In this section we will convey the approaches taken by the experts when presented with four specific situations. These four cases were selected based on the following criteria: (a) should be stated in a relatively vague manner, just like real problems, (b) should be about a problem not a solution, (c) should be such that the solution would likely involve a large software component, but would not necessarily be exclusively software, (d) should have no obvious solutions (when solutions are obvious, the choice of elicitation approach likely makes no difference), (e) should represent a variety of domains, and (f) should be outside the usual domains of most of the experts (otherwise, the expert could simply select whatever technique they "normally" use, without giving it more thought). Obviously, these four cases represent a small sample of the infinite number of possible cases that analysts face, but they sufficiently explore the richness and variety of the approaches taken by the analysts.

In order to hide the identities of the experts, each case's responses were randomized. For the same reason, we have neutralized the terminology, using our common vernacular rather than specific terms used by the experts.

When the experts were presented with the following situations, each made different assumptions about some of the conditions. Furthermore, some described just their first step, while others described two or more steps.

Case 1 Statement: "I am CEO of a large transportation conglomerate. Our constituent companies are in every aspect of transportation including aircraft, trucking, ocean shipping, automobiles and train service. We recently learned that there is a large demand for a new capability – transportation of people from New York to Tokyo in under 30 minutes. You are an expert on analyzing problems. Can you help me better understand how our company can address this need?"

Case 1 Expert Approaches: Six experts analyzed this case. All focused in on the seemingly unrealistic 30-minute goal. Three explicitly addressed the generality of the goal, seeking specific goals and constraints, identifying criteria for success, or defining the business case. The fourth focused on the goal from a reverse perspective, seeking "pain points." The last two implicitly challenged the goal by turning to the market and transportation experts to explore/identify possible solutions and, for the last, analyzing the sensitivity to value. Details of the experts' approaches follow.

A. I would immediately wear my "skeptical hat" and try to learn the constraints and goals. Why 30 minutes? Is 4 hours acceptable? When is it needed? Next year or is 25 years OK? What other similar projects are underway? I guess I'd attempt to create a context diagram (the highest level of a DFD), but I wouldn't have much faith that it would work; after all, what are my "edges?" I suspect I would be on a long journey just to ascertain the project's goals.

B. First I would define the criteria for success; otherwise I will not know if I succeeded. Then I would do brainstorming with all the stakeholders on the question of "Why are we doing this?" Assuming we can justify continuing, I would create a context diagram to make sure we know the system boundaries. Next, I would define all the external constraints such as schedule, budget, and laws of physics. Then I'd solicit proposals from key individuals to learn all the various general approaches to solving this problem.

C. I'd start be asking some general questions like "Why do it?", "What is the customer value?", and "What are the *business* requirements?" Assuming that I received adequate answers, I would work closely with the "project visionary," preferably the executive with the checkbook. To understand the constraints, I would interview subject matter experts. Next, I'd identify all classes of stakeholders. If representatives from each class are willing to participate, I'd conduct a group session; otherwise, I'd do more interviewing.

D. I would start by interviewing a variety of stakeholders. I'd ask them about their "pain points" (i.e., what keeps them awake at night). I would then try to understand the political climate and find the centers of power. I'd collect these centers of power in one group session and elicit scenarios. We would simultaneously work on the product architecture. I would also maintain a "risk list" of the biggest pending problems. As we proceed, we all would watch for architectural patterns, especially those that fill a gap between users and the system.

E. I would start by bringing in experts in each type of transportation system. I'd have a group session with all of them to explore possibilities.

F. Since technology needs to be found to do this, I would first find existing sources. Then, I'd focus on identifying the underlying requirements and performing a value analysis to assess the sensitivity to value.

Case 2 Statement: "I am mayor of a large city. I do not want us to be the victim of the next terrorist act. You are an expert on analyzing problems and defining solutions. How would you help me meet this challenge?"

Case 2 Expert Approaches: Four experts analyzed this case. A fifth turned down this 'job' as outside his area of expertise. The open-ended nature of this case led to several different starting points. One expert focused on refining the problem to contain the area of investigation, while two sought a clearer definition of the problem through identification of customers or stakeholders. Two recognized the critical role of the terrorist in this case, and recommended role-playing or war gaming to try to include the terrorists' perspectives. Details of the experts' approaches follow.

A. This is a very political problem. I'd start by trying to pin down the mayor by asking, "Why are you doing this?" "What are your goals?" From this, I could sketch out investigative boundaries. If the mayor vacillates, I would find out who else is involved and talk with them. My primary goal in the early phases is to contain the problem. If we don't have a well-defined problem, we'll just waste a lot of money.

B. I would first need to understand who the customer is (e.g., federal agencies, United Nations, the city) and what resources are available. I would also want to understand why the stakeholders believe such a system is needed and how quickly. This situation is more abstract than most problems I deal with regularly.

C. I would start by identifying all the stakeholders. The most essential stakeholder is the terrorist. Since direct participation by terrorists is not a good idea, I'd form a group of people paid to think like terrorists to work with representatives from the police, fire department, financial centers, etc., to brainstorm likely attacks.

D. I would create a war game in which individuals playing terrorists and defenders compete. This will get people out of linear-thinking mode and replace it with emotion. This will help us better uncover system dynamics, learn subtle feedback delays, and assess the degree of potential loss. I would do all of this because both the problem and the solutions are so ill-defined. In such a case, I could not do what comes natural for me, i.e., define the actors and events.

Case 3 Statement: "I am an architect of innovative buildings. I have just completed the detailed design of an office building to be built on the side of a mountain. The floors follow the contour of the steep building site, so that each floor only partially overlaps the floor below it. The only thing left to design is the elevator system to move people between floors. How would you help me define the requirements for this elevator?"

Case 3 Expert Approaches: Four experts described their approaches to this problem. The first objected to the assumption that the right solution, i.e., an elevator, was even known. The other 3 analysts focused on modeling the situation using scenarios, workflow diagrams, or features. In all three cases, they were clearly trying to understand how the occupants use the building.

A. I think it is too early to assume it will be an elevator. We must understand priorities, safety, schedule, resources, and parameters. I would keep the discussion at the more abstract level so we could arrive at an optimal solution, maybe not an elevator at all.

B. I'd start by identifying all the elevator usage scenarios, i.e., find out how people will use the building.

C. Analysts would study the current physical model (the building's design). Then I would try to learn about the tenants and create a workflow or traffic flow diagram to understand how people move between floors. I would not use my standard approach of constructing a DFD. I would talk with geologists and the building owner to get a list of viable solutions. I might suggest rearranging tenants to make the problem easier. What I am trying to do is understand the constraints, and what I can and cannot alter in the building's design.

D. This is a nice concrete problem. So, I'd start by learning about who would use the elevator. I would bring them together in a group session to discuss desired features, and then desired attributes. I also need to know more details (e.g., the grade angle/variability; if there is any recreational value for the system; whether it would be used only by people, for freight, or by cars; what building codes apply). I would then insist that the customer sign our document of understanding.

Case 4 Statement: "I am the registrar for a major university. Every semester, students complain because the student information system has denied the request to enroll in classes they had wanted to enroll in. However, we also seem to have many empty seats in the classrooms. Clearly, our student information system needs to be rebuilt. Can you help me?"

Case 4 Expert Approaches: Three experts analyzed this situation. The first two questioned whether the 'system' was really the problem at all. The third selected ethnographic techniques to learn the problems users were experiencing with the system. Interestingly, this was the expert who stated "I doubt we really know that the elevator is the solution" in the previous case.

A. The first thing I want to discover is whether the problem really is the information system. There are many other possible reasons for the empty seats.
B. I would look at historic patterns of data. Just what is the problem? Perhaps it is just a scheduling issue. We need to be aware that the solution may not lie in technology at all. For example, perhaps the university is using bad times, bad teachers, or bad rooms.
C. I'd interview and observe current users, perhaps videotape them, and collect "click statistics" to see what functions are performed most often. The analyst would also try to use the system looking for usability problems. Once these are located, I'd decide to either redesign the system or try to just fix it.

6.3 Other useful information

In this section, we report on other useful information gathered from the experts, including additional situational characteristics that they would consider when selecting an elicitation technique, and other general advice.

Situational Characteristics. Some of the experts offered specific conditions under which they would use certain techniques. For example,

- One expert recommended war-gaming when we have time and any of the following is true: the situation is emotional, there exists a risk of large loss, the problem is ill defined, or we don't know the actors or events.
- When there is little confidence that the developers understand the user needs, one expert suggested iterating toward a solution. When there is lack of trust in the developers, the expert suggested buying or licensing the solution rather than building it.
- The analyst's background, skills, and preferences may be a factor, but should never take priority over the needs of stakeholders.

General Advice. Some experts offered general advice:

- Several experts highlighted the need to identify users and other stakeholders first, and then to find a spokesperson for each. Others reflected sometimes contradictory views about the analyst's relationship to stakeholders stating: Analysts should always ask, not tell; Don't trust marketing personnel, always learn requirements yourself; Always work closely with people who have the checkbook; and Users are more important than the paying customers. Another emphasized that the analyst must demand that stakeholders define terms to ensure that different stakeholders are not using the same term for different purposes. Finally, one expert stated that you must remove toxic players from the process.
- Experts also discussed their recommended sequence of information discovery. One recommended that analysts find out users, features, and then attributes of the system, while two others recommended stakeholders, constraints, assumptions, and then scope of the system. Another emphasized that you should always understand and document the business case and quantify it financially prior to eliciting requirements.
- Finally, from a format/presentation perspective, one expert declared: Don't attempt to document requirements as a long list of shalls.

7. Discussion

Overall, the expert interviews resulted in an extremely rich source of information about requirements elicitation and technique selection. The structure and flow of the interviews also provided information, since they were a form of elicitation as well. For example, when asked how they performed elicitation, many of the experts told stories about their elicitation projects. This demonstrates the power of stories (i.e., scenarios, use cases) in elicitation. However, it is also interesting to note, that the stories the experts chose to tell us described their standard, default elicitation approach. While this may have been a result of our specific question, it may also be an indicator that individuals tend to choose common stories and must be prompted for exceptions. In our interviews, experts showed the most diversity when presented with the normalizing situations (see section 6.2). Finally, while the focus of our research is on technique selection, the experts were just as likely to describe the information they were seeking as the specific technique they would use. From this we can conclude that elicitation technique selection is not only a function of situational characteristics, but is also a function of what information (requirements) is still needed [31].

The research reported herein has some limitations:

- These interviews of nine experts may not (a) be representative of all expert analysts, (b) capture all their expertise or (c) provide sufficient coverage of all situations and techniques. We have not reached "theoretical saturation" [39] where we have learned all we need, so are planning additional interviews.
- Although we are both very experienced interviewers, our style of interviewing and previous experience may have biased the responses given by the subjects. We have tried to limit this bias through use of the interview protocol [41], delaying hypotheses formulation until after the interviews [37, 39], and following research-based interviewing guidelines [40].
- Like less experienced analysts, expert analysts may simply select the technique they are most comfortable with (one actually admitted this to us), and do not either consciously or subconsciously *select* a technique from the large set of available techniques. However, given the variety of approaches described by each of the experts, our conclusion is that while they may have a default approach, they do adapt that approach to the

situation if necessary. This became evident during the normalizing cases where we took many of the analysts out of the domains they were most familiar with.

- A difference may exist between what these experts *say* they do, and what they actually *do*. The best way to check this is to observe the experts doing elicitation rather than just asking them what they would do.

The study reported in this article represents early results of a larger research effort. Plans are underway to extend these interviews to many more experts in the field. Because of the preeminence of the nine analysts interviewed, we felt it important to report their unique and insightful comments. Other research that can be performed based on this paper includes:

- Integrate the results of these interviews with the results extracted from the other two sources of information as described in Section 4, i.e., our own experiences and the results of our document analyses.
- More fully understand the full range of situational characteristics that could affect technique selection. This work has begun [51].
- Create a more complete faceted classification of elicitation techniques. This work has also begun [51].
- Pivot the results reported in this paper, i.e., rather than discuss the conditions necessary to drive an elicitation technique, we need to discuss the technique alternatives that make sense under sets of conditions.
- Creation of a tool that uses the knowledge of the experts to map situational characteristics into the set of appropriate elicitation techniques.

8. Summary & Conclusions

Even though we have interviewed only a small sample of experts, some general trends are appearing. When analyzing the trends, new theories of elicitation technique selection, grounded in the data [37, 39], began to emerge:

- For each elicitation technique, there exists a specific, unique, small set of predicates concerning situational characteristics that drive experts to seriously consider that technique. We call these “Major Drivers.” For driving a car, an example is a green light. For collaborative sessions, the major drivers are multiple stakeholders, disparate needs, and a demand to reach consensus before proceeding.
- For each elicitation technique, there exists a set of additional predicates which if true cause experts to alter their primary choice. We call these “Anomalies.” For driving a car, this includes an ambulance in the intersection. For collaborative sessions, the anomalies include stakeholders who cannot know of each other’s existence, geographical distribution of stakeholders or no suitable venue (and no distributed meeting technology available), and not enough time to adequately prepare for the session.
- For each elicitation technique, there exists a set of basic analyst skills that must be present or the technique will not be effective. We call these skills “Prerequisite Skills.” For driving a car, these include the ability to press the accelerator and brake pedals. For collaborative sessions, they include communication, leadership, and the ability to facilitate meetings.
- For each elicitation technique, there exists a set of additional skills that are not universally needed, but that come into play during the technique’s execution without pre-knowledge. We call these skills “Success Enhancers.” For driving a car, this includes defensive driving. For collaborative sessions, these include modeling, conflict resolution, and creativity skills.

In addition, we believe that we can tentatively reach these conclusions about the use of modeling in conjunction with elicitation techniques:

- Creation of models (perhaps multiple models) seems to aid analysts in fully comprehending a situation and in communicating with stakeholders. Almost all the experts use them.
- For the experts (but probably not for most analysts), selecting a modeling notation that they are most comfortable with appears to be a good choice, but that notation must be palatable to the users.
- Many experts assess early the immediate gaps in knowledge among the stakeholders, and deliberately direct the elicitation in the direction of achieving these intermediate goals. Examples might include particular non-behavioral requirements, user interface, and database contents. This often drives the expert analyst toward specific modeling notations to supplement the elicitation technique.

To summarize, while the state of our research has not allowed us to reach definitive conclusions on all situations and techniques, this paper is an important contribution to understanding the techniques that experts use during elicitation and the situational factors they consider when choosing those techniques. Future research will allow us extend these conclusions to a wider range of experts, techniques and situations, provide specific guidance to practicing analysts, and ultimately improve the state of the practice in requirements elicitation.

9. Acknowledgements

The authors would like to thank all the experts who gave so unselfishly of their time to the cause of improving the state of requirements elicitation practice. We would also like to thank Professor Gary Klein and the UCCS College of Business for establishing an environment where research is rewarding, rewarded, and enjoyable.

10. References

[1] Jones, C., *Patterns of Software Failure and Success*, Thomson, 1996.
[2] Hickey, A., and A. Davis, "The Role of Requirements Elicitation Techniques in Achieving Software Quality," *Requirements Eng. Workshop: Foundations for Software Quality (REFSQ),* 2002.
[3] The Standish Group, "The Chaos Report," 1995, and "Chaos: A Recipe for Success," 1999, www.standishgroup.com.
[4] Davis, A., and A. Hickey, "Requirements Researchers: Do We Practice What We Preach," *Requirements Eng. J.*, **7**, 2 (June 2002), pp. 107-111.
[5] Hickey, A., and A. Davis, "Barriers to Transferring Requirements Elicitation Techniques to Practice," *2003 Business Information Systems Conf.*, IEEE CS, 2003.
[6] Moore, G., *Crossing the Chasm*, HarperCollins, 1991.
[7] Davis, A., "Operational Prototyping: A New Development Approach," *IEEE Software,* **9**, 5 (Sept 1992), pp. 70-78.
[8] Ross, D., "Structured Analysis (SA): A Language for Communicating Ideas," *IEEE Transactions on Software Engineering,* **3**, 1 (Jan 1977), pp. 16-34.
[9] Heninger, K., "Specifying Software Requirements for Complex Systems: New Techniques and Their Application," *IEEE Trans on Software Eng*, **6**, 1 (Jan 1980), pp. 2-12.
[10] DeMarco, T., *Structured Analysis and System Specification*, Prentice Hall, 1979.
[11] Gause, D., and G. Weinberg, *Are Your Lights On?*, Dorset House, 1990.
[12] Wood, J., and D. Silver, *Joint Application Development*, Wiley, 1995.
[13] Jackson, M., *Problem Frames*, Addison-Wesley, 2001.
[14] Booch, G., *Object-Oriented Analysis and Design*, Benjamin/Cummings, 1994.
[15] Brooks, F., "No Silver Bullet – Essence and Accidents of Software Engineering," *Computer,* **20**, 4 (Apr 1987), pp. 10-19.
[16] Kotonya, G., and I. Sommerville, *Requirements Engineering*, Wiley, 1998.
[17] Macaulay, L., *Requirements Engineering*, Springer, 1996.
[18] Maiden, N., and G. Rugg, "ACRE: Selecting Methods for Requirements Acquisition," *Software Engineering J.*, **11**, 5 (May 1996), pp. 183-192.
[19] Glass, R., "Searching for the Holy Grail of Software Engineering," *Comm. of the ACM,* **45**, 5 (May 2002), pp. 15-16.
[20] Yadav, S., et al., "Comparison of Analysis Techniques for Information Requirements Determination," *Communications of the ACM*, **31**, 9 (Sept 1988).
[21] Davis, A., *Software Requirements: Objects, Functions and States*, Prentice Hall, 1993.
[22] Wieringa, R., *Requirements Engineering*, Wiley, 1996.
[23] Lauesen, S., *Software Requirements: Styles and Techniques*, Addison-Wesley, 2002.
[24] Leffingwell, D., and D. Widrig, *Managing Software Requirements*, Addison-Wesley, 2000.
[25] Wiegers, K., *Software Requirements*, Microsoft, 1999.
[26] Hudlicka, E., "Requirements Elicitation with Indirect Knowledge Elicitation Techniques: Comparison of Three Methods," *International Conf. on Requirements Engineering (ICRE)*, IEEE CS Press, 1996, pp. 4-11.
[27] Byrd, T., et al., "A Synthesis of Research on Requirements Analysis and Knowledge Acquisition Techniques," *MIS Quarterly,* **16**, 1 (Mar 1992), pp. 117 – 138.
[28] Couger, D., "Evolution of Business System Analysis Techniques," *ACM Comp. Surveys,* **5**, 3 (Sept 1973), pp. 167-198.
[29] Davis, A., "A Taxonomy for the Early Stages of the Software Development Life Cycle," *Journal of Systems and Software*, **8**, 4 (Sept 1988), pp. 297-311.
[30] Goguen, J., and C. Linde, "Software Requirements Analysis and Specification in Europe: An Overview," *First Int'l Symp. on Requirements Engineering,* IEEE CS Press, 1993, pp. 152-164.
[31] Hickey, A., and A. Davis, "Requirements Elicitation and Elicitation Technique Selection: A Model for Two Knowledge-Intensive Software Development Processes," *Proceedings of the Thirty-Sixth Annual Hawaii International Conf. on Systems Sciences (HICSS),* IEEE CS, 2003.
[32] McDaniel, C., and R. Gates, *Marketing Research Essentials*, West Publishing, 1998.
[33] Gause, D., and G. Weinberg, *Exploring Requirements: Quality Before Design*, Dorset House, 1989.
[34] Gottesdeiner, E., *Requirements by Collaboration*, Addison-Wesley, 2002.
[35] Davis, A., "Software Prototyping," *Advances in Computers,* **40**, Academic Press, 1995, pp. 39-63.
[36] Kowal, J., *Behavior Models*, Prentice Hall, 1992.
[37] Strauss, A. and J. Corbin, *Basics of Qualitative Research,* 2nd edition, Sage, 1998.
[38] Marshall, C., and G. Rossman, *Designing Qualitative Research,* 3rd edition, Sage, 1999.
[39] Glaser, B., and A. Strauss, *The Discovery of Grounded Theory: Strategies for Qualitative Research,* Aldine, 1967.
[40] Seidman, I., *Interviewing as Qualitative Research: A Guide for Researchers in Education and the Social Sciences,* 2nd edition, Teachers College Press, 1998.
[41] Warren, C., "Qualitative Interviewing," in J. Gubrium and J. Holstein (eds.), *Handbook of Interview Research: Context and Method,* Sage, 2002, pp. 83-101.
[42] Johnson, J., "In-Depth Interviewing," in J. Gubrium and J. Holstein (eds.), *Handbook of Interview Research: Context and Method,* Sage, 2002, pp. 103-119.
[43] Polanyi, M., *Tacit Dimension*, Doubleday, 1966.
[44] Harel, D., "Statecharts: A Visual Formalism for Complex Systems," *Science of Comp. Prog.*, **8** (1987), pp. 231-274.
[45] Rational Software Corporation, *UML Summary*, Jan. 1997.
[46] McMenamin, S., and J. Palmer, *Essential Systems Analysis*, Prentice Hall, 1984.
[47] Chen, P., "The Entity-Relationship Model: Toward a Unifying View of Data," *ACM Trans. on Database Systems,* **1**, 1 (Mar 1977), pp. 9-36.
[48] Robertson, S., and J. Robertson, *Mastering the Requirements Process,* Addison-Wesley, 1999.
[49] Beck, K., *Extreme Programming Explained,* Addison-Wesley, 2000.
[50] McCracken, G., *The Long Interview,* Sage, 1988.
[51] Hickey, A., and A. Davis, "A Tale of Two Ontologies: A Basis for Systems Analysis Technique Selection," *Americas Conference of Computer Information Systems (AMCIS),* 2003.

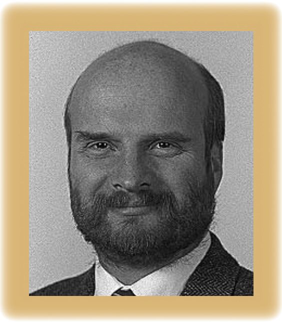
Editor: Neil Maiden ■ City University, London ■ n.a.m.maiden@city.ac.uk

10 Small Steps to Better Requirements

Ian Alexander

"The journey of a thousand miles begins with a single step." This Chinese proverb helps people focus on the present, rather than the unmanageable future.

Project teams can take several small, easy steps to improve requirements to the point where they're good enough. But every project is different. Your team might need to take steps that wouldn't be right in other situations.



The basic steps listed here, roughly in order, demand nothing more than a whiteboard or a pen and paper. Requirements management tools and specialized models can help later on.

Step 1: Mission and scope

If you don't know where you're going, you're not likely to end up there, said Forrest Gump. Is your mission clear? Write a short statement of what you want your project to achieve. Out of that will grow an understanding of the project's scope. The scope will develop as you more clearly understand exactly what achieving the mission entails.

A traditional context diagram is a good tool for describing scope in a simple outline. Unlike a use-case summary diagram, it indicates interfaces and the information or materials that must come in or out of them, rather than just roles (that is, UML actors). This becomes especially important when specifying physical systems where software is just a (large) component. Later, you can analyze each interface in detail.

Step 2: Stakeholders

Requirements come from people, not books. You can look in old system documents and read change requests, but you must still validate any candidate requirements you discover with people. Missing a group of stakeholders means your system could lack a whole chunk of functionality or other requirements.

Stakeholders are far more diverse than can be understood by listing their operational roles. Every system has a varied set of beneficiaries. The role of regulator is often critical. Stakeholders' negative opinions and actions can halt the project if not taken into account.

At the very least, you should list everyone with an interest in the system and consider what degree of involvement each person or group should have. A template might help you identify some stakeholders you'd otherwise overlook.

Step 3: Goals

Once you know who your stakeholders are, you can identify their goals. This can begin with a list or, perhaps better, a hierarchy (you could use an outliner, such as Microsoft Word's Outline View, or a mind-mapping tool).

Goals can be optimistic and even unrealizable—for example, a universally available product, a perfectly safe railway, or a ceaseless power source. In this way, they're unlike requirements,

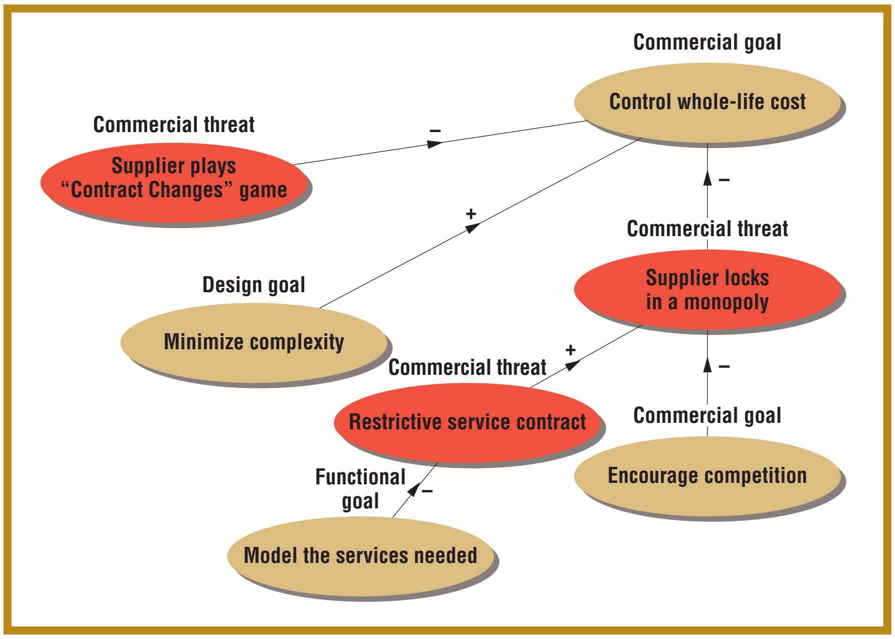


Figure 1. Analyzing goal and threat interactions quickly transforms vague targets into concrete functions. Here, it's applied to explore the core goals of a large public-service company that must upgrade its existing client-server system. The gold bubbles are desired goals; the red bubbles are threats.

which must be definitely realizable and verifiable.

Large initial goals are often vague qualities that stakeholders (for example, marketing or product management) want product users to experience. You can usually break such aspirations down into more concrete, realizable goals. For example, the goal "the car should feel luxurious" might be achieved with leather upholstery, air conditioning, soundproofing, walnut veneer, or hi-fi music. This second tier of goals suggests possible features and even software functions to help to meet the initial goal.

Step 4: Goal conflicts

Goals often conflict, but requirements must not. It's essential to discover and resolve any conflicts early on.

Some conflicts will be obvious from a plain list of goals; others are easier to see on a goal model (see figure 1). Draw each goal as a bubble; label it with its type (for example, commercial or functional). Draw an arrow marked with a plus sign (+) to show that one goal supports another; draw an arrow marked with a minus sign (–) to show a negative effect.

Step 5: Scenarios

Scenarios use the element of time to translate goals into connected stories. Isolated "shall" statements attempt to communicate needs as if they could be implemented independently, whereas connecting sequences of needs makes clear what dependencies exist.

At the very least, you should describe each operational situation as a brief story or numbered list of steps. Head each scenario with the name of the functional goal it implements. Write each step as a short statement in the form <role> <performs an action>.

When this is no longer sufficient, switch to writing use cases at a level of detail that suits your project. For example, Cockburn-style use cases include pre- and postconditions, making the transition to code far more controllable.

Step 6: Requirements

Scenarios don't cover everything. You will need techniques for interfaces, constraints (such as time and cost), and qualities (such as dependability and security).

Traditional "the system shall" statements are often convenient, not least because suppliers and standards expect them. Of course, you might think that the requirements consist of all the things listed here and more. However, not everyone shares that understanding.

Use cases do more than anything else to bridge the gap between the "now we do this" world of scenarios and the "you shall do that" world of traditional requirements. Neil Maiden has championed the systematic use of use cases to discover missing requirements ("Systematic Scenario Walkthroughs with Art-Scene," *Scenarios, Stories, Use Cases: Through the Systems Development Life-Cycle*, John Wiley & Sons, 2004).

Step 7: Justifications

If you note only what your system has to do and not why it has to do it, developers must guess what's really important. When resources start to run out or when planners see that demand will exceed available effort, project managers must cut or postpone something. A justification, like a priority, helps protect vital requirements from being dropped. It also helps developers understand how each requirement fits and why it's needed.

You can construct justifications in many ways—most simply with a few words explaining a requirement's purpose attached to individual requirements as attributes or listed separately and linked to requirements. You can construct more elaborate justifications as argumentation models, goals, risk models, or cost-benefit models.

Step 8: Assumptions

A system specification is a combination of things that you want to become true under system control (*requirements*) and things that you need to be true but can't control (*assumptions*).

Assumptions can include beliefs about existing systems, interfaces, competition, the market, the law, safety hazard mitigations, and so on. The most dangerous assumptions are tacit—the ones you didn't know you were making.

Like justifications, you can model assumptions as any kind of argumentation. The simplest approach is to state each assumption as a piece of text.

When planning a project, it can be valuable to examine the assumptions the

Further Reading

- **Mission and scope:** Dean Leffingwell and Don Widrig, *Managing Software Requirements,* Addison-Wesley, 2000.
  Suzanne Robertson and James Robertson, "Project Blastoff," *Mastering the Requirements Process,* Addison-Wesley, 1999.
- **Stakeholders:** Ian Alexander, "A Taxonomy of Stakeholders," *Int'l J. Tech. and Human Interaction,* vol. 1, no. 1, 2005, pp. 23–59.
  Ian Alexander and Suzanne Robertson, "Stakeholders without Tears," *IEEE Software,* vol. 21, no. 1, 2004, pp. 23–27.
- **Goals:** Alistair Sutcliffe, "Chapter 3: RE Tasks and Processes," *User-Centred Requirements Engineering,* Springer, 2002.
- **Goal conflicts:** Ian Alexander, "Misuse Cases: Use Cases with Hostile Intent," *IEEE Software,* vol. 20, no. 1, 2003, pp. 58–66.
- **Scenarios:** Alistair Cockburn, *Writing Effective Use Cases,* Addison-Wesley, 2001.
- **Requirements:** Ian Alexander and Richard Stevens, *Writing Better Requirements,* Addison-Wesley, 2002.
  Ian Alexander and Neil Maiden, "What Scenarios (Still) Aren't Good For," *Scenarios, Stories, Use Cases,* John Wiley & Sons, 2002.
- **Justifications:** Paul A. Kirschner, Simon J. Buckingham Shum, and Chad S. Carr, *Visualizing Argumentation,* Springer, 2003.
- **Assumptions:** James A. Dewar, *Assumption-Based Planning,* Cambridge University Press, 2002.
- **Agreed priorities:** Alan M. Davis, "The Art of Requirements Triage," *Computer,* vol. 36, no. 3, 2003, pp. 42–49.
- **Acceptance criteria:** Suzanne Robertson and James Robertson, "Fit Criteria," *Mastering the Requirements Process,* Addison-Wesley, 1999.

project depends on and to consider a course of action should any assumption fail and threaten the project's survival.

Step 9: Agreed priorities

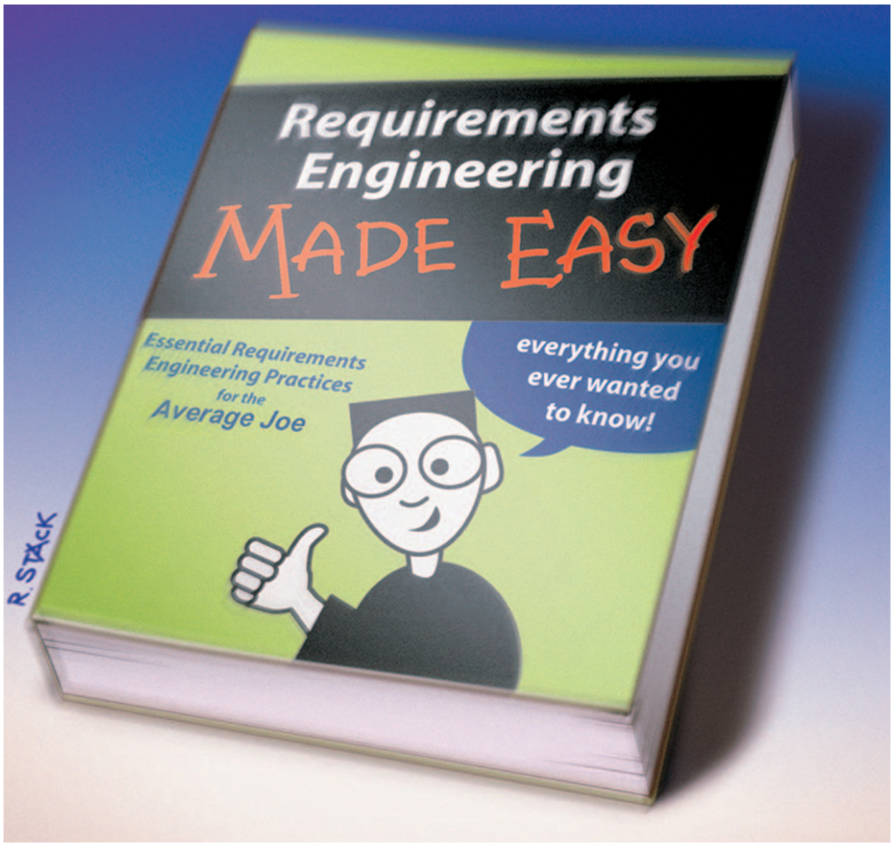
Some requirements are so obviously vital that you should just do them. Some are so obviously unnecessary that you should just drop them at once. You must prioritize the rest more closely. Dividing requirements into these three categories is a short, brutal process called *triage*.

Next, rank the remaining requirements into categories such as vital, desirable, nice to have, and luxury. You can do this by voting, reaching consensus among a panel, or allocating notional money.

Step 10: Acceptance criteria

Requirements don't end when you've completed the specifications. Rather, their work is just beginning—they guide development all the way to acceptance and service.

You should identify how you'll know when the system meets each requirement. This isn't a full description of each test case but criteria for deciding whether the system passes or fails against the requirement. Later, plan the test campaign on the basis of the scenarios (among other factors) and trace the test cases to the requirements to demonstrate coverage.


Requirements Engineering
MADE EASY
Essential Requirements Engineering Practices for the Average Joe
everything you ever wanted to know!
R. STACK

I can't guarantee that after taking these 10 steps, your requirements will be perfect. But if you follow them carefully, your requirements will improve.

Ian Alexander is an independent consultant specializing in requirements engineering for systems in the automotive, aerospace, transport, telecommunications, and public-service sectors. Contact him at iany@easynet.co.uk.

Analytics-Driven Dashboards Enable Leading Indicators for Requirements and Designs of Large-Scale Systems

Richard W. Selby, *Northrop Grumman Space Technology and University of Southern California*

Mining software repositories using analytics-driven dashboards helps us understand, evaluate, and predict the development, management, and economics of large-scale systems and processes.

Current and future technological systems are increasingly complex and large-scale in terms of system size, functionality breadth, component maturity, and supplier heterogeneity.[1–3] Organizations that tackle these large-scale systems and attempt to achieve ambitious goals often deliver incomplete capabilities, produce inflexible designs, reveal poor progress visibility, and consume unfavorable schedule durations.[1,2,4] Successful management of these systems requires the ability to learn from past performance, understand current challenges and opportunities, and develop plans for the future.[5] Effective management planning, decision making, and learning processes rely on a spectrum of data, information, and knowledge to be successful. However, many organizations and projects possess insufficient or poorly organized data collection and analysis mechanisms that result in limited, inaccurate, or untimely feedback to managers and developers. Organization and project performance suffers because managers and developers do not have the data they need or do not exploit the data available to yield useful information.

Mining software repositories using analytics-driven dashboards provides a unifying mechanism for understanding, evaluating, and predicting the development, management, and economics of large-scale systems and processes. This research examines dashboards used on actual large-scale software projects, focuses on leading indicators for requirements and designs, and determines useful empirical relationships.

Dashboards

Mining software repositories using analytics-driven dashboards provides the foundation for effective and efficient management of organizations and projects that develop large-scale systems.[6] Dashboards provide graphical displays of interactive measurement-driven gauges that depict trends, identify outliers, and support drill-down capabilities to more detailed information.[6,7] These information-intensive dashboards create value by collecting, analyzing, and synthesizing data so that decision makers can characterize progress, compare alternatives, evaluate risks, and predict outcomes.[8] Decision makers use dashboards to enable data-driven management as well as systematic process improvement and organizational change.[9–11] Many improvement methods, such as Six Sigma and Capability Maturity Model Integration (CMMI), utilize data-centric techniques and rely on accurate measurement-based information.[12]

Different dashboards address different sets of

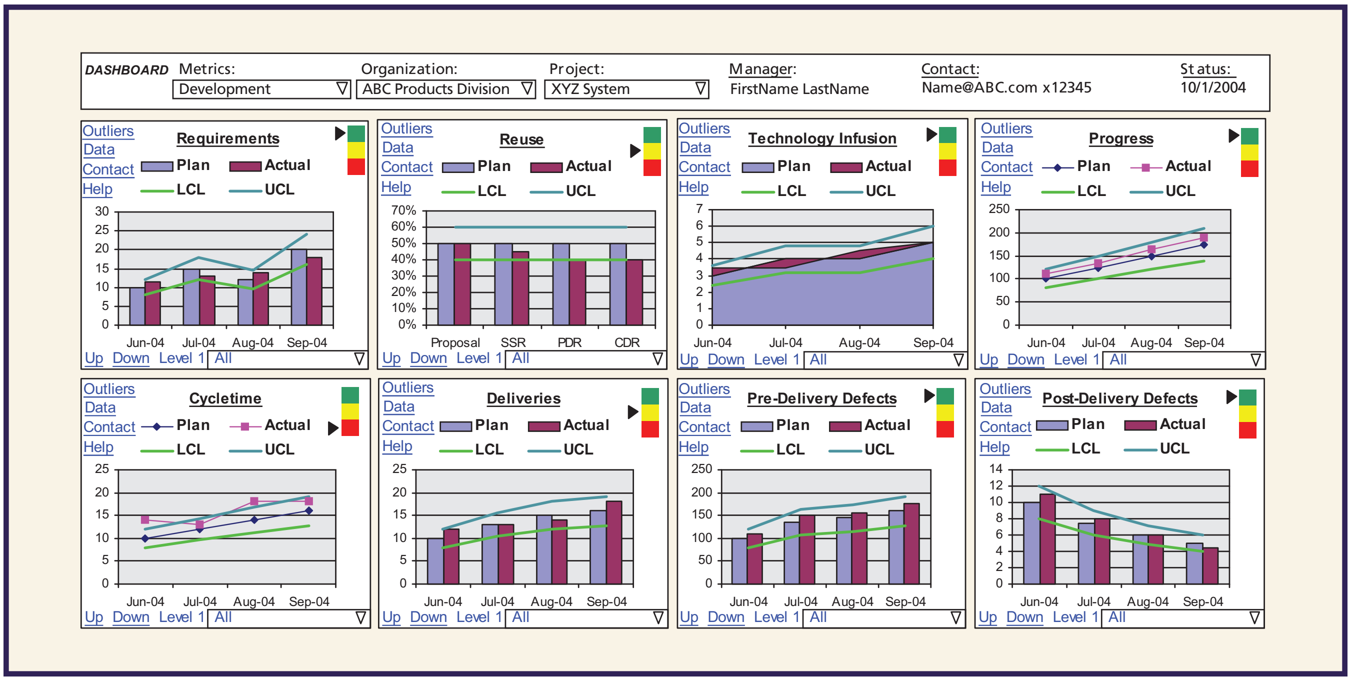


Figure 1. Example dashboard for software projects.

goals. For example, one dashboard might support the following development-oriented goal areas: requirements, reuse, technology infusion, progress, cycle time, deliveries, predelivery defects, and postdelivery defects. Another dashboard might support the following management-oriented goal areas: business scope, customer satisfaction, risk, business development, finance, personnel, process compliance, and process improvement. Figure 1 displays an example dashboard for software development-oriented goals.

Dashboards incorporate a variety of information and features to help managers and developers characterize progress, identify outliers, compare alternatives, evaluate risks, and predict outcomes. The design principles for dashboards include the following:

- Support different metric sets for different goals.
- Use different types of displays or gauges for different types of data.
- Enable organization- and project-level displays.
- Define points of contact that identify people responsible for feedback and follow-up.
- Define data trends and the date through which the data are current.
- Define lower and upper control limits and hyperlinks for displaying outliers.
- Provide hyperlinks that display tabular formats of the underlying data.
- Give context-specific help.
- Offer hyperlinks to view or “drill down” hierarchically to more detailed data.
- Highlight overall status using red-yellow-green or similar indicators.

Figure 2 redisplays the example dashboard from Figure 1 and identifies key features that support design principles for improving usefulness.

This article focuses on metrics and analyses from dashboards that have been used to mine software repositories on actual large-scale software projects as well as example empirical relationships revealed by the metric data. The investigation examines dashboard analytics across two sets of software projects containing 14 systems and 23 systems. Project personnel established product and process improvement goals for these environments and collected data using combinations of automated tools and manual data collection forms. Data collection occurred during project start-up, continued on a periodic basis throughout development, captured project completion information, and incorporated any postdelivery changes. Data validation occurred through a series of steps including extensive training, interviews, independent cross-checks, and reviews of results by objective personnel. For example, fault data were collected using manual forms while design specifications and source code versions were analyzed automatically using configuration management and static analysis tools. After developers released components into configuration management, they had to complete formal change request forms for modifications of any type. Developers identified the modifications that were caused by faults as opposed to

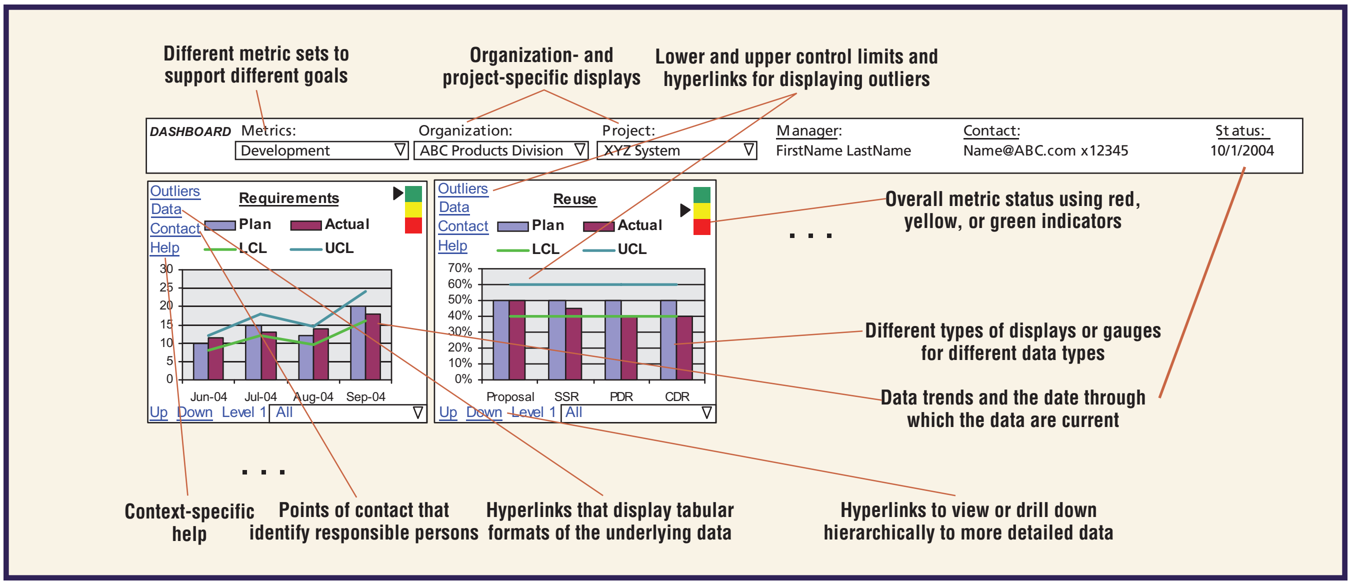

Different metric sets to support different goals
Organization- and project-specific displays
Lower and upper control limits and hyperlinks for displaying outliers
DASHBOARD Metrics: Development
Organization: ABC Products Division
Project: XYZ System
Manager: FirstName LastName
Contact: Name@ABC.com x12345
Status: 10/1/2004
Outliers
Data
Contact
Help
Requirements
Plan
Actual
LCL
UCL
Jun-04
Jul-04
Aug-04
Sep-04
Up Down Level 1 All
Reuse
Proposal
SSR
PDR
CDR
Overall metric status using red, yellow, or green indicators
Different types of displays or gauges for different data types
Data trends and the date through which the data are current
Context-specific help
Points of contact that identify responsible persons
Hyperlinks that display tabular formats of the underlying data
Hyperlinks to view or drill down hierarchically to more detailed data

Figure 2. Example partial dashboard for software projects. The highlights illustrate key features that support design principles for improving usefulness.

enhancements, and they also recorded fault severity and correction effort.

Software Requirements Metrics

Dashboards commonly include software requirements metrics because they are leading indicators of project scope, growth, volatility, and progress. Software requirements metrics characterize the "problem space" that a project is addressing, as opposed to metrics such as source-lines-of-code (LOC) that characterize the "solution space." Software requirements metrics are also available very early in a project and can form the basis for early analyses and predictions of project plans, alternatives, risks, and outcomes. Using software requirements metrics also helps resolve the definitional and counting issues associated with reused design or code and whether components are developed in-house or from commercial-off-the-shelf (COTS) suppliers. Software requirements metrics, in terms of functionality and performance, are still applicable regardless of whether the implementation reuses software design elements, reuses code implementations, or incorporates COTS components. Of course, the project requirements, in terms of organization and process, may vary depending on the degree of software reuse and usage of COTS components.

The requirements data analyzed originate from 14 large-scale projects that use measurement-driven dashboards to actively manage their development activities and evolving products. The projects are developing large-scale mission-critical embedded software systems that implement functionality for advanced robotic spacecraft platforms, high-bandwidth satellite communications, and high-power laser systems. Experienced developers apply domain-specific architectures, modular components, and mature development processes to design, develop, and enhance these systems in incremental software life cycles. System sizes range from 25,000 to 1 million source lines, and teams include 10 to 150 developers.

Figure 3 displays the projects' software requirements metrics. For these projects, the number of requirements is defined to be the number of "Shall" statements in the requirements specification documents. For example, a requirements document may contain the following statement "the system shall determine the three-dimensional location of an object within an accuracy of 0.01 meter." This Shall statement would count as one requirement. To facilitate consistency within and across projects, requirement specification standards and guidelines need to be defined to enforce the breadth and depth of functionality expressed in a single requirement. Of course, simply counting the Shall statements oversimplifies the project requirements, but this metric does provide an initial basis for project scope, growth, volatility, and progress.

Data Analysis for Software Requirements Metrics

Data analysis of the software requirements metrics reveals the following observations:

- The ratio of requirements in a software specification to delivered LOC averages 1:81 for mature projects and has a median of 1:35.
- When Project 14 is excluded (see Figure 3), the ratio of requirements in a software specification to delivered LOC averages 1:46 for mature projects and has a median of 1:33.

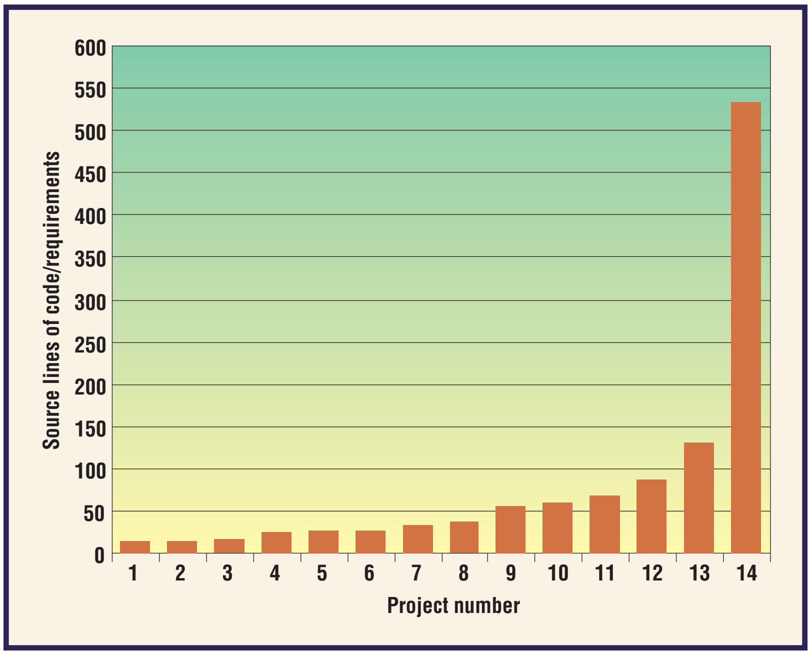


Figure 3. Ratio of implementation size to requirements. These software requirements metrics characterize the projects.

- Projects that far exceeded the 1:46 requirements-to-code ratio, such as Projects 13 and 14, tended to be more effort-intensive and fault-prone during verification.

This requirements metrics data analysis enhances visibility into large-scale systems by highlighting the interrelationships between requirement metrics and implementation metrics. Many models of software systems, such as those for cost estimation, fault isolation, or structure visualization, rely on some representation of software size. Modelers and developers commonly capture software size using implementation-based metrics that typically center on some variation of LOC. In order to enable models to be applicable earlier in the life cycle so that they identify leading indicators, these models need to use software size metrics based on requirements. If developers quantify their software sizes using requirements metrics, such as numbers of Shalls, and they understand the relationships between requirements metrics and implementation metrics, such as ratios of Shalls to LOC, they will be able to effectively understand, model, and evaluate scope, growth, volatility, and progress throughout the life cycle. For example, if a development environment has an average requirements-to-code ratio of 1:46, then simple predictive models can count Shalls, track growth in Shalls, and estimate implementation sizes and growth. If the actual implementation sizes far exceed the 1:46 requirements-to-code ratio, such as sizes greater than 1:130 as seen on Projects 13 and 14 in Figure 3, this provides a leading indicator of requirements that were possibly defined at too high a level or described incompletely. Dashboards can display control limits, such as an upper control limit that is the average plus two standard deviations and a lower control limit that is the average minus two standard deviations, to help identify when projects or subsystems are deviating from past experience and may be trending toward unusually favorable or unfavorable outcomes. Developers can also use these types of ratios to estimate and monitor the amount of functionality allocated to each software release in order to balance end-user value, developer workload, and testing effort across a series of releases.

Software Design Modeling Metrics

Dashboards containing design metrics enable early life-cycle system modeling that accelerates design analysis, trade-offs of proposed system capabilities, identification of reuse opportunities, and analytical decision making as well as early fault detection.[2,13] Design metrics derived from state- and interaction-based modeling provide a powerful approach to capture and analyze designs of complex systems. These models facilitate the identification of essential system characteristics in terms of intra- and inter-component structure, and they enable comprehensive analysis by being scalable to large systems.[14] Static analysis tools can derive the number of internal states for a component from its internal control flow design structure based on cyclomatic complexity, and they can calculate the number of interactions between components from the passing of control flow or data abstractions between components.

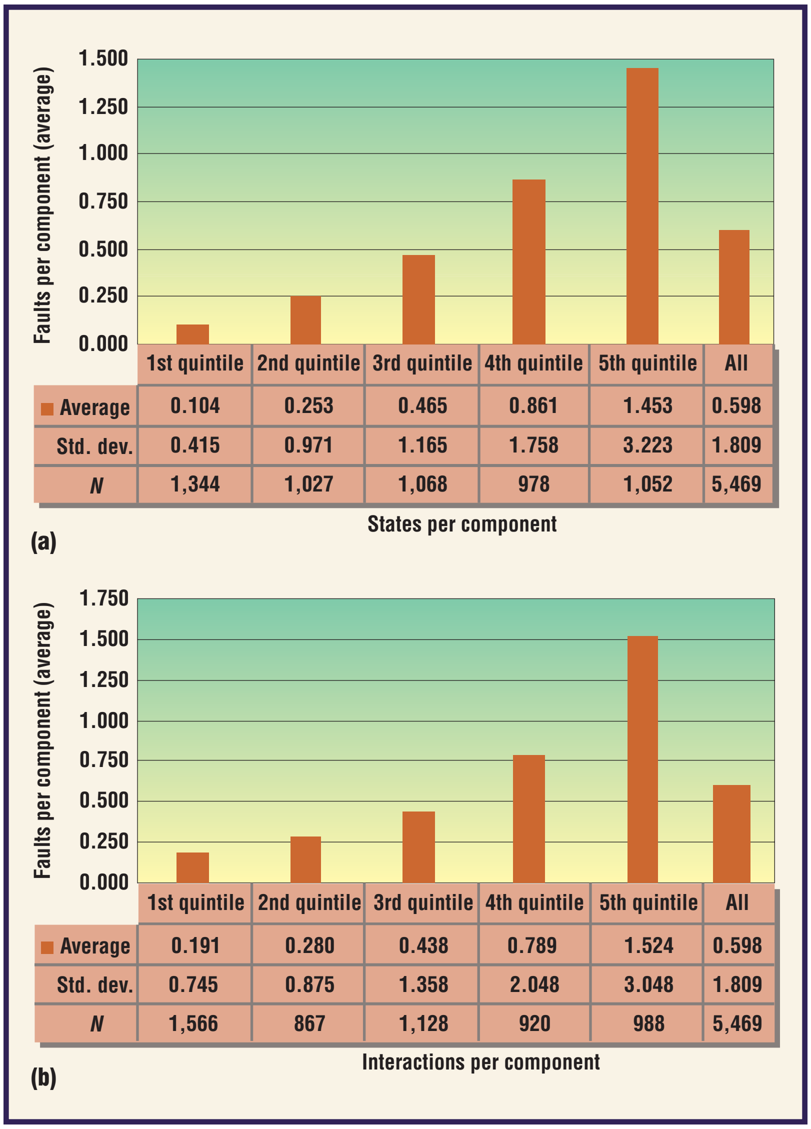
The design data analyzed originated from applying static analysis tools to the designs of 23 large-scale spacecraft systems containing over 5,000 software components in order to automatically generate state- and interaction-based models of their designs.[15,16] The 23 software systems provide ground support functions for unmanned spacecraft control. Each system typically includes several hundred components, where the term “components” refers to the main programs, subroutines, utility functions, macros, and data in the systems. The dashboards evaluate the design modeling in terms of component development fault-proneness.

During system development, the components had 0.60 fault on average. Figures 4a and 4b display the component averages for faults categorized by component states and component interactions, respectively (overall difference is statistically significant at $\alpha < 0.0001$ and pairwise differences are statistically significant at $\alpha < 0.05$ for each figure). The dashboards also calculated component averages for fault correction effort categorized by component states and component interactions as well as component averages for development effort categorized by component states and component interactions, even though the figures shown do not display these metrics.

Many design modeling metrics correlate with size, so developers often examine metric data both with and without size normalization. For component states, we normalized the data analysis of the component states quintiles and fault metrics by component size (in KLOC) as follows. The component faults-per-KLOC averages were 1.885, 3.300, 3.855, 4.013, and 4.047 for the first, second, third, fourth, and fifth quintiles of component states per KLOC, respectively. For component interactions, we normalized the component interactions quintiles and fault metrics by component size in KLOC and then analyzed the data as follows. The component faults-per-KLOC averages were 2.660, 2.876, 3.137, 3.919, and 4.526 for the first, second, third, fourth, and fifth quintiles of component interactions per KLOC, respectively.

Data Analysis for Software Design Modeling Metrics

Data analysis of the software design modeling metrics reveals the following observations, with comparative data being statistically significant at $\alpha < 0.05$. These types of empirical relationships could use either metrics based on univariate averages (called "absolute") or those that account for the simultaneous effects of component states, interactions, and origin (called "adjusted"). The empirical relationships we will summarize define factors or leading indicators using absolute metric ratios.



(a)	1st quintile	2nd quintile	3rd quintile	4th quintile	5th quintile	All
🟧 Average	0.104	0.253	0.465	0.861	1.453	0.598
Std. dev.	0.415	0.971	1.165	1.758	3.223	1.809
N	1,344	1,027	1,068	978	1,052	5,469

(b)	1st quintile	2nd quintile	3rd quintile	4th quintile	5th quintile	All
🟧 Average	0.191	0.280	0.438	0.789	1.524	0.598
Std. dev.	0.745	0.875	1.358	2.048	3.048	1.809
N	1,566	867	1,128	920	988	5,469

Figure 4. Fault-proneness for (a) component states and (b) component interactions. Component averages for faults are categorized by component states and component interactions (for 5,469 components).

- The static analysis tools successfully generated state- and interaction-based models of the 23 system designs comprising 5,469 components. The numbers of states per component were exponentially distributed with an average of 14.3 and a median of 7.0. The numbers of interactions per component were exponentially distributed with an average of 8.8 and a median of 4.0. During system development, the components had 0.60 fault on average.
- Components with the most states (top 20 percent) had 14.0 times more faults than did those with the fewest states (bottom 20 percent). Components with the most states also had 37.4 times more fault correction effort and 14.2 times more development effort than did those with the fewest states.
- Components with the most interactions (top 20 percent) had 8.0 times more faults than did those with the fewest interactions (bottom 20 percent). Components with the most interactions also had 12.4 times more fault correction effort and 10.3 times more development effort than did those with the fewest interactions.
- Components that had either the most states (top 20 percent) and a high number of interactions (top 40 percent), or a nominal or greater number of states (top 60 percent) and the most interactions (top 20 percent) resulted in more faults, fault correction effort, and development effort on average.

This design modeling metrics data analysis illustrates an approach for capturing and exploiting design information to define leading indicators for development faults and effort. Developers ideally capture design information during the creation of their systems but often do not record this

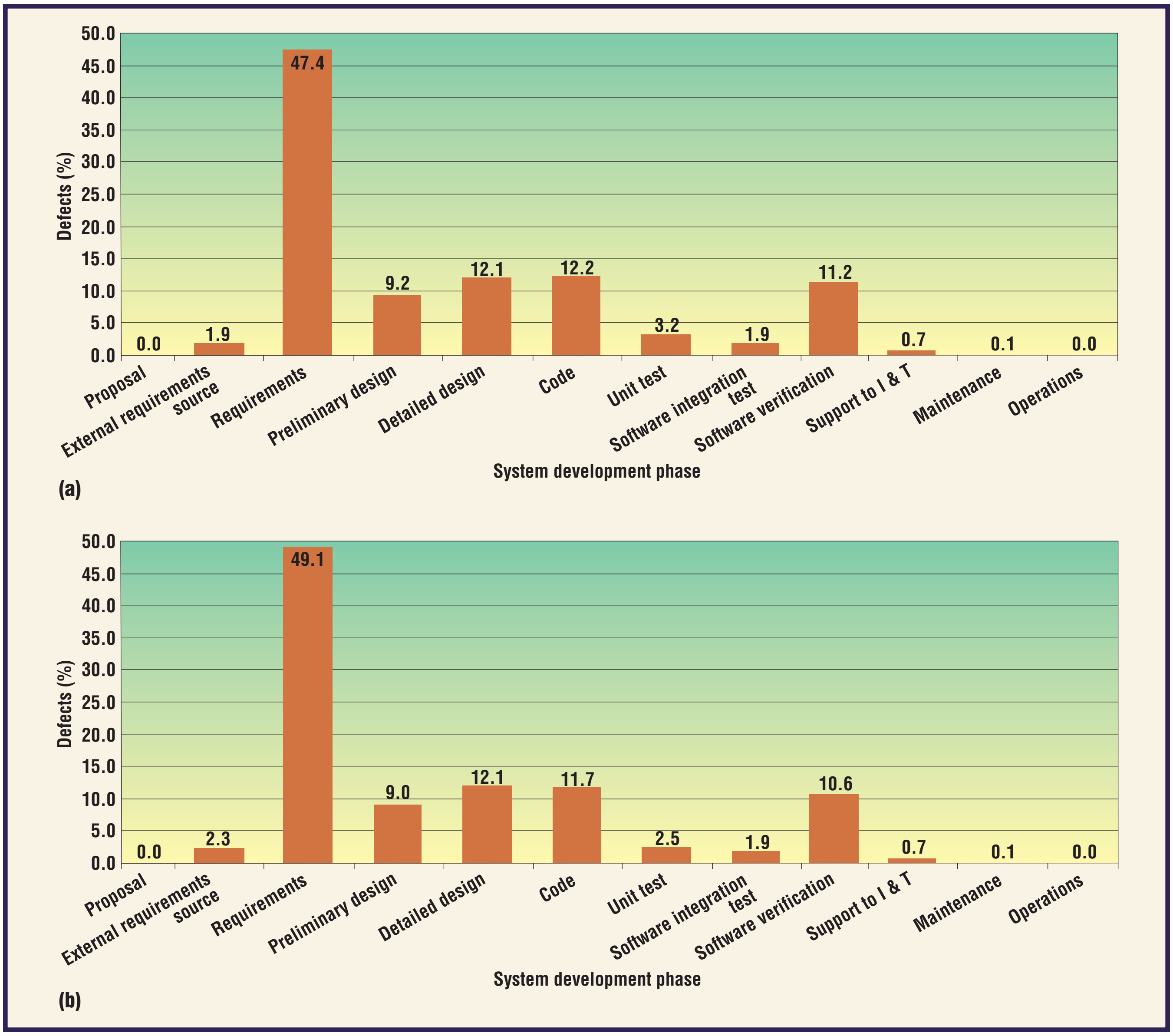


Figure 5. Software fault (or defect) (a) detection phase and (b) injection phase, summarized based on using peer reviews across 12 system development phases (3,418 faults, 731 peer reviews, 14 systems, 2.67 years).

information, and even if they do, the information needs to be kept current as the systems evolve. This challenge compounds significantly when developers tackle large-scale systems that include numerous heterogeneous components and complex interdependencies. Dashboards can capture and display design modeling metrics based on the automated calculation of components' states and interactions. Developers can use leading indicators and empirical relationships such as those in Figures 4a and 4b to identify which portions of a system design are likely to have high faults, high fault correction effort, and high development effort based on past experience. Developers can use these leading indicators to identify those portions of a system that may need further analysis or re-design. This type of empirical guidance can assist developers who are conceiving initial designs or evaluating existing designs. In the development environment analyzed, for example, designs that contain components with relatively high numbers of states or interactions suggest that these are high-payoff areas for further analysis or review.

Software Fault Detection and Injection Metrics

Dashboards containing metrics that capture fault detection and injection data enable characterization, evaluation, and prediction of fault detection effectiveness and identification of improvement opportunities.[3] This portion of the study investigates dashboards that analyzed software fault detection and injection data resulting from peer reviews across 12 system development phases on 14

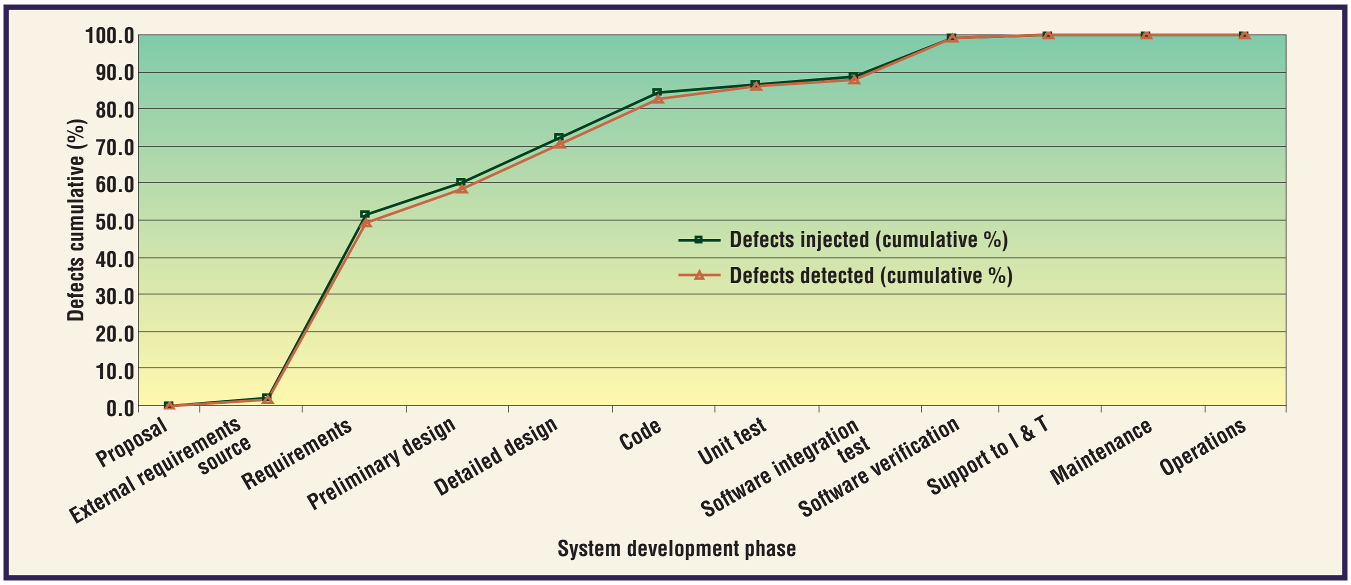


Figure 6. Software defect injection and detection phases. Cumulative distribution of software fault (or defect) injection and detection phases are summarized based on using peer reviews across 12 system development phases (3,418 faults, 731 peer reviews, 14 systems, 2.67 years). The small gap between the lines indicates out-of-phase faults.

large-scale systems. The 14 systems are the same as those discussed earlier on software requirements metrics. This analysis encompasses 3,418 faults from 731 peer reviews and benchmarks the fault injection and detection performance across the 12 system development phases. Figure 5a displays the distribution of fault detection across the 12 system development phases.

For each fault, developers identified the detection phase as well as the injection phase, which is the development phase in which the fault originated and was first detectable. Figure 5b displays the distribution of fault injection across the 12 system development phases. The distributions in Figures 5a and 5b are similar but not identical. The fault detection processes are imperfect, and developers sometimes miss faults and detect them in development phases that occur later than their injection phases. Figure 6 depicts the cumulative distribution of software faults across the injection and detection phases. The small gap between the fault injection and fault detection lines in Figure 6 identifies the faults that are not detected in the same phase in which they are injected, and they become "out-of-phase faults." Root cause analysis of the out-of-phase faults provides opportunities for improving the fault detection processes.

Detecting "in-phase" and "out-of-phase" faults provides benchmarks of fault detection effectiveness (Figures 7a and 7b, respectively). For 95 percent of the faults, the detection phase is the same as the injection phase (Figure 7a). This means that developers apply the software fault detection processes to successfully detect 95 percent of the faults in the same phase that they are injected. Out-of-phase faults originate in an earlier development phase, are undetected, and are detectable in a later phase (Figure 7b). The wide variation in detection effectiveness across phases helps identify opportunities for improving the fault detection processes.

Data Analysis for Software Fault Detection and Injection Metrics

Data analysis of the software fault detection and injection metrics reveals the following observations:

- The requirements phase dominates the fault detection with 47 percent of the faults (Figure 5a). Four other phases also provide substantial portions of the fault detection: preliminary design, detailed design, code, and verification.
- Approximately 50 percent of the faults are injected and detected by the requirements phase and 70 percent are injected and detected by the detailed design phase (Figure 6).
- Developers detect 95 percent of the in-phase faults (Figure 7a).
- Developers detect 47 percent of the out-of-phase faults (Figure 7b). The requirements, code, and verification phases show the highest levels of out-of-phase fault detection: 58, 64, and 100 percent, respectively (Figure 7b).

These fault metric analyses highlight that the vast majority of faults originate early in the system life cycles, so emerging technologies for representing, analyzing, and evaluating requirements and designs have the opportunity to yield dramatic improvements in fault injection rates. These analyses also reveal that although developers detected most (95 percent) of the in-phase faults throughout the life cycles,

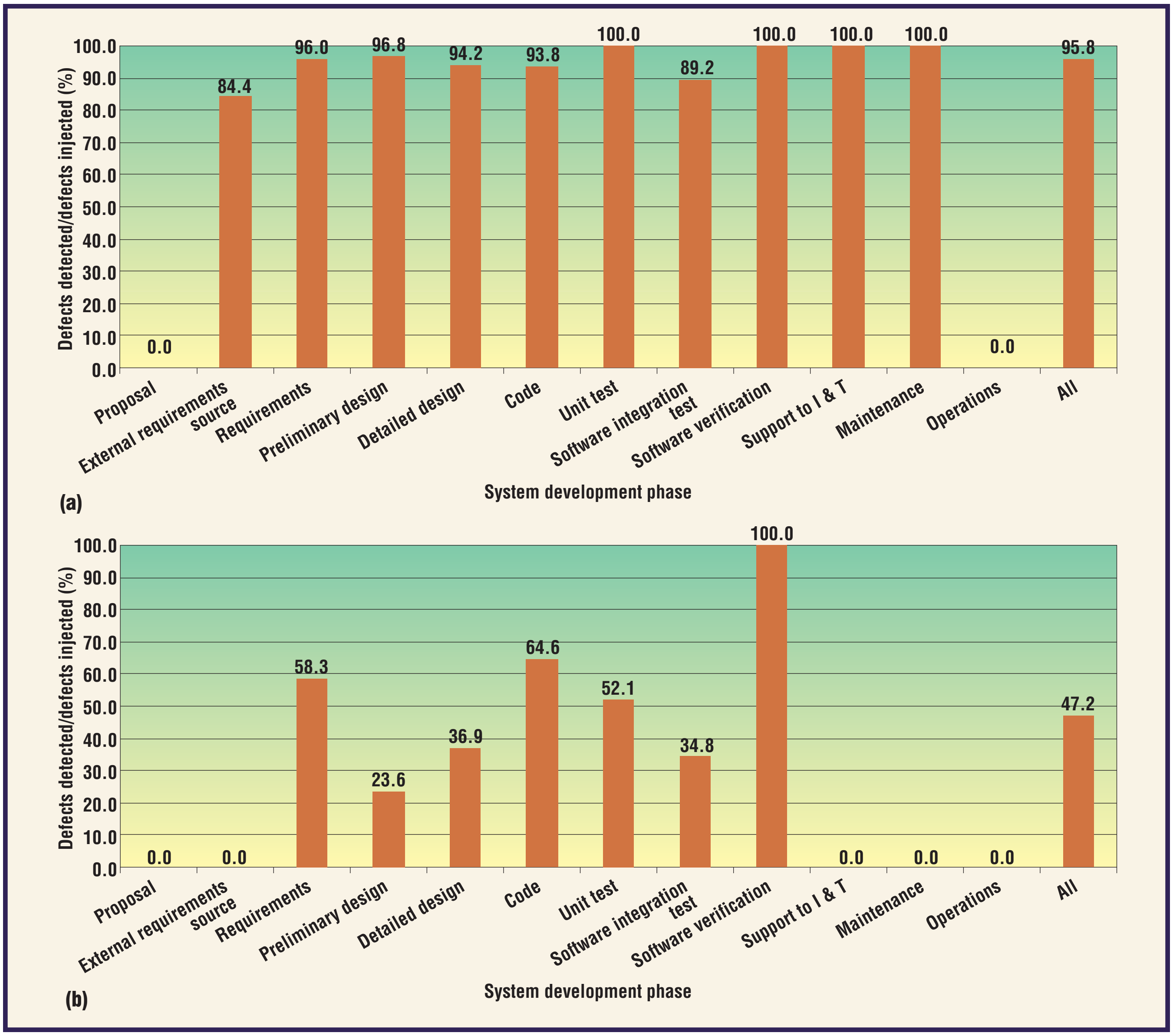


Figure 7. (a) Software faults (or defects) detected in the same development phase in which they were injected. (b) Software faults (or defects) detected in a development phase that were injected in an earlier development phase and undetected since the earlier development phase. These faults are summarized based on using peer reviews across 12 system development phases (3418 faults, 731 peer reviews, 14 systems, 2.67 years). "Injection phase" means the phase in which a fault originates.

developers critically need improvements in methods and techniques for detecting out-of-phase faults because they detected only 47 percent of those.

Future Research

Our ongoing research investigates principles for mining software repositories using analytics-driven dashboards for development and management of large-scale systems. Successful development, management, and improvement of large-scale systems require the creation of dashboards and underlying infrastructure to support data collection from organizations, projects, processes, products, teams, and resources. Flexible goal-driven dashboards and infrastructure enable support for different metrics for different users. Synergistic integration of dashboards and infrastructure enables numerous goal-driven capabilities, including user-specifiable interactive displays, flexible automated analyses, and customizable measurement breadth, depth, granularity, and frequency.

Our future research directions include further investigation of dashboards, underlying infrastructure, and software repositories with a complementary focus on leading indicators such as software requirements and design metrics. Software requirements and design metrics are especially useful because they are available much earlier in the life cycle than implementation metrics such as LOC, and they contribute to dashboards by enabling leading indicators for system scope, growth, volatility, and progress.

References

1. F.P. Brooks, *The Mythical Man-Month*, Addison-Wesley, 1975.
2. G. Bockle et al., "Calculating ROI for Software Product Lines," *IEEE Software*, vol. 21, no. 3, 2004, pp. 23–31.
3. B.W. Boehm, *Software Engineering Economics*, Prentice Hall, 1981.
4. J. Klein, B. Price, and D. Weiss, "Industrial-Strength Software Product-Line Engineering," *Proc. 25th Int'l Conf. Software Eng.* (ICSE 03), IEEE CS Press, 2003, pp. 751–752.
5. A. Engel and S. Shachar, "Measuring and Optimizing Systems' Quality Costs and Project Duration," *Systems Engineering*, vol. 9, no. 3, 2006, pp. 259–280.
6. R.W. Selby et al., "Metric-Driven Analysis and Feedback Systems for Enabling Empirically Guided Software Development," *Proc. 13th Int'l Conf. Software Eng.* (ICSE 91), ACM Press, 1991, pp. 288–298.
7. R.W. Selby and A.A. Porter, "Learning from Examples: Generation and Evaluation of Decision Trees for Software Resource Analysis," *IEEE Trans. Software Eng.*, vol. se-14, no. 12, 1988, pp. 1743–1757.
8. H. Scheffe, *The Analysis of Variance*, John Wiley & Sons, 1959.
9. B. Boehm, "Industrial Software Metrics Top 10 List," *IEEE Software*, vol. 4, no. 5, 1987, pp. 84–85.
10. L.J. Osterweil, "Software Processes Are Software Too," *Proc. 9th Int'l Conf. Software Eng.* (ICSE 87), ACM Press, 1987, pp. 2–13.
11. R. Smaling and O. de Weck, "Assessing Risks and Opportunities of Technology Infusion in System Design," *Systems Eng.*, vol. 10, no. 1, 2007, pp. 1–25.
12. W.S. Humphrey, "Characterizing the Software Process: A Maturity Framework," *IEEE Software*, vol. 5, no. 2, 1988, pp. 73–79.
13. J. Bosch, "Architecture-Centric Software Engineering," *Proc. 24th Int'l Conf. Software Eng.* (ICSE 02), ACM Press, 2002, pp. 681–682.
14. D. Harel, "Statecharts: A Visual Formulation for Complex Systems," *Science of Computer Programming*, vol. 8, no. 3, 1987, pp. 231–274.
15. *Software Engineering Laboratory: Database Organization and User's Guide, Revision 1*, tech. report SEL-81-102, Software Eng. Laboratory, NASA/Goddard Space Flight Center, July 1983.
16. R.W. Selby, "Enabling Reuse-Based Software Development of Large-Scale Systems," *IEEE Trans. Software Eng.*, vol. se-31, no. 6, 2005, pp. 495–510.

About the Author

Richard W. Selby is the head of software products at Northrop Grumman Space Technology and an adjunct full professor of computer science at the University of Southern California. His research focuses on development, management, and economics of large-scale mission-critical systems, software, and processes, and he has been responsible for developing and managing over 55 products with more than 330,000 system requirements and more than 11 million LOC. Selby received his PhD in computer science from the University of Maryland, College Park. He coauthored *Microsoft Secrets: How the World's Most Powerful Software Company Creates Technology, Shapes Markets, and Manages People* and edited *Software Engineering: Barry W. Boehm's Lifetime Contributions to Software Development, Management, and Research*. Contact him at rick.selby@ngc.com.

Use Case Pitfalls:
Top 10 Problems from Real Projects Using Use Cases

Susan Lilly
SRA International, Inc.
4300 Fair Lakes Ct.
Fairfax, VA 22033
susan_lilly@sra.com
703-227-5103

Abstract

One of the beauties of use cases is their accessible, informal format. Use cases are easy to write, and the graphical notation is trivial. Because of their simplicity, use cases are not intimidating, even for teams that have little experience with formal requirements specification and management. However, the simplicity can be deceptive; writing good use cases takes some skill and practice. Many groups writing use cases for the first time run into similar kinds of problems. This paper presents the author's "Top Ten" list of use case pitfalls and problems, based on observations from a number of real projects. The paper outlines the symptoms of the problems, and recommends pragmatic cures for each. Examples are provided to illustrate the problems and their solutions.

Introduction

Over the past few years, we have seen a number of projects make their first attempts at developing use cases. These projects have used use cases in a number of ways: as the entire system requirements specification, as part of the system requirements, as an analysis technique to elicit user requirements that were subsequently specified in other forms (e.g., traditional "shalls"), and as software subsystem-level requirements. The project teams that developed the use cases have included developers and/or analysts; in some cases the project teams have included customers or end users as well.

Although the project teams had little trouble getting started with use cases, many of them encountered similar problems in applying them on a larger scale. These problems include undefined or inconsistent system boundary, use case model complexity, use case specification length and granularity, and use cases that are hard to understand or never complete. These have been grouped and summarized here as a *"Top Ten"* list of use case pitfalls and problems, which may be encountered by inexperienced practitioners.

A sample problem is used to provide simple examples throughout this paper. *The Baseball Ticket Order System* is a computer system that is to be deployed to simplify customer sales for baseball games. Customers may view the season schedule and reserve tickets at kiosks placed in convenient locations, such as malls. Alternately, customers may call an 800 number and a phone clerk will reserve tickets for them. The customer may pay by credit card, or may pay at the time the tickets are picked up at the stadium on the day of the game.

The Top Ten List

Problem #1: The system boundary is undefined or inconsistent.

Symptom: The use cases are described at inconsistent system scope -- some use cases at business scope, others at system or even subsystem scope.

One element of the use case model is a labeled box that indicates the system boundary; the actors go outside of this box, and the use cases go inside. Before we determine the actors and use cases, we must be explicit about what we mean by "system." Is it a computer system? An application? A component or subsystem? Or is it a whole business enterprise? Use cases might be used to describe any of these "system" boundaries, but should only focus on one at a time. The actors and use cases appropriate at one system boundary are likely to be incorrect for a different system boundary. A common problem is the mix of both scopes in the same use case model, or even within a single use case specification.

Example: A *Kiosk Customer* uses the computer system to order tickets. Alternately, a *Phone Customer* may call the ticket business, and a *Phone Clerk* (an employee of the ticket business) may use the computer system to order tickets. Who are the actors? Figure 1 illustrates a mixed-up system boundary: The modelers have tried to show both the users of the business and the users of the system in the same use case model.

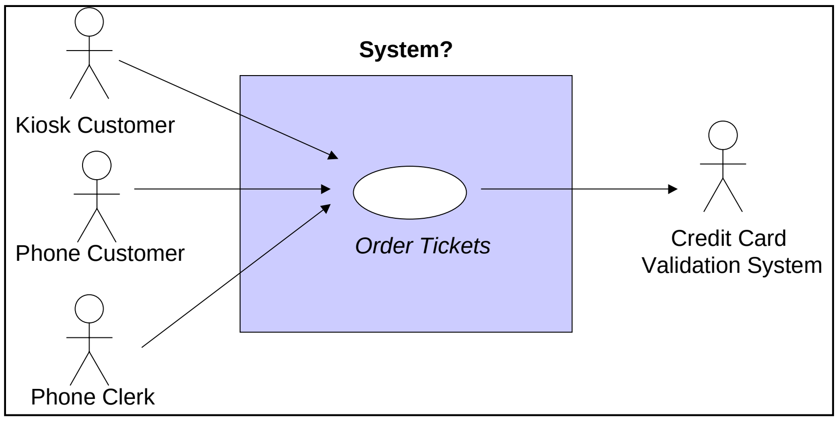


Figure 1: Use Case with Mixed-Up Scope

Cure: Be explicit about the scope, and label the system boundary accordingly. Say: "Yes, the business model is very interesting, but right now we are defining our use cases at the computer system scope" -- and then stick to it. *Example:* In Figure 2, the system boundary represents a computer system, and *Kiosk Customer* and *Phone Clerk* are actors who use the **Order Tickets** use case.

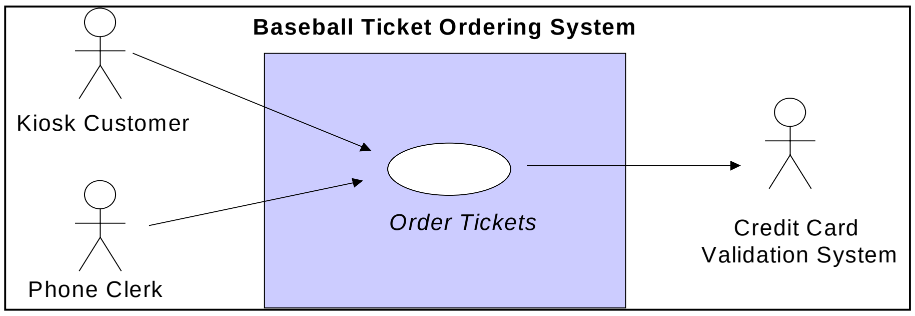


Figure 2: Use Case at Computer System Scope

In Figure 3, the system boundary represents a whole business enterprise. The actor, *Phone Customer*, is a user of the ticket business, but is not a user of the computer system. Both of these are appropriate ways to model; the choice between them depends on whether we are trying to define the requirements of a computer system (use Figure 2), or using use cases in business process modeling or reengineering (use Figure 3).

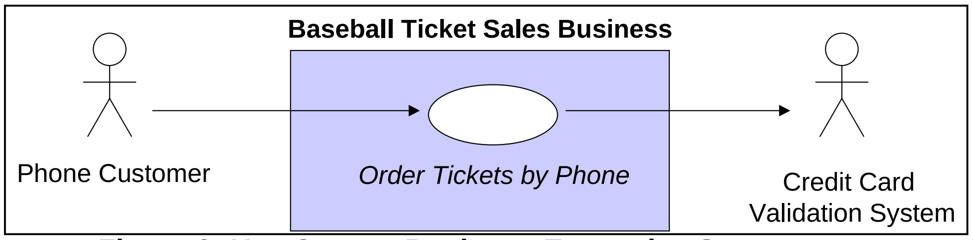


Figure 2: Use Case at Business Enterprise Scope

Symptom: Looking at the use case model, it's not really clear what's inside and what's outside the system. This problem often comes up when the use cases are modeled using a visual modeling/CASE tool (including the leading one on the market) that doesn't show the system boundary on the use case model.

Cure: Draw the system boundary (at least in your head). If the modeling tool does not draw a system boundary, place the use cases inside and the actors outside an imaginary box. *Example:* Figure 4 shows the same use case model, formatted in different ways. The model on the left has mixed up the actors and use cases; the one on the right has placed the use cases in the middle ("inside") with the actors on the "outside."

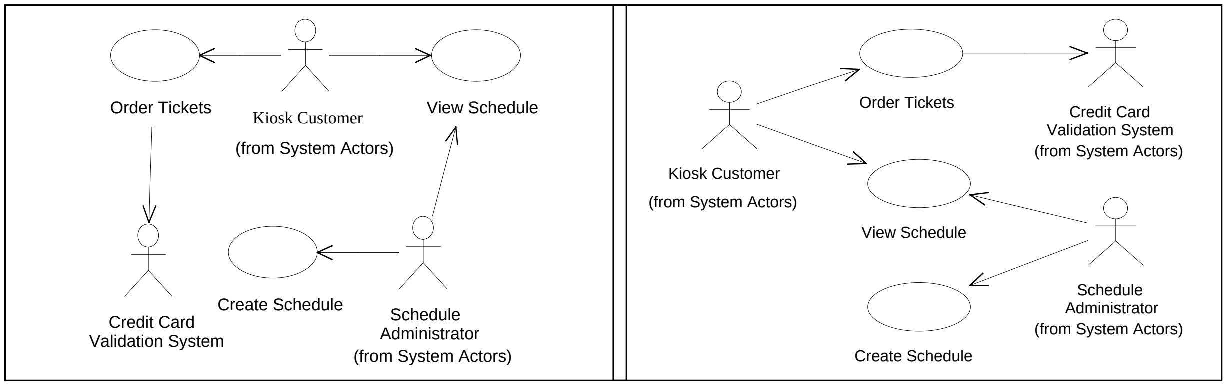


Figure 3: Use Case Model Formatting (bad and better)

Problem #2: The use cases are written from the system's (not the actors') point of view.

Symptom: The use case names describe what the system does, rather than the goal the actor wants to accomplish.

Cure: Name the use cases from the perspective of the Actor's goals. *Example:* **Process Ticket Order** and **Display Schedule** are things the system does (bad use case names). **Order Tickets** and **View Schedule** are goals of the system's users (good use case names).

Symptom: The steps in the use case specification describe internal functionality, rather than interactions across the system boundary.

Cure: Focus on what the system needs to do to satisfy the actor's goal, not how it will accomplish it.

Symptom: The use case model looks like a data/process flow diagram.
Cure: Watch out when the use case model includes use cases that are not directly associated with an actor, but are associated with **<<*uses*>>** or **<<*extends*>>** relationships. Sometimes this is an appropriate way to model the use cases. But many neophyte use case modelers (especially those who are programmers, or who have a process modeling background) misuse these associations, functionally decomposing the problem, rather than focusing on the interactions between actors and the system. Take a look at the specifications of used or extension use cases, to ensure that the steps in them describe interactions between the actor (of the base use case) and the system. If the steps are entirely focused on internal processing, the used and extended use cases are probably being used as a mechanism for functional decomposition. (If so, they don't belong in the use case model.)

Problem #3: The actor names are inconsistent.

Symptom: Different actor names are used to describe the same role. This is amazingly easy to do, since different sources of requirements often use variant names for the same thing -- and similar names for quite different things. When a problem is large, there are often multiple teams working on use case models for different parts of the problem, and the same (logical) actor may appear with variant names from model to model. *Example:* The role of the person who manages the online baseball schedule is called "Schedule Administrator" in one model, "Schedule Manager" in another, and "Scheduler" in a third.
Cure: Get agreement early in the project about the use of actor names (and other terms). Establish a *glossary* early in the project and use it to define the actors. The glossary should specify the actor name, its meaning, and any aliases that this name is known by. Include the glossary as an appendix to the use case document.

Problem #4: Too many use cases.

Symptom: The use case model has a very large number of use cases.
Cure: Make sure that the granularity of the use cases is appropriate. Use cases should reflect "results of value" to the system's users -- the attainment of real user *goals*.

- Combine use cases that describe trivial or incidental behavior that are actually fragments of the real use cases. Use cases are sometimes chopped into fragments when there is an attempt to associate user interface screens to use cases in a 1-to-1 relationship.
- Remove use cases that describe purely "internal" system processing ("internal" with respect to whatever system boundary is being used).

Example: In Figure 5, the Happy *Kiosk Customer* actor is associated with a use case called **Order Tickets** -- the customer's real goal in walking up to the kiosk in the mall. The Sad *Kiosk Customer* actor is associated with three different use cases. They all describe interactions between the *Kiosk Customer* and the system, but they represent incidental steps in the attainment of the actor's real goal (to order tickets). How did the "real" use case get split into three sub-goal use cases? The modelers were attempting to make a separate use case for each user interface element.

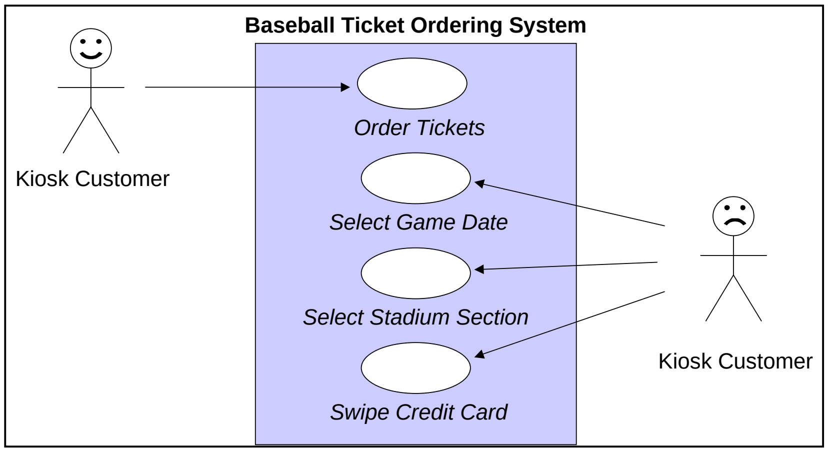
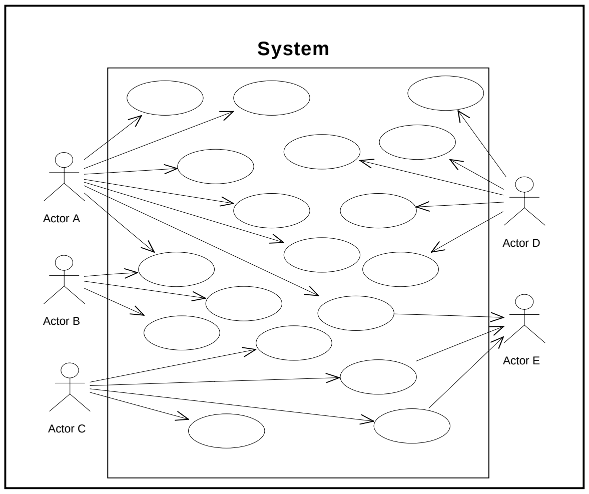


Figure 4: Real Use Cases vs. Incidental Actions

If the granularity of the use cases is right, but the system is simply very large, partition the set of use cases. Break the use case model into use case *packages*, each of which contains a cohesive set of use cases and a limited set of actors. *Example:* Figure 6 shows a use case model that has a large number of use cases. Figure 7 illustrates the same set of use cases, partitioned into 5 packages. Each package should contain a "cohesive" subset of the use cases, grouped around one or more actors who share common goals.



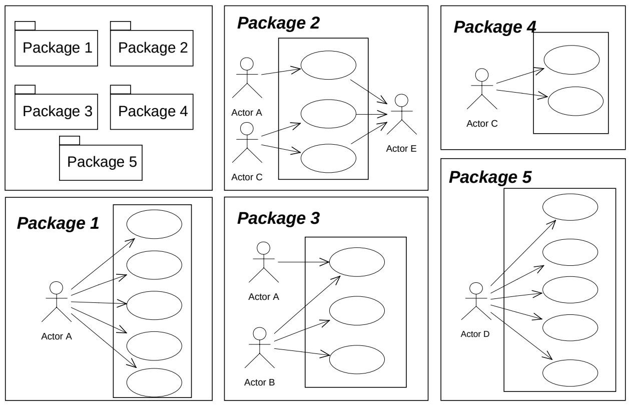


Figure 5: Model Needs Partitioning **Figure 6: Model with Packages**

Problem # 5: The actor-to-use case relationships resemble a spider's web.

Symptoms: (a) There are too many relationships between actors and use cases. (b) An actor interacts with every use case. (c) A use case interacts with every actor.

Cure: The actors may be defined too broadly. Examine actors to determine whether there are more explicit actor roles, each of which would participate in a more limited set of use cases. *Example: Employee* is very general, and is associated with a large number of use cases.

Phone Clerk and *Schedule Administrator* are more specific; each of these is associated with a smaller, more role-oriented set of use cases.

There may be cases where recognition of a more general class of actors helps to simplify a model. This often occurs where two or more actors are associated with the same set of use cases, because of some commonality in their roles. The resulting use case model has a spider's web of crossed lines between actors and use cases, as shown in Figure 8.

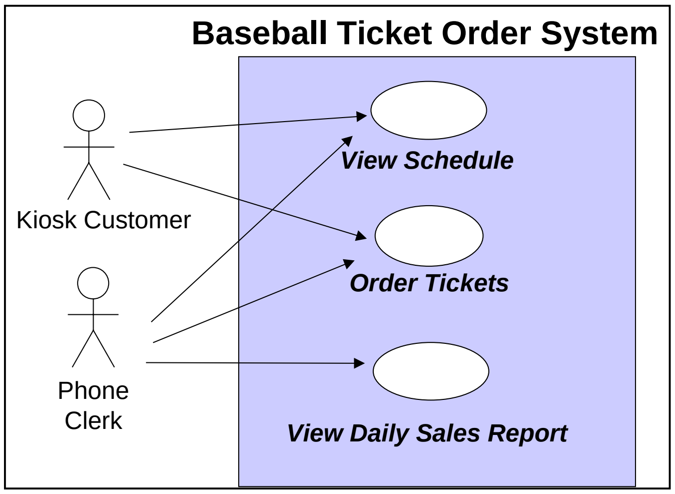


Figure 7: Actors with Overlapping Roles

The use case modeling notation provides a mechanism, **actor generalization**, for explicitly recognizing the commonality of actor roles. Figure 9 shows how the use case model can be redrawn with actor generalization to simplify the relationships between actors and use cases. This model says that *a Kiosk Customer* is a kind of *Ticketer* and that a *Phone Clerk* is a kind of a *Ticketer*. Any *Ticketer* may view a schedule or order tickets. A *Phone Clerk* (but not a *Kiosk Customer*) may additionally view a sales report.

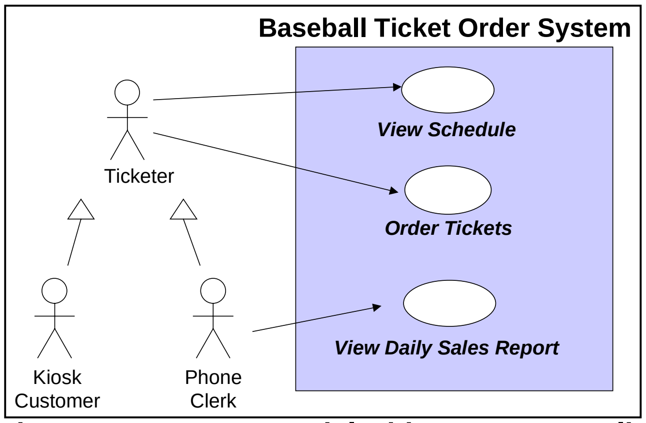


Figure 8: Use Case Model with Actor Generalization

Note that it would not have been correct to simply model *Phone Clerk* as a specialization of *Kiosk Customer*, in place of *Ticketer*. While the actor-to-use case relationships would be correct, the actor-to-actor relationship is semantically unsound: A *Phone Clerk* is **not** a kind of *Kiosk Customer*. (I saw that example in a recent use case book, which modeled a *Sales Rep* actor as a subclass of a *Customer* actor, in order to inherit the overlapping use case relationships.)

Problem #6: The use case specifications are too long.

Symptom: A use case specification goes on for pages.

Cure: The granularity of the use case may be too coarse. *Example:* **Use Schedule** (a use case that includes everything any user might want to do with a schedule) is too broad. More narrowly defined, specific use cases (such as **View Schedule** and **Create Schedule**) tend to be shorter and easier to understand.

Alternately, the granularity of the steps in the use case may be too fine. The steps may be too detailed or include purely internal processing (implementation). Rewrite them to focus on the essential interaction.

Problem #7: The use case specifications are confusing.

Symptom: The use case lacks context; it doesn't "tell a story."

Cure: Include a *Context* field in your use case specification template to describe the set of circumstances in which the use case is relevant. Make sure that the *Context* field puts each use case in perspective, with respect to the "big picture" (the next outermost scope). Don't just use it to summarize the use case.

Symptom: The steps in the normal flow look like a computer program.

Cure: Rewrite the steps to focus on a set of essential interactions between an actor and the system, resulting in the accomplishment of the actor's goal.

– Break out conditional behavior ("If...") into separately described alternate flows, leaving the normal flow shorter and easier to understand.
– Use case steps are not particularly effective for describing non-trivial algorithms, with lots of branching and looping. Use other, more effective techniques to describe complex algorithms (e.g., decision table, decision tree, or pseudocode).
– Make sure that the steps don't specify implementation. Focus on the external interactions. Consider expressing some of the behavior as "rules," rather than algorithms.

Problem #8: The use case doesn't correctly describe functional entitlement.

Symptom: The associations between actors and use cases doesn't correctly or fully describe who can do what with the system. This problem seems to occur for two reasons:

– The use case modelers were trying to be "object oriented," by making fat use cases that include all possible actions that might be performed on a business object. (I call these "CRUD use cases," since they often contain flows for creating, reading, updating, and deleting the object.) These use cases often have names that include the words "maintain," "manage," or "process."
– The use case modelers were trying to match up use cases to user interface screen. Faced with a view screen, that could also be edited (by a user with the right authority), they combined viewing and updating into a single use case that relates to the single screen design.

Example: Figure 10 shows a use case **Process Game Schedule**, that describes everything that any actor might want to do with a game schedule. Its specification has a "normal flow" for viewing the schedule, and alternate flows for updating the schedule. The *Kiosk Customer* actor may use the normal flow, but cannot use the alternate flow. Only the *Schedule Administrator* is functionally entitled to perform the schedule update.

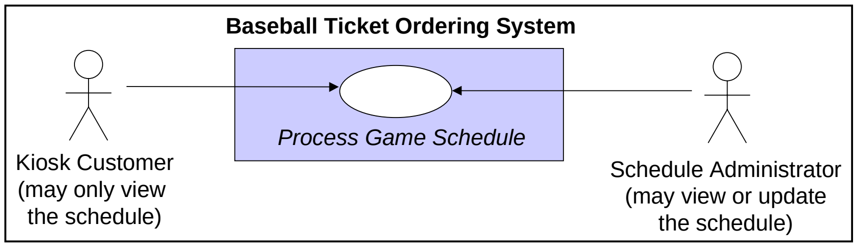

Baseball Ticket Ordering System
Process Game Schedule
Kiosk Customer (may only view the schedule)
Schedule Administrator (may view or update the schedule)

Figure 9: Confusing Functional Entitlement

Cure: Make sure that each actor associated with a use case is completely entitled to perform it. If an actor is only functionally entitled to part of the use case, the use case should be split. *Example:* The **Process Game Schedule** use case should be split into two: **View Game Schedule** and **Update Game Schedule**, as shown in Figure 11. Now it is clear, at a glance, that the *Kiosk Customer* may view, but not update, a schedule.

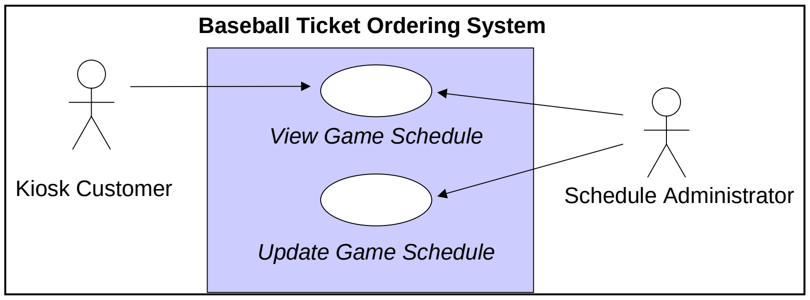


Figure 10: Use Cases with Correct Functional Entitlement

Problem #9: The customer doesn't understand the use cases.

Symptom: The customer doesn't know anything at all about use cases, but has been given a use case-based requirements document for review or approval. (Of course it's best when customers/end users have been included in the use case development. However, the person who reviews or approves the requirements may not have been involved in developing the use cases.)

Cure: Teach them just enough to understand.

– Put a short (1-2 page) explanation of use cases in the use case document, as a preface or appendix. The explanation should include a key to reading the model and specifications, and a simple example.
– Lead a short training session when use case document is distributed for review.
– Think long and hard about using **<<*uses*>>** and **<<*extends*>>** relationships in the use case model. They are a modeling convenience, but are not at all intuitive to the inexperienced reviewer.

Symptom: The use cases don't tell a story.

Cure: Add information to tell the story:

– Include a *Context* section in the use case template.

- Add an overview section that provides context to a set of related use cases (e.g., a package), and use this section to "tell the story."
- Include other kinds of models as needed. Often, a single use case will result in a state change to a major domain object, but the use case model alone won't tell the story of how the object changes state across many use cases over time. A state model (state transition diagram) of a major domain object may be an excellent way to show how several related use cases fit together over time.

Symptom: The use case organization doesn't match the way the customer thinks of the problem.

Cure: Determine what strategy for organizing the use cases makes the most sense to the customer. Listen to how the customer describes the business.

- How to partition the use cases into packages: Break out the use cases by major roles/actors, or by major events in the customer's business. *Example:* The customer talks a lot about "Spring Setup" -- when they put the new game schedule, stadium section definitions, and ticket prices into the system. Even if that's not the way the system developers think about the system, that's the package organization that makes sense to the customer.
- How to order use cases within a package: Order the use cases "chronologically," to describe a story of system use over time. Don't order the use cases alphabetically!

Symptom: Use case is written in "computerese."

Cure: Watch out for computer slang that is not part of the customer's vocabulary. *Example:* Say "The system displays the result screen," rather than "The result screen is invoked."

Symptom: The customer just hates the use cases.

Cure: Deliver what the customer wants. This doesn't mean that use cases can't be used as a requirements elicitation technique, if use cases are really the right technique for the job. But they might not be a primary delivered work product. *Example:* One customer has its own requirements document template, and it's not use-case based. But the system is highly operational in nature, and we feel that use cases are the best approach for eliciting and modeling the requirements. We perform the use-case based analysis, and then write the requirements in the format that the customer wants, based on what we learned in that analysis. The use cases might be included in an appendix to the requirements document, or they might not be a deliverable at all.

Problem #10. The use cases are never finished.

Symptom: Use cases have to change every time the user interface changes.

Cure: *Loosely couple* the user interface details and use case interactions. The user interface design is likely to change, and you don't want your system requirements to be dependent on design. (The dependency goes the other way -- the user interface design must satisfy the use case requirements.) A little coupling is okay -- "low fidelity" pictures of the user interface can aid understanding of the use case. But don't overly tie the fundamental interactions to the UI mechanisms (which are more likely to change). In the flows, focus on the essentials of what the actor does (e.g., "selects a game," "submits a request") rather than how the interaction is done (e.g., "double-click on the Submit button").

Specify use case "triggering" events as preconditions (e.g., "user has selected a game, and requested to order tickets"), rather than screen navigation details. Keep the screen navigation information in a (separate) user interface design document, not in the use case model.

Symptom: The use cases require change every time the design changes.

Cure: The easy answer is "Don't put design in your use cases." That's generally good advice when the use cases are at a computer system scope. The use cases should record what the system must do, not the design/implementation details. Make sure that your use case steps are not unnecessarily low level; that is, they should completely specify what they system must do, but no more than that. Put design information discovered during analysis into a separate Design Guidance document.

When use cases are defined at a subsystem scope, changes in system-level design (e.g., how functionality is partitioned between subsystems) can affect the subsystem requirements. Until the system design is stable (and explicitly documented), the subsystem requirements, including the use cases, will not stabilize.

Symptom: There are so many possible alternate cases!

Cure: Watch out for "analysis paralysis." There is a point at which the requirements are adequately specified, and further analysis and specification does not add quality. Cover the "80%" cases; do your best on the rest within the allocated budget of time and money.

Symptom: The requirements are just unknown.

Cure: Use cases have a simple, informal, and accessible format. This may lead to the deceptive conclusion that developing use cases is easy. However, the simplicity of the format does not mean that the requirements analysis process is any less critical or any easier. *Use cases are a mechanism for defining and documenting operational requirements, not magic.*

Conclusions

The pitfalls and problems described in this paper are not an indictment of use cases, but rather problems in the *application* of use cases by inexperienced practitioners. In our experience, most use case development teams include inexperienced members. Use cases may be a new technique to the organization, and are being used by the development team for the first time. Even when the analysts or system developers have experience with use cases, other team members may not. The ideal use case team includes customers, end users, and/or domain experts. In most cases, these team members will have no prior experience with use cases. The simplicity of the use case modeling notation and natural-language specifications make use cases extremely accessible to such team members. They may fully participate in the use case modeling and specification, but are likely to encounter the pitfalls described in this paper.

One suggestion for teams in which some or all of the members are new to use cases is perform periodic informal "in-progress" reviews of the use case models and specifications, in order to catch problems early in the development, and to educate the team members. Of course, a formal review or inspection of a finished use case document is also appropriate. The reviews can be made more effective by the use of a checklist to help identify these common problems. An example of such a checklist is available by email on request from the author.

Using Quality Models in Software Package Selection

Xavier Franch and Juan Pablo Carvallo, *Universitat Politècnica de Catalunya*

The growing importance of commercial off-the-shelf software packages[1] requires adapting some software engineering practices, such as requirements elicitation and testing, to this emergent framework. Also, some specific new activities arise, among which selection of software packages plays a prominent role.[2]

This methodology drives the construction of domain-specific quality models in terms of the ISO/IEC 9126-1 quality standard. These models can be used to describe the quality factors of software packages uniformly and comprehensively and to state requirements when selecting a software package in a particular context.

All the methodologies that have been proposed recently for choosing software packages compare user requirements with the packages' capabilities.[3–5] There are different types of requirements, such as managerial, political, and, of course, quality requirements.

Quality requirements are often difficult to check. This is partly due to their nature, but there is another reason that can be mitigated, namely the lack of structured and widespread descriptions of *package domains* (that is, categories of software packages such as ERP systems, graphical or data structure libraries, and so on). This absence hampers the accurate description of software packages and the precise statement of quality requirements, and consequently overall package selection and confidence in the result of the process.

Our methodology for building structured quality models helps solve this drawback. (See the "Related Work" sidebar for information about other approaches.) A structured quality model for a given package domain provides a taxonomy of software quality features, and metrics for computing their value. Our approach relies on the International Organization for Standardization and International Electrotechnical Commission 9126-1 quality standard,[6] which we selected for the following reasons:

- Due to its generic nature, the standard fixes some high-level quality concepts, and therefore quality models can be tailored to specific package domains. This is a crucial point, because quality models can dramatically differ from one domain to another.
- The standard lets us create hierarchies of quality features, which are essential for building structured quality models.
- The standard is widespread.

After we build the quality model, we can state the domain requirements as well as package features with respect to the model. We can then use the framework to support negotiation between user requirements and product capabilities during software package selection (see Figure 1).

The ISO/IEC software quality standards

Among the ISO and ISO/IEC standards related to software quality are the families of 9126 and 14598 for software product quality and evaluation. These standards can be used in conjunction with others concerning the software life cycle (ISO/IEC 12207), process assessment (ISO/IEC 15504), and quality assurance processes (ISO 9001).

ISO/IEC 9126-1 specifically addresses quality model definition and its use as a framework for software evaluation. A 9126-1 quality model is defined by means of general software characteristics, which are further refined into subcharacteristics, which in turn are decomposed into attributes, yielding a multilevel hierarchy. At the bottom of the hierarchy are measurable software attributes, whose values are computed using some metric. Throughout this article, we refer to characteristics, subcharacteristics, and attributes as quality entities, and we define quality requirements as restrictions over the quality model.

Table 1 shows the six quality characteristics defined in the 9126-1 quality standard and their decomposition into subcharacteristics.

Building ISO/IEC 9126-1 quality models

Our methodology comprises six steps (see Figure 2). Also, we consider a preliminary activity (Step 0) for analyzing the package domain. Although we present these steps as sequential, they can be intertwined or repeated.

Step 0: Defining the domain

First, you must carefully examine and describe the domain of interest, with the help of experts. To describe the domain, we recommend using conceptual modeling to keep track of relevant concepts.

One of the biggest problems is the lack of standard terminology among a domain's software packages. Different vendors refer to the same concept by different names, or even worse, the same name might denote different concepts in different packages. Discovering all these conflicts during this preliminary step is essential to avoid semantic mismatches throughout the software selection process.

The e-learning domain provides more than one example. Consider the term *virtual classroom*. Some packages use this term and *course* as synonymous, referring to the contents of particular subjects. Other packages use different semantics for *virtual classroom*—meaning, for instance, the interface (such as the base

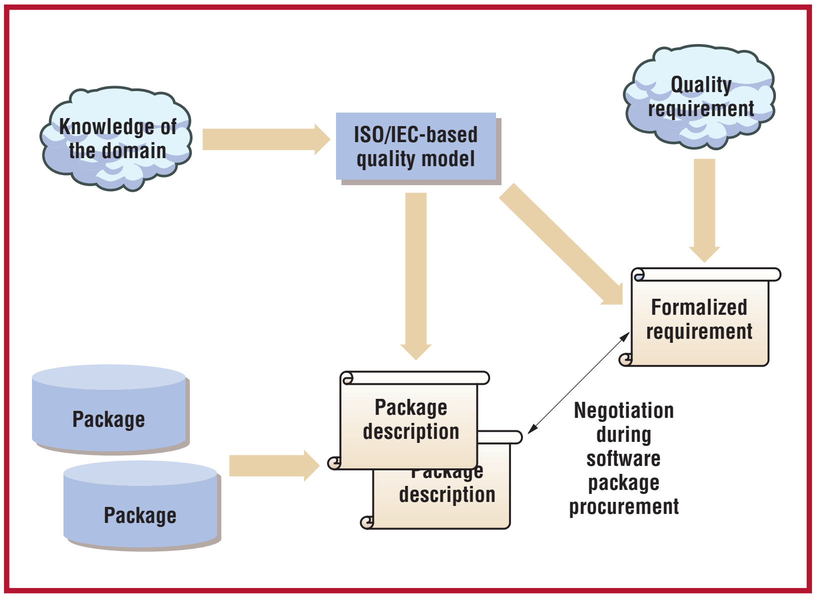


Figure 1. Using a quality model in software procurement.

Table 1
ISO/IEC 9126-1 characteristics and subcharacteristics

Characteristics	Subcharacteristics
Functionality	Suitability
	Accuracy
	Interoperability
	Security
	Functionality compliance
Reliability	Maturity
	Fault tolerance
	Recoverability
	Reliability compliance
Usability	Understandability
	Learnability
	Operability
	Attractiveness
	Usability compliance
Efficiency	Time behavior
	Resource utilization
	Efficiency compliance
Maintainability	Analyzability
	Changeability
	Stability
	Testability
	Maintainability compliance
Portability	Adaptability
	Installability
	Coexistence
	Replaceability
	Portability compliance

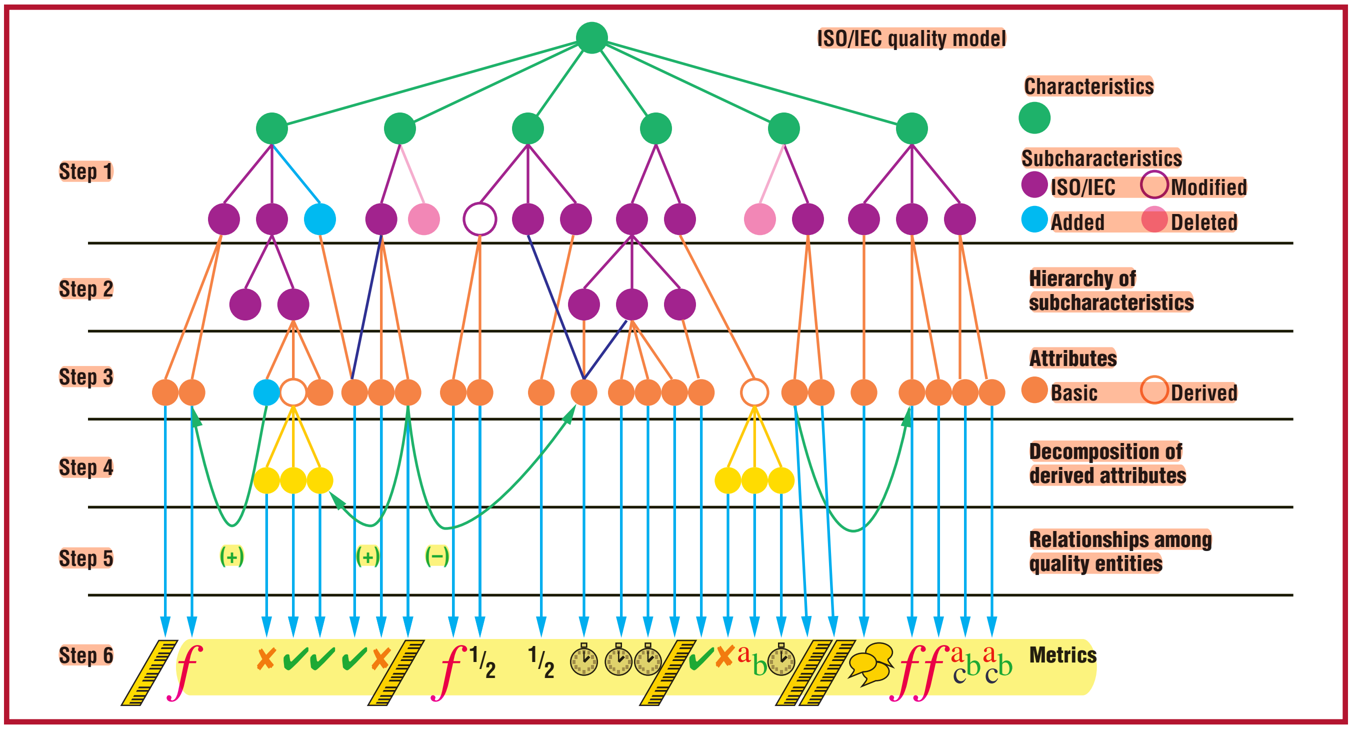


Figure 2. Our six-step methodology.

URL where the contents of a course are to be placed) or the list of tasks to be performed during the course.

Step 1: Determining quality subcharacteristics

The decomposition of characteristics into subcharacteristics that appear in the standard is quite reasonable and should be used unless good reasons for not doing so come out during domain analysis. In these cases, you might add new subcharacteristics specific to the domain, refine the definition of existing ones, or even eliminate some. For example,

- In the domain of ERP systems, you might add Functionality subcharacteristics for tracking the company areas covered (such as finance or staff).
- In the domain of data structure libraries, you might refine the Time Behavior subcharacteristic as "execution time of the methods provided by the classes inside the library."
- In domains that are purely pieces of software to be integrated in a system, you might omit the Attractiveness subcharacteristic as defined in the standard (which keeps track of quality attributes such as color use or graphical appearance) because it does not apply.

Step 2: Defining a hierarchy of subcharacteristics

Typically, you can further decompose subcharacteristics with respect to some factors, yielding a hierarchy.

A frequent situation appears in the Suitability subcharacteristic. Successful software packages tend to include applications that were not originally related to them. A usual reason for this is that product suppliers often try to include features to differentiate their products from their competitors'. These added applications are usually not shipped in the original packages; they are offered separately, as extensions. But, they often are referenced as essential to the functions the package provides (although additional software is often required). As a result, we can split the Suitability characteristic into two subcharacteristics, Basic Suitability and Added Suitability, keeping track of each inside the model but in a clearly separated way.

Step 3: Decomposing subcharacteristics into attributes

Quality subcharacteristics provide a comprehensible abstract view of the quality model. But next, we must decompose these abstract concepts into more concrete ones, the quality attributes. An attribute keeps track of a particular observable feature of the packages in the domain. For example, attributes in the Learnability subcharacteristic might include the Quality of the Product's Graphical Interface, the Number of Languages Supported, and the Quality of the Available Documentation. We must define attributes precisely to

Related Work

Other authors have proposed quality models as a basis for software evaluation, but most of these proposals deal with measuring custom software instead of selecting software packages. As a result, they focus mainly on internal attributes and their implications instead of external attributes, as is our aim.

However, similarities exist. Geoff Dromey suggests the use of ISO/IEC 9126 standards, although he does not require it.[1] In fact, he uses different models for parts of his proposal, making model reusability more difficult. There are two other fundamental differences between our and Dromey's work: he does not include metrics as part of the model, and his concept of quality is context-free, whereas we introduce the idea of context-dependent quality attributes as a means to measure quality, taking into account the package's context of use.

The Squid method focuses on quality for in-house software.[2] The aim of Squid is wider than our approach; it covers quality planning, control, and evaluation as well as modeling. Concerning quality models, Squid is closer to our proposal than is Dromey's: it takes external attributes as a primary concern (mapping external attributes onto internal ones to be observed in the product), it considers quality models to be context-dependent, and both methods search for reusability by identifying robust attributes. On the other hand, Squid does not propose any particular procedure for building the quality model, nor does it require the ISO/IEC standards. Perhaps the most significant difference is that Squid does not distinguish clearly between the quality model itself and its context of use (for example, the model includes upper target values for attributes), compromising model reuse in other contexts.

Our approach is also related to goal-oriented modeling.[3] Typical goal-oriented analysis focuses on modeling a given, ongoing software system, making explicit the participating agents and the goals and their decomposition. In contrast, our quality model approach is not bound to any particular system. It is as context-free as possible, without any assumptions about the model's context of use. Nevertheless, we are currently aiming at using goal-oriented models for modeling the rationale behind the construction process.[4]

References

1. R.G. Dromey, "Cornering the Chimera," *IEEE Software*, vol. 20, no. 1, Jan./Feb. 1996, pp. 33–43.
2. J. Bøegh et al., "A Method for Software Quality Planning, Control, and Evaluation," *IEEE Software*, vol. 23, no. 2, Mar./Apr. 1999, pp. 69–77.
3. E. Yu, "Towards Modeling and Reasoning Support for Early-Phase Requirements Engineering," *Proc. 3rd IEEE Int'l Symp. Requirements Eng.* (ISRE), IEEE CS Press, Los Alamitos, Calif., 1997, pp. 226–235.
4. B. Kitchenham and S.L. Pfleeger, "Software Quality: The Elusive Target," *IEEE Software*, vol. 20, no. 1, Jan./Feb. 1996, pp. 12–21.

clarify the underlying quality concepts that they represent and to link them with the appropriate subcharacteristics.

Because we focus on the framework of COTS package selection and package suppliers rarely give access to the package code, we are interested in external rather than internal attributes.

As the standard itself mentions, it is not possible from a practical point of view to measure all the subcharacteristics for all parts of a large software product, but it is certainly feasible to create a complete list with the most relevant ones. Concepts are the key elements when selecting quality attributes; the goal is to define a general framework for many applications of the same brand, not for a particular product.

When you decompose, some attributes are suited for more than one subcharacteristic. For instance, an attribute for the Fault Tolerance subcharacteristic in the domain of data structure libraries might be the Type of Error Recovery Mechanism. But in fact, this attribute might also be a constituent of the Testability subcharacteristic (a powerful error recovery mechanism makes testing the library easier). Therefore, we do not force attributes to appear in a single subcharacteristic; they can be part of many of them. The ISO/IEC standard allows this situation to occur.

Step 4: Decomposing derived attributes into basic ones

Some of the attributes emerging in Step 3 (for example, the Number of Languages Supported) can be directly measurable given a particular product, but others might still be abstract enough to require further decomposition. This is the case with the Quality of Graphical Interface attribute mentioned earlier; quality might depend on such factors as user friendliness, depth of the longest path in a browsing process, and types of interface supported. Therefore, we distinguish between *derived* and *basic* attributes. Derived attributes should be decomposed until they are completely expressed in terms of basic ones.

We can completely define derived attributes in terms of their components. However, in some situations, giving a concrete definition of the quality interface attribute could be considered harmful, because it would force us always to use the same definition without considering the requirements of a particular context.[7] Sometimes, requirements give more importance to the user friendliness factor (for example, for nonskilled users), sometimes to its type (for interoperability purposes), and so on. In this case, the definition of the derived attribute is delayed until a particular selection process takes place. The first case of

To obtain a really complete quality model, you also must explicitly state the relationships between quality entities.

derived attributes is context free, the second is context dependent.

Step 5: Stating relationships between quality entities

To obtain a really complete quality model, you also must explicitly state the relationships between quality entities. The model becomes more exhaustive, and, as an additional benefit, the implications of quality user requirements might become clearer.

Given two quality entities A and B, we can identify various types of relationships:

- *Collaboration*. Growing A implies growing B. For instance, the Security subcharacteristic collaborates with the Maturity one.
- *Damage*. Growing A implies decreasing B. For instance, the Type of Error Recovery Mechanism attribute collides with the Time Behavior subcharacteristic: the more powerful the mechanism is, the more slowly the program runs.
- *Dependency*. Some values of A require B fulfilling some conditions. For instance, having an exception-based error recovery mechanism requires that the programming language offer exception constructs.

In addition, you might build more elaborated types of these relationships.[8]

Step 6: Determining metrics for attributes

You must not only identify the attributes but also select metrics for all the basic attributes as well as metrics for those derived context-free attributes. You can use the general theory of metrics for this purpose. Also, ISO/IEC is currently working on writing the 9126-2 external metrics standard.[9]

Metrics for basic attributes are quantitative—for example, existence of some kind of data encryption, depth of the longest path in a browsing process, supported protocols for data transmission, and so on. Derived context-free attributes might be either quantitative or qualitative, with explicit formulas computing their value from their component attributes.

Some attributes require a more complex representation, yielding to structured metrics. Examples are sets (for example, a set of labels for the languages supported by the interface) and functions. Functions are especially useful for attributes that depend on the underlying platform. For instance, many attributes related to the Time Behavior subcharacteristic might fall into this category.

Metrics for some quality attributes can be difficult to define. However, as the standard states, having rigorous metrics is the only way to obtain a quality model useful for doing reliable comparisons.

A case study: Mail servers

As electronic mail services have grown in importance, companies are increasingly using them to improve inside and outside communication and coordination. An overwhelming number of mail-related products are available, and organizations face the problem of choosing among them the ones that best fit their needs. For some companies, an inappropriate selection would compromise their success. Selection relies on manufacturing documentation, public evaluation reports, others' experience, and hands-on experimentation. These information sources can be inaccurate, not fully trustable, and costly. For all these reasons, having a good quality model is especially useful in the domain of mail server packages, so we will use it to illustrate our general methodology (we will skip the preliminary step). The model is available at www.lsi.upc.es/dept/techreps/html/R02-36.html.

Step 1: Determining quality subcharacteristics

The subcharacteristics suggested in the 9126-1 standard are complete enough to be used as a starting point. We have adopted them with some minor modifications in their definition.

Step 2: Defining a hierarchy of subcharacteristics

According to the criteria mentioned earlier, we split the Suitability subcharacteristic into Mail Server Suitability and Additional Suitability subcharacteristics. Some examples of applications included in mail servers are chat, instant messaging, whiteboarding, videoconference, and workflow project management tools.

You might be tempted to apply this decomposition principle as often as possible, but you must do so carefully. Consider the following situation. In some contexts, you could consider the attributes categorized under the Operability subcharacteristic of Usability from two different viewpoints: the general user's and the administrator's. Because this is the

Table 2

Some relationships among quality attributes

Characteristics	Subcharacteristics	Attributes	Functionality	Efficiency
			Security	Time behavior
			Secure email protocols	Average response time
Functionality	Security	Certification system	Depend on	Collide with
		Encryption algorithm	Depend on	Collide with
Reliability	Recoverability	Online incremental backup		Collide with
		Single-mailbox backup and recovery		Collaborate with
		Online restore		Collide with
		Dynamic log rotation		Collide with
		Event logging		Collide with
Efficiency	Resource behavior	Concurrent mail users per server		Collide with
		Number of active Web mail clients		Collide with
		Management of quotas on message and mail file size		Collaborate with
		Single copy store		Collaborate with

case for mail server products, we considered whether it was convenient to divide this subcharacteristic into two. But we observed that general user operability on mail servers depends on the mail client and the privileges given by the administrator. We did not find attributes related to clients that were independent of those related to administrators, so we decided to keep only one subcharacteristic.

Step 3: Decomposing subcharacteristics into attributes

We had to research the domain extensively to identify the attributes; then, we had to assign them to subcharacteristics. Contrary to what you might expect, this process is neither simple nor mechanical.[10] Here are some of the problems you can run into:

- The number of elements can get very high, making handling them difficult.
- In some cases, the values of attributes can be confused with the attributes themselves.
- Sometimes attributes represent more than one concept and must be split. For example, we finally split the former Average Response Time attribute into two, the Average Response Time itself and Message Throughput.
- As mentioned earlier, some attributes are suited for more than one characteristic. For instance, Message Tracking and Monitoring might be seen as a functional attribute that grants Accuracy, or as an analyzability attribute of the Maintenance characteristic.

Hands-on experimentation is necessary to obtain really independent information. For instance, the documentation of almost every product mentioned attributes such as administrative or expert analysis tools but gave very vague descriptions of them. We had some specific hands-on experience to better understand these concepts. These experiences turned out to be valuable to validate some results obtained in this step.

Step 4: Decomposing derived attributes into basic ones

We have identified several decomposable attributes. For instance, we decomposed the Resource Administration attribute (characteristic Usability, subcharacteristic Operability) into the following basic attributes: Maximum Storage Time of Mail Messages, Maximum Time of Life for Inactive Accounts, Mailbox Quotes, Mail File Sizes, and Management of Groups of Servers.

Step 5: Stating relationships between quality entities

With about 160 attributes, it is quite natural that there are a lot of relationships between them. Table 2 presents some dependencies, collaborations, and damages—the attributes in the rows *depend on*, *collaborate with*, or *collide with* attributes in the columns. For instance,

- If you select a certification system, you must also use an encryption algorithm, because it is needed to grant confidentiality.
- The SMTP (Simple Mail Transfer Protocol) requires the MIME (Multipurpose Internet Mail Extensions) Support attribute to be true when sending multimedia attachments.

Table 3

Example quality requirements

1	Spanish language support
2	Support for the most commonly used certification standard
3	Support for accessing the server from other applications
4	Protection against viruses and any other risks
5	Mail delivery notifications, possibility of configuring parameters such as maximum number of delivery retries and time between them
6	Transmission time less than 1 minute for messages without attachments, and no more than 5 minutes per Mbyte for those with attachments

Step 6: Determining metrics for attributes

We determined metrics for all the basic and context-free attributes in the model, following the guidelines given earlier. We can evaluate some attributes by simple observation, such as Maximum Account Size and Default Folders Provided. Others are difficult to define. For example, Thwarting Spammers and Handling Bulk Junk Mail depend on the support of filters for incoming messages. Average Response Time and Message Throughput depend on hardware platform as well as other attributes such as Number of Concurrent Users or Message Sizes. In these cases, we must define the metrics as functions.

Package and requirement descriptions

Once we build the quality model for a package domain, we can describe packages in this domain and express the quality requirements for modeling a company's package procurement needs.

When we try to describe package quality characteristics, it turns out to be very difficult to find complete and reliable information on them. Manufacturers tend to give a partial view of their products. Either they put so much emphasis on their product's benefits, without mentioning weaknesses, or they give only part of the truth, making the product seem capable of more than it can really do. Some third-party reports look independent, but they have been refuted for the parties involved. Other noncommercial articles compare features but are often based on the evaluators' limited knowledge of the tools and their particular tastes, more than on serious technical tests.

The quality model can be used for describing quality requirements in a structured manner. In the mail server domain, we have introduced complete sets of quality requirements that have appeared in real mail server selection processes with very different characteristics (ranging from a public institution serving 50,000 people to a small software consultant and Internet service provider company).

Table 3 presents some requirements that illustrate typical situations we found when expressing requirements in terms of the quality model. Requirements such as 1 or 2 can be directly mapped into a single attribute of the model. The only difference is that Requirement 2 demands expert assessment to do the mapping from the expression "most commonly used certification standard" to a concrete value of the corresponding attribute, namely "X.509."

Requirements 3 and 4 are too general (what do "other applications" and "other risks" mean?). A more detailed specification is necessary to better classify them.

Requirement 5 either requires or implies a mixture of functionalities, which involve several attributes. Although we might need further feedback to better classify this kind of requirement, we succeeded in classifying this particular one.

Some requirements were originally expressed incorrectly but somehow were understandable. This was the case with Requirement 6, which was not accurate because the one-minute limit for messages without attachments could be unfeasible when they include a lot of data inline (for example, annual company reports). So, we reformulated it using the attributes' average response time and throughput, resulting in a requirement that could be satisfied.

Once you have incorporated all the requirements for a particular company into the model (after completing, discarding, and reformulating them), you can compare them extensively with respect to available package descriptions. This lets you detect differences between products as well as determine to what extent they cover the expressed needs, thereby facilitating the package procurement process. Once you have expressed all the requirements using the model, we recommend gathering feedback to refine and extend them.

Reliable processing of quality requirements demands a proper quality model to be used as a reference, especially in the context of software package selection. Although building a quality model is complex,[10,11] our methodology shows many advantages over ad hoc package evaluation:

- *Confidence*. Uniform, vendor-independent descriptions of numerous software packages facilitate package comparison. Also, rewriting quality requirements in terms of a model's concepts helps us discover ambiguities and incompleteness that, once solved, let us more easily compare requirements with package descriptions. Lastly, quality models obtained with our methodology can be expressed in a component description language,[12] making tool support for package selection feasible.
- *Productivity*. Consider the amount of repeated work that takes place in the mail server domain. Many organizations have faced exactly the same problems and have repeated the same processes, wasting human resources and money. The existence of a quality model of reference for this domain simplifies mail server procurement, once an organization's quality requirements have been expressed in terms of the reference model.
- *Experience and reusability*. Repeatedly using the same methodology and quality standard increases our model-building skills. Also, reusing parts of existing models in new domains becomes feasible, both for high-level subcharacteristics and for low-level attributes. We have confirmed this during the different experiences we have had in the domains of mail servers, ERP systems, e-learning tools, some component libraries, and so on.

Our work is compliant with the 9126-1 standard; we will aim at 9126-2 when that version is final. Also, our proposal can be used to support the evaluation and acquisition process defined in the 14598 standard, namely in the steps "Establish evaluation requirements" and "Specify the evaluation."

Acknowledgments

This work is partially supported by the Spanish Ministry of Science and Technology under contract TIC2001-2165. Juan P. Carvallo's work has been supported by an Agencia Española de Cooperación Internacional (AECI) grant.

References

1. *Proc. 1st Int'l Conf. COTS-Based Software Systems* (ICCBSS), Lecture Notes in Computer Science, no. 2255, Springer-Verlag, Berlin, 2002.
2. A. Finkelstein, G. Spanoudakis, and M. Ryan, "Software Package Requirements and Procurement," *Proc. 8th IEEE Int'l Workshop Software Specification & Design* (IWSSD), IEEE CS Press, Los Alamitos, Calif., 1996, pp. 141–145.
3. J. Kontyo. "A Case Study in Applying a Systematic Method for COTS Selection," *Proc. 18th IEEE Int'l Conf. Software Eng.*, IEEE Computer Soc. Press, Los Alamitos, Calif., 1996, pp. 201–209.
4. N. Maiden and C. Ncube, "Acquiring Requirements for COTS Selection," *IEEE Software*, vol. 15, no. 2, Mar./Apr. 1998, pp. 46–56.
5. X. Burgués et al., "Combined Selection of COTS Components," *Proc. 1st Int'l Conf. COTS-Based Software Systems* (ICCBSS), Lecture Notes in Computer Science no. 2255, Springer-Verlag, Berlin, 2002, pp. 54–64.
6. *ISO/IEC Standard 9126-1 Software Engineering—Product Quality—Part 1: Quality Model*, ISO Copyright Office, Geneva, June 2001.
7. J. Bøegh et al., "A Method for Software Quality Planning, Control, and Evaluation," *IEEE Software*, vol. 23, no. 2, Mar./Apr. 1999, pp. 69–77.
8. L. Chung et al., *Non-functional Requirements in Software Engineering*, Kluwer Academic Publishers, Dordrecht, Netherlands, 2000.
9. N.E. Fenton and S.L. Pfleeger, *Software Metrics: A Rigorous and Practical Approach*, 2nd ed., Int'l Thomson Computer Press, London, 1998.
10. R.G. Dromey, "Cornering the Chimera," *IEEE Software*, vol. 20, no. 1, Jan./Feb. 1996, pp. 33–43.
11. B. Kitchenham and S.L. Pfleeger. "Software Quality: The Elusive Target," *IEEE Software*, vol. 20, no. 1, Jan./Feb. 1996, pp. 12–21.
12. X. Franch. "Systematic Formulation of Non-functional Characteristics of Software," *Proc. 3rd IEEE Int'l Conf. Requirements Eng.* (ICRE), IEEE CS Press, Los Alamitos, Calif., 1998, pp. 174–181.

About the Authors

Xavier Franch is an associate professor in the Software Department at the Universitat Politècnica de Catalunya in Barcelona, Spain. His interests are COTS component selection and evaluation, software quality, software process modeling and OO component libraries. He received his BSc and PhD in Informatics from the UPC. Contact him at Universitat Politècnica de Catalunya, c/ Jordi Girona 1-3 (Campus Nord, C6), E-08034 Barcelona, Spain; franch@lsi.upc.es; www.lsi.upc.es/~gessi.

Juan Pablo Carvallo is a doctoral candidate at the Universitat Politècnica de Catalunya. His interests are COTS-based systems development and COTS selection, evaluation, and certification. He received his degree in computer science from the Universidad de Cuenca, Ecuador. Contact him at Universitat Politècnica de Catalunya, c/ Jordi Girona 1-3 (Campus Nord, C6), E-08034 Barcelona, Spain; carvallo@lsi.upc.es.

Exercice de modélisation des cas d’utilisation

Développez un modèle (diagramme) exhaustif de cas d’utilisation et fournissez la description détaillée (flux principal, flux alternatif, etc.) de chacun des cas d’utilisation du modèle pour un système informatique qui gère une pompe à essence automatisée telle que décrite ci-dessous dans une station-service. Au besoin, postulez clairement les hypothèses à la base de vos cas d’utilisation.

- Avant de pouvoir être utilisée par un client, la pompe doit être activée par le commis de la station-service. Pour ce faire, la pompe ne doit pas être déjà en fonction et le paiement de la transaction précédente doit être complété. Si toutes les conditions sont remplies, la pompe s’initialise, les compteurs (montant en dollars du plein et la quantité en litres) affichés sur son écran se remettent à zéro.

- Pour faire le plein, le client doit choisir le type d’essence désiré et retirer le pistolet de son socle ce qui actionne en même temps un levier qui met le moteur de la pompe en fonction. À ce moment, le client peut appuyer sur la gâchette du pistolet pour pomper l’essence dans le véhicule. La distribution de l’essence se termine lorsque le client remet le pistolet sur son socle et le moteur de la pompe est désactivé.

- Le client a le choix entre trois types d’essence (Ordinaire avec un indice d’octane minimal de 87, Extra avec un indice d’octane minimal de 89 et Suprême avec un indice d’octane de 91) et doit appuyer sur le bouton correspondant au type d’essence désiré. Le type d’essence doit être choisi avant que le moteur de la pompe puisse être démarré.

- La mesure de la quantité d’essence pompée dans le véhicule se fait à l’aide d’un débitmètre électromagnétique qui fait la pondération automatique du volume d’essence en fonction de la température ambiante et de la température de l’essence dans le réservoir sous-terrain.

- Le niveau d’essence dans un des réservoirs sous-terrain ne doit pas être plus bas que 5% de la capacité maximale pour qu’une pompe puisse être activée.

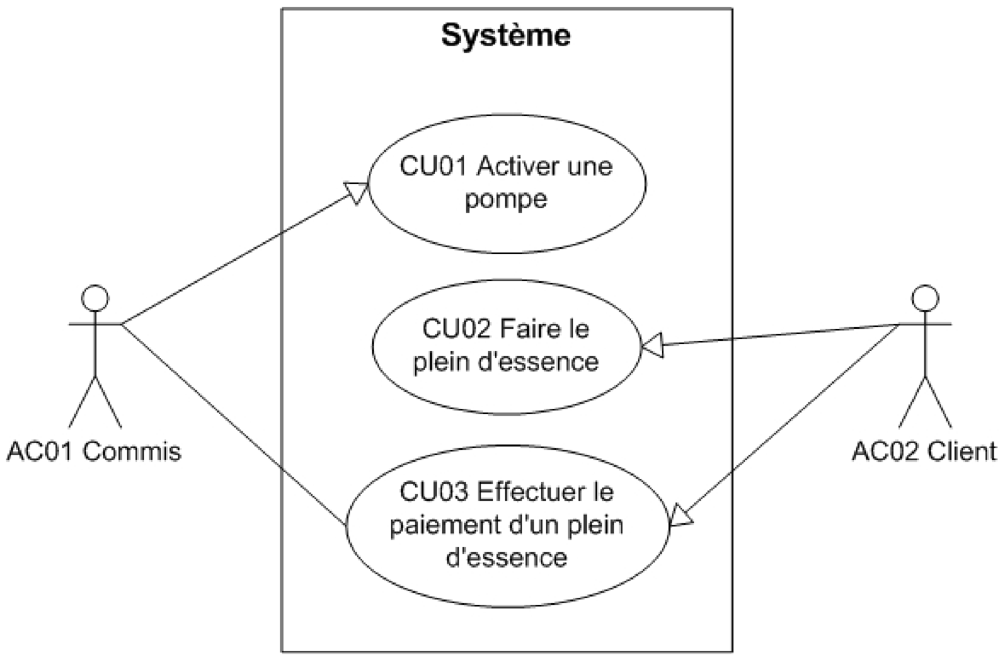
- Le paiement de l’essence doit se faire auprès du commis de la station-service en argent comptant ou par carte de crédit (bande magnétique seulement et sans puce électronique). De plus, si le client est membre d’un club automobile, il peut obtenir un rabais de 0,02$ par litre d’essence. À la fin de la transaction, le système imprime un reçu qui contient l’heure et la date de la transaction, le nombre de litres du plein d’essence et le montant détaillé comprenant les taxes. Toutes les transactions sont sauvegardées dans la base de données du système.

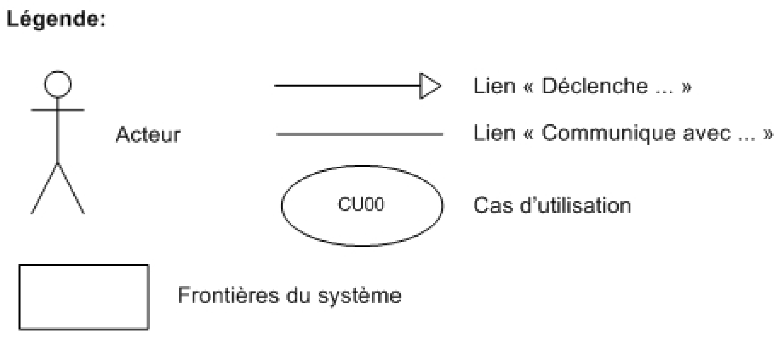
- Lorsque le client utilise une carte de crédit, l’authentification du paiement est réalisée en glissant la bande magnétique de la carte dans le terminal du système et indique au commis si le paiement est accepté ou non afin qu’il puisse valider la transaction. Si le paiement est accepté, le client doit signer un reçu émis spécialement pour le paiement par carte de crédit. Si le paiement est refusé, le client doit absolument payer en argent comptant.

Créé par David Guilmaine, 9 novembre 2011

Solution préliminaire de l'exercice

Modèle (diagramme) de cas d'utilisation


Système
CU01 Activer une pompe
CU02 Faire le plein d'essence
CU03 Effectuer le paiement d'un plein d'essence
AC01 Commis
AC02 Client



CU01 Activer une pompe

Acteurs

- AC01 : Le commis

Pré-conditions

- La pompe ne doit pas déjà être activée.
- Le paiement de la dernière transaction doit être complété.

Post-conditions

- Aucune

Scénario principal

1. Le commis désire activer une pompe.
2. Le commis active la pompe.
3. Le système initialise la pompe et les compteurs du montant et de la quantité d’essence se remettent à zéro.

Flux alternatifs

2a. La quantité d’essence dans le réservoir sous-terrain est inférieure à 5% de la capacité maximale.

1. La pompe ne peut pas être activée.
2. Fin du cas d’utilisation.

CU02 Faire le plein d’essence

Acteurs

- AC02 : Le client

Pré-conditions

- La pompe doit être activée par le commis selon le CU01.

Post-conditions

- Le client a fait le plein d’essence de son véhicule.
- Le nombre de litres du plein et le montant à payer sont sauvegardés.

Scénario principal

1. Le client désire faire le plein d’essence.
2. Le client choisit le type d’essence qu’il désire et décroche le pistolet de son socle.
3. Le système active le moteur de la pompe.
4. Le client insère le pistolet dans l’embouchure du réservoir du véhicule et démarre le moteur de la pompe en appuyant sur la gâchette.
5. Le système calcule le nombre de litres du plein d’essence et met à jour le compteur du nombre de litres d’essence et le montant en dollars.
6. Le client arrête le moteur de la pompe en relâchant la gâchette et replace le pistolet sur le socle.
7. Le système désactive le moteur de la pompe.
8. Le système sauvegarde le nombre de litres du plein et le montant à payer.

Flux alternatifs

2a. Le client décroche le pistolet de son socle avant de sélectionner le type d’essence.

1. Le système n’active pas le moteur de la pompe et attend la sélection du type d’essence.
2. Le client choisit le type d’essence qu’il désire.
3. Retour à l’étape 3 du scénario principal.

CU03 Effectuer le paiement d’un plein d’essence

Acteurs

- AC01 : Le commis
- AC02 : Le client

Pré-conditions

- Le client a fait le plein de son véhicule selon le CU02.

Post-conditions

- Le paiement du plein d’essence est effectué et la transaction est enregistrée dans la base de données.

Scénario principal

1. Le client désire faire le paiement d’un plein d’essence.
2. Le commis demande au système la quantité en litres d’essence et le montant du plein de la pompe utilisée.
3. Le système retourne la quantité en litres d’essence et le montant à payer.
4. Le client indique au commis qu’il veut payer avec de l’argent comptant et le remet au commis.
5. Le commis entre dans le système le montant en argent donné par le client et lui remet le change (s’il y a lieu).
6. Le système sauvegarde la transaction et imprime un reçu contenant l’heure et la date de la transaction, le nombre de litres du plein d’essence et le montant détaillé comprenant les taxes.

Flux alternatifs

3a. Le client est membre d’un club automobile et a droit à une remise.

1. Le commis indique au système que le client est membre d’un club automobile.
2. Le système calcule une remise de 0,02$ par litre d’essence du plein.
3. Le système soustrait ce montant au montant total du plein.
4. Retour à l’étape 3.

4a. Le client indique au commis qu’il veut payer avec une carte de crédit.

1. Le commis indique au système que le client veut payer à l’aide d’une carte de crédit et glisse la carte dans le terminal du système.
2. Le système lance la procédure d’autorisation du paiement.
   - 2a. Le paiement est accepté par le système et il imprime un reçu.
     - 2a1. Le client signe le reçu émis par le système et le remet au commis.
     - 2a2. Le commis valide la signature et termine la transaction.
     - 2a3 Retour à l’étape 6 du scénario principal.
   - 2b. Le paiement est refusé.
     - 2b1. Le commis informe le client du refus du paiement par carte de crédit.
     - 2b2. Retour à l’étape 4 du scénario principal.