Understanding Software Requirements

Stuart R. Faulk

"The hardest single part of building a software system is deciding precisely what to build. No other part of the conceptual work is as difficult as establishing the detailed technical requirements...No other part of the work so cripples the resulting system if done wrong. No other part is as difficult to rectify later." [Brooks 87]

1. Introduction

Deciding precisely what to build and documenting the results is the goal of the requirements phase of software development. For many developers of large, complex software systems, requirements are their biggest software engineering problem. While there is considerable disagreement on how to solve the problem, few would disagree with Brooks' assessment that no other part of a development is as difficult to do well or as disastrous in result when done poorly. The purpose of this tutorial is to help the reader understand why the apparently simple notion of "deciding what to build" is so difficult in practice, where the state of the art does and does not address these difficulties, and what hope we have for doing better in the future.

This paper does not survey the literature but seeks to provide the reader with an understanding of the underlying issues. There are currently many more approaches to requirements than one can cover in a short paper. This diversity is the product of different views about which of the many problems in requirements are pivotal and of different assumptions about the desirable characteristics of a solution. This paper attempts to impart a basic understanding of the many facets of the requirements problem and the tradeoffs involved in attempting a solution. Thus forearmed, the reader may make his own assessment of the claims of different requirements methods and their likely effectiveness in addressing his particular needs.

We begin with basic terminology and some historical data on the requirements problem. We examine the goals of the requirements phase and the problems that can arise in attempting to meet those goals. As in Brooks' article [Brooks 87], much of the discussion is motivated by the distinction between the difficulties inherent in what one is trying to accomplish (the "essential" difficulties) and those one creates through inadequate practice ("accidental" difficulties). We discuss how a disciplined software engineering process helps address many of the accidental difficulties and why the focus of such a disciplined process is on producing a written specification of the detailed technical requirements. We examine current technical approaches to requirements in terms of the specific problems each approach seeks to address. Finally, we examine technical trends and discuss where significant advances are likely to occur in the future.

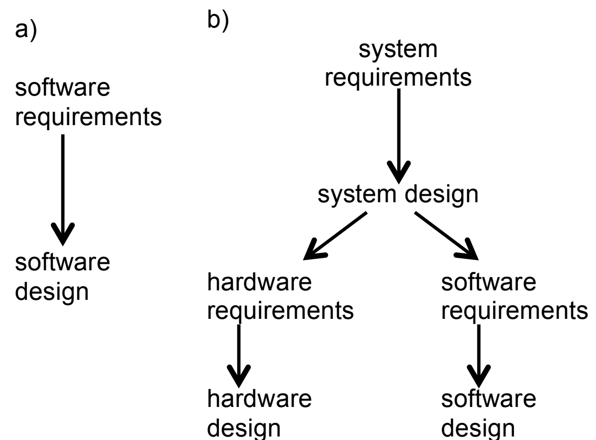

Figure 1: System vs. software requirements

2. Requirements and the Software Life Cycle

A variety of software life–cycle models have been proposed with an equal variety of terminology. While differing in the detailed decomposition of the steps (e.g., prototyping models) or in the surrounding management and control structure (e.g., to manage risk), there is general agreement on the core elements of the model. Figure 1 is a version of the model that illustrates the relationship between the software development stages and the related testing and acceptance phases

When software is created in the context of a larger hardware and software system, system requirements are defined first followed by system design. System design includes decisions about which parts of the system requirements will be allocated to hardware and which to software. For software–only systems, the life cycle model begins with analysis of the software requirements. From this point on, the role of software requirements in the development model is the same whether or not the software is part of a larger system, as shown in

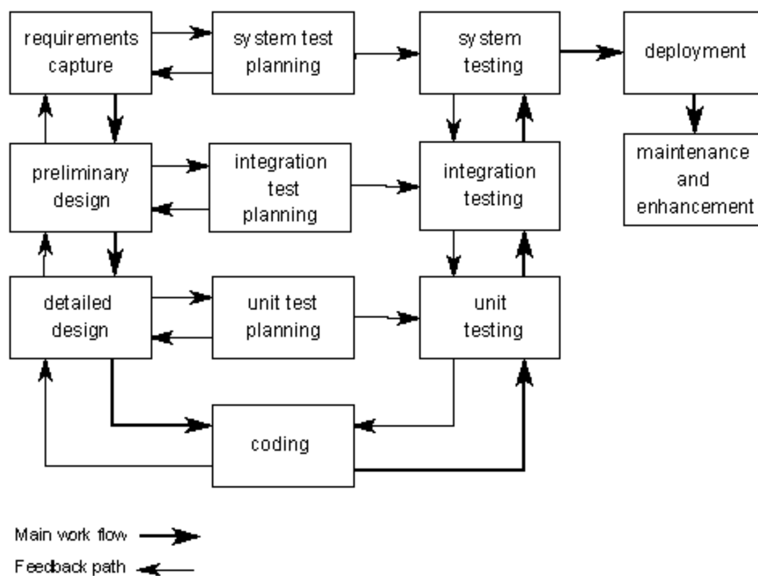

Figure 2: A conventional life-cycle model

Figure 2. For this reason, the remainder of our discussion does not distinguish whether or not software is developed as part of a larger system. For an overview of system versus software issues, the reader is referred to Dorfman and Thayer's survey [Thayer 90].

In a large system development, the software requirements specification may play a variety of roles:

- For customers, the requirements typically document what should be delivered and may provide the contractual basis for the development.
- For managers it may provide the basis for scheduling and a yardstick for measuring progress.
- For the software designers, it may provide the "design–to" specification.
- For coders it defines the range of acceptable implementations and is the final authority on the outputs that must be produced.
- For quality assurance personnel, it is the basis for validation, test planning and verification.

Such diverse groups as marketing and governmental regulators may also use the requirements. These, and any others with a interest in the outcome of system development are collectively referred to as the system's *stakeholders*.

It is common practice (e.g., see [Thayer 90]) to classify software requirements as "functional" or "non-functional." While definitions vary somewhat in detail, "functional" typically refers to requirements defining the acceptable mappings between system input values and corresponding output values. "Non-functional" then refers to all other constraints including, but not limited to, performance, dependability, maintainability, reusability, and safety.

While widely used, the classification of requirements as ”functional” and ”non-functional” is confusing in its terminology and of little help in understanding common properties of different kinds of requirements. The word "function" is one of the most overloaded in computer science and its only rigorous meaning, that of a mathematical function, is not what is meant here. The classification of requirements as functional and non–functional offers little help in understanding common attributes of different types of requirements since it partitions classes of requirements with markedly similar qualities (e.g., output values and output deadlines) while grouping others that have common only what they are not (e.g., output deadlines and maintainability goals).

A more useful distinction is between what can be described as "behavioral requirements" and "developmental quality attributes" with the following definitions [Bass 03]:

- *Behavioral requirements* - Behavioral requirements include any and all information necessary to determine if the run–time behavior of a given implementation is acceptable. The behavioral requirements define all constraints on the system outputs (e.g., value, accuracy, timing) and resulting system state for all possible inputs and current system state. By this definition, se-

curity, safety, performance, timing, and fault–tolerance are all behavioral requirements.

- *Developmental quality attributes* - Developmental quality attributes include any constraints on the attributes of the system's static construction. These include properties like testability, changeability, maintainability, and reusability.

Behavioral requirements have in common that they are properties of the run–time behavior of the system and can (at least in principle) be validated objectively by observing the behavior of the running system, independent of its method of implementation. In contrast, developmental quality attributes are properties of the system's static structures (e.g., modularization) or representation. Developmental quality attributes have in common that they are functions of the development process and methods of construction. Assessment of developmental quality attributes is necessarily relativistic - for example, we do not say that a design is or is not maintainable but that one design is more maintainable than another.

In addition, there may be constraints on the development process itself. For example, that the software must reuse certain legacy code, be developed on a particular platform, or be written in a specific language. Such requirements may be collectively referred to as *process requirements* [SWEBOK 04]. Process requirements are often imposed by regulatory agencies or internal company standards.

3. A Big Problem

Requirements problems are persistent, pervasive, and costly. Evidence is most readily available for the large software systems developed for the U.S. Government since the results are a matter of public record. As soon as software became a significant part of such systems, developers identified requirements as a major source of problems. For example, developers of the early Ballistic Missile Defense System noted that:

> *In nearly every software project that fails to meet performance and cost goals, requirements inadequacies play a major and expensive role in project failure* [Alford 79].

Nor has the problem been mitigated over the intervening years. A study of problems in mission critical defense systems identified requirements as a major problem source in two thirds of the systems examined [GAO 92]. This is consistent with results of a survey of large aerospace firms that identified requirements as the most critical software development problem [Faulk 92]. Likewise, studies by Lutz [Lutz 92] identified functional and interface requirements as the major source of safety-related software errors in NASA's Voyager and Galileo spacecraft. The GAO again identified requirements as a major issue in defense acquisition [GAO 04]. Requirements errors have also been cited as a major cause in the very public losses of the Mars Climate Orbiter and Mars Polar Lander spacecraft [Bahill 05].

Results of industry studies described by Boehm [Boehm 81], and since replicated a number of times, showed that requirements errors are the most costly. These studies all produced the same basic result: the earlier in the development process an error occurs and the later the error is detected, the more expensive it is to correct. Moreover, the relative cost rises quickly. As shown in Figure 3, an error that costs a dollar to fix in the requirements phase may cost 100 to 200 dollars to fix if it is not corrected until the system is fielded or in the maintenance phase.

Stage	Relative Repair Cost
Requirements	1-2
Design	˜ 5
Coding	˜ 10
Unit test	˜ 20
System test	˜ 50
Maintenance	˜ 200

Figure 3: Relative cost to repair a requirements error

The costs of such failures can be enormous. For example, a 1992 GAO report noted that one system, the Cheyenne Mountain Upgrade, would be delivered eight years late, exceed budget by $600 million, and had less capability than originally planned, largely due to requirements-related problems. Recently, requirements problems have been cited in cost overruns projected for the 2010 Census of up to $2 billion [GAO 08]. Broader GAO reviews (e.g. of troubled weapons programs [GAO 10]) suggest that such problems are the norm rather than the exception. While data from

private industry is less readily available, there is little reason to believe that the situation is significantly different.

In spite of advances in software engineering methodology and tool support, the requirements problem has not diminished. This does not mean that the apparent progress in software engineering is illusory. While the features of the problem have not changed, the applications have grown significantly in capability, scale, and complexity. A reasonable conclusion is that the growing ambitiousness of our software systems has outpaced the gains in requirements technology; at least as such technology is applied in practice.

4. Why are Requirements Hard?

It is generally agreed that the goal of the requirements phase is to establish and specify precisely what the software must do without describing how to do it. So simple seems this basic intent that it is not at all evident why it is so difficult to accomplish in practice. If what we want to accomplish is so clear, why is it so hard? To understand this, we must examine more closely the goals of the requirements phase, where errors originate, and why the nature of the task leads to some inherent difficulties.

Most authors agree in principle that requirements should specify “what” rather than “how.” In other words, the goal of requirements is to understand and specify the *problem* to be solved rather than the *solution*. For example, the requirements for an automated teller system should talk about customer accounts, deposits, and withdrawals rather than the software algorithms and data structures. The most basic reason for this is that a specification in terms of the problem captures the actual requirements without over-constraining the subsequent design or implementation. Further, solutions in software terms are typically more complex, more difficult to change, and harder to understand (particularly for the customer) than a specification of the problem.

Unfortunately, distinguishing “what” from “how” itself represents a dilemma. As Davis [Davis 88], among others, points out, the distinction between what and how is necessarily a function of perspective. A specification at any chosen level of system decomposition can be viewed as describing the “what” for the next level. Thus customer needs may define the “what” and the decomposition into hardware and software the corresponding “how.” Subsequently, the behavioral requirements allocated to a software components define its “what,” the software design, the “how, and so on. In other words, one person’s design becomes the next person’s requirements.

The upshot is that requirements cannot be effectively discussed at all without prior agreement on which system one is talking about and at what level of decomposition. One must agree on what constitutes the *problem space* and what constitutes the *solution space* - the analysis and specification of requirements then properly belongs in the problem space.

In discussing requirements problems one must also distinguish the development of large, complex systems from smaller efforts (e.g., developments by a single or small team of programmers). Large system developments are multi–person efforts. They are developed by teams of tens to thousands of programmers. The programmers work in the context of an organization typically including management, systems engineering, marketing, accounting, and quality assurance. The organization itself must operate in the context of outside concerns also interested in the software product, including the customer, regulatory agencies, and suppliers.

Even where only one system is intended, large systems are inevitably multi–version as well. As the software is being developed, tested, and even fielded, it evolves. Customers understand better what they want, developers understand better what they can and cannot do within the constraints of cost and schedule, and circumstances surrounding development change. The results are changes in the software requirements and, ultimately, the software itself. In effect, several versions of a given program are produced, if only incrementally. Such unplanned changes occur in addition to the expected variations of planned improvements.

The multi-person, multi-version nature of large system development introduces problems that are both quantitatively and qualitatively different from those found in smaller developments. For example, scale introduces the need for administration and control functions with the attendant management issues that do not exist on small projects. The quantitative effects of increased complexity in communication when the number of workers

rises are well documented by Brooks [Brooks 95]. The effort required for communication and other overhead tasks such as documentation or configuration management tend to rise exponentially with the size and complexity of the system. In the following discussion, it is this large system development context we will assume since that is the one in which the worst problems occur and where the most help is needed.

Given the context of multi–person, multi–version development, our basic goal of specifying what the software must do can be decomposed into the following subgoals:

1. Understand precisely what is required of the software.
2. Communicate the understanding of what is required to all of the parties involved in the development.
3. Control the software production to ensure that the final system satisfies the requirements (including managing the effects of changes).

It follows that the source of most requirements errors is in the failure to adequately accomplish one of these goals, i.e.:

1. The developers failed to understand what was required of the software by the customer, end user, or other parties with a stake in the final product.
2. The developers did not adequately capture the requirements or subsequently communicate the requirements effectively to other parties involved in the development.
3. The developers did not effectively manage the effects of changing requirements or ensure the conformance of down–stream development steps including design, code, integration, test, or maintenance to the system requirements.

The end result of such failures is a software system that does not perform as desired or expected, a development that exceeds budget and schedule, or, all too frequently, failure to deliver any working software at all.

4.1 Essential Difficulties

Even our more detailed goals appear reasonably straightforward; why then do so many development efforts fail to achieve them? The short answer is that the *mutual* satisfaction of these goals, in practice, is inherently difficult. To understand why, it is useful to reflect on some points raised by Brooks [Brooks 87] on why software engineering is hard and on the distinction he makes between essential difficulties - those inherent in the problem, and the accidental difficulties - those introduced through imperfect practice. For though requirements are inherently difficult, there is no doubt that these difficulties are many times multiplied by the inadequacies of current practice.

The following essential difficulties attend each (in some cases all) of the requirements goals:

- *Comprehension*. People do not know what they want. This does not mean that people do not have a general idea of what the software is for. Rather, they do not begin with a precise and detailed understanding of what functions belong in the software, what the output must be for every possible input, how long each operation should take, how one decision will affect another, and so on. Indeed, unless the new system is simply a reconstruction of an old one, such a detailed understanding at the outset is unachievable. Many decisions about the system behavior will depend on other decisions yet unmade, and expectations will change as the problem (and attendant costs of alternative solutions) is better understood. Nonetheless, it is a precise and richly detailed understanding of expected behavior that is needed to create effective designs and develop correct code.

- *Communication*. Software requirements are difficult to communicate effectively. As Brooks points out, the conceptual structures of software systems are complex, arbitrary, and difficult to visualize. The large software systems we are now building are among the most complex structures ever attempted. That complexity is arbitrary in the sense that it is an artifact of people's decisions and prior construction rather than a reflection of fundamental properties (as, for example, in the case of physical laws). To make matters worse, many of the conceptual structures in software have no readily comprehensible physical analogue so they are difficult to visualize.

In practice, comprehension suffers under all of these constraints. We work best with regular, predictable structures, can comprehend only a very limited amount of information at one time, and understand large amounts of information best when we can visualize it. Thus the task of capturing and conveying software requirements is inherently difficult.

The inherent difficulty of communication is compounded by the diversity of purposes and audiences for a requirements specification. Ideally a technical specification is written for a particular audience. The brevity and comprehensibility of the document depend on assumptions about common technical background and use of language. Such commonality typically does not hold for the many diverse groups (e.g., customers, systems engineers, managers) that must use a software requirements specification.

- *Control*. Inherent difficulties attend control of software development as well. The arbitrary and invisible nature of software makes it difficult to anticipate which requirements will be met easily and which will decimate the project's budget and schedule if, indeed, they can be fulfilled at all. The low fidelity of software planning has become a cliché yet the requirements are often the best available basis for planning or for tracking to a plan.

  This situation is made incalculably worse by software's inherent malleability. Of all the problems bedeviling software mangers, few evoke such passion as the difficulties of dealing with arbitrary requirements changes. For most systems, such changes remain a fact of life even after delivery. The continuous changes make it difficult to develop stable specifications, plan effectively, or control cost and schedule. For many industrial developers, change management is the most critical problem in requirements.

- *Inseparable concerns*. In seeking solutions to the foregoing problems, we are faced with the additional difficulty that the issues cannot easily be separated and dealt with piecemeal. For example, developers have attempted to address the problem of changing requirements by baselining and freezing requirements before design begins. This proves impractical because of the comprehension problem - the customer may not fully know what he wants until he sees it. Similarly, the diversity of purposes and audiences is often addressed by writing a different specification for each. Thus there may be a system specification, a set of requirements delivered to customer, a distinct set of technical requirements written for the internal consumption of the software developers, and so on. However, this solution vastly increases the complexity, provides an open avenue for inconsistencies, and multiplies the difficulties of managing changes.

  These issues represent only a sample of the inherent dependencies between different facets of the requirements problem. The many distinct parties with an interest in a system's requirements, the many different roles the requirements play, and the interlocking nature of software's conceptual structures, all introduce dependencies between concerns and impose conflicting constraints on any potential solution.

  The implications are twofold. First we are constrained in the application of our most effective strategy for dealing with complex problems - divide and conquer. If a problem is considered in isolation, the solution is likely to aggravate other difficulties. Effective solutions to most requirements difficulties must simultaneously address more than one problem. Second, developing practical solutions requires making difficult tradeoffs. Where different problems have conflicting constraints, compromises must be made. Because the tradeoffs result in different gains or losses to the different parties involved, effective compromise requires negotiation. These issues are considered in more detail when we discuss the properties of a good requirements specification.

4.2 Accidental Difficulties

While there is no doubt that software requirements are inherently difficult to do well, there is equally no doubt that common practice unnecessarily exacerbates the difficulty. We use the term "accidental" in contrast to "essential," not to imply that the difficulties arise by chance, but that they are the product of common failings in management, elicitation, specification, or use of requirements. It is these failings that are most easily addressed by improved practice.

- *Written as an afterthought*. It remains common practice that requirements documentation is developed only after the software has been written. For many projects, the temptation to rush into implementation before the requirements are adequately understood proves irresistible. This is understandable. Developers often feel like they are not really doing anything when they are not writing code; managers are concerned about schedule when there is no visible progress on the implementation. Then too, the intangible nature of the product mitigates toward early implementation. Developing the system is an obvious way to understand better what is needed and make visible the actual behavior of the product. The result is that requirements specifications are written as an afterthought (if at all). They are not created to guide the developers and testers but treated as a necessary evil to satisfy contractual demands.

  Such after–the–fact documentation inevitably violates the principle of defining what the system must do rather than the how since it is a specification of the code as written. It is produced after the fact so it is not planned or managed as an essential part of the development but is thrown together. In fact, it is not even available in time to guide implementation or manage production.

- *Confused in purpose*. Because there are so many potential audiences for a requirements specification, with different points of view, the exact purpose of the document becomes confused. An early version is used to sell the product to the customer so it includes marketing hype extolling the product's virtues. It is the only documentation of what the system does so it provides introductory, explanatory, and overview material. It is a contractual document so it is intentionally imprecise to allow the developer latitude in the delivered product or the customer latitude in making no–cost changes. It is the vehicle for communicating decisions about software to designers and coders so it incorporates design and implementation details. The result is a document in which it is unclear which statements represent real requirements and which are more properly allocated to marketing, design, or other documentation. It is a document that attempts to be everything to everyone and ultimately serves no one well.

- *Not designed to be useful*. Often in the rush to implementation little effort is expended on requirements. The requirements specification is not expected to be useful and, indeed, this turns out to be a self–fulfilling prophecy. Little effort is expended on designing it, writing it, checking it, or managing its creation and evolution. The most obvious result is poor organization. The specification is written in English prose and follows the author's stream of consciousness or the order of execution [Heninger 80].

  The resulting document is ineffective as a technical reference. It is unclear which statements represent actual requirements. It is unclear where to put or find particular requirements. There is no effective procedure for ensuring that the specification is consistent or complete. There is no systematic way to manage requirements changes. The specification is difficult to use and difficult to maintain. It quickly becomes out of date and loses whatever usefulness it might originally have had.

- *Lacks essential properties*. Lack of forethought, confusion of purpose, or lack of careful design and execution all lead to requirements that lack properties critical to good technical specifications. The requirements, if documented at all, are re-

dundant, inconsistent, incomplete, imprecise, and inaccurate.

Where the essential difficulties are inherent in the problem, the accidental difficulties result from a failure to gain or maintain intellectual control over what is to be built. While the presence of the essential difficulties means that there can be no “silver bullet” that will suddenly render requirements easy, we can remove at least the accidental difficulties through a well though out, systematic, and disciplined development process. Such a disciplined process then provides a stable foundation for attacking the essential difficulties.

5. Role of a Disciplined Approach

The application of discipline in analyzing and specifying software requirements can address the accidental difficulties. While there is considerable agreement on the desirable qualities of a software development approach, development processes have not been standardized. Further, the context and qualities of developments can differ such that no single process model will suit all developments. Nonetheless, it is useful to examine the characteristics of an idealized process and its products to understand where current approaches are weak and which current trends are promising. In general, a complete requirements approach will define:

- *Process:* The (partially ordered) sequence of activities, entrance and exit criteria for each activity, which work products are produced in each activity, and what skill sets are needed do the work.
- *Products:* The work products to be produced and, for each product, the resources needed to produce it, the information it contains, the expected audience, and the acceptance criteria the product must satisfy.

Conceptually, the requirements phase consists of two distinct but overlapping activities corresponding to the first two goals for requirements enumerated previously:

1. *Problem analysis:* The goal of problem analysis is to understand precisely what problem is to be solved. It includes identifying the system’s stakeholders and eliciting their requirements. It also includes deciding the exact purpose of the system, who will use it, the constraints on acceptable solutions, and the possible tradeoffs between conflicting constraints.
2. *Requirements specification:* The goal of requirements specification is to capture the results of problem analysis in a transferable form. The products of this activity typically include a written specification of precisely what is to be built in the form of a Software Requirements Specification (SRS). The SRS captures the decisions made during problem analysis and characterizes the set of acceptable solutions to the problem.

In practice, the distinction between these activities is conceptual rather than temporal. Where both are needed, the developer typically switches back and forth between analysis of the problem and documentation of the results. When problems are well understood, the analysis phase may be virtually non–existent. When the system model and documentation are standardized or based on existing specifications, the documentation paradigm may guide the analysis [Hester 81].

5.1 Problem Analysis

Problem analysis lies at the boundary between human concerns and the realization of some software system that seeks to address those concerns. It is necessarily informal in the sense that there is no effective, closed end procedure that will guarantee success. It is an information acquiring, collating, and structuring process through which one attempts to understand all the various parts of a problem and their relationships.

Problem analysis may be further divided into two closely related sub-activities: *requirements elicitation* and *requirements modeling and analysis*. Requirements elicitation focuses on the human side of problem analysis. It seeks to answer the question “What are the behavioral and developmental qualities of an acceptable system?” Modeling and analysis supports elicitation by capturing the answers to this question in a form that allows the stakeholders to understand, communicate, and reason about the results.

Requirements Elicitation

As our discussion of the essential difficulties suggests, understanding what constitutes an “acceptable system” to its stakeholders can be a daunting

task. People do not really know what they want in sufficient detail. Moreover, different people or types of stakeholders often have different and incompatible views of the problem, the purposes for developing the system, and what it should accomplish. In fact, since the scope of the system may be undetermined, it may not even be clear who the stakeholders are.

The purpose of a disciplined elicitation process is to systematically remove the uncertainty from problem understanding, resolve conflicting views, and arrive at a set of behavioral and developmental requirements that the stakeholders will agree to. To do so, the process must answer the following questions:

- What are the system boundaries?
- What is the rationale for creating the system? What are the current problems and what are the goals for the proposed system?
- What are the constraints on acceptable solutions?
- Who are the stakeholders?
- What are the different stakeholders' views of the problem and the system requirements?
- Where does the understanding differ or requirements conflict and how can those conflicts be resolved?

Developments differ in the extent to which the process must address such questions. For example, where there is a single customer, it may be unnecessary to expend any effort establishing who the stakeholders are or managing stakeholder conflicts. Thus, the activities necessary to answering these questions are incorporated into the elicitation process as needed.

Establish system boundaries: The purpose of this activity is to establish where system concerns properly begin and end. In practice, this means characterizing the system's external interfaces. It delimits and defines how the software interacts with users or with other systems (software or hardware).

In addition, establishing the system boundaries sets boundaries on the elicitation process itself. By defining what is inside the system and what is outside, it bounds the scope of inquiry about the problem and the system requirements. By identifying which concerns properly belong to the software it helps establish who the stakeholders are and which views or concerns are relevant. By establishing bounds on which persons and issues are relevant, it helps determine when elicitation is done.

Rationale and goal understanding: Fully understanding the problem requires understanding the rationale - why the system is being built in the first place. Understanding the rationale can be necessary for establishing system requirements and for maintaining consistency as real-world objectives or constraints change over time.

The rationale encompasses both the problems with any current system (automated or manual) and the objectives for the new system. System objectives may be codified in the form of goals where a *goal* characterizes "an objective the system under consideration should achieve" [Lamsweerde 01].

Goals provide a link between broader concerns like business objectives and the requirements that instantiate those concerns in the software context. Defining goals and providing traceability to the software requirements supports managing requirements changes as business objectives mature. Likewise, understanding the overall system goals and their relative priorities provides a basis for choosing among likely alternatives and resolving conflicting requirements. Specific approaches to goal-based requirements are discussed in the subsequent section on the state of practice.

Stakeholder identification: fully understanding the problem necessitates identifying all of the system's stakeholders, then understanding their interest in the system. In stakeholder identification, it is important to include both the individuals (or organizations) who stand to lose, as well as those who stand to gain, from development success or failure [Gause 89].

For many large developments it is not immediately obvious who all the stakeholders are, even to the stakeholders themselves. Further, the set of stakeholders may change as requirements evolve, system boundaries change, or the individual filling those organizational roles are replaced.

Since different stakeholders will have different attributes, concerns, and views of the system, identifying them is a necessary step toward selecting appropriate elicitation methods, gathering a

complete set of requirements, establishing priorities, and negotiating conflicts.

Elicitation: The core of requirements elicitation is the process of working with the stakeholders to obtain their understanding of the problem, goals, and system requirements. Since different classes of stakeholders typically have different perspectives on the problem, have different cultures, and communicate in different languages, a number of different elicitation methods may have to be used as part of an effective elicitation process. Determining which methods to use, incorporating them in the requirements process, and synthesizing the results are the concerns of effective practice (e.g, [Lauesen 02]).

Requirements Negotiation: Different stakeholders necessarily have different perspectives on the system requirements. For most real developments, there is no single set of requirements waiting to be discovered. Rather, there are many potential manifestations of stakeholder desires that lead to different, and often conflicting, sets of requirements.

Before development can proceed to implementation, there must be agreement on a single, consistent set of requirements. Modeling and analyzing the requirements can help identify where conflicts occur but does not resolve them. This almost always requires tradeoffs and compromises between conflicting goals. It follows that arriving at agreement requires an effective process for negotiating requirements tradeoffs among stakeholders (e.g., [Boehm 94]).

Requirements Modeling and Analysis

The inherent difficulties of software complexity and invisibility are typically addressed by developing one or more abstract models. "Model," in this sense, means a representation of some aspect of the software system, the system's context, or both. It is abstract in that it represents certain information (entities and relationships) about the system while omitting others.

The use of models can help make the intangible objects and relationships in a software system visible. For example, a behavioral model might show the required system transitions and the observable behavior in response to user inputs. Such models aid elicitation and understanding by providing a transferable representation of the problem or system requirements. The use of models also reduces complexity by allowing the user to focus on and reason about a limited, related set of information at one time.

That said, not all models or modeling languages are equal. In some cases, "abstract" is interpreted to mean vague, not well defined, or inaccurate. To support reasoning about a system, any model should have the property that anything that is true of the model is also true of the system it represents. One can then manipulate the model to achieve particular developmental goals with the understanding that corresponding transformations to the system will yield corresponding real-world properties. In many cases, modeling languages (e.g., UML) lack sufficiently well defined semantics to achieve this property. The result is a model that is open to conflicting interpretations.

In addition to supporting problem understanding, the creation of models can support various kinds of analysis. Where models provide a formal syntax and semantics, they may support analysis for properties like consistency and completeness, as well as reasoning about requirements like safety properties. Such analyses can help identify missing requirements, inconsistencies, and requirements conflicts during elicitation. While informal models may not support formal reasoning, they can be useful aids for visualizing and reasoning about system requirements, as long as their limitations are understood.

5.2 Requirements Specification

For substantial developments, the effectiveness of the requirements effort depends on how well the SRS captures the results of analysis and how useable the specification is. There is little benefit to developing a thorough understanding of the problem if that understanding is not effectively communicated to customers, designers, implementers, testers, and other stakeholders. The larger and more complex the system, the more important a good specification becomes. This is a direct result of the many roles the SRS plays in a multi–person, multi–version development [Parnas 86]:

1. The SRS is the primary vehicle for agreement between the developer and customer on exactly what is to be built. It is the document reviewed by the customer or his representative and often is the basis for judging fulfillment of contractual obligations.

2. The SRS records the results of problem analysis. It is the basis for determining where the requirements are complete and where additional analysis is necessary. Documenting the results of analysis allows questions about the problem to be answered only once during development.

3. The SRS defines what properties the system must have and the constraints on its design and implementation. It defines where there is, and is not, design freedom. It helps ensure that requirements decisions are made explicitly during the requirements phase, not implicitly during design or programming.

4. The SRS is the basis for estimating cost and schedule. It is management's primary tool for tracking development progress and ascertaining what remains to be done.

5. The SRS is the basis for test plan development. It is the tester's chief tool for determining the acceptable behavior of the software.

6. The SRS provides the standard definition of expected behavior for the system's maintainers and is used to record engineering changes.

For a disciplined software development, the SRS is the primary technical specification of the software and the primary control document. This is an inevitable result of the complexity of large systems and the need to coordinate multi–person development teams. To ensure that the right system is built, one must first understand the problem. To ensure agreement on what is to be built and the criteria for success, the results of that understanding must be recorded. The goal of a systematic requirements process is thus the development of a set of specifications that effectively communicate the results of analysis. The SRS is the primary vehicle for communicating requirements between the developers, managers, and customers so the document is designed to be useful to that purpose. A useful document is maintained.

5.3 Requirements Process and Plan

Requirements' accidental difficulties are addressed through the careful analysis and specification of a disciplined process. Rather than developing the specification as an afterthought, requirements are understood and specified before development begins. One knows what one is building before attempting to build it. Where requirements cannot be completely known in advance the process systematically revisits the requirements process and downstream activities (e.g., iterative development).

The facts that requirements cannot be fully known in advance, and often change, are sometimes used as justification for expending little effort on requirements planning. The thought is that the project will deal with requirements when and if they become manifest. Such an approach surrenders the notion of a controlled engineering process to chance.

As a system goes to code, every decision about the requirements necessarily gets made (by definition). The question is not whether any particular requirements decision will be made but when it will be made and by whom. By default, any decision that is not made earlier in the process will be made by the programmers. In many cases, the programmers have little visibility into the business implications of such decisions or their effects on stakeholder goals. This is seldom a desirable outcome.

Being in control of the process means that requirements decisions, including postponing or not making decisions, are conscious choices. Each decision is made at the appropriate time by those with the knowledge and skills necessary to choose the best available alternative. This kind of control requires that the complex activities around requirements be planned in advance.

While organizations that develop complex software systems should employ a disciplined requirements process, no one process will meet the needs of every organization. A company that is developing an application where development cost and time to market are primary business drivers should not use the same process as an organization developing safety critical aerospace software with a long life expectancy.

It follows that the requirements process is something that should be chosen or designed to fit the organizational and even developmental contexts. While every development will typically go through some form of elicitation, modeling, analysis, and specification, the emphasis on the different phases and products will differ from one situation to the next. Likewise, the choices among

methods, technologies, notations, and tools will vary.

For deploying processes that are a good fit for a particular organization or situation, it is useful to think of processes as products. That is, we want a process that meets particular organizational and developmental goals (e.g., short time-to-market). To meet those goals, the process will need to satisfy certain requirements (e.g., contain certain milestones or satisfy particular standards). We then must create (build) or choose (buy) a process that satisfies the requirements. We must communicate that process to those who will enact it, manage it, or monitor it. We must validate the process against the goals, verify its enactment, and so on.

In a disciplined organization, this means that there must be a written specification that records decisions about the process and provides a baseline for enactment, tailoring, or process improvement. While treating a process as a product in this manner may seem alien, in fact many organizations that have embarked on systematic process improvement (e.g., [SEI 06]) have done all of this and more. Thinking about the process as a product helps ensure that adequate consideration is given to planning, budgeting for, and managing process development or improvement.

At the project level, the requirements process should be instantiated in the form of a *requirements plan* [Young 04]. The requirements plan makes the abstract requirements processes concrete by mapping activities to tasks, people to roles, and artifacts to deliverables. It describes who will do what using which specific methods and tools. For example, it should describe which elicitation methods will be used to obtain which kinds of requirements information and which modeling methods will be used to capture that information.

The plan serves as the basis for team consensus on exactly what will be done, provides a yardstick for tracking progress, and serves as a guide to new personnel and other stakeholders. The exact plan contents should vary depending on the organization's process and the specific characteristics of the project. In general, however, it should answer the following kinds of questions for the reader:

- Roles and Responsibilities – Who is responsible for what?
- Project Background – What background information will help understand this project?
- Requirements Process – What idealized requirements process will we follow?
- Mechanisms, methods, techniques – How will we elicit, identify, analyze, define, specify, prioritize, track, etc.?
- Quality assessment – What methods will be used to assess requirements qualities and what are the acceptance criteria for the products produced?
- Detailed schedule, milestones – How are the activities and artifacts mapped to the project schedule and milestones?
- Resources and References – Who or what resources can answer questions about the product or process?

The instantiation of a well-defined process in the project plan helps ensure that the process actually enacted by project personnel will be consistent with the organization's overall process goals. Observing and measuring the results then provides metrics for systematic process improvement.

The final key to implementing the plan is providing adequate resources. Historical data from a large set NASA projects (Figure 4) shows that, in general, the projects that spent the least on developing requirements tended to have the highest cost overruns. Projects that spend 8% to 14% of

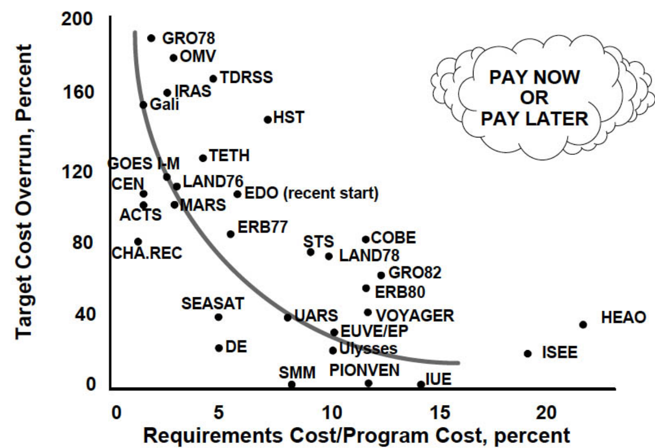

Figure 4: Requirements spending vs. cost overruns

the total project budget on acquiring and managing requirements reduced cost overruns by 50% ([NASA 05], [Young 06]).

6. Requirements for the Software Requirements Specification

The goals of the requirements process, the attendant difficulties, and the role of the requirements specification in a disciplined process determine the properties of a "good" requirements specification. These properties do not mandate any particular specification method but do describe characteristics an effective method should possess.

SRS Semantic Properties	SRS Packaging Properties
Complete	Modifiable
Implementation independent	Readable
Unambiguous and consistent	Organized for reference and review
Precise	
Verifiable	

Table 1: Semantic properties vs. packaging properties

In discussing the properties of a good SRS, it useful to distinguish **semantic** properties from **packaging** properties [Faulk 92]. Semantic properties are a consequence of *what* the specification says (i.e., its meaning or semantics). Packaging properties are a consequence of how the requirements are written down - the format, organization, and presentation of the information. The semantic properties determine how effectively an SRS captures the software requirements. The packaging properties determine how useable the resulting specification is. Table 1 illustrates the classification of properties of a good SRS. An SRS that satisfies the semantic properties of a good specification is:

- *Complete*. The SRS defines the set of acceptable implementations. It should contain all the information needed to write software that is acceptable to the customer and no more. Any implementation that satisfies every statement in the requirements is an acceptable product. Where information is not available before development begins, areas of incompleteness must be explicitly indicated [Parnas 86].
- *Implementation independent*. The SRS should be free of design and implementation decisions unless those decisions reflect actual requirements.
- *Unambiguous and Consistent*. If the SRS is subject to conflicting interpretation, the different parties will not agree on what is to be built or whether the right software has been built. Every requirement should have only one possible interpretation. Similarly, no two statements of required behavior should conflict.
- *Precise*. The SRS should define exactly the required behavior. For each output, it should define the range of acceptable values for every input. The SRS should define any applicable timing constraints such as minimum and maximum acceptable delay.
- *Verifiable*. A requirement is verifiable if it is possible to determine unambiguously whether a given implementation satisfies the requirement or not. For example, a behavioral requirement is verifiable if it is possible to determine, for any given test case (i.e., an input and an output), whether the output represents an acceptable behavior of the software given the input and the system state.

An SRS that satisfies the packaging properties of a good specification[1] is:

- *Modifiable*. The SRS must be organized for ease of change. Since no organization can be equally easy to change for all possible changes, the requirements analysis process must identify expected changes and the relative likelihood of their occurrence. The specification is then organized to limit the effect of likely changes.
- *Readable*. The SRS must be understandable by the parties that use it. It should

[1] *Reusability* is also a packaging property and becomes an attribute of a good specification where reusability of requirements specifications is a goal.

clearly relate the elements of the problem space as understood by the customer to the observable behavior of the software.

- *Organized for reference and review.* The SRS is the primary technical specification of the software requirements. It is the repository for all the decisions made during analysis about what should be built. It is the document reviewed by the customer or his representatives. It is the primary arbitrator of disputes. As such the document must be organized for quick and easy reference. It must be clear where each decision about the requirements belongs. It must be possible to answer specific questions about the requirements quickly and easily.

To address the difficulties associated with writing and using an SRS, a requirements approach must provide techniques addressing both semantic and packaging properties. It is also desirable that the conceptual structures of the approach treat the semantic and packaging properties as distinct concerns (i.e., as independently as possible). This allows one to change the presentation of the SRS without changing its meaning.

In aggregate, these properties of a good SRS represent an ideal. Some of the properties may be unachievable, particularly over the short term. For example, a common complaint is that one cannot develop complete requirements before design begins because the customer does not yet fully understand what he wants or is still making changes. Further, different SRS "requirements" mitigate toward conflicting solutions. A commonly cited example is the use of English prose to express requirements. English is readily understood but notoriously ambiguous and imprecise. Conversely, formal languages are precise and unambiguous, but can be difficult to read.

Although the ideal SRS may be unachievable, possessing a common understanding of what constitutes an ideal SRS is important [Parnas 86] because it:

- Provides a basis for standardizing an organization's processes and products,
- Provides a standard against which progress can be measured, and,
- Provides guidance - it helps developers understand what needs to be done next and when they are finished.

Because it is so often true that (1) requirements cannot be fully understood before at least starting to build the system and (2) a perfect SRS cannot be produced even when the requirements are understood, some approaches advocated in the literature do not even attempt to produce a definitive SRS. For example, some authors advocate going directly from a problem model to design or from a prototype implementation to the code. While such approaches may be effective on some developments, they are inconsistent with the notion of software development as an *engineering* discipline. The development of technical specifications is an essential part of a controlled engineering process. This does not mean that the SRS must be entire or perfect before anything else is done but that its development is a fundamental goal of the process as a whole. That we may currently lack the ability to write good specifications in some cases does not change the fact that it is useful and necessary to try.

7. State of the Practice

The past decade has brought a significant shift in requirements practice and the perception of the role of requirements in the development process. At the time the first version of this article was published, requirements analysis was generally treated as a distinct concern (e.g., [Davis 93]). There was the conceptual distinction that requirements should express an implementation independent specification of what the software should do. However, it was also treated as a development phase that divided the software process into distinct and relatively independent parts. It is this sequencing relationship that is represented in the waterfall model and its variations [e.g., Figure 1].

In this view, the requirements phase begins with requirements gathering, and ends with the delivery of some form of requirements specification to the software designers. While it is understood that the requirements activities and its products may be revisited in subsequent phases, it is assumed that the requirements specification can capture and communicate everything the developers need to know to design, implement, and maintain the software. In practice, this separation of concerns

was embodied in the notion of the "requirements handoff" – a process milestone in which the requirements specification is baselined and control passed to the software designers and coders.

The unstated assumption behind this model is that the dependencies between non-contiguous parts of the process do not require explicit understanding or management; that everything the stakeholders needed to know could be captured through work products like the SRS and supporting traceability matrices. Thus, for example, the designers do not need to understand the source of particular requirements or the underlying business rationale to design a good software architecture.

Over the past decade, a more holistic view of the software process has emerged. It has become clear that, for most complex software development, the decisions in each phase of development may have significant implications across the life cycle and, indeed, across more than one life cycle. Thus, controlling the downstream effects of development decisions requires explicit understanding and management of these dependencies. This requires a model of development that spans the software life cycle and, for some concerns, multiple life cycles.

In the remainder of this section we discuss the current state of practice, particularly as it embodies this broader, more interdisciplinary view of requirements.

7.1 Software Methodologies

Over the years, a number of analysis and specification methods have been developed as part of more comprehensive software engineering methods. The general trend has been for software engineering techniques to be applied first to coding problems (e.g., complexity, ease of change), then to similar problems occurring earlier and earlier in the life cycle. Thus the concepts of structured programming led eventually to structured design and analysis. Similarly, the concepts of object-oriented programming led to object oriented design and analysis.

The benefits of this approach are that a common set of conceptual structures and notations can be used across the software life cycle. It is unnecessary to translate from one set of abstractions to another (until code is produced), avoiding translation errors and inconsistencies between models. The drawback is that the same notations and structures must be used to represent concepts that we are trying to keep distinct. For example, the concept of *objects* is used to represent both entities in the problem domain (requirements) and entities in the implementation domain (code). This can make it difficult to distinguish requirements decisions from downstream concerns.

Since a number of the concepts used in current object-oriented approaches were introduced in Structured Analysis, and Structured Analysis is still in use in use in some application domains, our discussion will treat both.

Structured Analysis (SA)

Following the introduction of structured programming as a means to gain intellectual control over increasingly complex programs, structured analysis evolved from functional decomposition as a means to gain intellectual control over system problems.

The basic assumption behind SA is that the accidental difficulties can be addressed by a systematic approach to problem analysis using [Svoboda 90]:

- A common conceptual model for describing all problems,
- A set of procedures suggesting the general direction of analysis and an ordering on the steps,
- A set of guidelines or heuristics supporting decisions about the problem and its specification, and
- A set of criteria for evaluating the quality of the product.

What functional decomposition is still a part of SA, the focus of the analysis shifts from the processing steps to the data being processed. The analyst views the problem as constructing a system to transform data. He analyzes the sources and destinations of the data, determines what data must be held in storage, what transformations are done on the data, and the form of the output.

Common to the SA approaches is the use of data flow diagrams and data dictionaries. Data flow diagrams provide a graphic representation of the movement of data through the system (typically represented as arcs) and the transformations on

the data (typically represented as nodes). The data dictionary supports the data flow diagram by providing a repository for the definitions and descriptions of each data item on the diagrams. Required processing is captured in the definitions of the transformations. Associated with each transformation node is a specification of the processing the node does to transform the incoming data items to the outgoing data items. At the most detailed level, a transformation is defined using a textual specification called a "MiniSpec". A MiniSpec may be expressed in a number of different ways including English prose, decision tables, or a procedure definition language (PDL).

SA approaches originally evolved for management information systems (MIS). Examples of widely used strategies include those described by DeMarco [DeMarco 78] and Gane and Sarson [Gane 79]. "Modern" structured analysis was introduced to provide more guidance in modeling systems as data flows as exemplified by Yourdon [Yourdon 89].

Structured analysis is based on the notion that there should be a systematic (and hopefully predictable) approach to analyzing a problem, decomposing it into parts, and describing the relationships between the parts. By providing a well defined process, structured analysis seeks to address, at least in part, the accidental difficulties that result from ad hoc approaches and the definition of requirements as an afterthought. It seeks to address problems in comprehension and communication by using a common set of conceptual structures a graphic representation of the specification in terms of those structures, based on the assumption that a decomposition in terms of the data the system handles will be clearer and less inclined to change than one based on the functions performed.

While structured analysis techniques have continued to evolve and have been widely used, there remain a number of common criticisms. When used in problem analysis, a common complaint is that structured analysis provides insufficient guidance. Analysts have difficulty deciding which parts of the problem to model as data, which parts to model as transformations, and which parts should be aggregated. While the gross steps of the process are reasonably well defined, there is only very general guidance (in the form of heuristics) on what specific questions the analyst needs to answer next. Similarly, practitioners find it difficult to know when to stop decomposition and addition of detail. In fact, the basic structured analysis paradigm of modeling requirements as data flows and data transformations requires the analyst to make decisions about intermediate values (e.g., form and content of stored data and the details of internal transformations) that are not requirements. Particularly in the hands of less experienced practitioners, data flow models tend to incorporate a variety of detail that properly belongs to design or implementation.

Many of these difficulties result from the weak constraints imposed by the conceptual model. A goal of the developers of structured analysis was to create a very general approach to modeling systems; in fact, one that could be applied equally to model human enterprises, hardware applications, software applications of different kinds, and so on. Unfortunately, such generality can be achieved only by abstracting away any semantics that are not common to all of the types of systems potentially being modeled. The conceptual model itself can provide little guidance relevant to a particular system. Since the conceptual model applies equally to requirements analysis and design analysis, its semantics provide no basis for distinguishing the two. Similarly, such models can support only very weak syntactic criteria for assessing the quality of structured analysis specifications. For example, the test for completeness and consistency in data flow diagrams is limited to determining that the transformations at each level are consistent in name and number with the data flows of the level above.

This does not mean one cannot develop data flow specifications that are easy to understand, communicate effectively with the user, or capture required behavior correctly. The large number of systems developed using structured analysis show that it is possible to do so. However, the weakness of the conceptual model means that a specification's quality depends largely on the experience, insight, and expertise of the analyst. The developer must provide the necessary discipline because the model itself is relatively unconstrained.

Finally, structured analysis provides little support for producing an SRS meeting our quality criteria. Data flow diagrams are unsuitable for capturing mathematical relations or detailed specifications of value, timing, or accuracy so the detailed be-

havioral specifications are typically given in English or as pseudo–code segments in the Minispecs. These constructs provide little or no support for writing an SRS that is complete, implementation independent, unambiguous, consistent, precise, and verifiable. Further, the data flow diagrams and attendant dictionaries do not, themselves, provide support for organizing an SRS to satisfy the packaging goals of readability, ease of reference and review, or reusability. In fact, for many of the published methods, there is no explicit process step, structure, or guidance for producing an SRS, as a distinct development product, at all.

Object Oriented Analysis (OOA)

OOA has evolved from at least two significant sources, information modeling and object oriented design. Each has contributed to current views of OOA, and the proponents of each emphasize somewhat different sets of concepts. OOA techniques differ from structured analysis in their approach to decomposing a problem into parts and in the methods for describing the relationships between the parts. In OOA, the analyst decomposes the problem into a set of interacting objects based on the entities and relationships extant in the problem domain. An object encapsulates a related set of data, processing, and state (thus, a significant distinction between object oriented analysis and structured analysis is that OOA encapsulates both data and related processing together).

The structural components of OOA (e.g., objects, classes, services, aggregation) support a set of analytic principles. Of these, two directly address requirements problems:

1. From information modeling comes the assumption that a problem is easiest to understand and communicate if the conceptual structures created during analysis map directly to entities and relationships in the problem domain. This principle is realized in OOA through the heuristic of representing problem domain objects and relationships of interest as OOA objects and relationships. Thus an OOA specification of a vehicle registration system might model vehicles, vehicle owners, vehicle title, and so on as objects. The object paradigm is used to model both the problem and the relevant problem context.

2. From early work on modularization by Parnas [Parnas 72] and abstract data types, by way of object oriented programming and design, come the principles of information hiding and abstraction. The principle of information hiding guides one to limit access to information on which other parts of the system should not depend. In an OO specification of requirements, this principle is applied to hide details of design and implementation. In OOA, behavior requirements are specified in terms of the data and services provided on the object interfaces; the object encapsulates how those services are implemented. The principle of abstraction says that only the relevant or essential information should be presented. Abstraction is implemented in OOA by defining object interfaces that provide access only to essential data or state information encapsulated by an object (conversely hiding the accidentals).

The principles and mechanisms of OOA provide a basis for attacking the essential difficulties of comprehension, communication, and control. The principle of problem domain modeling helps guide the analyst in distinguishing requirements (what) from design (how). Where the objects and their relationships faithfully model entities and relationships in the problem, they are understandable by the customer and other domain experts; this supports early comprehension of the requirements.

The principles of information hiding and abstraction, with the attendant object structures, provide mechanisms useful for addressing the essential problems of control and communication. Objects provide the means to divide the requirements into distinct parts, abstract from details, and limit unnecessary dependencies between the parts. Object interfaces can be used to hide irrelevant detail and define abstractions providing only the essential information. This provides a basis for managing complexity and improving readability. Likewise objects provide a basis for constructing reusable requirements units of related functions and data.

The potential benefits of OOA are often diluted by the way the key principles are manifested in particular methods. While the objects and relations of OOA are intended to model essential aspects of the application domain, this goal is typically not supported by a corresponding conceptual model of the domain behavior. Object modeling mecha-

nisms and techniques are intentionally generic rather than application specific. One result is insufficient guidance in developing appropriate object decompositions. OOA practitioners often have difficulty choosing appropriate objects and relationships.

In practice, the notion that one can develop the structure of a system, or a requirements specification, based on physical structure is often oversold. It is true that the elements of the physical world are usually stable (especially relative to software details) and that real–world based models have intuitive appeal. It is not true, however, that everything that must be captured in requirements has a physical analog. An obvious example is shared state information. Further, many real world structures are themselves arbitrary and likely to change (e.g., where two hardware functions are put on one physical platform to reduce cost). While the notion of basing requirements structure on physical structure is a useful heuristic, more is needed to develop a complete and consistent requirements specification.

A further difficulty is that the notations and semantics of OOA methods are typically based on the conceptual structures of software rather than those of the problem domain the analyst seeks to model. Symptomatic of this problem is that analysts find themselves debating about object language features and their properties rather than about the properties of the problem. An example is the use of message passing, complete with message passing protocols, where one object uses information defined in another. In the problem domain it is often irrelevant whether information is actively solicited or passively received. In fact there may be no notion of messages or transmission at all. Nonetheless one finds analysts debating about which object should initiate a request and the resulting anomaly of passive entities modeled as active. For example, to get information from a book one might request that the book "read itself" and "send" the requested information in a message. To control an aircraft the pilot might "use his hands and feet to 'send messages' to the aircraft controls which in turn send messages to the aircraft control surfaces to modify themselves" [Davis 93]. Such decisions are about OOA mechanisms or design, not about the problem domain or requirements.

As mentioned in the previous section, where the decomposition into objects is driven only by use cases, the result is effectively a functional specification in object guise. The problems with such specifications are well understood [Parnas 72], in particular, being difficult to understand, change, or maintain.

A more serious complaint is that most OOA methods inadequately address our goal of developing a good SRS. Most OOA approaches in the literature provide only informal specification mechanisms, relying on refinement of the OO model in design and implementation to add detail and precision. There is no formal basis for determining if a specification is complete, consistent, or verifiable. Further, the approach does not directly address the issues of developing the SRS as a reference document. The focus is on problem analysis rather than specification. If the SRS is addressed at all, the assumption is that the principles applied to problem understanding and modeling are sufficient, when results are written down, to produce a good specification. Experience suggests otherwise. As we have discussed, there are inherently tradeoffs that must be made to develop a specification that meets the need of any particular project. Making effective tradeoffs requires a disciplined and thoughtful approach to the SRS itself, not just the problem. Thus, while OOA provide the means to address packaging issues, there is typically little methodological emphasis on issues like modifiability or organization of a specification for reference and review.

7.2 Use cases

Usage scenarios or *use cases* have been widely adopted as a method for specifying required system behavior from the user's point of view. Use cases are sometimes deployed as the primary focus of elicitation and problem modeling [Schneider 98]. Use cases are also frequently employed as a first step in many object-oriented approaches (e.g., [Jacobsen 92], [Kruchten 99]). Despite their prevalence in object oriented development, there is nothing intrinsically object-oriented about use cases and they are applied in other contexts. For these reasons, we will treat them separately.

Briefly, a use case describes a set of possible sequences of interactions between the system and a user seeking to accomplish a particular goal. Uses cases are intended capture a user-centric view of

the required system behavior – i.e., how the system should respond to different user inputs to accomplish specific tasks like checking the balance on an account or adding an item to an on-line shopping cart.

While many approaches attempt to structure use cases by providing standard formats or templates (e.g., [Cockburn 00]), use cases are ultimately an informal, natural-language specification. A use-case template captures the user's (or *actor's*) interaction with the system as a sequence of natural-language statements that alternate between describing user inputs ("the customer clicks the *checkout* button") and system responses ("the page displays the contents of the customer's shopping cart").

Because use cases directly capture interaction with the system in terms of the user's problem domain (e.g., work tasks), they are usually easy for non-technical stakeholders to read, understand, review, and even assist in creating. While writing good use cases requires expertise, there is a relatively natural transition from a description of what a user wants the system to do, to a specification of how the system might support that task in a use case. Similarly, marketing or business goals for a system (e.g., what new things the system will allow users to do) are often straightforwardly represented as use cases [Lee 99].

While there is evidence that use cases can be an effective informal modeling technique, they lack many of the properties necessary to a technical requirements specification:

- *Unambiguous and consistent*: Use cases necessarily have all the limitations of any natural language specification. They are inherently ambiguous and open to inconsistent interpretation by stakeholders or developers.
- *Modifiable*: Individually, use cases are relatively easy to modify, particularly where standard templates are used. Collectively, where there are a large number of use cases, it can become very difficult to find or identify all of the use cases relating to a particular change.
- *Organized for reference and review*: Where the number of use cases becomes large, it also becomes difficult to find specific use cases or specific information. There is generally no organizing principle that accurately characterizes exactly where to put or find a given piece of information among the set of use cases. Similarly, it can be difficult for reviewers to find key information or assess basic properties like consistency.
- *Complete*: Since use cases represent specific paths through the system behavior, it is usually impossible or impractical to write a complete set of use cases. The problem is analogous to trying to write a complete set of test cases. While the level of abstraction is higher, in general, the number of possible scenarios is very large and there is no way to check if the set of use cases is complete, nor to identify which ones might be missing.

There are also more important senses in which use cases are typically incomplete. Traditionally, use cases represent only users' interactions with the system. It follows that a specification written only in terms of use cases is an entirely functional specification. Other viewpoints as well as critical quality requirements are not addressed. Such an approach recapitulates the deficiencies of functional decomposition and discards decades of progress in software engineering. While there have been some efforts to modify use cases to represent quality requirements, (e.g., [Bass 03]) such approaches remain a work in progress.

These limitations suggest that use cases are more appropriate for informal business- or mission-oriented requirements capture. In many organizations there are two distinct audiences for the requirements: one audience that is versed in the organizational goals and problem domain and a second audience that is versed in technical goals and the solution domain. For businesses, the first audience typically includes customers, marketing, product management, and others on the business side of the organization. The second audience includes architects, coders, and others on the development side of the organization.

Because these two audiences tend to speak different languages and have different interests in the product, it is difficult to write any single specification that is suitable to both. In such cases, it often makes sense to create two distinct documents, one owned by the business side and a second owned by the technical side. The goal in dividing

the specification is to create a clear allocation of purpose, responsibility, and ownership.

The purpose of the business-oriented document is to capture the rationale for building the system. It includes the business case, solution approach, and the mapping between them. This document may be described as the*, Market Requirements Document (MRD), Business Requirements Document (BRD)* or, *Concept of Operation Document (ConOps)*. It should communicate the results of problem analysis and characterize the set of acceptable solutions to customers, managers, and others responsible for why the system is being developed. Because its purpose is to capture rationale, it is organized to "tell a story" [Fairley 97] rather than as a reference document.

The technical specifications are then captured in an SRS. By tracing requirements in the SRS to the BRD or similar document, one captures the origin and rationale for the technical requirements while maintaining the desirable properties of an SRS.

Use cases are a natural fit for the audience and purpose of a document like the ConOps or BRD. Use cases are written in terms of the problem domain and in a language that is accessible to those familiar with the problem domain. The format and organization is consistent with the objective that the document should "tell a story" and provides a vehicle for linking the system behavior to user tasks. While this comes at the expense of some redundancy in that the same requirements must be expressed in more than place, the benefits typically outweigh any issues in maintaining consistency.

7.2 Linking requirements to architecture

While a detailed discussion of software architecture is beyond the scope of this paper, one must have a clear understanding of the effect of architecture on important system qualities to understand the relationships between architectural design decisions and the requirements process.

We use the term *software architecture* to denote the structures of the system comprising a set of components, relations, and interfaces. For example, the *class structure* could refer to the set of classes in the system, the class interfaces, and the inheritance or instance relation. The *process structure* could refer to the organization of the system into processes or threads; interfaces are the inter-process operations (synchronization, communication), and the relations include exclusion and concurrency. By this definition, any software system comprises more than one architecture [Bass 03].

Architecture manifests the earliest set of design decisions. It is these decisions that enable or inhibit the system's quality attributes. These include essentially all of the system's developmental qualities (e.g., maintainability, reusability, etc.) and all of the system's behavioral qualities (e.g., performance, reliability, etc.) except functionality[2].

Inevitably, architectural design requires making tradeoffs among the system's quality attributes. For example, significantly increasing system security will tend to decrease performance and improving reliability will typically require longer development time.

Since different stakeholders have different interests in system properties, the process of choosing among architectural design alternatives directly affects the extent to which the design will, or will not, satisfy their desires and goals. Since making good architectural design decisions requires making tradeoffs among the concerns of different stakeholders, the architect must understand the rationale for different quality requirements, as well as the relative priorities among stakeholder goals, and, ultimately, negotiate compromises. The architect must understand both the source and nature of the system's quality requirements.

The implication is that it is not sufficient to communicate black-box requirements; an effective process must also capture and communicate contextual information. This includes the purpose of different requirements, their relationships to organizational goals, and their importance to the system's diverse stakeholders.

Where an organization goes on to develop subsequent versions of the software or similar systems, the dependencies also extend downstream. The architectural design decisions embodied in the current system tend to influence subsequent business goals, requirements, and architectural structures. For example, how easy a system is to extend or modify the software in particular ways can

[2] Without going into detail, precisely the same functionality can be realized by any number of different architectural decompositions.

significantly affect the ability to add specific features, address new customer needs, or target different markets.

These overlapping dependencies between developmental goals, requirements, and architectural design are captured in what Bass, et al [Bass 03] call the *architectural business cycle*. While our focus is on the role of requirements in that cycle, it expresses the key idea that there are important dependencies between the conceptually distinct activities of software development. Managing the implications of these dependencies requires explicit two-way communication between the business and technical parts of an organization. The activities and artifacts supporting this communication must be part of a disciplined process.

7.3 Elicitation Methods and Goal Modeling

Failing to understand what the stakeholders want leads to substantial rework [Boehm 88] or even rejection of a system. Because elicitation occurs at the beginning of development, errors in this step are the most expensive and difficult to correct later in the process. The importance of getting these early steps right has led to a wide range of efforts focused on understanding elicitation issues and supporting improved elicitation processes.

One significant result of these efforts has been a shift in the way researchers and practitioners view elicitation. While there were exceptions (e.g. [Gause 89]), the prevailing view in the past was that there existed some set of requirements characterizing the behavior of an ideal system. One could effectively elicit those requirements by asking a few key people, notably customers, and users, what the system should do.

For many of the reasons that we have discussed, this approach often proved ineffective. This reflects the fact that "what is wanted" is typically not well defined, fully understood, or even a single thing. Rather, the perception of the problem, developmental goals, and requirements will vary from one stakeholder to the next, and even for a single stakeholder, over time. Any individual stakeholder's answers will yield a view that is neither complete nor precise. Views from multiple stakeholders tend to be inconsistent or conflicting.

The upshot is that the notion of an ideal system or set of requirements that can be "discovered" is a poor approximation of reality. Rather, there are many different perspectives on the problem, partial views of solutions, and possible systems. The central challenge of elicitation is to obtain and reconcile these different perspectives to a single system definition that the stakeholders can live with.

Where, historically, this aspect of the requirements process received little attention, it has recently emerged as a distinct discipline in both practice and the literature. The understanding that elicitation must reconcile many different views from different kinds of stakeholders, and in different contexts, has stimulated research into the various facets of elicitation. This has, in turn, stimulated development of a number of elicitation methods targeted to different needs. An overview of the approaches is given in [Nuseibeh 00]; a more complete survey of different elicitation methods is given in [Lauesen 02].

Goal Modeling

An elicitation approach that integrates systematic modeling of objectives (e.g., business goals) with downstream requirements activities is *that of goal modeling* or *goal-oriented requirements*. A *goal* specifies some objective that the system should achieve [Lamsweerde 01]. The essential foci of goal-oriented requirements are:

1. To capture the stakeholder's objectives for the system in the problem context.
2. To systematically map those objectives to a detailed specification of the system requirements.

By beginning with goals, the approach seeks to capture each stakeholder's rationale for the system in the stakeholder's language and context. Thus, for example, business goals might be captured in terms of market opportunities and user needs in terms of ease of performing a work task. Expressing the system objectives using the stakeholder's perspective and language supports ease of understanding and elicitation. Integrating the different views of system goals provides an early opportunity for identifying and resolving conflicts [Robinson 89]. Subsequent refinement links rationale to specific system requirements. This supports two-way traceability and communication as goals or requirements evolve.

A relatively complete approach to requirements based on goals is the KAOS method by

Lamsweerde *et al* [Lamsweerde 09]. This work integrates goal-based elicitation with formal modeling and analysis. A formal language and tool support reasoning and the automated analysis of some completeness and consistency properties. Related publications include case studies of industrial experience (e.g., [Winter 01]). A good overview of goal-oriented requirements and set of references is given in [Lamsweerde 01].

7.4 "Agile" methods

Much recent attention has been given to a set of development approaches that their authors characterize as "agile," for example, extreme programming [Beck 04], scrum [Rising 02], or the Agile Unified Process [Ambler 02]. While there are differences among agile methods, they share a code-centered view of development – the view that the development effort should focus on the implementation rather than documentation (see the "agile manifesto[3]").

The emphasis on code at the expense of documentation particularly pertains to the software requirements. Requirements documentation ranges from small amounts of informal documentation to using the code as the primary repository for all requirements and design decisions. This more extreme view is reflected in statements like: "The urge to write requirements documentation should be transformed into an urge to instead collaborate closely with your stakeholders and then create working software based on what they tell you[4]."

It should be clear that the software engineering philosophy behind these methods is at odds with what we have characterized as a "disciplined approach." To understand why this difference arises, it is necessary to examine the differences in methodological goals and the underlying assumptions the different approaches make about software development. By understanding the extent to which each approach's assumptions do or do not hold, the reader has a basis for choosing the approach best fitting a particular development situation.

Agile approaches seek address the essential difficulties of *comprehension, communication,* and *control* by shortening the development cycle and bringing key stakeholders into the development loop. Many of the difficulties of traditional development processes (i.e., "waterfall" and its variations) arise from the temporal distance between project conceptualization and the delivery of any working software. In big projects, it may be months, or even years, between the time stakeholders begin describing their requirements and the time the developers can show them software that presumes to meet those requirements.

Because stakeholders typically do not know exactly what they want until they see it, this is often the point at which developers find out that what they have built is, in part or whole, not acceptable to the stakeholders. Because all of the work of design and implementation has been founded on incorrect requirements, fixing these errors is difficult and expensive. The result is a system that costs more than it should and delivers less than the stakeholders want.

Many of these problems can be avoided if it is possible to drastically shorten the development cycle. For agile methods, this cycle time is on the order of two to four weeks rather than months. Instead of eliciting all of the customer's requirements, the goal is to capture a small number of the most important ones (typically two or three). This small subset of requirements is then taken to code and validated with the customer. This cycle repeats until the customer is satisfied with the product. Little, if any, documentation is created or maintained. Rather, the code is the primary repository of the evolving set of requirements and design decisions.

With a short cycle time, the customer very quickly sees the expression of his requirements in the (partial) software. Errors and misunderstandings can be detected and corrected each cycle. Where errors occur, relatively little effort has been expended and the amount of rework may be limited to the length of the increment. Continuous communication between developers and the customer reduces the opportunity for misunderstanding. Because the developers are constantly integrating new requirements, requirements changes are addressed in the normal course of iterative development.

However, these benefits come at a substantial cost. Since only a small number of requirements can be considered at any time, there is no opportunity to understand the relationship of require-

[3] http://agilemanifesto.org/

[4] http://www.agilemodeling.com/essays/agileRequirementsBestPractices.htm

ments to long-term goals, relationships between requirements, or the relationship between requirements and system structure:

- Because requirements are not gathered or considered in advance, it is not possible for the designer to anticipate likely changes. There is constant rework as new requirements are added.
- Since only a very small subset of the requirements is examined at any one time, there is no mechanism to balance goals and make tradeoffs. Nor is there an opportunity to detect conflicting requirements before coding begins.
- Since the wide range of possible quality requirements that are whole-system properties (e.g., performance, safety, reliability, etc.) are not considered together, there is no opportunity to develop an architecture that balances such concerns. Similarly, constant restructuring (refactoring) makes it difficult to establish or maintain architectural properties.
- Constant interaction with the stakeholders is not just desirable, but essential. Without constant feedback validating the development, errors will accumulate over time, obviating the benefits of rapid increments.
- Because nothing is written down, progress depends on personnel who are intimately familiarity with the code. There is no mechanism to control the downstream effects of decisions on properties like maintainability or reusability.

Thus, realizing the benefits of agile methods depends on certain assumptions being true of the product, process, and people involved. It is a process that acts as if the development has neither a past nor a future, reacting only to immediate needs. Clearly there are many kinds of systems and development situations that are inconsistent with these assumptions, to name a few:

- Where there is limited availability or communication with stakeholders.
- Where stakeholders have conflicting views and requirements.
- Where there are critical behavioral and developmental properties that must be addressed by the architecture such as safety, reliability, or performance.
- Where requirements are relatively stable or predictable.
- Where there is a history of developing similar systems or the current system is a new version of a previous one.
- Where the development team is not co-located and frequent, high-bandwidth communication is not possible.
- Where the system is long lived and maintenance is a key concern - and so on.

In essence, agile approaches make an implicit assumption that the software requirements are relatively independent. It cannot be otherwise. If there are strong dependencies between requirements then the order in which requirements are addressed and design decisions are made significantly affects overall system properties including how easily the software can be changed to address subsequent requirements. These effects have been well understood for decades (e.g., [Parnas 76]). One obvious example is where requirements from different users conflict. Taking such requirements in arbitrary order (as opposed to considering them together) will result in an implementation that first meets one stakeholder's needs, then the other's, but never both.

It follows that there can be only limited circumstances in which the benefits of agile methods outweigh the costs and risks. The notion that most development efforts can abandon a disciplined approach to requirements in favor of coding is not supportable. Unfortunately, many proponents of these methods do not make the underlying assumptions clear nor provide a balanced discussion of the limitations. Leaving this as an exercise for the reader may be good salesmanship but is poor software engineering. A somewhat more even-handed view can be found in [Boehm 02]. A more critical view that encompasses some of the issues of agile methods and XP is given in [Stephens 01].

7.5 Software product-lines

A view of development that spans multiple product cycles is that of software product-lines. Briefly, a software product line is a family of systems that share a significant number of common re-

quirements, and are produced from a common set of reusable software assets. The reusable assets typically include a common software architecture, reusable, adaptable code modules, test cases, documentation, and so on.

Conventional software processes follow a "craftsman" production model – i.e., skilled individuals build each system by hand. Product-line development is more analogous to a manufacturing model where one builds a factory, then uses the factory to produce products. Software product lines are constructed by first creating a set of reusable assets, tools for deploying the assets (e.g., code generators), and a process for using the assets to produce members of the product line. Software systems are then created from the common assets.

Where applicable, software product-line approaches have been shown to significantly increase productivity (by as much as an order of magnitude), while decreasing cycle time and improving quality. Since code can be quickly created from reusable assets and validated with the customer, it provides the benefits of a rapid cycle time.

The approach, however, is applicable only where an organization is developing a number of reasonably similar systems. Refreshingly, the proponents of product-line approaches are careful not only to state the underlying assumptions (e.g., [Weiss 99]), but also to provide specific methods for assessing the costs and risks of applying a product-line approach to any particular application (also [Clements 01], [Pohl 05]).

The relevance of software product-lines to this discussion is that product line processes exemplify a disciplined approach to requirements that spans multiple software life cycles. Software product-lines work by amortizing the larger up-front development costs of the common asset base over the delivery of a number of similar software products. To create a reusable architecture and set of assets, the developers must understand not only the requirements for the next software system, but how those requirements are likely to vary over future instances of the product-line. In particular, which requirements should be the same across all members of the product-line (called *commonalities*) and which requirements are allowed to differ (called *variabilities*).

This entails understanding both the current business objectives and how those objectives are likely to change over time. It also requires an understanding of the relationship of the requirements to the architecture, and how architectural design decisions will affect the future ability to build different versions of the product-line.

A variety of approaches to product-line requirements have been proposed and used. A significant difference from other requirements approaches has been a substantial body of work focusing on identifying and managing variabilities and the relationships between them (e.g., [Svahnber 05], [Pohl 05]). These works provide useful insight into disciplined approaches to managing requirements across multiple products and development cycles.

7.5 Practical Formal Methods

Like so many of the promising technologies in requirements, the application of formal methods is characterized by an essential dilemma. On one hand, formal specification techniques hold out the only real hope for producing specifications that are precise, unambiguous, and demonstrably complete or consistent. On the other, industrial practitioners widely view formal methods as impractical. Difficulty of use, inability to scale, readability, and cost are among the reasons cited. Thus, in spite of significant technical progress and a growing body of literature, the pace of adoption by industry has been extremely slow.

In spite of the technical and technology-transfer difficulties, increased formality is necessary. Only by placing behavioral specification on a mathematical basis will we be able to acquire sufficient intellectual control to develop complex systems with any assurance that they satisfy their intended purpose and provide necessary properties like safety. While it is not necessary to apply formal methods to al systems, or even all parts of critical systems, they are needed where it is necessary to establish correctness of the essential parts of critical systems (e.g., safety critical aspects). The solution is better formal methods - methods that are practical given the time, cost, and personnel constraints of industrial development.

Engineering models and the training to use them are *de rigueur* in every other discipline that builds large, complex, or safety–critical systems. Build-

ers of a bridge or skyscraper who did not employ proven methods or mathematical models to predict reliability and safety would be held criminally negligent in the event of failure. It is only the relative youth of the software discipline that permits us to get away with less. But, we cannot expect great progress overnight. As Jackson [Jackson 94] notes, the field is sufficiently immature that "the prerequisites for a more mathematical approach are not in place." Further, many of those practicing our craft lack the background required of licensed engineers in other disciplines [Parnas 89]. Nonetheless, sufficient work has been done to show that more formal approaches are practical and effective in industry. The Naval Research Laboratory's (NRL) Software Cost Reduction (SCR) method and tools exemplify such an approach.

The Software Cost Reduction (SCR) Method: Where most of the techniques thus far discussed focus on problem analysis, the requirements work at the United States Naval Research Laboratory focused equally on issues of developing a good SRS [Heninger 80]. As part of an overall effort in validating software engineering methodologies the SCR project has developed rigorous approaches to requirements specification and documentation based on an underlying formal model.

The SCR approach uses formal, mathematically based specifications of acceptable system outputs to support development of a specification that is unambiguous, precise, and verifiable. It also provides techniques for checking a specification for a variety of completeness and consistency properties. The SCR approach introduced principles and techniques to support our SRS packaging goals including the principle of separation of concerns to aid readability and support ease of change. It includes the use of a standard structure for an SRS specification and the use of tabular specifications that improve readability, modifiability, and facilitate use of the specification for reference and review.

While other requirements approaches have stated similar objectives, the SCR project is unique in having applied software engineering principles to develop a standard SRS organization, a specification method, review method [Parnas 85a], and notations consistent with those principles. The SCR project is also unique in making publicly available a complete, model SRS of a significant system [Alspaugh 92].

More recently, NRL has extended the SCR work to provide a suite of supporting tools. Since the approach is based on a formal model, the tools not only assist the developer in creating well-formed specification, the tools provide automated checking for the specification's completeness and consistency ([Heitmeyer 95a], [Heitmeyer 95b]). Likewise, the model can be used to support automated proofs of semantic properties like system safety properties [Heitmeyer 98] or fault tolerance [Jeffords 09]. The work has also shown some of the promise of formal methods in supported automated test case generation [Gargantini 99] and even code generation [Rothamel 06].

While the SCR requirements approach is reasonably general, many of the specification techniques and models are targeted to real–time, embedded applications. More work needs to be done toward providing the benefits of practical formal methods to other types of systems.

8. Trends and Emerging Technology

There has been increasing agreement on the underlying problems in requirements as well as on the general characteristics of an effective requirements process. However, the overall trend has not been toward a common methodology, but toward a broadening of the concerns addressed and a proliferation of approaches.

These trends in requirements reflect more general trends in software engineering and software technology. As discussed in the section on processes, early life cycle models tended to treat the conceptually distinct activities of software development like requirements, design, and coding, as relatively independent phases. This reflected a desire to divide the development process into activities that addressed distinct concerns, with well-defined inputs and outputs.

With increasing application complexity and diversity of users, this paradigm has changed. More recent process models tend to reflect the view that the activities of the software life cycle are heavily interdependent and necessarily interleaved in time. Thus, for example, requirements activities may persist, if with diminishing effort, until the customer accepts the product. Where the software is developed in several versions, or part of a software product line, some requirements activities

may continue across multiple delivery cycles ([Clements 01], [Faulk 01]).

At the same time, software has become increasingly ubiquitous. The types of applications along with the number and kinds of stakeholders have grown almost as fast as the size and complexity of the systems we build. One result has been an increasing diversity of development contexts and kinds of stakeholders.

Requirements research and practice have followed suit in broadening the scope of requirements activities and the diversity of methods. Thus, for example, we have seen new elicitation methods emerge to address different contexts and stakeholders. Likewise, requirements activities have been extended to encompass an organization's long-term goals and, in the case of software product lines, multiple developments or development cycles. We see these trends continuing in several areas of research and development:

Domain specificity: Requirements methods will provide improved support for understanding, specification, analysis, and usefulness by being tailored or created to address particular classes of problems.

Historically requirements approaches have been advanced as being equally useful to a wide variety of types of applications. For example, structured analysis methods based on conceptual models that were intended to be "universally applicable" (e.g., [Ross 77]); similar claims have been made for object-oriented approaches and notations like UML (e.g., [OMG 05]).

Such generality comes at the expense of ease of use and amount of work the analyst must do for any particular application. Where the underlying models have been tailored to a particular class of applications, the properties common to the class can be embedded in the model. The amount of work necessary to adapt the model to a specific instance of the class is relatively small. The more general the model, the more decisions that must be made, the more information that must be provided, and the more tailoring that must be done. This provides increased room for error and, since each analyst will approach the problem differently, makes solutions difficult to standardize. In particular, such generality precludes standardization of sufficiently rigorous models to support algorithmic analysis of properties like completeness and consistency.

Jackson [94] has expressed similar points. He points out that some of the characteristics separating real engineering disciplines from what is euphemistically described as "software engineering" are well understood procedures, mathematical models, and standard designs specific to narrow classes of applications. Jackson points out the need for software methods based on the conceptual structures and mathematical models of behavior inherent in a given problem domain (e.g., publication, command and control, accounting, and so on). Such common underlying constructs can provide the engineer guidance in developing the specification for a particular system.

This trend is currently reflected in the proliferation of elicitation methods and models targeted to different development contexts. It is also evidenced in the trend toward tailoring the overall requirements processes [Young 06] to address the specific concerns of a project or organization. The trend toward better integration of requirements processes with business processes (e.g., [Middleton 05]) will also further the trend toward domain specificity to meet the needs of specific business areas.

Currently lacking are domain specific approaches that encompass the artifacts, activities and roles comprising the entire requirements process. Some earlier work (e.g., [Prieto-Diaz 94], [Lam 97]) explored the potential of requirements reuse using domain specific methods. Likewise, both product-line approaches and methods based on domain specific modeling necessarily incorporate aspects of domain-specific requirements. For example, the use of the Embedded System Modeling Language (ESML) [Balasubramanian 07] on a family of embedded avionics applications [Karsai 02]. However, developing new requirements languages and semantics for specific domains remains a labor-intensive task. Progress in this area should see improved tool support (see the subsequent section on meta-engineering), new methods for modeling requirements in specific domains, and better guidance in adapting components to provide integrated processes.

Distributed Development: Another way in which the requirements problem has broadened (in a literal as well as figurative sense) is in the trend to-

ward *distributed development*. We use the term "distributed development" to denote software projects where development teams and activities are located in multiple geographic sites around the globe, particularly where sites are separated by time zones, cultures, and languages. While the early focus of globalization was on reduced cost, factors like increased access to talent and proximity to markets have continued to push the trend forward.

Distributed development has proven to have its own set of costs and risks, often requiring more effort and taking much longer than similar co-located projects [Mockus 01]. A key reason is the difficulty in achieving a common understanding of the requirements. In a cross-domain survey of industrial distributed developments, issues with misinterpreted, changing, and missing requirements ranked as the top three sources of error above all other development issues [Komi 05].

Experience suggests that distributed development is different from co-located projects (e.g., [Battin 01], Bradner 02]). These differences are manifestations of what Herbsleb characterizes as the key problem of distributed development, *coordination at a distance* [Herbsleb 07]. "Coordination," here, denotes the need to manage dependencies between people, tasks, and artifacts in a complex software development. In turn, difficulties in coordination are largely the result of difficulties in communicating effectively at a distance [Olson 2000], particularly where there are cultural, language, and organizational differences.

These differences suggest that new methods, models, and processes will be needed to manage requirements in distributed developments [Damian 07]. These will include new work in areas like cross-cultural requirements elicitation and communication. Likewise, new process models are needed for managing requirements elicitation, allocation, verification, and validation in a distributed project.

Personalization, Monitoring, and Adaptation: The trend toward broadening the scope of requirements engineering is evidenced in the areas of requirements personalization [Sutcliffe 06], requirements monitoring (e.g., [Fickas 95]), and real-time adaptation (e.g., [Robinson 05]). While these are three distinct areas of requirements research, they share a common concern for software *contextualization*: adapting software to a particular context such as user characteristics, the real-time environment, or the specific task.

Contextualization extends the issues around changing requirements to a personal and real-time level. Personalized software is software that is produced to meet the requirements of small groups or even individuals. This can include software that is individually customized, software that the user can customize, or software that configures itself based on user preferences. Real-time adaptation is customization in response to changes over time. For example, software that changes behavior as the system moves through space (e.g., on a cell phone) or software that changes behavior depending on the time of day. Where the software itself does the adaptation, it must monitor parameters relating to the requirements (e.g., time of day or location) and change behavior accordingly.

While, historically, there have been many approaches to software customization and even personalization[5], these have not been systematically addressed as a type of requirements variation. Only recently have researchers begun to look at systematic approaches to understanding and managing contextual requirements.

Basically, contextualization embraces cases where requirements remain fluid even at run time. While we may continue to make tradeoffs between different stakeholder's requirements, we may also view the system as implementing more than one set of requirements at a time, switching between them depending on the context of use.

As more and more personal devices include increasingly powerful computing systems (or access to networks), the trend toward personalization and other forms of contextualization will grow. There is likewise a trend toward integrating the results of several requirements areas to address various dimension of the contextualization problem.

Personal Contextual - Requirements Engineering (PC-RE) [Sutcliff 06] addresses the issue that user goals tend to change with context. As the user moves through time and space, objectives and, hence, requirements change. PC-RE proposes a framework for relating changing goals, requirements, and modes of implementation.

[5] The infamous Microsoft® "Clippy" being one.

Meta-Engineering: "Meta-engineering" refers to the engineering of engineering practices. All engineering disciplines include meta-engineering practices. An obvious example is that manufacturing necessarily includes processes for creating processes that will be used in a factory design to produce specific kinds of products.

Meta-engineering is an area in which software engineering excels [Faulk 10]. While creating "abstractions of abstractions" or designing "processes to design process" may sound convoluted, it is precisely these kinds of capabilities that allow new methods, processes, and even tools to be created and introduced into practice at a pace commensurate with changing technology.

While not discussed in these terms, meta-engineering capabilities underlie some of the advances we have discussed in this paper. In particular, the ability to systematically create or adapt requirements processes to satisfy specific project constraints (i.e., the process requirements) is a meta-engineering activity. Likewise is the development of new methodologies like agile or product-line engineering.

Product-line engineering is a particularly instructive case since the product-line engineering process, itself, embeds a meta-engineering process. Whenever the domain engineers develop a set of product-line assets, it is also necessary to create a process for using those assets (common architecture, libraries of adaptable modules, etc.) to create any software product that is a member of the product line. Thus, any complete product-line process model includes a process for creating the application engineering process. Of course, the product-line process is itself a product of meta-engineering.

Improved meta-engineering capabilities will be necessary to much of the evolution of requirements practice. Facilitating the practice of defining new requirements processes for specific application domains requires providing systematic processes for producing new processes to satisfy specific developmental goals or constraints. Similar capabilities will be needed for fitting elicitation methods, modeling methods, and artifacts to specific needs.

The same argument can be made for tools. While we have not seen meta-engineering tools targeted specifically to requirements, meta-engineering tools exist in other disciplines. For example, there are already methods and "tool-building-tools" supporting product-line engineering [Kelly 08]. Such tools aim to create tools supporting application engineering based on a domain model. Output of the tool is a code generator that takes a specification of the requirements for member of the product line and generates the application code.

The potential for creating meta-engineering tools to support requirements modeling and analysis provides substantial opportunity for fruitful research.

9. Conclusions

Requirements are intrinsically hard to do well. Beyond the need for discipline, there are a host of essential difficulties that attend both the understanding of requirements and their specification. Further, many of the difficulties in requirements will not yield to technical solution alone. Addressing all of the essential difficulties requires the application of technical solutions in the context of human factors such as the ability to manage complexity or communicate to diverse audiences. A requirements approach that does not account for both technical and human concerns can have only limited success. For developers seeking new methods, the lesson is *caveat emptor*. If someone tells you his method makes requirements easy, keep a hand on your wallet.

Nevertheless, difficulty is not impossibility and the inability to achieve perfection is not an excuse for surrender. While all of the approaches discussed have significant weaknesses, they all contribute to the attempt to make requirements analysis and specification a controlled, systematic, and effective process. Though there is no easy path, experience confirms that the use of **any** careful and systematic approach is preferable to an *ad hoc* and chaotic one. Further good news is that, if the requirements are done well, chances are much improved that the rest of the development will also go well. Unfortunately, *ad hoc* approaches remain the norm in much of the software industry.

A final observation is that the benefits of good requirements come at a cost. Such a difficult and exacting task cannot be done properly by personnel with inadequate experience, training, or resources. Providing the time and the means to do the job right is the task of responsible manage-

ment. The time to commit the best and brightest is before, not after, disaster occurs. The monumental failures of a host of ambitious developments bear witness to the folly of doing otherwise.

10. Further Reading

Those seeking more depth on requirements methodologies than this tutorial can provide have access to a number of good texts on software requirements. Berenbach, *et al* [Berenbach 09] focuses on practical approaches with depth in elicitation and quality attribute requirements. Weigers [Weigers 03] provides broad coverage with emphasis on the voice of the customer and requirements management. Young [Young 06] addresses effective practices and the role of a requirements plan. Middleton and Sutton [Middleton 05] provide a business-oriented approach driven by customer value.

Acknowledgements

The quality of this paper has been much improved thanks to thoughtful reviews by Merlin Dorfman and Richard Thayer. Paul Clements, Connie Heitmeyer, Jim Kirby, Bruce Labaw, Richard Morrison, and David Weiss provided helpful reviews of the first version.

REFERENCES

[Alford 79] M. Alford and J. Lawson, Software Requirements Engineering Methodology (Development), *RADC–TR–79–168*, U.S. Air Force Rome Air Development Center, Jun. 1979.

[Alspaugh 92] T. Alspaugh, S. Faulk, K. Britton, R. Parker, D. Parnas, and J. Shore, *Software Requirements for the A–7E Aircraft*, NRL/FR/5530–92–9194. Washington, D.C.: Naval Research Laboratory, 1992.

[Ambler 02] S. Ambler and R. Jeffries, *Agile Modeling: Effective Practices for Extreme Programming and the Unified Process*, Wiley, Boston, Mar. 2002.

[Balasubramanian 07] K. Balasubramanian, J. Balasubramanian, J. Parsons, A. Gokhale, D.. Schmidt, A Platform-Independent Component Modeling Language for Distributed Real-time and Embedded Systems, *Journal of Computer and System Sciences*, 73 (2), Mar. 2007, 171-185

[Bahill 05] A. Bahill and S. Henderson, Requirements development, verification, and validation exhibited in famous failures, *Systems Engineering*, 8 (2), 2005, 1–14.

[Bass 03] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice* (Second Edition), Addison-Wesley, New York, 2003.

[Battin 01] R. Battin, Crocker, R., and Kreidler, J. Leveraging resources in global software development, *IEEE Software*, 18 (2), 2001, 70-77.

[Beck 04] K. Beck and Andres, C., *Extreme Programming Explained: Embrace Change (2nd Edition)*. Addison-Wesley Professional, Nov. 2004.

[Berenbach 09] B. Berenbach, D. Paulish, J. Kazmeier and A. Rudorfer, *Software & Systems Requirements Engineering in Practice*, McGraw-Hill, 2009.

[Boehm 81] B. Boehm, *Software Engineering Economics*, Prentice Hall, New Jersey, 1981.

[Boehm 88] B. Boehm and C. Papaccio, Understanding and Controlling Software Costs*, IEEE Transactions of Software Engineering*, Oct. 1988.

[Boehm 94] B. Boehm, P. Bose, E. Horowitz, and M. Lee, Software Requirements as Negotiated Win Conditions, in *Proceedings of the First International Conference on Requirements Engineering*, Colorado Springs, Colorado, Apr. 18-22, 1994, 74-83.

[Boehm 02] B. Boehm, and T. DeMarco, The Agile Methods Fray, *IEEE Computer*, Jun. 2002, 90-92.

[Bradner 02] E. Bradner, and Mark, G., Why distance matters: effects on cooperation, persuasion and deception. *Proceedings of the 2002 ACM conference on Computer Supported Cooperative Work.* New Orleans, 2002, 226 - 235.

[Brooks 95] F. Brooks, *The Mythical Man–Month: Essays on Software Engineering*, 2nd Edition, Addison–Wesley, 1995.

[Brooks 87] F. Brooks, No Silver Bullet: Essence and Accidents of Software Engineering, *IEEE Computer*, Apr. 1987, 10-19.

[Clements 01] P. Clements and Northrop, L., *Software Product Lines: Practices and Patterns*, 3rd ed. Addison-Wesley Professional, Aug. 2001.

[Cockburn 00] A. Cockburn, *Writing Effective Use Cases*, Reading, MA: Addison-Wesley, 2000.

[Damian 07] D. Damian, Stakeholders in global requirements engineering: Lessons learned from practice, *IEEE Software*, 2007, 21-27.

[Davis 88] A. Davis, A Taxonomy for the Early Stages of the Software Development Life Cycle, *Journal of Systems and Software*, Sep. 1988, 297-311.

[Davis 93] A. Davis, *Software Requirements (Revised): Objects, Functions, and States*, Prentice Hall, New Jersey, 1993.

[DeMarco 78] T. DeMarco, *Structured Analysis and System Specification*, Prentice Hall, New Jersey, 1978.

[Dorfman 90] M. Dorfman and R. Thayer, eds, *Standards, Guidelines, and Examples on System and Software Requirements Engineering*, IEEE Computer Society Press, Los Alamitos, California, 1990.

[Fairley 97] R. Fairley and R. Thayer, The Concept of Operations Document: The Bridge from Operational Requirements to Technical Specifications, in *Software Engineering*, R.H. Thayer and M. Dorfman (eds.), IEEE Computer Society Press, 1997.

[Faulk 92] S. Faulk, J. Brackett, P. Ward, and J. Kirby, Jr., The Core Method for Real–Time Requirements, *IEEE Software*, Vol. 9, No. 5, Sep. 1992.

[Faulk 93] S. Faulk, L. Finneran, J. Kirby Jr., and A. Moini, *Consortium Requirements Engineering Guidebook*, Version 1.0, SPC–92060–CMC, Software Productivity Consortium, Herndon, Virginia, 1993.

[Faulk 01] S. Faulk, Product-Line Requirements Specification (PRS): an Approach and Case Study, *Proceedings, Fifth IEEE International Symposium on Requirements Engineering*, Toronto, Canada, Aug. 27-31, 2001, 48-55.

[Faulk 10] S. Faulk and M. Young, Sharing What We Know About Software Engineering, *Proceedings: Foundations of Software Engineering*, FOSER 10, Santa Fe, NM, Nov. 2010.

[Fickas 95] S. Fickas and M. Feather, Requirements Monitoring in Dynamic Environments, *Proceedings, Second IEEE International Symposium on Requirements Engineering*, York, England, Mar. 1995, 140-150.

[GAO 79] U.S. General Accounting Office, *Contracting for Computer Software Development-Serious Problems Require Management Attention to Avoid Wasting Additional Millions*, Report FGMSD–80–4, Nov. 1979.

[GAO 92] U.S. General Accounting Office, *Mission Critical Systems: Defense Attempting to Address Major Software Challenges*, GAO/IMTEC–93–13, Dec. 1992.

[GAO 08] U.S. General Accounting Office, Significant Problems of Critical Automation Program Contribute to Risks Facing 2010 Census, GAO-08-550T, Mar., 2008.

[GAO 10] U.S. General Accounting Office, Defense Acquisitions: Assessments of Selected Weapon Programs, GAO-10-388SP, Mar. 2010.

[Gargantini 99] A. Gargantini and C. Heitmeyer, Using Model Checking to Generate Tests from Requirements Specifications, *Proc., Joint 7th European Software Engineering Conf. and 7th ACM SIGSOFT Intern. Symp. on Foundations of Software Eng.* (ESEC/FSE99), Toulouse, FR, Sept. 6-10, 1999.

[Gause 89] D. Gause and G. Weinberg, *Exploring Requirements: Quality before Design*, Dorset House, 1989.

[Gane 79] C. Gane and T. Sarson, *Structured Systems Analysis*, Prentice Hall, New Jersey, 1979.

[Heitmeyer 95a] C. Heitmeyer, B. Labaw, and D. Kiskis, Consistency Checking of SCR–Style Requirements Specifications, in *Proceedings, IEEE International Symposium on Requirements Engineering*, Mar. 1995.

[Heitmeyer 95b] C. Heitmeyer, R. Jeffords, and B. Labaw. Tools for Analyzing SCR–Style Requirements Specifications: A Formal Foundation, NRL Technical Report NRL-7499, U.S. Naval Research Laboratory, Washington, DC, 1995.

[Heitmeyer 98] C. Heitmeyer, J. Kirby, B. Labaw, M. Archer, and R. Bharadwaj, Using Abstraction and Model Checking to Detect Safety Violations in Requirements Specifications, *IEEE Transactions on Software Engineering*, 24, (11), November 1998.

[Heninger 80] K. Heninger, Specifying Software Requirements for Complex Systems: New Techniques and Their Application, *IEEE Transactions on Software Engineering*, 6 (1), Jan. 1980.

[Herbsleb 07] J. Herbsleb, Global software engineering: The future of socio-technical coordination, *International Conference on Software Engineering 2007 Future of Software Engineering*, IEEE Computer Society, 2007, 188-198.

[Hester 81] S. Hester, D. Parnas, and D. Utter, Using Documentation as a Software Design Medium, *Bell System Technical Journal*, 60 (8), Oct. 1981, 1941-1977.

[Jackson 94] M. Jackson, Problems, Methods, and Specialization, *IEEE Software*, Nov. 1994, 57-62.

[Jacobsen 92] I. Jacobson, Christerson, M., Jonsson, P., and Övergaard, G., *Object-Oriented Software Engineering: A Use Case Driven Approach*, Reading, MA: Addison-Wesley, 1992.

[Jeffords 09] R. Jeffords, C. Heitmeyer, M. Archer, and E. Leonard, A Formal Method for Developing Provably Correct Fault-Tolerant Systems Using Partial Refinement and Composition, *Proceedings, Formal Methods, Second World Congress (FM 2009),* Eindhoven, The Netherlands, November 2-6, 2009. 173-189.

[Karsai 02] G. Karsai, S. Neema, B. Abbott, and D. Sharp, A modeling language and its supporting tools for avionics systems, *Proceedings of the 21st Digital Avionics Systems Conference*, Oct. 2002, 6A3 1-13.

[Kelly 08] S. Kelly and Juha-Pekka Tolvanen, *Domain-Specific Modeling: Enabling Full Code Generation*, Wiley-IEEE Computer Society Press, Mar. 2008.

[Komi 05] S. Komi-Sirvio and M. Tihinen, Lessons learned by participants of distributed software development. *Knowledge and Process Management*, 2005, 108-122.

[Kruchten 99] P. Kruchten, *The Rational Unified Process: An Introduction*, Reading, MA: Addison-Wesley, 1999.

[Lam 97] W. Lam, Achieving Requirements Reuse: A Domain Specific Approach from Avionics, *Journal of Systems and Software*, 38 (3], Sept. 1997, 197-209.

[Lamsweerde 98] A. van Lamsweerde, R. Darimont and E. Letier, Managing Conflicts in Goal-Driven Requirements Engineering, *IEEE Trans. on Software. Engineering*, Nov. 1998.

[Lamsweerde 09 A. van Lamsweerde, *Requirements Engineering: From System Goals to UML Models to Software Specifications*, Wiley, Mar. 2009.

[Lauesen 02] S. Lauesen, *Software Requirements: Styles and Techniques*, Addison-Wesley Professional, London, 2002.

[Lee 99] J. Leand and N. Xue, Analyzing User Requirements by Use Cases: A Goal-Driven Ap-

proach, *IEEE Software,* 16 (4): Jul/Aug., 1999, 92-101.

[Lutz 93] R. Lutz, Analyzing Software Requirements Errors in Safety–Critical Embedded Systems, *Proceedings*, *IEEE International Symposium on Requirements Engineering*, Jan. 4-6, 1993, 126-133.

[Middleton 05] P. Middleton and J. Sutton, *Lean Software Strategies: Proven Techniques for Managers and Developers*, Productivity Press, NY, 2005.

[Mockus 01] A. Mockus and J. Herbsleb, Challenges of global software development, *Proceedings of the Seventh International Software Metrics Symposium*, 2001.

[Nuseibeh 00] B. Nuseibeh and S. Easterbrook. Requirements engineering: a roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, New York, NY, 2000. ACM, 35–46.

[NASA 05] S. Cavanaugh, A. Wilhite, Systems Engineering Cost/Risk Analysis Capability Roadmap Progress Review, Apr. 6, 2005.

[Olson 2000] G. Olson, and Olson, J.. Distance matters, *Human-Computer iIteraction, 15* (2), 2000, 139-178.

[OMG 05] Object Management Group, Introduction to OMG's Unified Modeling Language (UML), http://www.omg.org/gettingstarted/what_is_uml.htm, 2005

[Parnas 72] D. Parnas, On the Criteria to be Used in Decomposing Systems into Modules, *Communications of the ACM*, 15 (12), December 1972, 1053-1058.

[Parnas 76] D.Parnas, On the design and development of program families. I*EEE Transactions on Software Eng*ineering, 2 (l,) Mar. 1976, l-9.

[Parnas 85a] D. Parnas, and D. Weiss, Active Design Reviews: Principles and Practices, in *Proceedings of the Eighth International Conference on software Engineering*, London, England, Aug. 1985.

[Parnas 86] D. Parnas, and P. Clements, A Rational Design Process: How and Why to Fake It, *IEEE Transactions on Software Engineering*, 12 (2), Feb. 1986, 251-257.

[Parnas 89] D. Parnas, *Education for Computing Professionals*, Technical Report 89-247, Department of Computing and Information Science, Queens University, Kingston, Ontario, 1989.

[Parnas 91] D., Parnas and J. Madey, *Functional Documentation for Computer Systems Engineering (Version 2)*, CRL Report No. 237, McMaster University, Hamilton, Ontario, Canada, Sept. 1991.

[Pohl 05] K. Pohl, G. Böckle, and F. J. van der Linden, *Software Product Line Engineering: Foundations, Principles and Techniques*, 1st ed. Springer, Sept. 2005.

[Prieto-Diaz 97] R. Prieto-Diaz, M. Lubars, M. Carrio, DSSR: Support for Domain Specific Software Requirements, U.S. Army Communications-Electronics Command, Apr. 1994.

[Rising 02] L. Rising and Janoff, N. S., The Scrum software development process for small teams, *Software, IEEE*, 17 (4), pp. 26-32, Aug. 2002..

[Robinson 89] W. Robinson, Integrating Multiple Specifications Using Domain Goals, *Proceedings, 5th International Workshop on Software Specification and Design*, IEEE, 1989, 219-225.

[Robinson 05] W. Robinson, Implementing Rule-based Monitors within a Framework for Continuous Requirements Monitoring, in *Hawaii International Conference On System Sciences (HICSS'05)*, Big Island, Hawaii, USA, 2005.

[Ross 77] D. Ross and K. Schoman Jr., Structured Analysis for Requirements Definitions, IEEE *Transactions on Software Engineering*, 3 (1), Jan. 1977, 6-15.

[Rothamel 06] T. Rothamel, C. Heitmeyer, Y. Liu, and E. Leonard, Generating Optimized Code from SCR Specifications, in *Proceedings, ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES 2006),* Ottawa, Canada, June 14-16, 2006.

[Schneider 98] G. Schneider, and J. P. Winters, *Applying Use Cases: A Practical Guide*. Reading, MA: Addison-Wesley, 1998.

[SEI 06] *CMMI for Development, Version 1.2,* CMMI-DEV, Carnegie Mellon University Software Engineering Institute, Aug., 2006.

[Stephens 01] M. Stephens and D. Rosenberg, *Extreme Programming Refactored: The Case Against XP*. APRESS, Sep. 2003.

[Sutcliffe 06] A. Sutcliffe, S. Fickas, M. Sohlberg, *Journal of Requirements Engineering*, 11 (3), Jun. 2006.

[Svahnberg 05] M. Svahnberg, J. van Gurp, and J. Bosch, A taxonomy of variability realization techniques, *Software: Practice and Experience*, 35 (8), pp. 705-754, 2005.

[Svoboda 90] C. Svoboda, Structured Analysis, in *Tutorial: System and Software Requirements Engineering*, R. Thayer and M. Dorfman, eds., IEEE Computer Society Press, Los Alamitos, California, 1990, 218-237.

[Thayer 90] R. Thayer and M. Dorfman, eds, *Tutorial: System and Software Requirements Engineering,* IEEE Computer Society Press, Los Alamitos, California, 1990.

[Weigers 03] K. Weigers, *Software Requirements*, Microsoft Press, 2003.

[Weiss 99] D. Weiss and C. T. R. Lai, *Software Product-Line Engineering: a Family-Based Software Development Process*, Addison Wesley, 1999.

[Winter 01] V. Winter, R. Berg, and J. Ringland, Bay area rapid transit district advance automated train control system case study description, in *High Integrity Software*, Kluwer Academic Publishers, Norwell, MA, 2001.

[Young 06] R. Young, *Project Requirements: a Guide to Best Practices*, Management Concepts, 2006.

[Yourdon 89] E. Yourdon, *Modern Structured Analysis*, Yourdon Press/Prentice Hall, 1989.

In publication, do not distribute