SOFTWARE REQUIREMENTS: ARE THEY REALLY A PROBLEM?

T. E. Bell and T. A. Thayer

TRW Defense and Space Systems Group
Redondo Beach, California

Keywords and Phrases

Ballistic Missile Defense
Requirements
Requirements Problems
Software Engineering
Software Requirements
Software Requirements Engineering
Software Requirements Problems
SREM
SREP

Abstract

Do requirements arise naturally from an obvious need, or do they come about only through diligent effort -- and even then contain problems? Data on two very different types of software requirements were analyzed to determine what kinds of problems occur and whether these problems are important. The results are dramatic: software requirements are important, and their problems are surprisingly similar across projects. New software engineering techniques are clearly needed to improve both the development and statement of requirements.

I. Introduction

Identifying the cause of a problem in a software system is often very easy -- if the cause is a problem in code. Typically, identified coding problems result in clearly incorrect answers or in abnormal terminations of the software. Similarly, problems in a software system caused by deficiencies in design are often easy to identify from unexpected software operation or from extreme difficulty in maintaining and modifying the system. Problems in the system caused by deficiencies in software requirements, on the other hand, are often not identified at all, or are thought to be caused by bad design or limitations of computing technology. If there are problems in developing requirements, however, the software system meeting those requirements will clearly not be effective in solving the basic need, even if the causes of the problems are incorrectly identified.

The Ballistic Missile Defense Advanced Technology Center (BMDATC) is sponsoring an integrated software development research program [1] to improve the techniques for developing correct, reliable BMD software. Reflecting the critical importance of requirements in the development process, the Software Requirements Engineering Program (SREP) has been undertaken as a part of this integrated program by TRW Defense and Space Systems Group* to examine and improve the quality of requirements.

One of the first efforts in SREP has been to characterize the problems with requirements so that techniques can be developed to improve the situation. Instead of pursuing philosophical discussions about what the problems might be, we have undertaken empirical studies to determine what the problems actually are. A limitation on the number of Ballistic Missile Defense (BMD) systems currently being developed (there is only one) has led us to examine both BMD and more common data processing systems to ensure that our results are characteristic of software requirements in general, rather than just one particular project.

This paper reports on initial results that have set much of the direction pursued in the Software Requirements Engineering Methodology [2], the Requirements Statement Language [3], and the Requirements Engineering and Validation System [4]. The initial efforts were oriented to determine the magnitude and characteristics of the problems, and to indicate what types of techniques could correct the problems. The empirical study of software problems is continuing in parallel with technology development so that the technology can be refined and tested for effectiveness in solving the identified problems.

II. What are Software Requirements?

One school of thought maintains that software requirements arise naturally, and that they are correct by definition. If these requirements merely state a basic need (e.g., "do payroll"), then that's all that is needed. On the other hand, if the requirements state each subroutine's detailed characteristics, then those are the required characteristics, and the implementer should not question them.

Adherents to this school of thought have grown fewer and fewer as the software industry has gathered experience with this approach to developing software. When every requirement ranging in detail from needs statements to subroutine specifications is considered in the same way, the resulting systems tend to be seriously deficient. If coding personnel are assigned the task of implementing a system with only a needs statement, the critical phase of software design will likely be skipped -- with disasterous results. On the other end of the scale, if detailed subroutine specifications are accepted without ever having been

*Under Contract DASG60-75-C-0022

examined for correctness, the resulting software will probably fail to complete execution normally, much less produce correct results.

The evolution of approaches for the development of software systems has generally paralleled the evolution of ideas for the implementation of code. Over the last ten years more structure and discipline have been adopted, and practicioneers have concluded that a top-down approach is superior to the bottom-up approach of the past. The Military Standard set MIL-STD 490/483 recognized this newer approach by specifying a system requirements document, a "design-to" requirements document that is created in response to the system requirements, and then a "code-to" requirements document for each software module in the design. Each of these is at a lower level of detail than the former, so the system developers are led through a top-down process. The same top-down approach to a series of requirements statements is explained, without the specialized military jargon, in an excellent paper by Royce [5]; he introduced the concept of the "waterfall" of development activities. In this approach software is developed in the disciplined sequence of activities shown in Figure 1.

Each of the documents in the early phases of the waterfall can be considered as stating a set of requirements. At each level a set of requirements serve as the input and a design is produced as output. This design then becomes the requirements set for the designer at the next level down.

With so many levels of requirements documents, and with so few software projects mapping nicely into the scheme, we must be more specific about what we mean by the term "software requirements" as used in our studies. We do not mean all the various levels of requirements, but only a single one, one that can usually be identified in large software development projects that have ended successfully. At this level the software requirements describe functions that the software must perform, but not how they must be implemented. For example, they might contain a requirement that a BMD system identify erroneous RADAR returns that have any of five specific characteristics. However, the software to meet this requirement might be spread through twelve subroutines using any of a large number of identification techniques.

In weapon systems like BMD, these software requirements lie at lower (more detailed) levels of detail than the general, summary Data Processing Subsystem Performance Requirements (DPSPR); DPSPR requirements may fail even to mention error detection. The software requirements are also at a higher level of detail (less detailed) than the "code to" requirements that describe each subroutine and are described in MIL-STD 490 as being in a Type C-5 document. In information systems (not part of weapon systems) the same "code-to" (Type C-5) document should exist, but no DPSPR exists. Instead, a statement of a basic need is documented. In MIL-STD 490 terminology, this document is a Type A requirement.

Each of the "software requirements" that we are discussing lies at, or below, the level of a Type A requirement document but at, or above, the level of a Type B-5 requirement document of 490. Ideally, each of the requirements has been derived from the equivalent of a DPSPR or a Type A document (and may possess deficiencies introduced during the derivation) and will be used to generate a Type B-5 or a Type C-5 document in the design phase (when the deficiencies begin to be apparent). However, the nice situation of having a complete set of requirements, all at precisely one level, seldom occurs because requirements engineers find less difficulty in stating a specific design, or leaving statements very general, in certain areas [6]. Therefore, our empirical studies have concentrated on the documents that have their major emphasis on the software requirements that we have defined, even if they have some requirements at different levels of detail.

III. Empirical Data

The remembrances of engineers who participated on a requirements engineering project are usually faulty. Memories are biased by personal conflicts and occurrences from long after the software is implemented. An empirical study of requirements must therefore either be based on information documented during the project or be based on data collected during the project through observation. We have used data from two projects in our analyses; for one project we used only documented data while in the other we used both documented data and observation.

The pieces of paper recording software requirements deficiencies are typically called problem reports and are written either to record the results of reviews or to request changes in the requirements during the software development project. The points indicated in Figure 2 during a software development project's life are the times when most of the problem reports involving software requirements are generated; during reviews, during design, and during implementation. The same software requirements deficiency might be identified at any of several points in the project's life, so it may be documented on a design problem report, a software problem report, or a test problem report.

Problem reports are only written to document significant deficiencies -- those that could cause major or catastrophic problems in the ultimate system. The effort in writing a problem report is actually quite small, but the psychological cost of putting a criticism in written form, and then being willing to defind it, is high enough that irrelevant and minor problems (e.g., spelling errors and indentation) are ignored.

Problem reports do not generally document the correction of deficiencies (except for suggested revisions), but they do record the symptoms of the problem. Therefore, data are available for frequency analysis of symptoms, but not usually for analysis about the types of correction to the requirement statement. In some cases the problem report's author is asked to assign a classification to the report so that a simple frequency analysis can be performed. However, previous studies [7] have indicated that the authors of problem reports experience significant difficulty in assigning a classification, so problem reports often do not record the author's impressions.

Data Limitations. Data used in our study about software requirements problems largely come from problem reports because of their relatively objective nature. However, data from these reports have limitations that need to be recognized in interpreting our results and in understanding why previous studies have not dealt with the topic. These limitations involve the generally undisiplined manner in which the software industry undertakes development projects, the need for interpretation of textual material, and the lack of information about the causes of the problems.

The typical software development project is not managed with explicit, baselined requirements or with documented designs. Programmers make decisions about the system's requirements and design as they feel appropriate, and then they depend on informal communications to let other programmers know about their decisions. In this environment the quality of data about requirements problems is very poor (if it exists at all). Therefore, very little analysis has been done on the problems with software requirements, and the image may have been created that no real problems exist.

Only in a very disciplined environment do the data needed for analysis come to exist. In this environment software requirements engineers and design engineers tend to be much more careful than they would in an undisciplined situation. Fewer deficiencies will exist, and these will be identified earlier in the project. Therefore, our results probably describe a requirements situation less troublesome than those encountered by software development projects using conventional, undisciplined approaches.

Even in a disciplined environment, authors of problem reports find that they are unable to interpret their problems into a general classification scheme. This is sometimes due to ambiguity in categories, but it appears more often to be due to difficulty in abstracting from a specific problem statement that "something is wrong here". When the author of a problem report finally determines that the requirements engineer produced a deficient specification, his mind is concentrated on substantive issues of content rather than on classifying the problem. Therefore, our technique has been to examine the textual description of the problem to determine which category is most appropriate. The interpretation probably introduces errors, but its superiority has been obvious over depending only on authors of problem reports.

The empirical data about deficiencies in software requirements seldom include the reason that the deficiency has come to be there; by the time it is discovered, the project has progressed and information about cause can not usually be reconstructed. Therefore, empirical data about deficiencies in software requirements typically are only related to the symptoms and their detection rather than to the causes and their importance.

IV. Small System Case

Simply interpreting data from a software development project might be a dangerous technique for determining the nature and magnitude of software requirements problems. Without knowing the characteristics of the development project, inappropriate conclusions could easily be derived. Our first case involves a situation in which we could control the environment, observe the outcome directly, and easily understand the system decisions because of the relatively small size (implementation would cost about $100,000). This case was a project in a graduate software engineering class at UCLA, and it assigned the students to write a set of software requirements and a preliminary system design for a Student Employment Information System (SEIS).

Description. Dr. Barry W. Boehm divided his class into two teams, the requirements team and the design team. Both teams received the "needs statement" in the form of the memorandum of Figure 3, and the requirements team produced a functional specification (the requirements) after about a month. The design team then used this document to produce a preliminary design after about one more month.

The students met informally between class sessions while generating the requirements and the design. In addition, formal meetings occurred between the two teams during the design phase in order to resolve questions that the design team raised about the requirements. The members of both teams were urged to record the problems they detected in the requirements, but no controls were imposed to ensure complete documentation. Observation by the instructor during the formal meetings augmented data that were collected from documentation by the students.

The authors of this paper provided Boehm with a form for the students on the design team to use in documenting the requirements problems. The form (the Requirements Problem Report, RPR) is shown in Figure 4. Note that the students were urged to provide some classification of each problem by inclusion of eight categories under the title "Problem Description". Having a form to fill out eased the documentation job and helped to provide more consistent data.

The original statement of the need was contained in a single paragraph of the memorandum of Figure 3. This paragraph grew into 48 pages of text and charts for the requirements specification. The design team then produced a document with 63 pages describing two alternate designs, the proposed implementation plan, and the "boiler-plate" that would normally accompany a proposal. One of the proposed designs was a manual system, and the other one included a batch data processing system whose projected cost was inconsistent with the independent cost estimate generated by the requirements team. As frequently occurs, the designers projected that the system's implementation and its execution would be less costly (by about a factor of two) than the independent cost estimate projected by the "customer" team.

Requirements Problems. The requirements specification gave several senior TRW personnel the general impression of being superior to most typical requirement documents. It appeared to be more specific, direct, and complete than the commonly-reviewed requirement specification.

Number of Problems. In spite of the apparent quality of the requirement, the design team completed twenty (20) RPR forms. A wide variety of problems were documented, from costing assumptions, to the need for specific requirements, to identification of specific designs as requirements. As an example of the last, the requirement specified, "The SEIS Data Base shall consist of three files; one each for student data, employer data, and historical data". This is an overly constraining requirement since many possible designs could be implemented that would contain these three types of data; three separate physical files are not the only solution. We put description Number 6 ("Better Design Possible") on the RPR form specifically for the problems like this.

The total of 20 problems identified in the 48 pages of requirements is an under-representation of the number of the problems. Six of the twenty RPRs actually documented two problems each, and three RPRs documented three problems each. Therefore, in all, 32 problems were identified and documented.

Even this total is optimistic. In the fast give-and-take of the formal meetings between the two teams, identifying and recording each unique problem proved infeasible. However, Boehm observed the proceedings as an objective, experienced rater. His estimate [8] is that the number of problems exceeded 50 -- more than one per page of double-spaced material. Examination of his notes on the requirements specifications indicates that this estimate is probably low since several times that number are documented. It would be difficult, though, to determine an exact number of problems at this time. This involves determining retrospectively how many of the noted problems are multiple problems, how many are duplicate instances of the same problem, and how many other problems are present but not noted.

The design team examined the requirements specification only from the viewpoint of producing a design; they were not doing an explicit requirements review. They only hypothesized a design since the class schedule precluded any attempt to implement the system. Therefore, the requirements were neither reviewed with maximum care nor were they given the ultimate test of comparison with user needs through the implementation of the system. Therefore, even the total of more than 50 problems is certainly an underestimate of the total number of problems in the document.

Characteristics of Problems. The problems documented on RPR forms were the ones which were identified when the design team members were clear about what to document and which were also clearly important to the design team's work. Analysis of the characteristics of less important problems would be speculation, so our analysis is concerned only with documented problems. We performed the computations using both the classifications of problems provided by the design team and a separate classification that we generated. Table 1 presents the frequency of problem types computed from each of these data bases.

The design team produced 20 RPR forms, but two of these did not classify the problem into any of the eight categories. On the other hand, five of the RPR's (all of which documented two or three problems on each form) included classification into two categories. Therefore, 23 classifications were produced by the design team on 20 RPR's involving the 32 documented problems.

Our classifications (made by this paper's authors) were made separately for each problem, whether it was on an RPR form by itself or with another problem. Therefore, our classifications total 32, the number of problems identified on the RPR forms.

In addition to the difference in total number of problems classified, the distribution of problems is different for the design team's classification than the distribution from our classification. Most of this difference clearly resulted from different definitions. For example, we classified some problems as one requirement being inconsistent with another requirement (Number 5), but the design team always classified this type of problem as being ambiguous. The design team's logic appeared to be that the inconsistency resulted in the entire set of requirements being ambiguous about that point. Similarly, problems that we felt were design-constraining (Number 6) were sometimes classified by the design team as being due to an incorrect requirement (Number 4) or were not classified at all. The design team appeared to feel that some of these problems should be classified as incorrect requirements because alternatives had been rejected that were clearly superior. This rejection was incorrect, so category Number 4 was assigned by the design team.

The two instances where we classified the problems as "other" both involved identification of incomplete requirements by the design team. One of these instances was classified by the design team as simply a case of inadequate information (since no information was provided about whether some other requirement might have been intended instead). The other instance was not classified, even though it was clearly another case of an incomplete requirement.

We refrained from including a category titled "incomplete" in the fear that the RPR's would predominately describe places where the requirements could have included more detail. The category "other" or the category "more information needed" could be used for all the cases where an incomplete requirement was given; this did not occur. However, this was not because the requirements were complete in every place except for the two particulars noted above.

We examined the requirements document to determine whether obviously incomplete requirements existed. We found two types of statements that clearly indicated the presence of incomplete requirements. The first type of statement was included in requirements that the system must do functions "such as" a list, or that the system must be expandable to do functions "including but not limited to" a list. The second type of statement occurred at the end of lists of data and was "...etc". Both of these types of requirements are open-end since they leave the requirements specifier the option of legitimately demanding the inclusion of undocumented functions and data after system design and implementation.

We suspect that the design team failed to identify and/or document these situations because its members did not produce a detailed design or implementation in which these problems would have required resolution. The problems detected by the design team were therefore only the ones important during preliminary design. The distribution of problems, and perhaps the number of problems, may be different for projects that have, or will, actually produce operational code.

The data on the SEIS case clearly indicate that ambiguous and design-constraining requirements are the prominent classes of problems. In addition, the total number of problems seems to be in excess of one (and probably in excess of two) times the number of pages in the requirements document. This metric (number of problems per page) is clearly suspect, but the magnitude is alarming. We need a look at an actual, large software development project to determine whether requirements problems are really so frequent and of these types.

V. Large Systems

It is the large software system development project that represents a challenge to the data analyst. On one hand it is difficult to control the development environment during the requirements specification and analysis phases due to the number of participants and long time involved. And, it is difficult to accurately pinpoint causes for observed trends in the data; the variables are numerous and controls from comparable projects are virtually nonexistant because large systems tend to be unique, one-time products. On the other hand the large systems have a potential for producing tremendous volumes of useful data and for providing sufficient resources for analyzing these data in light of project characteristics. Our second case, the System Technology Program, is just such a project. It is a large (1 million machine instructions) real-time BMD system being developed with a top-down approach. This project is particularly attractive from the standpoint of requirements data analysis because it is truly a state-of-the-art project with a software requirements specification containing over eight thousand uniquely identifiable requirements paragraphs. These requirements are subject to change due to external reasons, as the perceived threat changes, and due to internal reasons, as the developers learn more about the problems to be solved and the actual needs of the system. It is appropriate to say that, while the software itself is state-of-the-art, the problems being encountered in specifying, analyzing, and satisfying software requirements are also state-of-the-art.

Description. Our principal source of data on the second case is the requirements review, which is conducted whenever a new software requirements specification, or B-5 document, is issued. However, reports documenting requirement-related problems or needed change can be generated throughout the development cycle, not just during appropriate requirement reviews. In fact, such information also comes from the design phase and as late as the formal testing phase. As was noted in Figure 2, a series of problem reports is generated throughout the development cycle. While the principal activity may be software design or testing, documented problems at any point in the development cycle may identify the source of the problem as any previous activity, e.g., a requirement error documented on a software problem report written during formal testing. Once a problem is documented it is processed by a configuration control board (CCB) which verifies that the problem is real and assigns some corrective action. Each documented problem is also tracked by the CCB to guarantee that all problems are solved, the requirements and/or design are updated in a controlled manner, and that all paperwork is eventually closed.

Analysis of requirements problem reports was performed by individuals acquainted with the characteristics of STP development techniques, and classification of software errors traceable to a source in requirements was done by the individual developers who corrected the errors.

STP requirements problem reports used in this report arose largely from two major requirements reviews conducted during the development cycle. The first review, conducted in 1973, serves as an indicator of early experience with requirements specification and analysis. A later review, conducted in 1975, is indicative of current experience and shows an increased awareness of software system needs on the part of the specifiers and reviewers of requirements. These data sets were treated separately so that differences attributable to learning might be recognized. Of course, we hope eventually to have a third set of data indicating experience with the system in operation since the frequency and magnitude of problems would probably be different at that time than for the earlier experiences.

Categories of Requirements Problem. Table 2 presents a list of typical problem categories generated from STP requirements review problem reports*, and used in categorizing requirements according to type. Note that each generic problem category is broken down further into more detailed categories, e.g., the "accuracy criteria missing" category is a detailed category under the general "missing/incomplete/inadequate" category. The numbering scheme in Table 2 reflects this break-down and is employed in our data collection and analysis to ease the job and reduce ambiguity.

Definitions for most of the problem categories in Table 2 are self-explanatory; however, several are peculiar to systems for which controlled change to the requirements set may not only be a necessary but desirable reality. These categories are 1-000 and 6-000 which register acceptable changes to previously existing requirements sets, and 2-000 which indicates that the requirement is not within the technical or contractual boundaries of the software system.

Quantity of Problems. One of the most obvious findings in our studies of requirements problems was the sheer volume of problems encountered. This alone was enough reason to convince even the casual observer that requirements represent a significant problem in software development. In the two major requirements reviews mentioned previously, a total of 722 problem reports documenting 972 uniquely identifiable problems were written. This was in a review of approximately 8248 requirements and support paragraphs in the 2500 page software requirements specification.

A less obvious finding than the raw occurrences of requirements problems, but of far greater consequence, is the criticality of the individual problems. In some instances failure to identify and fix requirements problems could result in ultimate non-responsiveness of the software system to a known threat, i.e., mission failure. Such a problem is specification of realizable timing requirements for real-time software systems. The designed solution for meeting a timing requirement (normal stimulus to normal response) may adequately meet the stated requirement. However, "unusual" stimuli may occur quite frequently, and ignoring them can cause average response to be bad enough that system performance is unacceptable. The designers of the software modules can hardly be expected to consider the system effects of sequences of stimuli unless the requirements mention them, but a particular, unfortunate sequence of stimuli could increase response time far above normal -- with unacceptable system effects. These problems must be resolved in the requirements to avoid such consequences; designers are concerned with other issues.

Characteristics of Problems. Translating the general error categories given in Table 2 into histogram form for the STP case, we get Figures 5 and 6. Again, the separation of early (1973) STP experience from current (1975) experience is intentional. In Figure 7 the SEIS data, classified in a fashion compatible with STP data, are presented for comparison purposes.

It will be noted that in all three cases the "incorrect" category occurred more frequently than other general categories, exceeding 30 percent in each case. This finding did not support a suspicion we had that early requirements reviews would uncover higher percentages of "missing", "incomplete", "inadequate", or "unclear" type errors and that in subsequent reviews the "incorrect" category would dominate. To some extent this hypothesis may be correct since the "incorrect" category increased percentagewise while the other categories decreased between the two STP reviews.

The "requirement unclear" category is of particu-

*These reports are similar in format to the RPR except that problem description categories are not part of the form.

lar interest because it is this category that criticizes the terminology and understandability of the requirement. In this category the requirement may not be in error; however, its statement leaves confusion or room for multiple interpretations on the part of the reader/reviewer. Note that, between the 1973 and 1975 reviews of STP, the occurrences of this category dropped from 25.4 percent to 9.3 percent, a result one might expect as a result of increased experience. Note, too, a similar high percentage of the "unclear" category for the SEIS case, also a result one might expect from a first review of requirements written by someone other than the reviewer.

One result which was not expected, nor can we explain it at this point, is the near constant percentage of errors of inconsistency and incompatability for each of the cases. This percentage ranged from a low of 9.1 percent for the recent STP review to a high of 10.0 percent for the SEIS review. We will be looking for a similar trend in our analysis of requirements data taken from another large software project.

Another fairly obvious result of our study is that the search for requirements problems should be a continual one. In a review of software problem reports documented during formal validation testing, i.e., problems found subsequent to the design and coding phases, it was discovered that errors could be traced to an origin in software requirements. Of course, design, coding, and software maintenance activities are also sources of error. However, one study [9] of the source of software errors found in the code indicates that the percentage attributable to requirements and not discovered until testing may be as high as 12 percent for large, complex software systems. The importance of continually reviewing requirements for their impact on design solutions is obvious, especially in a top-down environment where each iteration through the development cycle affords the opportunity to deal with changing or incorrect requirements, and to factor these changes into the evolving design.

Another important facet of the requirements problem can be seen by looking at errors documented during testing. Analysis of error data collected from four large software projects [9] showed that the most common software error type, representing between 8.0 and 17.8 percent of all problems reported during testing, were in the missing logic category. That is, some logic needed as part of a successful design solution to software requirements was missing. Although it was virtually impossible to retrospectively determine how many of these errors might have been precluded by more completely specified requirements, the requirements problem category for "missing", "incomplete", and "inadequate" problems is believed to be directly related to problems which eventually turn up in the code.

A final finding in analysis of the "incorrect" category of requirements problems was in the nature of the detailed problem statements. Incorrectness in the results of the early review of STP requirements related primarily to functional correctness. That is, criticism centered on what the software was to do. In the later review, problems of incorrectness related primarily to analytical correctness or how well the software must perform its functions.

VI. Conclusions

Our empirical data clearly show that the rumored "requirements problems" are a reality. Information needed for design and implementation of both small and large systems is often incorrect, ambiguous, inconsistent, or simply missing. The requirements for a system, in enough detail for its development, do not arise naturally. Instead, they need to be engineered and have continuing review and revision.

The relative frequencies of various types of requirements problems were surprisingly similar between radically different kinds of software projects. Of course, problems with analytical requirements occurred only on the one project with analytical requirements, so we could not compare relative frequencies of this type across projects. We are currently examining another project with analytical requirements that has been managed with a disciplined approach. From a comparison using these new data we will be able to test our hypothesis that the relative frequency of analytical problems is nearly constant across projects.

The types of problems detected in requirements changed during the life of a software development project. The system developers often determined a requirement deficiency only when they attempted to meet the requirement with a design. Clearly, techniques to ease detection and correction of deficiencies would be valuable. They could reduce the cost of improving the requirements and, if powerful enough, could aid in getting the improvements done early. Early improvement would reduce the cost of designing to requirements that are subsequently changed -- with the result that the design work must be repeated.

The types of requirements problems we have observed can probably be reduced in number by improving the manner in which they are developed and stated. For example, different names for the same item in different parts of the requirements specification often resulted in ambiguity or inconsistency. Some technique is needed to ensure that naming is consistent to preclude these problems. Similarly, some methodology appears needed to aid the verification and validation of software requirements during their development. The complexity of some problems is so great that we anticipate their continued presence even if powerful specification techniques are used to preclude problems; the remaining problems must be detected and corrected.

In summary, problems with requirements are frequent and important. Differences between types of requirement problems is quite small between projects. Improved techniques for developing and stating requirements are needed to deal with these problems.

References

1. BMD Advanced Technology Center, "BMDATC Software Development System: Program Overview," Ballistic Missile Defense Advanced Technology Center, Huntsville, Alabama, July 1975.

2. Alford, M., "A Requirements Engineering Methodology for Real-Time Processing Requirements," in these conference proceedings.

3. Bell, T. E., and D. C. Bixler, "A Flow-Oriented Requirements Statement Language," TRW Software Series, TRW-SS-76-02, April 1976.

4. Bell, T. E., D. C. Bixler, and M. E. Dyer, "An Extendable Approach to Computer-Aided Software Requirements Engineering," in these conference proceedings.

5. Royce, W. W., "Managing the Development of Large

Software Systems: Concepts and Techniques," TRW Software Series, TRW-SS-70-01, August 1970.

6. Meseke, D. W., "Safeguard Data Processing System: The Data Processing System Performance Requirements in Retrospect," Bell System Technical Journal, Special Supplement, 1975.

7. Thayer, T. A., "Understanding Software Through Analysis of Empirical Data," TRW Software Series, TRW-SS-75-04, May 1975.

8. Personal communication from B. W. Boehm, May 12, 1976.

9. Thayer, T. A., et. al., "Software Reliability Study: Final Technical Report," TRW Report 75-2266-1.9.-5, March 1976.

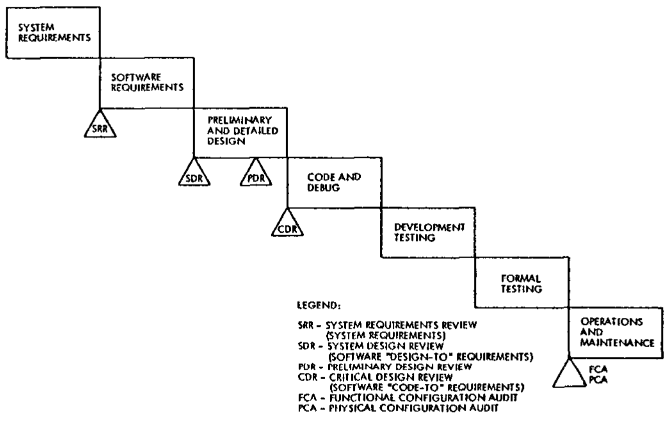

SYSTEM REQUIREMENTS
SOFTWARE REQUIREMENTS
PRELIMINARY AND DETAILED DESIGN
CODE AND DEBUG
DEVELOPMENT TESTING
FORMAL TESTING
OPERATIONS AND MAINTENANCE
SRR
SDR
PDR
CDR
FCA
PCA
LEGEND:
SRR - SYSTEM REQUIREMENTS REVIEW (SYSTEM REQUIREMENTS)
SDR - SYSTEM DESIGN REVIEW (SOFTWARE "DESIGN-TO" REQUIREMENTS)
PDR - PRELIMINARY DESIGN REVIEW
CDR - CRITICAL DESIGN REVIEW (SOFTWARE "CODE-TO" REQUIREMENTS)
FCA - FUNCTIONAL CONFIGURATION AUDIT
PCA - PHYSICAL CONFIGURATION AUDIT

Figure 1. Development Phases of the System Development Cycle

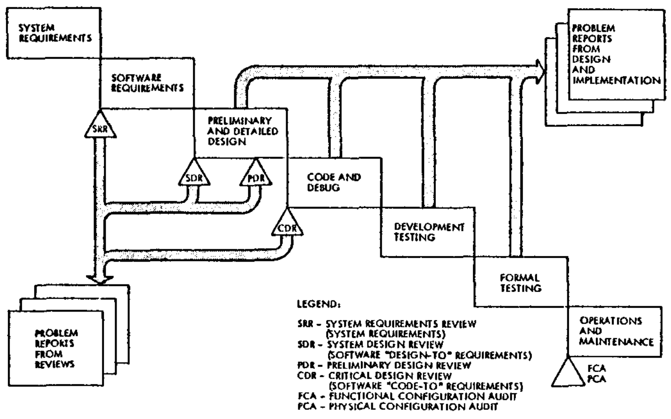
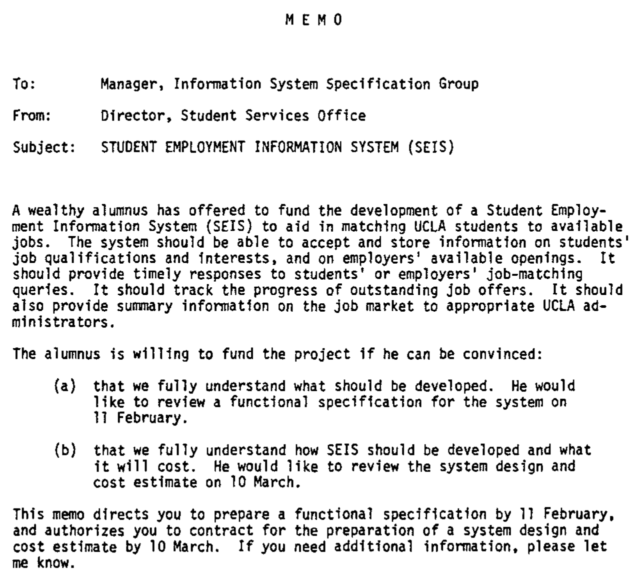

SYSTEM REQUIREMENTS
SOFTWARE REQUIREMENTS
PRELIMINARY AND DETAILED DESIGN
CODE AND DEBUG
DEVELOPMENT TESTING
FORMAL TESTING
OPERATIONS AND MAINTENANCE
PROBLEM REPORTS FROM DESIGN AND IMPLEMENTATION
PROBLEM REPORTS FROM REVIEWS
SRR
SDR
PDR
CDR
FCA
PCA
LEGEND:
SRR - SYSTEM REQUIREMENTS REVIEW (SYSTEM REQUIREMENTS)
SDR - SYSTEM DESIGN REVIEW (SOFTWARE "DESIGN-TO" REQUIREMENTS)
PDR - PRELIMINARY DESIGN REVIEW
CDR - CRITICAL DESIGN REVIEW (SOFTWARE "CODE-TO" REQUIREMENTS)
FCA - FUNCTIONAL CONFIGURATION AUDIT
PCA - PHYSICAL CONFIGURATION AUDIT

Figure 2. Sources of Problem Reports


M E M O

To: Manager, Information System Specification Group

From: Director, Student Services Office

Subject: STUDENT EMPLOYMENT INFORMATION SYSTEM (SEIS)

A wealthy alumnus has offered to fund the development of a Student Employment Information System (SEIS) to aid in matching UCLA students to available jobs. The system should be able to accept and store information on students' job qualifications and interests, and on employers' available openings. It should provide timely responses to students' or employers' job-matching queries. It should track the progress of outstanding job offers. It should also provide summary information on the job market to appropriate UCLA administrators.

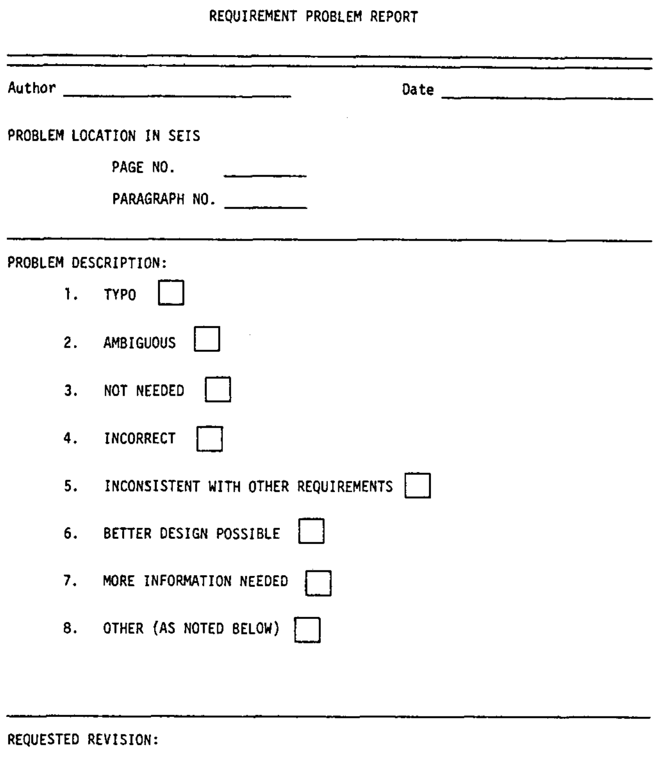
The alumnus is willing to fund the project if he can be convinced:

(a) that we fully understand what should be developed. He would like to review a functional specification for the system on 11 February.

(b) that we fully understand how SEIS should be developed and what it will cost. He would like to review the system design and cost estimate on 10 March.

This memo directs you to prepare a functional specification by 11 February, and authorizes you to contract for the preparation of a system design and cost estimate by 10 March. If you need additional information, please let me know.

Figure 3. Needs Statement


REQUIREMENT PROBLEM REPORT

Author ____________________ Date ____________________

PROBLEM LOCATION IN SEIS

PAGE NO. ________

PARAGRAPH NO. ________

PROBLEM DESCRIPTION:

1. TYPO ☐
2. AMBIGUOUS ☐
3. NOT NEEDED ☐
4. INCORRECT ☐
5. INCONSISTENT WITH OTHER REQUIREMENTS ☐
6. BETTER DESIGN POSSIBLE ☐
7. MORE INFORMATION NEEDED ☐
8. OTHER (AS NOTED BELOW) ☐

REQUESTED REVISION:

Figure 4. Requirement Problem Report

Table 1. Requirement Problem Frequencies

PROBLEM NUMBER	PROBLEM DESCRIPTION	DESIGN TEAM CLASSIFICATION	AUTHORS' CLASSIFICATION
1	Typo	0	1
2	Ambiguous	9	10
3	Not Needed	0	0
4	Incorrect	3	4
5	Inconsistent with Other Requirement	0	3
6	Better Design Possible	3	7
7	More Information Needed	8	7
8	Other	0	2
	Total:	23	32

Table 2. STP Requirements Problem Categories

ERROR CATEGORY	PROBLEM DESCRIPTION
1-000	Requirement Acceptable (but not in current design baseline)
2-000	Requirement out-of-scope
5-014	Requirement not applicable to loop
3-000	Missing/Incomplete/Inadequate
3-007	Elements of requirement not stated
3-001	Decision criteria inadequate or missing
3-002	Requirement paragraph has TBD
3-005	Interface characteristics missing
3-006	Accuracy criteria missing
3-008	Description of physical situation inadequate
3-010	Needed processing requirement missing
3-008	Processing rate requirement missing
3-009	Error recovery requirement missing
4-000	Requirement incorrect
4-001	Requirement satisfaction probabilistic (under selected conditions)
4-002	Timing requirement not realizable with present techniques
4-003	Requirement not testable
4-004	Accuracy requirement not realizable with present techniques
4-005	Requirement (possibly) not feasible in real-time software
4-006	Required processing inaccurate
4-007	Required processing inefficient
4-008	Required processing produces negligible effect
4-009	Parameter units incorrect
4-010	Equation incorrect
4-011	Required processing not necessary
4-012	Required processing not reflective of tactical hardware
4-013	Requirement overly restrictive/allows no design flexibility (includes requirements stated at too low a level)
4-014	Physical situation to be modeled incorrect
4-015	Required processing illogical/wrong
4-016	Required processing not/not always possible
4-017	Requirement reference incorrect (i.e., other documentation)
4-018	Interpretation of requirement different from updated version
4-019	Requirement redundant with other requirement
5-000	Inconsistent/Incompatible
5-001	Requirement information not same in two locations in Spec.
5-002	Requirement references other paragraphs that do not exist
5-003	Requirement information not compatible with other requirements
5-004	Requirement conventions (e.g., coordinate systems, definitions) not consistent with SDP understanding
6-000	New/Changed Requirement from PDR Baseline
7-000	Requirement Unclear
7-001	Terms need definition or requirement needs restatement in other words
7-002	Requirement doesn't make sense
8-000	Typographical
8-001	Text typo
8-002	Equation typo
8-003	Requirement identifier (number) typo
8-004	Requirement previously specified missing in updated Part I Spec.

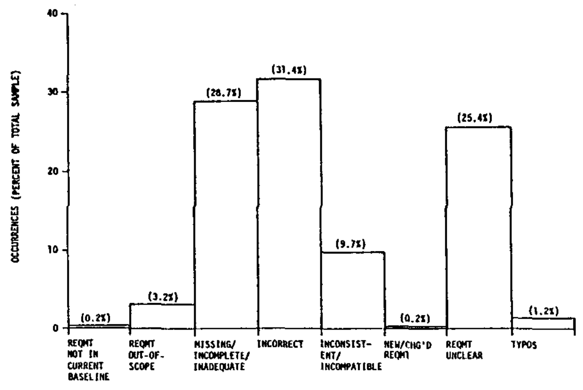


Figure 5. Software Requirements Problems - Early STP Experience (1973)

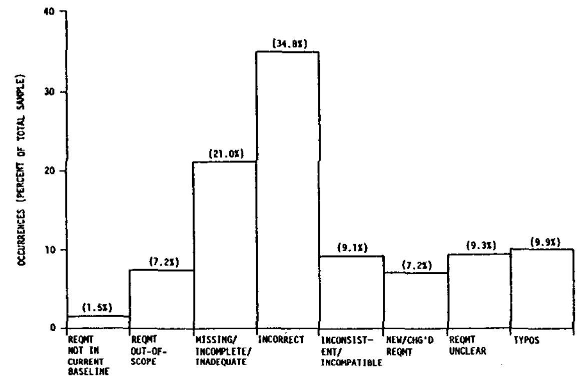


Figure 6. Software Requirements Problems - Recent STP Experience (1975)

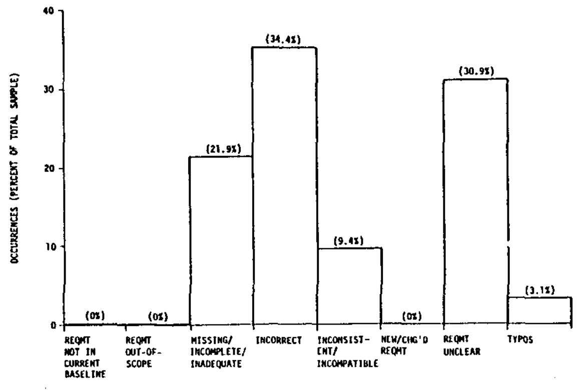


Figure 7. SEIS Problem Frequency