presented at ASSET 99, March, 1999, Dallas, TX

Systems Engineering for Software-Intensive Projects

Herbert Hecht
SoHaR Incorporated
Beverly Hills, California

Abstract

The technical shortcomings of major software-intensive projects, as well as their cost overruns and schedule slippages, are usually attributed to poor management practices. We hope to show that the application of established systems engineering techniques can help to overcome, or at least to reduce, the difficulties that are encountered in the development of large information processing and automated control systems. The paper focuses on two life cycle events that have been identified in government studies as causes of the unsatisfactory outcomes: requirements formulation and early assessment of system capabilities. It is shown that establishing a figure of merit (FOM) for the primary system function at the outset of development helps in the management of requirements as well as in the early assessment of the system capabilities. An example of trade-off methodology, an important system engineering tool that can be applied throughout the life cycle, is also presented.

Introduction

Systems Engineering emerged as a distinct discipline in connection with the ballistic missile projects that were started during the Cold War period. These projects were sufficiently large so that no one individual could keep track of required interface and resource allocation issues in the same way that a Chief Engineer usually made these decisions, at least at the top level, in WWII aircraft projects. A significant milestone in the acceptance of systems engineering as an academic discipline was the publication of Hall's *A Methodology for Systems Engineering* which drew on the author's experience in large scale communication systems at Bell Laboratories [1]. The key ingredient of systems engineering was recognition of the importance of the "Figure of Merit" (FOM), a quantification of the development objectives, around which trade-offs between contending or conflicting sub-objectives could be conducted. Eventually, the original goals of systems engineering were submerged in attempts to formally justify and document design decisions that had better be left to experts in the appropriate disciplines. This tendency transformed systems engineering from a creative design force into a paper mill. More recently we see a return to systems engineering as a specific discipline for dealing with complexity, as is evident from the subtitle *"Dealing with Complexity"* of a significant text on the subject[2]. This paper addresses the contribution that systems engineering can make to the highly complex software-intensive systems that are shaping our society.

Current Needs

The open literature on large information systems that are being developed for government agencies makes it obvious that there is a prevalence of problems for which a systems engineering approach is appropriate. But readers concerned with software in embedded computer systems and other applications that are not strictly information technology will probably find that their problems are not vastly different. First let us look at the scope of the problems in the information systems arena. "Information systems are now integral to nearly every aspect of over $1.5 trillion in annual federal government operations and spending, yet, despite years of experience in developing systems, agencies across the government continue to have chronic problems harnessing the full potential of information technology to improve performance, cut costs, and enhance responsiveness to the public…GAO reports and congressional hearings have chronicled numerous system development efforts that suffered from multimillion dollar cost overruns, schedule slippages measured in years, and dismal mission-related results." [3]

The referenced GAO report, like practically every article dealing with these problems in the popular press, cites "poor management" as the cause and lists more than 10 detailed reports to support its case. But not all managers are morons or lack motivation, and thus it may be profitable to inquire why the development of large software-intensive systems is so difficult. Specifics cited

in this and the underlying reports are failure to implement "a process for selecting, prioritizing, controlling, and evaluating the progress and performance of all major information systems and investments, [including specifically] disciplined, consistent procedures for software requirements management, quality assurance, configuration management, and project planning and tracking." The deficiencies in configuration management and quality assurance may be symptoms of a lack of discipline but, by themselves, they are not the root cause of the failure to meet the system objectives. Of greater import are persistent problems in requirements management and system effectiveness assessment in all development phases. And it is in these issues that systems engineering can offer considerable help, even if it is not a panacea. The following two sections explore these areas in some detail.

Requirements Management

Let us first realize that requirements for a large system, like DoD inventory management, air traffic control, or income tax collection, will never be complete and correct as of day *t* on day *t* or even on day *t+ x*, and that the true requirements on day *t+x* will probably be different from those on day *t*. These provocative statements are at least in part validated by the requirements documentation effort on the A-7 operational flight program[4,5] and the analysis of subsequent revisions of the document [6]. The A-7 requirements documentation was undertaken after the software was already in use, and the documenting group had access not only to the programmers but also to users. The documentation pioneered information hiding and formal specification. It was conducted by an elite group with participation by David L. Parnas. In spite of this excellent pedigree of the environment in which the requirements were documented, the later analysis found that 79 errors had to be corrected and nine modifications had been required over a period of roughly 15 months. The A-7 operational flight program was of almost trivial scope (well under 100K lines of code) compared to the software-intensive projects that are currently being undertaken.

The concept of complete and immutable requirements is a legacy of the "waterfall" model of system development that was the cornerstone of military standards. The waterfall model assumed that a user (or sponsor of the development) knew the requirements and stated them in a precise manner, reviewed a preliminary design prepared by the developer for conformance with these requirements, checked on further activities at defined milestones, and then accepted the product at the scheduled time, at the agreed to price, and obviously found it satisfactory since it had met all of the criteria for success. The methodology developed around this model provided no guidance for dealing with changing or incomplete requirements. The legacy of the waterfall continues (at least implicitly) in IEEE documents for requirements specification and the systems engineering process [7,8]. These IEEE documents form valuable check lists for gathering and presenting requirements, and for monitoring the subsequent stages of system development, but they provide no help for dealing with incomplete and changing requirements. Newer methodologies, such as spiral development [9] and rapid prototyping [10] recognize that requirements for complex systems evolve during their implementation. Rapid prototyping has been shown to provide benefits in some applications of modest complexity [11] but falls short of being a systematic way of dealing with incomplete requirements.

To arrive at an alternative to the current requirements formulation let us look at the sources from which requirements arise. Table 1 lists the major components of systems requirements and suggests how these can be treated to minimize the impact of changes.

The primary purpose of a system is a single function, such as air traffic control, communications switching, or transaction processing. And for every practical development project there is a resource constraint, usually in terms of a cost budget. The ratio of a quantitative index of the primary function to the resource unit forms the figure of merit (FOM) for the project and governs design decisions. In the air traffic example there may be a lower limit on the number of aircraft to be tracked, and there may be a hard limit on the budget, but subject to these limits a design that permits tracking more aircraft per unit resource will be

Table 1. Components of Requirements

Type	Example	Treat as
Primary function	Number of aircraft tracked	Figure of merit
Resources	Available budget	Figure of merit
Secondary function	Identify each aircraft	Trade-off
Regulatory requirements	Antenna height $\leq$30 ft	Constraint
Operational requirements	Start-up time $\leq$2 minutes	Trade-off
Interface requirements	220/380 V AC power	Constr/trade-off
Quality of service	No. of technicians required	Trade-off

preferable to one that has a lower figure of merit. Secondary system functions, operational requirements, and quality of service factors can be converted either into resource requirements (e. g., the cost of the technicians) or into the quantitative index of the primary function (e. g., a long start-up time may reduce the effective number of aircraft that can be tracked).

Regulatory requirements are not usually subject to trade-off and must be accepted as a design constraint. But even here the figure of merit approach can be beneficial by showing directly how much the constraint increases the resource requirements and possibly induce regulators to be more flexible in their demands. In principle, all interface requirements are subject to trade-off because they should represent the lowest cost to the entity that includes both sides of the interface. But sometimes it is manifestly much more costly to change an established communications methodology or data base than to accommodate these in the newly developed system. In that case the requirements posed by the existing systems become constraints. Therefore Table 1 lists the interface requirements as either constraints or trade-offs.

Table 1 forces the developer to "keep the eye on the ball" (the FOM) during requirements formulation as well as during later stages of the development. The FOM provides guidance for the formulation of secondary functional requirements, of quality of service requirements, and of most interface requirements. The approach avoids the need for stating all requirements at the outset and can be used with early prototyping as well as with other evolutionary design practices.

System Effectiveness Assessment

The lack of continuous assessment of the system effectiveness is cited in the critiques of major projects as another important reason for their failure to meet user needs. Although lack of resolve to evaluate the progress of a project is primarily a management issue, we would like to point out here again the contributions that systems engineering, and particularly the FOM concept, can make.

Practically all projects employ a resource (cost) model that can estimate the cost to complete. And most projects maintain a system model which may be either analytical or a discrete event simulation. These two components, if properly integrated, can thus generate an estimate of the FOM. In current practice this is very rarely done. Instead, the rate of expenditure is compared to "progress" which may be estimated subjectively or "measured" by labor hours or documents generated, without accounting for how the labor or documents relate to the primary system function. Thus, even if a FOM is used in requirements formulation, we lose sight of it in the later stages of system development. Little wonder then, that there is no early recognition of serious overruns or shortcomings in meeting user requirements.

By insisting on a constantly updated system model that can furnish realistic capability estimates of the primary function, and coupling this with the estimate of the cost to complete, management can get much better insight into problems that might affect the delivery of a satisfactory product of the user. And when problems are recognized, the system engineering approach provides tools for identifying secondary functions, quality of service items, and interface requirements that can be modified with the least effect on the FOM and hence on user satisfaction. Assessing progress in terms of a single FOM is both simpler and more meaningful to senior management than having to establish compliance with detailed provisions of a hundred page document.

A trade-off example

Trade-offs figured prominently in the discussion of requirements as well as that of effectiveness assessment. Although most engineers have been exposed to trade-off techniques as part of a design course, it may be useful demonstrate how trade-offs are conducted at the system level. For this example we assume a baseline aircraft tracking system that can handle up to 100 aircraft at an

Table 2. Alternatives for Resolving the Identification Problem

	Description	Capacity	Cost(M$)	FOM	Note
	Baseline (now infeasible)	100	10	10 apm	
A	Increase sweep time to 30 seconds	75	10	7.5 apm	
B	Identify 1/3 of aircraft each 10 sec sweep	100	11.2	8.9 apm	1
C	Identify 1/3 of aircraft each 10 sec sweep	85	10	8.5 apm	1
D	Install 3 identification processors	95	11	8.6 apm	2

Notes: 1 The target will move every 10 sec but the identifier only every 30 sec, thus adding to the operator workload. This requires hiring and training of additional personnel at an annual cost of $1.2 million in B or acceptance of lower capacity in C.

2. The operating and amortization expense for two additional processors is $1 million per yr. The capacity reduction is due to expected downtime of the added processors.

operating cost of $10 million per year (including amortization of the development cost). The baseline FOM is thus 10 aircraft/$10^6 which we will abbreviate as 10 apm. The aircraft identification software was intended to provide complete identification every sweep (10 seconds) but is now only capable of complete identification every 30 seconds. The proposed means of dealing with this difficulty are shown in Table 2.

Alternative B has the highest FOM and should be selected This selection procedure is very methodical and results in the highest FOM, and focus on the FOM can motivate a broad search for alternatives. However, a word of caution is in order: the identification of the alternatives is a creative step for which no ready-made instructions can be provided, and no automated aid will make up for overlooking a promising alternative.

This approach has not only identified the best way of coping with the software difficulty, but it has also furnished a quantitative measure of the effect that this shortcoming has on the system capability. This quantification of the deficiency may be useful in determining compensation to be obtained from the software developer.

The above example contains many simplifications; in a real problem much work will be required before it can be structured into a trade-off table. But investment in that effort produces a solution that can be presented, reviewed, and acted on with assurance that a rational decision has been made.

Conclusions

We have seen a well-documented history of failure to meet user requirements in major software-intensive systems and suspect that similar conditions prevail in smaller developments that are out of the limelight. Management deficiencies are cited as the root cause but the prevalence of the problem suggests that it is not the shortcomings of Peter, John, or Mary, but rather a lack of a suitable methodology for defining system requirements and tracking the implementation of these during the development cycle.

The contribution that established systems engineering techniques can make in these areas has been explored, with particular emphasis on a single figure of merit (FOM) that represents the ability to furnish the key capability (tracking aircraft, processing income tax returns, or switching phone calls) per unit resource. Use of the FOM permits some decisions on secondary characteristics (auxiliary functions, operational features and quality of service) to be completed when the architecture for the primary function has been defined and when the benefit and cost of the secondary characteristics can be better evaluated. Use of the FOM together with an evolutionary development methodology, such as rapid prototyping, also provides some protection against erroneous or missing requirements, a condition that must be accepted in a major project.

None of this is new. It's been around for forty or fifty years. Unfortunately, it is not being widely recognized or practiced. The benefits of these general systems engineering principles are much more difficult to demonstrate as a classroom project than those of more specific techniques such as requirements analysis tools. The latter are certainly valuable for systematizing requirements formulation, for removing some ambiguities or errors, and for reducing the workload. But they are not as potent as the FOM in focusing management attention on the key characteristics of the project, nor as convincing as a trade-off table when it comes to adopting or revising design decisions.

References

[1] Arthur D. Hall, *A Methodology for Systems Engineering*, Van Nostrand, Princeton NJ, 1962

[2] Richard Stevens et al., *Systems Engineering: Coping with complexity*, Prentice Hall Trade Publications, Jun 98

[3] General Accounting Office, *High Risk Series: Information Management and Technology*, GAO/HR97-9, Feb 97

[4] K. Heninger et al., *Software Requirements for the A-7E Aircraft*, Naval Research Lab., MR3876, Nov 78

[5] K. Heninger, "Specifying Software Requirements for Complex Systems: New Techniques and their Application", *IEEE Trans. Softw. Eng., SE-6*, pp. 2 – 13, Jan 80.

[6] V. R. Basili and D. M. Weiss, "Evaluation of a Software Requirements Document by Analysis of Change Data", *Proc. 5th Int. Conference on Software Engineering*, San Diego, March 81, pp. 314 - 323.

[7] IEEE Standdard 1233-1996 "Guide for Developing System Requirements Specifications", June 1996

[8] IEEE P1220(R) "Standard for Application and Management of the Systems Engineering Process" (currently in Draft format)

[9] B. Boehm, "A Spiral Model of Software Development and Enhancement", *IEEE Computer*, May 1988, p.61

[10] R. Balzer, N. Goldman and D. Wile, "Operational Specification as the Basis for Rapid Prototyping", *ACM Software Engineering Notes*, Dec 82, pp. 3 - 16

[11] V. Scott Gordon and James Bieman, "Rapid Prototyping: Lessons learned", *IEEE Software*, Jan1998, pp. 85-94