MANAGEMENT OF THE SOFTWARE PROBLEM

Edward L.Averill
Staff Engineer to the Software Department

Larry K.Rude
Associate Director Software Engineering
Training & Control System Operations
Honeywell Aerospace & Defense
West Covina, California

ABSTRACT

Two conceptual notions are identified as controlling the current system/software development operations. Both greatly impact the development process. The first concerns **requirements**, while the second concerns **program management responsibility.**

The experience of practitioners is reported to be quite different than these notions would lead us to expect. The paper examines the implications of changing the conceptual notions bringing them into consistency with the practical experience.

Requirements according to these notions should be defined with complete rigor, and design must be exactly as specified (no more and no less than). Practice shows us that requirements not only mature slowly, but also change through the development and also the life-cycle.

A development project, according to the second notion, needs to be treated (to optimize schedule & dollars for *this* project) as if both the product and the process, by which the product is built, are new work unqualified by previous work. However practitioners, carrying out the development, feel the work is *more similar to maintenance business than it is to new work* (i.e. that it is **qualified** by previous work). This is because requirements are interpretted in terms of previous experience. Old designs and implementations are adapted to the "new" project. And old code (at least the executive part and/or application framework) is used as a springboard to create the new code. In fact reusability occurs naturally to the degree that it is able. But under current notions the reusability *that is able to occur* makes little impact on development cost.

The paper explores a product line approach to project development in which only very specific *product line type requirements* are expected to be fixed, and in which the unique project requirements are implemented by a *product line development process.*

This approach is shown to be consistent not only with practical experience, but also with the current technological advances which make the new concepts practical in to-day's environment.

INTRODUCTION

The context of this paper is management of system/software development for training equipment (even though many of the concepts and ideas expressed apply to a much wider set of equipment).

The thought is that the software problem (as seen in our collective cash and time flow experience with the life-cycle of system software) is a result of our management paradigm. This thought focuses on the difficulty software engineer practitioners *(rather than managers)* have over creating an effective description by which to accurately define the software development process. With only an informal process to manage, managers have had to relie on the proven management experience with hardware development, and to apply this to the software product scene. With a hardware product a solution is the requirement for a product, and the specification is a blueprint.

This hardware background leads us to the two conceptual notions that are identified as controlling current system/software development operations, and identified as setting the stage for our adverse cash and time flow experiences. They are summarily stated to be:

- *I) Requirements must be defined with complete rigor, and the design must be exact to that specification (in the same sense as a hardware blueprint).*
- *II) A project is best served by managing with objectives for this single project only (which acts as a barrier to improvements in the technology for the software process due to the cost of making changes).*

At a recent Honeywell Software Engineering Council we observed that a greater than two-thirds majority of the attendees held views that were essentially contrary to these. This paper is about alternate ways of looking at system/software developments, and the implication that these could have on how we manage the development process.

SOFTWARE ENGINEERING VIEWPOINT

From a top level viewpoint, the software development process may be represented by the phases shown in Figure 1. In terms of this figure, 75% of the above Software Engineering Council identified Requirement Specification as the phase causing the majority of unmanageable schedule and cost difficulties. Further it was recognized that the quality of the Software Requirement Specification was totally dependent on the preceeding System Requirement Specification.

Another view of this Software Engineering Council was that the vast majority of software development was more like maintenance work than original development from scratch. Even divisions that respond to DoD RFPs for new systems very rarely break open virgin ground. For the great majority previous developments were in some way reorganized to become part of the new product. Hence the identification with 'maintenance business'. (The risks involved in developing totally new software within a fixed price contract are probably an order of magnitude greater than for developing in a already known area.)

Discussion

Figure 1 is accurate in the serial relation between requirement *(the what)*, design *(the organization of resources)*, implementation *(the how)*, and test. No matter what methodology is used, or what environment exists, it is not possible to start work on one part of a phase until the corresponding part in the previous phase is completed. If any part of a phase is started before at least the corresponding part of the previous phase is completed then assumptions *about that corresponding part have to be made.* If these assumptions turn out to be untrue then the work has to be changed.

The first conceptual notion (I) wants exact design, which plays into managing each project as something different inspite of similarities. From this position it is easy to extend into a belief that all different products need different processes to develop them. Hence there is no choice but to subscribe to notion II.

We believe most software engineers engaged in developing software for system products (such as for Training Equipment) find that requirements **cannot be fixed.** They change constantly and regularly, not only during development but also during the whole life-cycle. Clearly this experience and notion I are incompatible. We will make the case that if this is true then notion II is also incompatible.

The other software engineering practitioner's viewpoint is that development is more like maintenance than original first time new development *(i.e. experience shows that the similarities with previous projects make themselves felt inspite of notion II).*

This maintenance attitude to the development engineering process means that the Design phase does not use the specified software requirements only, but uses also design ideas and thinking from previous products. Similarly the specified design is taken from the ideas, thinking, and modules, used in previous implementations. The requirements are interpreted in terms of the design and implementation concepts with which we are familiar.

REQUIREMENT SPECIFICATION	>	DESIGN DEFINITION	>	IMPLEMENTATION REPRESENTATION	>	TEST & VALIDATION
(1)		(2)		(3)		(4)

Figure 1: The Essential Serial Nature of the Software Development Process.

Software Cost Estimation Viewpoint

R.W.Jensen's work (1, & 2) indicates there are real constraints concerning software cost (which he expresses in terms of person months and schedule time). These costraints are based upon:

1. *An Initial Staffing Rate* which relates person months to schedule time. Jensen's analysis shows there is an upper staffing rate limit above which people will adversely interact to the detriment of the project (Brooks Myth of the Man-month (3)).

2. *A Complexity Factor* of the application, and which relates the system/software product's complexity to schedule and cost. The complexity factor places an upper bound on the number of people that can be placed upon the job. Above this limit is where the application is treated as being simpler than it is. Complexity is reduced by breaking the application into separate pieces with fully defined interfaces (both horizontally for example using a properly layered operating system such as Hunter & Ready's VRTX or Ironics PSOS, and horizontally for example effective modularization and drivers): this is the activity that introduces simplicity and enables more people to work on the problem.

3. *A Size Factor* which relates size of the software to be developed to schedule time and cost. This sets a lower bound of people time to be placed on the project in order to make the schedule acceptable, and should prevent "sending a boy to do a man's job" thereby considerably extending the schedule.

4. *A Technology Constant* which relates size, schedule time, and life-cycle effort. A large number of factors, around twenty, are taken into consideration to evaluate this constant of proportionality which reduces or increases the number of people years required for the project.

What this analysis indicates is that the largest cost factors come from the process technology and not so much from the product technology. The process by which the product is built starting from the inception of the need through analysis of requirements into development and into configuration management is the real determinant of cost and problem. This supports the view of the paper that the most critical task facing industry today, as far as the "software problem" is concerned is to relook at our conceptual notions about how a system product should be developed.

WHAT IS BEING MANAGED, CONCEPTS & DECISIONS

A main objective of the paper is to show the relationship between **the schedule and cost performance we experience when managing a program** on the one hand and **the perceptions about the process used in the whole system development process** on the other. Management decisions cannot be better than these perceptions.

We suggest that most cost overruns and schedule slips in a technical development area are the result of how the technical realities of the process are perceived by those making both the explicit management decisions, as well as the implicit management decisions.

> *One of the concepts used by management is "requirements have to be fixed and stable, and design must be based on the requirements". This concept comes from a perception of the Figure 1 reality. The concept is used when there are schedules that have milestones such as:*
>
> - **System Requirements Specification accepted,**
> - **Software Requirement Document accepted,**
> - **Detailed Design Document accepted.**
>
> *Under this concept management concentrates on managing the resources, staffing and progress to complete the project on time and in budget.*

This schedule concept is totally inconsistent with the reality that requirements cannot in fact be fixed. What happens is the requirements invariably change part way through the development cycle. When the impact is too big for a work around, as is frequently the case, the schedule slips. Sometimes it is possible to reduce functionality in place of lenghtening the schedule. But experience shows that requirements continue to change through the life-cycle of the product. If our management concept is that requirements must or even should be fixed then all our management energy can be lost in keeping up with these changing requirements.

When requirements change it is frequently necessary to modify the architectural structure of the implementation, especially if the implementation has been especially fitted to the detail of the requirements for reasons of performance. This is often an underlying cause of a "software problem". This experience tells software engineers that great care must be taken to separate out the requirements that may change, and to make sure the implementation assumes no change only where there will be no change. However, software engineers frequently find that they do not have the data necessary to distinquish the requirements which will change from those that will not.

The second conceptual notion (II) is that "this" project has no connection to other projects. This concept is true for many aspects of the project, but in relation to system software development it is not really true. And it is not consistent with the fact that software development of every type is either totally or partially a maintenance type development.

By limiting our management focus to one program we prohibit explicit and formal re-use of any work products from previous development engineering. *The reality is that reusability has to be designed in and cannot be retrofitted.* Reusability can be achieved if the management perspective is able to

- take advantage of the 'maintenance business' nature of software development, and
- integrate the limited program view with the wider product line view.

This prohibition will be just as strongly in force when Ada [1] is the language we are using, inspite of Ada's capabilities to facilitate reusability via its generic package mechanism.

A CHANGE IN VIEWPOINT

There is an ongoing number of activities that are changing the realities of our development world, and this provides us with new options. To name a few of these activities:

- DoD STARS (Software Technology for Adaptable, Reliable Systems) shows their recognition that systems need to be adaptable to requirements change.
- Hardware is providing many more options to software, and software engineering is growing up fast; we are learning by effective hardware and software design to have our "high level cake" and to enjoy the performance that is necessary in the system product.
- Suppliers of Computer Aided Engineering resources are beginning to produce cost effective tools, and there is hope that the third party suppliers are in sight of making these talk to each other and be integratable into real automation support systems for system and software engineering use.

When we accept that requirements change thoughout the lifecycle of a product, we are not saying that all requirements will change equally. We believe the reality is that some requirements are fixed while others are variable. All requirements can be fitted into an hierarchical structure. The higher levels are fixed (if present) and the likelihood of variability increases with every step down the hierarchy. The top level in the hierarchy is occupied by a node for each user of the required system.

Example of Requirements in an Hierarchical Structure

For example the initial hierarchy for maintenance trainers is in Figure 2.

For each user at the next level there will be a node for each service required. The services associated with each node are mutually exclusive, but may have data interdependencies.

For the Instructor there will be such things as a service to:

- capture a lesson plan for each Trainee,
- modify an existing lesson,
- capture the performance record of a Trainee's lesson and make available for review,
- present a Trainee's current lesson situation, with the ability for message interchange.

[1] "Ada" is a registered trademark of the U.S. Department of Defense

For the Trainee such things as:

- present a step within a lesson and to respond to trainee input according to the requirements of the step, (which will depend on the malfunctions specified in the lesson plan),
- present help information on demand,
- return to an earlier step,
- control & assistance when the trainer detects an error in the Trainee's response.

Such a hierarchy needs to be supported by a data dictionary to define the meaning at every level of each noun and each verb used in the service descriptions.

Designing with Common Resources

If we take the Instructor services for an example, we see that the design can use common resources to support each specific service. For instance, **common protocol** *(rules by which the service is obtained, and rules governing the interaction),* **common presentation support** *such as editor, and windowing or menu control),* **common data base** *(how the file data is manipulated, stored, controlled and presented).*

Common resources *can* also be designed to work together in any combination that is appropriate. In fact there are basic common software resources appearing in the market for third party suppliers (and this trend will really get a boost as Ada becomes mature and widely used). These basic resources will be able to be used as the foundation for building the common product resource set.

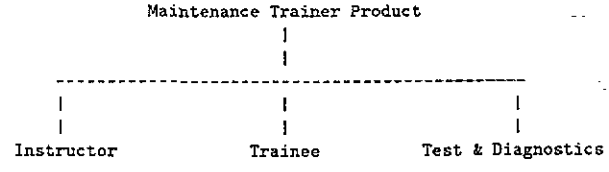

Figure 2: An Initial Requirement Hierarchy

The Software Content of a Maintenance Trainer

Figure 3 analyzes the software content of a Maintenance Trainer breaking it down into operational software *(that provides the required functionality),* and support software *(which is required to create the operational software and to enable it to run on its underlying hardware resources).*

Even the new software in the Figure 2 organization has strong links to the reusable software. For instance the Simulation Software breaks down into

- **Courseware:** this is generated using a high level courseware authoring language for specifying the lesson content of the Trainer. Software of this type (called AETRAN and CALGEN) was created and used for Honeywell procedural maintenance trainers to create the courseware.
- **Simulation Programs:** These will be supported by a software package by means of which application source code is generated. For Trainers the system or modelling engineers working the specification process, which defines the simulation software required, use this package interactively to represent the required simulation in a graphical language. This representation is translated into the simulation software in a source code form. The software package is called "the model builder". The relationship of these generated programs to the rest of the software is a standardized relationship which is independent of the simulation specifics.
- **Common Data Tables:** these can all be defined as part of the system engineering specification, (with some assistance from special software tools).

The really new items at the lower levels need only be the new Instructional Features, and the new types of Simulation Data Base Translators.

Currently all reusablitity is achieved by taking an existing product and making changes to the source level code. This means that only the coding phase which represents some 20% of the development costs has been impacted so far. The cost to implement a design, so that the result can have its own configuaration management and documentation with support specifically to permit reuse, probably represents an escalation of 2 to 10 times the cost (depending on the specifics such as language and operating system environment etc.). However if it was designed to be reusable then some 80% of the development cost and time could be eliminated on later contracts. But components will not be designed to be reusable while conceptual notion II rules and therefore prohibits any expenditure not directly useable by the current project.

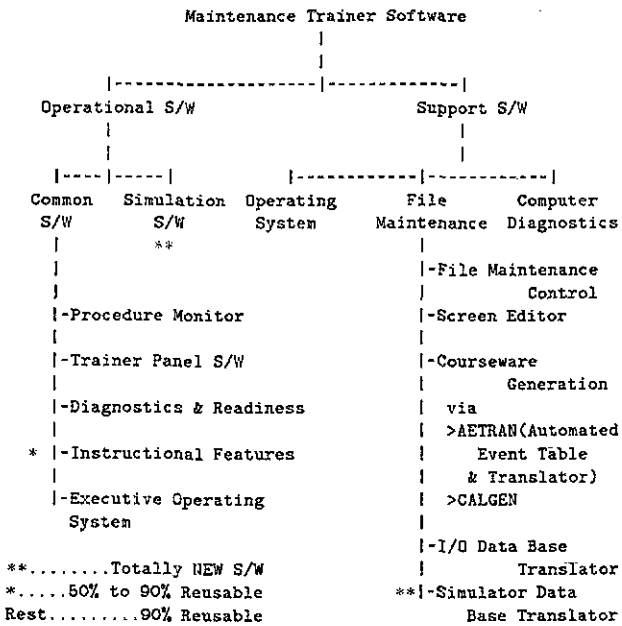

Figure 3: Reusable and New Maintenance Trainer Software

Significance of a Product Line Requirements Hierarchy

It is our experience that requirements do not change in the top levels of the requirements hierarchy. We have found requirements for similar products belong to a common hierarchical structure. *Different products will select differently from the common structure.* However at the lower functionality levels there will be many changes due to differences in requirement. If the design architecture for a new product uses the fixed requirements, then the implementation architecture for the product can be designed to allow the lower requirement levels to change. This Figure 4 type of approach makes it easier to respond to changes in the requirements.

The experience of using a very similar architecture for a variety of products reveals that one common architecture exists. Further, the way to discover this common architecture is to formally create a common hierarchy of fixed requirements for the product type. Once this relationship is digested and seen to be real then it is possible to see how to deal with varying requirements and software productivity issues.

If the common implementation architecture is made into a virtual machine then it is possible to create a special product-type requirements specification language. This specification language can be supported by a compiler which creates "object" code for the virtual machine (which is the common implementation architecture). Figure 5 shows that the variable requirements can be accommodated by writing requirements specification statements in this product type language. In this form the cost of a change in the varying requirements is made acceptable to all parties.

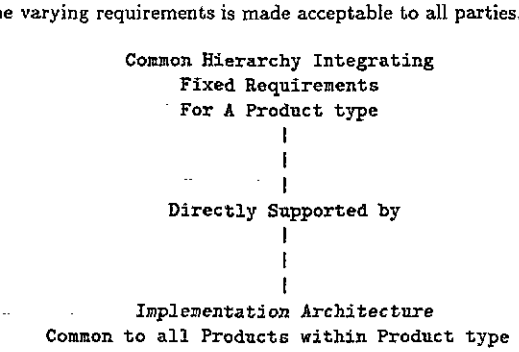

Figure 4: Relationship of Fixed Requirements to Implementation Architecture

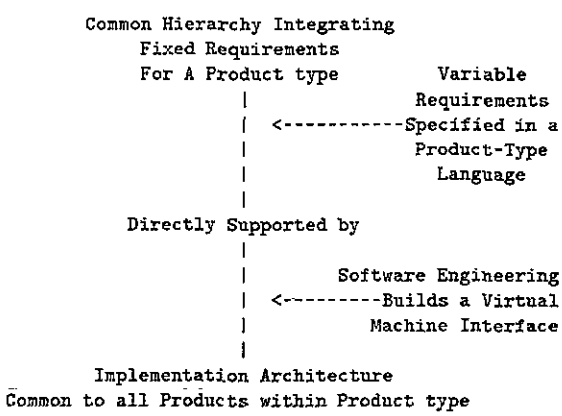

Figure 5: Relationship of Variable Requirements to Implementation Architecture

Roles & Responsibilities in the Development Process

There is significance in how management views the relationship between the skills being employed in the Figure 1 development process. The current view is of a relay race in which each skill is passed documented data and then told to run with that to play the part demanded by the phase itself. This is invariably the scenario leading into Phase (1), and between Phases (1) and (2).

However with the reality of a specification language for a product type it is possible to redefine the relationship between the Phases (1) and (2). It becomes possible to take the software engineers out of the application programming business. Instead they develop the software environment that allows the system engineer to fully specify the application by using the special product type specification language. There are potentially large savings associated with just one engineer being responsible inplace of two.

SUMMARY

What we are saying is that there is a strong commonality between the different requirement hierarchies for a set of products that belong to one product type (such as maintenance trainers or operator trainers). We have also said that for a particular product type we have been able to use a common implementation architecture for a number of products within that type.

In general we relate the fixed set of requirements, for a product type, to a common implementation architecture for that product type. Figure 4 expresses this relationship. Figure 5 illustrates that variable requirements can be handled by creating a specification language organized to deal specifically with the variable requirement set. The variable requirement set can not only vary within the life-cycle of one product, but can vary between products of the same type.

The benefit of this approach to product development is that the software engineer does not have to write application code from a software requirements document. Instead s/he maintains the support software by means of which the application code is generated from the system engineer's specification statements.

The customer also gets a benefit. Their system can be modified after delivery for a vastly reduced cost, schedule, and risk.

REFERENCES

(1) R.W.Jensen, "An Improved Macrolevel Software Development Resource Estimation Model", Proceedings of the Fifth International Society of Parametric Analysts Conference. St. Louis, MO, April 26-28, 1983.

(2) R.W.Jensen and S.Lucas, "Sensitivity Analysis of the Jensen Software Model", Proceedings of the Fifth International Society of Parametric Analysts Conference, St. Louis, MO, April 26-28, 1983.

(3) F.P.Brooks, Jr., "The Mythical Man Month", Addison-wesley, Reading, MA, 1975.

ABOUT THE AUTHORS

Edward Averill started his software career in 1955 working with machine language and paper input and output. For many years he interfaced directly with the users, and to satisfy their needs, worked all parts of the life-cycle; requirements, design, production, sell-off, and follow-up maintenance.

His current challenge is leveraging the on-going hardware and software support system advances to enable competitive software engineering development within the production of real-time system products. The target is the application software within the Training and Naval Combat System products made at the T&CSO operation within Honeywell Inc.'s Aerospace Defense Group.

Larry Rude is an Associate Director in charge of Software Engineering with Honeywell Inc.'s Training and Control Systems Operations (T&CSO). He is responsible for the software development for T&CSO system products, which include Training and Naval Combat Systems. He has a Bachelor's Degree in Applied Mathematics from the University of Idaho. He was formerly the Maintenance Trainer Software Section Head for T&CSO, where he was responsible for the software development on the F-16, F-15 and AWACS-Radar Maintenance Trainers. Prior to joining T&CSO, he was a Staff Engineer for Honeywell's Avionics Division. Before coming to Honeywell, Mr Rude was a Captain in the Air Force involved in the development of real time software for checkout and launch of Air Force Satellites.