

Project management and software quality assurance

D.W. Russell

*Penn State Great Valley, Malvern, PA 19355,
USA*

ABSTRACT

Project Management and Software Quality are a seeming paradox; some would go so far as to say an oxymoron. Because the software process is increasingly under scrutiny by a new breed of informed and intolerant consumers, the gap between the two must be bridged. The paper opens with a discussion of some often forgotten practical aspects of software personnel management and how crisis situations arise. There then follows a description of software project management and a new multi-layer, project-management system. This tool was designed to be *simple, intelligent, dexterous, powerful* and *graphically-competent*. With this tool, the software manager can predict costs and deadlines on a moment by moment basis and is able to monitor the overall *health* of the project. It is postulated, and not for the first time, that software projects that are managed poorly will produce systems that are error-prone and more likely to fail. The paper asserts that the *quality* of the product under "construction" is directly related to the manner in which the software project was managed.

INTRODUCTION

Producing software is considered by some as an "art form" and others as a "science." Each viewpoint has validity and each provides its own set of reasons why the software process is often so poorly managed, over-budget and late. If it is indeed an art, then freedom must be given for it to be created by a creative genius; if it is a science, then the writing of software is governed by laws that, if implicitly obeyed, will relentlessly bring the product into being. The discipline of software engineering attempts to reconcile the differences between the two ideologies and to introduce a notion of "process." It is unfortunate, but true, that it was generally engineers and computer scientists, who had the complex problems to solve and who had the mathematical skill to formulate formidable algorithms. It was also common knowledge that they themselves, or a close colleague, would probably be the only user of the code. As soon as software became

a commercially available commodity, written by teams of people and used by the general public, the necessity for software engineering became very obvious.

As a first step, flow charts followed by structure diagrams sprang into popularity in the belief that software blueprints could be produced which allowed every problem to be reduced to an inter-connection of state objects and modules. This was a noble effort to eliminate "stream of consciousness" programming (so popular with the FORTRAN generation) that was unstructured, unmaintainable and virtually unchangeable. However, the notion of cost and deadline was seemingly lost in this wave of elegance to all but the executive echelons of software companies. Latent within the question: "...how long will it take to write this code?" was the challenge "...how good are you?" Familiar software quotations, such as "Do you want it right or do you want it Friday" and "When the only tool you have is a hammer, every problem looks like a nail," have become widespread folklore. What has suffered most in all this has been software QUALITY that, down the road, makes the product not really work to the client's satisfaction. To correct these deficiencies, enormous resources (time and money) must often be expended by the software provider, adding to the tension between an irate client and a weary programmer. It is common knowledge that the cost of fixing software problems increases almost exponentially as the project moves towards completion. Blum [1] offers a more radical viewpoint when he suggests that it is time for a change in the orientation of software engineering away from the integration of components to solve problems towards the production of solutions to those same problems. Probably the major problem that a software manager has, is the correct allocation of resources to achieve a customer's deadline, rather than if the project team is technically qualified and capable of producing the code.

This paper focuses on the contribution that project management can make in achieving quality software in a timely fashion. The paper is in five sections: the first two (managing software engineers and handling emergencies) are based on the author's experiences as a software executive; the next two (a project management tool and a worked example) are taken from a current software project; and the concluding section offers a measure of hope to those who are floundering amid the throes of commercial software quality issues. Quality Assurance has its roots in manufacturing; Stevenson [2] defines it as:

*...concerned with the **entire** range of production, beginning with product..design, continuing through the transformation process, and extending to service after delivery.*

But first, a word about managing software engineers.

MANAGING SOFTWARE ENGINEERS

The management of software engineers is very different from that of managing personnel from other professions. Because the software process is so intractable, it is virtually impossible to set code quotas, deadlines, and manage timesheets. Software engineers will often work many hours on any particular day with little or nothing to show for their effort, but then produce many lines of code in an impossibly short time. Von Mayerhauser [3], observes the effect of coding *style* on predictive models in the section of her book entitled: *management by metrics*. Models and methods are valuable in obtaining some notion of macro-level effort and elapse times, but make gross assumptions which imply uniformity. It has been the author's experience that the more gifted software designers are the most difficult to manage--in the usual terms. It almost goes without saying, but the erratic and eccentric behavior patterns are allied closer to those of an *artist* than to those of an *engineer*; some work better after midnight, others in airports etc..

A common belief is that such personnel can be tamed and harnessed by the provision of automated tools. Gibson *et al.*[4] tabulate a list of major misconceptions concerning CASE and ICASE; the fifth item in this list states: '*.. using CASE will make you a better strategic manager or system professional*'. CASE can certainly increase efficiency, but one must also consider the question if it really can increase effectiveness.

The very nature of the software process is fraught with potential hazards, and the astute manager, desirous of software quality, must know something about handling emergencies.

HANDLING EMERGENCIES

It is a well known fact that emergencies will occur during the course of any project, be it software or construction. There are the usual sources of disasters that are unplannable; these are the familiar "earthquake, fire and flood," sickness and loss of employees; to which one may add viruses. However, in the software dimension there is a completely new and far-reaching set of circumstances that cause emergency situations to occur that are based on capability rather than incapacity. It is imperative that these factors be considered in order that software quality be maintained. The project manager may construe such occurrences as merely "delays," but if the problem is handled incorrectly, the whole system may never pass "proof of correctness" criteria or conform to the user's specification documents.



610 Software Quality Management

Incapable Hardware

There is an unfortunate, but popular, legend that the computer can do anything--if programmed correctly. This fallacy is promulgated mostly by the partially computer literate, under-informed, community of the general public; a group to which the "customer" usually and unfortunately belongs. As soon as a detailed specification exists that includes any notion of precision, speed or reliability, the requirement for high quality software can become a nemesis for the project manager. To put it simply; sixteen bit hardware will have difficulty with thirty-two bit data and the resulting inexactness will be almost impossible to detect in the short run, but glaringly obvious over long time periods. The hardware platform is critical in such systems and may be a virtually unknown quantity until the system uses live data. Most personal computer database packages offer a SORT facility, but one may well ponder how long it would take to sort a million records on multiple sort keys.

Incompatible Interfaces

If several vendors' equipment is to be connected, then it is naive to expect that the interface will be a trivial matter. The software engineer should be aware that protocol differences may require additional *hardware* purchases in order that gateway functions can be made operational. Some devices use different logic and/or voltage levels and careless connection can even damage communication ports. It is also a matter of concern that systems that are usually "servers" may be resistant to being used as "clients". For example, a device may have a built in "ENQUIRE" function that is asserted whenever no activity is detected on its bus within some time window; this can have a devastating effect on inter-processor communications--including port hanging. If software quality is to permeate the entire software process, the choice of hardware devices must be included in the specification and contract.

Incompetent Personnel

A difficult subject to broach is the possibility that the logic of a code section may be beyond the mathematical realm of the coder. For example, if the programmer is unsure about binary arithmetic then a cyclic redundancy check (CRC) algorithm will be most difficult to write. One apparent solution to this particular problem is to negotiate the removal of the CRC check, in favor of a less exacting data validation method, and accept the risk of low grade communications in the user's environment. The effect of this will only really be felt, when the system is "up and running" and those most affected will be the system installation crew at the client site.

Unfriendly Operating Systems

The incidence of OS peculiarities as potential disaster areas is too great to enumerate here. As each revision is announced, previously allowable "tricks" may cause an application, or the OS itself, to crash or behave unreliably. For example, if benchmark tests only used partial data sets, in the real system the number of open files may well exceed (a usually unpublished) capacity figure. The list goes on and on; every software developer has a volume of "horror stories" and "recurring nightmares." The software manager must keep control of the OS revision level and thoroughly investigate any code, written in any language, that utilizes operating system services.

These discussions were included, at the risk of sounding alarmist, purely to bring to recollection actual situations that may often seem amusing in retrospect, but which have grossly affected the quality and cost of software systems. Looking back, rapid decisions to alter a specification or embark on costly "code-arounds" are usually precipitated by an emergency. Whereas, maybe, just maybe, there really was time in the project to reflect a little longer before such actions were taken. Because there was no measurement and analysis tool available to assess the effect of a delay, a snap judgment may have been made. Decisions made in such haste have a nasty habit of coming back to "haunt" when the project is installed.

SOFTWARE FOR PROJECT MANAGEMENT

The tool to be described here is based on a variation of the time-scaled CPM paradigm. CPM (Critical Path Method) is familiar to project managers of all disciplines, but there are procedures that experienced practitioners use that give a much closer control and "feel" for the project that may perhaps be new to the reader.

The charter of the system was that it be first *simple* enough to allow a busy manager to enter and alter deadlines and available resources and to obtain a clear picture of the overall project status. Second, it had to be *intelligent* enough to detect and disallow logic loops and remove redundant task dependencies. It also had to be *dexterous* enough to allow the manager to drastically alter the inner structure of a project, based on unforeseen personnel or hardware variations. It must be *powerful* enough to allow the user to run "*what-if*" scenarios, using simple visually-directed procedures and *graphically-competent* enough to produce a variety of date-scannable screen displays and produce hard copy of each.

If the complexity of a task within a project is mistakenly over-simplified, the system must allow the user to expand (*fragnet*) the task into a separate



new project. After it is investigated and complete, the new project, and its resource data, must be incorporated back (*abstracted*) into the parent project and the project logic automatically resolved.

System Architecture.

Figure 1. shows the architecture of the Project Management Tool used in this paper.

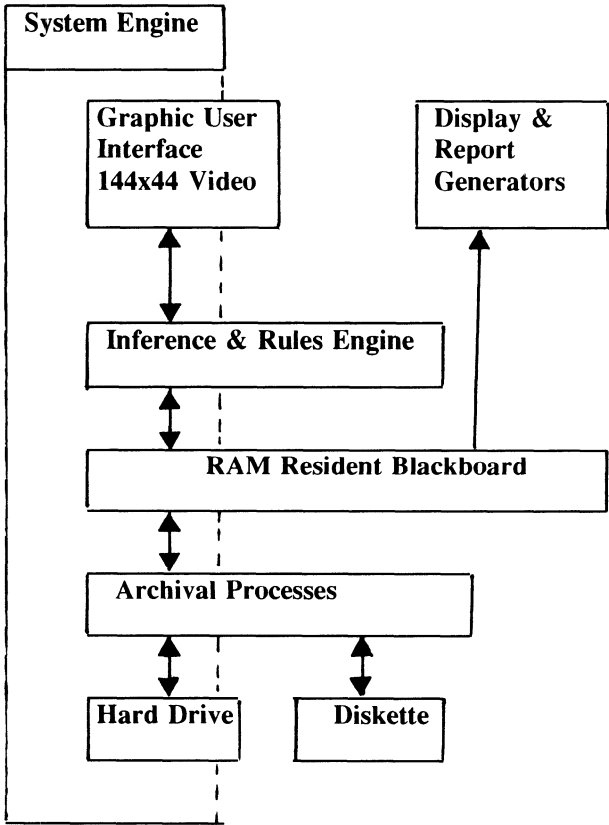


Figure 1. Architecture of the Project Management System Tool
The System Engine. The passing of data to and from the blackboard system and the logical operation of procedures within the system are managed by a system-engine that constantly interfaces with security, configuration and use-specific entities. The user is presented with the same visual interface and messaging system in all aspects of operations.

Graphic User Interface. The interface with the project manager is a crucial component in the system. Most managers are used to highly condensed, tabular and graphical, data on many sheets of paper. With this in mind, it was decided that the system screens should operate mainly in 144 columns and 44 row video mode. A super-VGA screen allows this mode of display and careful use of BIOS interrupts provided the full screen capability necessary for multi-line displays etc. The user's access to the system is through a simple visual interface. The navigation of the user through the various options within the system is clearly charted with helpful messages should an illegal option be selected. This is called the *Operations Diagram* and is used for all menu selections.

Display & Report Generation. The visual interface is used to request all displays and reports. Each screen returns to the operations diagram via a consistent "hot key" sequence.

RAM Resident Blackboard. Data pertaining to all projects is kept in a paged RAM-resident blackboard. The logic of interconnection, numeric data, and text for each task in any project resides in this area for rapid access and computation.

Inference & Rules Engine. The user may elect to perform levelling on a particular resource, or choose to add or change tasks to a project. In such instances the inference and rules engine ensures that the logic is preserved, illegal entries discarded and redundancies removed.

Archival Processes. There are two archival processes. The first handles the safe, permanent storage of project data that resides in "virtual" files in RAM. As each process is terminated by the user an archival task copies data to a permanent file. The second allows the user to safely back-off and restore any particular project to diskette. This enables similar projects to be shared among distinct users, without compromise of the security of the whole system. This is particularly useful in re-capping and re-using old projects for estimation purposes.

A WORKED EXAMPLE

To illustrate the use of the tool described above, the following is a worked example of a small project. This example of the system exercises: the project definition function, its task list, and a corresponding cpm chart.

Project Definition. This screen allows the user to define all necessary aspects of the project at a high level and is shown as Figure 2.



614 Software Quality Management

PROJECT DEFN.	PROJ.No : 01001-01005-----									
	Level	1	2	3	4	5	6	7	8	9
PROJECT NAME	SQM'94 DEMONSTRATION						COMPANY	PENN STATE GREAT VALLEY		UPDATED TO:
PROJECT MGR.	DR. DAVID W. RUSSELL						DIVISION	ENGINEERING		LOCATION PA
APPROVED BY	PROVOST						APPROVAL DATE	24FEB94		

PROJECT DESCRIPTION
DEMONSTRATION OF PROJECT MANAGEMENT SYSTEM
FOR SQM'94 PAPER

PROJECT GOAL
A SIMPLE SOFTWARE PLAN

PROJECT PURPOSE
SOFTWARE DEVELOPMENT

ASSUMPTIONS

TIME UNITS Days
 Work Days per Week

PLANNED START DATE	01MAY94	PLANNED DURATION	50	PLANNED COMP.DATE	01SEP94
FREQ OF UPDATE	WEEKLY	\$ AVAIL/Time Unit	1000	PLANNED PROJ. COST	50000

SUBPROJECTS	01- SMALL JOB	02- None	03- None
-------------	---------------	----------	----------

RESOURCE CODES/RESPONSIBILITIES

No:	BOX	Color	Description	\$Avail	\$/unit	Manager	Phone
1	CYN	RED	PROGRAMMERS	2	40	DWR	X233
2	CYN	BLU	ANALYSTS	1	100	CWC	X217

DISTRIBUTION LIST

CWR	CWC	DWR
-----	-----	-----

Figure 2. The Project Definition Interface.

The project is keyed from a project number that is grouped, hierarchically into up to nine, five-digit numbers. There is an implicit inheritance between project numbers. For example: Project 01001-01005 is related to 01001. The important fields are time-units, the start date, project duration, expected completion date, per unit cost and estimated total project cost. These fields provide information to the detailed workings of all succeeding operations

Task List. A project is made up of tasks, which are entered through the GUI provided. Tasks contain information about themselves and their relationship to other, existing tasks. Figure 3 shows an example of a task list. Tasks can be stand-alone (ie. "SET") or else be logically related to other tasks in the list. For example, a task may only be worked upon after another task end (ie. "FOLLOWS") or a task may be performed concurrently with another task (ie. "WITH"). Once a task list is established, the tool allows dependancies to be changed, tasks to be added, inserted or deleted. The inference and rules engine traps and disallows logic-loops and eliminates redundancies as they occur. This process is



complex and time-consuming and operates exclusively on the RAM-resident blackboard. The start date of the project is inherited from the project definition form of Figure 2.

TASK LIST

PROJECT

01001-01005

LEVEL

123456789

Page

1

PROJECT NAME

SQM'94 DEMONSTRATION

COMPANY

PENN STATE GREAT VALLEY

UPDATED TO:

PROJECT MGR.

DR. DAVID W. RUSSELL

DIVISION

ENGINEERING

LOCATION

PA

APPROVED BY

PROVOST

APPROVAL DATE

24FEB94

S TSK TASK	TSK RES RES OTHER	START	COMP	FOL	WITH	BY	TSK	TSK	FLOAT
P NO. DEFINITION	DUR NO. AMT COST	DATE	DATE	TASK	TASK	UNITS	LEV	SET	
Sub Project 01 : SMALL JOB									
01001 START BACKGROUND WORK	3 2 1		01MAY94 04MAY94					Y	
01002 SPECIFICATION WRITING	5 2 1		04MAY94 09MAY94 01001						
01003 LEARN NEW LANGUAGE	6 1 2		02MAY94 08MAY94					Y	
01005 START CODING	5 1 2		09MAY94 14MAY94 01002						30
			01003						
01006 TEST SOFTWARE	5 2 1		14MAY94 19MAY94 01005						
01007 FINALIZE PACKAGE	10 1 2		19MAY94 29MAY94 01006			5			
01004 TEST COMPILER	3 1 1		04MAY94 07MAY94 01003			2			

Figure 3. A Typical Task List.

The CPM Chart. A useful display is a CPM diagram. This is built from the task table and uses the blackboard logic to order and place tasks on a two-dimensional display. Figure 4 shows the CPM chart for the task list in Figure 3.

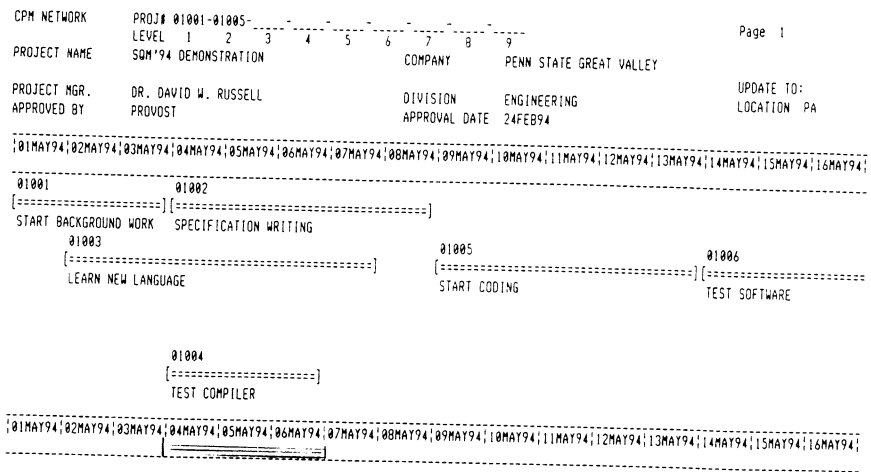


Figure 4. The Corresponding CPM Chart from Figure 3.



616 Software Quality Management

The Histogram. Once the logic of the various tasks within the project is established, the project management tool allows the user to view the individual resources required to staff and perform the project. The user selects a resource (see Figure 2.) and the histogram shows the total resources required for the planned project. If a resource requirement exceeds the availability, defined in Figure 2. then the summation cell blinks. One could immediately see, in the figures above, that the resources available are exceeded between May 4-7th!

Resource Management. In order to rectify this resource problem, the system will reposition tasks, based on available "slip" or "float" so that the manager can perform "what-if" analyses, before making snap decisions as to new hires or purchases. The system interconnects with its cartel of displays (and reports) so that visual feedback of consequences is automatic. In the example, it would appear that the programmers should begin the task of "learning new language" as early as possible, so that they would be ready to start coding on May 9th., and could then leave some of the compiler testing until later. This would solve the problem.

CONCLUDING COMMENTS

The paper has sought to attack the very difficult problem of software project management from several angles. The nature of the players in the task was explained as kindly as possible and a software tool that was designed to adapt to changes and delays described as a helpmeet in the managerial process. It is well known that software quality doesn't just "happen" but that it must be an integral part of the whole process [5]. Investigation of models that pertain to software reliability, e.g. [6], have affirmed a credence in the fact that software does, and to some extent *must*, fail! This is because it is virtually impossible to test every single data path in even a simple program. It is, therefore, all the more inexcusable that such failures be a direct result of poor project management and unwise decisions. It is asserted that software tools that are applicable easily to the peculiar nature of the software process must have a positive influence on the overall quality of the software product.

ACKNOWLEDGEMENTS

Grateful acknowledgement is made to Charles W. Cadwallader and Robert W. Hoeflein P.E., principals of MGT Inc. a project management software company, for their comments and use of the MGT SYSTEM_1 tool in this paper.



REFERENCES

1. Blum, B.I. 'Integration Issues Elucidated in Large-Scale Information System Development' *Journal of Systems Integration*, Vol 1, No. 1, pp. 35-53, 1993.
2. Stevenson, W.J. *Production/Operations Management*, 3rd. Ed. Richard D. Irwin, Boston. pp. 806. 1990.
3. Von Mayerhauser, A. *Software Engineering- Methods and Management*. Academic Press, San Diego, pp. 556-559. 1990.
4. Gibson, M.L., Snyder, C.A. and Rainer, R.K. Jr 'CASE: Clarifying Common Misconceptions' *Journal of Systems Management*, pp. 12-19, May 1989.
5. Jalote, P. *An Integrated Approach to Software Engineering*. Springer Verlag, New York, pp 107-112, 1991.
6. Knafl, G.J. and Semrl, A. 'A System to Support Software Reliability Modelling. *Proc. International Symposium on Engineered Software Systems--ISESS'93*. Malvern. World Scientific, Singapore, pp 156-165. 1993.