SOFTWARE QUALITY: PROCESS OR PEOPLE

by

Regina Palmer
Martin Marietta Astronautics
P.O. Box 179, M/S S1008
Denver, CO 80201
(303) 977-5748
&
Modenna LaBaugh
R&FC Group
3076 So. Hurley Circle
Denver, CO 80227
(303) 986-3729

This paper will present data related to software development processes and personnel involvement from the perspective of software quality assurance. We examine eight years of data collected from six projects. Data collected varied by project but usually included defect and fault density with limited use of code metrics, schedule adherence, and budget growth information. The data are a blend of AFSCP 800-14[1] and suggested productivity measures in *Software Metrics: A Practioner's Guide to Improved Product Development*.[2] A software quality assurance database tool, SQUID,[3] was used to store and tabulate the data.

The projects represented varying degrees of programmer expertise, acquaintance with software engineering techniques, and languages including Ada, C, FORTRAN, LISP, Pascal, and Prolog. The programs evaluated were engaged in the production of simulation, R&D, mission operations, flight, or ground support software. The size of the programs ranged from small, $500K to $2 million, to large, in excess of $40 million.

Amongst the projects, we were able to track the responsiveness of different programmers to improving quality based on assessment and feedback. When quality goals and standards were established, and stressed by management, the compliance of all could be obtained. Knowledge of management expectation was especially important. This could be seen in how peers reviewed each other's work according to the tone set by their lead.

At a company such as Martin Marietta which develops software for various agencies of the government, each with its own concept of getting the job done and the related cost, flexibility of process is often sought. Such flexibility usually translates into eliminating budget for independent assessment

[1] AFSCP 800-14, Software Management Indicators, Management Quality Insights, Air Force Systems Command, 20 Jan 1987.

[2] *Software Metrics: A Practioner's Guide to Improved Product Development*, K. H. Moller and D. J. Paulish, Chapman and Hall, 1993.

[3] Software Quality Assurance Interactive Database, produced by R&FC Group.

of the quality of products and processes. It is easily argued that the engineers themselves are responsible for the quality of goods and will see to it. As soon as signs of engineering neglect appear, this course is overturned and outside evaluators play catch up on project. Such action is not cost effective, but can produce an accepted delivery. The most cost effective quality program, in our experience, is one that assesses the output of individuals early in the process then concentrates on those who show the least commitment or understanding of the quality of work expected.

In our case studies, deviation from expected output seems to occur amongst programmers who have only produced software for internal use that does not need integration or coordination with other software producers, those without a familiarization with programming standards and procedures, and those who believe rules were made for someone else. The first two are helped by a well defined process, the latter are a roadblock in any effort.

The success or problems of the process used on six programs is presented here. Each program is characterized according to lines of code effort, number of programmers, experience level of programmers, criticality of software, degree to which contract requirements or budget guided the process selected, and success of the process implementation as measured by the quality assurance effort. The projects reported range from the best of all possible worlds to the worst.

The best world is one where all (most) parties agree on the process and are committed to adherence. Next best has management in agreement with a restrained acceptance by the programming staff. The worst world has a process imposed upon other habitual methods causing rework costs to soar and pitched battles on a daily basis over budget and schedule.

Case study 1 was a model small program involving non critical ground support software written to perform on a personal computer. Data from it is included to show nominal cost of quality when the process produces the desired result with little rework.

Case study 2 involved a small project of four to seven programmers developing non critical software in C and Prolog. Response of the programmers to metrics collection was examined and its influence on their subsequent work analyzed.

Case study 3 was a large project involving a software engineering staff of 20 to 40 people using Ada to develop software for a space experiment with human interaction and support software. The performance of groups of programmers in the process defined for a full life cycle program is examined and related to varying management expectation.

Case study 4 is our worst case model, where everything is wrong and the solution requires replacement of staff. The programming languages were C and assembly.

Case study 5 was a medium sized project producing flight and ground software for a space experiment written in Ada and C. It had a well defined process and the study documents how much rework was involved related to programmers per their level of acceptance of the process imposed.

Case study 6 was a small project that was an engineering support task for one of the NASA centers. It is used to show the difference in performance of the programmers involved based on their past experience with software engineering discipline. No contract criterion was available but each was totally responsible for code to work on a particular platform.

DEFINITIONS

Productivity where reported was calculated from total software engineering hours, including support from or for systems engineering, systems testing, program management, software quality assurance, and direct support from areas like finance and planning. It represents the deliverable lines of code divided by these hours times an eight hour day.

Fault Detection is reported in a form to show what activity was used to find defects and in what phase they were found. It is also sometimes displayed to emphasize the quantity of defects found in house versus at the customer's site. Types of defects are reported if needed to explain the other data. The process of tracking discrepancies in software provides information to help improve productivity and efficiency. When problems are discovered during integration and system test, the priority of the error is examined in addition to what caused the error. Priority of errors can range from errors that make the system inoperable to errors that do not disrupt the running of a test. The following details the levels of priority used by this paper:

A - error in the code in which the software did not meet the requirements or design, an error which was a documentation error which caused the code to not meet the requirements or a code error which crashed the system making the system non operational until the error was fixed.

B - error which crashed the system but there was a work around and the system could be used.

C - error found in the code which did not interfere with the operation of the system.

D - a minor error in the code such as a typographical error in a help message.

The cause is examined to identify the reason for the error. The cause could be a requirements error, design error, coding error, hardware interface error, or a requirements change directed by the customer.

In the few examples where code metrics are used to emphasize the difference between programming groups, items found of value were the size of modules, and adherence to coding standards.

Schedule adherence was based on planned dates for major milestones versus actuals and slippage in the final delivery. A major milestone was usually a formal review or delivery. In program 3, which was cancelled before delivery and stretched out twice before then, adherence to internal schedules was used to compare team performance and responsiveness.

PROGRAM 1 was a model small program involving non critical ground support software written to perform on a personal computer. Data from it is included to show nominal cost of quality when the process produces the desired result with little rework. This program consisted of 7500 lines of mission operations software. There was a software lead, six software engineers, one systems engineer, and a test engineer. The 7500 lines of code were required to be developed and delivered in six months. The entire team was experienced in developing and testing software. The program philosophy was to use senior personnel to ensure that every task was completed on schedule.

The program began by producing a software development plan to document the process to be implemented. As the SDP was being developed, the program conducted tabletop reviews with the engineers that would implement the plan, SQA, test, systems, and the customer representative.

The tabletops were used to ensure that a process was developed that incorporated good engineering practices as well as being streamlined. Once the SDP was approved and agreed upon by everyone, the requirements were finalized. The customer was very involved assuring that the requirements were finalized in a timely manner because of the tight schedule. A requirements review was held to baseline the requirements with the customer. Design and code walkthroughs were held to ensure that the design and code implemented the requirements and that the design and code standards were adhered to. At each of the walkthroughs the software lead, SQA, test, systems, and a customer representative were present. This ensured that everyone was aware of the state of the software and agreed upon the results of the walkthroughs.

Figure 1 shows the planned versus actual schedule adherence by program 1. One week of slip in the schedule occurred during the code phase but that slip was recovered during system test allowing the program to still deliver the software on time to the customer.

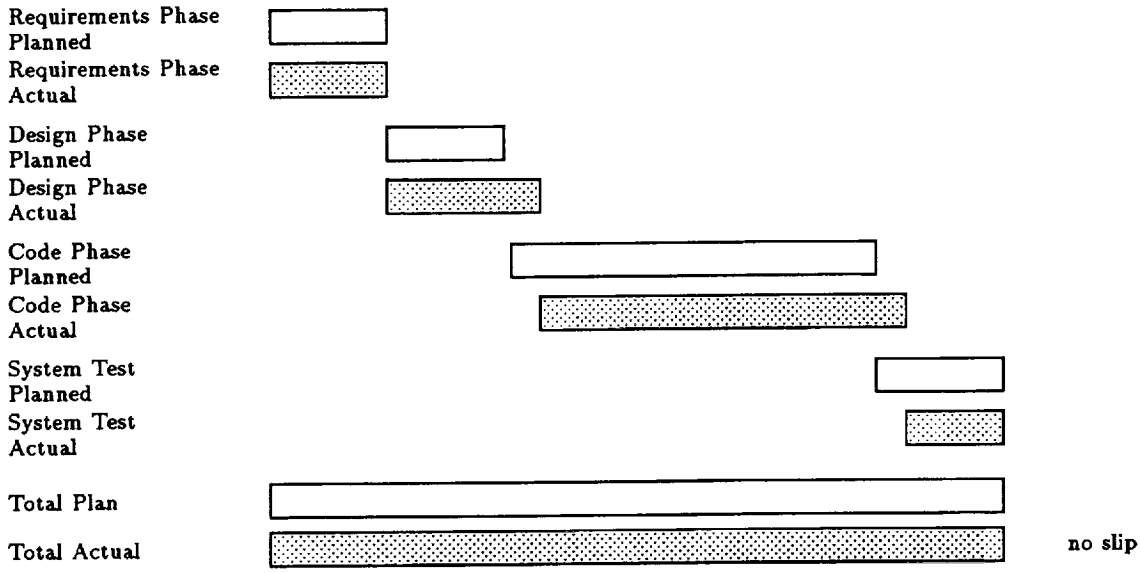


Figure 1 – Program 1 Schedule Adherence

The results of the reviews of the requirements document are shown in table 1. The requirements document improved after each review excluding the preliminary design phase which had a review cycle too short to allow incorporation of meaningful corrections.

Table 1. SRS Document Completion Index

Phase	DI Score (1.0 high)
Requirements	.67
Design (PDR)	.67
Design (CDR)	.77
Coding	.83

Figure 2 shows the breakout of the types of errors found in each phase of the life cycle. Most of the requirements errors were discovered during the requirements phase. The requirements errors

found during the system test phase were due to the customer changing the requirements prior to delivery. Overall the errors found during the life of the program resulted in a fault density score of 0.8 discrepancies per 1K line of code. There were no errors found after delivery of the software to the customer site. 93% of the discrepancies were found before system test.

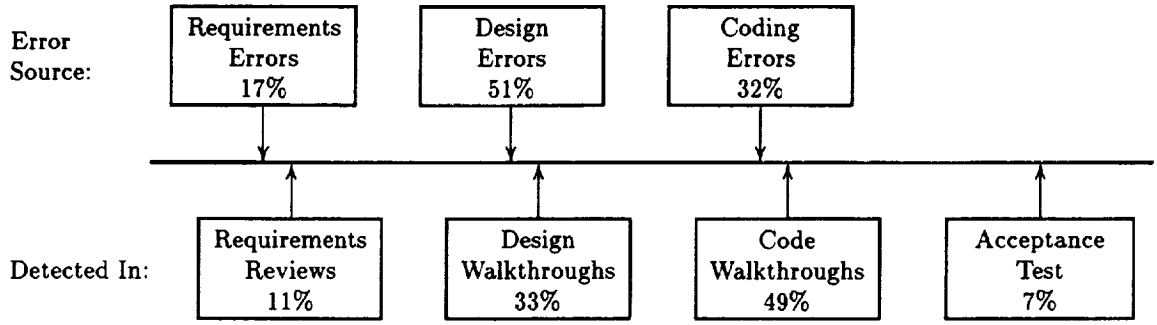


Figure 2 – Percentage of Errors Found by Software Phase

This small program was very team oriented. At the start of the program, the lead, software engineers, test, SQA, and the customer representative got together and decided that everyone would need to work together to meet the scheduled delivery date. All engineering staff were experienced in their fields and responded to action items from their peer reviews rapidly. The productivity was below expected standards but adherence to schedule and accuracy of the code were the drivers. Measured by those standards, the program was everything desired.

PROGRAM 2 was less than $2.5 million including delivery of commercial workstations used in development. Requirements and most design had been accomplished under a previous contract. The documentation produced under the previous contract included a system specification and functional description document. Effort reported was a planned two year implementation and system test with two deliverable prototypes and a final operating capability. New deliverable documentation to be produced to Air Force standards included program and database specifications, and the user's manual. Projected coding effort called for five to seven programmers, including three leads with more than two years experience at the company and the remainder being college graduates. Contract was bid with a 20 LOC per day goal for each engineer. 81KLOC were developed (70K C, 11K Prolog/LISP) and 199KLOC delivered (included legacy code from previous contract). The end product was software used in a lab environment.

The process involved informal walkthroughs involving the software lead, programmer, tester, and quality. Unit Development Folders were maintained till delivery. Programming standards described headers required for all code, commenting and self descriptive naming for variables. A guideline of less than 100 LOC per module was not enforced. A tester who was not part of the software development staff was used for Computer Software Component (CSC) integration into the prototypes and final delivery. Only test results of the CSC integration were reviewed. Discrepancy tracking was initiated at CSC integration.

Defects were measured only through the testing program. Software measures for code simplicity, self-descriptiveness, and conciseness were obtained on the C code. This was accomplished with a code reading tool that calculated the Halstead Measure,[4] branching complexity, lines of code,

[4] *Elements of Software Science*, M. Halstead, Elsevier, 1977.

commenting as a percentage of total non blank lines, and variable density. The numbers found for the code were not used as acceptance criteria. They provided a background from which to evaluate changes in the code resulting from error reports. Large changes in any score were viewed as cause to reconsider acceptability of proposed changes. Such a screening was used due to limited resources for people to review changes.

The programming standards on the project were dictated by two of the leads. They adhered to them, the third lead did not and the junior programmers did only after they became aware that lack of adherence was reported to management. Most programmers responded positively when management made metrics goals visible to them. In the first audit of code compliance to standards, two samples were taken representing code from senior programmers in sample A and less experienced programmers in sample B. 81% of sample A was above average in score, but only 59% was above average in sample B. Sample B ratings tended to be either very good or very poor with less than 20% of the modules falling in the middle. The results of the audit were distributed to the programmers and a limited amount of time was authorized for rework. The group represented in sample A reworked code that fell below the minimal acceptable level raising their mean score to 3.8 from 3.6. Sample B programmers reworked all code scoring average or below bringing up the mean score for sample B to 4.4 from 2.9. The entire sample rose to 4.1 (excellent) from 3.2 (good).

Table 2. Code Compliance Audit

Sample	Modules	Score	Rework
A	154	3.6	3.8
B	176	2.9	4.4
A + B	330	3.2	4.1

A major problem in the methodology used on this program was the lateness of finding the majority of errors. 85% of the errors in the code were not found till integration of the final deliverable though 60% existed in code baselined a year earlier. This occurred due to inexperience on the part of the integration tester and a flaw in the test philosophy of the development personnel. The tester assumed unit testing of low level functions had been performed by the developers. The software leads were more involved in code development than anticipated and did not exercise sufficient oversight of the unit test effort. Functional testing of the first baseline was not performed because it was legacy from the previous contract and assumed working because of acceptance at the customer's site.

Table 3. Discrepancies in the Baselines

Baseline	Size	Fault Density
IOC1	22KLOC	20
IOC2	33KLOC	3
FOC	70KLOC	3

After delivery of the second prototype, the test philosophy on the program changed. More unit testing was demanded before integration. Upon being told that code discrepancies were being tallied during integration testing, the programmers became very active in finding and documenting errors in the baselined code while performing unit testing prior to integration of their own code with that baseline.

The following table is in order of programming experience.

Table 4. Code Fault Density by Programmer

Programmer	Fault Density of Code
P1 - sw lead	8
P2 - sw lead	18
P3 - new graduate	10
P4 - new graduate	8
P5 - new graduate	17

P2 and P5 were reluctant to take time to test. P4 was used to do most correction of P2 code and P3 was used to correct P5 code because of the low fault density of code they wrote.

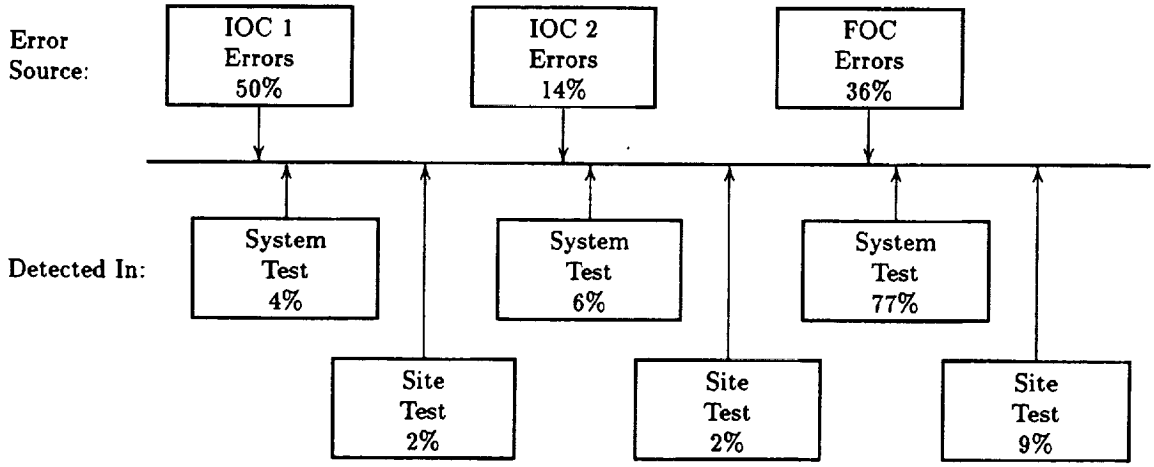


Figure 4 – Distribution of Errors in Baselines

The perceived error rate (that seen by the customer) was 1 fault per 1000 LOC. This was low in the customer's experience and the customer was pleased with the software and regularly used the prototypes from the first two deliveries.

Table 5. Error Detection Activity

Phase	Engineering Test Errors Found	Site Use Errors Found
IOC1	4%	2%
IOC2	6%	2%
FOC1	21%	8%
FOC2	56%	1%

Schedule slippage appeared after planned enhancements to old code were completed and newly developed code was nearing baseline. Up to three months before the planned delivery date for the final operating capability, the program manager was reporting the program was on schedule. Estimates made by the quality representative of test completeness projected that 90% of the errors

in the code had been found. This was based on the completeness of scheduled testing by the test department and the assumption that the error rate established in the first baseline testing of 2 defects/KLOC would not grow to more than 4. Unfortunately, this estimate was in error as became apparent when attempts to verify the completeness of unit testing for the final delivery were initiated.

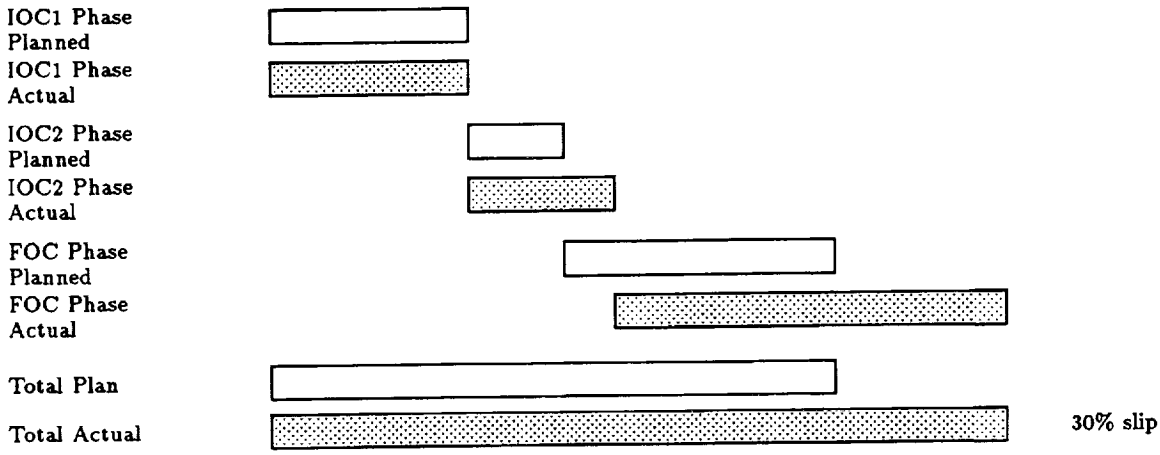


Figure 6 – Program 2 Schedule Adherence

The goal of 20 LOC per day per engineer was not met though at 16 LOC per day they did exceed the company expectation at that time for production of ground support software. General problems surfaced in relation to additional resources required to bring into compliance the code of the third senior programmer and to make functional legacy software that should have been working but was not. This effort, undertaken at the end of the development program, contributed to a significant cost overrun that ate all profit bid for the program plus additional company funds.

The planned quality assurance budget on the program was exceeded by 10% and engineering budget by 36%.

PROGRAM 3 was valued at more than $500 million including major space qualified hardware development but less than $30 million for software. Software effort was projected to include from 14 to 40 developers over three years. There were five leads with experience in the range of 5 to 15 years. 60 KLOC Flight and 50 KLOC test bed software were to be developed by mostly experienced programmers with subsequent updates for additional deliveries. 114 KLOC ground software were to be developed or flight code would be reused and modified by programmers of varying skill level. The programming language was Ada with less than 200 LOC of C used in the ground support software. SSP30000[5] was the required standard.

The process covered a full development life cycle, including formal reviews, massive documentation, independent test, and software system engineering at the start of contract conducted by a group separate from software development. A programming and procedures standard covered coding practices, defined the walkthrough process, software development folder contents, baseline activities, and unit and informal CSC integration testing. Independent tests at the top level CSC (TLCSC),

[5] SSP30000, Space Station Program Definition and Requirements, Section 2 Program Management Requirements.

CSCI and system level were to be conducted. At the outset of the program, a methodology with heavy involvement from groups separate from software development including system engineering, test, quality assurance and system safety was instituted. The on site customer representative became involved during the critical design phase.

Software documentation changes were controlled by the software manager after PDR through a software review board (SRB). Modification to the requirements and design documents was through redlines submitted to the SRB that were reviewed by the leads for all the CSCIs, software quality, software test, and system engineering.

Defect density was measured using the numbers of software change requests processed by the SRB and the action items generated by reviewers of walkthrough packages. An evaluation program run by software quality furnished a rating on the documentation and code produced which yielded a Document Completion Index value by phase.

The program was plagued by poor definition of requirements. It was a continuation of a previous Phase B study that supposedly brought the product to a PDR level. On the subsequent Phase C/D contract, more stringent requirements for process and products were imposed which necessitated regeneration of documentation thought to have been completed in Phase B, and presentation of a software requirements review (SRR). This had not been anticipated and directly impacted the CDR schedule.

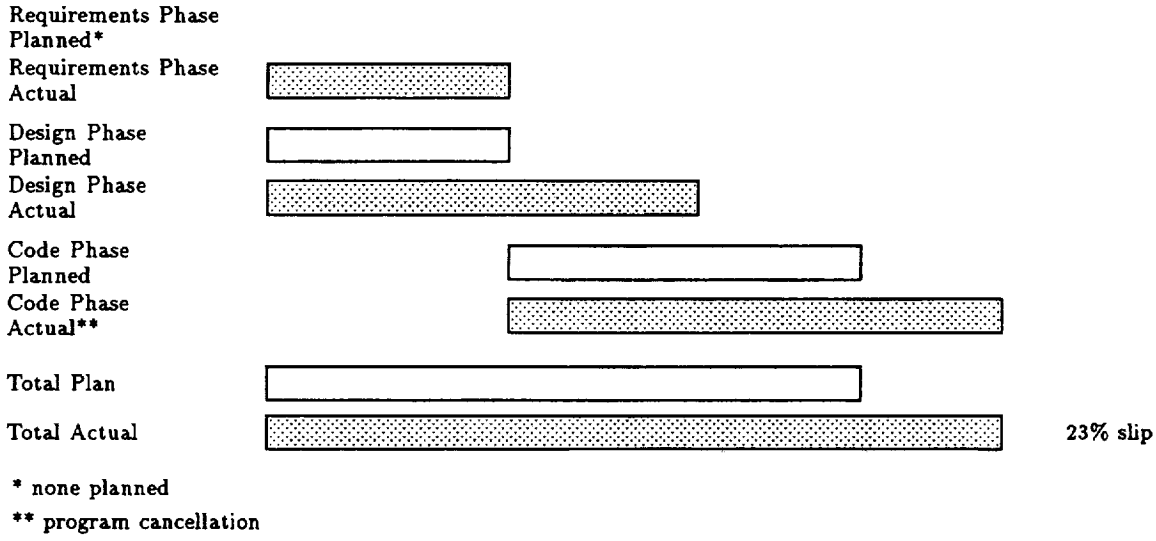


Figure 7 – Program 3 Schedule Adherence

The increase in overall schedule included stretch out of the program. The slip due to requirements definition accounted for all the slip in the design phase and a third of the overall slip in the program through coding. The rest was a program replan dictated by funding constraints for the program.

Even with the three months of additional requirements definition, the requirements document did not improve. The following table shows the progression of Document Completion Index for the three major documents produced by the software group for the flight CSCI. The SRS went through 10 iterations before the coding phase but did not get 50% of the available points.

Table 6. Document Completion Index

Phase	SRS	SDD	STP	Project
PDR	.17	.55	.50	.40
CDR	.36	.74	.61	.57
Coding	.64	.74	.61	.66

The program was terminated due to loss of funding half way into the flight software development cycle. The above table shows a steady progression but clearly indicates that the SRS did not meet the standards required. Its improvement for CDR was obtained by diverting design personnel from the software area to support system engineering in rewriting the specification. This still did not overcome the reluctance of the document authors to specify testable requirements. The rise in its rating shown in the last score it received in review was obtained through the cumulative effect of SRB actions to remove implementation detail and replace with testable requirements.

The program was terminated before system test but the following chart shows the distribution of known errors and what activities were used to find them.

A large number of requirement changes were generated by the customer and hardware designers after the software design was baselined. Of the changes directed against requirements, 32% were change summaries from the hardware subsystems. Software on this project responded to requirement change with software fixes, since it seldom could demonstrate that a hardware change for a problem would be better than a software fix. Better on this project always meant cheaper or quicker.

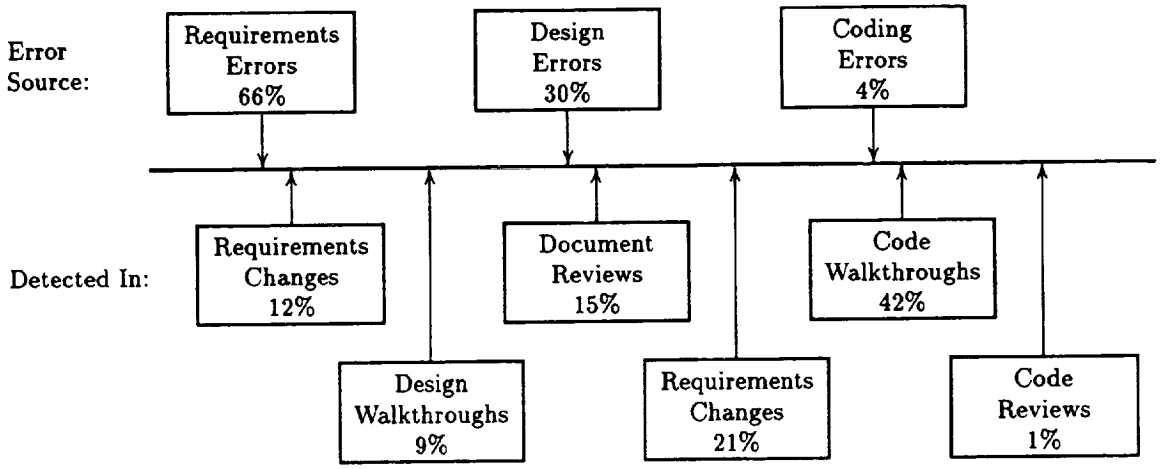


Figure 8 – Fault Detection

There was a noticeable difference in the attitude of the groups producing the four CSCIs. The three ground CSCI teams were reluctant to respond to action items and were lead by personnel who believed completion of code was the number one priority and if it worked all else would be forgiven. This corresponds to a black box mentality, i.e., the user should be happy with the result and not want to know what's inside. This is contrary to currently established specifications for software development.

Table 7. Action Item Response Time

CSCI Name	Number of Items	Average Response Time (days)
FLIGHT	24	33
EGSE	13	46
TRAINER	5	60
SIMULATOR	10	83

The items in the table, while not of top priority, were still non compliances to the contract which required correction.

The ground teams also were more reluctant to meet internally established dates for review of their work by peer groups.

Table 8. Code Schedule Adherence

CSCI NAME	W/T Dates Missed	Average Slip (days)	B/L Dates Missed	Average Slip (days)
FLIGHT	10%	19	23%	15
EGSE	26%	28	43%	21
TRAINER	14%	7	0%	0
SIMULATOR	18%	28	18%	14

At termination of the contract the quality assurance budget for evaluations was 173% over plan. The constant re-review of non compliant documentation had consumed 50% of total planned quality budget before the majority of code evaluations and testing were approached.

PROGRAM 4 is our worst case model, where everything seems wrong and the solution required replacement of staff. The programming languages were C and assembly. This program consisted of 10K of flight (RAM) software, 0.7K of flight (PROM) software, and 20K of Ground Support Equipment (GSE) software. There were four software engineers and one lead. The value of the program was approximately $22 million.

The software had a Software Development Plan (SDP) that documented the process that the engineers were to follow. This SDP was the model for programs at Martin Marietta and met the minimum standards. It consisted of the requirement for informal walkthroughs for requirements, design, and code. Unit Development Folders (UDF) were to be generated for both the flight and ground software during the requirements phase and updated with design, code, test cases and results, and problem reports. Design and coding standards were identified as well as standards for testing the software (i.e., unit, CSC Integration, CSCI testing). The program had a goal of 100 lines of code per module as part of the coding standards. Formal reviews were held for the system and software and consisted of a Preliminary Design Review (PDR) and Critical Design Review (CDR). There was a separate Acceptance Review for the software.

The software lead that started this effort was not an experienced software engineer, had no previous management position or training in software discipline. The lead was a hardware person that had done some analysis/simulation software in a lab environment and had never worked a deliverable software program. In an attempt to save money the SQA effort for this program was initiated

after the beginning of the code and unit test phase though the original proposal had called for an assurance effort from contract start. Although the SDP required that walkthroughs be held on the software, none were conducted.

There were three builds of the flight software and then the delivery to the customer site. The testing consisted of unit testing, CSC integration, and CSCI testing performed by the software engineers. Discrepancy tracking was initiated just prior to system level test. A requirement for formal testing with Quality was not levied until build 3 of the software. The ground software consisted of two formal builds and delivery of the software to the customer. Parts of the ground software were baselined as test software since the hardware needed software for test.

Figure 9 shows the schedule adherence for program 4. The program had a 14% slippage in schedule that began in the requirements phase. One of the causes for the slippage in schedule was that the program was placed on hiatus for two years in which no work was done. After those two years the program restarted but the personnel that originally worked the program were no longer available and the program had to use time to restaff and come up to speed. Another problem that caused the slippage was the unknown state of the requirements. The requirements were continually being changed by the program and the customer. Since there were no firm requirements, the design was not baselined before coding and the code continually changed.

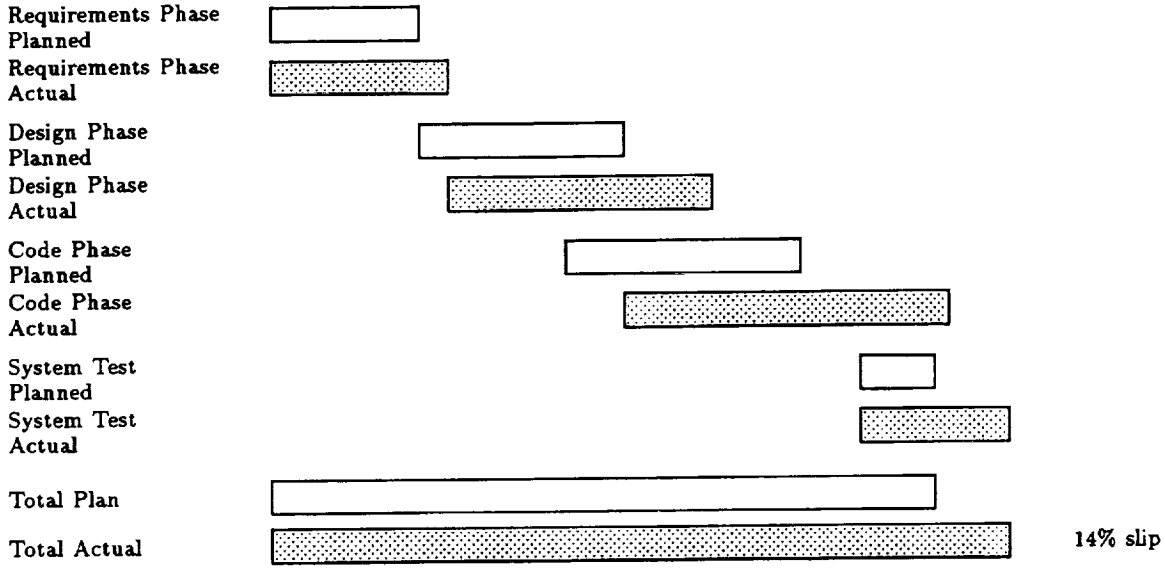


Figure 9 – Program 4 Schedule Adherence

Table 9 shows the various builds of the flight RAM software with the total lines of code for each build with the total number of modules and the average percent of comments for the CSCI. The increase in the fault density score during build 2 was a result of replacement of one of the engineers with a programmer who exercised greater test discipline.

Table 9. Flight RAM Software

	Total LOC	Average Percent Comments	Average Size Of Modules	Total Number Of Modules	Fault Density
Build 1	4665	19.11%	52	90	1.70
Build 2	7835	16.22%	61	128	6.15
Build 3	9459	19.42%	66	143	2.42
Delivery	9538	19.43%	67	142	0.84

Table 10 shows the various builds of the flight PROM software with the total lines of code for each build with the total number of modules and the average percent of comments for the CSCI for the C code. The large increase in the fault density score was directly related to the replacement of the PROM software engineer with a programmer who exercised greater test discipline.

Table 10. Flight PROM Software

	Total LOC	Average Percent Comments	Average Size Of Modules	Total Number Of Modules	Fault Density
Build 1	740	12.08%	49	15	10.81
Build 2	742	12.08%	49	15	32.35
Build 3	692	12.04%	46	15	5.78
Delivery	752	12.04%	50	15	0

Table 11 shows the various builds of the GSE software with the total lines of code for each build with the total number of modules and the average percent of comments for the CSCI. For build 1, of the 380 modules 18% of the modules had no comments at all; for build 2, 17% of the modules had no comments; and for the delivery, 18% of the modules had no comments. Only two of the files with no comments were changed after the initial baseline

Table 11. GSE Software

	Total LOC	Average Percent Comments	Average Size Of Modules	Total Number Of Modules	Fault Density
Build 1	18,624	8.1%	49	380	1.23
Build 2	19,317	7.9%	49	392	0.10
Delivery	19,002	7.8%	48	398	0

After start of system test, there were 18% priority *A* discrepancies, 23% priority *B* discrepancies, 53% priority *C* discrepancies, and 6% priority *D* discrepancies. Since the testing prior to baseline relied on the software engineers, a large number of high priority discrepancies show the lack of rigor used in unit testing. The number of priority *A* and *B* discrepancies found during system test could indicate insufficient time for the engineer to test the code in sufficient detail before turning over modified code, or it could indicate insufficient testing of the software prior to software and hardware integration.

Figure 10 shows the types of errors found throughout the program. There were several types of errors found. There were Coding Errors, Design Errors, Requirements Errors, Hardware Errors that

resulted in the software being changed, and Requirements Changes. Coding Errors represented 69% of all errors Even after the start of system test, 2% of the errors were in design or requirements.

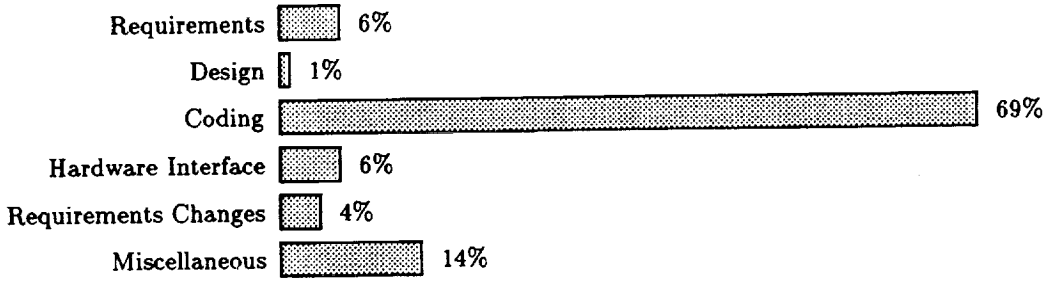


Figure 10 – Percent Errors by Type of Discrepancy

The rapid rise in Figure 11 from build 2 to build 3 is directly related to the independent test program instituted at this time. The original requirement for testing was that the software engineers perform the testing to their own comfort level. During various reviews and evaluations it was discovered by the new software lead that the level of testing was insufficient. Software had not been retested after modifications. The new software lead recommended to the customer that a more rigorous test program be implemented on the software under the cognizance of Quality. The customer agreed to this and the large number of discrepancies after build 2 is shown in the following figure.

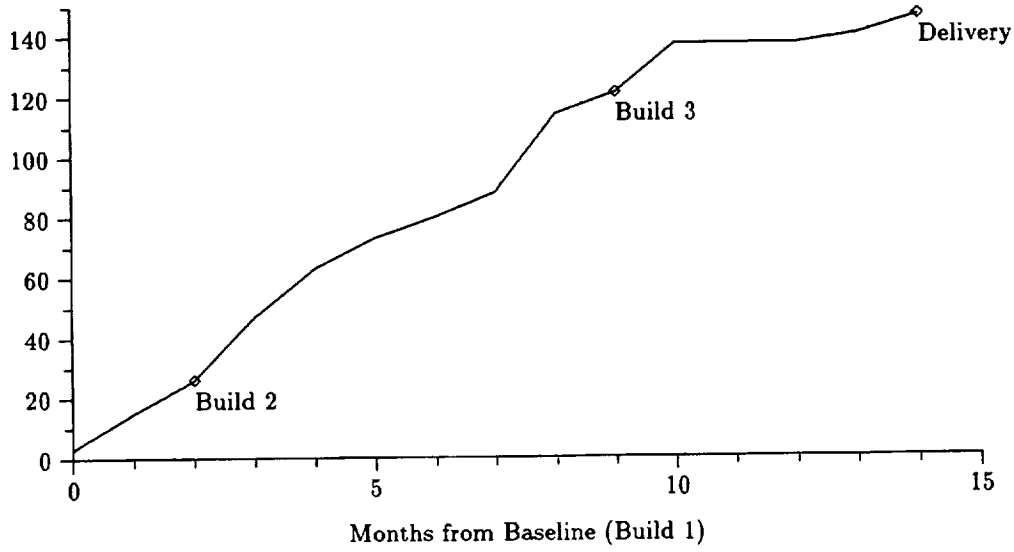


Figure 11 – Cumulative Flight Software Discrepancies

After delivery of the Flight RAM software, eight errors were discovered in the software. Of these errors one was a hardware error that resulted in the software being changed, one was a requirements error, and the rest were coding errors. Nine of the errors in Build 3 were a result of enhancements to the software requested by the customer. One of the causes for the errors in the flight software, was that the breadboard hardware used to simulate the flight hardware and used for testing the flight software had several differences between it and the flight hardware, which resulted in the software behaving differently between the two. Once the breadboard was changed to match the

flight hardware, the software was corrected to operate on the flight hardware. During the phase of Build 3 to Delivery of the flight PROM software, 18 errors were found in the software. These were coding errors with the exception of one requirements change from the customer. Four of the errors were customer requested enhancements to the software. The fault density of the flight software after delivery to the customer site was 0.78.

Table 12 shows the number of discrepancies for the GSE software by build. The lack of requirements and design errors can be directly attributed to the fact that the requirements and design were changed to match the software as a result of the software engineer not adhering to the requirements or design documents. After delivery of the GSE software, no errors were found at the customer site.

Table 12. Types of GSE Software Discrepancies by Build

Build	Coding Errors	H/W Interface Errors
Test Software	17	0
Build 1	20	1
Build 2	5	0

The software development effort resulted in an average of 12 lines of code being developed per day. There was an increase of 53% of software engineering hours from the original proposal submittal and an increase of 47% of SQA hours. The 53% increase in the software development effort was directly related to the fact that several of the software engineers and the software lead were replaced nine months prior to delivery of the software. The software did not meet schedule, the software did not work with the hardware, and none of the documentation met the documentation standards or matched the software. The problems with the engineers that were replaced entailed not testing the software sufficiently or not testing the software at all. The software lead, not having worked a deliverable software program, did not enforce the SDP and ensure that the requirements were met or that the software was tested prior to integrating the software with the hardware. The software lead was replaced as well as two of the engineers, one ground and one flight. There was an overall attitude between these three that their software was perfect, therefore there could not possibly be any errors in the software and so it did not need to be tested.

The ground software engineer exhibited a lack of regard for the established process. The ground software that was developed did not meet requirements or design and the documentation was changed to match what the engineer had done after the fact. This engineer was replaced when the new software lead found out that the software written did not meet requirements or design and that the software had not been tested because the engineer did not think the ground software was important enough to take time to test. The lack of discrepancies in the GSE software is due to the requirements and design being changed to match what the software engineer had done. The customer was apprised of the situation and agreed to changing the documentation to match the software, and waived the unit and CSC integration testing of the software to prevent a schedule impact.

The PROM software engineer had the attitude that the software did not need to be retested even if the software had been changed by more than 50%. This software engineer was replaced and the change in the fault density shows the significance of the new test program. The fault density of

the PROM software went from 10.81 to 32.35 and then back down to 5.78 with the new software engineer.

PROGRAM 5 was a medium sized project producing flight and ground software for a space experiment written in Ada and C. It had a well defined process and the study documents how much rework was involved related to programmers per their level of acceptance of the process imposed.

This program consists of 13K of Flight software and 18K of GSE software. The value of the program is approximately $20 million. The flight software is written in Ada and GSE in C. There are two leads and five programmers. The software lead has 10 years experience of software development, has not had a software lead position before but has developed deliverable software and knows the process and the importance of defining requirements and testing the software sufficiently. The software engineers that are used on the program have experience in software development but three of those engineers were replaced on program 4.

The software lead has established the rules for developing the software on this program and is ensuring that the software is developed and tested sufficiently. The software lead from program 4 has been given a position of testing the flight software on the breadboard unit and it has not been determined if the testing is sufficient. The software was required to be developed to European Space Agency (ESA) standard PSS-05-0[6] and ESA PSS-01-21.[7]

The program has an SDP which documents the process for developing the software and documentation. The SDP requires informal walkthrough for requirements, design, and code as well as formal reviews, (i.e., SRR, PDR, CDR). Software Development Folders (SDF) were required and initiated during the design phase. These contain the requirements, design, code and unit test cases. SQA's involvement with this program started at Authority to Proceed (ATP) and has continued throughout the program life cycle.

SQA participates in the various walkthroughs and reviews the requirements for traceability, testability, completeness, consistency, correctness, and understandability. SQA reviews the design for traceability of requirements, conformance to contractual requirements, compliance with design standards, completeness, correctness , understandability, and consistency. SQA reviews the code for compliance to coding standards, implementation of the design into the code, traceability of requirements, completeness, correctness, documentation of the code (i.e., comments), and consistency.

Figure 14 shows the breakout of the schedule for program 5 and the 17% slip in schedule. A slip in the schedule started in the design phase as a result of customer changes and impacted the Preliminary Design review date but was mostly made up in the Critical Design phase. The coding effort has also experienced a slip due to changes in design and additional requirement changes. This program will soon enter system test.

[6] ESA PSS-05-0 ESA Software Engineering Standards.
[7] ESA PSS-01-21 Software Product Assurance Requirements for ESA Space Systems.

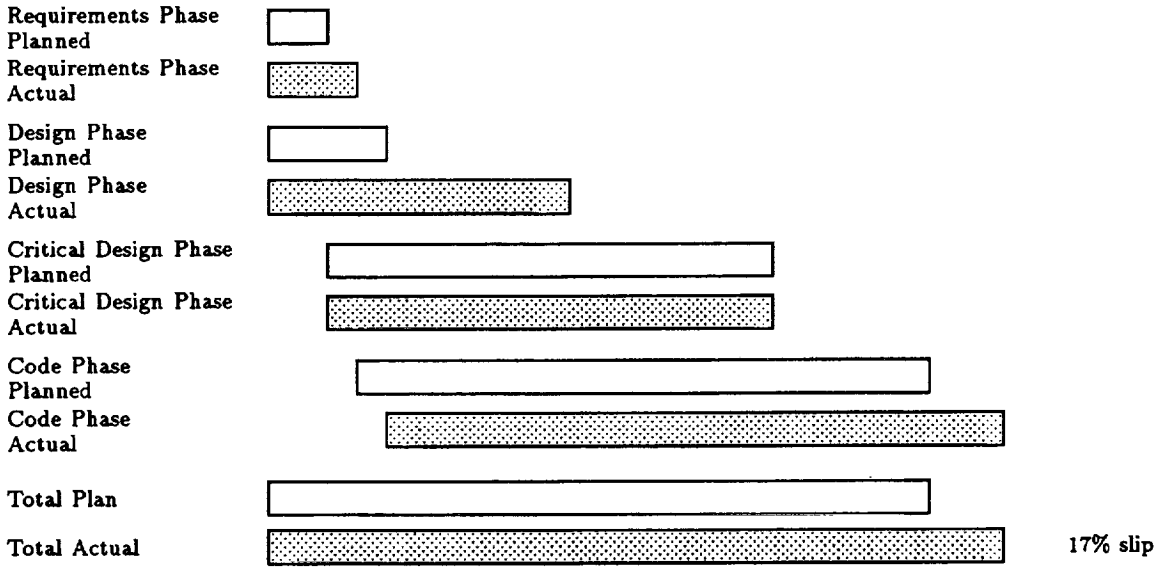


Figure 14 – Program 5 Schedule Adherence

As shown in Figure 15, the majority of the requirements errors to date were found during the requirements phase.

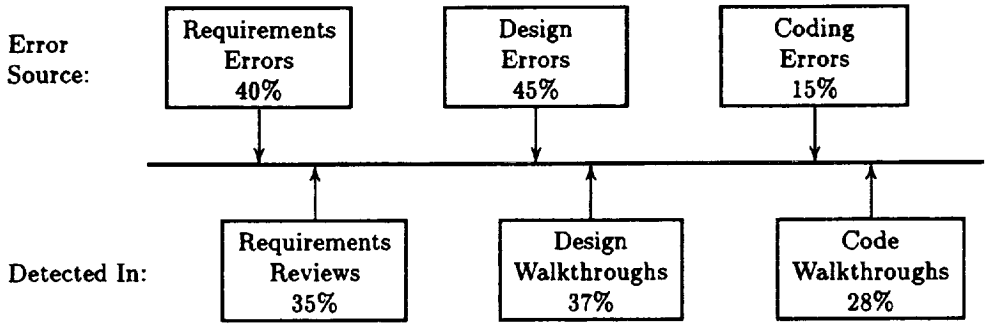


Figure 15 – Percentage of Errors Found by Phase

So far, this program seems to be on the right track. The only question mark is the fact that three people who demonstrated problems in adherence to process on program 4 are on this program. As shown in the following table, the average time to close action items of one of those inherited programmers is double the others.

Table 13. Programmer Responsiveness

Programmer	W/T Action Items	LOC	Average Time To Close
P1	49	9325	4 days
P2	31	1190	2 days
P3	70	1449	2 days
P4	45	3227	2 days
P5	10	1120	2 days

One flight programmer from program 4 seems to have learned the lesson from being replaced and has exhibited a different attitude throughout this program. The engineer has placed a great deal of emphasis on retesting software that has changed and when the customer brought up the issue if this testing was necessary, this engineer emphasized the importance of testing changes to the software. The ground support software engineer does not seem to have learned the lesson for implementing the requirements and design and of testing the software. The software lead has had to intervene more often to gain compliance to the process from this engineer. The other experienced engineers — ground and flight — have been working to the process documented in the SDP. The process for this program seems to be working in relation to the walkthroughs, implementation of requirements and design, and testing of the software. The software is still a little behind schedule but that is partially due to the hardware requirements changing.

The evaluation of the flight software requirements specification shows the evolution of the document for the program. The requirements document is reviewed throughout the software development cycle. Table 14 shows the results of the document reviews performed by SQA during the requirements, design (PDR and CDR), and code phases. The requirements document was first reviewed during the requirements phase and was found to be unacceptable due to the lack of testability and traceability of the requirements. The baseline of the document during the requirements phase was an improvement. The document continued to evolve throughout the development cycle and improved slightly with each new revision. This is due to the fact that the requirements were understood better than in the requirements phase and the hardware requirements were more firm than in the requirements phase of the software.

Table 14. SRS Document Completion Index

Phase	DI Score (1.0 high)
Requirements	.50
Requirements (Baseline)	.60
Design (PDR)	.65
Design (CDR)	.73
Coding	.75

PROGRAM 6 was an engineering support task for one of the NASA centers. It is used to show the difference in performance of the programmers involved based on their past experience with software engineering discipline. No contract criterion was available but each was totally responsible for code to work on a particular platform. This program was a small research program that developed software for a power system. The program consisted of a program manager, a software lead that also acted as the deputy program manager and a programmer, and two software engineers. The software lead (P1) was fresh out of college with a masters degree and had never worked a program before, one of the engineers (P2) was fresh out of college but had worked as a summer hire for Martin Marietta, and the last engineer (P3) had several years experience in developing deliverable software. The software was developed primarily in Lisp on a Unix based machine with some software developed on a PC in Pascal. The software was divided into two CSCIs, application software which totaled 116,000 lines of code and lower level processor (LLP) software which totaled 5800 lines of code. The program had a series of change orders which documented the requirements changes to the software and the hardware.

Although this program was a small research program, Martin Marietta has minimum standards that all software programs are required to meet. This consists of a software development plan, documented requirements and design, testing, and configuration control. The program generated a SDP that reflected how the program was going to operate while still meeting the minimum Martin Marietta requirements. A requirements document was generated for the software and maintained through the life of the program. The design for the software was documented in monthly progress reports to the customer. The software was tested at the system level but did not include lower level functional testing. A formal software test was run at the Martin Marietta facility prior to delivery and system test at the customer site. The level of testing was determined by the engineer. There were no coding standards per se, therefore style was dependent on the individual software engineer.

There were no slips in the schedule for program 6 due to the program's statement of work being written such that whatever software was developed at the time of delivery was what the customer accepted. As long as the software had the required functionality, the customer was satisfied.

Table 15 shows the two CSCIs with the total lines of code, the average size of a module and the fault density of the software. The majority of the application software was developed by programmers P1 and P3. The LLP software was developed by P2. The difference in fault density of the two types of software shows the level of unit testing that was performed by the engineers. Programmer P2's software was not sufficiently tested prior to baselining. There is a significant difference between the two engineers, P1 and P2, although they were both new graduates from college when they first started on the program.

Table 15. Program 6 Software

Software	Total LOC	Average Size Of Modules	Total Number Of Modules	Fault Density
Application	116,712	451	259	1.15
LLP	5,857	345	17	3.07

There were 152 discrepancies found in the software of which 12% were requirements changes by the customer and 88% were coding errors.

Table 16 breaks out the errors by priority. The *A* priority can be requirements errors/changes or crashes in the software.

Table 16. Program 6 Percent Discrepancies by Priority Level

Priority	Percent
A	13%
B	13%
C	70%
D	4%

Programmer P2 moved from the LLP software to the Application software when the development was done. When the total number of crashes (*A* priority) are reviewed 53% of the crashes (software that was not tested sufficiently after modification and incorporated into the system) were introduced by programmer P2.

As shown in Figure 16, the majority of the errors were found during engineering test. A large portion of the discrepancies were requirements changes requested by the customer. This resulted in a fault density of 0.05 after delivery of the software to the customer and 1.24 for the program overall.

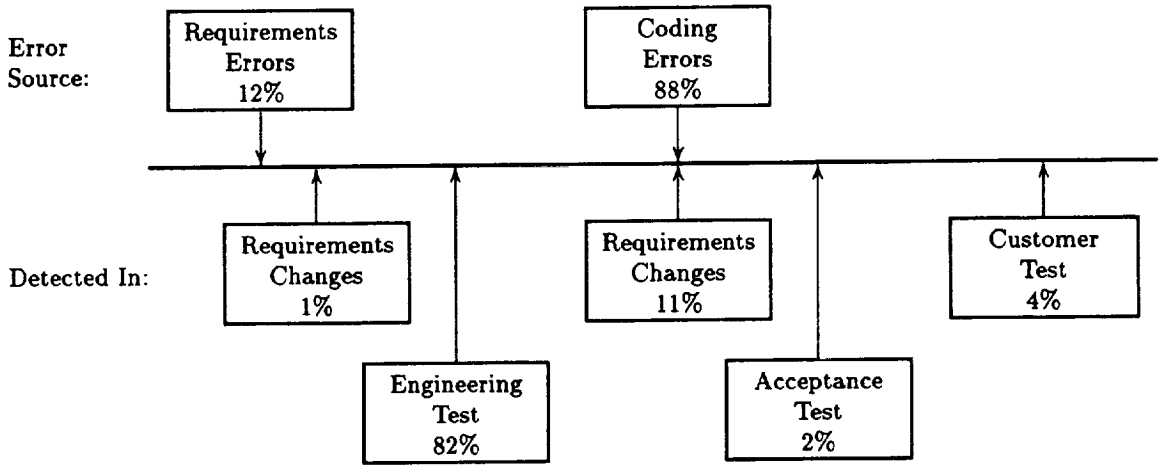


Figure 16 – Program 6 Percentage of Errors

For program 6, there were 122,569 lines of code developed over a five year span. This software changed drastically from the initial development effort because of change of platforms. The software originally started with a Xerox computer, a VME-10, and a Symbolics. The Xerox software was rehosted on a Solbourne, the VME-10 on a 386, and the Symbolics on the Solbourne. The lines of code developed per day could not be computed since the software effort was redefined through requirement changes including rehost efforts for developed software.

The difference in programmer discipline is shown on this program through the fault density measure. The software written by programmers P1 and P3 had a fault density of 1.15 and code written by programmer P2 had a fault density of 3.07. This difference is significant because of the large number of lines of code in the application software that were rehosted compared to the other software. The LLP software rehosted was 5.8K lines of code compared to 116K. Although the process defined for this program was a minimum set of standards, the programmers implemented the process differently. Programmers P1 and P3 were more conscientious in testing of their software than programmer P2. This was reflected in the number of *A* and *B* priority discrepancies found in the application software after programmer P2 moved to the application software. The majority of the rehosting of the application software was complete before programmer P2 moved over. Programmer P2 introduced double the number of *A* and *B* priority discrepancies as the other two programmers.

Table 17. Program Comparisons

Item	1	2	3	4	5	6
Involvement with Process Definition	YES	NO	NO	NO	YES	N/A
Stable Requirements	YES	YES	NO	NO	NO	YES
Adequate Unit Test	YES	NO	NO	NO	YES	NO
Quality Oversight for Complete Program	YES	NO	YES	NO	YES	NO
Schedule Slip	NO	30%	23%*	14%	17%	NO
Engineering Overrun	NO	36%	*	53%	NO**	NO
Quality Overrun	NO	10%	*	47%	NO**	NO
Planned Quality Budget versus Engineering	5%	13%	14%	12%	10%	2%
Actual Quality Budget versus Engineering	5%	10%	*	14%	**%	2%
Productivity Planned	6	20	9	22	7	25
Productivity Actual	6	16	*	12	**	25

* Program cancelled, data unavailable
** Program in process of completing

1. Following the Process

In all these programs a development plan was required by company standards and, on some, by contract requirement. Those programs that participated in defining the contents of the plan were more likely to adhere to it. Participation certainly strengthens understanding the contents which goes a long way toward following the process defined. Programmers accustomed to meeting standards adjust to new process definitions and respond to changes in their development environment by getting on with the job.

It is helpful to establish early who follows the process, so that additional resources can be brought to bear to gain adherence and reduce rework and action items.

Program 3 was, in the experience of the authors, a classic example of how direction from the software leads can effect the adherence to process. The worst adherence to schedule was shown by the group under the guidance of what we would describe as a whiner who wanted no rules, no oversight, and no process. The best adherence and best response for time to fix was demonstrated by the group directed by a person who expected adherence to schedule and expected the group to follow the agreed upon methodology as a team.

Having the people involved in the work responsive to the process is superior to bringing in the heavy artillery, i.e., upper management, to dictate compliance.

Program 6 is an example of what individuals bring to the job. This small program had little structure, no documentation other than task descriptions of a very high level, and the individual

programmers chose their own method of test and evaluation of the results. One set of software had a significantly higher fault density than the other, though neither rating was excessive. The programmer who produced code with the higher fault density also had the most errors causing system crashes. The programmer producing code with the lower fault density approached the development of the application in a structured manner going from requirement to design to code and test. The other sat at the terminal and wrote code till it appeared to work.

2. Changing Requirements

As discussed under programs 3, 4, and 5, changing and/or nebulous requirements handicap a program. Program 3 overcame poor software requirement definition by establishing preliminary design based on the system specification and continuing to work the software requirements through the design and coding phase. This represented the design and development personnel overcoming the process. The reluctance of the requirements group to meet their responsibilities because of an overall program problem of requirements flux was a losing situation. The process dictated that this group produce and it did in volumes of bad documentation. Eventually, the responsibility for the product migrated to a more focused group under the control of the end users that allowed the evolution of an acceptable product.

Program 4 requirement changes came mostly from the customer and usually involved functionality in the ground support software. Such changes are more easily accommodated than hardware or design change and had little impact on delivery of the flight unit. A problem on program 4 involved not keeping the breadboard used to test the software current with flight hardware design changes. This caused considerable consternation and finger pointing when system integration testing could not start because the software would not run the hardware. The lack of communication and understanding of need was corrected by replacing the software lead with someone experienced in development of software for an embedded system.

Program 5 still has time to overcome the program slip introduced in its design and coding phase caused by requirement changes.

Performing document evaluations and reporting the document completion index provides visibility into progress toward meeting program goals. This is especially important with the requirements document since inadequacy in it is felt through the following phases.

3. Discipline in Informal Testing

Informal test lacking objective goals accomplishes little but can give a false sense of security. Programs 2 and 4 suffered from too little structured unit testing. This is shown in the large jump in defects found when test philosophy and responsibility changed. Both programs experienced considerable engineering overruns and schedule slip.

Program 2 personnel adjusted to the new test philosophy and informally competed with each other for best time to fix and fixing it right the first time. Program 4 personnel were so burdened with ego that to accomplish adequate unit test and institute correctness of fixes programmers had to be replaced.

On Program 3, the considerable number of slips in scheduled code walkthroughs and baselines for the ground CSCI were caused by additional coding effort needed to complete informal integration

of the CSC into the CSCI. Except for the leads on the ground CSCIs, the programmers were used to developing stand alone code and lacked an understanding of meeting interface requirements.

The unit test on program 6 was defined and executed by the individual programmers involved. As reported previously, lack of rigor in unit testing surfaced in system test as more software crashes were introduced by one of the programmers than the total for the other two.

4. Reviewing Early

Trying to save budget by avoiding the use of early objective reviews as shown in program 4 can be self defeating. An inspection or review process that can point out errors early but not effect their correction, as shown on program 3, is a poor process in terms of cost and quality. The thorough review process used on program 1 was integral to keeping on a very tight schedule.

5. People

Pinpointing who is most likely to cause rework and frustration is an effective way to control rework. It is a matter of tracking number of action items and response to action items. If a problem with responsiveness or understanding of the importance of compliance to the process is identified, management must accept the responsibility of replacing the problem, redefining standards, or strong arming compliance.

6. Collecting Data

Using a database tool to pool data makes sense. However a project evolves, it will have reviews/evaluations of some type, discrepancy reporting and status to schedule. Collecting the result of objective reviews and other defect data should be a high priority for complete quality records. Current status of development progress is needed to flag problem areas and replan work to make up for known slips. The tool used to collect the information on these programs, SQUID, was designed around the Air Force pamphlet, Software Management Indicators, Management Quality Insights, Air Force Systems Command, 20 Jan 1987 and practical knowledge based on twelve years of performing software quality on projects. Reporting from a database is superior to digging through data retention boxes. Use of a tool can give structure to multiple quality programs and allow meaningful comparisons between different types of projects.

Each program undertaken yields a better idea of what is a meaningful measure. Program 2 was the first program that the authors had the opportunity to collect metrics on during the life of the program. Defect density and coding complexity measures were used because involvement began after coding start. After this experience the authors started looking more at early fault detection via walkthroughs and reviews. This in turn lead us to look at responsiveness of individuals in correcting deficiencies as a major driver in software quality.

In summary, we believe our observations support the conclusion that good programmers produce good code. A good programmer in the context of this paper is a programmer who is committed to project goals, highly disciplined, and responsive to constructive criticism that is based on meeting those goals. Of itself, a process does not make a product. The best a process definition can do is let producers know what is expected before they are evaluated for method and output. Proper execution of a process requires the cooperation of the participants. The more readily this cooperation is given the less the cost of rework.

Software Quality: Process or People

Regina Palmer
Martin Marietta Astronautics
P. O. Box 179, M/S S1008
Denver, Colorado 80201
&
Modenna LaBaugh
R&FC Group
3076 So. Hurley Circle
Denver, Colorado 80227

12/2/93 GSFC Software Engineering Workshop

Software Quality: Process or People

- Agenda
  - Introduction
  - Background of Programs
    - Process
    - Success
    - Problems
  - Lessons Learned

12/2/93 GSFC Software Engineering Workshop

Software Quality: Process or People

- Introduction
  - The Paper Describes the Following for Each Study
    - Involvement with process definition
    - Stability of requirements
    - Thoroughness of unit and system test
    - Degree of Quality oversight
    - Schedule Adherence
      Variance from planned completion dates
    - Overruns – Engineering and Quality
    - Planned Quality Budget as percent of Planned Engineering
    - Actual Quality Budget as percent of Actual Engineering
    - Planned Productivity
      Lines of code per engineering day
      Includes Program Management, Engineering, and Quality hours
      Excludes Business Operations and Property Management — not readily available to authors
    - Actual Productivity

Software Quality: Process or People

- Program 1
  - Very involved with process definition.
  - Very stable and well defined requirements. Clearly defined end of phase before beginning of next. As shown by the high Document Completion Index scores.

Phase	DI Score (1.0 high)
Requirements	.67
Design (CDR)	.77
Coding	.83

  - Thorough unit and system test
  - Quality oversight from beginning of contract
  - Schedule Adherence – no slip
  - Overruns – none
  - Planned Quality Budget as percent of Planned Engineering – 5%
  - Actual Quality Budget as percent of Actual Engineering – 5%
  - Planned Productivity – 6 LOC
  - Actual Productivity – 6 LOC

Software Quality: Process or People

- Program 2
  - Little involvement with process definition. Programmers were unfamiliar with testing rigor and programming standards. Table is in order of programming experience.

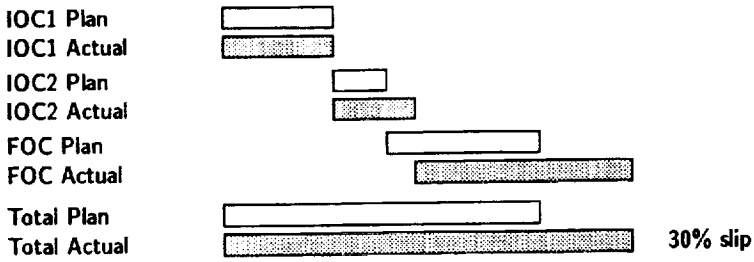
Programmer	Fault Density of Code Produced
P1 - sw lead	8
P2 - sw lead	18
P3 - new graduate	10
P4 - new graduate	8
P5 - new graduate	17

  - P2 and P5 were reluctant to take time to test. P4 was used to do most correction of P2 code and P3 was used to correct P5 code because of the low fault density of code they wrote.
  - Disagreement late in program on requirements
  - Quality oversight on program from beginning of coding

12/2/93 GSFC Software Engineering Workshop

Software Quality: Process or People

- Program 2 (continued)
  - Lack of discipline in unit test – 60% of errors found in final testing were in the first baseline.
  - Schedule Adherence – 30% slip in final delivery of software. Legacy code required upgrades to meet customer expectation of usability that was not anticipated in contract bid.



12/2/93 GSFC Software Engineering Workshop

Software Quality: Process or People

- Program 2 (continued)
  - Overruns
    - Engineering exceeded plan by 36%
    - Quality exceeded plan by 10%
  - Planned Quality Budget as percent of Planned Engineering – 13%
  - Actual Quality Budget as percent of Actual Engineering – 10%
  - Planned Productivity – 20 LOC
  - Actual Productivity – 16 LOC

Software Quality: Process or People

- Program 3
  - No involvement with process definition except for defining coding standards
  - Very unstable and poorly defined requirements. Document completeness scores for documents highlight this.

Phase	SRS	SDD	STP	Project
PDR	.17	.55	.50	.40
CDR	.36	.74	.61	.57
Coding	.64	.74	.61	.66

  - Unit test had only begun at contract close
  - Quality oversight from beginning of project but no authority for action item resolution

CSCI Name	Number of Items	Average Response Time (days)
FLIGHT	24	33
EGSE	13	46
TRAINER	5	60
SIMULATOR	10	83

Software Quality: Process or People

- Program 3 (continued)
  - Schedule Adherence – 23% slip due to requirement definition during design phase
    - Attitude of software leads to maintain internal schedules varied.

CSCI NAME	W/T Dates Missed	Average Slip (days)	B/L Dates Missed	Average Slip (days)
FLIGHT	10%	19	23%	15
EGSE	26%	28	43%	21
TRAINER	14%	7	0%	0
SIMULATOR	18%	28	18%	14

  - Overruns
    - Engineering – Program cancelled, unable to compute
    - Quality – at beginning of code phase had exceeded evaluation budget by 173%.
  - Planned Quality Budget as percent of Planned Engineering – 14%
  - Actual Quality Budget – Program cancelled, unable to compute
  - Planned Productivity – 9 LOC
  - Actual Productivity – Program cancelled, unable to compute

Software Quality: Process or People

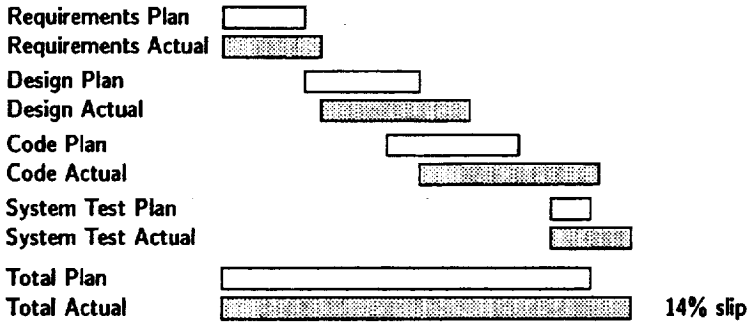
- Program 4
  - No involvement with process definition
  - Unstable requirements – 33% of change traffic for the last build before delivery were enhancements requested by the customer.
  - Lack of test discipline in unit test – Large number of errors found after build 1 when other programmers were brought in to validate software.

	Number of Defects
Build 1	16
Build 2	72
Build 3 thru Delivery	35

  - Quality oversight begun in coding phase
  - Overruns
    - Engineering – 53%
    - Quality – 47%
  - Planned Quality Budget as percent of Planned Engineering – 12%
  - Actual Quality Budget as percent of Actual Engineering – 14%

Software Quality: Process or People

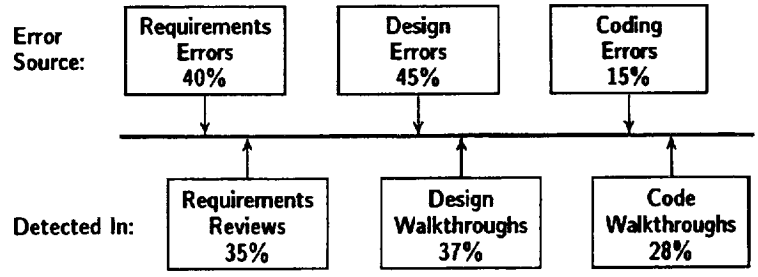
- Program 4 (continued)
  - Schedule Adherence – 14% slip



  - Planned Productivity – 22 LOC
  - Actual Productivity – 12 LOC

Software Quality: Process or People

- Program 5
  - Very involved with process definition. Reviews used in process very successful for finding errors early.



Software Quality: Process or People

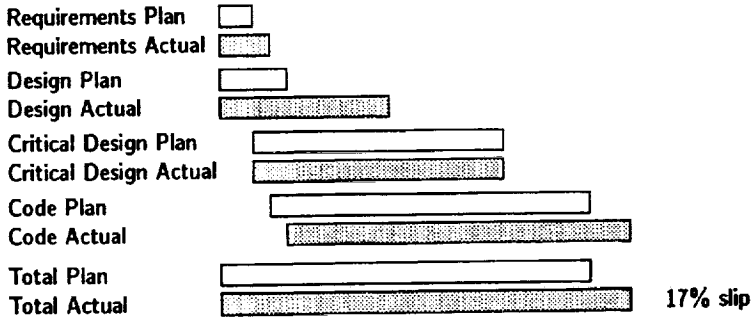
- Program 5 (continued)
  - Responsiveness of individual programmers to the corrective action process is shown in the table.

Programmer	W/T Action Items	LOC	Average Time To Close
P1	49	9325	4 days
P2	31	1190	2 days
P3	70	1449	2 days
P4	45	3227	2 days
P5	10	1120	2 days

  - Stability of requirements – requirement changes were imposed during the design phase impacting Preliminary Design schedule but time was made up during critical design. Hardware requirements changed after design baseline.
  - Thoroughness of unit test due to stress of software lead
  - Quality oversight from beginning of program
  - Planned Quality budget as percent of planned Engineering – 10%
  - Actual Quality Budget as percent of actual engineering (to date) – 10%

Software Quality: Process or People

- Program 5 (continued)
  - Overruns – Program has not completed code phase yet, not calculated to date
  - Schedule Adherence – 17% slip appeared in code phase, to be made up in test



  - Planned Productivity – 7 LOC
  - Actual Productivity – Program not due to complete till next year

Software Quality: Process or People

- Program 6
  - No process definition
  - Requirements changes caused new work order
  - Unit test performed at a level defined by the individual programmer. The difference in testing thoroughness is shown in the fault density of the software as measured by discrepancies found during system test.

Software	Total LOC	Average Size Of Modules	Total Number Of Modules	Fault Density
Application	116,712	451	259	1.15
LLP	5,857	345	17	3.07

  - Quality provided configuration control only for delivery to customer
  - Schedule Adherence – as a level of effort contract the schedule is always met
  - Overruns – N/A, level of effort
  - Planned Quality Budget as percent of Planned Engineering – 2%
  - Actual Quality Budget as percent of Actual Engineering – 2%
  - Planned Productivity – 25 LOC
  - Actual Productivity – 25 LOC

Software Quality: Process or People

- Lessons Learned
  - Process definition must involve the participants to assure acceptance.
  - People who do not accept the process cause rework expense. Those likely to cause rework or delay are identifiable by tracking action items and response times.
  - Programmers must be made aware of objective goals for unit testing.
  - Fault density by itself is a deceptive measure. Using it, program 2 was estimated to be on schedule two months before the final delivery. Before end of test it became obvious that the unit test program had been inadequate and a 7 month slip occurred.
  - Programs which only collect metrics during test miss opportunities for early detection of problems.
  - Program knowledge disappears soon after each milestone on a program unless someone collects it as it happens.
  - Each program undertaken yields a better idea of what is a meaningful measure within a company culture.
  - Data for this paper were scattered amongst individuals involved on the programs reported and not readily available till input into the database tool used.

Software Quality: Process or People

Program Comparisons

Item	1	2	3	4	5	6
Involvement with Process Definition	YES	NO	NO	NO	YES	N/A
Stable Requirements	YES	YES	NO	NO	NO	YES
Adequate Unit Test	YES	NO	NO	NO	YES	NO
Quality Oversight	YES	NO	YES	NO	YES	NO
For Complete Program						
Schedule Slip	NO	30%	23%*	14%	17%	NO
Engineering Overrun	NO	36%	*	53%	NO**	NO
Quality Overrun	NO	10%	*	47%	NO**	NO
Planned Quality Budget To Engineering	5%	13%	14%	12%	10%	2%
Actual Quality Budget To Engineering	5%	10%	*	14%	**%	2%
Productivity Planned	6	20	9	22	7	25
Productivity Actual	6	16	*	12	**	25

* Program cancelled, data unavailable

** Program in process of completing

12/2/93 GSFC Software Engineering Workshop