Building an Experience Factory for Maintenance

Jon D. Valett

Software Engineering Branch
NASA Goddard Space Flight Center
Greenbelt, Maryland 20771

Steven E. Condon

Computer Sciences Corporation
10110 Aerospace Road
Lanham-Seabrook, Maryland 20706

Lionel Briand, Yong-Mi Kim, Victor R. Basili

Department of Computer Science
University of Maryland
College Park, Maryland 20742

Abstract

This paper reports the preliminary results of a study of the software maintenance process in the Flight Dynamics Division (FDD) of the National Aeronautics and Space Administration/Goddard Space Flight Center (NASA/GSFC). This study is being conducted by the Software Engineering Laboratory (SEL), a research organization sponsored by the Software Engineering Branch of the FDD, which investigates the effectiveness of software engineering technologies when applied to the development of applications software.

This software maintenance study began in October 1993 and is being conducted using the Quality Improvement Paradigm (QIP), a process improvement strategy based on three iterative steps: understanding, assessing, and packaging. The preliminary results presented in this paper represent the outcome of the understanding phase, during which SEL researchers characterized the maintenance environment, product, and process.

Findings indicate that a combination of quantitative and qualitative analysis is effective for studying the software maintenance process; that additional measures should be collected for maintenance (as opposed to new development); and that characteristics such as effort, error rate, and productivity are best considered on a "release" basis rather than on a project basis. The research thus far has documented some basic differences between new development and software maintenance. It lays the foundation for further application of the QIP to investigate means of improving the maintenance process and product in the FDD.

Introduction

Goddard Space Flight Center (GSFC) manages and controls NASA's Earth-orbiting scientific satellites and also supports Space Shuttle flights. For fulfilling both these complex missions, the Flight Dynamics Division (FDD) developed and now maintains over 100 different software systems, ranging in size from 10 thousand source lines of code (KSLOC) to 250 KSLOC, and totaling 4.5 million SLOC. Of these systems, 85% are written in FORTRAN, 10% in Ada, and 5% in other languages. Most of the systems run on IBM mainframe computers, but 10% run on PCs or UNIX workstations.

The Software Engineering Laboratory (SEL) has been researching and experimenting in the FDD since 1976 with the goal of understanding the software development process in this environment; measuring the effect of software engineering methodologies, tools, and models on this process; and identifying and applying successful practices (Reference 1). The SEL has developed an approach to process improvement known as the Quality

Improvement Paradigm (QIP) and has established a supporting organizational structure, the Experience Factory, for maintaining the experience base, which is a key element of this work. These concepts, and their application specifically in this study of software maintenance are described in detail in Sections 1 and 2 of this paper.

One of the key features of this research is the combination of qualitative and quantitative approaches used to characterize the current practice of software maintenance in the FDD. These methods affected the design of the experience base developed for the study, by influencing which maintenance products and projects would be examined and which specific measures would be collected. The structure of the study is described in Section 3. Sections 4 and 5, respectively, elaborate on the qualitative analysis of the maintenance process and the quantitative analysis of the product and process characteristics. Section 6 discusses lessons learned and early recommendations for process improvement, and Section 7 poses questions that will guide future direction for this research.

1. The Quality Improvement Paradigm

The QIP is a three-step iterative process that provides an organization with a framework for continuously improving its methods of doing business. These steps–*understanding*, *assessing*, *packaging*–are shown in Figure 1.

The QIP begins with *understanding*, because before an organization can begin planning for improvement, it must thoroughly understand its current processes, products, and environmental characteristics. At the current time, the FDD maintenance study is completing its first pass through this step.

During the second phase of the maintenance study, corresponding with the *assessing* step of the QIP, improvement goals will be set, experiments conducted, and their results assessed. The experiments will test new methods or tools that show promise of helping this organization achieve its improvement goals. If these experiments demonstrate significant improvements in the process or

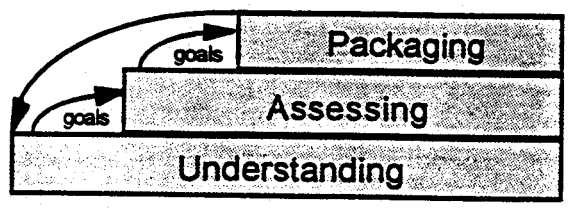

Figure 1. Quality Improvement Paradigm

products, these lessons will be incorporated into the overall FDD organization.

This third and final phase of the QIP, the *packaging* step, requires significant investment to truly capitalize on the time and money spent in the understanding and assessing steps. It may require developing new standards as well as implementing and fielding comprehensive training in these new standards.

After completing the packaging step, researchers will baseline the new process by returning to the understanding step, to verify the positive effect of process evolution on the system. Thus begins a new iteration of the QIP.

1.1 The QIP and Software Development Projects

The QIP has been used many times within the SEL to investigate the potential of new tools or processes on software development projects. In its more detailed application, the QIP consists of six steps (Reference 2):

1. Characterize the current project and its environment with respect to models and measures. Begin by characterizing the development project relative to the environment. What kind of product is being developed? How large is the project? What is the schedule? How is the project similar to and different from previous projects? This is used to provide models of similar experiences from similar projects.
2. Set quantifiable goals for successful project performance and improvement. Is the goal to shorten cycle time, reduce errors, achieve higher software reuse?

3. Choose an appropriate process model and supporting methods and tools for this project. Choose processes for the project that show promise of achieving the stated goals based upon past experience with projects of this type. Identify projects with similar characteristics and similar goals.
4. Execute the processes, construct the products, collect and validate the prescribed data, and analyze them to provide real-time feedback for corrective action.
5. Analyze the data to evaluate the current practices, determine problems, record findings, and make recommendations for future project improvements.
6. Package the experience as updated and refined models and other forms of structured knowledge gained from this project and prior projects. Save it in an experience base to be reused on future projects.

1.2 The QIP and Software Maintenance

For maintenance, the implementation of the QIP is slightly different, because past releases of the same project provide additional experience. The underscored phrases below indicate maintenance-specific foci of the QIP.

1. Characterize the current project release and proposed set of modifications and its environment.
2. Set quantifiable goals for successful project performance and improvement and the future evolution of this product. Remember that this release will soon be followed by another release and yet another release.
3. Choose an appropriate process model and supporting methods and tools for this project based on both domain class and specific product knowledge. When studying maintenance, there is an advantage over applying the QIP to new development projects because knowledge and experience are available about this specific product.
4. Execute the processes, construct the products, collect and validate the prescribed data, and analyze them to provide real-time feedback for corrective action, including real-time preventive maintenance on the current project.
5. Analyze the data to evaluate the current practices and their effects on this product. Characterize the current product, determine problems, record findings, and make recommendations for this product and future project improvements.
6. Package the experience as updated and refined models and other forms of structured knowledge gained from this project and prior projects. Save it in an experience base for future projects and the evolution of this product.

2. The Experience Factory

The SEL researchers and database team act as an *experience factory* for the software developers in the FDD (Reference 3). The experience factory organization is separate from the project organization. It serves the project organization by analyzing and synthesizing knowledge into models that support the improvement of software development (see Figure 2). It does so by concentrating on the analysis and packaging activities of the QIP, while the project organization focuses on developing the software. The project organization supplies process and product data to the experience factory and carries out experiments under the guidance of the experience factory team. The experience factory collects and analyzes the data from the project organization. It stores these data and analyses in an experience database. It also packages the best of these experiences into products, guidelines, and models, which it feeds back to the project organization to help improve its process.

The experience factory for maintenance operates the same as the experience factory for development, with three differences: First, the experience factory for maintenance

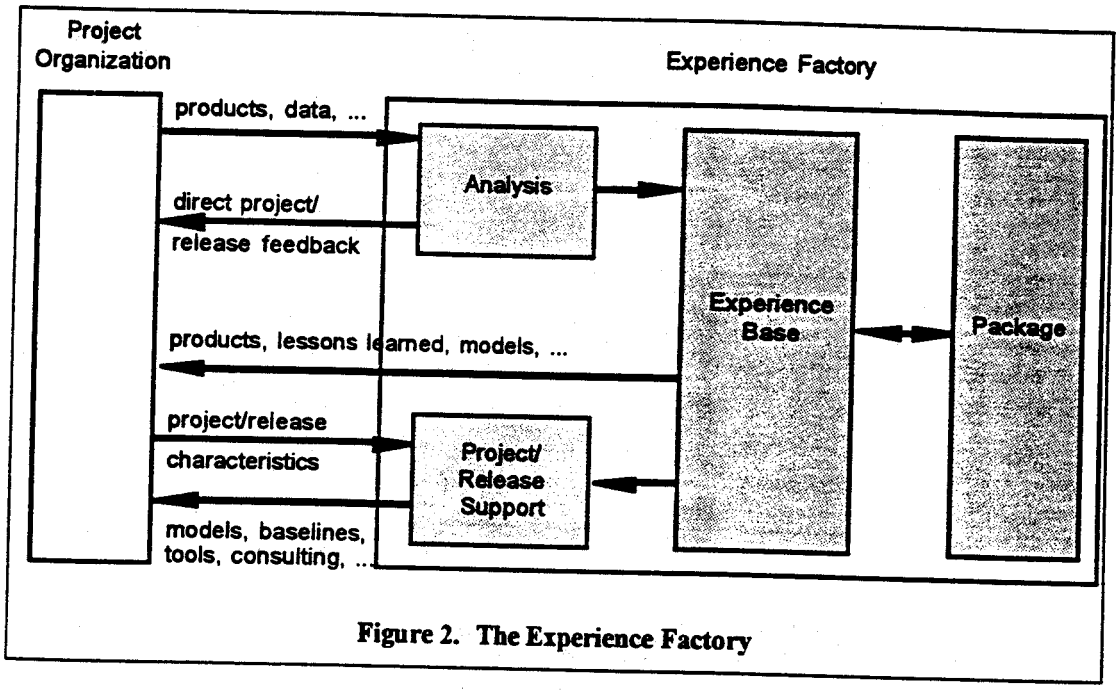

Figure 2. The Experience Factory

must address releases. Second, analysis for release feedback requires quicker response; development life cycles are on the order of 18–24 months, whereas maintenance release cycles are on the order of 6 months. Third, software maintenance emphasizes product evolution more than software development does, so experience includes past experience on the same project.

3. Building the Experience Base for Software Maintenance

Because there are many similarities between software development and software maintenance, the SEL experience of software development was used as a starting point for understanding maintenance. The measurement program for maintenance was modeled on the measurement program that is used for understanding software development. This influenced both the goals that were set and also the specific data that were identified for collection. To characterize the process, data were collected on maintenance effort distribution by activity, similar to the measures collected for new development, with some tailoring for maintenance-specific activities. To characterize the products, data were collected on a number of measures, including the amount of code modified for a release and the number of errors introduced by the maintenance work. The specific measures are discussed in more detail below.

The study team consisted of a team leader from NASA, three researchers from the University of Maryland, and one researcher from Computer Sciences Corporation. The team leader drew up the initial study plan containing the overall goals, the specific questions to be answered, and the list of maintenance measures to be collected for analysis. Data were collected on eleven maintenance projects. In addition, researchers closely monitored four of these projects and stayed in close contact with the maintenance teams on those projects. The entire study team met regularly throughout the study to refine the study plan and assess progress. These meetings also resulted in some revisions to the collected measures.

Following the lead of Lionel Briand, one of the University of Maryland researchers, a general qualitative analysis methodology was adopted, tailored, and applied to the four closely monitored maintenance projects (Reference 4). This methodology provided an objective but qualitative project characterization that complemented the quantitative

characterization that was provided by the measurement data. By supplying the researchers with a characterization of the organization structures, processes, issues, and risks of the maintenance environment, the qualitative analysis also helped them refine the data collection measures. In return, the quantitative data helped researchers to understand the qualitative data. This qualitative analysis methodology also provided a process for determining the causal links between maintenance problems, on the one hand, and flaws in the maintenance process or maintenance organization, on the other hand. The following two sections describe the combined qualitative and quantitative approach in detail.

4. Six-Step Process to Qualitative Understanding

The qualitative analysis methodology consisted of six steps, depicted in Figure 3. Researchers accomplished each step by reviewing release documents and process description documents, and also by interviewing maintenance team members.

Steps 1 through 3 provided an understanding of the maintenance organization and the release process followed by the project. With this information for several projects, researchers were able to draw comparisons between projects and to check each project for adherence to maintenance policies. Steps 4 through 6 provided the mechanism for identifying where problems existed for each project and for demonstrating flaws in the maintenance organization or the maintenance process (as followed by the project).

4.1 Understanding Steps (1-3)

Step 1 called for identifying the organizational entities involved in the maintenance process. Researchers identified distinct teams, their roles, and the information flows among these teams. For example, for each project, release approval passed from the configuration control board to the maintenance team.

In Step 2, researchers identified the phases of the release process and the major milestones that bounded these phases. For example, the *change analysis phase* culminated in the Release Contents Review meeting, and the *solution analysis & design phase* culminated in the Release Design Review meeting.

Step 3 required identifying the activities involved in each phase. Researchers selected a list of generic maintenance activities and

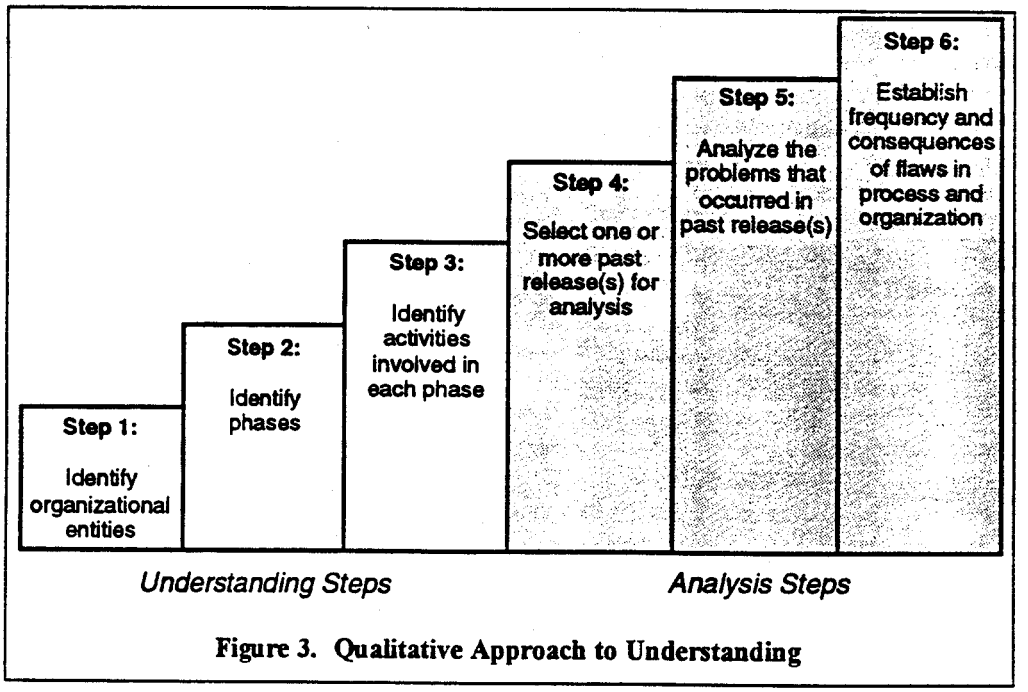

Figure 3. Qualitative Approach to Understanding

mapped them into the various phases identified in Step 2. In Step 3, researchers also identified the inputs and outputs for each phase. For example, in one project, the *solution analysis & design phase* activities included release scheduling and planning, understanding the requirements of changes, changing the designs, some coding, and some quality assurance. Inputs included the Release Contents Review document; offline discussions among maintainers, users, analysts, and testers; and answers to formal questions submitted to analysts. The outputs included the preliminary designs, test plans, prototypes, release schedule, and size estimates.

4.2 Analysis Steps (4-6)

In Step 4, researchers chose a previous software maintenance release for analysis. Researchers took care to select a recent release, so that the studied release reflected the current process, and so that complete release documentation was available. This choice also made it more likely that the technical lead from the release would be accessible for interviews.

In Step 5, researchers studied the release documentation and interviewed the appropriate parties to define and analyze the problems encountered in developing this release. For each software change request in the release, researchers determined the size of the change, assessed the relative difficulty of the change, and identified any errors or delays that resulted from implementing this change request. If errors or delays resulted from this work, researchers then attempted to determine the maintenance process flaws (if any) that caused these. For example, in one project, a change request for a major enhancement resulted in 11 subsequent errors, substantial rework, and up to 1 month of lost effort on the release. The errors stemmed initially from incomplete or ambiguous change requirements written by the users. The maintainers designed the enhancement based on these written requirements. The fact that the requirements were deficient and that design nevertheless proceeded on the enhancement, was judged by researchers to represent a maintenance process flaw. The effect of this flaw, however, was then compounded by a subsequent lack of communication between the users and maintainers. The users neglected to attend the Release Contents Review and then voiced no objections to the design presented by the maintainers at the Release Design Review. When later, at the Release Acceptance Test Readiness Review, the users finally objected to the implementation of the enhancement, much time had been lost. This lack of communication revealed either an unclear definition of release responsibilities or a lack of adherence to the defined responsibilities.

In Step 6, researchers assessed the frequency and the consequences of flaws in the maintenance process and organization as provided by the data gathered in Step 5, and made recommendations for improvements to the process. For this study, the analysis led to three recommendations: 1) provide guidelines for content and format of change requests; 2) explicitly define the content of documents and review materials; 3) enforce stricter adherence to the maintenance process, especially attendance at review meetings and review/approval of designs.

5. Quantitative Approach to Understanding

In past studies of development projects, tracking the developers' estimates of effort, product size, and schedule has been useful, so similar data were collected for maintenance releases. For maintenance, however, the schedule milestones are somewhat different from development. Thus data were collected on effort hours between release start, release contents review, release design review, release acceptance test readiness review, and release operational readiness review. Researchers monitored and attempted to model the effort that programmers, testers, and managers expend on a maintenance release by breaking the effort down into types of software activity, such as coding, documenting, regression testing, and acceptance testing. Additional activities specific to (or more prominent in) maintenance were included, such as impact analysis, cost benefit analysis, and error isolation time.

The purpose of the quantitative approach was to define and collect those measurements that would most meaningfully characterize the maintenance process and products. Analysis of these data should establish a baseline model of the current maintenance process that answers the following questions:

1. What is the distribution of effort among software activities during maintenance?
2. What are the characteristics of a maintenance release?
3. What are the characteristics of maintenance errors?
4. What are the error rates and change rates?

To achieve the maintenance study goal and to answer these specific questions, the following data were collected:

1. Effort by activity (i.e., impact analysis/cost benefit analysis, isolation, change design, code/unit test, inspection/certification/consulting, integration test, acceptance test, regression test, system documentation, user/other documentation, other hours)
2. Effort by type of maintenance change (i.e., adaptation, error correction, enhancement)
3. Error and change data
   - Time spent (i.e., effort to isolate, effort to fix)
   - Source of error (i.e., previous change, code, design, requirements, other)
   - Class of error (i.e., initialization, logic, external interface, internal interface, computational, or other)
4. Release estimates and actuals (i.e., schedule, effort, number of lines of code, number of modules)
5. Size of software under maintenance (lines of code)

In January 1994, the SEL began collecting data on the eleven target maintenance projects. A new software release estimates form was created and introduced at this time. Two existing data collection forms (a weekly effort form and a software change request form) had already been in use for some time within the organization, and were already being used by three of the eleven target projects. These two existing forms continued to be collected, but now were required for all eleven target projects. In August 1994, following completion of some of the qualitative analysis and after discussions with a wider circle of maintainers, the weekly effort form was revised to capture effort by release and by change request instead of merely by project. The software activities list also was broadened. The preliminary results of the quantitative data analysis are summarized below.

5.1 Maintenance Effort

The average distribution of maintenance effort by activities is presented in Figure 4. The activities (listed above) have been grouped into four categories (design, implementation, test, other). This figure represents the overall distribution based on total effort expended on the eleven maintenance projects from January through October 1994. It includes both entire release cycles and some partial release cycles. This distribution is dominated by the six busiest projects, which contributed 93% of the hours used in the calculation of Figure 4. The distributions for the individual projects vary significantly from each other and also from this average distribution. When more data are available for complete release cycles, there may be some reduction in the variability of this distribution among projects.

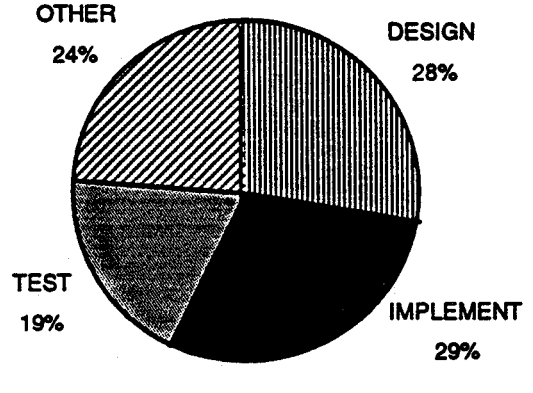

Figure 4. Maintenance Effort Distribution

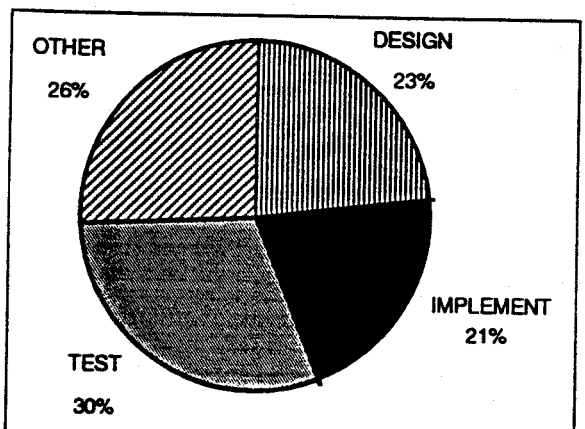

Figure 5. Development Effort Distribution

The distribution of effort during the original development was not available for many of these projects. Figure 5, however, presents the distribution of effort for the original software development of eleven fairly typical projects from this environment.

As illustrated by these two figures, design and code (implement) activity constitute a larger percentage of effort during maintenance than during software development (57% versus 44%). This contrast reinforces the belief that design and implementation are more costly in maintenance than in development. There are many possible reasons for this, for example, the difficulty in isolating errors and the relatively large overhead required to make small code changes. One might expect that this cost increase would be more pronounced for error corrections than for enhancements, because adding major enhancements is more like doing new development work. The data in the next section support this hypothesis, showing greater productivity for enhancements than for error corrections.

5.2 Release Characteristics

When programmers, testers, and managers reported their time spent on maintenance effort each week, they recorded their hours by software activities. Prior to mid-August, when weekly effort collection forms were revised, they also classified their hours by the type of change requests on which they worked (i.e., adaptation, error correction, or enhancement) and *other* hours (e.g., management, meetings). This provided researchers insight into the distribution of types of changes requested and the amount of effort each type requires.

Figure 6 presents the average distribution of effort hours by type of change. These data represent all the effort data for the eleven target maintenance projects from January to mid-August 1994. It includes both entire release cycles and some partial release cycles. This distribution is again dominated by the same six busiest projects, which contributed 93% of the hours used in the calculation in Figure 6. The distributions for the individual projects vary significantly from each other and also from this average distribution. For example, effort spent on enhancements varied from 51% to 89% among the six dominant projects.

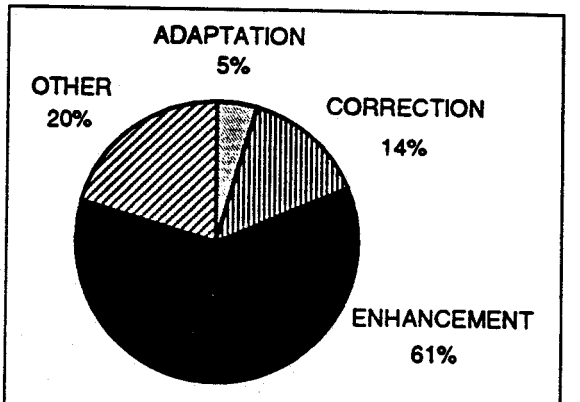

Figure 6. Effort Distribution by Type of Change

Figure 7 presents the distribution of change requests by type. The data are limited to completed releases from the last 2 years for which complete change request data were available. This amounted to nine releases containing 83 change requests (4 adaptations, 37 enhancements, 42 error corrections). Only five of the eleven maintenance projects under study are represented. As more data from complete releases become available, this distribution may change. Again there was much

variability. The percentage of changes that were enhancements in a release varied from 20% to 83%, excluding one release that consisted entirely of error corrections.

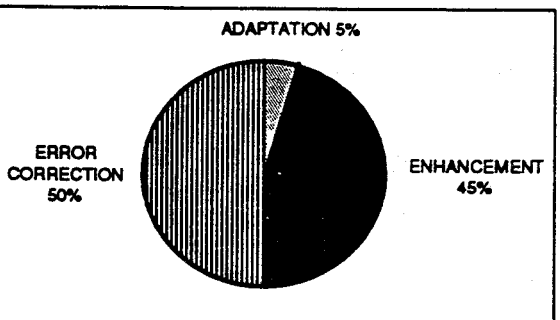

Figure 7. Changes By Type

These last two figures demonstrate that in the FDD enhancements typically are larger than error corrections and require more effort to implement. This is shown by the fact that although the number of enhancements was slightly smaller than the number of error corrections (45% versus 50%), the ratio of effort spent on enhancements to effort spent on error corrections was 4.3:1.

The difference in size is even more dramatic than the difference in effort. The 37 enhancements in these nine releases accounted for 96.6% of the lines of code added, changed, or deleted, whereas the 42 error corrections accounted for only 3.1%, for a ratio of 31:1. By comparing the size ratio (31:1) to the effort ratio (4.3:1), the productivity (lines of code added, changed, or deleted per hour) is about seven times greater for enhancements than it is for error corrections.

5.3 Error Characteristics

The 83 change requests described above represent the *original content* of these nine releases. These are all requests to change the operational version of the software; in this paper, these changes are referred to as *operationally indigenous* changes. During the implementation of each release, however, some errors usually are introduced by the maintenance work. If these errors are caught by the testers, they in turn generate additional change requests which usually become part of the same release delivery. These latter changes are termed *release indigenous* changes. In this study, an attempt was made to separate these two categories of changes. (The effort distribution in Figure 5, however, includes effort on both operationally indigenous and release indigenous change requests. Revised data collection since mid-August will allow effort to be separated by change request.)

The next two figures demonstrate the sources of the errors in these nine releases, both operationally indigenous and release indigenous. The 83 operationally indigenous changes included 42 error corrections (see Figure 8). Note that requirement specification, code, and design each represent a significant portion of the source of errors, 20% to 35% each. These nine releases also included 29 release indigenous change requests, all of which were error corrections (see Figure 9).

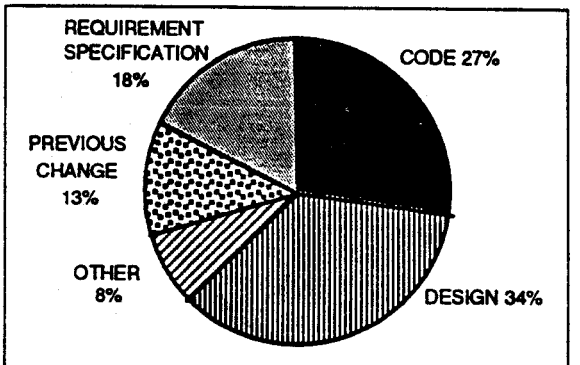

Figure 8. Operational Errors

Note that requirement specification and design represent much smaller portions of the release indigenous errors than of the operationally indigenous errors. Previous change is somewhat higher, and coding is much higher, for release indigenous errors. The distribution of errors found in release testing is similar to the distribution of errors found during acceptance testing of new development projects. This similarity suggests that release testing and development acceptance testing both

uncover similar kinds of errors with similar degrees of success. On the other hand, software operations seem to uncover a different distribution of errors, suggesting that operations are more effective than these testing processes at uncovering certain types of errors, such as design errors, for example. More study is needed to explain why testing and operations should have such different error detection distributions.

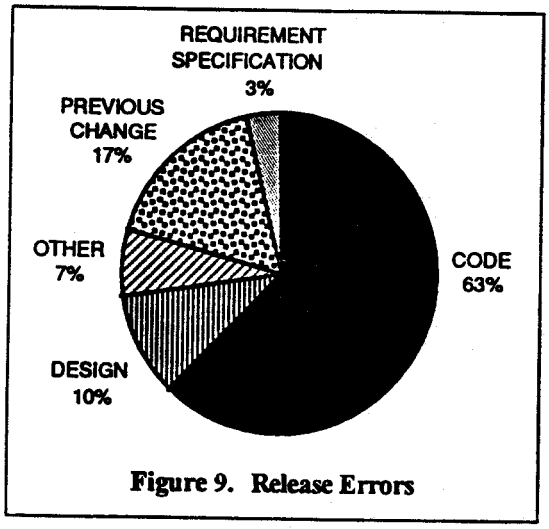

Figure 9. Release Errors

5.4 Error and Change Rates

When the error rate was analyzed for operationally indigenous errors, errors were normalized by both the size of the project (SLOC) and the time period during which they were detected. This adjustment was made for the following reasons: It was expected that, all other things being equal, a larger piece of software would tend to have more errors than a smaller piece of software, so errors/SLOC would be a more meaningful measure of software quality than raw errors. It was also suspected that, all other things being equal, the piece of software that had been exercised operationally for a longer time probably would have more errors uncovered. When comparing error rates for many projects, this dual normalization resulted in more uniform error rates across projects, more so than when either normalization was done separately, or when no normalization was performed at all.

Error rate data were available for ten of the eleven projects in this study, reaching back 2 years for most projects. Analysis of the error rates for these ten projects over the last 2 years (less than 2 years for some of the newer projects) resulted in a mean value of 11 errors per 100 KSLOC per year (minimum 5, maximum 32). Project size ranged from 42 to 263 KSLOC.

Release indigenous errors are those that are introduced by the maintenance process. It was expected that the more code that was modified in a release, the more errors were likely to be introduced. Therefore release indigenous errors were normalized by the modified KSLOC in the original content of the release. *Modified KSLOC* is the sum of KSLOC added, changed, and deleted. For the nine maintenance releases mentioned above, the mean error rate for release indigenous errors was 0.8 errors per modified KSLOC (minimum 0, maximum 6.9). Correcting the release indigenous errors required more lines of code to be added, changed, or deleted before delivering the release. The overall ratio of this additional modified code to the original modified code for the nine was 2.5% [25 additional modified SLOC (minimum 0, maximum 172) per original modified KSLOC].

6. Lessons Learned

This study demonstrated the importance of closely consulting with the software project personnel (here maintainers) when carrying out any software development study. Both the researchers and the maintainers benefited by the close working relationship on this study. The researchers gained a better understanding of the difficulties and peculiarities of the maintenance process; the maintainers gained some insights into the difficulties of the data definition, collection, and analysis process that leads to useful models.

The qualitative analysis that was done for four of the maintenance projects in this study helped ensure that the maintainers were intimately involved in the baselining process. This analysis also helped the researchers to rethink and to begin to redefine the measurement program. For example, weekly personnel effort data is now grouped by release and

by software change, instead of merely by project. Researchers have also redefined and expanded the list of software activities to which maintainers apportion their effort. In addition, the qualitative analysis has suggested the usefulness of reexamining error taxonomies, which the study team hopes to address at a later date.

As the researchers studied the release process, it became evident that there was a need to differentiate between those errors that were *operationally indigenous* and those errors that were *release indigenous*. One obvious reason was that reduction of release indigenous errors is an important improvement goal for maintenance. A second reason is that each of these error sets has something important to say about the maintenance process. In trying to resolve operationally indigenous errors (and adaptations and enhancements), maintainers sometimes introduce release indigenous errors. When such errors are introduced, both the original change request and the change request for the resulting release indigenous error must be examined to learn how effective the maintenance process is and how it might be improved.

Although the definitions given above for these terms imply that the two error sets are distinct, in practice, the actual error populations do not fit the definitions one hundred percent. For example, the set that this study termed the operationally indigenous error set should include only those errors that were introduced during the original development of the software. In reality, this set may also include a few errors that were introduced during maintenance, but which were not identified until the maintenance release became operational. The release indigenous error set should include only errors that were introduced by the maintenance process. In reality, this set may contain some errors that, although caught by release testers, were in fact residing in the operational software and were not new to the maintenance release. Despite these imperfections, there was enough consistency in each set to treat them separately.

In characterizing the size of a release, some measure other than the total number of changes is necessary, because some changes (especially enhancements) tended to be more complex and time consuming than others. For this study, the total modified lines of code (new SLOC + changed SLOC + deleted SLOC) for all changes was used as the measure of release size.

The release characterization demonstrated that, on average, FDD releases are composed of about an equal number of error corrections and enhancements, but that the enhancements require significantly more effort and far more code. Comparing this effort and size data between enhancements and error corrections revealed that the productivity for enhancements was approximately seven times greater than for error corrections. Why this is so, and whether it is good or bad, remains to be seen. The characterization of maintenance errors revealed surprisingly few errors attributed to requirement specifications or to design. This deserves further investigation, especially since the qualitative analysis suggested that requirements deficiencies on software change requests were a problem. The preliminary characterization of error rates resulted in two different ways to normalize errors, one appropriate for operationally indigenous errors and another appropriate for release indigenous errors.

Qualitative analysis suggested that the FDD needs to provide better guidelines for content and format of change requests and release documents. The FDD also needs to enforce stricter adherence to the maintenance process, especially attendance at review meetings. The preliminary quantitative analysis provided many insights into FDD maintenance but also spawned as many new questions. The preliminary effort distributions indicated that design and implementation require more effort in maintenance than they do in new development. Exactly why this is so is not clear at this time.

7. Future Study of Software Maintenance in the SEL

The combination of qualitative and quantitative analysis methods has provided a comprehensive look at the software maintenance process in the FDD. From this researchers have made a good start at baselining this

process. Preliminary quantitative data analysis is based on only nine complete maintenance releases. More releases need to be studied. Also baseline models need to be extended to include an understanding of maintenance cost and cost estimation, plus a better understanding of error rates. Beyond this, future maintenance study activities need to provide a more complete understanding of the testing process and the inspection and certification process. The impact of software development practices on later software maintenance also must be measured.

The FDD has recently embarked on a major effort to port most of its software from IBM mainframes to UNIX workstations. This effort will result in a great many maintenance change requests of the adaptation type. The current study needs to analyze whether and how it should adapt itself to make the most use of the data that this transition will generate.

Once the understanding phase of the current study is completed, the assessing phase will begin. Researchers will design and carry out experiments through which they will be seeking answers to these questions and others:

1. How might we know when a product has outlived its usefulness?
2. What is the "right size" for a maintenance release?
3. Can we predict the most error-prone modifications, and if so how?
4. How can we more accurately estimate the cost of software changes?

This application of the QIP has expanded the SEL's understanding of the maintenance process and product in this environment. Further baselining, experimentation, and research should lead to recommendations for improvements to the maintenance process that can be packaged and instituted in the FDD.

References

1. McGarry, F., G. Page, V. R. Basili, et al., *An Overview of the Software Engineering Laboratory*, SEL-94-005, December 1994.
2. Basili, V. R., "Quantitative Evaluation of Software Engineering Methodology," *Proc. of the First Pan Pacific Computer Conference*, Melbourne, Australia, September 1985 [also available as Technical Report, TR-1519, Department of Computer Science, University of Maryland, College Park, July 1985].
3. Basili, V. R., G. Caldiera, F. McGarry, et al., "The Software Engineering Laboratory – An Operational Software Experience Factory," *International Conference on Software Engineering*, May 1992, pp. 370-381.
4. Briand, L., V. R. Basili, Y. M. Kim, D. Squier, "A Change Analysis Process to Characterize Software Maintenance Projects," *International Conference on Software Maintenance 1994*, Victoria, British Columbia, Canada, September 1994.