Results of modern software engineering principles applied to small and large projects

by PETER R. H. McCONNELL and WOLFGANG B. STRIGEL

MacDonald, Detwiler, and Associates
Richmond, British Columbia, Canada

ABSTRACT

This paper discusses the software development environment, tools, techniques, and methodology as applied in two mediums to large real-time software projects. Both quantitative and qualitative measures of success obtained in these projects are discussed. The quantitative measures are statistics representing the size of produced code, the manpower over the project life cycle, and other data relevant to software engineering management. The qualitative evaluation is more concerned with results obtained from walkthroughs and various aspects of the applied methodology. Results are compared with those reported in the literature. Recommendations and suggestions for further improvements are presented.
INTRODUCTION

The literature abounds with details of the increasing demand for software and the limited increases in productivity that have been obtained. Current predictions hold little hope for major breakthroughs in the future. The results of a recent report by Musa et. al indicate that data processing expenditure has doubled every five years, but with only minimal increases in programmer productivity. These authors' figures indicate that software productivity doubles every 25 years. It is through the better understanding of the software development process and the application of new tools and techniques to this process that industry will improve this productivity factor and meet the increasing demand for software in the future.

The results of a recent survey indicate that a large number of companies are familiar with the modern software techniques but have not applied them in their work environment for various reasons. This same survey indicates that a large number of companies had moderate to excellent results with some of the techniques. One of the problems facing many companies that wish to adopt these techniques to improve their software quality and productivity is deciding which techniques to adopt. The next problem is finding the results of applications of these various techniques in a commercial environment as opposed to a university or an experimental environment. This paper reports on the experiences in applying some of the modern tools and techniques to two medium-sized software development projects in a commercial environment. The tools and techniques which were applied have been reported in the literature and are easily transportable to other environments.

The two projects differed substantially in size, duration, operating environment, and several other ways. What was important, however, was that they shared some of the same tools, techniques, and features of the development environment. Table I summarizes the important characteristics of the two projects.

PROJECT DESCRIPTIONS AND RESULTS

In this section the two projects are discussed in some detail. Subjects such as productivity, languages used, and methodologies applied are highlighted.

Project A

Project A is a real-time data acquisition system for the reception and processing of meteorological satellite imagery. The system is intended as a tool for weather observation and forecasting. It provides the capability to receive, process, and store meteorological data transmitted by geostationary as well as orbiting satellites. A VAX 11/750 performed most of the operator dialogue and all image-processing and display functions. The VAX computer was linked via dual ported disks to a multi-microprocessor system, based on Intel 8086 CPUs, which was designed and built by MacDonald, Dettwiler, and Associates (MDA). The software consisted of an MDA operating system and software for real-time image reception and storage.

First, the system as a whole is broken down into two subsystems, one hosted on the VAX, the other consisting of the multi-microprocessor system. The second step was software oriented and defined how the subsystems were to be implemented in software components. The resulting 100 components are almost evenly distributed between the two subsystems. A software component was sized so that it could be handled by one intermediate-level software engineer, the key designer. Key designers defined components during the design phase and supervised up to three junior programmers during the coding phase. The largest component had 3,700 lines of code, the smallest only 50. In general, however, an average of 1,280 lines was observed as the typical component size.

During the detailed design phase, each component was subdivided into about 20 modules. Each module was defined as one self-contained subroutine with one entry and one exit point. The average module contained 64 executable instructions.

Using strict coding standards, we were able to compile all code for the microprocessors on the corporate VAX computing facilities by using standard DEC compilers. After module testing the code was recompiled with a cross-compiler and loaded into microprocessors. Software quality assurance was implemented by adapting IEEE Software Quality Assurance

<table>
<thead>
<tr>
<th>Table I—Characteristics of Projects A and B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Characteristic</td>
</tr>
<tr>
<td>Effort (programmer-months)</td>
</tr>
<tr>
<td>Duration (months)</td>
</tr>
<tr>
<td>Maximum staff loading</td>
</tr>
<tr>
<td>Principal language</td>
</tr>
<tr>
<td>Lines of source code, made up of:*</td>
</tr>
<tr>
<td>Executable source</td>
</tr>
<tr>
<td>Header</td>
</tr>
<tr>
<td>Commented PDL</td>
</tr>
<tr>
<td>Comments</td>
</tr>
</tbody>
</table>

*The percentages for Project B relate only to the PASCAL and PLM components.
Integra-mental integration necessitated some higher level test drivers, which were developed and maintained by the integration team. During the coding phase a total of 128,000 statements were produced. Not all of the 128,000 source lines were new code. Some modules were ported from a predecessor system. Although reusable, this code still had to be modified and adapted to the new system. Therefore one-half of the ported code is counted to determine the code production rate below. Roughly a quarter of the VAX code could be reused on the microprocessors, since it covered identical functions. Again, some modifications were necessary to use different system calls, etc., and approximately one-half of the original effort can be related to this porting effort. As a result, the total new lines of executable code are now reduced to 98,470 lines. It will be shown later that the total effort of technical staff through all phases of the life cycle amounted to 5,479 person-days. Only the effort of technical personnel is included in the graph. It is not possible to indicate a strict separation between the requirement specification phase and the subsequent high-level design. The line was drawn at the point at which the customer had accepted the specification document. The peak in September 1981 coincides with the first major system design review. A significant increase in staff can be noted at the beginning of detailed design. This is because the first small components were ready for coding and junior staff were added to the team for the actual coding. The system test phase includes the customer-witnessed acceptance test, which ends with customer acceptance. The installation phase was not included in this presentation, since it does not contribute to the development effort.

During all life cycle phases each functional entity went through a thorough review process. Software components as defined during high level design formed the basic entities for review sessions. Aside from the widely published benefits of

![Figure 1—Effort curve for Project A](From the collection of the Computer History Museum (www.computerhistory.org))
the review process, each team member gained and maintained a high level of confidence in the whole development process as well as in the expected quality of the resulting system. During the coding phase the walkthroughs assured strict adherence to our quality assurance standards.

The software integration team started up shortly after the beginning of the detailed design phase. Initially it consisted of an integration leader, an integrator, and the librarian. Key designers submitted software components to the integration team as soon as they had completed the module level testing. The integrator then ran the component test plan, which exercised the new component in the environment of previously submitted components. If this test was successful, the key designer was discharged of any further responsibility for the components, and the integration team maintained the component from then on. As a result of this approach, the key designer was freed to concentrate on another component and was not further disturbed with problems that might show up in previously submitted components. On the other hand, the problem-solving effort was not always the most efficient, since the problem solver had to familiarize himself with a component before efficiently attacking a problem.

The data gathered throughout the project invites the application of some of the existing mathematical models in order to determine the extent to which practical experience meets theoretical expectations. In Figure 1 the Second Level Build-up curve, as used in the Aron model,4 is shown with the data of Project A.

In attempts to improve software productivity, one recurring question is the optimal software component size. A plot of productivity—non-comment source lines per person-day (NCSL/PD)—versus NCSL per component indicated a maximum performance for components of 1,000 to 2,000 source lines. This result was consistent for both the microcomputer-based real-time software and the more application-oriented VAX software. In this calculation the ratio of NCSL/PD did not include the problem-solving effort for a given component, since this effort was booked against the integration team. With an average of 1,280 source instructions per component, Project A has taken full advantage of the optimal component size.

During the implementation and system test phase, close to 1,000 software problem reports (PRs) were filed. The PR reporting mechanism was automated and maintained by the integration team. Once a PR was filed, the integration leader evaluated its importance and assigned it to a problem solver. The PR solver updated the PR to describe the solution approach and the changes applied to the component. This report was again verified by the integration leader, and the integration team took care of regression testing and reintegration of the component. Figure 2 shows the incidence of PRs during the implementation and testing phases. It also shows the time

![Figure 2—Number of problems found per month and the time to fix them](From the collection of the Computer History Museum (www.computerhistory.org))
spent in solving the problems. It can be seen that during the earlier months problems were quite easy to solve, whereas toward the end of the reported period the time to solve a PR increased significantly. This confirms the well-known principle that the later a software problem is discovered, the greater the cost to repair the problem. The problems reported in the beginning were mostly trivial. They were easy to analyze and took an average of only a few hours each to be fixed. Most of those problems were categorized as being related to implementation. As soon as system test started in July 1983, the amount of reported PRs increased remarkably. At about the same time the complexity of problems increased significantly. A high proportion of those PRs was related to design faults, which meant that the detailed design documentation also required some updates.

Project B

The second project consisted of real-time software to control an airborne synthetic-aperture radar system. This software was required to provide all operator interface to the radar system, as well as to respond to and service several different hardware-generated interrupts. The particular system being discussed was a first-time development effort with concurrent hardware and software development. The complete system was developed over a period of two years and the software over a period of nine months.

One of the problems with this type of environment is that the development team has only limited access to the hardware for the software development, and it cannot always be ensured that the hardware is fully operational. In this case this problem required that all of the software be developed on multiuser development facility and ported to the target system, providing maximum system access for the development team.

Many of the software development techniques used on this project had been applied successfully on projects operating in a different environment, and several new techniques were applied.

The actual implementation plan for the entire project is shown in Figure 3. This plan was divided into four distinct activities, which could be associated with various software staff. These were as follows:

1. Requirements analysis and system specification, which were carried out by the senior software engineer with assistance from intermediate software staff.
2. Software system design, which was carried out by the senior software engineer with assistance from an intermediate software engineer.
3. Unit detailed design, code, and test, which were carried

![Graph showing effort curve for Project B](From the collection of the Computer History Museum (www.computerhistory.org))
out by a number of junior and intermediate software engineers.

4. System integration, which was carried out by an intermediate software engineer.

Tools applied at the various steps consisted mainly of methodologies and techniques that have appeared in the literature. A survey was done prior to the start of the work to identify tools that could be best applied to the various phases of the development life cycles. (The term tools is used here because these methodologies and techniques serve to aid the software engineering process and make the overall effort more productive.)

The software development life cycle of this project is characteristic of most small- to medium-sized software projects that use modern software development methodologies. This type of project has been referred to by Aron as "First Level." What distinguishes this from a second- or higher-level project, which is typically a larger size project with a labor expenditure curve described by the Rayleigh-Norden curve, is the amount of effort and duration required for the project. The results for this project are compared in Figure 3 with the labor expenditure curve for a second-level project. This curve shows good qualitative agreement with the curve presented by Aron for a second-level project buildup. The shape of this curve results from the software development life cycle adopted and the tools applied at these various steps. These steps were requirements analysis and system specification, system detailed design, system design review, unit detailed design, unit detailed design review, unit code, unit test, unit code and test review, system integration, system test, and system delivery.

The extensive reviews, which required about 10% of the overall effort, ensured that reasonable errors could be removed early in the development of the software. There is extensive information in the literature to support the fact that it is much costlier to remove errors discovered late in the development cycle.

During the initial phases of the project there was some overlap in the requirements analysis and system design phases. The reason for this was that the use of the Petri nets in describing the operator interaction with the system caused us to consider a different approach, which accomplished the same operational goals but resulted in considerable simplification of a major software component. The system design was then completed by means of SADT-like activity diagrams and Petri nets. Following the system design phase, an extensive implementation phase began. In this phase each individual software engineer was responsible for the detailed design, coding, and testing of his assigned component. This component consisted of an average of 379 lines of executable source code and 840 lines of header, PDL (program design language), and comments. These components were further broken down into procedures or modules, which averaged about 50 lines of executable code each.

As part of the detailed design process the component designer was expected to perform the design process on the development VAX and deliver the design material in machine-readable form for the design review process. This material consisted of component headers and a PDL that described the design. Each module header in the component provided information on all module inputs, all module outputs, an example of the module usage, an English language description of the procedure, a revision history for that procedure, and a list of all required procedures and included files external to that component. Following the procedure header was the PDL, or pseudocode, for that procedure. This PDL served the purpose of providing a structured-English description of how the procedure processed the inputs and generated the outputs. This material was reviewed during the detailed design review process to examine the headers and the PDL for adherence to the project standards and to check that the module interface and the PDL were correctly specified. The tool used in creating and judging the detailed design was the Myers composite/structured design. Upon acceptance of the design, the component designer was responsible for implementing that component in the specified language. Most of the components were implemented in PASCAL or PL/M with some of the code being done in assembler. In the case of PASCAL, it was possible to use a subset of the VAX PASCAL that was compatible with the cross-compiler for the target computer. In this way the software engineer could create code and test it on the company development machine. In the case of the PL/M and assembly code components, testing had to be done on the target system. The use of the multiuser development computer allowed the software engineer to use the tools available on that system, such as source code control and a symbolic debugger. The tools available for the target hardware were very limited, basically consisting of a monitor/debugger and task image transfer utility. When it was desired to test a component or subset of the final system, the executable image was created on the VAX and transferred via a serial link to RAM memory in the target system for testing.

Upon completion of the coding and unit testing, the component was submitted for the code and test review. The items of interest here were the adherence to the project standards, a one-to-one correspondence between the PDL and the code, and any weaknesses in the coding.

At the completion of the unit design, code, and test, the component was submitted to the integration directory, where it was available to the integration engineer. This person was responsible for incorporating the components into the total system. In terms of integration, a crude skeleton system was created early in the development, and all missing components were implemented as stubs. As the finished components became available, they replaced these stubs and were tested as a functioning part of the entire system. This process continued until a complete system was available for systems testing according to an internal acceptance test. The purpose of this test was to expose the system to a rigorous set of tests which would verify the correct functioning of the software. Any problems that were encountered were corrected and the tests performed again.

Throughout the software development process, quantitative measures were obtained of the effort expended in each of the life cycle steps. These results showed that coding of the software occupies a rather small portion (20%) of the total effort required to develop the final system product. About
42% of the actual effort expended was dedicated to requirements specification, system design, analysis, and detailed design.

Over the duration of the project, about 30,000 lines of executable code and 37,000 lines of nonexecutable code were developed. The required effort was 744 person-days, providing an average productivity figure of 40 non-comment source lines (NCSL) per programmer-day. About 60% of this consisted of PASCAL and PL/M; the remaining 40% was done in assembly language. An interesting number obtained during the implementation phase of the project was the number of terminal connect hours per programmer day. This is the average number of hours per day that a programmer is signed on to a terminal. These results indicated an average of three hours per day at the start of the detailed design and an average of seven hours per day at the peak of the integration. It is readily apparent that a software engineer makes significant use of the computer system available, especially in the later stages of the project, when intensive integration and testing efforts are under way. This fact lends support to the belief that a programmer’s work station forms an important part of the software development environment.

CONCLUSIONS

Project A was considered a successful project. The results obtained confirm most commonly found theories on software development. Perhaps one of the weakest points in Project A was the lack of automated tools. For each of the life cycle phases more design and implementation tools should be made available. Enforcing standards on software engineering methodology is extremely difficult without computerized assistance. The few tools available in Project A were mainly geared to integration mechanisms and configuration control. This proved to be highly beneficial, even though only a minimal amount of time was spent to develop those tools specifically for Project A.

The area of problem solving still seems to leave room for improvement. If the problem solvers are not the original developers, high demands are put upon them to familiarize themselves with each new component. Our experiment in incremental integration seems to have been at least as successful as the top-down approach for integration. Low-level modules were the ones exercised for the longest period during integration. This is highly desirable, since an error in low-level routines is not only harder to find at later stages of the project but also has a more detrimental effect on system uptime and stability.

The basic conclusions derived from Project B are that even simple modern software engineering techniques can be successfully applied to a project and can offer positive benefits. This has been evidenced by the reasonably high productivity figure obtained—40 NCSL/PD—compared to averages in the literature of 10 to 20. Although only qualitative statements can be made about the resulting software quality, the very rapid fall-off of effort after the system integration phase indicates that the tools and techniques applied resulted in a very low occurrence of errors. The software developed for this project is being used as foundation software for a more complex system, and preliminary indications are that it is easily adapted to the new environment. Tools to be singled out as the most positive contributors to the success of the project would have to be the walkthroughs and inspections following design, code, and test. It is felt that the following positive benefits resulted from reviews:

1. Impending reviews caused engineers to put more thought into their work. In addition, most errors were caught early, and little rework was required.
2. They served as a learning forum for other engineers on the team.
3. They provided the opportunity to define exact end points for the design and code/test phases of the development. This is very useful for tracking a project’s actual status against that planned prior to the implementation phase.
4. They provided convenient checkpoints to ensure that the products complied with the project standards.

The application of simple but effective design tools to the project cannot be overlooked. These tools enabled the software engineer to achieve correct designs. The dedication of effort to up-front design is also a must. Any decrease in this effort would only show up as problems in the later phases of the project.

SUMMARY

The application of modern software development techniques to a software project requires a commitment of both time and money by a company that wishes to benefit from these techniques. Although the two projects discussed in this paper did not implement state-of-the-art tools, they both benefited from the application of simple tools and techniques. The results of the application of these tools and techniques can be summarized as follows:

1. Modern tools and techniques contribute significantly to increased software productivity and quality.
2. Metrics must be developed that can be applied at every step of the software development life cycle. These can then be used to judge the quality of the results at each stage.
3. Results of the application of new tools and techniques must be reported in the literature to allow comparison of the various tools and techniques.
4. Industry must be involved in the evaluation and development of new tools and techniques for software development, since this is where the greatest benefits can be realized.
5. Research must continue to define the software process better and to develop tools and techniques that better serve the user in the design and development process.
REFERENCES
