A Final Report
Grant No. NAG-1-1073

November 22, 1989 - November 21, 1990

A RESEARCH PROGRAM IN EMPIRICAL COMPUTER SCIENCE

Submitted to:

National Aeronautics and Space Administration
Langley Research Center
Hampton, VA 23665

Attention:

D. E. Eckhardt, Jr.
ISD, M/S 478

Submitted by:

J. C. Knight
Associate Professor

Department of Computer Science
SCHOOL OF ENGINEERING AND APPLIED SCIENCE
UNIVERSITY OF VIRGINIA
CHARLOTTESVILLE, VIRGINIA

Report No. UVA/528334/CS91/101
February 1991

Copy No._____

TABLE OF CONTENTS

1. INTRODUCTION

During the grant reporting period our primary activities have been to begin preparation for the establishment of a research program in experimental computer science. The focus of research in this program will be safety-critical systems.

Many questions that arise in the effort to improve software dependability can only be addressed empirically. For example, there is no way to predict the performance of the various proposed approaches to building fault-tolerant software. Performance models, though valuable, are parameterized and cannot be used to make quantitative predictions without experimental determination of underlying distributions. In the past, experimentation has been able to shed some light on the practical benefits and limitations of software fault tolerance.

It is common, also, for experimentation to reveal new questions or new aspects of problems that were previously unknown. A good example is the Consistent Comparison Problem that was revealed by experimentation and subsequently studied in depth. The result was a clear understanding of a previously unknown problem with software fault tolerance.

The purpose of a research program in empirical computer science is to perform controlled experiments in the area of real-time, embedded control systems. The goal of the various experiments will be to determine better approaches to the construction of the software for computing systems that have to be relied upon. As such it will validate research concepts from other sources, provide new research results, and facilitate the transition of research results from concepts to practical procedures that can be applied with low risk to NASA flight projects.

The target of experimentation will be the production software development activities undertaken by any organization prepared to contribute to the research program. Experimental goals, procedures, data analysis and result reporting will be performed for the most part by the University of Virginia.

This report is organized as follows. In section 2, a review of the background and the major issues concerning empirical computer science are presented. Some of the statistical issues faced by researchers undertaking experiments are discussed in section 3. A new paradigm for experimentation is outlined in section 4, and a preliminary evaluation experiment is summarized in section 5. Finally, a bibliography of recent papers on the subject is included. So many papers have been written that relate to this project that most are not cited individually in the body of the report.

2. BACKGROUND AND JUSTIFICATION IN GENERAL

Many important questions in software engineering remain unanswered because there is insufficient opportunity for experimental evaluation of issues. There is no national resource for experimentation in software engineering despite the fact that software is a major industry. There are national facilities for experimentation in other areas, high energy physics for example, even though in many cases such areas are not associated with a specific industry.

Some experimentation has taken place at universities but the results, though frequently useful, do not necessarily apply to industrial environments. Much less experimentation has been performed in realistic production software developments. An important exception is the *Software Engineering Laboratory* (SEL) operated jointly by the University of Maryland, NASA Goddard Space Flight Center, and Computer Sciences Corporation [22].

The SEL has been operating for approximately thirteen years and has produced a wealth of important research results during that period. The emphasis of the SEL is efficient development of ground-based software. The research undertaken has been very varied in nature covering topics such as measurement of programmer activities to help validate cost models, performance comparison of programmers using Ada and FORTRAN, and various evaluations of test methods on production software.

Experimentation in software engineering is limited for three major reasons:

(1) *It is expensive.*

Any effort to perform experiments in the area of software engineering involves building software and that is expensive. Worse still, for results to be believed, they should come from a statistically valid sample of data. That might involve repeating the same software engineering activity several times in order to acquire adequate data. The expenditure of sufficient resources to perform these experiments with professional programming staffs and equipment is beyond the capacity of industrial software development organizations. It is for this reason that many of the experiments that are performed take place in universities using student programmers and teaching equipment.

(2) *It requires flexibility in the development process.*

The approach to experimentation employed in the SEL reduces the cost substantially by using production software development as the target of experimentation. With this method, a piece of software that is actually needed is produced with designated funds but the process of production is observed and measured as the target of experimentation. This process is not perfect in that it is not possible to control all the independent variables in the way that a researcher might prefer. For example, the total staff assigned to the development cannot be changed, the programming language and target computers cannot be changed, and the overall software development method cannot be changed. However, the approach does offer considerable opportunities for useful experimentation and some relaxation of the restrictions just outlined are possible by performing some experiments separately from development. For example, new concepts in testing can be explored by taking the software as it is produced and testing it in an experimental manner in parallel with the conventional testing performed by the development team.

Unfortunately, even the approach used by the SEL is not without cost. Any

experimentation involving observation disturbs the subject being observed. In order to perform experiments on production software development activities, those performing the activities must be prepared to be observed, the cost of observation must be met, and the disturbance to the development operation resulting from the observation must be tolerated. Industrial software development activities are typically performed under contract and according to a prescribed schedule. Often the disturbance associated with even limited experimentation is sufficient that industrial organizations are not willing to participate in such experiments even though they admit their value.

(3) *Industrial software development often has restricted access.*

Although some industrial organizations are prepared to undertake experiments in software engineering, it is often not possible because the software that would be the subject of investigation is either classified or proprietary.

Much of the software development undertaken by NASA and its contractors is free of the various restrictions outlined above. The very nature of the agency includes a desire for research and experimentation, and where obstacles are present that would normally inhibit experimentation, there is a desire to remove the obstacles to promote better and more extensive research. The disturbance resulting from experimentation mentioned above is inevitable but likely to be tolerated within NASA provided it is not excessive. In addition, much of the software produced is neither classified nor proprietary yet it is completely realistic allowing meaningful experimentation.

3. STATISTICAL ANALYSIS OF EXPERIMENTAL DATA

The basic goal of a research program in empirical computer science is to determine which tools and techniques can be depended upon to support the development of software for safety-critical systems. As noted in section 1, many of the results that must be obtained can only be

obtained empirically. Virtually none of the significant results depend upon simple constants. Rather they depend on the comparison of random variables. For example, an important question is whether a formal specification technique will permit systems to be built with higher reliability than informal specification techniques. This cannot be determined definitively by a simple comparison of single systems built using the two specification methods. The degree of difference between the two is a random variable and what is required is information about its distribution.

The most appropriate way to perform such a comparison is with a statistical hypothesis tests. Such tests allow conclusions to be drawn of the form "method A is better than method B" with a certain probability, or confidence, that the conclusion is correct. Such hypothesis tests allow higher levels of confidence to be used if more data are available about the underlying populations. In the limiting case, where all the population data are available, clearly the confidence level is 100%.

Obtaining confidence levels that are usefully high implies having a large set of data points from the two distributions being compared. In the context of the experimentation being discussed here, this means that observations of *several* development activities need to be observed, some using the original method and some using the proposed new method. Unfortunately, such experimentation is out of the question in software engineering. More importantly, even experimentation in which a *single* control project is available for comparison with a *single* project using a new technique is obviously very expensive. Funding for control studies is very unlikely to be available.

The results of this situation are:

(1) It is unlikely that statistically valid conclusions about the effect of a new technique, method, or tool could ever be drawn. Thus statements of the form "method B provides an improvement of Y% in quantity Q over method A with confidence C" are unlikely ever to

be possible. At best, observed values of some quantity will be available and reported. This is a serious yet unavoidable problem and forces the user of such results to draw informal conclusions and hope they are valid. This does not mean that such experiments should not be performed. It means that trustworthy quantitative conclusions cannot be drawn. However, data collection under such circumstances can give great insight and permit informal conclusions to be drawn that are almost certainly right.

(2) On the brighter side, a single data point is sufficient to reject certain hypotheses and this can be very useful. For example, a hypothesis of the form "method B provides an improvement of Y% in quantity Q over method A" can be rejected if an experiment with a control does not obtain a Y% improvement. Of course, if a Y% improvement is obtained, the hypothesis cannot be accepted.

(3) An area where good results can be obtained is feasibility. At this stage in our understanding, there are many proposed techniques that have not even been shown to be feasible. For example, the use of formal specifications on a project involving many programmers has never been shown to be a realistic approach. An experiment in which the question of feasibility were investigated could obviously permit positive conclusions to be drawn.

4. A PARADIGM FOR EXPERIMENTATION

In the area of dependable computing, we find ourselves in the same situation that faced the general software engineering community when the Goddard SEL was formed. It is tempting, therefore, to establish a program of experimentation to support dependable computing using the SEL as a model.

Upon closer examination of the SEL program, it is clear that some changes have to be made before the SEL model can be used. As noted above, the cost of experimentation in the SEL is

kept within manageable limits by, for the most part, using production software development as the target of observation. While providing the great benefit of reducing cost, this also limits the range of experiment that can be undertaken. Experiments involving the development of production software must be relatively low risk or they might jeopardize the successful completion of the product. Thus an experiment that wished to use a totally novel and untried tool or technique would be very hard to perform. In the context of the SEL, this is not a major limitation since there are so many important but low-risk experiments that can be performed. This results largely from the fact that an established and extensive development method is in place and generating production software on time at NASA Goddard. A characterization of the SEL experimentation process is shown in figure 1. Note that the emphasis is on technique selection rather than the creation of new tools or techniques.

The situation with development methods for safety-critical systems is such that a conservative approach to experimentation cannot be taken. There is no corresponding established approach to software development to which a program of experimentation could add technique selection or modification. In the area of safety-critical software development, many completely fundamental questions remain. For example, a central issue is the role of formal methods and, specifically, whether an entire development method based on formal methods could offer a route to the routine development of software with adequate dependability. The experiments required are driven by questions that are associated with substantial risk.

The paradigm for experimentation that is proposed, therefore, is one in which production software is built in a laboratory setting but is subjected to industrial constraints. The development would, however, involve new and untried methods or methods that have not been tried previously in an industrial setting. The risks would be high in that useful products might not be produced. This is precisely why such experiments are required since resolving the risk is a step that must be undertaken before more detailed information on methods can be obtained and

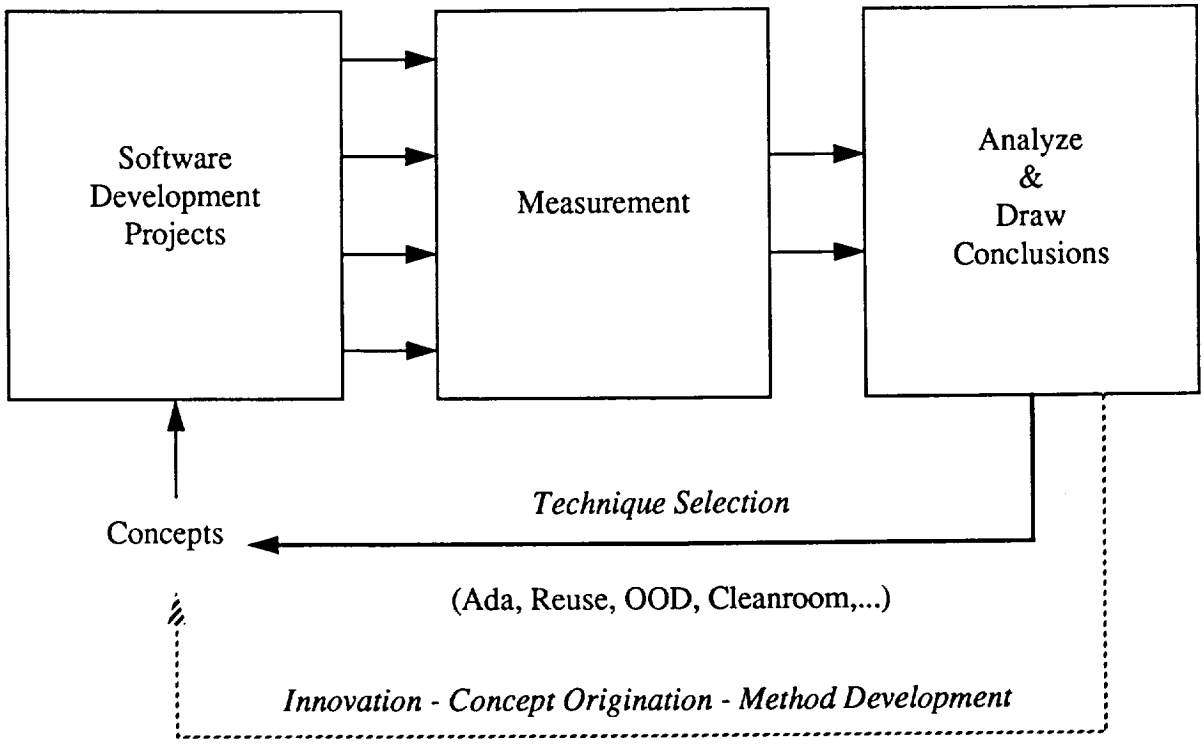

Fig. 1 - GSFC/UMd SEL Operation

before the methods can be applied routinely with confidence in industrial production development.

Figure 2 shows the proposed paradigm for experimentation. It focuses on innovation in tools, techniques, and methods. It admits that such concepts might result from observational experiments, and that they will need to be evaluated empirically. Thus a major aspect of the paradigm is to seek new concepts, pose research questions concerning the feasibility, relevance, or performance of the concept, and to then design and carry out experiments based on these questions.

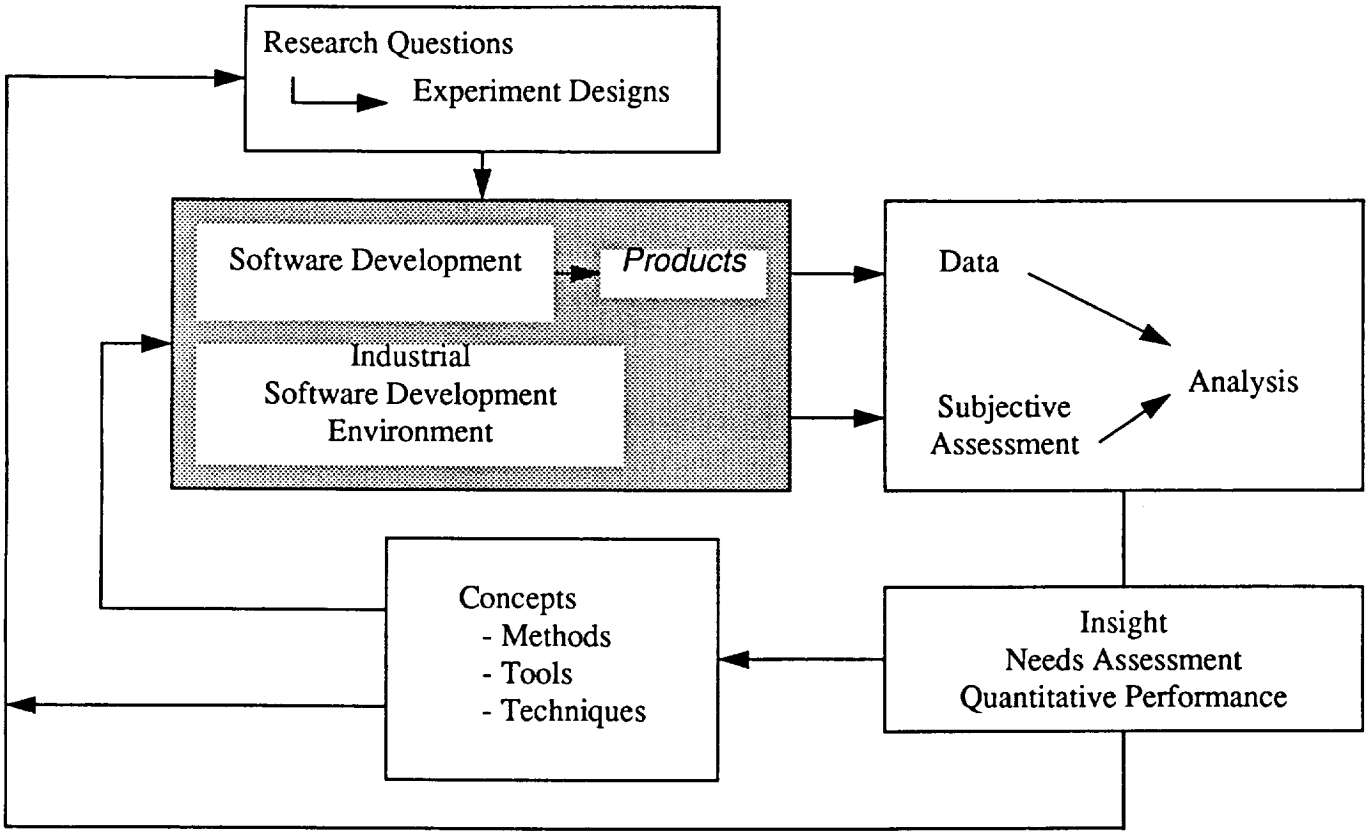

Fig. 2 - Paradigm For Experimentation

Within the general paradigm of experimentation, there are essentially three types of experiment that can be performed. They will be referred to here as *fully controlled*, *semi-controlled*, and *non-controlled*.

Fully controlled experiments are just that, fully controlled. All of the independent variables having influence over the outcome and all quantities affecting the statistical results can be set by the researcher. A predefined application is developed in a statistically significant number of replicates by separate staffs carefully selected to eliminate statistically meaningful differences in experience, abilities, education, etc. The individual staffs would use all the same techniques and

tools but one to develop the application. The resulting software would be analyzed to determine whether any of the differing techniques produces *better* results according to some metric. For example, an experiment might develop software with two different programming languages, showing whether one language better lends itself to producing reliable code.

Fully controlled experiments are expensive, but very desirable. A fully controlled experiment could be used, for example, to explore the benefits of using formal specifications versus informal specifications. Informal specifications for a predefined application would be rewritten in various formal notations. Groups of programmers, carefully selected to minimize differences in experience and ability, would develop software independently from the different forms of the specifications. During the development process, measurements and observations would include:

(1) Tools required during the development process.

(2) Acceptability of the formal specifications to the programmers.

(3) Questions that arise about the specifications (formal and informal).

(4) Tools required to write formal specifications.

(5) Errors found in specifications (formal and informal).

The experiment would ultimately compare the reliability of software developed from formal specifications with software developed from informal specifications.

Semi-controlled experiments control some but not all aspects of the development process. Those factors that are not controlled vary under whatever influences usually operate, and the results of the experiment are conditional on the values that the non-controlled independent variables take. The extents and types of change that will be tolerated by the development

environment determine to what degree these types of experiments can be done.

In the context of assessing the performance of formal specifications, a semi-controlled experiment could be used to indicate how difficult is it to develop software with formal specifications. Informal specifications for an existing application would be rewritten in a formal notation. Programmers assigned to the development would then use the formal specifications. In such an experiment, the application, the staff, the language and computers used would not be controlled by the researcher, but the results might reveal useful information such as whether using formal specifications is feasible in a productive development environment, what programmer training is required, what tools might be useful, etc.

Experiments not controlled by a researcher interfere very little with the existing development process. These types of experiments observe and measure the development process, providing very useful information about the effectiveness of the development process. However, it is virtually impossible to get meaningful quantitative data for comparative purposes from such efforts.

While non-controlled experiments on existing applications do not control the development process, they do disturb it because of the inevitable intrusion resulting from data collection. How data collection is done depends on what data are available and in what form. For example, are the specifications, the cost estimates, the expected code size, the staff levels, the application details, the development tools, and the development hardware available? Many times even non-controlled experiments fail because even minimal data collection is not performed by the development organization.

Interference with the development process can be reduced by automating the data collection as much as possible. How much automation is possible depends on whether access to code and other documents in electronic form is provided and whether modifications to the operating system

used for the development are possible.

Removing code and other artifacts from the development environment for testing and analysis at the laboratory can also reduce the disturbance of experimentation and provide opportunities to perform more controlled, desirable experiments. Of course, the laboratory has to be made aware of any special purpose hardware required by the code and artifacts that might restrict analysis.

Considering once again the example of assessing the benefits of formal specifications, If a non-controlled experiment is all that can be achieved, useful results can still be obtained. An experiment could determine, for example, the feasibility of formal specifications. Using non-development staff, an attempt could be made to rewrite informal specifications for an existing application in various formal notations in parallel with the production development. Such an experiment would indicate whether formal notations could be prepared that are adequate to describe the kinds of applications currently being developed. Specific quantities that might be measured even in a non-controlled experiment with minimal impact on the development organization include:

(1) Resources expended in developing formal specifications.

(2) Errors in the formal specifications.

(3) Tools for supporting formal specification development.

(4) Acceptability of such specifications to programmers.

5. SUMMARY OF EVALUATION EXPERIMENT

In order to evaluate the proposed paradigm for experimentation, we have carried out a preliminary evaluation experiment. We performed this experiment to gain experience with the

advocated paradigm and determine its practicality. In this section only a summary of the experiment is presented. A complete report will be supplied under separate cover [23]. The experiment is in the category of fully controlled since all aspects were under our control.

The topic we chose to study was software inspections. We chose inspections because there is substantial evidence that they are highly effective at locating defects in software when carried out carefully. However, we suspected that improved techniques might be possible, and that determining the suitability and performance of new ideas in this area could only be determined by inspection. The experimental procedure we followed was to:

(1) study an industrial implementation of software inspections,

(2) define a radically different approach to inspections that we hypothesized would be an improvement,

(3) define a toolset that supports the advocated procedure,

(4) implement a prototype version of the toolset for evaluation,

(5) perform a set of trial inspections using the revised inspection approach supported by the prototype toolset,

(6) revise the process and the toolset based on the results of the trial inspections,

(7) seek industrial partners to assess the technology in a practical context.

5.1. Existing Techniques

Software inspections have been employed for a long time in various forms. They have been referred to variously as walkthroughs, code readings, inspections, Fagan inspections [13] and audits. They have been applied to all work products that are generated during software development including requirements specifications, designs, source code, and test plans. By far

the most popular application of inspections is the examination of source code.

The basic idea behind all of these techniques is for human readers to examine a work product and look for algorithmic defects. Procedures differ and the members of an inspection team differ according to the particular approach being applied, but all rely on human examination of a paper version of the inspection target.

Empirical evidence has emerged showing that such activities, as part of a systematic software development process, can have considerable benefit [13]. Most of the benefit that accrues is a lowering in the rate of faults in the deployed software. Since inspections typically take place before any form of verification, they can be highly cost effective because they eliminate algorithmic defects very early in the lifecycle.

Despite this success, many major difficulties remain. We summarize three important ones here. First, inspections are in no sense rigorous. This leads to situations in which, although a work product may have been inspected, it is not possible to specify the precise benefits achieved. In a statistical sense, inspections produce valuable results but a given inspection does not necessarily ensure that a work product has any specific quality.

A second important difficulty is that the human resources involved are not used effectively. The process known as Fagan inspections, for example, includes a step in which the author of a work product presents an overview of the product to the inspection team. This is quite inappropriate since it suggests that vital design or implementation information about the product is conveyed to the inspectors verbally. Such information should be readily available in associated documents. As a second example, anecdotal evidence also suggests that inspectors often use inspection time ineffectively by discussing essentially trivial difficulties with the work product.

A third difficulty is the dependence of traditional inspection methods on human effort with essentially no computer support. It is possible to supplement the inspection process considerably

with computer resources. This permits far more efficient use of human time and more complete coverage of items that have to be inspected.

We take the position that inspections should be viewed as an approach to informal proof that a work product possesses certain properties. Further, we consider that establishment of these properties should be undertaken with an approach that permits assurance that the properties exist for a given work product after an inspection. There should be as little dependence on statistical chance to achieve results as possible. This amounts to making inspections a rigorous process and by doing so we suggest that they would be a far more valuable element of the software development process.

5.2. Phased Inspections

We have defined a new approach to inspections termed *phased inspections*. Phased inspections are intended to *ensure*, to the extent possible with this technology, that work products possess certain useful properties. These properties are not limited to freedom from algorithmic defects but include properties such as freedom from programming practices that tend to be associated with high rates of defects even if specific instances turn out to be correct. Other example properties include important elements of program style that are known to improve the maintainability of software. The goal with phased inspections is to make the process rigorous, repeatable, as efficient as possible, and as dependent on computer support as possible.

The concept of phased inspections is simple. It is only summarized here because of space limitations. A phased inspection consists of a series of partial inspections termed phases. Each phase addresses one or a small set of related properties that it is deemed desirable for the software to have. Phases are conducted in series with each depending on the properties established in preceding phases. Each inspector associated with each phase is required to sign a statement after the phase that the software possess the prescribed property to the best of his or her knowledge.

Each phase is carried out by an individual or team and the goal is to establish the presence of the desired property in the work product. To the extent possible, check lists are used to ensure that the required property has a precise definition. Some of the latter phases of an inspection involve establishing correctness properties and these cannot be based on statically defined checklists. Such properties are defined to the extent possible by checklists that are derived from the work product itself. For example, correctness in the definition and use of internal interfaces is based on checklists developed according to prescribed rules by the *author* of the work product.

5.3. Phased Inspection Support Toolset

Computer support for phased inspections is supplied by a set of tools that are presently in prototype form. The toolset provides service in three areas:

(1) *Support for management in controlling the inspection process.*

This element of the toolset is designed to deal with configuration management of the work products, allocation of staff to the various inspection phases, and management information concerning the state of various inspections.

(2) *Support for inspectors.*

Various tools are available to support the actual process of examining the work product. Some examples include a general display, scrolling, and searching facility that allows textual work products such as source code to be reviewed rapidly, a facility to permit inspectors to note their conclusions electronically, a syntax-based highlight mechanism that permits various important syntactic structures to be made readily visible, and a display of the checklists, their associated background and justification information.

(3) *Support for compliance.*

Where items are to be checked by human inspectors, it is essential that the checks be complete. Every instance of the item to be checked must actually be checked by the

inspector. The compliance support facility monitors the inspector's use of the tool and the checklists, and ensures, to the extent possible, that the inspector is achieving complete coverage.

5.4. Trial Inspections

The key research questions initially with phased inspections were practical. First, it had to be determined whether the basic concept provides a useful benefit to software developers. Benefit is defined to be a cost-effective improvement in some aspect of software quality. The only way to answer this question is by experimentation.

The second important research question was the degree to which the concept met its major goal of establishing rigor in the inspection process. In principle it does. The issue was whether this can be carried through to practice and so, once again, the way to answer this question is by experimentation. Many other research questions exist and all are best addressed in whole or in part by observing and measuring the ideas and tools in practice.

We performed an empirical study of phased inspections in order to get information on the feasibility and performance of the concept and the toolset. Development of the concept to the point where it can be applied readily to production software development requires extensive data on the feasibility of various aspects of the concept and performance data on the whole process. The preliminary experiment was limited by the available resources.

Trial phased inspections were conducted by graduate students at the University of Virginia. The subject of the inspections was the source code for the phased-inspection toolset and the experiment focused on the feasibility of the process and the toolset. The results of this preliminary study led to extensive enhancements to the toolset and minor changes to the process.

The results of the trial inspections led to extensive revisions to the toolset concept and minor changes to the process of phased inspections. We have begun to develop a tailored phased-inspection process and toolset for Science Applications International Corporation (SAIC). This activity is in support of SAIC's work in Ada reuse, and will lead to an inspection process in which the reusability of Ada software components is determined. We will be using this activity to gather preliminary data on the use of phased inspections in an industrial setting.

5.5. Conclusions

The evaluation experiment is ongoing. The prototype toolset is being developed and plans are proceeding for industrial assessment of the technique and the toolset. The most significant conclusion that can be drawn at this time is that experimentation that attempts to define and evaluate new tools and techniques is workable and very beneficial. At this stage, phased inspections appear to be a substantially better technology than those already existing, and the toolset designed to support this technology appears to be highly successful.

BIBLIOGRAPHY

(1) Kemmerer, Richard A., "Testing formal specifications to detect sign errors", *IEEE Transactions on Software Engineering*, Vol. 11, No. 1, January 1985, pp. 32-43.

(2) Weiss, David M., "Evaluating software development by analysis of changes; software engineering laboratory data", *IEEE Transactions on Software Engineering*, Vol. 11, No. 2, February 1985, pp. 157-168.

(3) Hayes, Ian J., "Applying formal specification to software development in industry; case study, IBM's CICs", *IEEE Transactions on Software Engineering*, Vol. 11, No. 2, February 1985, pp. 169-178.

(4) Shen, Vincent Y., "Identifying error-prone software; empirical study", *IEEE Transactions on Software Engineering*, Vol. 11, No. 4, April 1985, pp. 317-324.

(5) Urban, Susan D., "Utilizing an executable specification language for an information system", *IEEE Transactions on Software Engineering*, Vol. 11, No. 7, July 1985, pp. 598-605.

(6) Chi, Uli H., "Formal specification of user interfaces; comparison and evaluation of four axiomatic approaches", *IEEE Transactions on Software Engineering*, Vol. 11, No. 8, August 1985, pp. 671-685.

(7) Okumoto, Kazuhira , "Statistical method for software quality control", *IEEE Transactions on Software Engineering*, Vol. 11, No. 12, December 1985, pp. 1424-1430.

(8) Cavano, Joseph P., "Software management approach to achieving high-confidence software", *IEEE Transactions on Software Engineering*, Vol. 11, No. 12, December 1985, pp. 1449-1455.

(9) Knight, John C., "An experimental evaluation of the independence assumption in multiversion programming;", *IEEE Transactions on Software Engineering*, Vol. 12, No. 1, January 1986, pp. 96-109.

(10) Dunham, Janet R., "Experiments in software reliability for life-critical applications", *IEEE Transactions on Software Engineering*, Vol. 12, No. 1, January 1986, pp. 110-123.

(11) Card, David N., "Empirical study of software design practices", *IEEE Transactions on Software Engineering*, Vol. 12, No. 2, February 1986, pp. 264-271.

(12) Basili, Victor R., "Experimentation in software engineering", *IEEE Transactions on Software Engineering*, Vol. 12, No. 7, July 1986, pp. 733-743.

(13) Fagan, Michael E., "Advances in Software inspections", *IEEE Transactions on Software Engineering*, Vol. 12, No. 7, July 1986, pp. 744-751.

(14) Bloomfield, Robin E., "Formal methods applied to assessment of high-integrity software for nuclear reactor protection", *IEEE Transactions on Software Engineering*, Vol. 12, No. 9, September 1986, pp. 988-993.

(15) Rombach, H. Dieter, "Impact of software structure on maintainability; controlled experiment", *IEEE Transactions on Software Engineering*, Vol. 13, No. 3, March 1987, pp. 344-354.

(16) Card, David N., "Evaluating software engineering technologies; methodology and evaluation of various technologies on 22-project sample", *IEEE Transactions on Software Engineering*, Vol. 13, No. 7, July 1987, pp. 845-851.

(17) Selby, Richard W., V.R. Basili, and F.T. Baker, "Cleanroom software development; an empirical evaluation", *IEEE Transactions on Software Engineering*, Vol. 13, No. 9, September 1987, pp. 1027-1037.

(18) Baker, C. T., "Effects of field service on software reliability", *IEEE Transactions on Software Engineering*, Vol. 14, No. 2, February 1988, pp. 254-258.

(19) Yu, Tze-Jie, "Analysis of software defect models using data from large commercial projects", *IEEE Transactions on Software Engineering*, Vol. 14, No. 9, September 1988, pp. 1261-1270.

(20) Munoz, Carlos Urias, "Testing large software products using combination of techniques", *IEEE Transactions on Software Engineering*, Vol. 14, No. 11, November 1988, pp. 1589-1596.

(21) Lew, Ken S., "Software complexity and its impact on software reliability", *IEEE Transactions on Software Engineering*, Vol. 14, No. 11, November 1988, pp. 1645-1655.

(22) Software Engineering Laboratory, "Collected Software Engineering Papers: Volume VII", SEL-89-006, Goddard Space Flight Center, Greenbelt, Maryland, November 1989.

(23) Myers, E.A., "Phased inspections and their implementation", M.S. Thesis, University of Virginia, Department of Computer Science, March 1991.

DISTRIBUTION LIST

1 - 3 National Aeronautics and Space Administration
Langley Research Center
Hampton, VA 23665

Attention: D. E. Eckhardt, Jr.
ISD, M/S 478

4 - 5* National Aeronautics and Space Administration
Scientific and Technical Information Facility
P. O. Box 8757
Baltimore/Washington International Airport
Baltimore, MD 21240

6 National Aeronautics and Space Administration
Langley Research Center
Acquisition Division
Hampton, VA 23665

Attention: Richard J. Siebels
Grants Officer, M/S 126

7 - 8 E. H. Pancake, Clark Hall

9 - 10 J. C. Knight, CS

11 A. K. Jones, CS

12 SEAS Preaward Administration Files

*One reproducible copy

JO#3690:ph