Technical Report TR-1519 July 1985

Quantitative Evaluation of Software Methodology

Victor R. Basili

Department of Computer Science
University of Maryland
at College Park

Abstract

This paper presented a paradigm for evaluating software development methods and tools. The basic idea is to generate a set of goals which are refined into quantifiable questions which specify metrics to be collected on the software development and maintenance process and product. These metrics can be used to characterize, evaluate, predict and motivate. They can be used in an active as well as passive way by learning from analyzing the data and improving the methods and tools based upon what is learned from that analysis. Several examples were given representing each of the different approaches to evaluation.

This study is funded by NASA grant NSG-5123 and Air Force Research Grant AFOSR F49620-80-C-001. This paper will be presented at the First Pan Pacific Computer Conference, Melbourne, Australia, September 10-13, 1985.

Quantitative Evaluation of Software Methodology

Victor R. Basili

Department of Computer Science
University of Maryland

INTRODUCTION

One of the major problems in the development of software is the lack of management's ability to (1) find criteria for selecting the appropriate methods and tools to develop and maintain software and (2) evaluate the goodness of the software product or process. In a survey of the software development industry, [Thayer and Pyster 1980] listed the twenty major problems reported by software managers. Of these twenty, over half (at least thirteen) delineated the need for management to find selection criteria for the choice of technology or be able to judge the quality of the existing software development process or product. In some sense this may have been a surprise. Management's priority was not to ask for new technology but they wanted to find out how to use the existing technology. This is in fact a major aspect of the technology transfer problem.

For many cases, there does exist a fair amount of technology available for software development. However, it is not always apparent to the manager which of these techniques or tools to invest in, and whether or not they are working as predicted for the particular project. What is needed in almost all cases is a quantitative approach to software management and engineering that uses models and metrics for the software development process and product. There are such models and metrics available. They cover everything from resource estimation and planning to the complexity of the product.

This quantitative methodology is needed for understanding, comparing, evaluating, predicting, motivating, and good management practices. In many cases, it is still a primitive technology and should be used by management and engineering as a tool to augment good judgement, not to replace it. Typically, we need to establish the validity of the models and metrics in the individual environments to be sure that they capture the appropriate activities.

METHODOLOGY LEVELS

Before I discuss the available models and metrics for quantitative management and engineering, I will begin with the issue of methodology. There are various levels at which the software development process can be viewed. At the top most level, we often will think about a particular technique, some approach to solving a specific aspect of the software development problem. For example, structured coding is a mechanism for developing code in a particular programming language using a select set of control structures. It is a logically sound approach to code development since it allows ease of testing, readability, and permits the use of a checkable standard.

Unfortunately, it was thought of as the solution to the software development problem back in the 1960's. That now appears rather naive given what we know about software development. Structured coding is clearly only one part of the software development process, attacking only one phase of the process and a single product, the code. Taken in isolation it can even cause a problem. Given an unstructured design, it would be very difficult for the coder to redesign at the code level. If the project is not performing inspections or doing reading or writing tests based upon the structure of the code, then many of the benefits of structured coding are lost. Thus, the technique of structured coding, used in isolation can be a drawback and even increase the cost of a project.

The problem is that one cannot take a method or tool and place it into a foreign environment and expect it to work. What is needed, as we now understand, is an integrated set of methods and tools that work together across the whole life cycle. The use of structured coding in conjunction with structured design, a structured process design language, and reading techniques, have been shown to pay off well. What we want is an integrated set of techniques that provide a methodology for software development across the entire life cycle. Tools should be provided, whenever possible to support the methods.

Unfortunately, this is still not 'the solution'. An integrated set of methods must by definition be an abstraction. These techniques must be engineered for a particular environment. In this sense, software engineering involves the application of an integrated set of techniques to a specific project, with its unique problems, constraints, and environment. This approach requires an understanding of the project and the environment in which it is to be developed so that the right set of techniques can be (1) chosen from the integrated set and (2) refined for the environment.

The following are examples of both choosing the appropriate techniques and refining them. An integrated set of techniques does not mean a standard fixed set. An integrated set should mean a set of techniques from which the manager may choose the most appropriate given the project characteristics, knowing that whatever set is chosen they will interface well with one another. For example, suppose the project is one in which the developer has very little experience, and the requirements will be changing on a regular basis. Then one should choose a subset of techniques that lend themselves to a changing environment. This calls for an evolutionary approach, such as iterative enhancement [Basili and Turner 1975], in which the developer builds subset versions of the product, evaluating each of the subsets as it is completed. Clearly, the standard waterfall model would not be effective in this environment. However, many techniques, such as structured design and coding within a version, are useful.

An example of the refinement of a technique might be based upon the history of errors. Knowing the error pattern in a particular environment, e.g. 40% of the errors are errors of omission and 60% errors of commission, then reading the design without having the requirements document available might miss as much as 40% of the errors. Thus the reading approach would require that consistency checks between documents always be done. The error pattern always warns about total

reliance on a structural testing technique. If it were known that 10% of the errors were due to failure to initialize variables, then the readers could be advised to check for the initialization of variables in their reading.

In either case, it is apparent that the more we know about our environment, the better we can choose and tailor the appropriate techniques for development and maintenance.

MODELS AND METRICS

In order to evaluate the methods being used, we must first understand the software development process and product. This requires hypothesizing models. A model is simply an abstraction of a real world process or product. It attempts to explain what is going on by making assumptions and simplifying the environment. It gives a viewpoint of the software development process or product by classifying various phenomena, abstracting from reality, and isolating the aspects of interest. There may be many models of the same thing, each attempting to analyze a different aspect. The thing being modeled may then be described as the sum of all the models or viewpoints. There are models which take the viewpoint of resource use, complexity, reliability, change, etc. Based upon the models, there are metrics which are simply quantitative measures of the extent or degree to which the software possesses and exhibits a certain characteristic, quality, property, or attribute. These metrics provide us with measurements: numbers with an associated unit of measure which describe some aspect of the software.

Metrics can be viewed in many ways [Basili 1981]. They can be thought of as **objective** or **subjective.** Objective metrics are absolute measures taken on the product or process, e.g. the time for development, the number of lines of code, the number of errors or changes. Subjective metrics are an estimate of the extent or degree in the application of some technique, or the classification or qualification of a problem or experience, usually done on a relative scale. Here there is no exact measurement but an opinion or consensus of opinions. Examples include a rating on the use of a process design language (PDL) or a rating of the experience of the programmers in an application.

Typically a subjective metric is used when we do not know how to quantify an objective metric. For example, it is difficult to define an objective metric for how well a PDL was used in the development of a project. However, if we are to evaluate the effect of the PDL we need to know whether the technique was used well or not, so that its effect can be judged appropriately. Even though we cannot come up with an objective rating, we can ask two or three people to rate the use based upon some rating scale, e.g.

0 - wasn't used at all,
1 - used only partly and as a coding specification
2 - used almost everywhere but as a coding specification
3 - used at a higher level than as a coding specification
4 - used at multiple levels of specification with limited success
5 - used effectively at multiple levels of design

Although the rating will not be exact, it will provide reasonable subjective information that could not be available otherwise. Sometimes there is an objective metric we can use, but it is less accurate than the subjective information. For example, to evaluate the experience of a programmer in an application, an objective metric might be years of experience. However, several studies have shown that years of experience is not a reliable metric past two or three years. A subjective rating by management and colleagues would probably be a more accurate measure.

Metrics can be measures of the product or the process. A product metric would be a measure of the actual product developed, e.g. source code, object code, documentation, etc. Sample metrics are lines of code (an objective metric) and readability of the source code (a subjective metric). A process metric would be a measure of the process model used for developing the product. Sample product metrics would be the use of a methodology (a subjective metric) and the effort for development in staff months (an objective metric).

Metrics can be used to measure cost or quality. A cost measure is some expenditure of resources in dollars including capital investment usually normalized according to some value component. For example, staff months, computer use, size per time slice. A quality measure represents some form of value of the product. For example, reliability, ease of change, correctness, number of errors remaining, amount of code reusable. Actually cost can be considered a quality metric since low cost might be thought of as a valuable quality. However, we typically are trying to maximize quality and minimize cost so it is interesting to see them as separate types of metrics useful in tradeoffs.

There are several general uses of metrics. First and most important, metrics can be used to characterize and understand. A characterizing metric is one that helps distinguish the process or product or environment. For example, the use of a methodology, the number of externally generated changes, or the size. Each of these tell us something about the project so that we can better understand it. Characterizing metrics can be used for schedule tracking, providing information on where the project stands with respect to percent of resource use, with respect to calendar time, etc. They can be used to help define the model of the process or the product.

Metrics can be used for evaluation. The metric is a good evaluative measure if it correlates with or shows directly the quality of the process or product, e.g. the number of errors reported during acceptance testing or work productivity. Where almost all metrics can be used for characterization, only a subset can be used for evaluation. The schedule tracking metrics mentioned above can be used for evaluation, only if we know the planned schedule is reasonable. If it is, we can use conformance to schedule as a means of evaluating the effect of the methods used.

Metrics can be used for prediction and estimation. A predictive metric is one that is estimable or calculable at some point in time and can be used to predict some information at a later point in time. For example, estimating size as a predictor of effort is a standard predictive relationship. It becomes interesting to try to establish metrics such as the use of a particular methodology as a metric that predicts (correlates) with various aspects of quality, e.g. ease of modification.

Metrics can be used for motivation. Letting the developers know what is important in a quantitative way defines what it is we are looking for. For example, one of the major issues in software productivity is the need for reusability. However, management does not motivate reusability, it actually unknowingly discourages it. By using schedule and cost as the primary motivators for success, it discourages a manager from using extra time or money that might make parts of the product reusable. If reusability were listed as one of the prime motivators, to be traded off with cost and schedule, we might see more reusability. For example, we can motivate a project manager to try to develop reusable design or code by rewarding him/her for all code that gets used in another project. This would help encourage the manager to consider tradeoffs of reuse with time and cost. Another manager might be motivated to reuse someone else's code by rewarding him/her by counting any reused code as part of their total source code count or even adding extra rewards for reuse. Motivational metrics need to be carefully thought out, i.e. we need to be sure we want what we are asking for. But even the generation of such metrics helps us better understand what we are telling managers versus what we should be telling managers, i.e. what are the actual goals of the company and the project.

MEASUREMENT AND EVALUATION PARADIGM

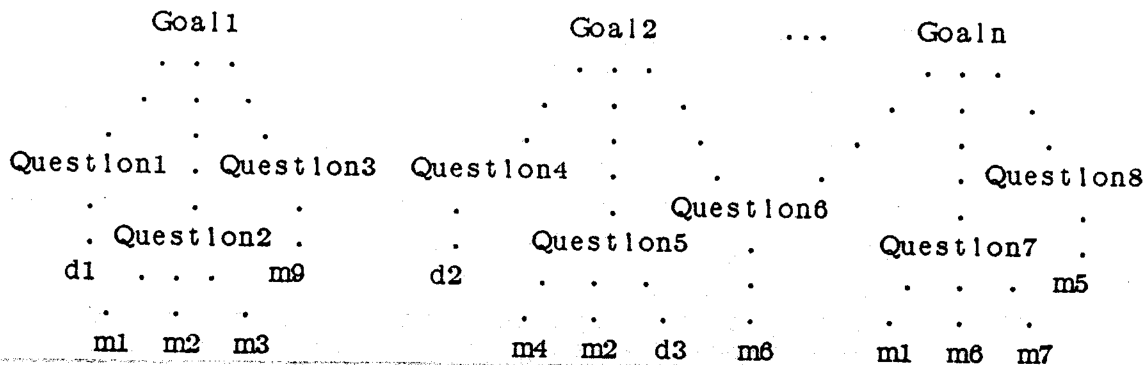
The measurement and evaluation process requires a mechanism for determining what data is to be collected; why it is to be collected; and how the collected data is to be interpreted [Basili & Weiss 1984]. The process requires an organized mechanism for determining the purpose of the measurement; defining that purpose in a traceable way into a quantitative set of questions that define a specific set of data for collection. The purpose of the measurement and evaluation flows from the needs of the organization. These may include: the need to evaluate some particular technology; the need to better understand resource utilization in order to improve cost estimation; the need to evaluate the quality of the product in order to determine when to release it; or the need to evaluate the benefits and drawbacks of a research project.

The goals tend to be vague and ambiguous, often expressed at an imprecise level of abstraction. For example, the words understand, evaluate, quality, benefits, and drawbacks carry different meanings to different people or vary with different environments. The need to better understand resource utilization in order to improve the cost estimation process explains what I want to do but leaves many questions about what kind of data needs to be collected. The need to evaluate the use of a technology, like design inspections, requires the perspective of the expectations from the methodology as does the evaluation of a research project. The goals need to be carefully articulated but also refined in a quantitative way in order to give precision and to clarify their meaning with respect to the particular environment.

The data collection process itself requires a basic paradigm that traces the goals of the collection process, i.e. the reasons the data are being collected, to the actual data. It is important to make clear at least in general terms the organization's needs and concerns, the focus of the current project and what is expected from it. The

formulation of these expectations can go a long way towards focusing the work on the project and evaluating whether the project has achieved those expectations. The need for information must be quantified whenever possible and the quantification analyzed as to whether or not it satisfies the needs. This quantification of the goals should then be mapped into a set of data that can be collected on the product and the process. The data should then be validated with respect to how accurate it is and then analyzed and the results interpreted with respect to the goals.

The actual data collection paradigm can be visualized by a diagram:



Here there are n goals shown and each goal generates a set of questions that attempt to define and quantify the specific goal which is at the root of its goal tree. The goal is only as well defined as the questions that it generates. Each question generates a set of metrics (mi) or distributions of data (di). Again, the question can only be answered relative to and as completely as the available metrics and distributions allow. As is shown in the above diagram, the same questions can be used to define different goals (e.g. Question6) and metrics and distributions can be used to answer more that one question. Thus questions and metrics are used in several contexts.

The paradigm is important not just for focusing management, engineering, and quality assurance interests but also for interpreting the questions and the metrics. For example, m6 is collected in two contexts and possibly for two different reasons. Question6 may ask for the size of the product (m6) as part of the goal to model productivity (Goal2). But m6 (size of the product) may also be used as part of a question about the complexity of the product (e.g. Question7) related to a goal on ease of modification (e.g. Goaln).

If a measure cannot be taken but is part of the definition of the question, it is important that it be included in the goal/question/metric paradigm. This is so that the other metrics that answer the question can be viewed in the proper context and the question interpreted with the appropriate limitations. The same is true for questions being asked that may not be answerable with the data available. For example, to determine the effectiveness of a method in reducing errors, I need to know the total number of faults over the system life time. I cannot know this number during the development phase. I should still include the metric in the paradigm so that I know the information is incomplete.

It could then be assumed that although there may be many goals and even many questions, the metrics do not grow as the same rate as the goals and questions. Thus a set of metrics could be collected for characterizing the software process and product that will allow many questions generated by different goals to be answered.

Given the above paradigm, the data collection process consists of six steps:

1. Generate a set of goals based upon the needs of the organization.

The first step of the process is to determine what it is you want to know. This focuses the work to be done and allows a framework for determining whether or not you have accomplished what you set out to do. For example, the organization may wish to know whether the use of a specific method or tool improves the productivity of the project personnel or the quality of the product. It may wish to define a set of goals for a research project and then determine whether that project has achieved those goals. The goal may be simpler. It may be to characterize the resource usage across the project. In any case the goals should be clearly stated. The goals do not have to be quantifiable. It is the next step in the process to take the goal and make it measurable.

It is difficult to provide an organization with a set of guidelines for generating goals. These should be based upon the particular needs and concerns of the organization and its purpose for beginning a data collection activity. The goals can be management oriented, engineering oriented, quality assurance oriented or even research oriented. As stated above, many of the questions or metrics may be the same for the different orientations but they may be combined in different ways and the interpretation will have a different focus and impact.

Management oriented goals will typically deal with resource allocation and monitoring for the purpose of prediction and estimation. For example managers may wish to estimate cost, track resource expenditures, and predict the quality of the project. An engineering orientation may be to evaluate the technology being used in the development of the project, discover the problems in terms of errors and resource use in order to improve the quality of the process or the product. A quality assurance orientation may be to characterize the product or even the process to judge adherence to standards, isolate parts of the product that require rework, or evaluate the product for delivery. A research orientation may be to focus on the benefits and drawbacks of the development of a new technology and demonstrate its effectiveness. Each of these orientations have goals in common. It is the interpretation that may be different. Many of the questions and metrics (e.g. about resource allocation) will be replicated for different goals so that the same data can answer many questions and allow for the achievement of many goals.

The goals to characterize, evaluate and predict aspects of the software process and product cover a large area. We can set goals to characterize the effort expended, the changes generated, the errors committed, the dimensions of the products such as size and complexity at various points in time, the methods and tools, the documentation, the application, the experience of the developers, the computer and the constraints set on the project, and the various execution time issues such as performance, space utilization, and test coverage. We can set goals to evaluate the

effectiveness of the tools and methods used, the environment in which the product is developed, and even the models for the process and product. We can set goals to predict the cost, reliability or quality of the product.

2. Derive a set of questions of interest or hypotheses which quantify those goals.

The goals must now be formalized by making them quantifiable. This is the most difficult step in the process because it often requires the interpretation of fuzzy terms like quality or productivity within the context of the development environment. These questions define the goals of step 1. The aim is to satisfy the intuitive notion of the goal as completely and consistently as possible. For example with the above goal of characterizing resource usage across the project, questions of interest might be: How much time (in minutes, hours, weeks, months or years) was spent by all personnel of interest (programmer, librarian, support staff, managers, reviewers, etc.) in total and across subcategories, in each phase (requirements, specification, design, code, test, and operation) or activity (training, reviewing, making changes, etc.) for each product part (module, subsystem, full system)? How much computer time was spent by all personnel of interest in total and across all subcategories, for each phase or activity, for each product part? These questions actually generate sets of questions parameterized by each of the subcategories above.

After all possible resource usages have been defined and transposed into questions, the questions posed must be evaluated as to whether they provide a complete definition of the goal. This process is a heuristic one and the judgement of whether or not the goal is satisfied by the questions will be subjective. The process is often iterative and after collecting resource characterization data the collector may discover new questions that were missed. These could then be added to the question list for later projects. It might even be possible that the data has been collected to answer these questions because it was collected to answer another question. However before applying the data directly, the question/metric paradigm should be developed to assure proper interpretation of the question.

It will often be the case that the set of questions do not fully satisfy the goal. This may be because we do not know how to phrase a question in a quantifiable way or because we cannot interpret the fuzzy terms of the goal in a well defined way or the cost for collecting the data may not be worth it for the achievement of the goal. In these cases the missing aspects of the goals should be noted so that later interpretations of the results can be qualified appropriately.

3. Develop a set of data metrics and distributions which provide the information needed to answer the questions of interest.

In this step, the actual data needed to answer the questions are identified and associated with each of the questions. In the above example this is a simple count of people and computer time by the various subcategories. However, the identification of the data categories is not always so easy. Sometimes new metrics or data distributions must be defined. Other times data items can be defined to answer only part of a question. In this case, the answer to the question must be qualified and interpreted in the context of the missing information. As the data items are identified,

thought should be given to how valid the data item will be with respect to accuracy and how well it captures the specific question.

These data items may be objective or subjective. If they are subjective, some mechanism must be defined for quantifying the evaluation, e.g. an integer scale of 0 to 5, and eliminating variations in judgement, e.g. a consensus of three people.

4. Define a mechanism for collecting the data as accurately as possible

The data can be collected via forms, interviews, or automatically by the computer. If the data is to be collected via forms, they must be carefully defined for ease of understanding by the person filling out the form and clear interpretation by the analyst. An instruction sheet and glossary of terms should accompany the forms. Care should be given to characterizing the accuracy of the data and defining the allowable error bounds.

5. Perform a validation of the data

The data should always be checked for accuracy. Forms should be reviewed as they are handed in. They should be read by a data analyst and checked with the person filling out the form when questions arise. Sample sets should be set to determine accuracy the data as a whole. As data is entered into the data base, validity checks should be made by the entering program. Redundant data should be collected so checks can be made.

The validity of the data is a critical issue. Interpretations will be made that will effect the entire organization. One should not assume accuracy without justification.

6. Analyze the data collected to answer the questions posed

The data should be analyzed in the context of the questions and goals with which they are associated. Missing data and missing questions should be accounted for in the interpretation.

The process is top down, i.e before we know what data to collect we must first define the reason for the data collection process and make sure the right data is being collected, and it can be interpreted in the right context. To start with a set of metrics is working bottom up and does not provide the collector with the right context for analysis or interpretation.

EXAMPLE TECHNIQUE EVALUATION

As an example consider the goal of evaluating the effectiveness of a method such as design inspections. This appears to be a clearly stated goal at first but the goal does not say with respect to what are we to evaluate the technology. Let us help define this better by asking a set of questions.

Question 1: How well were the inspections performed? Use a subjective rating 0 to 5.

This question provides us with a basis for evaluation. We would not like to evaluate the technical benefits of the method if it was not applied well. We may even wish to rate how well different aspects of the technique were applied. This

rating might be done by the moderator, a project person and the instructor of the technique.

Question 2: How many errors were uncovered? Characterize the errors by different classification categories.

This might tell us whether the technique is better at finding certain kinds of errors and if we have any history of other projects as a basis, it can tell us whether we are doing better or worse than the norm.

Question 3: How much calendar time was spent?

This question addresses the cost of applying the technique. For example we might wish to analyze the effect on the schedule.

Question 4: How many staff hours were spent?

This question addresses the cost and resources spent. We can compare the number of hours spent finding errors in this way to the various testing techniques used.

Question 5: What percent of the errors were found?

We will not fully be able to answer this question until the product has been in the field for several years but at each milestone, e.g. acceptance test, one year in the field, etc. We will be better able to understand the effectiveness of the technique.

Question 6: What was the cost of error isolation? error fix?

This question allows us to analyze the cost of discovering and fixing errors during inspections as opposed to during testing.

etc.

There are many more questions we might ask based upon what it is we want to know. As stated above, these questions permit us to better define the goals, help us to specify what data needs to be collected (e.g. subjective ratings on how well the method was applied, error counts and distributions, effort in inspection by person by activity), and how the data should be interpreted (e.g. we may not be able to judge the total effectiveness until the project has been out in the field for a while).

METHODOLOGY IMPROVEMENT PARADIGM

All this leads us to the following basic paradigm for evaluating and improving the methodology used in the software development and maintenance process.

1. Characterize the approach/environment.

This step requires an understanding of the various factors that will influence the project development. This includes the problem factors, e.g. the type of problem, the newness to the state of the art, the susceptibility to change, the people factors, e.g. the number of people working on the project, their level of expertise, experience, the product factors, e.g. the size, the deliverables, the reliability requirements, portability requirements, reusability requirements, the resource factors, e.g. target and development machine systems, availability, budget, deadlines, the process and tool factors, e.g. what techniques and tools are available, training in them,

programming languages, code analyzers.

2. Set up the goals, questions, data for successful project development and improvement over previous project developments.

It is at this point the organization and the project manager must determine what the goals are for the project development. Some of these may be specified from step 1. Others may be chosen based upon the needs of the organization, e.g. reusability of the code on another project, improvement of the quality, lower cost.

3. Choose the appropriate methods and tools for the project.

Once it is clear what is required and available, methods and tools should be chosen and refined that will maximize the chances of satisfying the goals laid out for the project. Tools may be chosen because they facilitate the collection of the data necessary for evaluation, e.g. configuration management tools not only help project control but also help with the collection and validation of error and change data.

4. Perform the software development and maintenance, collecting the prescribed data and validating it.

This step involves the collection of data by forms, interviews, and automated collection mechanisms. The advantages of using forms to collect data is that a full set of data can be gathered which gives detailed insights and provides for good record keeping. The drawback to forms is that they can be expensive and unreliable because people fill them out. Interview can be used to validate information from forms and gather information that is not easily obtainable in a form format. Automated data collection is reliable and unobtrusive and can be gathered from program development libraries, program analyzers, etc. However, the type of data that can be collected in this way is typically not very insightful and one level removed from the issue being studied.

5. Analyze the data to evaluate the current practices, determine problems, record the findings and make recommendations for improvement.

This is the key to the mechanism. It requires a post mortem on the project. Project data should be analyzed to determine how well the project satisfied its goals, where the methods were effective, where they were not effective, whether they should be modified and refined for better application, whether more training or different training is needed, whether tools or standards are needed to help in the application of the methods, or whether the methods or tools should be discarded and new methods or tools applied on the next project.

6. Proceed to step 1 to start the next project, armed with the knowledge gained from this and the previous projects.

This procedure for developing software has a corporate learning curve built in. The knowledge is not hidden in the intuition of first level managers but is stored in a corporate data base available to new and old managers to help with project management, method and tool evaluation, and technology transfer.

CASE STUDIES OF METHODOLOGY EVALUATION

With all the different methods and tools available, we need to better quantitatively understand and evaluate the benefits and drawbacks of each of them. There are several different approaches to quantitatively evaluating methods and tools: blocked subject-project, replicated project, multi-project variation, and single project case study [Basili & Selby 84]. The approaches can be characterized by the number of teams replicating each project and number of different projects analyzed as shown in Table 1.

		# of projects: one	# of projects: more than one
# of teams per project	one	single project	multi-project variation
	more than one	replicated project	blocked subject-project

TABLE 1

The approaches vary in cost and the level of confidence one can have in the result of the study. Clearly, an analysis of several replicated projects costs more money but will generate stronger confidence in the conclusion. Unfortunately, since a blocked subject-project experiment is so expensive, the projects studied tend to be small. The size of the projects increase the costs go down so is possible to study very large single project experiments and even multi-project variation experiments if the right environment can be found. In what follows, at least one example of each of these approaches will be given as performed by the Laboratory for Software Engineering Research (LASER) at the University of Maryland.

METHODOLOGY EVALUATION USING BLOCKED SUBJECT-PROJECT ANALYSIS

This type of analysis allows the examination of several factors within the framework of one study. Each of the technologies to be studied can be applied to a set of projects by several subjects and each subject applies each of the technologies under study. It permits the experimenter to control for differences in the subject population as well as study the effect of the particular projects.

The sample study discussed here is a testing strategies comparison [Basili & Selby 85]. The goal was to compare the effects of code reading, functional and structural testing with respect to 1) fault detection effectiveness, 2) fault detection cost,

and 3) classes of faults detected. A secondary goal was to compare the performance of software type and expertise level but only the first goal will be discussed here.

The experimental approach involved three replications of the experiment using 74 subjects on four different projects. The projects were a text formatter, a plotter, an abstract data type, and a database program varying in length between 145 and 365 lines of code. The programs each contained software faults (9, 6, 7, 12 respectively) that were either made during the actual development of the program or were seeded based upon characteristic faults found in the local environment. The experimental design was a fractional factorial design blocked according to experience level and the program tested. Each subject used each technique and tested each program.

Two of the questions generated from this study were:

Question 1: Which of the validation techniques detects the greatest number of faults in the programs?

The data collected for this question is the number of faults found in each project by each subject. The results of the study were that 4 faults were found on the average and that code reading was more effective than both testing techniques but functional testing was more effective than structural testing. Reading found 5.1 faults on the average, functional testing found 4.5 faults on the average and structural testing found 3.3 faults on the average.

Question 2: Which of the techniques has the highest fault detection rate (number of faults detected per hour)?

The data collected to answer this question was the number of faults found and the time spent by the subject in detecting faults. The results were that code reading was more cost effective than functional and structural testing. Code reading found 3.3 faults per hour on the average while each of the testing techniques found 1.8 faults on the average.

Because of the experimental design of this type of analysis there were many other questions that were posed and answered by this experiment, e.g. Is the fault detection rate dependent on the type of software? Is the number of faults observed dependent on the type of software? Do the methods tend to capture different classes of faults? What classes of faults are observable but go unreported?

The experimental design for this study permits a great amount of statistical analysis and provide the experimenter with a fair amount of latitude in studying the different aspects of the project. The drawbacks to the study are that the projects studied are small module size projects and the results do not necessarily scale up to the acceptance test phase of very large projects. The interpretation is more accurate for the unit test phase. The study does not provide sufficient insight into how the techniques might work on larger projects. This drawback is of necessity because the cost of replication is too expensive.

METHODOLOGY EVALUATION USING REPLICATED PROJECT ANALYSIS

The replicated project analysis involves several replications of the same project by different subjects. Each of the technologies to be studied is applied to the project by several subjects but each subject applies only one of the technologies. It permits the experimenter to establish control groups.

The goal of the sample study was to quantitatively evaluate the effect of a disciplined approach to software development [Basili & Reiter 81]. The disciplined approach included the use of an integrated set of techniques that included top down design, a process design language, walk-throughs, chief programmer teams, and the use of a librarian.

The experimental approach involved the replication of the same project by 19 teams, including 7 three person disciplined teams (DT), 6 three-person ad hoc teams (AT), and 6 ad hoc individuals (AI). The project was to build a compiler for a small language, anticipating about 1200 source lines of code in a high level language. All the data was collected automatically so that the subjects did not know what was being measured. The drawback to this is that the information was typically one level removed from what we really wanted to know. The statistical analysis performed were the non-parametric Mann-Whitney U and Kruskal-Wallis H tests.

Specific questions included:

Question 1: Does a disciplined approach reduce the average cost and complexity of the process?

The data collected was a count of the (1) number of job steps, i.e. any aspect of computer access such as module compilations and program executions, and (2) program changes, i.e. the number of changes to a program that indicated an error or omission. Job steps were used to represent effort and program changes were used to represent errors.

The results of the study showed that for all categories of job steps and program changes, the disciplined teams had statistically less of both than either the ad hoc teams or the ad hoc individuals.

Question 2: Does a disciplined team behave more like an individual programmer than a team in terms of the resulting product? This was an attempt to measure conceptual integrity.

The data collected here was various product measures such as size (number of segments, number of lines of code, number of decisions) and complexity, e.g. a comparison of cyclomatic complexity [McCabe] for the top quartiles of modules.

The results of this study showed that the ad hoc individuals had a smaller number of segments than either the disciplined teams or the ad hoc teams. The ad hoc individuals had less lines of code than the disciplined teams which had less lines of code than the ad hoc teams, and the ad hoc individuals and disciplined teams had less decision than the ad hoc teams. Comparing the cyclomatic complexity of the modules in the upper quartiles, the results were that the disciplined teams created the least complex projects and the ad hoc individuals the most complex project with the ad hoc teams lying in between, depending upon the mechanism for counting decisions.

Thus it was felt that the questions were both answerable in the affirmative. The benefit of the study is that the results were soundly supported statistically because of the number of replications and the projects were of a more reasonable size than the modules studied in the testing experiment. The drawback to this study again is that the projects were still smaller than many projects one might encounter and it is not clear that the results would still hold if the project sizes were increased by an order of magnitude.

METHODOLOGY EVALUATION USING MULTI-PROJECT VARIATION ANALYSIS

Multi-project variation analysis involves the measurement of several projects where controlled factors such as methodology can be varied across similar projects. This is not a controlled experiment as the previous two approaches were, but allows the experimenter to study the effect of various methods and tools to the extent that the organization allows them to vary on different projects.

The goal of this sample study was to examine the relationship between methodology and various factors such as productivity and quality. [Bailey & Basili 1981], [Basili & Bailey 1980], [Basili 1981]. The study was conducted in the Software Engineering Laboratory, at joint project between NASA Goddard Space Flight Center, the University of Maryland, and Computer Sciences Corporation.

The approach was to study a series of projects that involve ground support software for satellites. Each project was rated with respect to a large set of factors, covering environment, methodology, experience, performance, etc. When the metrics were subjective they were given on a six point scale, e.g. rating on the basis of the use of a methodology.

The methodology factors used in the study were very similar to the methodology factors used in the replicated project study discussed above. This allowed us to see if the methods could work on larger projects than in the controlled study. This has been a common mechanism in the Laboratory for Software Engineering Research. We run both controlled experiments on small projects and case studies or multi-project analysis on large projects to verify the effects of the technologies. The combination of both approaches provides us with a deeper confidence that the technologies are effective as well as allowing us to understand their effects in different environments.

The three major questions asked in this study were:

Question 1: Did the projects with a high methodology use come from a different population than those projects with a low or medium methodology use?

Question 2: Do any other factors or sets of factors show a significant effect on productivity?

Data used to answer these questions were lines of source code per staff month for productivity and such factors as customer interface complexity; customer originated program design changes; the complexity of such things as the application, the program flow, the internal communication, the external communication, the data

base; constraints such as I/O capability, timing, main storage; programming group experience such as machine familiarity, language familiarity, application experience; hardware changes during development.

The approach to answering these first two questions was based upon a similar type of study at IBM/FSD [Brooks 1981]. A statistical test was performed to see if projects with high methodology came from a different environment with respect to productivity than projects with a low methodology use. The data used was based upon a relative ranking rather than an absolute rating. The approach was to divide the ratings for each technique into 3 categories: low (-1), medium (0), high (1). This was done to offset differences in scales. The ratings were added to get a cumulative methodology rating. The projects were then divided into groups based upon their rating and analyzed using the Mann-Whitney U test.

In analyzing the relationship between productivity and various factors, no significant relationship was found between productivity and size. However there were statistically significant results in demonstrating that those projects with high methodology use came from a different (and much higher) productivity population than those projects with low or even medium methodology use. So the answer to the first question was yes. The answer to this question was no.

Question 3: What are the factors that predict quality?

The metrics were compressed into three factors: quality, methodology and complexity. Methodology and complexity were not significantly correlated. Quality was significantly correlated with methodology ($r = .67$) and complexity ($r = -.64$) at less than quality, we got an R**2 of .45. Using the methodology and complexity metrics to predict quality we got an R**2 of .65. Based upon this study, it was clear that quality can be predicted from the use of methodology.

The benefit to this approach is that it does not require special experimental projects but allows for the evaluation of methodology in the normal development environment. The improvement algorithm discussed earlier can be applied to the environment in order to improve both the productivity and the quality of the software.

However, there are several drawbacks to the approach. First, it requires that there is enough differences in the projects use of methodology and there are enough projects using each of the methods, i.e. there must be enough of a sampling to generate a statistical result. Second, since the experiment is not controlled, there is always the possibility of making mistakes in the interpretation, i.e. other factors that have not been controlled for may be causing the differences in productivity or quality. Third, if the methodology improvement paradigm is being used, we are losing our control group of projects where little or no methodology is being used.

METHODOLOGY EVALUATION USING SINGLE PROJECT/CASE STUDY ANALYSIS

Unfortunately, this is where most methodology evaluation begins. There is a project and the management has decided to make use of some new method or set of methods and wants to know whether or not the method generates any improvement

in the productivity or quality. A great deal depends upon the individual factors involved in the project and the methods applied.

This sample study had a set of goals that dealt with the effectiveness of certain development techniques; information hiding, abstract interfaces, and formal specifications, as well as the effectiveness of the data collection process [Basili & Weiss 1981]. The project involved was the redevelopment of the on-board operational flight program for the A-7 aircraft. The development was done at the Naval Research Laboratories in Washington D.C. The analysis reported here was done after the requirements document was baselined with the subgoal of trying to judge the effectiveness of the requirements document which was developed using a formal specification technique, a state machine model and abstract interfaces.

One of the subgoals was that the requirements document should be easy to change. Based upon that goal the following questions were generated.

Question 1: Is the document easy to change?

Question 2: Is it clear where a change has to be made?

Question 3: Are the changes that are likely to occur, predicted correctly?

Question 4: Are changes confined to a single section?

The data collected to answer these questions consisted of various distributions of data such as the types of changes, effort to change, confinement of changes and changes by section. Given the data distributions:

Types of Changes:
85% were original error corrections
6% were to complete or correct a previous change
2% were to reorganize
7% were other changes (none of which were more than 1%)

Effort to Change:
68% were trivial (less than 1 hour)
26% were easy (1 hour to 1 day)
5% were medium (1 day to 1 week)
0% were hard (1 week to 1 month)
1% were formidable (more than 1 month)

Confinement of Changes:
85% were to one section
15% were to more than one section

The following conclusions were drawn:

The document was not very hard to change since most of the changes were trivial or easy. The only formidable change involved the change of a coordinate system that the developers did not know and the time for the change included the learning of that coordinate system. It should be noted that that change was confined to one section.

Since most of the changes were confined to a single section of the report one might argue that the document was organized in a way that the likely changes were

predicted correctly, that it was clear where a change had to be made, and that the changes were confined to a single section.

So the conclusion was drawn that the document was easy to change. However, that conclusion is based on comparing the data with experience and intuition. Most experienced people who have seen the data agree that the requirements document was a successful development but there is no statistical evidence and there is no solid basis for comparison. If similar data had been collected from other similar projects, and we were able to do a comparison, as we did with the multi-project analysis, our confidence level in the results might have been higher.

SUMMARY AND CONCLUSION

This paper has presented a set of quantitative approaches to evaluating software development methods and tools. The basic idea is to generate a set of goals which are refined into quantifiable questions which specify metrics to be collected on the software development and maintenance process and product. These metrics can be used to characterize, evaluate, predict and motivate. They can be used in an active as well as passive way by learning from analyzing the data and improving the methods and tools based upon what is learned from that analysis. Several examples were given representing each of the different approaches to evaluation. The cost of the approaches varied inversely with the level of confidence in the interpretation of the results.

It is hoped that this paper has demonstrated that there are quantitative mechanisms for evaluating methodologies. These mechanisms can be used in industry and in the research laboratories to provide better insights into the benefits and weaknesses of technology.

ACKNOWLEDGEMENT

This research was supported in part by the National Aeronautics and Space Administration Grant NSG-5123 and by the Air Force Office of Scientific Research under Contract AFOSR-F49620-80-C-001 to the University of Maryland.

REFERENCES

[Bailey & Basili 1981]

John W. Bailey and Victor R. Basili, A Meta-Model for Software Development Resource Expenditures, Proceedings of the Fifth International Conference on Software Engineering, San Diego, California, pp 107-116, 1981.

[Basili 1981]

Victor R. Basili, Evaluating Software Development Characteristics: Assessment of Software Measures in the Software Engineering Laboratory, Proceedings of the Sixth Annual Software Engineering Workshop, December 1981.

[Basili & Bailey 1980]

Victor R. Basili and John W. Bailey, The Software Engineering Laboratory: Measuring the Effects of Software Methodologies within the Software Engineering Laboratory, Proceedings of the Fifth Annual Software Engineering

Workshop, November 1980.

[Basili & Reiter 1981]
Victor R. Basili and Robert W. Reiter, Jr., A Controlled Experiment Quantitatively Comparing Software Development Approaches, IEEE Transactions on Software Engineering, Vol. SE-7, No. 3, pp 299-320, May 1981.

[Basili & Selby 1984]
Victor R. Basili and Richard W. Selby, Jr., Data Collection and Analysis in Software Research and Management, Proceedings of the American Statistical Association, pp 21-30, 1984.

[Basili & Selby 1985]
Victor R. Basili and Richard W. Selby, Jr., Comparing the Effectiveness of Software Testing Strategies, University of Maryland Technical Report TR-1501, May 1985.

[Basili & Turner 1975]
Victor R. Basili and Albert J. Turner, Iterative Enhancement: A Practical Technique for Software Development, IEEE Transactions on Software Engineering, pp 390-396, December, 1975.

[Basili & Weiss 1981]
Victor R. Basili and David M. Weiss, Evaluation of a Software Requirements Document by Analysis of Change Data, Proceedings of the Fifth International Conference on Software Engineering, San Diego California, pp 314-323, March 9-12, 1981.

[Basili & Weiss 1984]
Victor R. Basili and David M. Weiss, A Methodology for Collecting Valid Software Engineering Data, IEEE Transactions on Software Engineering, Vol. SE-10, No. 3, pp 728-738, November 1984.

[Brooks 1981]
W. Douglas Brooks, Software Technology Payoff: Some Statistical Evidence, Journal of Systems and Software, Volume 2, Number 1, pp 3-10, February 1981.

[McCabe 1976]
Thomas J. McCabe, A Complexity Measure, IEEE Transactions on Software Engineering, pp 308-320, December 1976.

[Thayer & Pyster 1980]
Richard H. Thayer, Arthur Pyster, and Roger C. Wood, The Challenge of Software Engineering Project Management, IEEE Computer Magazine, pp 51-59, August 1980.