contest.

Members of the winning IBM team were Feng-hsiung Hsu, Murray S. Campbell and Arthur J. Hoane, Jr. Running five times faster than its predecessor, DEEP THOUGHT II was said to be capable of analyzing more than 7,000,000 chess positions per second. It's host was an IBM RISC System/6000 with 16 added chess processors.

Second-place ZARKOV's program was written by John Stanback of Hewlett Packard, running on an HP735 workstation. STAR-SOCRATES came from the National Center for Supercomputing in Champaign-Urbana, IL and ran on a 512-way Thinking Machines's CM-5. It tied for third with NOW that was written by Mark Lefler of Bryans Road, MD, and ran on a PC clone.

Design and Analysis in Software Engineering

Part 1: The Language of Case Studies and Formal Experiments

Shari Lawrence Pfleeger
Centre for Software Reliability
City University
Northampton Square
London EC1V 0HB England
+44-71-477-8426
fax: +44-71-477-8585
shari@csr.city.ac.uk

INTRODUCTION

Software engineers have many questions to answer. The test team wants to know which technique is best for finding faults in code. Maintainers seek the best tool to support configuration management. Project managers try to determine what types of experience make the best programmers or designers, while designers look for models that are good at predicting reliability. To answer these and other questions, we often rely on the advice and experience of others, which is not always based on careful, scientific research [Fenton et al. 1994]. As a software practitioner, it is important to make key decisions or assessments in an objective and scientific way. So we need to know two things: what assessment techniques are available to us, and which should we use in a given situation? Will Tracz has asked me to write an on-going column in *SIGSOFT Notes* to address these issues. In this first article, I will explain some of the terminology to be used in subsequent articles. In future articles, I and my invited colleagues will try to point out some of the key items affecting decisions about how to do research and how to evaluate the research of others. In the course of doing this, we will also show how software engineering research sometimes differs from research in other fields. For instance, in medicine, it is usually easy to test a new drug by giving a placebo to the control group. But in software engineering we cannot do that – we cannot ask a group not to use a design technique; the group has to use some technique if it is to complete the design, and we end up losing control rather than having a carefully-controlled comparison. As always, your comments are welcome, preferably through electronic mail; I will try to address in future columns the major points you may raise in your queries.

CHOOSING A RESEARCH TECHNIQUE

Suppose you are a software practitioner trying to evaluate a technique, method or tool. You have decided to investigate the tool, technique or method in a scientific way, rather than rely on "common wisdom." You can use three main types of assessment to answer your questions: case studies, formal experiments and surveys. Surveys are usually done after some use of the technique or tool has already taken place; they are retrospective. Case studies and formal experiments are planned in advance of the usage. Initially, we will look at primarily at case studies and formal experiments, saving a discussion of surveys for later columns.

Stating the hypothesis

The first step in your investigation is deciding what you want to investigate. That goal helps you to decide which type of research technique is most appropriate for your situation. The goal for your research can be expressed as a hypothesis you want to test. That is, you must specify what it is that you want to know or investigate. The hypothesis is the tentative theory or supposition that you think explains the behavior you want to explore. For example, your hypothesis may be "Using the ABC design method produces better quality software than using the XYZ design method." Whether you evaluate a "snapshot" of your organization using ABC (a case study) or do a carefully controlled comparison of those using ABC with those using XYZ (a formal experiment), you are testing to see if the data you collect will confirm or refute the hypothesis you have stated. Thus, wherever possible, you should try to state your hypothesis in quantifiable terms, so that it is easy to tell whether the hypothesis is confirmed or refuted. For instance, rather than saying "Using the ABC design method produces better quality software than using the XYZ method," you can define "quality" in terms of the defects found and restate the hypothesis as "Code produced using the ABC design method has a lower number of defects per thousand lines of code than code produced using the XYZ method."

It is important to recognize that quantifying the hypothesis often leads to the use of surrogate measures. That is, in order to identify a quantity with a factor or aspect you want to measure (e.g. quality), you must measure the factor indirectly using something associated with that factor (e.g. defects). Because the surrogate is an indirect measure, there is danger that a change in the surrogate is not the same as a change in the original factor. For example, defects (or lack thereof) may not accurately reflect the quality of the software: finding a large number of defects during testing may mean that testing was very thorough and the resulting product is nearly defect-free, or it may mean that development was sloppy and there are likely to be many more defects left in the product. Similarly, delivered lines of code may not accurately reflect the amount of effort required to complete the product, since that measure does not take into account issues like reuse or prototyping. Therefore, along with a quantifiable hypothesis, you should document the relationship between the measures and the factors they intend to reflect. In particular, you should strive for quantitative terms that are as direct and unambiguous at possible.

Control over variables

Once you have an explicit hypothesis, you must decide what variables can affect its truth, and how much control you have over each variable. The key discriminator between experiments and case studies is the degree of control over behavioral events and the variables they represent. A case study is preferable when you are examining events where relevant behaviors cannot be manipulated. For example, if you are investigating the effect of a design method on the quality of the resulting software, but you have no control over who is using which design method, then you probably want to do a case study to document the results. Experiments are done when you can manipulate behavior directly, precisely and systematically. Thus, in the same example, if you can control who uses the ABC method, who uses XYZ, and when and where they are used, then an experiment is possible. This type of manipulation can be done in a "toy" situation, where events are organized to simulate their appearance in the real world, or in a "field" situation, where events are monitored as they actually happen.

This difference between case studies and formal experiments can be stated more rigorously by considering the notion of a **state variable**. A state variable is a factor that can characterize your project and influence your evaluation results. Examples of state variables include the application area, the system type, or the developers' experience with the language, tool or method. In a formal experiment, you identify the variables and sample over them. This means that you select projects exhibiting a variety of characteristics typical for your organization and design your research so that more than one value will be taken for each characteristic. For example, your hypothesis may involve the effect of language on the quality of the resulting code. "Language" is a state variable, and an experiment would involve projects where many different languages would be used. You would design your experiment so that the projects involved represent as many languages as possible. In a case study, you sample from the state variable, rather than over it. This means that you select a value of the variable that is typical for your organization and its projects. With the language example, a case study might involve choosing a language that is usually used on most of your projects, rather than trying to choose a set of projects to cover as many languages as possible.

Thus, a state variable is used to distinguish the "control" situation from the "experimental" one in a formal experiment. (When you can't differentiate control from experiment, you must do a case study instead of a formal experiment.) You may consider your current situation to be the control, and your new situation to be the experimental one; a state variable tells you how the experiment differs from the control. For example, suppose you want

to determine whether a change in programming language can affect the productivity of your project. Then language is a state variable. If you currently use Fortran to write your programs and you want to investigate the effects of changing to Ada, then you can designate Fortran to be the control language and Ada to be the experimental one. The values of all other state variables should stay the same (e.g. application experience, programming environment, type of problem, and so on), so that you can be sure that any difference in productivity is attributable to the change in language.

Uses of formal experiments

There are many areas of software engineering that can be analyzed using formal experiments. One key motivator for using a formal experiment rather than a case study is that the results of an experiment are usually more generalizable than those of a case study. That is, if you use a case study to understand what is happening in a certain organization, the results apply only to that organization (and perhaps to organizations that are very similar). But because a formal experiment is carefully controlled and contrasts different values of the controlled variables, its results are generally applicable to a wider community and across many organizations. In the examples below, we will see how formal experiments can help answer a variety of questions.

Confirming theories and "conventional wisdom"

Many techniques and methods are used in software engineering because "conventional wisdom" suggests that they are the best approaches. Indeed, many corporate, national and international standards are based on conventional wisdom. For example, many organizations use standard limits on McCabe's cyclomatic complexity measure or rules of thumb about module size to "assure" the quality of their software. However, there is very little quantitative evidence to support claims of effectiveness or utility of these and many other standards, methods or tools. Case studies can be used to confirm the truth of these claims in a single organization, while formal experiments can investigate the situations in which the claims are true. Thus, formal experiments can be used to provide a context in which certain standards, methods and tools are recommended for use.

Exploring relationships

Software practitioners are interested in the relationships among various attributes of resources and software products. For example,

- How does the project team's experience with the application area affect the quality of the resulting code?
- How does the requirements quality affect the productivity of the designers?
- How does the design complexity affect the maintainability of the code?

Understanding these relationships is crucial to the success of any project. Each relationship can be expressed as a hypothesis, and a formal experiment can be designed to test the degree to which the relationship holds. Usually, many factors are kept constant or under control, and the value of one attribute is carefully varied to observe the effects of the changes. As part of the experiment's design, the attributes are clearly defined, consistently used and carefully measured.

For example, suppose you want to explore the relationship between programming language and productivity. Your hypothesis may state that certain types of programming languages (e.g. object-oriented languages) make programmers more productive than other types (e.g. procedural languages). A careful experiment would involve measuring not only the type of language and the resulting productivity but also controlling other variables that might affect the outcome. That is, a good experimental design would ensure that factors such as programmer experience, application type, or development environment were controlled so that they would not confuse the results. After analyzing the outcome, you would be able to conclude whether or not programming language affects programmer productivity.

Evaluating the accuracy of models

Models are often used to predict the outcome of an activity or to guide the use of a method or tool. For example, size models such as function points suggest how large the code may be, and cost models predict how much the development or maintenance effort is likely to cost. Capability maturity models guide the use of techniques such as configuration management or the introduction of testing tools and methods. Formal experiments can confirm or refute the accuracy and dependability of these models and their generality by comparing the predictions with the actual values in a carefully controlled environment.

Models present a particularly difficult problem when designing an experiment, because their predictions often affect the outcome of the experiment. That is, the predictions become goals, and the developers strive to meet the goal, intentionally or not. This effect is common when cost and schedule models are used, and project managers turn the predictions into targets for completion. For this reason, experiments evaluating models can be designed as "double-blind" experiments, where the participants don't know what the prediction is until after the experiment is done. On the other hand, some models, such as reliability models, do not influence the outcome, since reliability measured as mean time to failure cannot be evaluated until the software is ready for use in the field. Thus, the time between consecutive failures cannot be "managed" in the same way that project schedules and budgets are managed.

Validating measures

Many software measures have been proposed to capture the value of a particular attribute. For example, several measures claim to measure the complexity of code. A measure is said to be **valid** if it reflects the characteristics of an attribute under differing conditions. Thus, suppose code module X has complexity measure C. The code is augmented with new code, and the resulting module X' is now perceived to be much more difficult to understand. If the complexity measure is valid, then the complexity measure C' of X' should be larger than C. In general, an experiment can be conducted to test whether a given measure appropriately reflects changes in the attribute it is supposed to capture.

Validating measures is fraught with problems. Often, validation is performed by correlating one measure with another. But surrogate measures used in this correlation can mislead the evaluator, as described above. It is very important to validate using a second measure which is itself a direct and valid measure of the factor it reflects. Such measures are not always available or easy to measure. Moreover, the measures used must conform to human notions of the factor being measured. For example, if system A is perceived to be more reliable than system B, then the measure of reliability of A should be larger than that for system B; that is, the perception of "more" should be preserved in the mathematics of the measure. This preservation of relationship means that the measure must be objective and subjective at the same time: objective in that it doesn't vary with the measurer, but subjective in that it reflects the intuition of the measurer.

Factors to consider when choosing formal experiments

Several general guidelines can help you decide whether to perform a case study or a formal experiment. As stated before, the central factor is the level of control needed for a formal experiment. If you have a high level of control over the variables that can affect the truth of your hypothesis, then you can consider an experiment. If you do not have that control, an experiment is not possible; a case study is the preferred technique. But the level of control satisfies the technical concerns; you must also address the practical concerns of research. It may be possible but very difficult to control the variables, either because of the high cost of doing so, or the degree of risk involved. For example, safety-critical systems may entail a high degree of risk in experimentation, and a case study may be more feasible.

The other key aspect to consider is the degree to which you can replicate the basic situation you are investigating. For instance, suppose you want to investigate the effects of language on the resulting software. Can you develop the same project multiple times using a different language each time? If replication is not possible, then you cannot do a formal experiment. However, even when replication is possible, the cost of replication may be prohibitive.

Table 1 summarizes these concerns. The next article will assume that you have used this table and have decided that a formal experiment is the appropriate research technique for your problem.

Factor	Experiments	Case Studies
Level of Control	High	Low
Difficulty of Control	Low	High
Level of Replication	High	Low
Cost of Replication	High	Low

Table 2: Factors relating to choice of research technique

REFERENCES

Caulcutt, Roland (1991), *Statistics in Research and Development*, Chapman and Hall, London, England.

Chatfield, Christopher (1993), *Statistics for Technology, Chapman and Hall*, London, England.

Fenton, Norman, Shari Lawrence Pfleeger and Robert L. Glass (1994), "Science and Substance: A Challenge to Software Engineers," *IEEE Software*, July 1994.

Greene, Judith and Manuela d'Oliveira (1990), *Units 16 & 21 Methodology Handbook (part 2)*, Open University, Milton Keynes, England.

Lee, Wayne (1975), *Experimental Design and Analysis*, W.H. Freeman and Company, San Francisco, CA.

McCabe, T. J. (1976), "A Complexity Measure," *IEEE Transactions on Software Engineering*, SE-2(4), 308- 320.

Ostle, Bernard and Linda C. Malone (1988), *Statistics in Research*, Fourth Edition, Iowa State University Press, Ames, Iowa.

Pfleeger, Shari Lawrence (1993), *DESMET Experimental Design and Analysis Procedures (EXPDA)*, National Computing Centre, Manchester, England.

Pfleeger, Shari Lawrence (1994), "Experimental Design and Analysis in Software Engineering," *Annals of Software Engineering* 1(1), to appear.