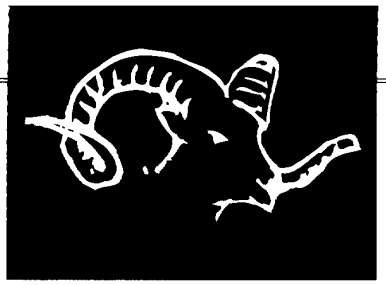

Case Studies for Method and Tool Evaluation

BARBARA KITCHENHAM and LESLEY PICKARD,
National Computing Centre
SHARI LAWRENCE PFLEEGER, *City University*

Case studies help industry evaluate the benefits of methods and tools and provide a cost-effective way to ensure that process changes provide the desired results. However, unlike formal experiments and surveys, case studies do not have a well-understood theoretical basis. This article provides guidelines for organizing and analyzing case studies so that they produce meaningful results.

You have read about a new technique or tool in *IEEE Software* or elsewhere, and you are considering its use on your project. If it worked for someone else, how do you know it will work for you? The last decade has seen explosive growth in the number of software-engineering methods and tools, each one offering to improve some characteristic of software, its development, or its maintenance. With an increasing awareness of the competitive advantage to be gained from continuing process improvement, we all seek methods and tools that will make us more productive and improve the quality of our software. But disaster can result from introducing inappropriate technology to a software-production department.[1] How do we ensure that our changes lead to positive improvement?

Norman Fenton, Shari Lawrence Pfleeger, and Robert Glass suggest that rigorous experimentation is needed to evaluate new technologies and their effects on our organizations, processes, and products.[2] Such scientific investigation is essential to understanding our processes and products, to increasing our customers' confidence in our products, and to making software engineering a science rather than an art.

Suppose you have decided to evaluate a technology. How do you proceed? Do you do a survey? An experiment? A case study? In this article, we discuss the conditions under which each type of investigation is appropriate. Then, because good case studies are as rare as they are powerful and

0740-7459/94/$04.00 © 1994 IEEE

informative, we focus on how to do a proper and effective case study. Although they cannot achieve the scientific rigor of formal experiments, case studies can provide sufficient information to help you judge if specific technologies will benefit your own organization or project. Even when you cannot do a case study of your own, the principles of good case-study analysis will help you determine if the case-study results you read about are applicable to your situation.

EMPIRICAL INVESTIGATION METHODS

In their landmark paper, Victor Basili, Richard Selby, and David Hutchens described a framework for quantitative software-engineering studies.[3] They defined software-engineering experiments in terms of a two-dimensional classification scheme:

- *Single-project studies*, which examine objects across a single team and a single project.
- *Multiproject studies*, which examine objects across a single team and a set of projects.
- *Replicated-project studies*, which examine objects across a set of teams and a single project.
- *Blocked subject-project studies*, which examine objects across a set of teams and a set of projects.

Many published software-engineering experiments and case studies refer to this classification scheme when explaining how their studies were carried out, and it is very useful for understanding how the investigation was done; the box on pp. 54 defines some other common experimental terms. However, we believe this classification must be extended to consider the formality of the experimental design.

- If the study focuses on a single project, we prefer to call it a *case study*, because it is not possible to have a formal experiment without replication.
- If the study involves many projects or a single type of project that is replicated several times, it can be either a case study or a *formal experiment*. A formal experiment requires appropriate levels of replication, and experimental subjects and objects that are chosen at random within the constraints of an experimental design.
- If the study looks at many teams and many projects, it may be a formal experiment or a *survey*, depending on whether the selection of teams and projects was planned or post hoc. Thus, any investigation can be considered a case study, formal experiment, or survey.

DIFFERENT METHODS YIELD DIFFERENT ENVIRONMENTAL DESIGNS, ANALYSIS TECHNIQUES, AND CONCLUSIONS.

However, the differences among these methods are also reflected in their scale. By their nature, since formal experiments must be carefully controlled, they are often small in scale: "research-in-the-small." Case studies usually look at what is happening on a typical project: "research-in-the-typical." And surveys try to capture what is happening broadly over large groups of projects: "research-in-the-large." The differences among research methods is important because the experimental design, analysis techniques, and conclusions they yield differ with each type.

Choosing a technique. Thus, the choice of investigative method depends in part on the size and nature of the organization or project that you want to investigate. It also depends on whether you are studying the technology in advance or after the fact. If you are trying to choose among several competing methods or tools, you may organize your study as a formal experiment or a case study. If you are establishing a pilot project to assess the effects of a change, you will probably choose to do a case study. But after the change has already been implemented across a large number of projects, a survey will help you to document the benefits of the change.

For all three investigative techniques, you must understand which variables you can control and how to measure the results. Formal experiments are sometimes difficult to conduct when the degree of control is limited. In order to impose full control, formal experiments are often small, which is a problem when you try to increase the scale from the laboratory to a real project. Thus, case studies are particularly important for industrial evaluation of software-engineering methods and tools because they can avoid scale-up problems. Whereas formal experiments sample over the variables that are being manipulated (so that you have a case representing each possible situation), case studies sample from the variables (representing the typical situation).

Case studies are easier to plan than experiments but are harder to interpret and difficult to generalize. A case study can show you the effects of a technology in a typical situation, but it cannot be generalized to every possible situation. For example, a case study may show you that the use of object-oriented languages increases the level of reuse on your banking-system project, but it cannot verify that object orientation always improves reuse.

On the other hand, a formal experiment is likely to be useful for investigating alternative methods of performing self-standing tasks. For example, you can perform an experiment to assess the effects of several program-design notations, such as flowcharts or pseudocode, on the resulting reliability or understandability. Here, formal experiments are appropriate because

- self-standing tasks can be isolated from the overall product-development process and investigated formally without being unrepresentative of the way they are actually performed;
- the results of self-standing tasks

EXPERIMENTAL TERMINOLOGY

The most important concept in a formal experiment is the *experimental hypothesis*, which defines what the experiment is intended to test. For example, a software experiment may investigate if design method A leads to better quality software than design method B. The corresponding hypothesis is: Design method A yields better quality software than design method B.

An experimental hypothesis usually asserts that different *treatments* have different effects on experimental subjects or objects. In the context of software experiments, a treatment is usually a method or tool. To draw any conclusions from an experiment, *there must be at least two treatments*, because hypothesis testing is comparative. Thus, the result of applying one treatment is compared with the result of applying another treatment, to determine if there is any difference.

In many experiments, one of the treatments, the *control*, is equivalent to the status quo. The use of a new method or tool is then compared with the control. However, software experiments have sometimes used the concept of a control incorrectly by assuming that the alternative to using method X is not using the method at all. However, if software staff using method X produce better products than software staff who do not use X, we cannot draw a valid conclusion about the effectiveness of X. We cannot tell if the difference in product quality is due to using X or simply due to the discipline of using a method. Moreover, we cannot tell if "not using X" means not using a method at all, or whether the status-quo group is actually using an informal or undocumented method of some kind. This distinction may not be important if your project is deciding whether or not to use X, but it is very important if another project or company wants to apply experimental results generated by other research groups. Thus, for experimental results to be generalized, there must be either two alternative treatments or a well-defined control.

We measure the effects of the change in method or tool by measuring the *response variables*, measures taken to test the hypothesis. A difference in treatments should be visible by examining differences in the values of the response variables. The specific response variables should be derived directly from the hypothesis. However, we often use *surrogate* measures instead of direct measures. For example, we may measure product reliability by counting the number of faults detected during testing, even though reliability reflects problems encountered by the user. Use of surrogate measures should be explicitly justified, because poor surrogate measures can invalidate the results of an experiment.

Experimental subjects and *experimental objects* are the people or things involved in an experiment. In software experiments, experimental subjects are individuals or groups (teams) who use a method or tool. Experimental objects may be the programs, algorithms, or problems to which the methods or tools are applied.

State variables are measures used to describe the experimental subject, objects, and conditions. They capture facts that are likely to affect the response variables.

can be judged immediately, rather than awaiting the results of a long development process, so that the experiment does not delay project completion; and

- the results of self-standing tasks can be assessed in isolation from other project processes, so that small benefits can be identified and distinguished from other variables.

Interpreting results. The reward for a well-designed experiment is results that are easier to generalize. Formal experiments are essential if you are looking for results that are broadly applicable across many types of projects and processes. Thus, formal experiments are important for the software-engineering research community, but they may not be necessary for a process-improvement program that applies only to your particular organization. For example, if you want to find out if using Ada will improve your project's software, but you do not need to know if using Ada will improve everyone's software, then a formal experiment may be overkill — you can rely on a case study.

However, even with formal experiments you must be careful — formal experiments do not generalize outside the controlled experimental conditions. For example, if you demonstrate that Ada improves real-time software using a formal experiment, you cannot guarantee Ada will improve software for data-processing systems.

A case study is usually preferable to a formal experiment if

- the process changes are very wide-ranging. This means that the effect of the change can be assessed only at a high level because the proces change represents many detailed changes throughout the development process. For example, if your project is changing from structured to object-oriented methods, the repercussions could affect all aspects of your processes and products — too much for you to control and measure.
- the effects of the change cannot be identified immediately. For example, if you want to know if a new design tool increases reliability, you may have to wait until after delivery to assess the effect on failures.

Case studies are a standard method of empirical study in various "soft" sciences such as sociology, medicine, and psychology, but there is little formal documentation available on how to perform a proper case study; Robert Yin's book is a notable exception.[4] However, Yin says that a case study should be used when "a how or why question is being asked about a contemporary set of events, over which the investigator has little or no control." For software engineering, we need case studies to evaluate not only how or why, but also "which is better." In this article we concentrate on the "which is better" type of case study.

Survey advantages. By combining the advantages of case studies (applicability to real-world projects) with those of experiments (replication that minimizes the problems of unusual results) surveys are particularly useful. Surveys

CHECKLIST FOR CASE-STUDY PLANNING

This checklist, along with the seven steps to design and administer case studies, will help you undertake a valid investigation.

Case study context

1. What are the objectives of your case study?
2. What is the baseline against which you will compare the results of the evaluation?
3. What are your external project constraints?

Setting the hypothesis

4. What is your evaluation hypothesis?
5. How do you define, in measurable terms, what you want to evaluate (that is, what are your response variables and how will you measure them)?

Planning

6. What are the experimental subjects and objects of the case study?
7. When in the development process or life cyle will the method be used?
8. When in the development or life cycle will the response variables be measured?

Validating the hypothesis.

9. Can you collect the data you need to calculate the selected measures?
10. Can you clearly identify the effects of the treatment you want to evaluate and isolate them from the other influences on the development?
11. Have you taken adequate procedures to ensure that the method or tool is being correctly used?
12. If you intend to integrate the method or tool into your development process, is the method or tool likely to have an effect other than the one you want to investigate?
13. Which state variables or project characteristics are most important to your case study?
14. Do you need to generalize the result to other projects? If so, is your pro posed case study project typical of those projects?
15. Do you need a high level of confidence in your evaluation result? If so, do you need to do a multiproject study?

Analyzing the results

16. How are you going to analyze the case study results?
17. Is the type of case study going to provide the level of confidence you require?

can be used to ensure that process changes are successful throughout an organization, because they collate experience from several different projects. However, data collection can take a great deal of time, and the results may not be available until after many projects are completed. In medical research, millions of patients may undergo a particular treatment or use a particular drug simultaneously, so it is relatively easy to build up a large amount of data quickly. There are fewer such opportunities in software engineering because it is more difficult to find comparable experimental objects, because software measures are not used consistently, and because there is no framework to review and collate experimental results.

The most common form of survey is based on distributing questionnaires that elicit opinions about the benefits of technology.[5] In a different type of study, David Card, Frank McGarry, and Gerry Page[6] analyzed project data from the University of Maryland's Software Engineering Laboratory, looking at the effects of technology on NASA's productivity and quality. Card's group analyzed existing data, rather than soliciting new information, a technique used frequently in other disciplines.

No one type of empirical study is better than any other; each is appropriate in particular situations. But experiments and surveys are traditional "hard-science" techniques that are supported by a rich literature describing how to design and administer them. Thus, for the rest of this article, we concentrate on case studies in order to provide more rigor to a neglected discipline of investigation.

CASE STUDY GUIDELINES

There are seven steps to follow in designing and administering case studies:

1. Define the hypothesis.
2. Select the pilot projects.
3. Identify the method of comparison.
4. Minimize the effect of confounding factors.
5. Plan the case study.
6. Monitor the case study against the plan.
7. Analyze and report the results.

These steps, which help ensure that you can draw valid conclusions from your investigation, are related to the four criteria for research-design quality:[4]

- *Construct validity.* Establish correct operational measures for the concepts being studied.
- *Internal validity.* Establish a causal relationship and distinguish spurious relationships.
- *External validity.* Establish the domain to which a study's findings can be generalized.
- *Experimental reliability.* Demonstrate that the study can be repeated with the same results.

For simplicity, we explain the steps by assuming that you are testing a new method on an actual software-development project. The box on this page provides a checklist to help you plan a case study.

Define the hypothesis. You begin by defining the effect you expect the method to have. This definition must be detailed enough to make clear what measurements are needed to demonstrate the effect. For example, if you expect the new method to improve productivity, you must state if effort

TABLE 1
COMPARISON OF PRODUCTIVITY MEASURES

Variable	Method A	Method B
Productivity (function points/hour)	0.054	0.237
Size (function points)	118	168
Team experience (years)	1	1
Project management experience (years)	1	1
Duration (months)	10	9
Function point	25	27

and duration will be affected and how. Without this information, you cannot identify, measure, and collect the data you need to draw valid conclusions.

It is also important to define what is *not* expected to happen. Formally, we can never prove hypotheses, we can only disprove them, so we state a null hypothesis to say that there is no difference between treatments. However, research is proposed and funded based on studying the alternative hypothesis: there is a significant difference between treatments. The formal case-study data analysis and evaluation addresses the null hypothesis, but you should be ready to present your findings to managers and staff in terms of the alternative.

The more clearly you define your hypotheses, the more likely you are to collect the right measures, test them properly, and achieve construct validity. You must specify carefully what really interests you. For example, process-improvement programs often define quality as the reduction of rework and waste, presenting quality in terms of defect rates from the perspective of a software developer. However, this definition differs from the user's point of view, in which operational reliability, efficiency, and usability reflect how the user sees the software.

Select the pilot projects. The pilot projects you choose must be representative of the type of projects your organization or company usually undertakes. Ideally, you can describe projects in terms of significant characteristics, such as application domain, programming language, design method, and degree of reuse, and then use this state-variable information to select projects that are most typical. Your selection should consider not only project type but also the frequency with which each type is developed. In practice, it may be difficult to control the choice of case-study projects. However, the extent to which the case-study project is typical of the organization is central to the issue of external validity. If your case study is atypical of the projects you usually undertake, you will not get very useful results.

Identify the method of comparison. Your case study is by nature comparative, contrasting the results of using one method with the results of using another. To avoid bias and ensure internal validity, you must identify a valid basis for assessing the results of the case study. There are three ways to organize your study to facilitate this comparison:

- *Select a sister project with which to compare.* Here, the case study involves two projects, one that uses the new method and another that uses the current method. Each project should be typical of your organization, and both should have similar characteristics according to the state variables you have chosen. The box on this page describes variants of this design.
- *Compare the results of using the new method against a company baseline.* In this case, your company gathers data from projects as a standard practice and makes data available on such things as average productivity or defect rate. You can compare the response-variable values from your case study, which involves a single project using the new method, to the corresponding variables from previous projects or a subset of similar projects.
- *If the method applies to individual components, apply it at random to some product components and not to others.* Here, the case study resembles a formal experiment, because you can use replicated values and standard statistical methods to analyze the response variables. But because the projects are not drawn at random from the population of all projects, this is not a true formal experiment. This kind of study is useful for methods that may be applied to different degrees. For example, if you want to know what level of structural testing is most cost-effective, you can measure the level of structural testing achieved for different modules and compare testing effort and subsequent defect rates (or defect-detection efficiency, if you have seeded errors).

Minimize the effect of confounding factors. When the effect of one factor cannot be properly distinguished from the effect of another factor, the two factors are *confounded.* For example, if expert software engineers tested tool A and novice software engineers tested tool B, we cannot tell if the higher quality software produced by the experts was the result of their experience or of using tool A. Confounding factors can affect the internal validity of the study.

Software case studies often have confounding factors. The most significant are likely to be:

- *Learning how to use a method or tool as you try to assess its benefits.* In this case, the effects of learning to use the method or tool might interfere with the benefits of using it. For example, a decrease in productivity caused by the learning curve might hide productivity improvements. To avoid this effect, you must separate activities aimed at learning how to use a new technology from those aimed at evaluating it.
- *Using staff who are either very enthusiastic or very skeptical about the method or tool.* Staff morale can have a large effect on productivity and quality. Differences in the response variable

may be due to staff enthusiasm, or to differences in enthusiasm from one developer to another. To minimize this effect, you must staff a case-study project using your normal staff-allocation method.

♦ *Comparing different application types.* For example, the productivity of real-time system developers is usually lower than for data-processing systems, so case studies should not compare across application domains. Appropriate selection of case-study projects will avoid this problem.

Sometimes it is possible to control a confounding effect rather than eliminate it. This usually involves designing a multiproject case study in which the different projects experience different conditions. For example, to investigate if the benefits of some method or tool are influenced by application type, we can identify a pair of case-study projects for each application type: one to use the new method and one to use the current method.

You can sometimes control confounding by measuring the confounding factor and adjusting the results accordingly. For example, to study how different levels of reuse affect quality and productivity, you may select a case-study project in which components (specifications, designs, or code) are being reused, measure the amount of each component that is reused, the development productivity for each component, and the defect rate. If you suspect that, in addition to reuse, component complexity affects productivity and defect rates, you can record component complexity and use partial correlation to assess the relationship between percentage reuse, productivity, and defect rates, adjusted for complexity.

Plan the case study. Basili, Selby, and Hutchens emphasize that organizations undertaking experiments should prepare an evaluation plan.[3] This plan identifies all the issues to be addressed so that the evaluation runs smoothly, including the training requirements, the necessary measures, the data-collection procedures, and the people responsible for data collection and analysis. Attention to detail contributes to experimental reliability.

The evaluation should also have a budget, schedule, and staffing plan separate from those of the actual project. A separate plan and budget is needed to ensure that the budget for the evaluation does not become a contingency fund for the project itself! Clear lines of authority are needed for resolving the inevitable conflicts of interest that occur when a development project is used to host an evaluation exercise.

Monitor the case study against the plan. The case study's progress and results should be compared with the plan. In particular, ensure that the methods or tools under investigation are used correctly, and that any factors that would bias the results are recorded (such as change of staff, or a change in the priority of the case-study projects). It is essential that you audit conformance to the experimental plan and record any changes. At the end of the study, you should write an evaluation report including recommendations for changes in procedures.

Analyze and report the results. The analysis procedures you follow depend on the number of data items you must analyze (that is, the number of response-variable values that are available). If your case study compared treatments assigned to components at random, you can use standard statistical methods, such as analysis-of-variance and contingency tables. Data distribution is important in choosing an analysis technique. If you cannot guarantee that the data is distributed normally (according to a bell-shaped Gaussian curve), then you must use nonparametric tests such as the Kruskall-Wallis method, which bases the analysis on rank rather than on raw data. (See the box on pp. 59 for references to useful analysis texts.) If you have only one value from each method or tool being evaluated, no analysis techniques are available; you can only present the results as we describe next.

MAKE SURE TO SEPARATE THE EVALUATION BUDGET SO THAT IT DOES NOT GET SPENT ON THE PROJECT ITSELF.

ANALYSIS METHODS FOR CASE STUDIES

Once you have designed your case study and collected the data, you must analyze it to determine what has happened and if the results are significant. Suppose your case study involves a sister experiment with one response value per project. For example, for each project participating in the study, you measure productivity in function points per staff hour using method A (the current method) and method B (the new method). Table 1, using real data,[7] shows what you might find.

The data in Table 1 indicate that the projects are quite similar with respect to the state variables: size, team experience, project-manager experience, duration, and function-point adjustment factor. Thus, the results suggest that using method B would improve productivity. However, to draw that conclusion, you must be sure that both projects are typical of those undertaken by the organization. You must also understand the factors that are important for software development in the organization that might affect the successful use of methods A and B.

In addition to looking at the quantitative results, you can investigate how typical these projects are by reviewing the distribution of state-variable values over all the projects undertaken by the organization. Simple frequency plots are useful for depicting the distribution of discrete state-variable values for an organization. For example, Figure 1

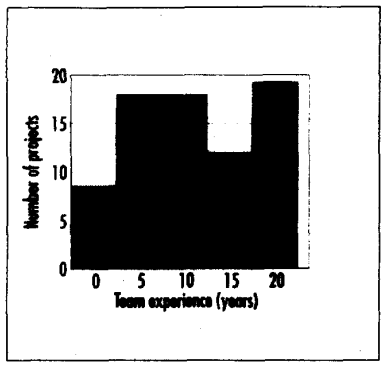

Figure 1. Frequency plot showing the distribution of discrete values, in this case the team experience for the set of projects from which the case-study projects were selected. The plot shows that it is not unusual for a team to have only one year of experience.

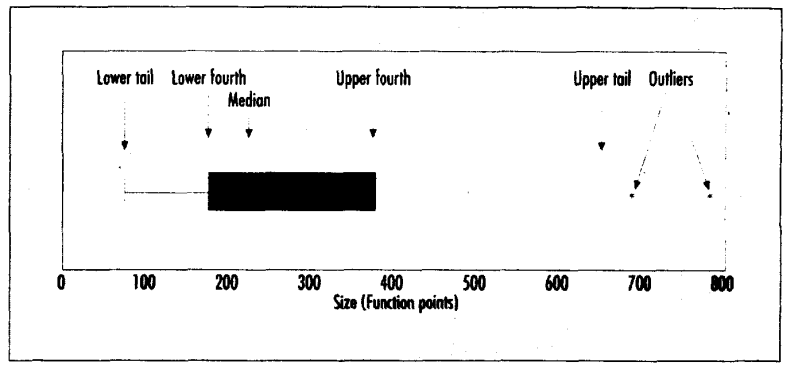

Figure 2. Boxplot showing a distribution of data values over a wide range, in this case the product size. Box plots are constructed from five statistics: the median, the upper fourth (or upper quartile), the lower fourth, the upper tail, and the lower tail. The upper and lower fourths are the 75- and 25-percentile points. The upper tail is constructed by multiplying the box length by 1.5, adding the value to the upper fourth, and truncating to the nearest actual value. The lower tail is constructed by a similar process. Values that are larger than the upper tail or smaller than the lower tail are called outliers.

shows the team experience for the set of projects from which the case-study projects were selected. As you can see, it is not unusual for a team to have only one year of experience.

When you have state variables that cover a wide range of values (such as counts or ratios), a boxplot can help you evaluate the distribution of data values, particularly when data values are skewed. Figure 2 shows a boxplot of product-size data.

Boxplots give a simple visual display of the distribution of a data set and help you see how representative a single point is. If the data set were distributed as a classic Gaussian (normal) distribution, the mean would be in the center of the box, the tail lengths would be approximately equal, and the distance from the median to the upper (or lower) tail would be approximately three standard deviations.

It is clear from Figure 2 that the product-size data set is skewed, and that the two pilot projects were relatively small ones (in the lower 25-percent range). Thus, there is some doubt about whether the case-study projects were truly representative of the organization's projects. Any productivity improvements resulting from method B might occur only on smaller projects.

Boxplots are also useful for constructing a company baseline. Figure 3 shows productivity distributions data from 46 projects that used method A. There are no outliers in the data set, so the baseline for average projects is some productivity value between the upper (0.044) and lower (0.076) fourths; the upper and lower tails give the upper and lower bounds for the organization. If you place the productivity of a case-study project using method B on the figure as an asterisk, it becomes clear that the case study had unusually high productivity compared to the company baseline. The baseline can be refined further by reconstructing it using projects that have similar state-variable characteristics to the case study.

SAMPLE CASE STUDIES

To see how software-engineering case studies can be improved, we turn now to three studies[8,9] aimed at assessing the benefits of Fagan inspections.[10] The studies represent not only the different types we have discussed but also the many problems that can result from improper case-study planning and administration.

The first study compared differently treated components, the second used a company baseline, and the third involved sister projects. Each study was run for ICL's VME development group. VME is a large general-purpose operating system (approximately two million lines of code) that has been under continual evolution since its first release in the early 1970s. When the case studies were performed, fairly small teams (two to eight people) worked on specific functional subsystems. Staff turnover was low, and people worked on the same team for many years. The operating system was written in a variant of Algol 68 and supported by a special-purpose database environment that maintained records of literals, data types, and module interfaces, all supported with configuration control.

Case study 1. The first case study used a single project to investigate if Fagan inspections would increase software quality without resulting in a decrease in productivity. Formally, the hypothesis stated that the use of Fagan inspections has no effect on quality or productivity. Forty-three of the project's 73 programs were given detailed design inspections; the rest were not. Thus, it was possible to compare the postdesign fault profile of inspected programs with the postdesign fault profile of uninspected programs. The response variables were fault counts and staff effort, with faults related to where they were discovered: the major stage in the development process or postrelease (for a six-month period). Total project effort and inspection effort were both recorded.

♦ *Design-inspection results.* The case-study procedure ensured that several productivity and quality measurements were made. The design

inspections detected 50 percent of all faults found for this development (up to nine months postrelease). The inspections accounted for 6 percent of the total development costs. The fault-detection rate was approximately 1.2 hours per fault. However, a major problem was that there was no basis for identifying if these results were good or bad because there was nothing to compare them with!

♦ *Postinspection fault rates.* Table 2 shows the main response variable — the number of faults detected subsequent to code production, as measured for each group of modules for different defect types. However, the modules were not allocated to design inspection randomly. In fact, the project staff selected for inspection only those modules they thought were "difficult"; "easy" modules were not given design inspections.

The lower overall-error rate indicates that the uninspected programs were simpler. But the inspected programs revealed their faults earlier in the development process than the uninspected programs. By the time they reached system test, inspected programs appeared to have higher quality than uninspected programs, a situation confirmed by the postrelease fault rates.

♦ *Problems with case study 1.* This pilot project was chosen because the team wanted to participate. There was no formal selection to ensure that the pilot was representative of typical ICL projects. Furthermore, this predisposition to be helpful probably biased the results in favor of the inspection technique. But the major problem with this study was the nonrandom selection of modules that were subjected to detailed design inspections. The development staff members themselves decided which modules would be given detailed inspections, and they selected only those that were difficult. This was a sensible approach for the project, but it had a disastrous effect on the evaluation's validity. Had the allocation been random, an analysis-of-variance on the postdesign quality of each module

USEFUL ANALYSIS TEXTS

Experiment design

♦ W.G. Cochran and G.M. Cox, *Experimental Designs*, 2nd ed., John Wiley & Sons, New York, 1957: Standard statistical text.

♦ D.T. Campbell and J. Stanley, *Experimental and Quasi-Experimental Designs for Research*, Rand McNally, Chicago, 1966: Practical industrial experimental design.

♦ S.L. Pfleeger, "Experimental Design and Analysis in Software Engineering," *Annals of Software Engineering*, Vol. 1 No. 1, pp. 1-20; Design issues for software experiments.

Survey analysis

♦ W.G. Cochran, *Sampling Techniques*, 2nd ed., John Wiley & Sons, New York, 1963: Discussion of methodological issues of surveys, in particular how to sample a finite population so that survey results can be generalized.

♦ D. Coggon, G. Rose, and D.J.P. Barker, *Epidemiology for the Uninitiated*, 3rd. ed., British Medical Journal, London, 1993: Survey techniques used in medical research.

Data analysis

♦ P.G. Hoel, *Introduction to Mathematical Statistics*, 3rd. ed., John Wiley & Sons, New York, 1962.

♦ S. Siegel and N.J. Castellan, Jr., *Nonparametric Statistics for the Behavioral Sciences*, 2nd ed., McGraw-Hill, New York, 1988: Classic text on nonparametric analysis techniques.

♦ D.C. Hoaglin, F. Mosteller, and J.W. Tukey, *Understanding Exploratory Data Analysis*, John Wiley & Sons, New York, 1983: Descriptions of boxplots and other exploratory data-analysis methods.

TABLE 2
COMPARISON OF FAULTS DETECTED

Test Method	Faults per 100 lines of code	
	Code inspected (13,334 lines of code)	Code not inspected (8,852 lines of code)
Code reading	0.97	0.45
Unit test	0.82	0.68
System test	0.20	0.36
Customer	0.012	0.043
Overall	2.0	1.54

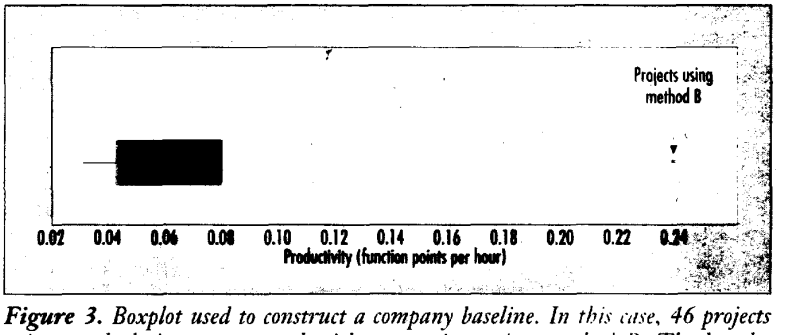

Figure 3. *Boxplot used to construct a company baseline. In this case, 46 projects using method A are compared with one project using method B. The boxplot shows that this single case study had unusually high productivity compared to the company baseline.*

(measured in defects per hundred lines of code) would have revealed if the inspections made a significant difference. But because difficult modules exhibit more defects than simple modules even after design reviews, this analysis was not valid. Thus the only useful result is the overall defect rate for each major postdesign activity. This problem could have been avoided if the case study had been planned and controlled as an activity in its own right, rather than as an adjunct to the development effort.

Other problems resulted from this lack of planning. For example, several other response variables were collected but could not be properly interpreted because there was no basis of comparison. Thus, it was impossible to tell if inspections decreased productivity.

Case study 2. The second case study looked at whether Fagan inspections would increase software quality through a cost-effective detection of defects. A single project was used and compared with a baseline made up of all other concurrent projects. Thus, it was possible to compare the postdesign fault profile of the pilot project with the postdesign fault profile of other projects.

The response variables were fault counts and staff effort. Here, faults were again related to each major stage in the development process. In addition, faults were classified as design or coding faults. Total project effort, effort for conventional testing, and inspection effort were recorded. As before, the pilot project was self-selected because the development team was interested in the inspection technique.

♦ *Design-inspection results.* This second case study, involving the production of a new subsystem of approximately 39,000 lines of code, gave results broadly similar to the first. However, because this case study collected data on testing and inspection effort, it was possible to assess the relative costs of fault detection and correction. The inspections detected 41 percent of in-house faults at a cost of 9 percent of the project-development effort. The cost-per-fault was approximately 1.6 hours, with an average cost-per-fault detected postdesign of 8.5 hours. This result suggests that inspections are a very cost-effective fault-detection method.

♦ *Postdesign fault profile.* When the fault profile of the pilot project postdesign was compared with the fault profile found for other projects during the same time, it appeared that postdesign faults were again being found earlier in the development process, as Table 3 shows. However, the baseline does not include any assessment of variability.

♦ *Fault types.* Table 4 shows the types of fault found postdesign, indicating that inspections reduced the number of design faults but not the number of interface faults (which can be regarded as a kind of design fault). These findings were reviewed with the development group, which pointed out that the faults were found in code that interfaced to a subsystem developed by team members who refused to attend inspections. Thus, they reasoned, the results actually supported the need for inspections. This result emphasizes the importance of monitoring the pilot project for unexpected effects. If the interface problems had not been traced back to the nonparticipating group, the results might have been misinterpreted.

♦ *Problems with case study 2.* As in the first case study, the pilot project was not chosen using any formal selection process. However, the more significant problems occurred at the analysis stage. The construction of the baseline would have been greatly improved by using a boxplot to indicate the extent of natural variability. In some circumstances, it might have been better to use a direct measure of faults per hundred lines of code during system test as the response measure rather than percentages. However, in this particular environment, there was a considerable variation in basic fault rates from different types of project. The pilot project was a new utility project and was expected to have lower fault rates than some of the more complex enhancement projects; a baseline based on faults per hundred lines of code would have to have been derived from a very small selection of similar

REPLICATED PRODUCT DESIGN

It is sometimes possible to develop a product a second time using a different development method. This is called a *replicated product design*. To use it:

1. Replicate an existing product using the new method or tool.
2. Measure the response variables on both versions of the product.
3. Compare the two sets of response variables.

The advantage of this design is that some of the differences between the sister projects is eliminated because they both produce the same project. However, usually only one of the products is produced under normal commercial conditions.

This method is often used when a research group wants to demonstrate the superiority of a new method compared with current development methods. However, if the research group also undertakes the replication project, the results will be biased because the research group will usually have more experience with the new method than would the development staff and are more motivated to see it succeed.

These problems can be overcome if the research group sponsors the development group to undertake both projects to commercial standards, and the product that performs best in final system test (or acceptance test) is the one released to customers.

projects. A baseline based on internal distribution of faults was valid for all projects because they all use the same development process.

Finally, it is important to note that the case study used a surrogate measure of quality. Actual quality depends on the defects found during use, but the analysis was based on the defects found during in-house testing. Thus the conclusions may be misleading.

Case study 3. The third investigation was not planned as a case study. Rather, it occurred naturally as two parts of the same project were developed in different ways. At first, the primary part of the project was planned; however, later a large additional functional development was required. The same team produced both subprojects, and testing on each was done by the same staff member. Although these subprojects were not selected to be part of a case study, they were typical of the commercial projects undertaken by the group.

Because the project manager wanted to get the second part of the project completed as quickly as possible, he did not permit any detailed design inspections. His unstated hypothesis was, therefore, that detailed design inspections cause delays to product development and do not have a major influence on quality. In effect, what resulted was a case study based on sister subprojects, with response variables defined as time to complete production, effort, and fault rates.

♦ *Results.* The results, shown in Table 5, indicate that trading quality for productivity simply did not work, and the hypothesis can be firmly rejected. The subproject without inspections took far longer to produce than the much larger "high-quality" subproject. Additional time and effort were needed to test the code that had not been subject to design inspections. This diminished productivity is clear, even though the state variables show that the subproject without inspections was much smaller than the other subproject in terms of absolute size and number of modules. In addition, code from the subproject done without design inspections was of poor quality (in terms of fault rates) and was more expensive than the inspected code.

TABLE 3
COMPARISON OF FAULT PROFILES

Test Method	Percentage Faults	
	Pilot Project	All Projects
Code reading	57.5	37.6
Unit test	38.4	51.2
System test	4.1	11.2

TABLE 4
COMPARISON OF FAULT TYPE

Fault Type	Percentage Faults	
	Pilot Project	All Projects
Interface	5.8	2.1
Design	8.1	13.3
Code	81.0	70.3
Other	5.1	14.2

The case study is convincing because the difference in results is so dramatic. In addition, many of the typical problems with case-study control and variation were absent because the same personnel were involved, the same development environment was used, and the applications were related.

♦ *Problems with case study 3.* Clearly, the study was not planned in advance. Nevertheless, it conformed quite well to case-study requirements and resulted in sufficient information to reject the hypothesis that inspections increase time to market and do not affect quality. However, this study is not completely without problems. The quality measure was based on prerelease rather than postrelease defects, again reflecting a developer's rather than a user's view of software quality. A more subtle problem involves whether or not the two parts of the project are really comparable. According to the subjective opinion of the staff involved, the two subprojects were similar in complexity; however, there were no objective measures to confirm this claim.

What is the next step?

Software-engineering experimentation is a necessary adjunct to process improvement, and objective, meaningful case studies can help us understand our processes and control the improvements. Many case studies are performed, but few are done well. The case-study process is itself in need of improvement.

Good case studies involve:

♦ Specifying the hypothesis under test.

♦ Using state variables for project selection and data analysis.

♦ Establishing a basis for comparisons.

♦ Planning case studies properly.

♦ Using appropriate presentation and analysis techniques to assess the results.

We must stop and assess each tool and technology before we jump on a promotional bandwagon. Even when formal experiments are not available or possible, we can perform case studies to determine if the tool or technology is helpful on our typical projects. That is, we need not wait until a method is proven effective in every environment; we can run careful tests to see if the method is useful in our particular environments.

But such investigation requires the

investment of time and effort, not only in planning and carrying out the case studies, but also in analyzing and reporting the results. The findings of academic experiments are often widely publicized, as universities encourage their staff to publish and disseminate results. But the results of industrial case studies, less often made available to the public, are no less relevant to practitioners who are seeking new or improved ways of developing and maintaining software.

The results of case studies are context-dependent, but we can be more confident that a method is generally beneficial if encouraging results are reported by a number of different organizations under a number of different conditions. We can also better understand the limits of methods and tools if we get conflicting reports from different case studies.

We encourage you to assess the work of others, not only in terms of the issues raised here, but also in terms of whether it is applicable to your projects. And we encourage you to publish your case-study results, to the benefit of the general software-engineering community. ◆

ACKNOWLEDGMENTS

This article is based on research undertaken as part of the Desmet project, a collaborative project funded by the UK Department of Trade and Industry and the Science and Engineering Research Council. The aim of the project was to develop and validate a methodology for evaluating software-engineering methods and tools. We are also indebted to the referees, who provided valuable suggestions that we have incorporated into the final version.

REFERENCES

1. D.R. Lindstrom, "Five Ways to Destroy a Development Project," *IEEE Software*, Sept. 1993, pp. 55-58.
2. N. Fenton, S.L. Pfleeger, and R.L. Glass, "Science and Substance: A Challenge to Software Engineers," *IEEE Software*, July 1994, pp. 86-95.
3. V.R. Basili, R.W. Selby, and D.H. Hutchens, "Experimentation in Software Engineering," *IEEE Transactions Software Eng.*, July 1986, pp. 758-773.
4. R.K. Yin, *Case Study Research Design and Methods*, Sage Publications, Beverley Hills, Calif., 1984.
5. C.R. Necco, R.N.W. Tsa, and K.W. Hoogeston, "Current Usage of CASE Software," *J. Systems Management*, May 1989.
6. D.N. Card, F.M. McGarry, and G.T. Page, "Evaluating Software-Engineering Technologies," *IEEE Transactions Software Eng.*, July 1987, pp. 845-851.
7. J.-M. Desharnais, *Analyse Statistique de la Productivitie des Project de Developpment en Informatique a Partir de la Technique des Point des Fonction*, master's thesis, University of Quebec, Montreal, 1989; in French.
8. B.A. Kitchenham, A.P. Kitchenham, and J.P. Fellows, "The Effects of Inspections on Software Quality and Productivity," *ICL Technical J.*, May 1986, pp. 112-122.
9. B.A. Kitchenham, "Management Metrics," *Software Reliability Achievement and Assessment*, B. Littlewood, ed., Blackwell Scientific Publications, Barking, UK, 1987, pp. 113-124.
10. M.E. Fagan, "Design and Code Inspections to Reduce Errors in Program Development," *IBM Systems J.*, Mar. 1976, pp. 219-248.

Barbara Kitchenham is a software-engineering consultant at Britain's National Computing Centre. Her interests are software metrics and their application to project management, quality control, and evaluation of software technologies. She was a programmer for ICL's Operating System Division before becoming involved with a number of UK and European research projects on software quality, software-cost estimation, and evaluation methodologies for software technology. She has written more than 30 papers on software metrics.

Kitchenham received a PhD from the University of Leeds. She is an associate fellow of the Insitute of Mathematics and Its Applications and a fellow of the Royal Statistical Society.

Lesley Pickard is an independent consultant. In the last 10 years she has been involved in researching the use of statistical techniques and software metrics for the monitoring and control of software development. Much of her work has been part of European collaborative projects. She has written several technical papers.

Pickard received a BSc in applied mathematics from Abertay University, Scotland, and a PhD in computer science from the City University, London. She is a fellow of the Royal Statistical Society.

Shari Lawrence Pfleeger is president of Systems/Software, a consultancy that specializes in software engineering and technology transfer. Her clients have included major corporations, government agencies, and universities. Pfleeger has been a principal scientist at both the Contel Technology Center and Mitre. She is currently a visiting professorial research fellow at the Centre for Software Reliability, investigating how software-engineering techniques affect software quality. She has written two software-engineering texts and several dozen articles.

Pfleeger received a PhD in information technology from George Mason University. She is an adviser to *IEEE Spectrum*.

Address questions about this article to Kitchenham at National Computing Centre, Oxford House, Oxford Rd., Manchester M1 7ED, UK; barbara.kitchenham@ncc.co.uk.